RATIONALE

The purpose of the Error Model Annex in this document is to enable modeling of different types of faults, fault behavior of individual system components, modeling of fault propagation affecting related components in terms of peer to peer interactions and deployment relationships between software components and their execution platform, modeling of aggregation of fault behavior and propagation in terms of the component hierarchy, as well as specification of fault tolerance strategies expected in the actual system architecture. The objective of the Error Model Annex is to support qualitative and quantitative assessments of system dependability, i.e., reliability, availability, integrity (safety, security), and survivability, as well as compliance of the system to the specified fault tolerance strategies from an annotated architecture model of the embedded software, computer platform, and physical system.

This Architecture Analysis & Design Language (AADL) standard document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

Change log:

Changed separator of error type products to *. In E.3(2) top of p. 13: changed "operational behavior specification" to "modes". (brl 2012-01-23)

Simplified type system by supporting type sets of type hierarchies that include type products instead of supporting power sets of type hierarchies. (thanks to Brian).

Revised the formal specification of error types (thanks to Brian).

Added ability to declare connections as error sources to model interface mismatches.

Changed error detection declarations to specify an error code being reported within the system as event data.

Cleaned up Hazard property definition.

Added support for declaring Hazard property association for contained subcomponents, i.e., describe Hazards in their use context.

Added properties to characterize errors as design/operational errors and their persistence as permanent, transient, singleton.
Moved elaborated examples of error model use into a separate document to reduce the size of the Annex document. Currently I have left in the large example we had in the original annex.
# TABLE OF CONTENTS

1 REFERENCES .......................................................................................................................... 7

1.1 NORMATIVE REFERENCES ......................................................................................... 7

1.2 INFORMATIVE REFERENCES ..................................................................................... 7

ANNEX DOCUMENT E  ERROR MODEL ............................................................................... 9

ANNEX E.1 SCOPE ................................................................................................................. 9

ANNEX E.2 CONCEPTS AND TERMINOLOGY ................................................................. 11

ANNEX E.3 ERROR MODEL LIBRARIES .......................................................................... 13

ANNEX E.4 ERROR MODEL SUBLAUSES ....................................................................... 14

ANNEX E.5 ERROR TYPES, TYPE SETS, AND TYPE TOKENS ........................................ 15

ANNEX E.6 A COMMON SET OF ERROR TYPES ............................................................. 20

E.6.1 SERVICE RELATED ERRORS ................................................................................... 24

E.6.2 VALUE RELATED ERRORS ....................................................................................... 25

E.6.3 TIMING RELATED ERRORS ...................................................................................... 26

E.6.4 RATE RELATED ERRORS ........................................................................................ 26

E.6.5 SEQUENCE RELATED ERRORS .............................................................................. 27

E.6.6 REPLICATION RELATED ERRORS ......................................................................... 27

E.6.7 CONCURRENCY-RELATED ERRORS ........................................................................ 28

ANNEX E.7 ERROR PROPAGATION .................................................................................... 29

E.7.1 ERROR PROPAGATION AND CONTAINED ERROR DECLARATIONS ..................... 31

E.7.2 ERROR FLOW DECLARATIONS ................................................................................. 34

E.7.3 ERROR PROPAGATION PATHS ................................................................................ 36

E.7.4 ERROR PROPAGATION PROPERTIES AND ANALYSES .................................. 39

ANNEX E.8 ERROR BEHAVIOR STATE MACHINES ......................................................... 43

E.8.1 ERROR, RECOVER, AND REPAIR EVENTS ......................................................... 48

E.8.2 ERROR BEHAVIOR STATES AND TRANSITIONS ............................................... 49

E.8.3 TYPED ERROR BEHAVIOR STATE MACHINES .................................................. 51

ANNEX E.9 COMPONENT ERROR BEHAVIOR SPECIFICATION ................................. 53

E.9.1 OUTGOING ERROR PROPAGATION CONDITIONS ............................................. 56

E.9.2 ERROR DETECTIONS ............................................................................................... 57

E.9.3 OPERATIONAL MODES AND FAILURE MODES .................................................. 58

ANNEX E.10 COMPOSITE ERROR BEHAVIOR ............................................................... 59

ANNEX E.11 CONNECTION ERROR BEHAVIOR .......................................................... 61

ANNEX E.12 ERROR TYPE MAPPINGS AND TRANSFORMATIONS .............................. 63

ANNEX E.13 ERROR MODELS AND FAULT MANAGEMENT ........................................... 66

ANNEX E.14 A RECONFIGURABLE TRIPLE REDUNDANT SYSTEM (EXAMPLE FROM ERROR MODEL ANNEX V1) .......... 67
Foreword

(1) The AADL standard was prepared by the SAE Avionics Systems Division (ASD) Embedded Computing Systems Committee (AS-2) Architecture Description Language (AS-2C) subcommittee.

(2) This AADL standard document defines the third volume of annexes to the SAE AADL standard AS-5506B published in 2012, with volume 1 published in 2006 [SAE AS-5506/1] and volume 2 published in 2011 [SAE AS-5506/2].

(3) This AADL standard includes an Error Model Annex that extends the AADL core language with a state machine-based sublanguage annex notation for specifying different types of faults, fault behavior of individual system components, fault propagation affecting related components in terms of peer to peer interactions and deployment relationship between software components and their execution platform, aggregation of fault behavior and propagation in terms of the component hierarchy, as well as fault tolerance strategies expected in the actual system architecture.

(4) The objective of the Error Model Annex is to support qualitative and quantitative assessments of system dependability, i.e., reliability, availability, integrity (safety, security), and survivability, as well as compliance of the system to the specified fault tolerance strategies from an annotated architecture model of the embedded software, computer platform, and physical system.
Introduction

(5) The SAE Architecture Analysis & Design Language (referred to in this document as AADL) is a textual and graphical language used to design and analyze the software and hardware architecture of performance-critical real-time systems. These are systems whose operation strongly depends on meeting non-functional system requirements such as reliability, availability, timing, responsiveness, throughput, safety, and security. AADL is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. It can be used to describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). AADL can also be used to describe how components interact, such as how data inputs and outputs are connected or how application software components are allocated to execution platform components. The language can also be used to describe the dynamic behavior of the runtime architecture by providing support to model operational modes and mode transitions. The language is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. Extensions can take the form of new properties and analysis specific notations that can be associated with components and are standardized themselves.

(6) AADL was developed to meet the special needs of performance-critical real-time systems, including embedded real-time systems such as avionics, automotive electronics, or robotics systems. The language can describe important performance-critical aspects such as timing requirements, fault and error behaviors, time and space partitioning, and safety and certification properties. Such a description allows a system designer to perform analyses of the composed components and systems such as system schedulability, sizing analysis, and safety analysis. From these analyses, the designer can evaluate architectural tradeoffs and changes.

(7) AADL supports analysis of cross cutting impact of change in the architecture along multiple analysis dimensions in a consistent manner. Consistency is achieved through automatic generation of analysis models from the annotated architecture model. AADL is designed to be used with generation tools that support the automatic generation of the source code needed to integrate the system components and build a system executive from validated models. This architecture-centric approach to model-based engineering permits incremental validation and verification of system models against requirements and implementations against systems models throughout the development lifecycle.

(8) This document consists of the Error Model Annex as supplement to the SAE AADL standard that enables modeling of different types of faults, fault behavior of individual system components, modeling of fault propagation affecting related components in terms of peer to peer interactions and deployment relationships between software components and their execution platform, modeling of aggregation of fault behavior and propagation in terms of the component hierarchy, as well as specification of fault tolerance strategies expected in the actual system architecture. The objective of the Error Model Annex is to support of qualitative and quantitative assessments of system dependability, i.e., reliability, availability, integrity (safety, security), and survivability, as well as compliance of the system to the specified fault tolerance strategies from an annotated architecture model of the embedded software, computer platform, and physical system.
Information and Feedback

(9) The website at http://www.aadl.info is an information source regarding the SAE AADL standard. The website provides information and a download site for the Open Source AADL Tool Environment. It also provides links to other resources regarding the AADL standard and its use.

(10) The public AAD Wiki (https://wiki.sei.cmu.edu/aadl) maintains a list of AADL related publications by the community, and provides guidance on the use of extension of the open source OSATE tool set for AADL.

(11) Questions and inquiries regarding working versions of annexes and future versions of the standard can be addressed to info@aadl.info.

(12) Informal comments on this standard may be sent via e-mail to errata@aadl.info. If appropriate, the defect correction procedure will be initiated. Comments should use the following format:

!topic Title summarizing comment
!reference AADL-ss.ss(pp)
!from Author Name yy-mm-dd
!keywords keywords related to topic
!discussion
text of discussion

(13) where ss.ss is the section, clause or subclause number, pp is the paragraph or line number where applicable, and yy-mm-dd is the date the comment was sent. The date is optional, as is the !keywords line.

(14) Multiple comments per e-mail message are acceptable. Please use a descriptive “Subject” in your e-mail message.

(15) When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [ ] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example:

!topic [c]{C}haracter
!topic it’s meaning is not defined
1 References

1.1Normative References

(1) The following normative documents contain provisions that, through reference in this text, constitute provisions of this standard.


(6) SAE ARP-4754A, Guidelines For Development Of Civil Aircraft and Systems, December 2010.


(10) DO-254 Design Assurance Guidance for Airborne Electronic Hardware, April 2000.


1.2Informative References

(1) The following informative references contain background information about the items with the citation.


Annex Document E  Error Model

Normative

Annex E.1 Scope

(1) The purpose of the Error Model Annex is to enable qualitative and quantitative assessments of system dependability, i.e., reliability, availability, integrity (safety, security), and survivability, as well as compliance of the system to specified fault tolerance strategies from an annotated architecture model of the embedded software, computer platform, and physical system.

(2) This annex defines a sublanguage that may be associated with components of an embedded system architecture expressed in core AADL through error annex clauses. The language features defined in this annex enable specification of fault types, fault behavior of individual components, fault propagation affecting related components, aggregation of fault behavior and propagation in terms of the component hierarchy, specification of fault tolerance strategies expected in the actual system architecture.

(3) The purpose of AADL is to model the computer based embedded system, including the runtime architecture of embedded systems. In that context the Error Model Annex defines a sublanguage that can be used to declare error models within an error annex library and associate them with components in an architecture specification.

(4) In this document we use the terms impairment, fault, failure, etc. according to its IFIP WG10.4 definition (see Section Annex E.2). For the description of the language constructs of the Error Model sublanguage and as keyword in the language we use the word error.

(5) The Error Annex sublanguage supports the declaration of collections of error types and their use in specifying error propagations. These error types are associated with interactions point of components (features such as ports as well as deployment bindings) to represent incoming and outgoing error propagations with related components. Users can also specify error types that are expected to not be propagated. For each component we can also specify an error flow, i.e., whether a component is the source or sink of an error propagation, or whether it passes on an incoming propagation as an outgoing propagation of the same or different error type (path). Figure 1 illustrates an error source and two error paths for component C, one path from an incoming binding related error propagation to an outgoing port. The figure also illustrates explicit specification of an error type that is expected to not be propagated.

![Figure 1 Error Propagations and Error Propagation Flows](image)

(6) The connection topology of the architecture as well as the deployment binding of software components to platform components determines which components are affected by outgoing error propagations of other components. This allows us to identify hazards and assess the impact of failures across interacting system components. Figure 2 illustrates a port connection based error propagation path between software components, error propagation paths between hardware components connected by bus, and error propagation paths as a result of software to hardware bindings. The figure also illustrates the specification of error behavior of individual components.
(7) The Error Annex sublanguage supports the specification of error behavior in terms of an error behavior state machine with a set of states and transitions that occur under specified conditions, as well as specification error, recover, and repair events that are local to a component. They are associated with components to specify how the error state of a component changes due to error events and error propagations as well as due to repair events. Error events and states can indicate the type of error they represent by referring to error types. The error behavior specification also declares the conditions for outgoing error propagation in terms of the component error behavior state and incoming error propagations. For example, the error state of a component might change due to an error event of the component itself, and/or due to an error propagated into that component from some other component. This allows us to characterize the error behavior of an individual component in terms its own failures, impact of other component failures on it, and whether it impacts other components.

(8) The Error Annex sublanguage supports the specification of a component’s composite error behavior as error states expressed in terms of the error states of its subcomponents. For example, a component having internal redundancy might be in an erroneous state only when two or more of its subcomponents are in an erroneous state. The resulting composite error behavior of the component must be consistent with an abstracted error behavior specification expressed by an error behavior state machine. This allows us to model error behavior at different levels of architectural abstraction and fidelity and keep these specifications consistent.

(9) The Error Annex sublanguage supports the specification of the impact of errors propagating from the components involved in performing a transfer specified by a connection on the communicated information.

(10) Finally, the Error Annex sublanguage supports the specification of how components detect and mitigate errors in their subcomponents or the components on which they depend through redundancy and voting. In addition constructs are provided to link the specified error behavior with the health monitoring and management elements of a system architecture and its behavior expressed in core AADL and the AADL Behavior Annex published in SAE AS5506-2.

(11) The language features defined in this annex can be used to specify the risk mitigation methods employed in embedded computer system architectures to increase reliability, availability, integrity (safety, security), and survivability. The error behavior of a complete system emerges from the interactions between the individual component error behavior models. This annex defines the overall system error behavior as a composition of the error models of its components, where the composition depends on the structure and properties of the architecture specification. More formally, a component error model is a stochastic automaton, and the rules for composing component stochastic automata error models to form a system error model depend on the potential error propagations and error management behaviors declared in the architecture specification.

(12) The Error Model Annex can be used to annotate the AADL model of an embedded system to support a number of the methods cited in SAE ARP4761, “Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.” An architecture specification with error annex annotations may be subjected to a variety of analysis methods ranging from Functional Hazard Assessment (FHA), Failure Mode and
Effect Analysis (FMEA), Common Cause Analysis (CCA) and Fault Tree Analysis (FTA) to stochastic reliability and availability analysis. For example, fault trees can be generated from such specifications to assess safety; stochastic process models (Markov or Stochastic Petri nets) can be generated to assess reliability and availability.

(13) From the bottom-up, the error models of low-level components typically capture the results of failure modes and effects analysis (e.g. as failure modes and effects analysis is defined in SAE ARP 5580). From the top-down, the error models of the overall system and high-level subsystems typically capture the results of system hazard analysis (e.g. as hazard analysis is defined in SAE ARP 4761). The rules defined in this annex assure that the results of these analyses as captured in an architecture specification are consistent and complete with respect to each other. For example, this enables an integrated approach that insures consistency and completeness between hazard analysis, failure modes and effects analysis, and the safety and reliability analyses that relate the two.

(14) This annex supports a compositional approach to modeling different dependability concerns. This enables reuse of error models and leverages reuse of component specifications in core AADL. Modifications to architecture specifications are propagated into safety and reliability models by automatically regenerating them. Architectural abstraction and composite error behavior specifications support mixed-fidelity modeling, and enable improved traceability and consistency between architecture specifications and models and analysis results.

(15) The Error Model Annex definition is organized into the following sections. Section Annex E.2 introduces concepts and terminology used in this annex document. Section Annex E.3 describes the two major groups of Error Model Annex language constructs, reusable Error Model libraries, and component-specific Error Model subclauses. Section Annex E.5 introduces constructs to define hierarchies of error types and type sets. Section Annex E.6 describes a set of predeclared error types. Section Annex E.7 introduces constructs in support of error propagation specification. Section Annex E.8 introduces constructs to define reusable error behavior state machines with error and repair events, states, and transitions. Section Annex E.9 describes constructs to support specification of error behavior of components in terms of an error behavior state machine, transition trigger conditions in terms of incoming error propagations, conditions for outgoing error propagation in terms of error behavior states and incoming error propagations, and error detection events. Section Annex E.10 describes constructs to support specification of composite error behavior of components based on the error state behavior of a set of subcomponents. Section Annex E.11 introduces constructs that allow you to define reusable error type mappings and transformations. Section Annex E.13 discusses the interaction between the error behavior specification expressed by Error Model Annex constructs and the health monitoring and fault management capabilities in the actual system architecture. The document closes with examples of system architecture models annotated with Error Model Annex specifications.

Annex E.2 Concepts and Terminology

(1) The definitions of this section are based on the concepts and terminology defined by IFIP Working Group 10.4 on Reliable Computing and Fault Tolerance, and the IEEE Computing Society Technical Committee on Fault Tolerant Computing [IFIP WG10.4-1992]. They address impairment to system and component functionality and means of mitigating the impairment. The terms fault, error, failure, and latency of a fault or error are those of [IFIP WG10.4-1992].

(2) The terms fault, fault latency, error, error latency, failure, and error propagation have the same meaning as defined in Section 1.4 of AS5506B, but are restated here in words that relate somewhat more clearly to other terms used in this annex.

(3) A fault is a root (phenomenological) cause of an error or a failure, i.e., an anomalous undesired change in the structure or data within a component that may cause that component to eventually fail to perform according to its nominal specification. Examples of faults overheating of hardware circuits, or programmers making coding mistakes when producing source text. Faults are the phenomenological cause of a failure. Failures and errors are effects of a fault.

(4) An error is the difference in state from a correct state. The activation of a fault places a component into such an error state (possibly after some delay called the fault latency). An erroneous component may persist in that error state for some period of time before it behaves in a way that violates its nominal specification (called the error latency). For example, a burned out transistor (an activated fault) in an adder circuit does not cause a processor to violate its nominal specification until that circuit is used (after a fault latency) and produces an incorrect output value (erroneous state information).
A failure is a deviation in behavior from a nominal specification as a consequence of an error. The deviation, i.e., the failure, can be characterized by type, persistence, and degree of severity. The degree to which a failure affects nominal behavior is referred to as criticality of the failure.

A propagation is the creation of a new error or failure due to an error or failure, i.e., a component failure may affect the nominal behavior other components that the failed component interacts with. This may occur with some error latency after the component first enters an erroneous state. For example, an erroneous component may send an incorrect value in a message or exceed its specified worst-case execution time.

The error behavior of a component is specified through an error behavior state machine that consists of states and transitions. The states of this error behavior state machine can represent both error-free (working) states and error (nonworking) states. Fault activation is represented as an error event. An error event can cause a change in the error behavior state of a component.

Error propagation is represented in an Error Model specification by specifying outgoing and incoming error propagations for each of the component features and for component bindings to the computer platform. The error propagation path from an outgoing error to an incoming error is determined by the core AADL model through the connections between the respective features and the binding declarations. A component can be an error propagation source, sink, or pass on a propagated error as the same of different type. This role is specified by an error propagation flow, which allows for fault impact analysis without needing to know the internal error behavior of a component.

A propagated error may place the receiving components into an error state under certain conditions that can be specified in transition conditions. A component receiving the propagated error may detect and mask the propagated error.

A failed component may be able to recover from the failure and return to a working state. This is expressed by a recover event that can trigger a transition from a nonworking state to a working state after a specified amount of time.

A failed component may also encounter a repair action. For example, a failed component may be replaced. A repair action restores a component to an error-free state. This is expressed by a repair event that can trigger a transition indicating the start or completion of a repair action.

System components may be composed of subcomponents. The error behavior of the subcomponents affects the error behavior of the enclosing component. For example, a system component with internal redundancy may enter a nonworking state if both is its redundant subcomponents are in a nonworking state. Similarly, a subcomponent may propagate an error to components outside the enclosing component if it interacts with those components via the enclosing component interface. The Error Model Annex supports a compositional approach to specifying such error behavior. A component may be annotated with an error behavior state machine that characterizes its error behavior abstractly, i.e., independent of whether it contains subcomponents. Separately, the modeler can specify a composite error behavior in terms of subcomponent error behavior, which must be consistent with the abstract error behavior specification.

In the terminology of a failure modes and effects analysis, an error state is a failure mode that, if associated with an outgoing propagated error, can affect other components. A failure effect is resulting error state of a component due to an incoming propagated error. The relationship between the failure mode and the failure effect is determined by the connection between the components or by the deployment binding.

It is common practice to assume that the risk due to generic or design errors introduced by development-time faults (also known as design defects or bugs) has been reduced to an acceptable level prior to the start of system operation through the use of appropriate design assurance methods. For example, RTCA DO-178B and RTCA DO-254 provide guidelines for assuring that fielded avionics software code and hardware circuits have acceptably low risk of consequential design errors. For mechanical components the error behavior specified by the Error Model Annex primarily reflects physical failure of the component in use. In the case of software the error behavior specification reflects the activation of design and coding errors and their impact on the system. In other words, a error event corresponds to a run-time flow of control and data that reveals a design defect (introduced a run-time error due to a design defect), rather than to the development-time introduction of that design defect.
There are many means of mitigating impairment of systems or components. **Fault avoidance** is to apply techniques that focus on not introducing faults or eliminating them before the system goes into operation. **Fault tolerance** focuses on fault detection, fault isolation, and fault recovery. Fault recovery elements include fault containment, fault masking, fault repair, and fault correction. The Error Model Annex supports specification of faults that are expected to be avoided, masked, propagated, detected, and recovered from. Fault tolerance support in the actual system is modeled through the AADL language and the Behavior Annex.

### Annex E.3 Error Model Libraries

1. Error Model libraries contain reusable declarations, such as sets of error types and error behavior state machine specifications that include error and repair events. Error Model libraries are declared in packages. Those reusable declarations can be referenced in annex subclauses by qualifying them with the package name.

   **Syntax**

   ```
   error_model_library ::= 
   annex EMV2 ( 
     {** error_model_library_constructs **} ) 
   | none ) ;
   ```

   ```
   error_model_library_constructs ::= 
   [ error_type_library ] 
   { error_behavior_state_machine }* 
   { type_mapping_set }* 
   { type_transformation_set }* 
   ```

   ```
   error_model_library_reference ::= 
   package_or_package_alias_identifier
   ```

   **Naming Rules**

   (N1) An Error Model library provides an Error Model specific namespace for reusable items, such as error types, type sets, error behavior state machines, type mapping sets, and type transformation sets. Defining identifiers of reusable items must be unique within this Error Model specific namespace.

   (N2) An Error Model library contained in a different package is referenced by the package name that contains the Error Model library. The referenced package name must be listed in the with clause of the package that contains the Error Model library reference (see AS-5506B Section 4.2).

   **Semantics**

   (2) Error Model annotations to a core AADL model in the form of error model libraries and error model subclauses specify the fault behavior in a system and its components. The nominal operational behavior of the system and its components as well as the response of the system to the failure and to its detection and recovery/repair through fault management is represented by modes in the core language and Behavior Annex annotations. The interaction between the error behavior specification and the modes of a system and its components is addressed in section Annex E.9.3.

   (3) An Error Model library provides reusable specifications of sets of error types and of error behavior specifications expressed through error behavior state machines. Those reusable declarations can be referenced by annex subclauses by qualifying them with the package name.
Annex E.4 Error Model Subclauses

(1) Error Model subclauses allow component types and component implementations to be annotated with Error Model specifications. Those component-specific Error Model specifications define incoming and outgoing propagated errors for features, error flows from incoming to outgoing features. They also specify component error behavior in terms of an error behavior state machine augmented with transition conditions based on incoming propagated errors, conditions for outgoing propagation, and event signaling the detection of errors in the system architecture. Finally, they specify the composite error behavior of a component in terms of the error behaviors of its subcomponents.

Syntax

\[
\text{error\_model\_subclause} ::= \\
\text{annex EMV2 (} \\
\quad ( \{** \text{error\_model\_component\_constructs} **} ) \\
\quad \mid \text{none} \\
\quad [ \text{in\_modes} ] ;
\]

error_model_component_constructs ::= 
\[
\quad \text{[ error\_propagations ]} \\
\quad \text{[ component\_error\_behavior ]} \\
\quad \text{[ composite\_error\_behavior ]} \\
\quad \text{[ observable\_connections ]} \\
\quad \text{[ connection\_error\_behavior ]} \\
\quad \text{[ subclause\_properties\_section ]}
\]

Naming Rules

(N1) The mode identifiers in the \text{in\_modes} statement must refer to modes in the component type or component implementation for which the annex subclause is declared.

Legality Rules

(L1) Annex subclauses can only be declared in component types and component implementations.

Semantics

(2) An Error Model subclause allows component types and component implementations to be annotated with error events to represent activation of component fault activations, repair events to represent initiation and completion of a repair activity, incoming and outgoing propagation of errors through component features as well as through bindings to platform components, error behavior of the component expressed as a state machine in terms of its fault and repair events and incoming error propagations as well as resulting in outgoing error propagations, error behavior of a component as a composite of the error behavior of its subcomponents, and a mapping between the error behavior expressed in the Error Model specification and the behavior of the fault/health management portion of the embedded system architecture expressed in core AADL and the Behavior Annex specifications.

(3) Error Model subclauses can be declared to be applicable to specific modes by specifying them with a \text{in\_modes} statement. An Error Model subclause without an \text{in\_modes} statement contains Error Model statements that are applicable in all modes. This capability allows users to attach mode specific Error Model annotations to core AADL models. In particular, it permits mode-specific error behavior transitions, outgoing propagation conditions, and detection conditions to be specified. Similarly, it permits mode-specific composite error behavior specifications to be declared.
(4) Error Model libraries and Error Model subclauses can contain property associations. Those property associations may have mode specific values. This allows for mode specific parameterization of Error Model libraries and Error Model subclauses without requiring separate library declarations or subclauses to be declared for each mode. For example, we can associate different occurrence probability values to an error event for different operational modes.

(5) Error Model subclauses are declared within component types and component implementations. The error propagations section, component error behavior section, and composite error behavior section of an Error Model subclause may be inherited. An implementation may inherit these sections from a component implementation it extends, or from its respective component type. Similarly, a component type may inherit these sections from another component type it extends. Local declarations of these sections override inherited sections. For example, for a component type that is an extension of another component, the local error propagations section replaces any error propagation declarations in an error propagations section of the component type being extended.

Annex E.5 Error Types, Type Sets, and Type Tokens

(1) In this section we introduce the concepts of error type and error type set. They are declared in an Error Type library, i.e., the error types section of an Error Model library.

(2) An error type is used to indicate the type of fault being activated, the type of error being propagated, or the error type represented by the error behavior state of a system or component.

(3) Error types can be organized into different type hierarchies, e.g., types relating to value errors and types relating to timing errors. Error types that are part of the same type hierarchy are considered mutually exclusive alternatives. For example, an error propagation may represent a late or an early value, but not both simultaneously.

(4) An error type product represents a combination of error types that can occur simultaneously. For example, the combination of OutOfRange and LateDelivery expressed as error type product OutOfRange * LateDelivery.

(5) An error type token is an instance of an error type or error type product. It represents an instance of an error occurrence, e.g., an error event or error propagation occurrence. For example, a token may represent an error propagation that consists of an out of range value that is on time, expressed as single-valued token OutOfRange, or an out of range value that is also late, expressed as a two-valued token OutOfRange * LateValue.

(6) An error type set is defined as a set of mutually exclusive elements. An error type set element can be an error type or error type product. For example, an error type set may be defined as consisting of the elements BadValue and LateDelivery, expressed as {OutOfRange, LateDelivery}. When an error type has subtypes, all of the subtypes are included. In the case of a product all subtype combinations are included.

(7) Error types and type sets can be defined to have alias names that better reflect the application domain. The error type or type set and its alias are considered to be equivalent.

(8) An Error Type library can be defined as an extension of an existing Error Type library, adding new error types into the error type hierarchy, defining new error type sets as well as aliases for error types and error type sets.

(9) Predeclared sets of error types have been defined with this Error Model Annex standard (see Section Annex E.6). They can be extended with user defined error types or renamed with aliases.

(10) Error Model Annex properties have been defined in the property set EMV2.

Syntax

```
error_type_library ::= 
  error types 
  [ extends error_type_library_list with ] 
  { error_type_library_element }* 
  [ properties ]
```
error_type_library_list ::= error_model_library_package_reference { , error_model_library_package_reference }

error_type_library_element ::= 
  { error_type_definition | error_type_alias | error_type_set_definition | error_type_set_alias }

error_type_definition ::= defining_error_type_identifier : type
  [ extends error_type_reference ];

error_type_alias ::= defining_error_type_alias_identifier renames type error_type_reference ;

error_type_set_definition ::= 
  defining_error_type_set_identifier : type set error_type_set_constructor ;

error_type_set_alias ::= 
  defining_error_type_set_alias_identifier renames type set error_type_set_reference ;

error_type_set_constructor ::= 
  { element_error_type ( , element_error_type ) }*

error_type_set ::= 
  error_type_set_reference |
  error_type_set_constructor

element_error_type ::= 
  error_type_reference | error_type_product

error_type_product ::= 
  error_type_reference ( * error_type_reference )+

error_type_token ::= 
  { element_error_type }
error_type_reference ::=  
  [ error_model_library_reference :: ] error_type_identifier

error_type_set_reference ::=  
  [error_model_library_reference :: ] error_type_set_identifier

-- adapted from AS5506B 11.3

emv2_contained_property_association ::=  
  unique_property_identifier => [ constant ] assignment applies to  
    emv2_annex_specific_path  
    { , emv2_annex_specific_path }*  
    [ in_binding ] ;

emv2_annex_specific_path ::=  
  named_element_identifier { . named_element_identifier }*

subclause_properties_section ::=  
  properties  
    { emv2_contained_property_association }+

Naming Rules

(N1) The Error Type library utilizes the namespace of the enclosing Error Model library, sharing it with defining identifiers for Error Behavior State Machines, Type Mapping Sets, and Type Transformation Sets.

(N2) An Error Type library may be an extension of one or more Error Type libraries listed in the extends with clause. The defining error type and type sets of those libraries are inherited into the namespace, i.e., become accessible as part of this Error Type library, e.g., in use types clauses. The inherited identifiers from different Error Type libraries must not conflict with each other.

(N3) The defining identifier of an error type or error type alias must be unique within the namespace of the Error Type library of the package that contains the defining error type declaration, i.e., must not conflict with locally defined or inherited identifiers.

(N4) The defining identifier of an error type set or error type set alias must be unique within the namespace of an Error Type library of the package that contains the defining error type declaration, i.e., must not conflict with locally defined or inherited identifiers.

(N5) The error type or type set reference of an error type alias or error type set alias declaration must exist in the namespace of the Error Model library containing the reference, unless it is qualified with an Error Model library reference.

(N6) The error type reference in the extends clause of an error type declaration must exist in the namespace of the Error Model library containing the reference, unless it is qualified with an Error Model library reference.
The optional qualifying Error Model library reference of an error type or type set reference must adhere to Naming Rule (N2) in Section Annex E.3.

All error types referenced in an error_type_set_constructor must exist in the namespace of the Error Model library containing the reference.

All error type references of error_type_token or error_type_set_constructor, when contained in Error Model subclauses and not qualified by an Error Model library reference, are resolved according to the Naming Rules for use types clauses (see Section Annex E.7 Naming Rule (N2)).

Predeclared Error Model properties must be qualified with the property set name EMV2.

The emv2_contained_property_association is used to declare property associations within EMV2 annex libraries and annex subclauses. The containment path starts as an annex specific path (see AS5506B Section 11.3).

The emv2_annex_specific_path of an error_type_emv2_contained_property_association must consist of reference to an error type identifier or error type set identifier that is defined in the namespace of the Error Type Library.

The emv2_annex_specific_path of an emv2_contained_property_association may start with a sequence of subcomponent references before referencing elements in emv2 annex of the last referenced component. Such a path does not require the special syntax for identifying the annex as required by the contained property association of AS5506B Section 11.3.

**Legality Rules**

An Error Type library can contain more than one error type hierarchy, i.e., a root error type that is not a subtype of another error type.

An error type cannot be the subtype of more than one other error type. This is enforced syntactically.

For different elements in a type set that reference a single error type, one must not be a subtype of another.

Different element types of an error type product must not be from the same error type hierarchy.

Different error type products in a type set must not represent overlapping sets of error type tokens. In other words, for two error type products with the same number of element types and whose element types are from the same type hierarchies, the element type of one must not be a subtype of the other.

**Semantics**

An Error Type library allows the modeler to declare error types and error type sets. An error type or error type set can be referenced by its identifier if it is defined within the same Error Type library, or qualified by the Error Model library containing the Error Type library. In Error Model subclauses, the use types clause makes the Error Type library namespace accessible without requiring qualification.

An Error Type library can be defined as an extension (extend with) of one or more existing Error Type libraries. All the error type and error type set declarations from those libraries become accessible as part of the Error Type library. This allows new error types to be added into the error type hierarchy, new error type sets to be introduced, and aliases to be defined for error types and type sets.

Error types are used to indicate the type of an error event, an error flow, an incoming or outgoing error propagation, or an error behavior state.

An error type can be placed into an inheritance type hierarchy by declaring it as a subtype of another error type using the extends keyword. Error types that are subtypes of a given error type are assumed to be mutually exclusive alternatives. For example, an error propagation may propagate an error of type IncorrectValue or of type OutOfRange, but not simultaneously if both error types are part of the same type hierarchy.
An error type product represents combinations of error types that can occur simultaneously, i.e., elements of an error type product cannot be from the same type hierarchy. For example, the combination of `ValueError` and `LateDelivery` expressed as error type product `ValueError * LateDelivery`. Error type products are unordered, i.e., `ValueError * LateDelivery` is the same as `LateDelivery * ValueError`. If the elements of the error type product have subtypes, then the product represents combinations of the subtypes from each element types. In our example, the product includes late delivery of benign and subtle value errors, with benign expanding into out of range and out of bounds value errors.

An error type token is an instance of an error type or error type product. It represents an instance of an error occurrence, e.g., an error event or error propagation occurrence. For example, a token may represent an error propagation that consists of an out of range value that is on time, expressed as single-valued token `OutOfRange`, or an out of range value that is also late, expressed as a two-valued token `OutOfRange * LateValue`.

An error type set represents a set of error types that can be associated with a typed error event, flow, propagation, or state. An error type set is defined as a consisting of elements of one type or of error type products of two or more types. The elements of an error type set are mutually exclusive, i.e., represent non-overlapping sets of error type tokens.

If an element of an error type set is a single error type, then it represents single-valued error type tokens of the specified type of all its subtypes. If two elements specifying a single error type are from the same type hierarchy, then one cannot be a subtype of the other, but they can be different subtypes of the same super root type.

If an element of an error type set is an error type product of two or more (k) error types, then represents k-valued error type tokens. If the element types of an error type product have subtypes, then error type tokens exist for each of the subtypes of each element type in the error type product. For example, we may specify that error propagations may propagate errors that may be a combination of value and timing errors `{ValueError * TimingError}` (see Section (6)). Examples of a specific instance of a propagation are `OutOfRange * LateDelivery, OutOfRange * EarlyDelivery, or SubtleValueError * LateDelivery`.

An error type set acts as a constraint on error type tokens for error propagations, error flows, error behavior states, as well as conditions for error behavior state transitions, conditions for outgoing error propagations, and conditions for error detection. For example, an error type set associated an error event specified as trigger condition for an error state transition, indicates that the transition trigger is only satisfied if the error type token of an actual error event is contained in the specified error type set.

An error event or error flow source is a source of error type tokens. A generated error type token must be contained in the specified error type set. In the case of an error type set whose elements have subtypes only tokens with the leaf subtypes are generated. If an occurrence probability is specified an error event with the appropriate error type token is generated with the specified probability.

Error type sets on outgoing and incoming error propagations represent contracts and assumptions. In other words, the outgoing error type set must be contained in the incoming error type set.

An error type may be declared as an alias (renames) of another error type. The two error types are considered to be equivalent with respect to the type hierarchy. The renames clause allows domain specific names to be introduced without extending the type hierarchy. For example, the alias `OutOfCalibration` may be a more meaningful name for the error type `SubtleValueError` in the context of a sensor.

An error type set can be declared as an alias (renames) of another error type set. The two error type sets are considered to be equivalent with respect to type matching of propagations.

**Type System of Error Types**

Error types and type sets impose a type system onto an error model. Different elements of an Error Model specification must be type consistent. Type consistency is characterized in terms of type containment. For that purpose, we treat the error type that is associated with an error event, propagation, or state as a type set with a single element type. We define type containment as follows:
Annex E.6 A Common Set of Error Types

This section introduces a common set of error types organized into type hierarchies as predeclared error types. The focus of these predeclared error types is on characterizing types of errors propagated between components along error propagation paths. Error propagation paths are explicitly recorded connections and bindings in the AADL model. The predeclared error type hierarchies can also be used to characterize error events and error states.

1. Aliases can be defined for existing error types that are more meaningful to a specific component, such as No Power instead of Service Omission. Furthermore, the predeclared error types can be extended with additional subtypes.

2. User-defined error type hierarchies can be introduced to characterize error propagations, error events, and error states that are not covered by the predeclared error types.

3. In the case of software components that operate in a fault container (process with runtime enforced address space protection, or partition with both space and time partition enforcement), the error propagation is limited to propagation paths along explicit interaction channels (port connections, shared data access, subprogram service calls) and execution platform bindings (see Section E.7.3). This allows us to map a large number of software component faults into a limited number of error propagation error types. For example, a divide by zero in an arithmetic expression or a deadline miss by a periodic thread may manifest itself as an omission of output that can be observed by the recipient of this output. Observable propagation points can be used to record propagation paths that are not explicit, such as hardware components affecting each other through heat due to location proximity, or threads affecting each other by modifying code or data intentionally.

4. Various safety analyses, e.g., Hazard and Operability Studies (HAZOP) in the process industry [CISHEC 1977, HAZOP 1992], and its adaptation to software as Software Hazard Analysis and Resolution in Design (SHARD) [SHARD 1994], have developed a set of guide words to help identify hazards to be addressed. The Dependability Analysis and Modeling (DAM) profile for UML report [DAM 2008] includes a survey of literature regarding dependability analysis concepts including failure modes. [Bondavalli 1990, Powell 1992] have defined error types in the value and time domain based on the concept of a service delivered by a system (or system component) as a sequence of service items. We adapt this approach to introduce a common set of error types. [Walter 2003] propose a Customizable Fault/Error Model (CFEM) that organizes diverse fault categories into a cohesive framework by classifying faults by the effect they have on system services.

First we define a model of system service delivered to a user in terms of the sequence of service items. We then give definitions of service errors in the value and time domains as perceived by an omniscient observer of that
service. An omniscient observer can decide (a) whether the observed service item was indeed expected and (b) whether its value and time are correct. Note that a real (non-omniscient) observer may not be able to detect each of the types of error.

(7) An action is something done by a component visible outside the component, i.e. port output, subprogram call.

(8) An action sequence, A, performed by a component on feature f can be defined in terms of a sequence of actions, aᵢ, i = 1,2,... each characterized by a pair (vi, δᵢ) where vi is the value or content of action aᵢ and δᵢ is the duration (time interval) of action aᵢ or the empty action, ε, which has no value nor duration. Durations have starting and ending times, start(δᵢ) and end(δᵢ), which may the same for those actions which occur in an instant.

(9) An action is defined to be correct, i.e., have no error, iff: (vi ∈ Vi) ∧ (δᵢ ∈ Di) where Vi and Di are respectively the correct sets of values and durations for action aj.

(10) For many systems, the specified value and time sets are reduced to the special cases Vi = {vi} (a single value) and Di = {[start(δᵢ), end(δᵢ)]} (a single time interval). Examples of systems where the general case must be considered are: for multiple value sets, one variant of a set of diverse-design software systems and, for multiple time period sets, a system accessing a shared channel by some time-slot mechanism.

(11) Let V represent expected range (or set) of possible values delivered by actions, i.e., ∀i, (Vi ∈ V). Let D represent the expected duration of operation, i.e., ∀i, (Di ∈ D).

(12) We define error types for the action as a whole, as Value errors and Timing errors of individual service items, Rate errors and Sequence errors with respect to a sequence of service items, Replication errors in terms of sets of replicated service items, and Concurrency errors with respect to access of a shared resource. We consider these error types as independent dimensions of error types. An error propagation may have a combination of these error type categories, e.g., we may have a service that delivers incorrect values at the wrong rate but as consistent replicates.

Figure 3 Service Error Type Hierarchy
package ErrorLibrary
public
annex EMV2 {**

error types

ServiceError: type;
ItemComission: type extends ServiceError;
ItemOmission: type extends ServiceError;
ServiceOmission: type extends ItemOmission;
LateServiceStart: type extends ServiceOmission;
EarlyServiceTermination: type extends ServiceOmission;
BoundedOmissionSequence: type extends ServiceOmission;
BoundedOmissionInterval: type extends ServiceOmission;
ServiceCommission: type extends ServiceError;
EarlyServiceStart: type extends ServiceCommission;
LateServiceTermination: type extends ServiceCommission;
TimingError: type;
EarlyDelivery: type extends TimingError;
LateDelivery: type extends TimingError;
ValueError: type;
SubtleValueError: type extends ValueError;
BenignValueError: type extends ValueError;
OutOfRange: type extends BenignValueError;
OutOfBounds: type extends BenignValueError;
BelowRange: type extends OutOfRange;
AboveRange: type extends OutOfRange;
RateError: type;
VariableRate: type extends RateError;
HighRate: type extends RateError;
LowRate: type extends RateError;
SequenceError: type;
ValueChange: type extends SequenceError;
OutOfOrder: type extends SequenceError;
ReplicationError: type;
InconsistentValue: type extends ReplicationError;
InconsistentApproximateValue: type extends InconsistentValue;
InconsistentTiming: type extends ReplicationError;
InconsistentItemOmission: type extends ReplicationError;
InconsistentServiceOmission: type extends InconsistentOmission;
ValueCorruption renames type ValueError;
SubtleValueCorruption renames type SubtleValueError;
BenignValueCorruption renames type BenignValueError;
IncorrectValue renames type ValueError;
BadValue renames type SubtleValueError;
EarlyData renames type EarlyDelivery;
LateData renames type LateDelivery;
AsymmetricValue renames type InconsistentValue;
SymmetricValue renames type ValueError;
AsymmetricItemOmission renames type InconsistentItemOmission;
SymmetricItemOmission renames type ItemOmission;
AsymmetricServiceOmission renames type InconsistentServiceOmission;
SymmetricServiceOmission renames type ServiceOmission;
AsymmetricTransmissive renames type InconsistentValue;
SymmetricTransmissive renames type ValueError;
ConcurrencyError: type;
RaceCondition: type extends ConcurrencyError;
ReadWriteRace: type extends RaceCondition;
WriteWriteRace: type extends RaceCondition;
MutExError: type extends ConcurrencyError;
Deadlock: type extends MutExError;
Starvation: type extends MutExError;
end types;
**}
end ErrorLibrary;

E.6.1 Service Related Errors

(13) Service errors \([ServiceError]\) are errors with respect to the action sequence as a whole rather than individual actions. We distinguish between Item Omission errors to represent lack of action when it ought, and Item Commission errors to represent actions when it oughtn’t.

(14) Item Omission \([ItemOmission]\) is perceived as a permanent error in that no actions are performed for a period during the action sequence. In other words, sometime during operation, some actions were erroneously omitted. Late service start and early service termination are interpreted as a form of service omission.

Item Omission: \(\exists a_i \in A \mid a_i = \epsilon\) where \(\epsilon\) is the empty action.

(15) Item Commission \([ItemCommission]\) is perceived as an impromptu service in that service items are provided when not expected. Real observers may observe such errors as Value Error or Timing Error. Early service start and late service termination are interpreted as a form of service commission.

Item Commission: \(\exists a_i \in A \mid v_i \notin V\) and \(\delta_i \notin D\)

(16) Late Service Start \([LateServiceStart]\) is represents a error where no service items are provided for a period at the beginning of the service. The first action is \(a_i\).

Late Service Start: \(\exists i \mid \forall j < i \mid a_j = \epsilon\)
(17) **Early Service Termination** [EarlyServiceTermination] is perceived as a permanent error in that no service items are provided after the point of failure. The last action is $a_i$.

*Early Service Termination*: $\exists i \mid \forall j > i \mid a_i = \varepsilon$

(18) **Early Service Start** [EarlyServiceStart] is represents a fault where service items are provided for a period before the beginning of the expected service. Let $\delta_1$ be the time of the first correct action.

*Early Service Start*: $\exists a_i \in A \mid \delta_1 < \delta_i$

(19) **Late Service Termination** [LateServiceTermination] is perceived as a fault in t hat service items are provided after the end of a service period. Let $\delta_f$ be the time of the final correct action.

*Late Service Termination*: $\exists a_i \in A \mid \delta_i > \delta_f$

(20) **Service Omission** [ServiceOmission] occurs when multiple item omissions preclude delivery of a service. For example, the CRC on satellite transmission allows some lo st packets, but beyond the limit of the CRC, further packet loss causes loss of communication. Let $[a_i, a_j]$ for $i < j$ represent the sequence of actions between $a_i$ and $a_j$, inclusive, and predicate $S$ when applied to action sequences, express conditions constituting service omission.

*Service Omission*: $\exists [a_i, a_j] \subset A \mid S([a_i, a_j])$ with $S$ defined as $\forall a_k \in [a_i, a_j] \mid a_k = \varepsilon$.

(21) A $k$ **Bounded Omission Sequence** error occurs when more than a certain number of consecutive service item omissions occur. A parameter $k$ specifies the number of acceptable consecutive item omissions.

*Bounded Omission Sequence error*: $\exists [a_i, a_{i+k-1}] \subset A \quad \forall \ a_j \in [a_i, a_{i+k-1}] \mid a_k = \varepsilon$.

(22) A $k$ **Bounded Omission Interval** error occurs when a service item omission is followed by a second service item omission before $k$ correct service items are delivered. A parameter $k$ specifies the expected minimum interval between two item omissions.

*Bounded Omission Interval error*: $\exists [a_i, a_{i+k-1}] \subset A \quad \exists a_j \in [a_i, a_{i+k-1}] \mid a_k = \varepsilon$.

**E.6.2 Value Related Errors**

(23) **Value Error** [ValueError] represents any kind of erroneous value.

*Value Error*: $\exists a_i \in A \mid v_i \notin V_i$

(24) We distinguish between the following value errors: Subtle and Benign value errors [Walter 2003]. Subtle value errors are not detectable without information from additional sources (inline redundancy such as CRC, or redundancy by replication), while benign value errors are detectable by examination of the value alone. An example of subtle value error is a **Value Corruption** error. An example of benign value error is a **Out Of Range** error with two subtypes **Below Range**, and **Above Range**.

(25) **Benign Value Error** [BenignValueError] is detectable from the value itself, perhaps because it's out of range or has parity error. Let predicate $B$ represent detection of a benign error, $B(v)$.

*Benign Value Error*: $\exists a_i \in A \mid v_i \notin V_i \land B(v_i)$

(26) **Out Of Range** [OutOfRange] error is perceived when a service item value falls outside the range of expected values for the service. If the correct expected range is known to a real observer this error is detectable. We also define two error sub-types called **Above Range** [AboveRange] error and **Below Range** [BelowRange] error. The expected range of values is represented by $[\min(V), \max(V)]$.

*Below Range error*: $\exists a_i \in A \mid v_i < \min(V)$

*Above Range error*: $\exists a_i \in A \mid v_i > \max(V)$

*Out Of Range error*: $\exists a_i \in A \mid v_i > \max(V) \text{ OR } v_i < \min(V)$
Out Of Bounds error is perceived when a service item value falls outside an acceptable set of values as determined by an application domain function, e.g., the stable control bounds of a control algorithm. Let predicate $O$ represent detection of an out-of-bounds error, $O(v)$.

$$Out\ Of\ Bounds\ error: \exists a_i \in A \mid v_i \not\in V_i \land O(v_i)$$

Subtle Value Error occurs when an action's value falls within the range of expected values for the service, but is not a correct value (as perceived by an omniscient observer). Examples are value expressed in terms of a wrong measurement unit, erroneous values due to an incorrect computation, or readings from an incorrectly calibrated sensor that fall within the expected range of values. Such errors require additional contextual information to be detectable.

$$Subtle\ Value\ Error: \exists a_i \in A \mid v_i \not\in V_i \land \neg B(v_i)$$

Value Corruption error results from failure in the resources used by a system to perform its service, such as memory, or to communicate its service items, such as buses and networks. The effect is a value error in the service item that may be benign or subtle. Corrupted value errors can become detectable through the use of value redundancy. Value redundancy can take the form of inline redundancy, such as error-detection codes that are carried with the value, or replication redundancy (see Replication Errors). NOTE: ValueCorruption is an alias for ValueError to indicate that it could be a benign or subtle value error. Two additional aliases identify BenignValueCorruption as BenignValueError and SubtleValueCorruption as SubtleValueError.

### E.6.3 Timing Related Errors

Timing Error represents a service item being delivered outside its expected time range. This applies to a single communication step or to an end-to-end flow. Remember $D_i$ is the correct duration for action $a_i$.

$$Timing\ error: \exists a_i \in A \mid \delta_i \not\in D_i$$

We distinguish between the following Timing errors: Early Delivery error, Late Delivery error, infinitely late Item omission errors, and unexpected Item Commission errors.

Early Delivery error occurs when a service item value is delivered before the expected time range. Note that an early delivery may be perceived if an impromptu service item delivery occurs (see Sequence errors).

$$Early\ Delivery\ error: \exists a_i \in A \mid \delta_i < D_i$$

Late Delivery error occurs when a service item value is delivered after the expected time range. Note that a late delivery may be perceived if a service item delivery is skipped (see Sequence errors).

$$Late\ Delivery\ error: \exists a_i \in A \mid \delta_i > D_i$$

NOTE: We could introduce Delivery Jitter as an error type. This would express timing variability over a sequence of service items.

### E.6.4 Rate Related Errors

Rate Error occurs when actions occur at a rate that is different from the expected rate.

$$Rate\ Error: \exists a_i \in A \mid \delta_i - \delta_{i-1} \neq P$$ where $P$ is the expected period between two consecutive actions.

We distinguish between the following Rate Errors for a sequence of service items: High Rate error, Low Rate error, and Variable Rate error.

High Rate error occurs when a service delivers service items at a rate that is higher than the expected rate.

$$High\ Rate\ error: \exists a_i \in A \mid \delta_i - \delta_{i-1} < P$$ where $P$ is the expected period between two consecutive actions.

Low Rate error occurs when a service delivers service items at a rate that is lower than the expected rate.
Low Rate error: \( \exists a_i \in A \mid \delta_i - \delta_{i+1} > P \) where \( P \) is the expected period between two consecutive actions.

(38) Variable Rate [VariableRate] error occurs when a service delivers service items at a rate that is different at different points in time.

Variable Rate error: \( \exists a_i, a_j \in A \mid \delta_i - \delta_{i+1} \neq \delta_j - \delta_{j+1} \)

E.6.5 Sequence Related Errors

(39) We distinguish between the following Sequence errors: Bounded Value Change error, Out Of Order error, and Stuck Value error.

(40) Bounded Value Change [BoundedValueChange] error occurs when a service delivers service items whose value changes by more than an expected value.

Bounded Value Change error: \( \exists a_i \in A \mid v_i - v_{i+1} > C \) where \( C \) is the maximum expected value change between two consecutive actions.

(41) An Out Of Order [OutOfOrder] error occurs when a service delivers service items whose value changes by more than an expected value.

Out Of Order error: \( \exists a_i \in A \mid \delta_i = \delta_j \) for \( i \neq j \)

(42) A Stuck Value [StuckValue] error occurs when a service delivers service items whose value stays constant starting with a given service item.

Stuck Value error: \( \exists a_i \in A \mid \forall j > i: v_j = v_i \)

E.6.6 Replication Related Errors

(43) Replication related errors can occur, when replicates of a service item are delivered to the same or multiple users. Replication errors represent inconsistency in the replicates, where a subset of the replicates has a particular error.

(44) Replicates may be the result of fan-out from a single source, or they may represent independent observations of the same entity. A replication error may be introduced as result of inconsistent fan-out or as independent error occurring to individual replicates.

(45) We define a replicated service item as a set of actions that are supposed to be the same. In the case with n-way replication, let \( a_i = \{a_i(1), ..., a_i(n)\}, i = 1,2,... \) where \( V_i \) and \( D_i \) are respectively the correct sets of values and durations for action-set \( a_i \). Individual actions, \( a_i(k) \), have values \( v_i(k) \) and durations \( \delta_i(k) \). In a non-failed replication system we expect \( \forall k \in [1,n] \mid v_i(k) \in V_i \land \delta_i(k) \in D_i \).

(46) We distinguish between Inconsistent Value errors, Inconsistent Omission errors, and Inconsistent Timing errors.

(47) An Inconsistent Value [InconsistentValue] error occurs when the value of at least one replicated service items differs from the other replicates. The value of the replicate service item may be correct \( (v_i(k) \in V_i) \) or incorrect \( (v_i(k) \notin V_i) \).

Inconsistent Value error: \( \exists a_i \in A \mid \exists k \in [1,n] \mid v_i(k) \notin V_i \)

(48) We distinguish between inconsistent value errors and Inconsistent Approximate Value [InconsistentApproximateValue] errors. In the case of inconsistent exact value, the error occurs if the value comparison does not show identical values. In the case of inconsistent approximate value, the error occurs if the values in the comparison differ by more than a threshold. The threshold may be defined in terms of the difference, or with respect to a reference value. Let \( h \) be the threshold for approximate inconsistency.

Inconsistent Value error: \( \exists a_i \in A \mid \exists k \in [1,n] \mid v_i(k) > V_i + h \lor v_i(k) < V_i - h \)
An Inconsistent Omission [InconsistentOmission] error occurs when some replicates have an Item Omission or Service Omission error, while others do not. Such an error may be perceived as an inconsistent replicate value error. In two subtypes we distinguish between inconsistent item omission and inconsistent service omission.

\[ \exists a_i \in A \mid \exists k \in [1, n] \mid a_i(k) = \varepsilon \]

An Inconsistent Timing [InconsistentTiming] error occurs when at least one of the replicated service items is delivered outside the expected time interval. Let \( \Delta \) represent the maximum expected time variation of the replicates.

\[ \exists a_i \in A \mid \exists j, k \in [1, n], \delta_i(j) - \delta_i(k) > \Delta \]

These error types apply to the replicate set and can be combined with other error types into error type sets. For example, an asymmetric value error is represented by \( \{ \text{InconsistentValue} \} \) associated with a replicate set, while a symmetric or consistent value error is represented by \( \{ \text{ValueError} \} \) associated with a replicate set. Error type aliases have been defined to refer to the error types as symmetric and asymmetric replicates errors [CFEM 2003]. Replicate sets may be represented by a composite component, by a feature group, by a component or feature array, or by a property indicating the fact that a model element is to be replicated.

In a redundant system with active redundancy, a replicates consistency gate keeper, such as a voter or agreement protocol, may encounter all three of the above error types. In a system with a stand-by replica, an inconsistent value error may occur in that the stand-by replica has a value that is inconsistent with the primary value. This error is not propagated until the stand-by replica becomes active. Periodic exchange of state between the replicas is a common strategy to detect and correct this error.

### E.6.7 Concurrency-Related Errors

Concurrency-related errors [ConcurrencyError] are either race conditions [RaceCondition: ReadWriteRace WriteWriteRace], or mutual exclusion errors [MutExError: Deadlock Starvation].

Examples

-- This example is extending an existing error type with an additional subtype
package MyErrors
public
with ErrorLibrary;
annex EMV2 {**
error types extends ErrorLibrary with
  Jitter: type extends TimingError ;
end types;
**};
end MyErrors;
-- This example defines error types for use error propagation through ports
-- The namespace includes both the original error type names and the local ones
package PortErrors
public
with ErrorLibrary;
annex EMV2 {**
error types extends ErrorLibrary with
  NoData renames type ServiceOmission ;
}
ExtraData renames type ServiceCommission;
WrongValue: type extends IncorrectValue;
EstimatedValue: type extends IncorrectValue;
end types;
**};
end PortErrors;

-- This example defines error types for use in error propagation from processors
-- Only the locally declared error types are part of this error type library namespace
package ProcessorErrors
public
with ErrorLibrary;
annex EMV2 {**
error types
  NoResource renames type ErrorLibrary::ServiceOmission;
  NoDispatch: type extends NoResource;
  NoCycles: type extends NoResource;
  UnexpectedDispatch renames type ErrorLibrary::ServiceCommission;
  MissedDeadline renames type ErrorLibrary::LateDelivery;
  BadDispatchRate renames type ErrorLibrary::VariableRate;
end types;
**};
end ProcessorErrors;

Annex E.7 Error Propagation

(1) In this section we introduce the concept of error propagation. Error propagation occurs when a component fails and the failure can be observed by other components, i.e., an error is propagated from one component to another.

(2) For each component we specify the types of errors that are propagated through its features and bindings or are not to be propagated by the component. We also specify the role of the component in the flow of error propagations, i.e., whether it is the error source, error sink, or error path from incoming propagations to outgoing propagations. The propagation paths between components are determined by the core AADL model, i.e., they follow interactions between components through their features and along software to hardware component binding relations.

Syntax

error_propagations ::= 
  error_propagations 
  [ use types error_type_library_list ; ] 
  [ use behavior error_state_machine_reference ; ] 
  { error_propagation | contained_error }* 
  [ flows 
    { error_flow }* ] 
  [ properties


Naming Rules

(N1) The Error Model subclause introduces a namespace for identifiers Error Model entities defined within this subclause. This namespace follows the inheritance rules of its enclosing component type or implementation, e.g., the namespace of an Error Model subclause of a component type that extends another component type inherits the defining identifiers of that component type’s Error Model subclause.

(N2) The use types clause makes the defining identifiers of error types and type sets from the listed Error Type libraries referable without qualification. If multiple Error Type libraries provide the same defining identifiers, references to these identifiers must be disambiguated by a qualifying package name or through the local definition of aliases.

(N3) The use behavior clause makes the namespace of the Error Behavior State Machine accessible to the component error behavior specification.

(N4) The reference to an error behavior state machine in the use behavior clause must be qualified with the name of the package that contains the declaration of the error behavior state machine being referenced unless the error behavior state machine is located in the same package as the reference.

(N5) The reference to an error behavior state machine in the use behavior clause must exist in the namespace of the Error Model library in the specified package.

(N6) The emv2 annex specific containment path of an contained property association of an error propagation, contained error or observable error propagation must be an error_propagation_point reference, optionally followed by an error_type_reference, separated by a dot ("."). In the former case a property is associated with the error propagation, while in the latter case it is associated with a specific error type of an error propagation. This error type reference must be an error type included in the error type set associated with the error propagation.

(N7) The emv2 annex specific containment path of an error_flow_contained_property_association must be an error_propagation_reference, optionally followed by an error_type_reference, separated by a dot ("."). In the former case a property is associated with the flow, while in the latter case it is associated with a specific error type of an error flow. This error type reference must be an error type included in the error type set associated with the error flow.

Semantics

(3) An error propagations annex subclause consists of error propagation and contained error declarations for each of the features and the bindings of components as well as error flow declarations. The error propagation declarations specify the types of error being propagated in and out of features and bindings, while the contained error declarations specify that certain error types are not intended to be propagated. The types of errors being propagated or contained are expressed by error types or type sets. The error flow declarations indicate whether a component is the source or sink of an error propagation, or whether it passes error propagations on to other component, possibly transforming the error type to a different error type. These declarations for each component are combined with error propagation paths between instances of the components to determine an error propagation flow graph for a system architecture instance.

(4) The use types clause makes the namespaces of Error Type libraries accessible to the error propagations annex subclause. Error propagation declarations, contained error declarations, and error flow declarations can reference the error type or error type set by name without further qualification. If the same error type or type set name exists
in more than one imported Error Model library, then the reference must be qualified with the Error Model library alias.

(5) The use behavior clause makes the namespaces of an Error Behavior State Machine accessible to the error propagations annex subclause. Error source declarations may reference error behavior states of this Error Behavior State Machine by name without further qualification.

(6) Contained property associations allow the modeler to associate properties with error propagations and error flows. For convenience, the syntax for contained property association from AS5506B 11.3 is presented here.

E.7.1 Error Propagation and Contained Error Declarations

(7) An error propagation declaration specifies that errors of the specific types are propagated into or out of a component through the feature, binding, or observable propagation point. A contained error declarations allow the modeler to explicitly specify, which error types are expected not to be propagated.

Syntax

```
error_propagation ::= 
    error_propagation_point : 
    ( in | out ) propagation [ error_type_set ] ;

contained_error ::= 
    error_propagation_point : 
    not ( in | out ) propagation [ error_type_set ] ;

error_propagation_point ::= 
    feature_reference | binding_reference 
    | observable defining_observable_error_propagation_point_identifier

error_propagation_point_reference ::= 
    feature_reference | binding_reference 
    | observable observable_error_propagation_point_identifier

feature_reference ::= 
    feature_identifier | access

binding_reference ::= 
    processor | memory | bus | device | virtual bus | virtual processor 
    | binding | bindings
```

Naming Rules

(N1) References to an error type or error type set of an error_type_set statement in an error propagation declaration or contained error declaration must exist in the namespace of one of the Error Type libraries listed in the use types clause.
The defining identifier of an observable error propagation point identifier must be unique within the namespace of the subclause for which the observable error propagation point is defined.

The error_behavior_state reference declared as the fault source must exist in the namespace of the Error Behavior State Machine identified in the use behavior clause.

Legality Rules

The error type set specified by the error propagation declaration of a feature or binding reference must not intersect with the error type set specified by the contained error declaration for the same feature reference or binding reference.

The feature reference of error propagations for data components and bus components must be an access feature reference or the keyword access.

The binding reference of error propagations for software components must only contain the keyword processor, memory, bus, device, virtual bus, or virtual processor.

The binding reference of error propagations for virtual bus, virtual processor, and system components may include the keywords processor, memory, bus, device, virtual bus, virtual processor, binding, and bindings.

The binding reference of error propagations for processor, memory, bus, and device components must only contain the keyword bindings.

The direction of the error propagation must be consistent with the direction of the feature being referenced. For an incoming propagated error the feature must support incoming information flow. For an outgoing propagated error the feature must support outgoing information flow. Binding related propagations can occur in both directions.

For incoming features there must be at most one incoming error propagation declaration and at most one incoming contained error declaration. For outgoing features there must be at most one outgoing error propagation declaration and at most one outgoing contained error declaration. Binding error propagation points are considered both incoming and outgoing.

Consistency Rules

For a feature or binding every error type token must be contained in the error type set of an error propagation declaration or a contained error declaration. Any error type token that is not covered is referred to as unspecified error propagation type.

Semantics

An error propagation declaration specifies that errors of the specific types are propagated into or out of a component through the feature, binding, or observable propagation point. The type can be any error type of an error type hierarchy, or an explicitly specified subset of types in a type hierarchy. The error propagation can be error instances of one of the specified error types, or of combination of error types that occur simultaneously. Acceptable types are specified by an error type set.

A contained error declaration allows the modeler to explicitly specify, which error types are expected not to be propagated. When declared for an outgoing feature (or binding) it is an indication that the component intends to not propagate the error, i.e., mask it, if it occurs within the component or is propagated into it. When declared for an incoming feature it is an indication that the component does not expect an error of this type to be propagated to it.

Features may have incoming information flow, e.g., in ports and read-only data access, outgoing information flow, e.g., out ports and write-only data access, or bi-directional information flow, e.g., in out ports and read-write data access. Error propagations follow the same flow direction. Error propagation may also occur along bus access connections.

An error propagation declaration indicating direction in specifies expected incoming errors, and direction out specifies intended outgoing errors. A bi-directional feature can have the same or different incoming and outgoing
error types being propagated. This is specified by separately declaring the incoming and the outgoing error type for the feature.

(12) In the case of data or bus access connections the data component or bus component may be the source of a connection. In this case, the keyword *access* is used to identify the access point since no named access feature is specified.

(13) Errors can propagate between software components and execution platform components they are bound to. The keywords *processor*, *bus*, *virtual processor*, *virtual bus*, *memory*, and *device* are used to identify the binding point of a software component with the execution platform component it is bound to. The keyword *binding* is used for connections and virtual buses to identify their binding to execution platform components. Similarly, the keyword *bindings* is used in execution platform components to identify the binding point of components bound to them. Propagations with respect to bindings can be in both directions.

(14) An error propagation may occur between two components that do not have an explicitly declared error propagation path. For example, the temperature of one processor may affect a second processor that is located in close proximity, although the two processors are not connected via a bus. In other words, there is an error propagation path between two components that is being observed but is not expressed in the core AADL model, but along which error propagation occurs.

(15) Modeling is such observed error propagations is supported by the specification of observable error propagation points and by observable error propagation paths. Observable error propagation points include an error type set specification to indicate the types of errors being propagated. The observable error propagation points can be referenced in error flow declarations like any other error propagation point.

(16) Contained error declarations complement the error propagation declarations, such that a modeler can provide a complete record of the types of errors explicitly being addressed by the Error Model annotation for a component. This allows a consistency checking tool to determine whether an error of a given type is not intended to be propagated or whether the Error Model specification is incomplete and unspecified error types may be propagated.

![Figure 7 Error Propagations and Flows between Software and Hardware Components](image)

(17) Figure 7 shows two software components and two hardware components with error propagations along various error propagation paths between components (see Section E.7.3) and error flows through a component (see Section E.7.2). Examples of error propagation paths shown are a port connection, a bus access connection, and a processor binding. In the case of the port connection between component A and component B component A intends to propagate *BadData* errors and *NoData* errors and not propagate *LateData* errors, while component B
expects NoData errors, does not expect BadData errors, and is silent on LateData errors. In the case of the processor binding NoResource and MissedDeadline errors are shown as propagating to the software and ETOverrun error is shown as propagating overrun of execution time budget as software error to the processor. Examples of shown error flows for component A to be the source of BadData errors. It also shows an error flow from incoming NoData to outgoing NoData as well as an error flow mapping a processor NoResource error to a NoData error in the software component A.

E.7.2 Error Flow Declarations

(18) The purpose of error flow declarations is to indicate the role of a component in the propagation of errors in terms of being an error propagation source (error source), an error propagation sink (error sink), to pass-through incoming error propagations as outgoing errors of the same type, or to transform an incoming error of one type into an outgoing error of a different type (error path). For example, a component may be the source of bad data; a component may compensate for early arrival of data by delaying its delivery to others until the expected time, i.e., act as an error sink; a component may pass on the error of missing incoming data by not producing output (pass through of an error type on an error path); or a component may respond to incoming bad data by not producing output (transformation of one error type to another error type).

Syntax

error_flow ::= 
error_source | error_sink | error_path

error_source ::= 
defining_error_source_identifier :
    error source outgoing_error_propagation_point_reference 
    [ error_type_set ] [ when fault_source ] ;

error_sink ::= 
defining_error_sink_identifier :
    error sink incoming_error_propagation_point_reference [ error_type_set ] ;

error_path ::= 
defining_error_path_identifier :
    error path 
    incoming_error_propagation_point_reference [ error_type_set ] -> 
    outgoing_error_propagation_point_reference
    [ error_type_set | mapping type_mapping_set_reference ] ;

fault_source ::= error_behavior_state | error_type_set_reference

Naming Rules

(N1) The defining identifier of an error flow must be unique within the namespace of the Error Model annex subclause in which it is defined.
The first feature identifier of a feature_path in an error propagation point reference must exist in the namespace of the component type or implementation that contains the error propagations clause. Succeeding feature identifiers of a feature path must exist in the namespace of its predecessor feature group type.

The observable error propagation point identifier in an error propagation point reference must exist in the namespace of the Error Model subclause containing the error propagation point reference.

References to an error type or error type set of an error_type_set statement in an error flow declaration must exist in the namespace of one of the Error Type libraries listed in the use types clause.

The error_behavior_state reference declared as the fault source must exist in the namespace of the Error Behavior State Machine identified in the use behavior clause.

The package reference in a type_mapping_set_reference must be named in the with clause of the enclosing package declaration or must be referencing a package alias defined in the enclosing package declaration (see AS5506B Section 4.2).

The type mapping set identifier in a type_mapping_set_reference must exist in the Error Model library identified by the package reference.

Legality Rules

The feature reference of error flows for data components and bus components must be an access feature reference or the keyword access.

The binding reference of error flows for software components must only contain the keyword processor, memory, bus, device, virtual bus, or virtual processor.

The binding reference of error flows for virtual bus, virtual processor, and system components may include the keywords processor, memory, bus, device, virtual bus, virtual processor, binding, and bindings.

The binding reference of error flows for processor, memory, bus, and device components must only contain the keyword bindings.

The direction of the error propagation must be consistent with the direction of the error flow. An incoming error propagation must be the incoming propagation point of an error sink or error path. An outgoing error propagation must be the outgoing propagation point of an error source or error path. Binding related propagations can occur in both directions.

The error_type_set specified for an incoming error propagation point must be contained in the error type set of the incoming error propagation referenced by the flow.

The error_type_set specified for an outgoing error propagation point must be contained in the error type set of the outgoing error propagation referenced by the flow.

Semantics

By default a component is the source of all its outgoing error propagations and all incoming error propagations can potentially result in outgoing error propagations on all of its outgoing features or bindings, if no error flows are specified.

Error flows are intended to be an abstraction of the error flow represented by component error behavior specifications in terms of error behavior state machines, error and repair events, and conditions under which transitions and outgoing propagations are initiated. The error flows of a component must be consistent with its component error behavior specification.

Error flows may occur between features for which flow specifications exist in the component type, and between features for which no component flow is specified.
Multiple flows may interact. An outgoing error propagation of a component feature or binding can be an error flow source as well as the destination of an error flow path. For example, a component may produce bad data due to a fault in its source code or due to an incoming data value that is bad. Similarly, incoming bad data may be propagated as bad data, but in addition become late data due to processing delays within the component.

An incoming error propagation through a component feature or binding can be an error flow sink as well as an error flow path to an outgoing feature. For example, an error may occasionally get masked, or an error of one type gets masked while an error of another type becomes an outgoing propagation of the same or a different type.

We may have situations where more than one error type is being propagated at a given instant of time. This can be modeled by introducing error types, whose naming convention reflects the combination of types, e.g., `LateAndBad` as error type, or it can be explicitly modeled by an error type set that combines different error types, such as value errors and timing errors.

If an error flow declaration does not include the optional error type set, then the error type set specified for the referenced error propagation point applies. For flow sources this means that the component is the source of outgoing error propagation instances of error type tokens contained in the error type set of error propagation point. If the flow source includes an error type set the component is the source of error type tokens that are contained in the error type set specified for the error source. That error type set must be contained in the error type set specified for the error propagation. Note that outgoing error propagation on the same error propagation point may also come from an incoming error propagation as specified by an error path.

If an error sink does not include the optional error type set, the component is the sink for all incoming errors on the referenced error propagation point. If the error sink includes an error type set, then incoming error type tokens that are contained in this error type set are masked. This error type set must be a subset of the error type set specified for the incoming error propagation point. Note that an incoming error propagation on the same error propagation point may also be passed on as outgoing error propagation as specified by an error path.

If an error path does not include the optional error type set, then all incoming error type tokens are passed through as outgoing error propagations. If the error path includes an error type set, then the subset of error type tokens contained in this error type set are passed through. If a reference to a type mapping set is specified then the error type token of the outgoing error propagation is determined by mapping the error types of the incoming tokens according to this type mapping.

### E.7.3 Error Propagation Paths

Error propagation paths represent the flow of error propagations between components. Error propagation paths are determined by the connections between components, both application components and platform components, as well as by the binding of application components to platform components. In addition, error propagations may be observed between components for which there is no explicitly specified path in the AADL core model.

**Syntax**

```plaintext
observable_connections ::= 
  observable_connections 
  observable source_qualified_observable_error_propagation_point -> target_qualified_observable_error_propagation_point ;
```

**end connections;**
(subcomponent_identifier . )? observable_error_propagation_point_identifier

type_mapping_set_reference ::= package_reference :: type_mapping_set_identifier

Naming Rules

(N1) The defining identifier of an observable error propagation connection identifier must be unique within the namespace of the subclause for which the observable error propagation connection is defined.

(N2) The qualified observable error propagation point reference in an observable propagation connection declaration must exist in the Error Model subclause namespace of the component classifier of the qualifying subcomponent, if present, or in the namespace of the Error Model subclause containing the connection declaration.

Legality Rules

(L1) The direction of a source and a target observable error propagation point must be consistent with the connection as defined for port connections. For connections between two subcomponents the source must be outgoing and the destination must be incoming. For a connection from a subcomponent to the enclosing component the direction must be outgoing. For a connection from an enclosing component to a subcomponent the direction must be incoming.

(L2) The error type set of the outgoing error propagation must be contained in the error type set of the incoming error propagation.

(L3) The error type set of the incoming contained error declaration must be contained in the error type set of the outgoing contained error declaration.

(L4) The direction of the error propagation or contained error for the source must be outgoing and for the destination must be incoming.

Consistency Rules

(C1) The error type set for the error propagation source of an error propagation path must not intersect with the error type set of the destination contained error declaration or with or the set of unspecified error propagation types.

(C2) The set of unspecified error propagation types of an error propagation path source must not intersect with the error type set of the destination contained error declaration or the set of unspecified error propagation types.

(C3) The destination of an error propagation path is robust against unintended error propagations if the type set of its incoming error propagation declaration contains the error type set of the source error propagation, contained error declaration, and any unspecified error propagation type.

Semantics

(29) The following rules specify error propagation paths that are defined in a core AADL architecture model. Propagations may occur from

- a processor to every thread bound to that processor and vice versa
- a processor to every virtual processor bound to that processor and vice versa
- a processor to every connection bound to that processor and vice versa
- a virtual processor to every virtual processor bound to that virtual processor and vice versa
- a virtual processor to every thread bound to that virtual processor and vice versa
• a virtual processor to every connection bound to that virtual processor and vice versa
• a memory to every software component bound to that memory and vice versa
• a memory to every connection bound to that memory and vice versa
• a bus to every connection bound to that bus and vice versa
• a bus to every virtual bus bound to that bus and vice versa
• a virtual bus to every connection bound to that virtual bus and vice versa
• a device to every connection bound to that device and vice versa
• a component to each component it has an access connection to and vice versa, subject to read/write restrictions
• a component from any of its outgoing features through every connection to components having an incoming feature to which it connects
• a subprogram caller to every called subprogram (expressed by subprogram access connections or call bindings (and the opposite direction)
• a subcomponent to every other subcomponent of the same process (within a process there is no enforced address space boundary)
• a process (and contained thread group or thread) to every other process (and contained thread group or thread) that is bound to any common virtual processor, processor or memory, except for processes for which space and time partitioning is enforced on all shared resources
• a connection to every other connection that is routed through any shared bus, virtual bus, processor or memory, except for connections for which space and time partitioning is enforced on all shared resources
• an event connection to every mode transition that is labeled with an in event port that is a destination of that connection.

(30) Error propagation paths between observable error propagation points in the form of observable propagation connections are declared as part of the error model subclause of a component. These are propagation paths between observable error propagation points of two subcomponents, or between a subcomponent observable error propagation point and an observable error propagation point of the enclosing component.

(31) Error propagation and contained error declarations on outgoing features and bindings that are the source of an error propagation path must be consistent with those of incoming features of the target of an error propagation path. Figure 8 illustrates these consistency rules visually. The first rule shows that it is acceptable when a source indicates it does not intend to propagate an error of a certain type and the destination indicates it does not expect such an error type or the destination is silent regarding a known error type, i.e., it has not specified an error propagation or contained error for the given type. The second rule indicates that it is acceptable for the destination to indicate that it expects error of a given type, while the source indicates that it does not intend to propagate errors of the same type. The third rule indicates that it is acceptable for the destination to indicate that it expects error of a given type, and the source indicates that it propagates errors of the same type or nothing is specified for the given error type. The last two rules indicate inconsistent error propagation match up. The fourth rule indicates that it is not acceptable for the destination to indicate that it does not expect errors of a given type, while the source indicates that it intends to propagate such errors or is silent with respect to that error type. The fifth rule indicates that it is not acceptable for the destination or source to be silent on the propagation of a known error type or the source to indicate propagation and the destination be silent. See (L2) through (L4) and (C1) through (C3) of Section Annex E.7.3 for details of these consistency rules.
A component may specify that it intends to contain errors of a given type. This is under the assumption that the component is implemented according to specification. A component may unintentionally propagate an error although it was declared to be contained. A destination component is robust to such unintentional error propagations if it expects such error types to be propagated.

E.7.4 Error Propagation Properties and Analyses

The Error Model Annex includes a set of predeclared properties defined the property set EMV2. This section describes properties relevant to fault impact related analyses based on error propagation specifications.

Properties

One set of properties, FaultKind and Persistence, allow the user to further characterize the error propagations.

FaultKind : EMV2::FaultKindEnum
applies to (error event, error propagation, error type, type set, type token);
FaultKindEnum: type enumeration (Design, Operational);

The Persistence property allows the user to specify whether an error source, i.e., 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.

Persistence : EMV2::PersistenceEnum
applies to (all);
PersistenceEnum: type enumeration (Permanent, Transient, Singleton);

(38) The next property supports stochastic modeling and analysis.

(39) An occurrence probability property indicates the probability with which the entity occurs, with which the property is associated. For example, as a property associated with an error source and optionally an error type token it indicates the probability with which a component is an error source.

```
OccurrenceProbability : record (  
    ProbabilityValue : aadlreal;  
    Distribution : EMV2::Distribution;
  )
```

Distribution : type enumeration (Poisson, Fixed);

(40) The next set of properties support the characterization of components as sources of hazards. These properties are grouped into a record property called Hazard. A generic variant of the Hazard property is defined in the property set EMV2.

(41) The Hazard property can be associated with an error source, an error type in the error type set of an error source, the originating error behavior state (when) in an error source, an error type in the originating error type set (when) in an error source.

```
SeverityLevel : type aadlinteget;  
LikelihoodLevel : type enumeration (A, B, C, D, E);  
Hazard: record  
  (crossreference: aadlstring;  
    failure : aadlstring;  
    phase : aadlstring;  
    environment : aadlstring;  
    description : aadlstring;  
    severity : EMV2::SeverityLevel;  
    likelihood : EMV2::LikelihoodLevel;  
    verificationmethod : aadlstring;  
    risk : aadlstring;  
    comment : aadlstring;)  
```

applies to (error source, error propagation, error type, error behavior state);

(42) The crossreference property value allows for a cross reference into an external document. <Turn into a list for references to multiple documents?>

(43) The phase property value indicates the operational phase (mode) in which the hazard is relevant.

(44) The environment property value indicates the operational environment in which the hazard is relevant.

(45) The description property value allows for a textual description of the hazard.

(46) The severity property value indicates the severity of the hazard ranging from 1 (high) to 5 (low). MIL-STD 882D suggests descriptive labels (Catastrophic, Critical, Marginal, Negligible) as shown in Figure 9. ARP 4761 defines descriptive labels (Catastrophic, Hazardous, Major, Minor, NoEffect) as shown in Figure 10.

(47) A likelihood property value indicates the likelihood with which the hazard occurs. Likelihood is expressed in terms of levels ranging from A (high) to E (low). Each level typically has an associated probability of occurrence (p)
threshold. MIL-STD 882D suggests likelihood levels (Frequent: $p>10^{-1}$, Probable: $10^{-1}>p>10^{-2}$, Occasional: $10^{-2}>p>10^{-3}$, Remote: $10^{-3}>p>10^{-6}$, Improbable: $p<10^{-6}$) for probability of occurrence over the life of an item (see Figure 9). In the context of ARP 4761 or DO178 defines descriptive labels (Probable: $p>10^{-5}$, Remote: $10^{-5}>p>10^{-7}$, Extremely Remote: $10^{-7}>p>10^{-9}$, Extremely Improbable: $p<10^{-9}$) for probability of occurrence per operational hour (see Figure 10).

(48) A verification method property value provides a way of textually describing the verification method used to address the hazard.

(49) A risk property value provides a way of textually describing the potential risk of the hazard.

(50) MIL-STD-882 and ARP 4761 define separate sets of labels for different severity and criticality levels. According to (see Figure 10).

```plaintext
property set MILSTD882
  is
  SeverityLevel : type enumeration (Catastrophic, Critical, Marginal, Negligible);
  LikelihoodLevel : type enumeration (Frequent, Probable, Occasional, Remote, Improbable);

Hazard: record
  (crossreference: aadlstring;
   failure : aadlstring;
   phase : aadlstring;
   environment : aadlstring;
   description : aadlstring;
   severity : MILSTD882::SeverityLevel;
   likelihood : MILSTD882::LikelihoodLevel;
   verificationmethod : aadlstring;
   risk : aadlstring;
   comment : aadlstring;) applies to (all);

end MILSTD882;
```

**Figure 9 MIL STD 882D Variant of Hazard Severity and Likelihood**

```plaintext
property set ARP4761
  is
  SeverityLevel : type enumeration (Catastrophic, Hazardous, Major, Minor, NoEffect);
  LikelihoodLevel : type enumeration (Probable, Remote, Extremely Remote, Extremely Improbable);

Hazard: record
  (crossreference: aadlstring;
   failure : aadlstring;
   phase : aadlstring;
   environment : aadlstring;
   description : aadlstring;
   severity : ARP4761::SeverityLevel;
   likelihood : ARP4761::LikelihoodLevel;
   verificationmethod : aadlstring;
   risk : aadlstring;
   comment : aadlstring;) applies to (all);

end ARP4761;
```

**Figure 10 ARP 4761 Variant of Hazard Severity and Likelihood**
This section describes the use and interpretation of the occurrence probability property for stochastic analysis in an AADL model annotated with error propagations and error flows.

An occurrence probability property value can be associated with an error flow, error propagation, error behavior state, and error behavior event. It represents the probability with which the error flow, propagation or event occurs. An occurrence value can be associated with an error flow, propagation, or event by naming the respective model element as the target of a contained property association without including a specific error type. The occurrence property value can be associated with an error flow, error propagation, or error event of a specific error type by adding the error type after the model element, separated by a dot ("."). In this case it represents the probability with which an error flow, propagation, or event of the specified error type occurs. If the error type has error subtypes, then it represents the occurrence probability for any of the subtypes.

An occurrence probability property value can be associated with a flow source to indicate the probability of an error of any of the specified types occurring, or with a specific error type of a flow source to indicate the probability an error of a specific error type occurring.

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

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

The occurrence probability property value of an outgoing error propagation is determined by the probability of the component being the error source and the probability of an incoming error propagation being passed through or transformed into an outgoing error propagation. For example, in Figure 7 the outgoing error propagation NoData may originate from within the component A based on the occurrence probability of the error source declaration, and may pass on an incoming error propagation with a different probability. In the latter case, the occurrence probability is determined by the occurrence probability of the incoming error propagation and the probability of the incoming error propagation being passed through or transformed.

The occurrence probability 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 destination.

Examples

Figure 11 illustrates an example hazard specification. The Hazard property is associated with the error behavior state that is the error source. Such hazard specifications are characterized with severity and criticality. Catastrophic hazards will be included in a Functional Hazard Assessment (FHA) report. The remaining hazards remain in the model for safety analysis activities in later phases.
The error propagation and error flow annotations of an AADL model specify an error propagation graph. Fault impact analysis can be performed, e.g., in the form of a Failure Modes and Effects Analysis (FMEA), by tracing the propagation of error from error sources through the system. The automation of the fault impact analysis allows for the analysis to be repeated for alternative architecture designs as well as refinements of the architecture design. In this case, the origin of an error source, expressed by the `when` clause, represents the originating failure mode, and the respective outgoing propagations the effect on other components. When present error types are taken into account as different types of error sources and effects.

Annex E.8 Error Behavior State Machines

(1) In this section we introduce the concept of an error behavior state machine. An error behavior state machine consists of a set of states and transitions between the states. The trigger conditions for the transitions are expressed in terms of error events, recover events, and repair events as well as incoming propagated errors. An outgoing transition from a state in the error behavior state machine may branch to one of several target states, one of which is always selected with a specified probability. The resulting state can affect outgoing error propagations.

(2) The error behavior state machine can be defined as a typed token state transition system similar to a Colored Petri net, resulting in a more compact representation. This is accomplished by associating error types and type sets with error events and with states to specify acceptable types of tokens. The actual typed token of the state at any point in time is represented by an error type token. Section Annex E.8.3 elaborates on typed error behavior state machines.

(3) A component can show nominal behavior, represented by one or more working states, or it can show anomalous behavior, represented by one or more nonworking states. A component can transition from a working state to a nonworking state as result of an activated fault (error event) or due to the propagation of a failure (error propagation) from another component. Similarly, recover and repair events can transition the component from a
nonworking state to a working state. An error behavior state machine is a reusable specification that can be associated with one or more component type and implementation through a component error behavior subclause (Annex E.9) and a composite error behavior subclause (Annex E.10).

(4) Component error behavior subclauses allow the user to specify transition trigger conditions in terms of the error type token of a typed error behavior state, error behavior events, and incoming error propagations. In addition, these subclauses allow the user to specify conditions in terms of the current state and incoming error propagations under which outgoing error propagations occur, and errors are detected. Composite error behavior subclauses allow the user to specify conditions under which a composite error state is the current state – expressed in terms of error states of subcomponents.

Syntax

```
error_behavior_state_machine ::= error_behavior defining_state_machine_identifier
[ extends state_machine_reference ]
[ use types error_types_library_list ; ]
[ use transformations type_transformation_set_reference ; ]
[ events { error_behavior_event }* ]
[ states { error_behavior_state }* ]
[ transitions { error_behavior_transition }* ]
[ properties { error_behavior_state_machine_emv2_contained_property_association }* ]
end behavior ;
```

```
state_machine_reference ::= [error_model_library_package_reference :: ] error_behavior_state_machine_identifier
```

```
error_behavior_event ::= error_event | recover_event | repair_event
```

```
error_event ::= defining_error_behavior_event_identifier : error_event
[ error_type_set ]
[ when error_event_condition ] ;
```

Discussion: error_event_condition states what condition must be met for the event to trigger. Example: temp > Max_Temperature (Above_Range error type). This can be expressed by a property or by a constraint sublanguage, which references properties in the model. This event condition may be different for different error types in the error type set. The condition may be different for different components using the same error behavior state machine.

```
recover_event ::= defining_error_behavior_event_identifier : recover_event
[ recover_event_initiation ] ;
```

Discussion: recover_event_initiation allows the modeler to specify the event or mode transition in the AADL core model that initiates recovery.
repair_event ::=  
    defining_error_behavior_event_identifier : repair_event  
    [ repair_event_initiation ] ;  

Discussion: repair_event_initiation allows the modeler to identify the origin of the repair event in terms of the core AADL model, e.g., in terms of an event or mode transition.

error_behavior_state ::=  
    defining_error_behavior_state_identifier : [ initial ] state  
    [ error_type_set ] ;  

error_behavior_transition ::=  
    transition | branching_transition  

transition ::=  
    [ defining_error_transition_identifier : ]  
    error_transition_source ->  
    ( error_transition_target | error_transition_branch ) ;  

error_transition_source ::=  
    all | ( source_error_state_identifier [ source_error_type_set ] )  

error_transition_target ::=  
    mask | ( ( target_error_state_identifier [ target_error_type_token ] )  

error_transition_branch ::=  
    ( error_transition_target with branch_probability  
    { , error_transition_target with branch_probability }* )  

error_transition_condition ::=  
    error_transition_trigger  
    | ( error_transition_condition )  
    | error_transition_condition { and error_transition_condition }  
    | error_transition_condition { or error_transition_condition }  
    | numeric_literal ormore  
    ( error_transition_trigger { , error_transition_trigger }* )  
    | numeric_literal orless  
    ( error_transition_trigger { , error_transition_trigger }* )  

error_transition_trigger ::=
error_behavior_event_identifier [error_type_set ]
| incoming_error_propagation_point [ error_type_set ]
| incoming_error_propagation_point { noerror }

branch_probability ::= 
   fixed_probability_value | others

fixed_probability_value ::= 
   real_literal |
   ( [ package_identifier :: ] real_constant_identifier )

discussion: the symbolic identifier was used in V1. We could also allow property constant or property references.

need to allow constants defined in separate package.

Naming Rules

(N1) The defining identifier of an error behavior state machine must be unique within the namespace of the Error Model library, i.e., must not conflict with defining identifiers of other error behavior state machines, of error type, type sets, type mapping sets, and type transformation sets.

(N2) The error behavior state machine represents a namespace for error behavior events, error behavior state, and error behavior transitions. Their defining identifier must be unique within the namespace of the error behavior state machine.

(N3) If the error behavior state machine extends another error behavior state machine, then the defining identifiers in that namespace are inherited. In other words, locally defined transitions, states, and events must be unique with respect to both locally declared defining identifiers and inherited defining identifiers.

(N4) The reference to an error behavior state machine must be qualified with the package name of the Error Model library that contains the declaration of the error behavior state machine being referenced unless the referenced error behavior state machine is declared in the same Error Model library as the reference.

(N5) The package name of a qualified error behavior state machine reference or a type transformation set reference must be named in the with clause of the enclosing package declaration or must be referencing a package alias defined in the enclosing package declaration (see AS-5506B Section 4.2).

(N6) The use types clause makes the defining identifiers of error types and type sets from the listed Error Type libraries referable within the error behavior state machine declaration. If multiple Error Type libraries provide the same defining identifiers, references to these identifiers must be disambiguated by a qualifying package name or an alias declaration (see also Section Annex E.7 Naming Rule(N2)).

(N7) References to an error type or error type set of an error_type_set statement in an error event declaration or error behavior state declaration must exist in the namespace of one of the Error Type libraries listed in the use types clause.

(N8) References to an error type or error type set of an error_type_set statement in an transition declaration must exist in the namespace of one of the Error Type libraries listed in the use types clause.

(N9) The source state reference and target state reference must identify a defining state identifier in the namespace of the error behavior state machine containing the reference.
The behavior event reference of a trigger event must identify a defining error event, recover event, or repair event in the namespace of the error state machine containing the reference.

The type transformation set reference in a use transformations statement must exist in the namespace of the Error Model library containing the reference or in the Error Model library identified by the qualifying package name.

The emv2_annex_specific_path of an emv2_contained_property_association in an error behavior state machine properties section must consist of reference to an error event, recover event, repair event, error state, or error state transition identifier that is defined in the namespace of the Error Behavior State Machine. For error events and error states this reference may optionally be followed by an error_type_reference, separated by a dot ("."). This error type reference must be an error type included in the error type set associated with the error event or error state.

Legality Rules

(L1) The optional error_type_set of a transition source state or transition target state must be contained in the error_type_set declared with the referenced state.

(L2) The optional error_type_set of a transition condition element must be contained in the error_type_set declared for the referenced error event or incoming error propagation.

(L3) The probabilities of the outgoing branch transitions must add up to 1, or be less than one if one branch transition is labeled with others.

(L4) The error_transition_trigger of a transition must only refer to error behavior events, when the error behavior transition is declared as part of the error behavior state machine.

(L5) The error_transition_trigger of a transition must only refer to error behavior events and incoming error propagation points, when the error behavior transition is declared as component specific transition in a component error behavior specification.

(L6) The optional error_type_set of a source error state must be contained in the error type set specified with the defining state declaration.

(L7) The optional error_type_set of a target error state must be contained in the error type set specified with the defining state declaration.

(L8) The optional error_type_set of a transition condition element in a component specific transition condition expression must be contained in the error type set specified with the defining error event or incoming error propagation declaration.

(L9) The logical and operator takes precedence over the logical or operator. The orless, and ormore constructs represent logical primitives and take precedence over the logical operators.

Semantics

(5) An error behavior state machine declaration consists of a specification of error, recover, and repair events, and of a specification of error behavior states and transitions. An error behavior state machine can be declared in terms of another error behavior state machine using the extends mechanism.

(6) An error behavior state machine specification can be reused by associating it with components in component error behavior specifications (Annex E.9) and composite error behavior specifications (Annex E.10). The component error behavior specifications conditions under which incoming propagated errors are masked or trigger a transition, and specify which error behavior states are observable by other components as outgoing propagated errors.

(7) An error behavior state machine can be defined as a typed token state transition system by associating error types and type sets with error events and state. This leads to a more compact error behavior specification.
E.8.1 Error, Recover, and Repair Events

(8) The Error Model Annex distinguishes between three kinds of error behavior events: error events, recover events, and repair events. Error events represent fault activation within a component, which can cause the component to transition from a working state to a nonworking state. Occurrence of an error event can result in a state transition to a nonworking state and in an outgoing error propagation. Recover events are used to represent recovery from a nonworking state to a working state. This is used to model recovery from transient errors. Repair events are used to represent model longer duration repair action on the actual system, whose completion results in a transition back to a working state.

(9) Separately declared error, recover, and repair events are considered to occur independently. Simultaneously occurring events may be handled in non-deterministic order. For example, the declaration of an error event representing out of range values and a separate error event representing late delivery

(10) An error event is identified by the name of the error type that identifies the activated fault. An error event may be named in a transition indicating that its occurrence will trigger the transition.

(11) An error event may be annotated with the system condition that results in the activation of the fault. This condition is specific to the component and may be expressed in terms of properties of the component and its features.

(12) An occurrence probability can be associated with error behavior events. It can be 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 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 (subcomponent) of the classifier. Finally, a subcomponent-specific value can be assigned by declaring it in the error model subclause properties section of an enclosing component implementation or in the core AADL model using a contained property association with an annex-specific fragment of the containment path (see AS5506B Section 11.3).

(13) For error events that have been declared with an error type set, occurrence probabilities can be specified for specific error types in the type set. It represents the probability with which an error event of that type can occur. If it is specified for an error type that represents a type hierarchy, i.e., has subtypes, then it represents the probability of an error type token of the specified type, i.e., the probability with which any of the subtypes can occur without an explicit probability allocation to each individual type unless a separate occurrence probability is assigned to each of the subtypes.

(14) If the error type set of an error event includes both single error types and error type products, then the occurrence probability value assigned to an error type represents the probability that the error type occurs either as a single valued type token or as part of a token instance of a type product. In this case, the occurrence probability for a single valued type token and for type token representing a product can be inferred from the specified probabilities.

(15) A recover event may be used to model transient failure behavior of a component in that it represents the trigger to return from a nonworking state to a working state. A Duration property indicates a distribution over a time range as the length of time the component transiently stays in a non-working state.

(16) A repair event represents a repair action. In some modeling scenarios it may be sufficient to represent the completion of a repair action as a repair event, while in other modeling scenarios it is useful to distinguish between the initiation of the repair action and the completion.

(17) A duration property and an occurrence property characterize the repair event. A Duration property indicates a time range reflecting the duration of a repair as well as the distribution over the duration time range. An OccurrenceProbability property is used to indicate when a repair is initiated. This property value takes into account the role of a “repairman”, i.e., the resources required to perform such a repair.

(18) Recovery or repair may succeed or fail. This is represented by a branch transition that is triggered by a recover or repair event and has two branches, one for successful recovery or repair and one for recovery or repair failure. The probability specified for each branch indicates the probability of success or failure.
Properties

(19) Properties described in Section E.7.4 are also applicable to elements of an error behavior state machine.

(20) When applied to error behavior events, i.e., Error Events, Recover Events, or Repair Events, the OccurrenceProbability property specifies a probability according to a specified distribution according to which error behavior events are expected to occur.

(21) When applied to error events, the OccurrenceProbability property can be applied to specific error types.

(22) When applied to a recover event, the OccurrenceProbability property specifies the probability with which recovery is initiated.

(23) When applied to a repair event, the OccurrenceProbability property specifies when a repair is initiated. The value of this property takes into account the role of a “repairman”, i.e., the resources required to perform such a repair. The actual value may be a computed value (compute) to take into account the availability of the “repairman”.

DurationType : type record (  
  TimeInterval : Time_Range; Distribution : EMV2::DistributionFunction;);  
Duration : EMV2::Duration applies to (repair event, recover event);

(24) The Duration property specifies a time range to reflect the duration as a distribution over the time range. This property can be attached to repair events to indicate the duration of the repair, once started. When applied to a recover event, it represents the duration of the recovery, or a repair event to represent to duration of the repair.

E.8.2 Error Behavior States and Transitions

(25) An error behavior state machine consists of a set of error behavior states and transitions between them. Transitions can be triggered by error events, repair events, and incoming error propagations.

(26) An error behavior state can be marked as working state, or nonworking state through the StateKind property. A working state indicates that the component is operational, while a nonworking state indicates that the state represents a failure state. A component can have one or more working states and one or more non-working states.

(27) Error behavior events and incoming error propagations of an error behavior state machine can trigger transitions to a new error behavior state. Transitions can be declared as part of the error behavior state machine declaration in terms of error behavior events, or as component specific transitions in terms of incoming error propagation points of the component as well as error behavior events.

(28) An error behavior transition specifies a transition from a source state to a target state if a transition condition is satisfied. The keyword all may be used that the transition applies to all source states.

(29) A transition can be a branching transition with multiple target states. One branch may be declared as masking error behavior events. Once the transition is taken, one of the specified target states is selected or masking occurs according to a specified probability. The probabilities of all branches must add up to one. One of the branches may specify others – taking on a probability value that is the difference between the probability value sum of the other branches and the value one.

(30) An example use of a branching transition is that an error event may trigger a transition with two branches, one to a target state representing a permanent error and the other target state representing a transient error. Failure in a recover or repair action can be modeled in a similar fashion by one branch representing a successful recovery or repair and the other representing recovery or repair failure.

(31) The transition condition expression of an error behavior transition declaration can specify one or more alternative conditions, one of which must be satisfied in order for the transition to be triggered. Multiple error behavior transition declarations may name the same source and target state. In this case the transition condition expression of each transition declaration is considered to be an alternative transition condition.
An alternative transition condition specifies all the error behavior events and error propagations that must be present in order for the condition to hold (conjunction). Any error propagation point not specified must not have an error propagation present. For example, assume a component with two incoming ports \texttt{port1} and \texttt{port2}.

- If an alternative transition condition specifies a single error propagation point, e.g., \texttt{port1\{BadValue\}}, by itself, then all other incoming error propagation points must not have a propagation present. If the alternative transition condition specifies \texttt{port1\{BadValue\} and port2\{BadValue\}}, then the condition is satisfied if error propagations are present on both ports.

- If each port is referenced by itself in a separate alternative transition condition, i.e., \texttt{port1\{BadValue\} or port2\{BadValue\}}, then the transition condition is satisfied if \texttt{port1} has an error propagation present and \texttt{port2} does not have an error propagation present, and vice versa, but is not satisfied when both ports have an error propagation present (exclusive or of alternatives).

- If the alternative transition condition specifies \texttt{1 ormore (port1\{BadValue\},port2\{BadValue\})}, then the condition is satisfied if error propagations are present on either port or on both ports.

Note: we chose to interpret listing a single error propagation point as all others being error free, because modelers often assume that they are dealing with one incoming error propagation at a time. Alternatively we could have required the modeler to explicitly indicate that the other error propagation point have \texttt{NoError}.

Separately declared error behavior events may occur separately or simultaneously. An error behavior specification may specify one error event or combinations of error events as a transition condition.

Simultaneous occurrence of errors of more than one type can be modeled by a typed error event with an error type set of more than one element type. For example, an error event declared with \{\texttt{BadError},\texttt{LateError}\} represents the occurrence of \texttt{BadValue} errors, of \texttt{LateValue} error, and error events representing \texttt{BadValue} and \texttt{LateValue} occurring simultaneously.

The set of outgoing error behavior transition from the same source error behavior state to different target states must be unambiguous for a given component, i.e., they must uniquely identify the target state for a given state, error behavior events, and incoming error propagations. The consistency rules expressing this can be found in Section Annex E.9.

A transition can be declared as masking error behavior events in a given state, expressed by the keyword \texttt{mask} instead of a target state. In this case the source error state remains the current state and, if typed, its error type token remains the same. Specifying conditions under which the error behavior state of a component is affected or not affected by incoming error propagations allows us to check for full coverage of incoming propagated errors as well as for consistency with error propagation declarations, contained error declarations, and error flow specifications for the component. The consistency rules expressing this can be found in Section Annex E.9.

Properties

\begin{verbatim}
StateKind : EMV2::StateKindEnum
    applies to (error behavior state);

StateKindEnum: type enumeration (Working, NonWorking);
\end{verbatim}

The \texttt{StateKind} property specifies whether an error behavior state is considered to be a working state or a non-working state. A component can have multiple error behavior states that are tagged as working states.

Example

\begin{verbatim}
package RecoverErrorModelLibrary
public
annex EMV2 {**
error behavior Example
    events
\end{verbatim}
SelfCheckedFault: error event;
UncoveredFault: error event;
SelfRepair: recover event;
Fix: repair event;

states
Operational: initial state;
FailStopped: state;
FailTransient: state;
FailUnknown: state;

transitions
SelfFail: Operational -[SelfCheckedFault]->
   (FailStopped with 0.7, FailTransient with 0.3);
Recovery: FailTransient -[SelfRepair]-> Operational;
UncoveredFail: Operational -[UncoveredFault]-> FailUnknown;

end behavior;
**};
end RecoverErrorModelLibrary;

E.8.3 Typed Error Behavior State Machines

(39) A typed error behavior state machine represents a typed token state transition system with error type tokens.

(40) An error event may be declared with an error type that represents a type hierarchy. In this case an instance of the error event will be of one of the types in the type hierarchy. If the error event has been specified with an error type set, then an instance of an error event will have a type that is a token of the element types.

(41) An error behavior state may be declared with an error type or type set. When an error behavior state machine has a typed state as its current state, then its current typed token is represented by an error type token that is contained in the specified type set.

(42) The set of error type tokens making up the error type set of a state can be viewed as sub-states. A transition into a typed state with a given error type token effectively is a transition into the respective sub-state. While in a typed state, error events or incoming propagations can trigger a transition to a different state or a change of the error type token for the current state, effectively transitioning between the sub-state representing the original error type token and the sub-state representing the new error type token.

(43) A transition out of a source error state can optionally be constrained by declaring an error type set on the error state. This constraint determines for which error type tokens of the current state the transition applies.

(44) The optional constraint on an error event reference in transition condition expression determines which error type tokens of the error event trigger the transition.

(45) The optional constraint on an error propagation point reference in transition condition expression determines for which error type tokens of the incoming error propagation affects the transition.

(46) The error type token of a typed target state of a transition is determined as follows:

- If the target state of a transition has a target error type token declared, then it represents the target state error type token.
• If the target state of a transition does not have a target error type token declared, then type transformation rules associated with the error behavior state machine are used to determine the target error type token.

• If no target type token or type transformation rules are specified then default rules apply.

(47) Type transformation rules determine the error type token of the target state by matching the source error type token with the source element of the transformation rules and the contributor error type token with the contributor element of the transformation rules (see Section Annex E.11 for details). In the case of multiple contributors, e.g., a conjunction in a transition trigger condition, the transformation rule is applied repeatedly in the order of the conjunction elements.

(48) The default rule to determine the target error type token works as follows:

(49) If the source state of the transition is not typed, then the error type token of the target state is that of the triggering error event if there is one triggering error event that is typed, or there is one incoming error propagation as condition element.

(50) If the source state is typed and the error event is not typed, then the target error type token is that of the source state.

(51) If neither the source state nor the error event is typed, then the target error type token must be explicitly specified in the transition declaration.

(52) If the source state is typed and the contributor is typed, then the target error type token consists of union of type token elements of the source and the contributor. If both have an element of the same type hierarchy the element of the contributor takes precedence. In other words, the target error type token \( \text{tot} \) is determined from the source error type token \( \text{tos} \) and the contributor error type token \( \text{toc} \) as follows: \( \forall i: \text{tot}_i \in \text{tot} \mid \text{tot}_i \in \text{toc} \lor (\text{tot}_i \in \text{tos} \land (\forall \text{tock} \in \text{toc} \mid r(\text{tot}_i) \neq r(\text{tock}))) \). For multiple typed contributors the rule is applied to each in order.

(53) Otherwise, the type token for the target is undefined.

(54) The example below shows a typed error behavior state machine for which the default rule applies to determine the error type token of the target state.

```java
package TypedErrorModelLibrary
public
annex EMV2 {**
error types
  MyFault: type;
  DetectedFault: type extends MyFault;
  BITFault: type extends DetectedFault;
  StuckBit: type extends BITFault;
  BadBlock: type extends BITFault;
  NoValueFault: type extends DetectedFault;
  NoServiceFault: type extends DetectedFault;
  UndetectedFault: type extends MyFault;
end types;
error behavior Example
use types TypedErrorModelLibrary;
  events
```

Example
SelfCheckedFault: error event {DetectedFault};
UncoveredFault: error event {UndetectedFault};
SelfRepair: recover event;

states
Operational: initial state;
FailStopped: state {DetectedFault};
FailTransient: state {DetectedFault};
FailUnknown: state {UndetectedFault};

transitions
SelfFail: Operational -> FailStopped
    (FailStopped with 0.7, FailTransient with 0.3);
Recovery: FailTransient -> Operational;
UncoveredFail: Operational -> FailUnknown;

end behavior;
**
end TypedErrorModelLibrary;

Annex E.9 Component Error Behavior Specification

(1) A component can have two error behavior specifications: a component error behavior specification of the component as a “black-box” abstraction, and a composite error behavior specification of the component in terms of error states of its subcomponents. This section focuses on component error behavior specifications, while section Annex E.10 defines composite error behavior specifications.

(1) A component error behavior specification associates an error behavior state machine with a component type or component implementation (representing a component variant) and allows for component specific refinement of the component error behavior. The refinement consists of specification of transitions in terms of incoming error propagations and the source error behavior state, the specification of conditions under which outgoing error propagations occur in terms of incoming error propagations and the target error behavior state.

(2) A system cannot diagnose, recover from, or repair a failure until it is detected. The declaration of error detections as part of the component error behavior specification allows the modeler to declare the condition being detected in terms of error behavior state and incoming error propagations. It also allows the modeler the way that the system reports a component failure as event or event data via ports.

(3) A component error behavior specification associates properties values that are component specific and possibly specific to operational modes of the component with error events, recover events, repair events, and error propagations.

Syntax

cOMPONENT ERROR BEHAVIOR ::= 

COMPONENT ERROR BEHAVIOR 
[ use types error_types_library_list ; ]
use behavior error_state_machine_reference ;
[ use transformations type_transformation_set_reference ; ]
[ events { error_behavior_event } ]
[ transitions { component_specific_error_transition } ]
[ propagations { outgoing_propagation } ]
[ detections { error_detection } ]
[ mode mappings { error_state_to_mode_mapping } ]
[ properties { error_behavior_emv2_contained_property_association } ]
end component;

error_state_machine_reference ::= [ package_identifier :: ] error_state_machine_identifier

outgoing_propagation ::= 
[ defining_outgoing_propagation_identifier : ]
error_transition_source
-[[ [ error_transition_condition ] ]]->
propagation_target ;

propagation_target ::= 
  error_propagation_point_identifier [ propagated_error_type_token ]

error_detection ::= 
[ defining_error_detection_identifier : ]
error_transition_source
-[[ [ error_transition_condition ] ]]->
error_detection_effect ;

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

-- Numeric literal and property constant term are defined in AS5506B
error_code_value ::= integer_literal | enumeration_identifier | property_constant_term

error_state_to_mode_mapping ::= 
  error_behavior_state_identifier in modes ( mode_name { , mode_name } )

Naming Rules

(N1) The use behavior clause makes the namespace of the Error Behavior State Machine accessible to the namespace of the component error behavior specification.

(N2) Defining identifiers of component specific error events, recover events, repair events, error behavior transitions, outgoing error propagation conditions, and error detections must be unique within the namespace of the Error Model annex subclause that contains the component error behavior specification, i.e., they must not conflict with defining identifiers from the error propagation section or from the Error Behavior State Machine made accessible by the use behavior clause.
The reference to an error behavior state machine in the use behavior clause must be qualified with the name of the package that contains the declaration of the error behavior state machine being referenced unless the error behavior state machine is located in the same package as the reference.

The reference to an error behavior state machine in the use behavior clause must exist in the namespace of the Error Model library.

The package name of a qualified error behavior state machine reference or type transformation set reference must be named in the with clause of the enclosing package.

The use types clause makes the defining identifiers of error types and type sets from the listed Error Model libraries referable within the error behavior state machine declaration. If multiple Error Model libraries provide the same defining identifiers, references to these identifiers must be disambiguated by a qualifying package name or an alias declaration (see also Section Annex E.7 Naming Rule(N2)).

References to an error type or error type set of an error_type_set statement in an error event declaration must exist in the namespace of one of the Error Type libraries listed in the use types clause.

The behavior event reference of a trigger event must identify a defining error event, recover event, or repair event in the namespace of the error state machine containing the reference or in the name space of the component error behavior specification.

References to an error type or error type set of an error_type_set statement in component specific transition, outgoing propagation, or error detection declarations must exist in the namespace of one of the Error Model libraries listed in the use types clause.

References to an error type in an error_type_set statement in component specific transition, outgoing propagation, or error detection declarations must exist in the namespace of one of the Error Model libraries listed in the use types clause.

The identifier of an error behavior state reference in a component specific transition, outgoing propagation, error detection or error state to mode mapping declaration must exist in the namespace of the error behavior state machine identified in the use behavior clause.

The identifier of an error propagation point reference of an outgoing propagation declaration must exist in the error propagations clause of the component that contains the component error behavior specification.

The port identifier in a detection event must exist in the namespace of the component containing the component error behavior specification.

The type transformation set reference must exist in the namespace of the Error Model library containing the reference or in the Error Model library identified by the qualifying package name.

The mode name in an error state to mode mapping declaration must exist in the namespace of the component type or component implementation that contains the component error behavior specification.

The enumeration identifier referenced as error code value must exist as enumeration literal in the data type of the port.

The emv2_annex_specific_path of an emv2_contained_property_association in a component error behavior properties section must consist of reference to an error event, recover event, repair event, error state, error state transition, outgoing propagation, or error detection identifier that is defined or accessible in the namespace of the component error behavior specification. For error events and error states this reference may optionally be followed by an error_type_reference, separated by a dot ("."). This error type reference must be an error type included in the error type set associated with the error event or error state.

Legality Rules

The use transformations statement may override any type transition set specified in the error behavior state machine declaration.
(L2) The optional error_type_set of an error behavior state reference in a component specific transition, outgoing propagation, or error detection declaration must be contained in the error_type_set declared for the referenced state.

(L3) The optional error_type_set of an outgoing error propagation point in a outgoing propagation declaration must be contained in the error type set specified with the defining error propagation point declaration.

(L4) The optional error_type_set of a propagation condition element in a propagation condition expression must be contained in the error type set specified with the defining incoming error propagation declaration.

(L5) The optional error_type_set of a detection condition element in a detection condition expression must be contained in the error type set specified with the defining incoming error propagation declaration.

Consistency Rules

(C1) If an error event occurs in a given error behavior state and no outgoing transition, including masking transitions, names the error event then the error event is considered unhandled (erroneous).

(C2) If a recover or repair event occurs in a given error behavior state and no outgoing transition, including masking transitions, names the event then the error event is considered unhandled (erroneous).

(C3) For each transition there must be at most one transition condition specification with the same resulting token. For each transition there must be at most one transition condition specification, if no resulting token is specified.

(C4) The result of evaluating the outgoing propagation declarations for an outgoing error propagation must be at most one propagated error type token.

(C5) The result of evaluating the outgoing propagation declarations for an outgoing error propagation must not result in both a propagated error type token and masking of the outgoing error propagation.

(C6) The result of evaluating the error detection declarations must be at most one detected error type token.

(C7) The resulting error type tokens of masking outgoing propagation declarations must not be contained in the error type set of the referenced feature or binding.

(C8) If a component type or component implementation has modes, then each mode must be specified in at least one state to mode mapping.

Semantics

(4) A component error behavior specification associates an error behavior state machine with a component type or component implementation. In addition it specifies conditions under which error events, recover events, repair events, and incoming error propagations trigger an error behavior transition, conditions under which an error behavior state and incoming error propagations result in outgoing error propagations at various error propagation points. It also specifies conditions under which a failure, i.e., error behavior state or incoming error propagation, is actually detected by a system component and reported as an event or event data (message).

E.9.1 Outgoing Error Propagation Conditions

(5) The component error behavior specification allows the user to specify under what conditions an error propagation occurs on an outgoing error propagation point, and under what conditions an error is masked for all error propagation points or for specific an error propagation point. The conditions are expressed in terms of error behavior states and in terms of the presence or absence of propagated errors on incoming error propagation points.

(6) An outgoing propagation declaration can specify that an outgoing error propagation is triggered solely by an error behavior state. In this case, a component failure reflected in the error behavior state becomes observable to other components via the error propagation.
The error behavior state referenced in an outgoing propagation declaration is the new (target) state of a transformation, if a transformation occurs; otherwise it is the current state. This allows the modeler to specify the outgoing transition just in terms of the (new) state without having to repeat the condition under which the new state is reached.

An outgoing propagation declaration can also specify that an outgoing error propagation is triggered when the component is in a particular error behavior state and incoming propagations occur. For example, a component may propagate incoming bad data as bad data only if it is in a working state, while in a nonworking state it may always propagate an omission error type of any incoming propagation. The keyword all instead of the error behavior state indicates that it applies to all states, i.e., the outgoing propagation is solely determined by incoming propagations.

An outgoing propagation declaration can specify error type token constraints on the state and on the incoming propagations.

An outgoing propagation declaration can also explicitly specify the error type token to be propagated. If this propagated error type token is not specified, type transformation rules or default rules are used to determine the propagated error type token – the same way the target error type token of a transformation is determined.

An outgoing propagation declaration can specify under which conditions incoming error propagations and the current error state do not result in an outgoing propagation on any or a specific outgoing error propagation point. This is specified with the keyword mask. Specifying conditions, under which the outgoing propagations occur and do not occur, allows us to check for full coverage of incoming propagated errors and error states with respect to outgoing error propagation declarations, contained error declarations, and error flow specifications for the component.

E.9.2 Error Detections

The component error behavior specification supports the declaration of conditions under which a failure is actually detected by a system component. This can be a failure of the component itself, which is reflected in the error behavior state, or the failure of another component, which is reflected as an incoming error propagation. These conditions are expressed as logical expressions in terms of the component error state and incoming error propagations.

An error detection declaration can specify that an error is detected in terms of an error behavior state alone. In this case, transition into the specified error behavior state is detected and reported with an appropriate error code. In other words, the specified error behavior state is the new (target) state of a transformation. This allows the modeler to specify the error detection just in terms of the (new) state without having to repeat the condition under which the new state is reached.

An error detection declaration can also specify that an error detection is triggered when the component is in a particular error behavior state and incoming propagations occur. For example, a component may detect that its own function produces bad data or bad data is propagated in only if it is in a working state, while in a nonworking state it may always propagate out an omission error type, which then must be detected by the recipient. The keyword all instead of the error behavior state indicates that it applies to all states, i.e., detection is solely base on incoming propagations.

An error detection declaration can specify error type token constraints on the state and on the incoming propagations.

An error detection declaration can also explicitly specify the error type token representing the failure that has been detected. If this detected error type token is not specified, type transformation rules or default rules are used to determine the detected error type token – the same way the target error type token of a transformation is determined.

An error detection declaration identifies the mechanism used to detect the error at runtime. This is accomplished by a Detection_Mechanism property associated with the error detection declaration.

An error detection is reported by a component through an event port or event data port. In the case of an event data port the data value sent is the value of the error_detection_effect.
The error detection may also be made known in the core model of the system as an event or data event within the component using the `self` clause.

### E.9.3 Operational Modes and Failure Modes

A component may have operational modes and mode transitions. 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 only be initiated between modes specified in the mode mapping of a given error behavior state. When a transition occurs between error behavior states and the new error behavior state does not include the current mode in its mode mapping, then a forced transition occurs to the first mode listed in the mode mapping.

Figure 12 shows a component on the left that consists of two sensors and a processing unit. It can operate in high precision mode (HiP) using two sensors, low precision mode (LoP) using one sensor, or it can be turned off (Off). The user of the component can initiate a change in operational mode by command (shown as incoming event that triggers the appropriate mode transition. Color coding shows that in LoP mode Sensor1 and Processing are active (green background), in HiP mode both sensors and Processing are active (dashed outline), and in Off mode none of the components are active.

Figure 12 shows an error model for the component on the right. This error model shows three error behavior states: Operational when everything is in working condition; Degraded when one sensor has failed, and FailStop when two sensor have failed or processing has failed. The error model also indicates that the component can recover from a single sensor failure through a reset operation. Finally, the error model indicates that in Degraded error behavior state a value error may be propagated, while in the FailStop an omission error is propagated.

The following specifies a mapping of error behavior states onto modes:

Operational in modes (HiP, LoP, Off);
Degraded in modes (LoP, Off);
FailStop in modes (Off);

By doing so we superimpose the error behavior states onto the mode state machine as shown in Figure 13. 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.

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 the component is in HiP mode and an sensor fails error event causes a transition the degraded error behavior state, then a forced mode transition occurs to the LoP mode (shown as dashed arrow). Similarly, a second sensor fails or a processing fails error event forces a mode transition to the Off mode. The figure is also showing that while in degraded mode, event initiated mode transitions can only occur between modes that are part of its mode mapping.
In this section we introduce the concept of composite error behavior of an individual component. The composite error behavior of a component is its error behavior state expressed in terms of error behavior states of its subcomponents.

Syntax

composite_error_behavior ::= composite_error_behavior
   [ use types error_types_library_list ; ]
   use behavior error_state_machine_reference ;
   states { composite_error_state }*
   [ properties { composite_error_env2_contained_property_association }* ]
end composite;

composite_error_state ::= [ subcomponent_state_expression ]->
   composite_state_identifier [ result_error_type_token ] ;

subcomponent_state_expression ::= subcomponent_state_element
   | ( subcomponent_state_expression )
   | subcomponent_state_expression { and subcomponent_state_expression }*
   | subcomponent_state_expression { or subcomponent_state_expression }*
   | numeric_literal ormore
      ( subcomponent_state_element { , subcomponent_state_element }* )
   | numeric_literal orless
      ( subcomponent_state_element { , subcomponent_state_element }* )

subcomponent_state_element ::= subcomponent_error_state [ error_type_set ]
subcomponent_error_state ::= 
   { subcomponent_identifier . }" error_behavior_state_identifier

**Naming Rules**

(N1) The reference to an error behavior state machine in the *use behavior* clause must be qualified with the name of the package that contains the declaration of the error behavior state machine being referenced unless the error behavior state machine is located in the same package as the reference.

(N2) The package name of a qualified error behavior state machine reference must be named in the *with* clause of the enclosing package.

(N3) The *use_error_types (use types)* clause makes the defining identifiers of error types and type sets from the listed Error Model libraries referable within the error behavior state machine declaration. If multiple Error Model libraries provide the same defining identifiers, references to these identifiers must be disambiguated by a qualifying package name (see also Section Annex E.7 Naming Rule (N1)).

(N4) References to an *error type* or *error type set* of an *error_type_set* statement of a subcomponent error state clause must exist in the namespace of one of the Error Model libraries listed in the *use types* clause.

(N5) The identifier of the first subcomponent reference must exist in the namespace of the component that contains the component error behavior specification. Subsequent referenced subcomponents must exist in the classifier namespace of the preceding subcomponent.

(N6) The error behavior state reference in a subcomponent error state clause must exist in the namespace of the error behavior state machine associated with the last subcomponent, i.e., with its classifier.

(N7) The *emv2_annex_specific_path* of an *emv2_contained_property_association* in a composite error behavior specification must consist of reference to an error state identifier that is accessible in the namespace of the composite error behavior specification. For error states this reference may optionally be followed by an *error_type_reference*, separated by a dot ("."). This error type reference must be an error type included in the error type set associated with the error state.

**Legality Rules**

(L1) The *error_type_set* of the error behavior state reference in a subcomponent error state clause must be contained in the error type set specified with the defining state declaration.

**Consistency Rules**

(C1) For each composite state there must be at most one composite error behavior state specification with the same resulting typed token token.

(C2) For each composite state there must be at most one composite state declaration, if no resulting typed token token is specified for the composite state or the composite state is not typed.

(C3) For each composite state there must be a corresponding error state in the component error behavior specification for the same component. This means that both the component and composite error behavior specification of a component must refer to the same error behavior state machine or an extension that does not add states.

(C4) Each incoming and outgoing error propagation in the component error behavior specification of a component must be consistent with the incoming or outgoing error propagations associated with the subcomponents, for which connections or observable paths have been declared. This means that the error type set of an incoming error propagation point of a component must be contained in the error type set of the subcomponent error propagation point. Similarly, the error type set of connected outgoing error propagation points of subcomponents must be contained in the error type set of the outgoing component error propagation point.
Semantics

(2) Composite error behavior states represent error behavior states of the composite component in terms of its subcomponents. For example, an operational error behavior state may reflect the fact that a component with redundant parts may continue to be operational even though one of the parts has failed.

(3) A composite error behavior specification allows us to derive reliability models, e.g., MTTF of a system (subsystem) in terms of its subcomponent probabilistic error behavior. In other words, MTTF is the probability that we enter a nonworking state. Similarly, a composite error behavior specification allows us to derive fault trees based on the error behavior states of the subcomponents and the subcomponents contribute to the working condition of the composite component.

(4) A component can have a composite error behavior model expressed in terms of the subcomponent error behaviors, and component error behavior specification that represents an abstraction of the component error behavior used in the interaction with other components and in determining the error behavior of its enclosing component. The error behavior states of the component error behavior specification must be consistent with the composite error behavior states, i.e., consist to the same error behavior states.

(5) The component error behavior specification includes incoming and outgoing error propagations, including observable error propagations, as well as error flows. These must be consistent with the error propagations specified for the subcomponents following the error propagation paths between the subcomponent and the enclosing component error propagation points.

(6) The component error behavior specification includes an error behavior state machine with error behavior events. These represent abstractions of error behavior events associated with the composite component. For example, the composite component may have a no service error event that is an abstraction of one or more redundant subcomponents failing. The occurrence probability of such an event must be consistent with the probability the component is in the error behavior state that is the target of a transition triggered by this event. This probability is computed based on the specified subcomponent error behavior state condition.

Example

composite error behavior
use behavior ErrorLibrary::BabblingSM;
composite states
[A.Operational and B.Operational
 or A.Operational and B.Fail_Stop
 or A.Fail_Stop and B.Operational
]-> Operational;
[ A.Fail_Stop and B.Fail_Stop
]-> Stopped;
[ A.Fail_Babbling or B.Fail_Babbling
]-> Babbling;
end composite;

Annex E.11 Connection Error Behavior

(1) Error propagation through connections takes on a special form. Connections are bound to virtual buses representing protocols and virtual channels and buses, processors, or devices to represent the logical and physical medium that transfers the data from the connection source to its destination. Erroneous behavior of platform component during the transfer, i.e., by the virtual bus, bus, and other hardware components, can propagate errors
into the connection through the binding relationship and affect the error propagation to the connection target. This is illustrated conceptually in Figure 14.

Figure 14 Error Propagation and Connections

(2) A connection can be the source of errors due to interface mismatch between the connection source and the connection target. Examples of such mismatches are different expected measurement units or value reference point for data, or mismatches in the base type representation. Such mismatches typically manifest themselves as value errors and become detectable through design time checking or at runtime if such information is carried as Meta data.

(3) Type transformation sets are used to determine how an error propagation from the source component of a connection is affected by any error propagation from a hardware component such as bus, virtual bus, processor, device, or system, resulting in a changed error type token at the destination component of the connection.

Syntax

connections_error_behavior ::= 

connection error 

[ use_error_types ] 

[ use transformations type_transformation_set_reference ; ] 

{ connection_error_source }* 

[ properties 

{ error_source_emv2_contained_property_association }' 

] 

end connection;

connection_error_source ::= 

defining_error_source_identifier : 

error source ( connection_identifier | all ) 

[ error_type_set_reference ] [ when fault_source_error_type_set_reference ] ;
Naming Rules

(N1) Reference to the type transformation set exit in the Error Model library and must be qualified with the package name of the Error Model library if the connection error behavior is declared in a different Error Model library.

(N2) The package name of a qualified type transformation set reference must be named in the with clause of the enclosing package.

(N3) The error type set reference or error type references in the error type set must exist in an error type library listed in the use types clause.

(N4) The Error Type library references in the use types clause must exist as package names in the with clause of the package containing the connection behavior declaration.

(N5) The connection identifier must exist in the component implementation, whose subclause contains the connection error behavior declaration.

Semantics

(4) Connection error behavior consists of error propagations from the platform components that perform the communication transport, i.e., the component identified by the connection binding, and of the connection being an error source due to interface mismatch between the source and the target of the connection.

(5) Connection as error source can be defined for each connection separately or for all connections within the component implementation that contains the connection behavior declaration.

(6) The error source declaration specifies the possible error types being propagated into the transported data as an error type set. Optionally, the error source declaration also specifies the error types that are the original error source, i.e., types that reflect different kinds of interface mismatches.

(7) The specified type transformation set is used to determine the target type token arriving at the connection destination from the source type token of the connection source, when combined with contributor type tokens, first the connection error source and then the incoming propagations from the connection binding.

Annex E.12 Error Type Mappings and Transformations

(1) There are three scenarios in which mappings or transformations between error types occur. First, an error flow path from an incoming propagation to an outgoing propagation of a component may map the error type of one or more elements of an type token to a different type. Second, the type token of a transition target may be determined by the type token of the source and that of the transition trigger, i.e., by the token of the triggering event or incoming error propagation. For example, the type token of the source may stay the same or one or more of its token elements may change their type due to the type token contributed by the error event or transition condition. Third, the type token of an error propagation along a connection may be affected by the type token being propagated from the virtual bus, bus, or other platform component that the connection is bound to. This section introduces constructs to specify reusable sets of such type mappings and type transformations.

(2) Type mapping sets are introduced to allow for specification of reusable sets of mapping rules between source and target types. The rules of a type mapping set specify how the error type of individual token elements are mapped into other error types, or how an error type token that satisfies a constraint is mapped into another error type token. A type mapping set can be associated with error flow paths.

(3) Type transformation sets are introduced to allow for specification of reusable sets of transformation rules for combining a source type and contributor type into a target type. The rules of a type transformation set specify how a source error type token that satisfies a constraint and a contributor error type token that satisfies a second constraint is mapped into a target error type token. A type transformation set can be associated with a component to be used on connections and as default transformation on transitions.
Syntax

type_mapping_set ::= 
    type mappings defining_type_mapping_set_identifier 
    [ use_error_types ] 
    ( ( type_token_mapping )+ 
    | ( element_type_mapping )+
    )
end mappings;

element_type_mapping ::= 
    source_error_type_reference -> target_error_type_reference ;

type_token_mapping ::= 
    source_error_type_set -> target_type_token ;

type_transformation_set ::= 
    type transformations defining_type_transformation_set_identifier 
    [ use_error_types ] 
    ( ( type_token_transformation )+ 
    | ( element_type_transformation )+
    )
end transformations;

element_type_transformation ::= 
    source_error_type_reference ->[ contributor_error_type_reference ]-> 
    target_error_type_reference;

type_token_transformation ::= 
    source_error_type_set ->[ contributor_error_type_set ]-> 
    target_error_type_token ;

type_transformation_set_reference ::= 
    [ package_reference :: ] type_transformation_set_identifier

Naming Rules

(N1) The defining identifier of the type transformation set or of a type mapping set must be unique within the namespace of the Error Model library.

(N2) References to the type transformation set must be qualified with the package name of the Error Model library if the reference is declared in a different Error Model library.
The package name of a qualified type transformation set reference must be named in the with clause of the enclosing package.

The use_error_types (use types) clause makes the defining identifiers of error types and type sets from the listed Error Model libraries referable within the error behavior state machine declaration. If multiple Error Model libraries provide the same defining identifiers, references to these identifiers must be disambiguated by a qualifying package name (see also Section Annex E.7 Naming Rule (N1)).

References to an error type or error type set of an error_type_set statement in a type mapping or type transformation declaration must exist in the namespace of one of the Error Model libraries listed in the use types clause.

References to an error type or error type set of an error_type_set statement in a type transformation declaration must exist in the namespace of one of the Error Model libraries listed in the use types clause.

References to an error type in an error_type_set statement must exist in the namespace of one of the Error Model libraries listed in the use types clause.

Consistency Rules

A type mapping set must define a unique set of mappings, i.e., every given source error type token must result in exactly one target error type token. This means that the source error type token constraints of two type mapping rules must not intersect.

A type transformation set must define a unique set of transformations, i.e., for any given source token and contributor token at most one transformation rule must apply.

A type mapping set must cover all possible source tokens of the entity the type mapping set is applied to.

A type transformation set must cover all possible source error type tokens and contributor error type tokens of the entity the type mapping set is applied to.

Semantics

Type mapping sets are used with error flow paths to change an incoming type token into a type token with different element types.

A type mapping set specifies a set of mapping rules of source error type tokens into target error type tokens. These rules take one of two forms: an element type mapping rule, or a type token mapping rule.

An element type mapping rule specifies the mapping of a single token element of an error type token into another error type value of the same type hierarchy or an error type from a different type hierarchy, replacing the original token element.

When element type mapping rules are applied to a type token with a type product, then each element of the type product is mapped according to the mapping rules. One element type mapping rule is applied to each token elements of the source error type token to determine the target error type token. The elements of the resulting target product type must be from different type hierarchies. Note in this form of type mapping the cardinality of the type product in the source and target error type tokens remain the same.

A type token mapping rule specifies the mapping of an error type token that is contained in the source error type set into another error type token. One type token mapping rule is applied to the source error type token to determine the target error type token. In this form of type mapping the cardinality and the element types of the source and target error type tokens may be different.

Type transformation sets are used with transitions and with connections to indicate how a source error type token is combined with a contributor error type token, i.e., a transition trigger or binding propagation of a connection, is changed into a target error type token. If multiple contributor error type tokens are involved, e.g., due to a
conjunction condition or a connection binding to multiple platform components, the transformation rules are applied to each contributor in order.

(10) A type transformation set specifies a set of transformation rules that take one of two forms: an element type transformation rule, or a type token transformation rule.

(11) An element type transformation rule specifies the transformation of a single token element of a source error type token combined with a token element from the same type hierarchy of a contributor error type token into another error type value of the same or a different type hierarchy replacing the original token element.

(12) In the case of type tokens that consist of a type product, the transformation rule is applied to each product type element. The elements of the resulting target product type must be from different type hierarchies. In this form of type mapping the cardinality of the source and target error type tokens remain the same.

(13) A type token transformation rule specifies the transformation of an error type token that is contained in the source error type set combined with an error type token that is contained in the contributor error type set into an error type token specified as target. One type token transformation rule is applied to a pair of source and contributor error type tokens to determine the target error type token. In this form of type mapping the cardinality of the source, contributor, and target error type tokens may be different.

(14) The target type token in a type mapping or type transformation rule refers consists of the specified error type or type product, even when the referenced error type has subtypes.

Example

type transformations Mytransformations
use types ErrorModelLibrary;
  {MyType} -{ {NoService} }-> {NoValue};
end transformations;

Annex E.13 Error Models and Fault Management

(1) Error detection declarations map error occurrences, reflected in the error behavior state of a component and incoming error propagations into events or event data with an error code value in the AADL core model. The event or event data can be communicated via port connections to the appropriate health monitoring component. The behavior specification of such a component will indicate the processing and decision logic involved in tolerating the fault occurrence. One of the actions may be a reconfiguration of the runtime architecture through a mode switch.

(2) A reconfiguration mode switch in the fault management architecture of the actual system may be an explicit recovery action. Such a recovery action can be mapped back into the Error Model of a component as a recover event that initiates an error behavior transition to reflect the recovery.

(3) A reconfiguration mode switch in the fault management architecture of the actual system may exclude a failed component from the active configuration. This allows the component to be repaired through a repair agent. A repair action may not be explicitly modeled, but its completion is mapped into a repair event. This allows the effect of a repair to be reflected back into the error behavior model. In this case the occurrence and duration of the repair are represented by properties on the repair event (see Section Annex E.8.1). Alternatively, a repair action can be explicitly represented in the Error Behavior model as a separate state.

(4) Section Annex E.9.3 described a component with operational modes and failure modes. The resulting combined operational and failure mode behavior model represents the actual behavior of the component under nominal and anomalous conditions without explicit detection of the failure. Such a state change and restriction in available operational modes must first be detected by the system before, expressed by a detection declaration, if the system is to give feedback to the initiator of an operational mode change that such a transition is not feasible while in the given error state.

(5) A component receiving data of the failing component may detect error propagations. This is modeled by an error detection on incoming error propagations for the receiving component. The receiving component may then take
A component may be monitoring the connection between two components or may send a heartbeat and monitor the response. Upon error detection it then takes an appropriate fault management action, e.g., by switching to a configuration through a mode transition that does not include the failed component.

Annex E.14 A Reconfigurable Triple Redundant System (Example from Error Model Annex V1)

This section presents an example high-level specification of a reconfigurable system having 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 operating subsystems 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).

A subsystem is assumed to include self-checking algorithms, so that a subsystem is able to 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), the other that produces a subsystem failure that either cannot be self-checked or that prohibits a subsystem from fail-stopping (Bad_Data).

It is also assumed that a subsystem can be dynamically powered up or powered down as a result of 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, i.e., the powered and unpowered modes do not have to be represented as separate error behavior states as was done in the Error Model Annex V1 version of the example.

A subsystem is initially in the Operational error state. A failed system can exist in one of two error 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, e.g. it remains active sending misleading data, and it cannot be powered-down at a mode switch.

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 Fail_Stopped state it will stay in Fail_Stopped upon subsequent error events. It is assumed that when the subsystem is in Fail_Unknown state it transitions to Fail_Stopped state if a subsequent Self-Checked_Fault event occurs.

```prolog
package packageET
public
annex EMV2 {**
error types
    Data_Fault: type;
    No_Data: type extends Data_Fault;
    Bad_Data: type extends Data_Fault;
end types;
**};
end packageET;

package packageEM`
public
annex EMV2 {**
error behavior Example
use types packageET;

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]-> Fail_Stopped;
UncoveredFail: Operational -[Uncovered_Fault]-> Fail_Unknown;
UFailUFault: Fail_Unknown -[Uncovered_Fault]-> Fail_Unknown;
SFailUFault: Fail_Stopped -[Uncovered_Fault]-> Fail_Stopped;

end behavior;
**};
end packageEM;

(7) Next we specify error propagation behavior of a component as perceived by other components using the error propagations clause. 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 vote using other redundant sources of data to detect the failure. The error flow declarations indicate that the component can be the source of propagations, can pass the propagations on, or detect and mask them, i.e., act as sink. The conditions under which an incoming error propagation will be masked (error sink) or passed on (error path) will be specified in the component error behavior specification.

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 {**
error propagations

use types packageET;

    A : in propagation {Data_Fault};
    B : in propagation {Data_Fault};
    O : out propagation {Data_Fault};

flows

    es1: 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;

**};
end Subsystem;

(8) The component error behavior clause specifies the details of the component error behavior of the subsystem.

(9) 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 Occurrence property specified for Bad_Data while remaining in the Fail_Unknown state.

(10) 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 (self-testing coverage approximately 95%). Faults in unpowered subsystems occur at 1/10 the rate of faults in powered ones.

(11) 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.

system implementation Subsystem.Notional

annex EMV2 {**

    component error behavior

    use types packageET;

    use behavior packageEM::Example;

    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}] -> mask;
        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;
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 are specifications of nominal behavior. They are written assuming this subsystem is error-free and typically only name error states of other components. An erroneous subcomponent by definition no longer obeys its nominal specification, and 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 individual subsystem.

The voting protocol must be able to detect No_Data, e.g., by the absence of a fresh value on the incoming port, and Bad_Data, e.g., by comparison with own and other incoming value. 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 only has two outgoing events, one for 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 states could be included in an error model, but this example only models 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 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 state and propagating Bad_Data errors. This specification assumes that Bad_Data will be detected by an error-free subsystem providing at least one of the other subsystems is error-free. In an actual system a Byzantine error may occur, which is to say 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.

```plaintext
system Dependable_System
end Dependable_System;

system implementation Dependable_System.Notional
subcomponents
  S1: system Subsystem.Notional in modes (S123, S124, S134);
  S2: system Subsystem.Notional in modes (S123, S234, S124);
  S3: system Subsystem.Notional in modes (S123, S134, S234);
  S4: system Subsystem.Notional in modes (S124, S134, S234);
connections
  c1: port S2.0 -> S1.A in modes (S123);
  c2: port S3.0 -> S1.B in modes (S123);
  c3: port S1.0 -> S2.A in modes (S123);
  c4: port S3.0 -> S2.B in modes (S123);
  c5: port S1.0 -> S3.A in modes (S123);
  c6: port S2.0 -> S3.B in modes (S123);
  c11: port S2.0 -> S4.A in modes (S234);
  c12: port S3.0 -> S4.B in modes (S234);
  c13: port S4.0 -> S2.A in modes (S234);
  c14: port S3.0 -> S2.B in modes (S234);
  c15: port S4.0 -> S3.A in modes (S234);
  c16: port S2.0 -> S3.B in modes (S234);
  c21: port S4.0 -> S1.A in modes (S134);
  c22: port S3.0 -> S1.B in modes (S134);
  c23: port S1.0 -> S4.A in modes (S134);
  c24: port S3.0 -> S4.B in modes (S134);
  c25: port S1.0 -> S3.A in modes (S134);
  c26: port S4.0 -> S3.B in modes (S134);
  c31: port S2.0 -> S1.A in modes (S124);
  c32: port S4.0 -> S1.B in modes (S124);
  c33: port S1.0 -> S2.A in modes (S124);
  c34: port S4.0 -> S2.B in modes (S124);
  c35: port S1.0 -> S4.A in modes (S124);
  c36: port S2.0 -> S4.B in modes (S124);
modes
```
A dependable system 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.

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 raise events that are ultimately connected to mode transitions. The Behavior Annex subclause specifies that a system will only switch modes when two subsystems raise an event requesting the same 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 can take an error-free subsystem off-line and leave a failed subsystem on-line. This 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.

(22) To give an example illustrating the above 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 FailUnknown error state and propagate BadData) 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.

(23) 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 would be taken into consideration if the modeling and analysis tool considered all possible subsets of consequent in error propagation transitions, in particular considered the case where a BadData 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 on-line. The system is now only one fault away from unsafe system failure.

(24) The error state of the system as a whole is defined as a function of the error 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 unsafe.