SAE Aerospace
An SAE International Group

AEROSPACE STANDARD

SAE AS5506/3

Draft 2013-12-09
Draft 0.97

SAE Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model Annex

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 reliability, availability, 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:

Dec 2013: V0.97

Updated consistency rules.

Made Hazards properties a separate section and added ARP4761 and MILSTD882 specific definitions.

Added HazardAssignment property

Consistent use of {} around error types and type sets. Exception is containment path for properties uses dot to separate EMV2 element from type, i.e., subcompname.erroreventname.errortypename

Updated wording in terms and definitions section.

Aug 2013: V0.96

*Added **all** source and any (optional) contributor to type transformation rules.*

*Specification of type equivalence (**use type equivalence**) for independently developed error type libraries. Makes use of type mapping sets.*

***Use mappings** declaration for error paths now at subclause level.*

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: custsvc@sae.org

SAE WEB ADDRESS: **http://www.sae.org**

Error propagation declarations require a type set (was optional before).

The composite error behavior specification can also refer to incoming propagations to reflect how the composite state is not only impacted by subcomponent states reflecting component internal error sources, but also by external error sources.

Type sets can be referenced as elements of type sets. The effect is a union of the types.

Updated definitions of terms and hazards properties (thanks to input from Steve Hosner)

July 2013: V0.95 (reflects discussions from July 2013 meeting)

Reduced keywords for binding related propagation points to those used in the binding properties.

*Added **all** indicator for all outgoing or incoming propagation points in error flows, as well as in outgoing propagation conditions and error detection conditions. We already did allow it for the source of a transition.*

*Allow explicit declaration in outgoing propagation condition that incoming propagations are masked by specifying **(noerror)***

Added syntax to identify the initiator of a recover event in the core model, i.e., a mode transition or an event.

*Eliminated **mask** keyword for steady state transition. We use **same state** instead.*

Eliminated the use of action in the definition of service and service items. The use of action resulted in a confusion between duration of a service action to produce a service item and the delivery time of a service item.

Rewording to distinguish between failures and error propagation.

Fixed OutOfOrder definition

Added UndetectableValueError and DetectableValueError with SubtleValueError and BenignValueError as aliases

Added Calibration Error, Stuck Value, Delayed Service, and Early Service as predeclared error types.

April 2013: V0.94

Observable propagation points: Now a separate section to declare propagation points and connections

Added detection conditions referring to subcomponent out propagations

Renamed Contained Error to Error Containment to align with Error Propagation

Added inheritance of EMV2 declarations due to component extends hierarchy

Added several distribution functions (thanks to Myron for the input)

April 2013: V0.93 Discussed at Feb 2013 Mtg

Added error propagation support for features in feature groups

Removed extends from Error Behavior State Machines

Simplified type mapping and type transformation sets (removed element mapping rules)

Type token => type instance. Syntax use () instead of {} to distinguish from type set which uses {}

Feb 2013: V0.92

Error type library and EBSM dependency into other packages not required in package with clause. Property set dependency still required in enclosing package.

Single properties section in the error annex subclause instead of separate ones for propagation, component, and composite clauses. Single use types and use behavior clause.

Events section in component error behavior section to allow for component-specific error events.

*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. Separate Severity and Likelihood properties.

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

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 reliability, availability, 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 reliability, availability, 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 AADL 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

1.1 Normative References

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

(2) ISO/IEC/IEEE 24765:2010 Systems and software engineering — Vocabulary, Dec 2010.

(3) SAE AS-5506B:2012, Architecture Analysis & Design Language (AADL), Sept 2012.

(4) SAE AS-5506/1:2006, Architecture Analysis & Design Language (AADL) Annex Volume 1, June 2006.

(5) SAE AS-5506/2:2011, Architecture Analysis & Design Language (AADL) Annex Volume 2, Jan 2011.

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

(7) SAE ARP-4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

(8) SAE ARP-5580, Recommended Failure Modes and Effects Analysis (FMEA) Practices for Non-Automobile Applications, July 2001.

(9) DO-178B/C Software Considerations in Airborne Systems and Equipment Certification, December 1992. Revised, December 2011.

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

(11) MIL-HDBK-217F Reliability Prediction of Electronic Equipment, December 1991.

(12) MIL-STD-882D Standard Practice for System Safety, February 2000.

1.2 Informative References

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

(2) [LAP 1992] Laprie, J.-C., editor, “Dependability: Basic Concepts and Terminology,” Dependable Computing and Fault Tolerance, volume 5, Springer-Verlag, Wien, New York, 1992. Prepared by IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance.

(3) [IFIP WG10.4-1992] IFIP WG10.4 on Dependable Computing and Fault Tolerance, 1992, J.-C. Laprie, editor, “Dependability: Basic Concepts and Terminology,” *Dependable Computing and Fault Tolerance*, volume 5, Springer-Verlag, Wien, New York, 1992.

(4) [BNF 1960] Naur, Peter (ed.), "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM*, Vol. 3 No. 5, pp. 299-314, May 1960.

(5) [CISHEC 1977] “A Guide to Hazard and Operability Studies”, The Chemical Industry Safety and Health Council of the Chemical Industries Association Ltd., 1977.

(6) [HAZOP 1992] T. Kletz, “Hazop and Hazan: Identifying and Assessing Process Industry Hazards”, Institution of Chemical Engineers, third edition, 1992.

(7) [SHARD 1994] J. A. McDermid and D. J. Pumfrey, “A development of hazard analysis to aid software design”, in COMPASS ’94: Proceedings of the Ninth Annual Conference on Computer Assurance. IEEE / NIST, Gaithersburg, MD, June 1994, pp. 17–25.

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

(9) [Powell 1992] D. Powell, “Failure Mode Assumptions and Assumption Coverage”, Twenty-Second International Symposium on Fault-Tolerant Computing, 1992. FTCS-22.

(10) [DAM 2008] Bernardi, S., Merseguer, J., Petriu, D.: “An UML profile for Dependability Analysis and Modeling of Software Systems”. Technical Report RR-08-05, Universidad de Zaragoza, Spain (2008) http://www.di.unito.it/~bernardi/DAMreport08.pdf.

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

(12) [Miller 2005] Miller S., Whalen M., O’Brien D., Heimdahl M., and Joshi A., A Methodology for the Design and Verification of Globally Asynchronous/Locally Synchronous Architectures, NASA/CR-2005-213912, Sept 2005.

(13) [Rugina 2007] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaaniche. 2007. A system dependability modeling framework using AADL and GSPNs. In *Architecting dependable systems IV*, Rogrio de Lemos, Cristina Gacek, and Alexander Romanovsky (Eds.). Lecture Notes In Computer Science, Vol. 4615. Springer-Verlag, Berlin, Heidelberg 14-38.

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 reliability, availability, 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 for architecture fault modeling referred as Error Model V2 (EMV2). Declarations in expressed in EMV2 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 defect, error, fault, failure, hazard, etc. according to the definitions in [ISO/IEC/IEEE 24765:2010] (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) EMV2 supports architecture fault modeling at three levels of abstraction with the following concepts:

- Error propagation: Focus on fault sources in a system and their impact on other components or the operational environment through propagation. Architecture fault model specifications at this level of abstraction correspond to the Fault Propagation and Transformation Calculus (FPTC) [Paige 2009] and allows for safety analysis in the form of hazard identification, fault impact analysis, and stochastic fault analysis.
- Component error behavior: Focus on a system or component fault model identifying faults in a system (component), their manifestation as failure, the effect of incoming propagations, and conditions for outgoing propagation. This component error behavior specification views a component as a single unit, i.e., characterizes the possible fault occurrences and resulting failure states of the component as a whole. A component error behavior specification reflects handling of redundant input to tolerate failures external components, differences in handling error events and incoming propagation in different failure modes, as well as restrictions failure modes place on operational modes. This level of architecture fault model specification allows for fault tree analysis of a system stochastic reliability and availability analysis of systems in terms of its components and their interactions.
- Composite error behavior: Focus on relating the fault model of system components to the abstracted fault model of the system. The composite error behavior specification provides a mapping between the component error behavior specifications of subsystems to the abstracted component error behavior specification of the enclosing system. This layered abstraction allows for scalable compositional analysis.

(6) EMV2 introduces the concept of error type to characterize faults, failures, and propagations. Sets of error types are organized into error type libraries and are used to annotate error events, error states, and error propagations. The Error Model Annex includes a set of predefined error types as starting point for systematic identification of different types of fault propagations – providing an error propagation ontology. Users can adapt and extend this ontology to specific domains.

(7) EMV2 supports the error propagation abstraction through *error propagation* declarations, i.e., specification of propagated error types 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 assumed to not be propagated (*error containment*). For each component we can also specify an *error flow*, i.e., whether a component is the *error source* or *error sink* of a propagation, or whether it passes on an incoming propagation as an outgoing propagation of the same or different error type (*error 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.

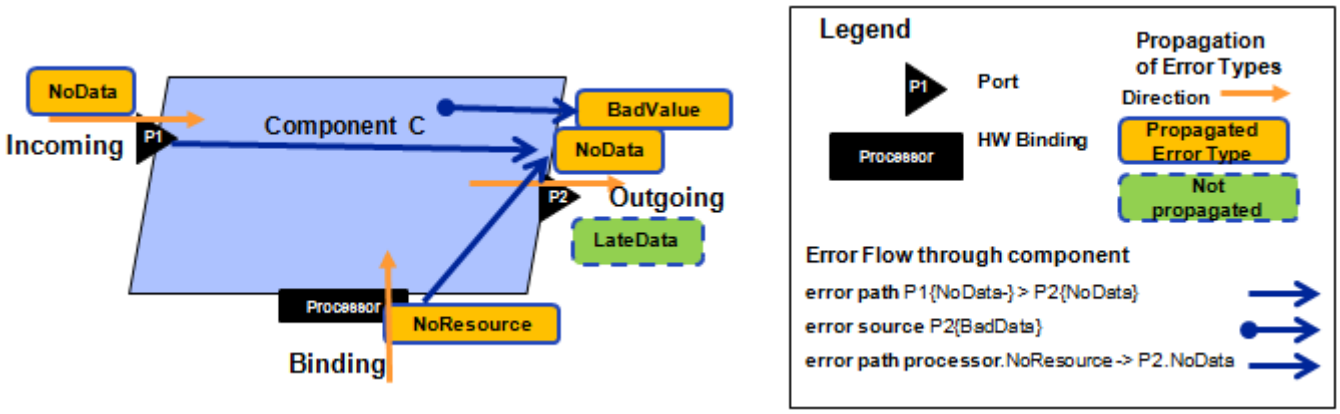

Figure 1 Error Propagations and Error Propagation Flows

(8) The EMV2 supports the specification of *component 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 error events and in terms impact of incoming error propagations from other components, as well as conditions under which outgoing error propagations occur that can impact other components.

(9) 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 and erroneous behavior on 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.

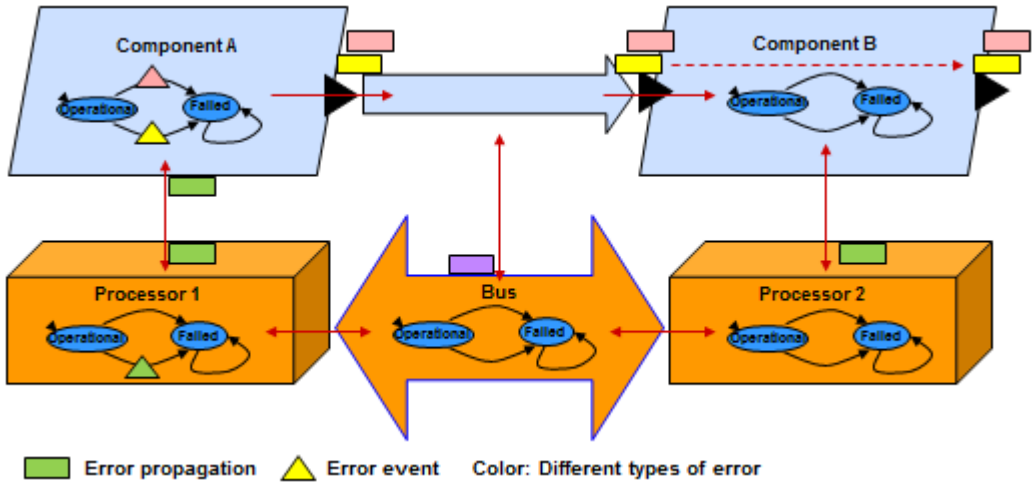

Figure 2 Error Propagation Paths Defined by Architecture

(10) The EMV2 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.

(11) The EMV2 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.

(12) Finally, the EMV2 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.

(13) 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.

(14) The Error Model Annex can be used to annotate the AADL model of an embedded system to support a number of the assessments/analyses cited in SAE ARP4761, "Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment" and MIL-STD-882E System Safety. An architecture specification with error annex annotations may be subjected to a variety of assessment/analysis methods ranging from SAE ARP4761's Functional Hazard Assessments (FHA) to Fault Tree Analysis (FTA) to stochastic reliability and availability analysis or from MIL-STD-882E's Preliminary Hazard List to Safety Assessment Reports to failure mode effects and criticality analysis. For example, fault trees can be generated from such specifications to qualitatively determine/verify a hazard's contribution to a failure condition or its quantitative probability of occurrence; stochastic process models (Markov or Stochastic Petri nets) can be generated to assess reliability and availability.

(15) The error models of low-level components are also typically used to capture the results of bottom-up analyses (e.g. failure modes and effects analysis as defined in SAE ARP 5580). The error models of the overall system and high-level subsystems are also typically used to capture the results of top-down analyses (e.g. functional hazard assessments as 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 functional hazard assessments, failure modes and effects analysis, and the safety and reliability analyses that relate the two.

(16) 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.

(17) 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.9 introduces constructs to define reusable error behavior state machines with error and repair events, states, and transitions. Section Annex E.10 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.11 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.12 discusses how error propagation affecting connections is represented. Section Annex E.13 introduces constructs that allow you to define reusable error type mappings and transformations. Section Annex E.14 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 an example architecture model of a triple redundant system annotated with Error Model Annex specifications.

Annex E.2 Concepts and Terminology

(1) The Error Model Annex supports specification

(2) The definitions of this section are based on the definition of terms in Systems and software engineering — Vocabulary [ISO/IEC/IEEE 24765:2010].

(3) *Error* is defined as 1. a human action that produces an incorrect result, such as software containing a fault. 2. an incorrect step, process, or data definition. 3. an incorrect result. 4. the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition cf. failure, defect.

- The definition of error encompasses mistakes by humans (the effect of a failure by the human), defects in a process that can lead to defects in a design or operational system, the effect of incorrect system behavior, and a characterization of anomalous behavior as an indication of a failure.
- In EMV2 we consistently use the term *error* as keyword to avoid confusion with similar constructs in the AADL core language or other annexes, e.g., event vs. error event or state vs. error state. See also defects, failures, and effects.

(4) *Defect* is defined as a generic term that can refer to either a fault (cause) or a failure (effect).

- See also fault and failure.

(5) *Fault* is defined as 1. a manifestation of an error in software. 2. an incorrect step, process, or data definition in a computer program. 3. a defect in a hardware device or component. Syn: bug NOTE: A fault, if encountered, may cause a failure.

- The definition of fault includes one of the definitions for error. It also is defined in terms of defect, which includes a definition in terms of fault. It also is the effect of human or process errors.
- In EMV2 we represent fault types as error types. A property lets us distinguish between design faults and operational faults. The presence of the fault in a component is expressed as an error source with the appropriate error type as the origin. In a Level 2 abstraction it is expressed as an error event with an error type. An instance of an error event represents the activation of a fault (see failure).

(6) *Failure* is defined as 1. termination of the ability of a product to perform a required function or its inability to perform within previously specified limits; 2. an event in which a system or system component does not perform a required function within specified limits.

- The definition of failure talks the event of a fault activation and the manifestation of the fault activation in the component. The manifestation can be a malfunction, i.e., loss of service or anomalous behavior. See also failure mode.
- In EMV2 we represent failures as error sources and as error events. The manifestation of the failure is represented by an error state. A transition specification maps error events into error states. See also failure.

(7) *Failure mode* is defined as the physical or functional manifestation of a failure.

- According to the definition, a failure mode is associated with a physical or logical component.
- In EMV2 we represent failure modes as error states. Error states are defined as part of error behavior state machines. Error states can have error types to characterize different types of failure modes. Different types of faults are mapped to the respective type of failure mode through a transition specification. See also failure.

(8) *Failure mode and effect analysis* (FMEA) is defined as 1. [Technique] an analytical procedure in which each potential failure mode in every component of a product is analyzed to determine its effect on the reliability of that

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

- Although [ISO 2010] refers to effects it does not define the term. In the context of the FMEA definition, effect refers to failure mode. The effect of a failure is its propagation to other components and their response to this propagation. Such propagation is also referred to as fault propagation. The outgoing propagation can be due to a component failure or due to an incoming propagation. From the receiver's perspective there is no difference, but from a diagnostic perspective it is useful to be able to identify the source.
- In EMV2 we represent propagation of failure modes and incoming effects as error propagations. Conditions for outgoing propagations provide traceability to whether it is due to a failure (error state) or due to the effect of an incoming propagation. The effects of a propagation are represented by error sink and path specifications at Level 1 abstraction. In terms of Level 2 abstraction the effect resulting in a failure mode is expressed in a transition condition referring to the incoming propagation, and the effect without affecting a failure mode or conditionally on a failure mode as outgoing propagation condition. Probability of occurrence is associated with error events, error sources, error states, and error propagations to support stochastic analysis. The logic conditions also reflect fault tolerance strategies such as redundancy.

(9) *Hazard* is defined as 1. an intrinsic property or condition that has the potential to cause harm or damage. 2. a potentially unsafe condition resulting from failures, malfunctions, external events, errors, or a combination thereof. 3. a source of potential harm or a situation with a potential for harm in terms of human injury, damage to health, property, or the environment, or some combination of these.

- The definition of hazard has its root in safety engineering. It refers both to the effect and the source of a failure. Failures that result in loss of service are hazards that affect reliability. From a safety perspective even minor failures must be considered as hazards as combinations of them can result of catastrophic effects (see [Leveson 2012]). The definition of hazard focuses in effects of failures in terms of injury and damage, i.e., represent safety hazards. We include security hazards under the concept of hazard (see below).
- In EMV2 we represent hazards by a multi-valued property that can be associated with the error source, error state, and error propagation to support both definitions of hazard.

(10) *Security* is defined as the protection of system items from accidental or malicious access, use, modification, destruction, or disclosure.

- The definition of security includes accidental malicious indication of anomalous behavior either from outside a system or by unauthorized crossing of system internal boundaries –typically taking advantage of faults. The term system item covers information as well as physical components.

(11) *Threat* is defined as 1. a state of the system or system environment which can lead to adverse effect in one or more given risk dimensions. 2. a condition or situation unfavorable to the project, a negative set of circumstances, a negative set of events, a risk that will have a negative impact on a project objective if it occurs, or a possibility for negative changes.

- The definition of hazard focuses in effects of failures in terms of injury and damage, i.e., represent safety hazards.
- In the context of EMV2 we include security hazards under the concept of hazard. Threats are one class of a security hazards.

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

(13) *Error tolerance* is defined as 1. the ability of a system or component to continue normal operation despite the presence of erroneous inputs.

- The definition of fault tolerance focuses on the presence of hardware and software faults, while the definition of error tolerance focuses on propagated errors. Fault tolerance focuses on fault detection, fault isolation, and fault

recovery. Fault recovery elements include fault containment, fault masking, fault repair, and fault correction. Fault avoidance focuses on not introducing faults or eliminating them before the system goes into operation.

- In EMV2 tolerance of both fault and errors is supported by transition and outgoing propagation conditions referring to error states and incoming propagations. Expectations on fault tolerance by the system can be specified at Level 1 abstraction through error paths and error sinks. At Level 2 expected detection of faults by the system is specified by error detection declarations as well as recover and repair events. This provides traceability into the fault tolerance architecture of the system.

(14) 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/C and RTCA DO-254 provide guidelines for assuring that fielded avionics software code and hardware circuits have acceptably low risk of consequential design errors.

(15) For mechanical components the error behavior specified by EMV2 can reflect fault occurrences in the physical component in use as well as latent design faults. 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, an 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.

(16) The following definitions of safety terms are based on definitions and usages in MIL-STD-882E, SAE ARP4754A and SAE ARP4761.

(17) A *failure condition* is a condition with an effect on the aircraft and its occupants, both direct and consequential, caused or contributed to by one or more failures, malfunctions, external events, errors, or a combination thereof, considering relevant adverse operation or environmental conditions.

(18) The *failure condition effect* is description of the effects on the aircraft and its occupants, both direct and consequential, caused or contributed to by one or more failures, malfunctions, external events, errors, or a combination thereof, considering relevant adverse operation or environmental conditions.

(19) The *failure condition classification* is an assignment of a discrete scale which categorizes the severity of the effects of a failure condition. The classification levels are defined in the appropriate CFR/CS advisory material (section 1309): Catastrophic, Hazardous/Severe-Major, Major, Minor, or No Safety Effect.

(20) A *failure effect* is a description of the operation of a system or item as the result of a failure; i.e., the consequence(s) a failure mode has on the operation, function or status of a system or an item.

(21) A *mishap* is an event or series of events resulting in unintentional death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment. For the purposes of this Standard, the term “mishap” includes negative environmental impacts from planned events.

(22) A *likelihood* is a non-specific relative indication of how likely it is that a given event will occur. (e.g., high likelihood, low likelihood, reduced likelihood, etc.)

(23) A *probability* is a qualitative or quantitative indication of the *likelihood* that a given event will occur. A qualitative likelihood indication is a specific gradation of likelihood that a given event will occur. A quantitative likelihood indication is a specific numerical value between zero and one. (e.g., Qualitative: Improbable - So unlikely, it can be assumed occurrence may not be experienced in the life of an item. Quantitative: $1.0X10^{-7}$ per hour).

(24) A *risk* is the probability (likelihood) and severity of a hazard.

(25) A *target* risk is a defined risk (combination of hazard severity and probability) that has been determined to be acceptable.

(26) The term *development assurance* refers to all of those planned and systematic actions used to substantiate, at an adequate level of confidence, that errors in requirements, design and implementation have been identified and corrected.

(27) The term *development assurance level* (DAL) refers to the level of rigor of development assurance tasks performed to substantiate, to an adequate level of confidence, that development errors have been identified and corrected such that the system satisfies the applicable certification basis.

- The Error Model Annex includes a set of predeclared properties to support recording of safety analysis related information. The Requirements Definition and Analysis Language (RDAL) Annex provides additional support for recording requirements in the context of an architecture model as well as assurance evidence to demonstrate that the requirements are met.

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.

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 error events, error propagations, in the form of detection and recovery/repair through fault management is represented by modes in the core language and by 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.10.3.

(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 error propagations 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. They specify the composite error behavior of a component in terms of the error behaviors of its subcomponents. Finally, the properties section of the Error Model subclause associates property values with elements of the Error Model annex, such as error behavior events, error propagations, error flows, and error states.

Syntax

```
error_model_subclause ::=
    annex EMV2 (
      ( {** error_model_component_constructs **} )
      |  none )
      [ in_modes ] ;

error_model_component_constructs ::=
    [ use types error_type_library_list ; ]
    [ use type equivalence type_mapping_set_reference ; ]
    [ use mappings type_mapping_set_reference ; ]
    [ use behavior error_behavior_state_machine_reference ; ]
    [ error_propagations ]
    [ component_error_behavior ]
    [ composite_error_behavior ]
    [ connection_error_behavior ]
    [ propagation_paths ]
    [ subclause_EMV2_properties_section ]

error_type_library_list ::=
  error_model_library_reference { , error_model_library_reference }*

error_behavior_state_machine_reference ::=
    [error_model_library_reference :: ] error_behavior_state_machine_identifier

-- adapted from AS5506B 11.3
emv2_contained_property_association ::=
  unique_property_identifier => [ constant ] assignment applies to
     emv2_containment_path { , emv2_containment_path }* ;
```

```
emv2_containment_path ::=
     [ aadl2_core_path @ ] emv2_annex_specific_path

aadl2_core_path ::=
  named_element_identifier { . named_element_identifier }*

emv2_annex_specific_path ::=
  named_element_identifier { . named_element_identifier }*

EMV2_properties_section ::=
  properties
    { emv2_contained_property_association }+
```

Naming Rules

(N1) An Error Model subclause introduces a namespace for component-specific named error model elements, such as error propagations, error flows, error behavior events, etc. (see respective annex section).

(N2) The mode identifiers in the `in_modes` statement of an Error Model Annex subclause must refer to modes in the component type or component implementation for which the annex subclause is declared.

(N3) The **use types** clause makes the defining identifiers of error types and type sets from the listed Error Type libraries accessible to an Error Model subclause, i.e., allowing them to be referenced without qualification. Different Error Type libraries in the **use types** clause must not provide the same defining identifiers.

(N4) Error types and type sets made accessible through a **use types** clause are not considered to be part of the Error Model subclause namespace, i.e., they do not create name conflicts with named error model elements that are part of this namespace, such as error flows or error events.

(N5) The reference to an Error Type library in the **use types** clause is identified by the package name of the Error Model library that contains the Error Type library. The referenced package does not have to be listed in the **with** clause of the package containing the classifier with the Error Model subclause.

(N6) The reference to a type mapping set in the **use type equivalence** clause must exist in the Error Model library identified by the qualifier. The referenced package does not have to be listed in the **with** clause of the package containing the classifier with the Error Model subclause.

(N7) The type mapping set identified in the **use type equivalence** clause associated with a component is inherited by all subcomponents unless overwritten in the error model subclause for that component.

(N8) The reference to a type mapping set in the **use mappings** clause must exist in the Error Model library identified by the qualifier. The referenced package does not have to be listed in the **with** clause of the package containing the classifier with the Error Model subclause.

(N9) The **use behavior** clause includes the named elements in the namespace of the Error Behavior State Machine into the Error Model subclause namespace, i.e., they can be referred to without qualification. Named elements defined within the Error Model subclause must be unique with respect to these named elements as well as other locally defined named element.

(N10) 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 qualifying package of an Error Behavior State Machine reference does not have to be listed in the **with** clause of the package containing the classifier with the Error Model subclause.

(N11) 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, if qualified with a package name, or in the containing Error Model library.

(N12) Predeclared Error Model properties are contained the property set *EMV2*. References to these properties must be qualified with this property set name. The EMV2 property set does not have to be included in the **with** clause of the enclosing package.

(N13) The containment path of an `emv2_contained_property_association` consists of an optional `aadl2_core_path` and an `emv2_annex_specific_path`. The `aadl2_core_path` identifies a sequence of subcomponent references starting with a subcomponent of the component implementation that contains the Error Model subclause containing the property association. The `aadl2_core_path` may end with a connection identifier in the last subcomponent. The `emv2_annex_specific_path` identifies an error model element associated with the last element of the core model path or the classifier that contains the Error Model subclause with the property association. The `emv2_annex_specific_path` may end with a reference to an error type associated with the previous error model element.

(N14) The Error Model subclause of a component type is inherited by component implementations of that type. The Error Model subclause of a classifier that is an extension of another classifier is inherited. The local Error Model subclause may add Error Model elements, or redefine inherited error model elements or **use** declarations. An error model element is redefining an inherited element with the same defining identifier. The following redefinition rules apply, effectively replacing the original declaration:

- Error propagation: change the type set associated with the error propagation point named by the error propagation.
- Error containment: change the type set associated with the error propagation point named by the error containment.
- Error source, sink, and path: change the flow type, e.g., change a sink to a path; change the source and its type set; change the target and its type instance; change the component internal cause of an error source (**when** clause). This includes adding a type set or type instance, when there was none previously, or specifying the source or target without type set or type instance when there was one.
- Error events: change the type set associated with an error event (including addition or removal). Note that error events declared in an Error Model subclause can redefine error events declared as part of an error behavior state machine that is made accessible through **use behavior**.
- Transitions: change the condition, the originating state or its type set, the resulting state or its type instance. Note that transitions declared in an Error Model subclause can redefine transitions declared as part of an error behavior state machine that is made accessible through **use behavior**.
- Outgoing propagation conditions: change the condition, the originating state or its type set, the outgoing propagation point or its type instance.
- Error Detection: change the condition, the originating state or its type set, the resulting event or its error code.
- Composite error states: change the condition, the composite state or its type instance.
- Mode mapping: change the set of modes for the specified error state and type set.
- **Use types** clause: replace the set of type libraries.
- **Use type equivalence** clause: replace the type mapping set reference.
- **Use mappings** clause: replace the type mapping set reference.
- **Use behavior** clause: the referenced error behavior state machine must be the same.

Legality Rules

(L1) Error Model subclauses must only be declared in classifiers, i.e., in component types, component implementations, and in feature group types.

(L2) Composite error behavior declarations, propagation path declarations, and connection error behavior declarations must only exist in Error Model subclauses declared in component implementations.

(L3) Error Model subclauses declared in feature group types must only contain error propagation and containment declarations.

(L4) EMV2 contained property associations must not associate property values with named elements in the core model. This is enforced by the naming rules above.

(L5) If a component type or implementation has mode-specific Error Model subclauses, then all subclauses must reference the same error behavior state machine.

(L6) If a **use behavior** clause is declared locally as well as inherited, then it must refer to the same error behavior state machine.

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) An Error Model subclause allows feature group types to be annotated with error propagation declarations.

(4) Error Model subclauses declared within component types and component implementations apply to all subcomponents (component instances) referencing this component classifier. Error Model subclauses declared within feature group types apply to all feature groups referencing this feature group classifier.

(5) The Error Model subclause in a component implementation may declare an error propagations section, component error behavior section, and a composite error behavior section. The error propagations section and component error behavior section of the implementation may add or override declarations in the corresponding section of the component type subclause. For example, a component implementation may add an error behavior transition, or specify an implementation-specific set of error types for an error propagation declared in the component type.

(6) Similarly, an Error Model subclause in a component type or implementation extension may add or override declarations in the error propagations, component error behavior, and composite error behavior sections of the original subclause. For example, a system type extending another system type by adding a port, may add error propagation declarations for the added port. Similarly, a system type extending another system type may change the error types being propagated on a port.

(7) For properties sections in an Error Model subclause the rules specified in AS5506B apply, i.e., for each individual property the value may be overridden according to contained property associations, the extends hierarchy, and component implementations over component types.

(8) The **use types** clause makes the namespaces of Error Type libraries accessible to the error propagations annex subclause. This allows error types and type sets as well as their aliases to be referenced from within Error Model subclauses without Error Model library qualification. Note that those namespaces are not inherited by the namespace of the Error Model subclause, i.e., will not result in potential name conflicts with error behavior events, error behavior states, etc.

(9) The **use type equivalence** clause specifies the type mappings to be used when components with error models are combined into a system, where the error models have been developed independently using separate error type libraries. When consistency between outgoing and incoming error propagation types is checked along error propagation paths, the mapping is applied to the outgoing error types if they are from error type libraries different from those used for incoming error propagation types. The left-hand side of a type mapping rule is interpreted as the originating error type and the right-hand side is interpreted as the resulting equivalent error type. If type mapping in both directions is desired two mapping rules have to be defined. The **use type equivalence** clause is inherited down the component hierarchy, i.e., it is used in the context of all subcomponents when such mappings are necessary.

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

(11) The **use behavior** clause associates the referenced Error Behavior State Machine with a component. The state machine is augmented with component-specific behavior, such as transitions whose trigger condition involves incoming error propagations, or outgoing error propagations that are conditional on specific error behavior states and incoming error propagations.

(12) The **use behavior** clause causes the namespace of an Error Behavior State Machine to be included in the namespace of the Error Model subclause. This allows error behavior states, transitions, and events to be referenced without qualification. However, they may cause name conflicts with locally declared named elements, such as error flows.

(13) Property values can be associated with named elements in the Error Model subclause, such as error propagations or error events, by declaring EMV2 specific contained property associations in the **properties** section of the Error Model subclause.

(14) The property value may be specific to an error type by optionally identifying the error type as the last element in the path. This allows for error type specific property values, e.g., for different occurrence probabilities for error events of different error types.

(15) The property value may be retrieved for an Error Model element of interest without a specific error type:

- When working with an instance model, we look for a contained property association to the Error Model element of interest in the properties section of the Error Model subclause of the component instance highest in the in the instance hierarchy.
- If not found, we look for a property association that identifies the Error Model element in the properties section of the Error Model subclause of the component of interest.
- If not found and the Error Model element is defined in an Error Behavior State Machine (error event, recover event, repair event, error state, transition), we look for a property association for the element of interest in the state machine properties section.
- If not found and the Error Model element is defined in the error type library (error type, type set), we look for a property association in the properties section of the error type library that defines the error type of interest.

(16) The property value may be retrieved for an Error Model element of interest with a specific error type. These elements are error event, error state, error propagation, error source/path/sink:

- When working with an instance model, we look for a contained property association to the Error Model element and error type of interest in the properties section of the Error Model subclause of the component instance highest in the in the instance hierarchy.
- If not found, we look for a property association that identifies the Error Model element and the error type of interest in the properties section of the Error Model subclause of the component of interest.
- If not found, we look for a property association that identifies closest error (super) type that has the type of interest as subtype, or a type set that contains the error type.
- If not found, we look for a property association that identifies the Error Model element without an error type or type set.

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

(17) The property value may be specified for an error model element of a subcomponent. This allows context specific property values to be associated with error model element, e.g., hazard information that is specific to an instance of a component.

(18) Note that an error source or error sink specification references an outgoing or incoming error propagation. If a property value cannot be found for the error source or error sink an analysis may look for the property in the referenced propagation.

(19) EMV2 specific property associations may have mode specific property values. This allows for mode specific parameterization Error Model elements without requiring separate subclauses to be declared for each mode. For example, we can associate different occurrence distribution values to an error event for different (operational) modes in the core model.

(20) Error Model subclauses as a whole can be declared to be applicable to specific modes by specifying them with an `in_modes` statement. An Error Model subclause without an `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.

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

(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) 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`.

(4) An *error type set* is defined as a set of unique elements, i.e., single instances of error types and error type products. For example, an error type set may be defined as consisting of the elements *BadValue* and *LateDelivery*, expressed as `{OutOfRange, LateDelivery}`.

(5) When an error type set is listed as an element of another type set, its elements are included in the other type set. If two type sets are listed as elements of another type set the resulting type set effectively represents the union of the two type sets.

(6) Error types can be organized into type hierarchies. Error types that are part of the same type hierarchy are assumed to not occur simultaneously, i.e., they cannot be different elements of the same error type product. For example, a service item cannot be early and late at the same time. Therefore, `EarlyDelivery` and `LateDelivery` are defined as subtypes of `TimingError`.

(7) When an error type that has subtypes is listed as an element of an error type set, then all of its subtypes are included in the type set. In other words, an error type with subtypes acts like a type set, whose elements are the subtypes.

(8) When an error type with subtypes is listed as an element of a type product, each of the subtypes is combined with the remaining elements of the type product. If several elements of the type product are types with subtypes, all combinations of subtypes from each type hierarchy are considered.

(9) 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.

(10) 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.

(11) 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.

Syntax

```
error_type_library ::=
  error types
  [ extends error_type_library_list with ]
    { error_type_library_element }+
  [ properties
    { error_type_emv2_contained_property_association }+ ]
  end types;

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_product ::=
  error_type_reference ( * 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 ::=
   { type_set_element ( , type_set_element )* }
```

```
type_set_element ::=
  error_type_or_set_reference | error_type_product

error_type_set ::=
   error_type_set_constructor

error_type_set_or_noerror ::=
   error_type_set_constructor | { noerror }

error_type_or_set_reference ::=
   error_type_set_reference | error_type_reference

error_type_reference ::=
    [ error_model_library_reference :: ] error_type_identifier

error_type_set_reference ::=
    [error_model_library_reference :: ] error_type_set_identifier

target_error_type_instance ::=
  { error_type_or_set_reference | error_type_product }
```

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 is identified by the name of the Error Model library that contains the error type declarations, i.e., by the package name of the package that contains the Error Model library.

(N3) 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. The inherited identifiers from different Error Type libraries must not conflict with each other.

(N4) 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.

(N5) 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.

(N6) An error type reference in an Error Type library must exist in the namespace of the Error Model library containing the reference if it is not qualified.

(N7) The error type set reference in an Error Type library must exist in the namespace of the Error Model library containing the reference if it is not qualified.

(N8) Error type and type set and their alias references that are qualified with an Error Model library do not have to exist in the namespace of the Error Type library containing the reference, i.e., the qualifying Error Model library does not have to be part of the **extends** list. They must exist in the error type library identified by the qualifying Error Model library.

(N9) 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, i.e., its package name does not have to be listed in the **with** clause of the package containing the reference.

(N10) Error type, type set, and their alias reference from within an Error Model subclause must not be qualified. Their names are made accessible through a **use types** clause (see Section Annex E.4).

Legality Rules

(L1) 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.

(L2) An error type cannot be the subtype of more than one other error type. This is enforced syntactically.

(L3) For different elements in a type set that reference a single error type, one must not be a subtype of another.

(L4) Different element types of an error type product must not be from the same error type hierarchy.

(L5) 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.

(L6) The elements of a type set must be unique.

(L7) If two elements of a type set 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 of root type.

Semantics

(12) 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 must be qualified with 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.

(13) 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.

(14) An Error Type library can define aliases or subtypes for error types and type sets in another Error Type library without including the namespace of the original Error Type library. This is done by qualifying the referenced error type or type set and by not including the Error Type library of the referenced error type or type set in the **extends** clause. When such an Error Type library is identified in a **use types** clause, only the identifiers of that library are made accessible to a subclause.

(15) An *error type* is used to indicate the type of an error event, an error flow, an incoming or outgoing error propagation, or an error behavior state.

(16) 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 unique, i.e., each error type or type product is contained only once. If one of the elements in an error type set is another type set, its elements become part of the type set with the reference. In other words, type sets can be combined resulting in a union of error types and type products.

(17) 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.

(18) 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.

(19) An instance of an error type, type set, or error type product (*type instance*) represents an error event or error propagation occurrence, or the error type of the current error state, if typed. For type sets and error types with subtypes this instance represents any of the elements or subtypes. For example, an error propagation may be defined to propagate `ValueError` and `ValueError * TimingError.` A particular error propagation instance may be an out of range value that is on time, expressed as single-valued type `OutOfRange,` or an out of range value that is also late, expressed as a two-valued type product `OutOfRange * LateValue`.

(20) If an element of an error type set is an error type product of two or more (k) error types, then represents k-valued product of error types. If the element types of an error type product have subtypes, then a product type instance exists 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 Annex E.6). Examples of a specific instance of a propagation are `OutOfRange * LateDelivery`, `OutOfRange * EarlyDelivery`, or `SubtleValueError * LateDelivery`.

(21) An error type set acts as a constraint on 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 type instance of an actual error event is contained in the specified error type set.

(22) An error event or error flow source is a source of errors. A type instance representing this error 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 type instance is generated with the specified probability.

(23) 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.

(24) 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.

(25) 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

(26) 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:

(27) $t_1 < t$ indicates that t_1 is declared as a direct or indirect subtype of a given type t. $t_1 \leq t$ indicates that t_1 is a subtype of t or the same as t.

(28) $r(t_1)$ identifies the root type of type t_1, i.e., $t_1 \leq r(t_1) \wedge \nexists\, t \mid r(t_1) < t$.

(29) Let T_1 be the types that are subtype of a given type t_1, i.e., $\forall\ t_i \in T_1 \mid t_i \leq t_1$. Similarly for T_2 and t_2, $\forall\ t_j \in T_1 \mid t_j \leq t_1$. T_1 is *contained* in T_2 ($T_1 \subseteq T_2$), iff $\forall\ t_i \in T_1 \mid t_i \in T_2$. This is condition is satisfied iff $t_1 \leq t_2$.

(30) Let U be an error type product $u_1 * u_2 * . * u_n$. U is a *type product* of simultaneously occurring error types if-and-only-if each element of U is in a separate type hierarchy; Product(U) iff $\forall\ u_j\ u_k \in U \mid r(w_{k)} \neq r(u_{j)}$.

(31) Let U be an error type product $u_1 * u_2 * . * u_n$. Let W be an error type product with the same number of elements, $w_1 * w_2 * . * w_n$. W is *contained* in of U, $U \subseteq W$ if-and-only-if each element of W is a subtype of the corresponding element in U; $U \subseteq W$ iff $\forall\ u_j \in U\ \exists\ w_k \in W \mid w_k \leq u_j$.

(32) Let E be a type set element, error type or type product, and TS be a type set. E is *contained* in TS, ($E \subseteq TS$) if-and-only-if E is a subtype of some element of S; $E \subseteq TS$ iff $\exists\ ts_i \in TS \mid E \leq ts_i$.

(33) Type set TS is a *type set* of unique (mutually exclusive) elements if-and-only-if no element is contained in another element; TypeSet(TS) iff $\forall\ E_j\ E_k \in TS \mid E_j \not\subset E_k \wedge E_k \not\subset E_j$.

(34) Type set TS_1 is *contained* in type set TS_2 ($TS_1 \subseteq TS_2$), if-and-only-if for every element in TS_1 there is some element in TS_2 that contains it; $TS_1 \subseteq TS_2$ iff $\forall\ ts_1 \in TS_1\ \exists\ ts_2 \in TS_2 \mid ts_1 \subseteq ts_2$.

Annex E.6 A Common Set of Error Propagation Types

(1) This section introduces a common set of error types as a standard Error Type library called *ErrorLibrary*. 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.

(2) 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.

(3) 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.

(4) 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.8.3). 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.

(5) 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.

(6) First we define a model of service delivered by a component through its features or via bindings 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) A *service* S is defined as a sequence of *n service items* s_i. A service item characterized by a pair (v_i , δ_i) where v_i is the value or content of service item s_i and δ_i is the delivery time of service item s_i or the empty service item, ε, which has no value nor duration.

(8) A service item is defined to be *correct*, i.e., have no error, iff: $(v_i \in V_i) \wedge (\delta_i \in D_i)$ where V_i and D_i are respectively the correct range of values and range of delivery times for service item s_i. V_i and D_i are ranges to represent value and timing tolerance

(9) For many systems, the specified value and time sets are reduced to the special cases $V_i = \{v_i\}$ (a single value) and $D_i = \{[\min(\delta_i), \max(\delta_i)]\}$ (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.

(10) Let V represent expected range (or set) of possible values delivered by a service, i.e., $\forall i, (V_i \in V)$. Let D represent the expected duration of the service operation, i.e., $\forall i, (D_i \in D)$.

(11) We define error types with respect to the number of service items of a *service*, as *value* and *timing* errors with respect to the service as a whole, the sequence of service items, and individual service items, r*eplication* errors in terms of sets of replicated service items, and *concurrency* errors with respect service as a shared resource. The error types are grouped into separate type hierarchies that can be combined to characterize an error event, error state or error propagation.

(12) In addition to error types of the above mentioned type hierarchies, the standard Error Type library *ErrorLibrary* also contains a set of aliases for the error types.

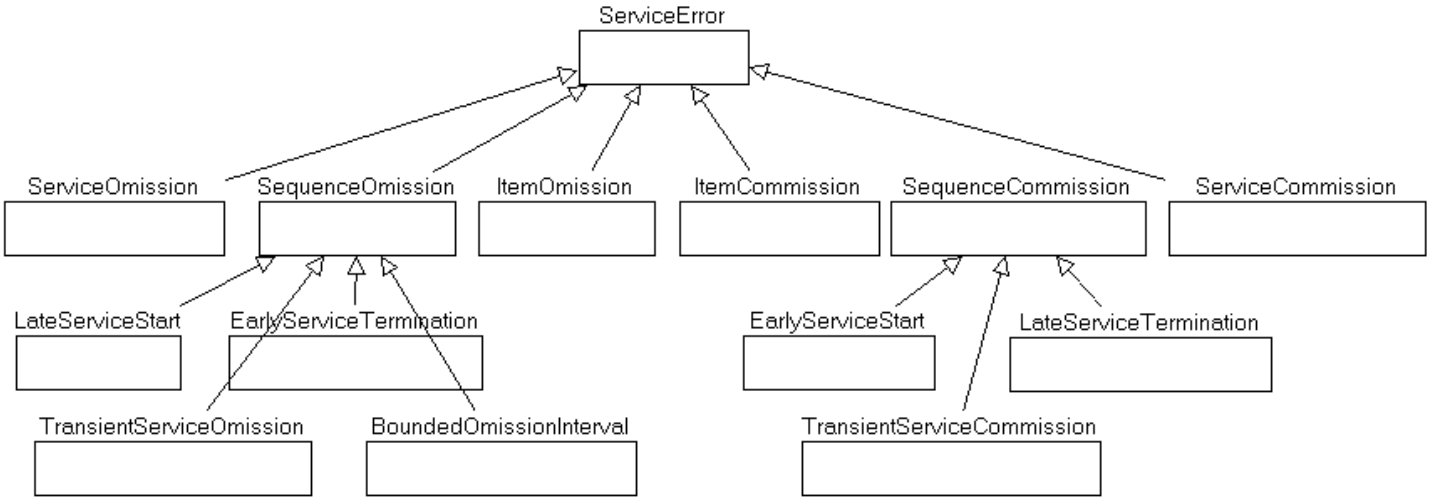

Figure 3 Service Error Type Hierarchy

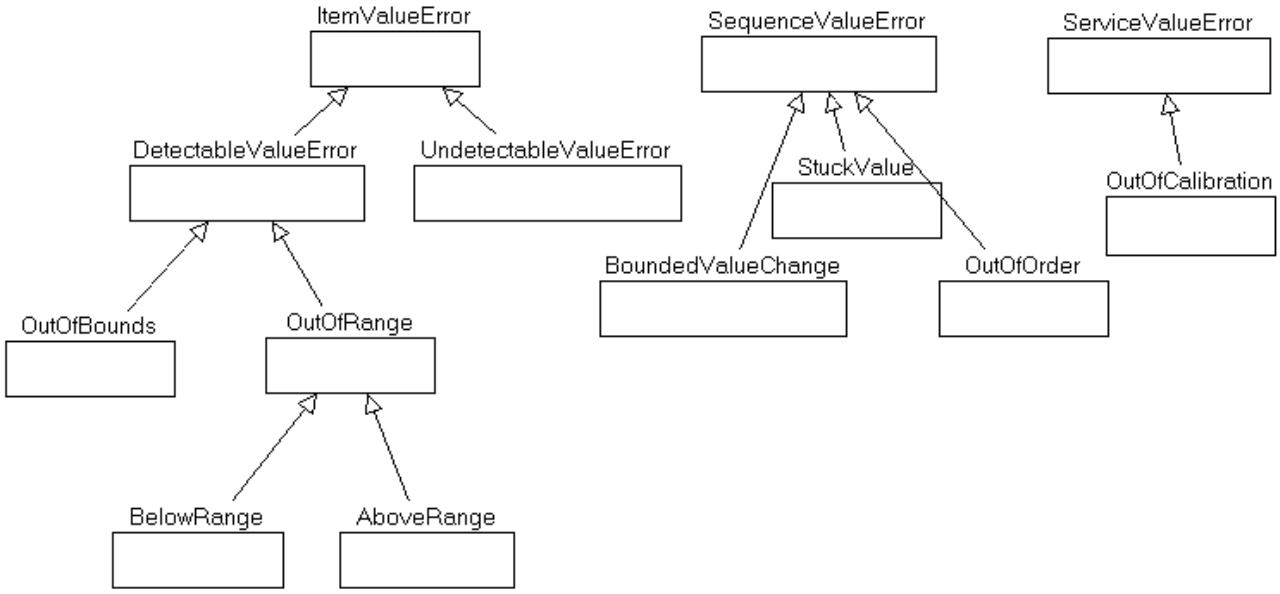

Figure 4 Value Related Error Type Hierarchies

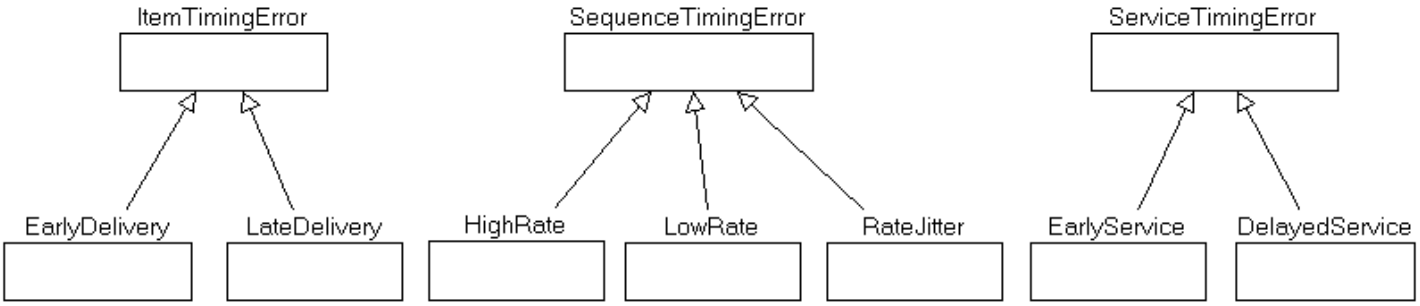

Figure 5 Timing Related Error Type Hierarchies

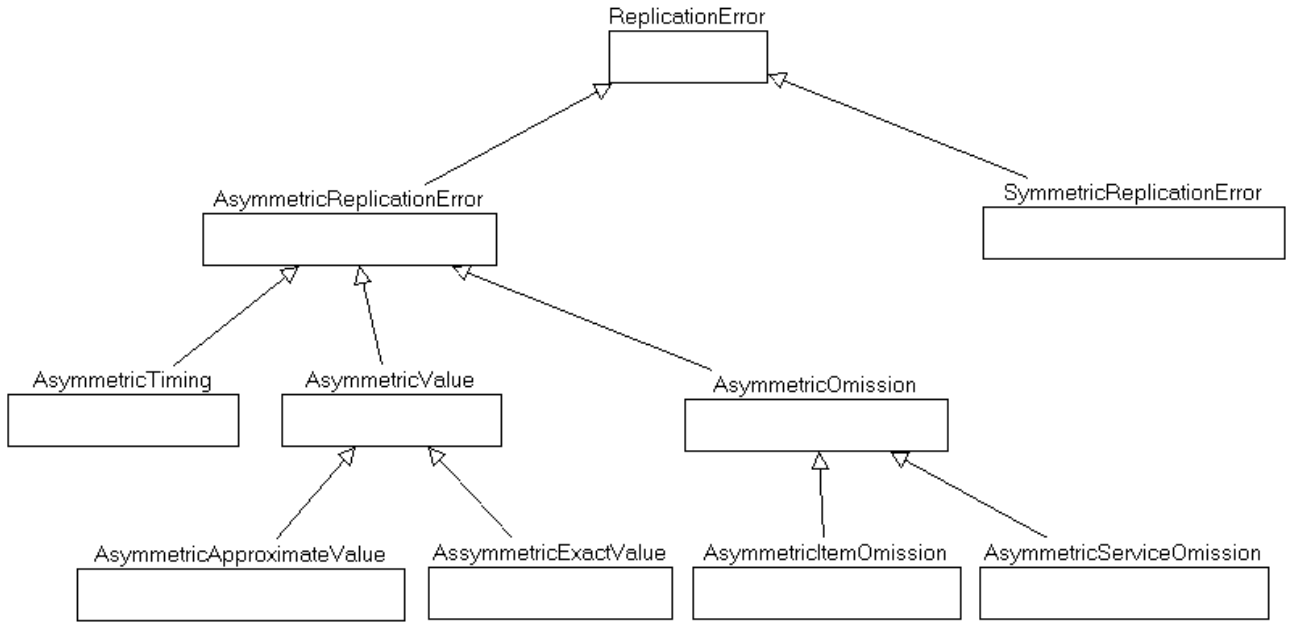

Figure 6 Replication Related Error Type Hierarchies

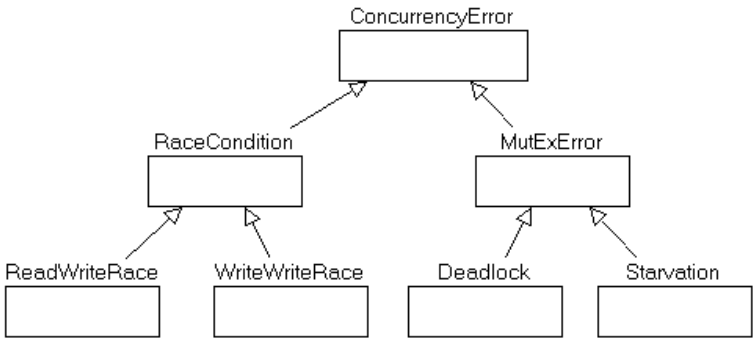

Figure 7 Concurrency Related Error Type Hierarchy

```
package ErrorLibrary
public
annex EMV2 {**
error types
CommonErrors: type set { ServiceError, TimingRelatedError, ValueRelatedError, ReplicationError,
ConcurrencyError};
--service related errors
ServiceError: type;
ItemOmission: type extends ServiceError;
ServiceOmission: type extends ServiceError;
SequenceOmission: type extends ServiceError;
TransientServiceOmission: type extends SequenceOmission;
LateServiceStart: type extends SequenceOmission;
EarlyServiceTermination: type extends SequenceOmission;
BoundedOmissionInterval: type extends SequenceOmission;
ItemComission: type extends ServiceError;
ServiceCommission: type extends ServiceError;
SequenceCommission: type extends ServiceError;
EarlyServiceStart: type extends SequenceCommission;
LateServiceTermination: type extends SequenceCommission;
```

```
--timing related errors
TimingRelatedError: type set {ItemTimingError, SequenceTimingError, ServiceTimingError};
-- Item timing errors
ItemTimingError: type;
EarlyDelivery: type extends ItemTimingError;
LateDelivery: type extends ItemTimingError;
--Rate/sequence timing errors
SequenceTimingError: type;
HighRate: type extends SequenceTimingError;
LowRate: type extends SequenceTimingError;
RateJitter: type extends SequenceTimingError;
-- Service timing error
ServiceTimingError: type;
DelayedService: type extends ServiceTimingError;
EarlyService: type extends ServiceTimingError;

-- aliases for timing errors
TimingError renames type ItemTimingError; -- legacy
RateError renames type SequenceTimingError;
EarlyData renames type HighRate;
LateData renames type LowRate;
ServiceTimeShift renames type ServiceTimingError;

--value related errors
ValueRelatedError: type set {ItemValueError, SequenceValueError, ServiceValueError};
-- item value errors
ItemValueError: type;
UndetectableValueError: type extends ItemValueError;
DetectableValueError: type extends ItemValueError;
OutOfRange: type extends DetectableValueError;
BelowRange: type extends OutOfRange;
AboveRange: type extends OutOfRange;
OutOfBounds: type extends DetectableValueError;
-- sequence errors
SequenceValueError: type;
BoundedValueChange: type extends SequenceError;
StuckValue: type extends SequenceError;
OutOfOrder: type extends SequenceError;

ServiceValueError: type;
OutOfCalibration: type extends ServiceValueError;

-- Common aliases for value related errors
ValueError renames type ItemValueError;
IncorrectValue renames type ItemValueError;
ValueCorruption renames type ItemValueError;
BadValue renames type ItemValueError;
SequenceError renames type SequenceValueError;

SubtleValueError renames type UndetectableValueError;
BenignValueError renames type DetectableValueError;
SubtleValueCorruption renames type DetectableValueError;
-- Detectability (Benign/Subtle) represent a characteristic of error types

--replication errors
ReplicationError: type;
AsymmetricReplicatesError: type extends ReplicationError;
```

```
AsymmetricValue: type extends AsymmetricReplicatesError;
AsymmetricApproximateValue: type extends AsymmetricValue;
AsymmetricExactValue: type extends AsymmetricValue;
AsymmetricTiming: type extends AsymmetricReplicatesError;
AsymmetricOmission: type extends AsymmetricReplicatesError;
AsymmetricItemOmission: type extends AsymmetricOmission;
AsymmetricServiceOmission: type extends AsymmetricOmission;

SymmetricReplicatesError: type extends ReplicationError;
SymmetricValue: type extends SymmetricReplicatesError;
SymmetricApproximateValue: type extends SymmetricValue;
SymmetricExactValue: type extends SymmetricValue;
SymmetricTiming: type extends SymmetricReplicatesError;
SymmetricOmission: type extends SymmetricReplicatesError;
SymmetricItemOmission: type extends SymmetricOmission;
SymmetricServiceOmission: type extends SymmetricOmission;

-- aliases for replication
InconsistentValue renames type AsymmetricValue;
InconsistentTiming renames type AsymmetricTiming;
InconsistentOmission renames type AsymmetricOmission;
InconsistentItemOmission renames type AsymmetricItemOmission;
InconsistentServiceOmission renames type AsymmetricServiceOmission;
AsymmetricTransmissive renames type AsymmetricValue;

--concurrency errors
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 number of service items delivered by a service. We distinguish between *Service Omission* errors to represent service items not delivered, and *Service Commission* errors to represent delivery of service items that were not expected to be delivered.

(14) *Service Omission* [`ServiceOmission`] represents an error where no service items are delivered.

Service Omission: $\forall\ s_i \in S \mid s_i = \varepsilon$ where ε is the empty service item.

(15) *Item Omission* [`ItemOmission`] represents an error where a single service item is not delivered.

Item Omission: $\exists\ s_i \in S \mid s_i = \varepsilon$ where ε is the empty action.

(16) *Bounded Omission Sequence* [`BoundedOmissionSequence`] represents an error where a certain number of consecutive service item omissions occur. A parameter *k* specifies the number of consecutive item omissions. For example, the CRC on satellite transmission allows some lost packets, but beyond the limit of the CRC, further packet loss causes loss of communication.

Bounded Omission Sequence error: $\exists\ [s_i\ s_{i+k-1}] \subset S \mid\ \forall\ s_j \in [s_i\ s_{i+k-1}]\ \mid s_j = \varepsilon$.

NOTE: Often ServiceOmission detection is based on a bounded sequence omission condition.

(17) *Late Service Start* [`LateServiceStart`] represents an error where no service items are provided for a period at the beginning of the service. The first service item is s_i.

Late Service Start: $\exists i \mid \forall j < i \mid s_j = \varepsilon$

(18) *Early Service Termination* [`EarlyServiceTermination`] represents an error where no service items are provided after service item s_i. This may represent permanent item omission due to a failure.

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

(19) *Transient Service Omission* [`TransientServiceOmission`] represents an error where a certain number of consecutive service item omissions occur before delivery of service items resumes. A parameter *k* specifies the number of consecutive item omissions. This represents transient item omission sequences.

Transient Omission Sequence error: $\exists [s_i\ s_{i+k}] \subset S \mid \forall s_j \in [s_i\ s_{i+k-1}] \mid s_j = \varepsilon \wedge s_{i+k} \neq \varepsilon$

(20) A *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: $\forall s_i \in S\ \exists [s_i\ s_{i+k}] \subset S \mid s_i = \varepsilon \wedge \forall s_j \in [s_i\ s_{i+k}]\ s_j \neq \varepsilon$.

(21) *Item Commission* [`ItemCommission`] represents an error where an extra service item is provided that is not expected.

Item Commission: $\exists s_i \notin S \mid s_i \notin \varepsilon$

(22) *Early Service Start* [`EarlyServiceStart`] represents an error where extra service items are provided for a period before the beginning of the expected service.

Early Service Start: $\exists s_i \notin S \mid \delta_i < D$

(23) *Late Service Termination* [`LateServiceTermination`] represents an error where extra service items are provided after the end of a service period.

Late Service Termination: $\exists s_i \notin S \mid \delta_i > D$

NOTE: A time related error for a service is when the service is time shifted, i.e., head start or delay.

E.6.2 Value Related Errors

(24) Value related errors deal with the value domain of a service. We distinguish between value errors of individual service items [`ItemValueError`], value errors that relate to the sequence of service items [`SequenceValueError`], and value errors related to the service as a whole [`ServiceValueError`]. Each is the root of a separate type hierarchy allowing us to characterize them independently, e.g., to specify that we have a `BoundedValueChange` error that may be `OutOfRange`. Note that both sequence and service value errors imply item value errors. Therefore, the type `ItemValueError` represents individual service item value errors that are singletons, i.e., not already covered by `SequenceValueError` and `ServiceValueError`.

(25) *Item Value Error* [`ItemValueError`] represents any kind of erroneous value for an individual service item.

Item Value Error: $\exists s_i \in S \mid v_i \notin V_i$

(26) We distinguish between detectable and undetectable item value errors.

(27) *Detectable Value Error* [`DetectableValueError`] is detectable from the value itself, perhaps because it's out of range or has parity error. Let predicate B represent detection of a value error, B(v).

Detectable Value Error: $\exists\ a_i \in A \mid v_i \notin V_i \wedge B(v_i)$

(28) *Undetectable Value Error* [`UndetectableValueError`] occurs when is not a correct value as perceived by an omniscient observer, but cannot be recognized based on available information. Such errors require additional contextual information to become detectable.

Undetectable Value Error: $\exists\ a_i \in A \mid v_i \notin V_i \wedge \neg B(v_i)$

(29) In the case of value errors the CFEM framework [Walter 2003] distinguishes between Subtle and Benign value errors. 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 benign value error is an *Out Of Range* error. Aliases have been defined to equate benign with detectable and subtle with undetectable. We have also introduced a property that allows the user to characterize the detection mechanism used by the system to detect different error types.

(30) We distinguish between the following detectable item value errors: *Out Of Range* error with two subtypes *Below Range*, and *Above Range, Out Of Bounds, and Incorrect Value*.

(31) *Out Of Range* [`OutOfRange`] represents an error where 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: $\exists\ s_i \in S \mid v_i < \min(V)$

Above Range error: $\exists\ s_i \in S \mid v_i > \max(V)$

Out Of Range error: $\exists\ s_i \in S \mid v_i > \max(V)$ OR $v_i < \min(V)$

(32) *Out Of Bounds* [`OutOfBounds`] represents an error where 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: $\exists\ s_i \in S \mid v_i \notin V_i \wedge O(v_i)$

(33) *Value Corruption* [`ValueCorruption`] error results from erroneous behavior of 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.

(34) *Sequence Value Error* [`SequenceValueError`] represents value errors related to the sequence of service items.

(35) *Bounded Value Change* [`BoundedValueChange`] represents an error where a service delivers service items whose value changes by more than an expected value.

Bounded Value Change error: $\exists\ s_i \in S \mid abs(v_i - v_{i-1}) > C$ where C is the maximum expected value change between two consecutive service items, and abs is absolute value.

(36) *Stuck Value* [`StuckValue`] represents an error where a service delivers service items whose value stays constant starting with a given service item.

Stuck Value error: $\exists\ s_i \in S \mid \forall j > i: v_j = v_i$

(37) *Out Of Order* [`OutOfOrder`] represents errors where a service delivers a service item in a time slot D_j other than its expected time-slot.

Out Of Order error: $\exists\ s_i \in S \mid \delta_i = D_j$ for $i \neq j$

(38) *Service Value Error* [`ServiceValueError`] represents value errors related to the service as a whole. *Out Of Calibration* is such an error.

(39) *Out Of Calibration* [`OutOfCalibration`] represents an error where the actual values of a sequence differ by more than a tolerance but roughly constant offset C from the correct value.

Out Of Calibration error: $\forall s_i \in S \mid v_i \notin V_i \wedge (C+v_i) \in V_i$

E.6.3 Timing Related Errors

(40) Timing related errors deal with the time domain of a service. We distinguish between timing errors of individual service items [`ItemTimingError`], timing errors that relate to the sequence of service items [`SequenceTimingError` or `RateError`], and timing errors regarding the service as a whole [`ServiceTimingError`]. Each is the root of a separate type hierarchy allowing us to characterize them independently, e.g., to specify that we have a time shifted service executing at the wrong rate. Item timing errors and sequence timing errors refer to a timeline with respect to service start time, while service timing errors are use to clock time as reference time. Therefore, service timing errors are independent of the other two, while sequence timing errors imply item timing errors. Therefore, `ItemTimingError` represent singleton item timing errors that are not covered by `SequenceTimingError`.

(41) *Item Timing Error* [`ItemTimingError`] represents errors where a service item being delivered outside its expected time range D_i of service item s_i.

Item Timing error: $\exists\ s_i \in S \mid \delta_i \notin D_i$

(42) General timing errors are distinguished as: *Early Delivery* error, *Late Delivery* error.

(43) *Early Delivery* [`EarlyDelivery`] represents errors where a service item 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: $\exists\ s_i \in S \mid \delta_i < D_i$

(44) *Late Delivery* [`LateDelivery`] represents errors where a service item 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: $\exists\ s_i \in S \mid \delta_i > D_i$

(45) *Sequence Timing Error* [`SequenceTimingError`] or its alias *Rate Error* [`RateError`] represents errors with respect to the inter-arrival time of service items, i.e., the time interval between deliveries of successive service items. The inter-arrival time for service item s_i is defined as $r_i \in R_i \mid r_i = \delta_i - \delta_{i-1}$. Let R represent the expected inter-arrival time, i.e., $\forall i, (R_i \in R)$. Many periodically sampling systems operate at a fixed inter-arrival time r, such that $R = \{r\}$ and $R_i = R$. In this case we have $\forall i \mid r - r_i \mid > x \Rightarrow r_i \notin R$. Acceptable variation in the inter-arrival time is expressed by Δr such that $R = [r-\Delta r\ r+\Delta r]$.

Sequence Timing error: $\exists\ s_i \in S \mid r_i \notin R_i$

(46) *High Rate* [`HighRate`] represents errors where the inter-arrival time of all service items is less than the expected inter-arrival time.

High Rate error: $\forall s_i \in S \mid r_i < R_i$

(47) *Low Rate* [`LowRate`] represents errors where the inter-arrival time of all service items is greater than the expected inter-arrival time.

Low Rate error: $\forall s_i \in S \mid r_i > R_i$

(48) *Rate Jitter* [RateJitter] represents errors where a service delivers service items at a rate that varies from the expected rate by more than an acceptable tolerance.

Rate Jitter error: $\exists\, s_i \in S \mid r_i \notin R$

(49) *Service Timing error* [ServiceTimingError] with alias *Service Time Shift* [ServiceTimeShift] represents errors where a service delivers all service items time shifted by a time constant TD, but otherwise correctly. NOTE: individual items may be correct with respect to the start time.

Service Time Shift error: $\forall\, s_i \in S \mid (\delta_i + TD) \in D_i$

(50) *Delayed Service* [DelayedService] represents errors where a service delivers all service items late with a constant time delay TD, but otherwise correctly. NOTE: individual items may be correct with respect to the start time.

Delayed Service error: $\forall\, s_i \in S \mid (\delta_i + TD) \in D_i \ \wedge\ TD > 0$

(51) *Early Service* [EarlyService] represents errors where a service delivers all service items early with a constant time shift TD, but otherwise correctly. NOTE: individual items may be correct with respect to the start time.

Early Service error: $\forall\, s_i \in S \mid (\delta_i - TD) \in D_i \ \wedge\ TD < 0$

E.6.4 Replication Related Errors

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

(53) A *replicated service item* is a set of replicates of one service item that are supposed to be the same. In the case with n-way replication, let $s_i = \{s_i(1),..., s_i(n)\}$, $i = 1,2,...$ where V_i and D_i are respectively the correct sets of values and delivery times for service item s_i. An individual replicate of a service item, $s_i(k)$, has the value $v_i(k)$ and delivery time $\delta_i(k)$. In a non-failed replication system we expect $\forall k \in [1,n] \mid v_i(k) \in V_i \wedge \delta_i(k) \in D_i$.

(54) Replication errors [ReplicationError] represent errors in the set of replicates. We distinguish between Asymmetric and Symmetric Replicates errors. Asymmetric Replicates error means the replicates are inconsistent with each other. Replicates may be inconsistent because some are correct with respect to value or time while others are not. The incorrect items may represent the same or different instances of an error occurrence. Symmetric Replicates error means that the replicates reflect a single error ocurrence.

(55) We distinguish between Asymmetric/symmetric Value errors, Asymmetric/symmetric Omission errors, and Asymmetric/symmetric Timing errors.

NOTE: Symmetric and asymmetric replicates errors are part of the CFEM framework [Walter 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.

(56) An *Asymmetric Value* [AsymmetricValue] error with the alias *Inconsistent Value* [InconsistentValue] 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 ($v_i(k) \notin V_i$).

Asymmetric Value error: $\exists\, s_i \in S \mid \exists\ k \in [1,n] \mid v_i(k) \notin V_i$

(57) We distinguish between *Asymmetric Exact Value* [AsymmetricExactValue] errors and *Asymmetric Approximate Value* [AsymmetricApproximateValue] errors. In the case of asymmetric exact value, the error occurs if the value comparison does not show identical values. In the case of asymmetric approximate value, the error occurs if the values in the comparison differ by more than a threshold. The threshold is defined in terms of a delta from a reference value. Let h be the threshold for approximate inconsistency.

Asymmetric Approximate Value error: $\exists\ s_i \in S \mid \exists\ k \in [1,n] \mid v_i(k) > V_i + h \vee v_i(k) < V_i - h$

(58) An *Asymmetric Omission* [AsymmetricOmission] error with alias *Inconsistent Omission* [AsymmetricOmission] 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.

Asymmetric Omission error: $\exists\ s_i \in S \mid \exists\ k \in [1,n] \mid a_i(k) = \varepsilon$

(59) An *Asymmetric Timing* [AsymmetricTiming] error with alias *Inconsistent Timing* [InconsistentTiming] error occurs when at least one of the replicated service items is delivered outside the expected time interval. Let Δ represent the maximum expected time variation of the replicates.

Asymmetric Timing error: $\exists\ s_i \in S \mid \exists\ j,k \in [1,n],\ \delta_i(j) - \delta_i(k) > \Delta$

(60) A *Symmetric Value* [SymmetricValue] error occurs when the value of all replicated service items have the same value and are incorrect.

Symmetric Value error: $\exists\ s_i \in S \mid \forall\ j,k \in [1,n] \mid v_i(j) = v_i(k) \wedge v_i(j) \notin V_i$

(61) A *Symmetric Omission* [SymmetricOmission] error occurs when all 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.

Symmetric Omission error: $\exists\ s_i \in S \mid \forall\ k \in [1,n] \mid a_i(k) = \varepsilon$

(62) A *Symmetric Timing* [SymmetricTiming] error occurs when all replicated service items is delivered outside the expected time interval. Let Δ represent the maximum expected time variation of the replicates.

Symmetric Timing error: $\exists\ s_i \in S \mid \forall\ k \in [1,n],\ \notin D_i$

(63) 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.

NOTE: a replicate set may have more than two elements. Do we care to distinguish between MofN error? Or are we talking about a fault tolerance condition? What does 2of3 mean? Do the two reflect a symmetric error in terms of the error type or in terms of the actual value?

E.6.5 Concurrency Related Errors

(64) 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::RateJitter ;
end types;
**};
end ProcessorErrors;
```

Annex E.7 Predeclared Error Model Properties and Analyses

(65) 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.

E.7.1 Descriptive and Stochastic Error Model Properties

Properties

(66) The *Description* property allows the user to attach descriptive information with error model elements.

```
Description : aadlstring
      applies to (all);
```

(67) The *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.

```
StateKind : EMV2::StateKindEnum
      applies to ({emv2}**error behavior state);

StateKindEnum: type enumeration (Working, NonWorking);
```

(68) The *DetectionMechanism* property allows the user to specify the detection mechanism used to detect an error.

```
DetectionMechanism : aadlstring
         applies to ({emv2}**error detection);
```

(69) The *FaultKind* 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.

```
FaultKind : EMV2::FaultKindEnum
          applies to ({emv2}**error event, {emv2}**error propagation, {emv2}**error
source, {emv2}**error type, {emv2}**type set);
FaultKindEnum: type enumeration (Design, Operational);
```

(70) The *Persistence* property allows the user to specify whether an error is *permanent*, *transient*, or a *singleton*. A permanent error typically requires a repair action (see repair event in Section Annex E.9) to the component with the fault occurrence. A transient error has a limited duration and typically requires a recovery action (see recover event in Section Annex E.9). In a discrete event system a transient error may last over several discrete events, e.g., a corrupted message may be sent over multiple periods. A singleton error occurs in the context of a single discrete event. For example, a divide by zero error may be specific to a computation on a particular input.

(71) Persistence may be an inherent property of an error type, e.g., ServiceOmission is considered a permanent error. All errors from a particular error source may have a specific persistence character, e.g., a component may be fail silent and only propagate ServiceOmission. Persistence may also be the property of a particular error instance represented by a type token. The persistence value may be determined by the state of a error behavior state machine and a transition branch probability may determine whether the persistence value is *Permanent* or *Transient* (see Section E.9.2).

```
Persistence : EMV2::PersistenceEnum
          applies to ({emv2}**error event, {evm2}**error propagation, {emv2}**error type,
{emv2}**type set);
PersistenceEnum: type enumeration (Permanent, Transient, Singleton);
```

(72) The next properties support stochastic modeling and analysis.

(73) An *OccurrenceDistribution* 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.

```
OccurrenceDistribution: EMV2::ProbabilityDistributionSpecification
    applies to ( {emv2}**error behavior event, {emv2}**error propagation, {emv2}**error
flow, {emv2}**error behavior state, {emv2}**error type, {emv2}**type set);

-- reusable property type to introduce stochastic properties of this type
ProbabilityDistributionSpecification : type record (
        OccurrenceRate : aadlreal;
        MeanValue : aadlreal;
        StandardDeviation : aadlreal;
        ShapeParameter : aadlreal;
        ScaleParameter : aadlreal;
        SuccessCount : aadlreal;
        SampleCount : aadlreal;
        Probability : aadlreal;
        Distribution : EMV2::DistributionFunction;
);
DistributionFunction : type enumeration (Fixed, Poisson, Exponential, Normal, Gauss,
Weibull, Binominal);
```

(74) *DistributionFunction* property type provides a predefined set of distribution functions.

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

- *Exponential* represents an exponential distribution and takes a single parameter *OccurrenceRate*.
- *Normal* aka. *Gauss* represents a distribution with an explicitly specified *MeanValue* and *StandardDeviation*.
- *Weibull* represents a shaped distribution with a *ShapeParameter* and a *ScaleParameter*.
- *Binominal* represents a discrete distribution with a *SuccessCount*, a *SampleCount*, and a *Probability* parameter.

(75) A *PropagationTimeDelay* property indicates the delay in propagating and error as a distribution over a time interval. For example, as a property associated with a connection it indicates the time delay of the error propagation.

```
PropagationTimeDelay: EMV2::DurationDistributionSpecification
    applies to (connection, {emv2}**propagation point connection);

DurationDistributionSpecification : type record (
Duration : Time_Range;
Distribution : EMV2::DistributionFunction; );
```

(76) The *DurationDistribution* property specifies a time range to reflect the duration as a distribution over the time range. This property can be attached to repair events to indicate the duration of the repair, once started. When applied to a recover event, it represents the duration of the recovery. Alternatively, the property can be associated with an error behavior transition that is triggered by one of these events.

```
DurationDistribution : EMV2::DurationDistributionSpecification applies to ({emv2}**repair
event, {emv2}**recover event, {emv2}**error behavior transition);
```

Semantics

(77) 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.

(78) An *occurrence distribution* 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, in terms of a distribution function. 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.

(79) An occurrence distribution 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.

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

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

(82) The occurrence distribution 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 11 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 pass through or transformed.

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

(84) When applied to error behavior events, i.e., Error Events, Recover Events, or Repair Events, the *OccurrenceDistribution* property specifies a probability according to a specified distribution according to which error behavior events are expected to occur. This property can be error type specific.

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

(86) When applied to a repair event, the *OccurrenceDistribution* property specifies when a repair is initiated. The value of this property takes into account the role of a “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”.

(87) The *DurationDistribution* 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.7.2 Hazard Related Error Model Properties

Properties

(88) The next set of properties support the characterization of components as sources of hazards. These properties are grouped into a record property called *Hazards*. The Hazards property accepts a list of records characterizing one or more hazards associated with a particular Error Model element.

(89) Severity and Likelihood are fields of the Hazards record and also available as separate properties.

(90) The Hazards property is defined in the property set EMV2. In addition, tailored versions of the Hazards property are defined in the property set ARP4761 and MILSTD882 (see below). Users can utilize the appropriate subsets of fields and ARP4761 or MILSTD882 specific labels for severity and likelihood, or they can use the tailored versions of the Hazards property.

(91) The EMV2 defined Hazards property is defined as follows:

```
Hazards: list of record
  (
   CrossReference : aadlstring;   -- cross reference to an external document
   HazardTitle : aadlstring;      -- short descriptive phrase for hazard
   Description : aadlstring;      -- description of the hazard (same as hazardtitle)
   Failure : aadlstring;          -- system deviation resulting in failure effect
   FailureEffect : aadlstring;    -- description of the effect of a failure (mode)
   Phases : list of aadlstring;   -- operational phases in which the hazard is relevant
   Environment : aadlstring;      -- description of operational environment
   Mishap : aadlstring;           -- description of event (series) resulting in
                                  -- unintentional death, etc.(MILSTD882)
   FailureCondition : aadlstring; -- description of event (series) resulting in
                                  -- unintentional death, etc.(ARP4761)
   Risk : aadlstring;             -- description of risk. Risk is characterized by
                                  -- severity, likelihood, and occurrence probability
   Severity : EMV2::SeverityRange;              -- actual risk as severity
   Likelihood : EMV2::LikelihoodLabels;         -- actual risk as likelihood/probability
   Probability: EMV2::ProbabilityRange;              -- probability of a hazard
   TargetSeverity : EMV2::SeverityRange;        -- acceptable risk as severity
   TargetLikelihood : EMV2::LikelihoodLabels;   -- acceptable risk as likelihood/prob
   DevelopmentAssuranceLevel : EMV2::DALLabels; -- level of rigor in development
                                                -- assurance (ARP4761)
   VerificationMethod : aadlstring; -- verification method to address the hazard
   SafetyReport : aadlstring;       -- analysis/assessment of hazard
   Comment : aadlstring;            -- additional information about the hazard
) applies to ({emv2}**error type, {emv2}**type set, {emv2}**error behavior state,
              {emv2}**error source, {emv2}**error propagation, {emv2}**error event);
```

```
ProbabilityRange: type aadlreal 0.0 .. 1.0;
SeverityRange: type aadlinteger 1 .. 5;
LikelihoodLabels: type enumeration (A, B, C, D, E);
DALLabels: type enumeration (A, B, C, D, E);
```

(92) The *CrossReference* property value allows for a cross reference into an external document.

(93) The *HazardTitle* property value allows for a short descriptive phrase for the hazard. It may be the same as the FailureEffect property value.

(94) The *Description* property value allows for a textual description of the hazard. It may be the same as the HazardTitle.

(95) The *Failure* property value provides a way of capturing a description of the system deviation in behavior from its nominal specification. This will be the error which gives rise to the FailureEffect.

(96) The *FailureEffect* property value provides a way of capturing a description of the operation of a system or item as the result of a failure; i.e., the consequence(s) a failure mode has on the operation, function or status of a system or an item.

(97) The *Phases* property value indicates the operational phase (e.g., acceleration, takeoff, long-range search, etc.) in which the hazard is relevant. It accepts a list of string values.

(98) The *Environment* property value indicates the operational environment in which the hazard is relevant.

(99) The *Mishap* property value provides a way of capturing the event or series of events caused or contributed to by the hazard which result in unintentional death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment.

(100) The *FailureCondition* property value provides a way of capturing the effects caused or contributed to by the hazard on the aircraft and its occupants, both direct and consequential, caused or contributed to by one or more failures, malfunctions, external events, errors, or a combination thereof, considering relevant adverse operation or environmental conditions assessed hazard severity classification (used in ARP4761).

(101) The *Risk* property value provides a way of textually describing the potential risk of the hazard. Risk is also characterized by the Severity and Likelihood properties. Note that Likelihood implies a quantified occurrence probability. This probability can be explicitly specified by the OccurrenceDistribution property.

(102) The *Severity* property value indicates the severity of the hazard as 1 (high) to 5 (low).

(103) MIL-STD 882D uses descriptive labels (Catastrophic, Critical, Marginal, Negligible), which are defined as property constants in the property set *MILSTD882* as shown in Figure 8 for use in the EMV2 definition of Hazards. Note that in MIL-STD 882D the numeric values for Severity are referred to as *SeverityCategory*, while the labels are referred to as *Severity Level*.

(104) SAE ARP 4761 defines descriptive labels (Catastrophic, Hazardous, Major, Minor, NoEffect), which are defined as property constants in the property set *ARP4761* as shown in Figure 9 for use in the EMV2 definition of Hazards.

(105) The *Likelihood* property value indicates the likelihood with which the hazard occurs. Likelihood is expressed in terms of qualitative levels ranging from A (high) to E (low). Each level typically has an associated probability of occurrence (p) threshold.

(106) MIL-STD 882D uses the term *ProbabilityLevel* for the levels A – E and *QualitativeProbability* with descriptive labels and associated quantitative probabilities (*Frequent*: p>10e-1, *Probable*: 10e-1 > p > 10e-2, *Occasional*: 10e-2 > p > 10e-3, *Remote*: 10e-3 > p > 10e-6, *Improbable*: p < 10e-6) for probability of occurrence over the life of an item. Figure 8 shows the property constants for the descriptive labels that can be used for the EMV2 definition of Hazards.

(107) In the context of ARP 4761 or DO178 uses the term *QualitativeProbability* with descriptive labels and associated quantitative probabilities (*Probable*: p > 10e-5, *Remote*: 10e-5 > p > 10e-7, *ExtremelyRemote*: 10e-7 < p > 10e-9,

ExtremelyImprobable: p < 10e-9) for probability of occurrence per operational hour. Figure 9 shows the property constants for the descriptive labels that can be used for the EMV2 definition of Hazards.

(108) The *Probability* property specifies the occurrence probability of a hazard as a value between zero and one.

(109) The *TargetSeverity* and *TargetLikelihood* properties provide a way of capturing the target risk for the hazard.

(110) The *DevelopmentAssuranceLevel* property value provides a way of capturing the development assurance levels required in ARP4761 to mitigate development errors.

(111) The *VerificationMethod* property value provides a way of textually describing the verification method used to address the hazard.

(112) The *Comment* property value allows for additional information about the hazard.

(113) The *Severity*, *Likelihood*, and *Hazards* 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.

(114) There is a one-to-one correspondence between the ARP4761 labels for Severity and the values of the target likelihood and development assurance level property values. Such a correspondence represents the target likelihood and development assurance level assigned to the severity classifications by ARP4761. For example, for the Catastrophic severity classification, ARP4761 assigns the Extremely Improbable likelihood implying the occurence probability objective of 1.0X10-9 per operational hour and a development assurance level of A.

(115) The MILSTD882 property set defines a tailored version of the Hazards property with fields aligning with MIL-STD 882D – shown in Figure 8.

```
property set MILSTD882 is
  with EMV2;
-- Severity labels: Can be used with EMV2::Hazards and Severity
Catastrophic   : constant EMV2::SeverityRange => 1;
Critical   : constant EMV2::SeverityRange => 2;
Marginal   : constant EMV2::SeverityRange => 3;
Negligible : constant EMV2::SeverityRange => 4;

-- Likelihood labels: Can be used with EMV2::Hazards and Likelihood
Frequent    : constant EMV2::LikelihoodLabels => A;
Probable    : constant EMV2::LikelihoodLabels => B;
Occasional  : constant EMV2::LikelihoodLabels => C;
Remote      : constant EMV2::LikelihoodLabels => D;
Improbable  : constant EMV2::LikelihoodLabels => E;

SeverityLabels: type enumeration (Catastrophic, Critical, Marginal, Negligible);
SeverityRange: type aadlinteger 1 .. 4;

ProbabilityLabels: type enumeration (Frequent, Probable, Occasional, Remote, Improbable);
ProbabilityLevelLabels: type enumeration (A, B, C, D, E);

Hazards: list of record
  (
   CrossReference : aadlstring;   -- cross reference to an external document
   HazardTitle : aadlstring;      -- short descriptive phrase for hazard
   Description : aadlstring;      -- description of the hazard (same as hazardtitle)
   Failure : aadlstring;          -- system deviation resulting in failure effect
   FailureEffect : aadlstring;    -- description of the effect of a failure (mode)
   Phases : list of aadlstring;   -- operational phases in which the hazard is relevant
   Environment : aadlstring;      -- description of operational environment
   Mishap: aadlstring;           -- description of event (series) resulting in
                                 -- unintentional death, etc.(MILSTD882)
```

```
   Risk: aadlstring;             -- description of risk. Risk is characterized by
                                 -- severity, likelihood, and occurrence probability
   SeverityLevel: MILSTD882::SeverityLabels;         -- actual risk as severity level
   SeverityCategory: MILSTD882::SeverityRange;       -- equivalent severity category
   QualitativeProbability: MILSTD882::ProbabilityLabels; -- actual risk as probability
   ProbabilityLevel: MILSTD882::ProbabilityLevelLabels;  -- equivalent probability level
   QuantitativeProbability: EMV2::ProbabilityRange;          -- probability of a hazard
   TargetSeverityLevel: MILSTD882::SeverityLabels;    -- target severity
   TargetProbabilityLevel: MILSTD882::ProbabilityLevelLabels; -- target probability level
   VerificationMethod: aadlstring; -- verification method to address the hazard
   SafetyReport: aadlstring;       -- analysis/assessment of hazard
   Comment: aadlstring;            -- additional information about the hazard
   )
     applies to ({emv2}**error type, {emv2}**type set, {emv2}**error behavior state,
                 {emv2}**error source, {emv2}**error propagation, {emv2}**error event);
end MILSTD882;
```

Figure 8 MIL STD 882D Specific Property Set

(116) The MIL-STD-882 property set contains two property values that provide a way of capturing the assessed mishap severity, *SeverityLevel* and *SeverityCategory*. The *SeverityLevel* enumerated values are the terms that MIL-STD-882 uses to label the mishap severity descriptions using descriptive labels defined in *SeverityLabels*. The *SeverityCategory* numbers refer to the same mishap severity descriptions as the enumerated values of SeverityLevel. That is, they are two different ways of referring to the same mishap severity descriptions. (e.g., the Catastrophic SeverityLevel refers to the same mishap severity descriptions as does a SeverityCategory of 1)

(117) The MIL-STD-882 property set contains two property values that provide a way of capturing the assessed qualitative probability of a hazard, *QualitativeProbability* and *ProbabilityLevel*. The *QualitativeProbability* enumerated values – defined by *ProbabilityLabels* - are the terms that MIL-STD-882 uses to refer to the various qualitative probability descriptions. The *ProbabilityLevel* enumerated letters – defined by *ProbabilityLevelLabels* - refer to the same qualitative probability descriptions as the enumerated values of QualitativeProbability. That is, they are two different ways of referring to the same qualitative probability description. (e.g., the QualitativeProbability word value Frequent refers to the same qualitative probability description as does a ProbabilityLevel letter of A).

(118) The MIL-STD-882 property set contains the property value *TargetProbabilityLevel*, which specifies the target probability level.

(119) The MIL-STD-882 property set contains the property value QuantitiativeProbability which provides a way of capturing the quantitative probability of a hazard. QuantitativeProbability varies between zero and one.

(120) The ARP4761 property set defines a tailored version of the Hazards property with fields aligning with SAE ARP 4761 – shown in Figure 9.

```
property set ARP4761 is
with EMV2;
-- Severity labels: Can be used with EMV2::Hazards and Severity
Catastrophic   : constant EMV2::SeverityRange => 1;
Hazardous      : constant EMV2::SeverityRange => 2; -- backward compatibility
SevereMajor    : constant EMV2::SeverityRange => 2;
Major          : constant EMV2::SeverityRange => 3;
Minor          : constant EMV2::SeverityRange => 4;
NoEffect       : constant EMV2::SeverityRange => 5;

-- Likelihood labels: Can be used with EMV2::Hazards and Likelihood
Frequent           : constant EMV2::LikelihoodLabels => A;
Probable           : constant EMV2::LikelihoodLabels => B;
Remote             : constant EMV2::LikelihoodLabels => C;
ExtremelyRemote    : constant EMV2::LikelihoodLabels => D;
ExtremelyImprobable : constant EMV2::LikelihoodLabels => E;
```

```
SeverityClassification: type enumeration (Catastrophic, SevereMajor, Major, Minor,
NoEffect);

ProbabilityLabels: type enumeration (Frequent, Probable, Remote, ExtremelyRemote,
ExtremelyImprobable);


Hazards: list of record
  (
   CrossReference : aadlstring;   -- cross reference to an external document
   HazardTitle : aadlstring;      -- short descriptive phrase for hazard
   Description : aadlstring;      -- description of the hazard (same as hazardtitle)
   Failure : aadlstring;          -- system deviation resulting in failure effect
   FailureEffect : aadlstring;    -- description of the effect of a failure (mode)
   Phases : list of aadlstring;   -- operational phases in which the hazard is relevant
   Environment : aadlstring;      -- description of operational environment
   Mishap : aadlstring;           -- description of event (series) resulting in
                                  -- unintentional death, etc.(MILSTD882)
   Risk : aadlstring;             -- description of risk. Risk is characterized by
                                  -- severity, likelihood, and occurrence probability
   FailureCondition : aadlstring; -- description of event (series) resulting in
                                  -- unintentional death, etc.(ARP4761)
   FailureConditionClassification: ARP4761::SeverityClassification; -- severity
   QualitativeProbability: ARP4761::ProbabilityLabels; -- actual risk as probability
   QuantitativeProbability: EMV2::ProbabilityRange;          -- probability of a hazard
   QualitativeProbabilityObjective: ARP4761::ProbabilityLabels; -- acceptable prob
   QuantitativeProbabilityObjective: EMV2::ProbabilityRange; -- prob objective for hazard
   DevelopmentAssuranceLevel : EMV2::DALLabels; -- level of rigor in development
                                                -- assurance (ARP4761)
   VerificationMethod : aadlstring; -- verification method to address the hazard
   SafetyReport : aadlstring;       -- analysis/assessment of hazard
   Comment : aadlstring;            -- additional information about the hazard
   )
) applies to ({emv2}**error type, {emv2}**type set, {emv2}**error behavior state,
                {emv2}**error source, {emv2}**error propagation, {emv2}**error event);
end ARP4761;
```

Figure 9 SAE ARP 4761 Specific Property Set

(121) The ARP4761 property set contains the property value *FailureCondition* which provides a way of capturing the effects caused or contributed to by the hazard on the aircraft and its occupants, both direct and consequential, caused or contributed to by one or more failures, malfunctions, external events, errors, or a combination thereof, considering relevant adverse operation or environmental conditions assessed hazard severity classification.

(122) The ARP4761 property set contains the property value *SeverityClassification* which provides a way of capturing the enumerated terms that ARP4761 uses to refer to the various hazard severities.

(123) The ARP4761 property set contains the property value *FailureConditionClassification* which provides a way of capturing the assessed SeverityClassification of the FailureCondition.

(124) The ARP4761 property set contains two property values, *QualitativeProbability* and *QuantitativeProbability*, which provide a way of capturing the assessed probability of the hazard. The *QualitativeProbability* enumerated values are the labels that ARP4761 uses as probability descriptives. *QuantitativeProbability* varies between 0 and 1.

(125) The ARP4761 property set contains two property values, *QualitativeProbabilityObjective* and *QuantitativeProbabilityObjective*, which provide a way of capturing the probability objective assigned to a given hazard severity. QualititativeProbabilityObjective is of type ProbabilityLabel so its enumerated values are the same as those of QualitativeProbability. *QuantitativeProbabiltyObjective* is the set of real numbers ARP4761 has chosen for its required quantitative probability objectives.

(126) There is a one-to-one correspondence between the enumerated values of the ARP4761 property set SeverityClassification and the values of the probability objective and development assurance level property values. Such a correspondence represents the probability objective and development assurance level assigned to the severity classifications by ARP4761. (e.g., For the Catastrophic severity classification (the first enumerated value of SeverityClassification), ARP4761 assigns the Extremely Improbable qualitative probability objective (the first enumerated value of QualitativeProbabilityObjective) or a quantitative probability objective of 1.0X10-9 per operational hour (the first value in the set of values for QuantitativeProbabilityObjectieve) and a development assurance level of A (the first enumerated value of DevelopmentAssuranceLevel).)

Examples

(127) Figure 10 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. Such Hazard specifications provide as many property values from the Hazard record definition as are available or applicable to the hazard to properly characterize it. This example illustrates a MIL-STD-882 specification that is near the end of the design phase where almost all of the property values from the Hazard record definition are available. The provided property values in the Hazard specification are used in various MIL-STD-882 and ARP4761 analyses and assessments.

```
  device PositionSensor
    features
      PositionReading: out data port DataDictionary::Position;
    flows
      f1: flow source PositionReading {
        Latency => 2 ms .. 3 ms;
        };
    annex EMV2 {**
    use types ErrorLibrary;
    use behavior ErrorModelLibrary::Simple;
   error propagations
     PositionReading: out propagation  {ServiceOmission};
   flows
     ef1:error source PositionReading {ServiceOmission} when Failed;
   end propagations;
   properties
   --property is associated with the error behavior state as the source
   --it uses the generic version of the hazard property
    MILSTD882::hazards => ([
      CrossReference => "1.1.1";
      HazardTitle = > “PositionSensor does not provide PositionReading to
PositionDisplay1”;
      Failure  = > “ef1”;
      FailureEffect = > “PositionSensor does not provide PositionReading to
PositionDisplay1”;
      Phases = > (“all”);
      Environment = > “IMC”;
      Mishap = > “Loss of redundancy for provision of PositionReading”;
      --severity as determined by a higher level assessment
      SeverityCategory = > 1;
      ProbabilityLevel = > C;
      QuantitativeProbability : = > 3.45X10-5;
      TargetSeverityCategory = > 1;
      TargetProbabilityLevel = > C;
      VerificationMethod = > “Analysis and test”;
      SafetyReport = > “SSHA Number PS-SSHA-543 Rev A Date: 23 August 1013”;
      Comment = > “Based on higher system safety analysis results, this becomes severe
major hazard, if fault and design independent redundant sensors are designed in”;
     ]) applies to ef1.Failed;
    **};
  end PositionSensor;
```

Figure 10 Hazard Specification

(128) 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 Propagation

(1) The **error propagations** section of the Error Model subclause is used to define error propagations and error flows. 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 connections between components through their features and along software to hardware component binding relations. In some cases components affect each other along paths that are not declared in the core AADL model.

Syntax

```
error_propagations ::=
    error propagations
     { error_propagation | error_containment }+
     [ flows
        { error_flow }+ ]
    end propagations;
```

Naming Rules

(N1) All defining identifiers of declarations in the error propagations section of the Error Model subclause are part of the Error Model subclause namespace (see Section Annex E.4 Naming Rule (N1)).

Semantics

(2) An **error propagations** section of an Error Model subclause consists of error propagation and error containment declarations, and error flow declarations through error propagation points. Error propagation points are those present in the core AADL model, i.e., features and deployment bindings, or propagation points declared within Error Model subclauses (see Section E.8.3). The error propagation declarations specify the types of error being propagated in and out of error propagation points, while the error containment 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.

E.8.1 Error Propagation and Error Containment Declarations

(3) An *error propagation* declaration specifies that errors of the specific types are propagated into or out of a component through a feature, a binding, or a propagation point not defined in the core AADL model. An *error containment* declaration allows 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 ;

error_containment ::=
    error_propagation_point_reference :
      not ( in | out ) propagation error_type_set ;

error_propagation_point ::=
  feature_reference | binding_reference
  | propagation_point_identifier

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

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

NOTE: keywords match the names used in the binding properties. ***Binding*** *represents a generic binding point and is used for functional_binding*

Naming Rules

(N1) Features referenced in an error propagation or error containment declaration must exist in the name space of the enclosing component of the subclause. This feature reference may identify a feature in a feature group. In this case, the feature group identifier must exist in the name space of the enclosing component of the subclause, and the succeeding identifier must exist in the feature group type of the feature group.

(N2) Propagation points referenced in an error propagation or error declaration must exist within the namespace of the subclause that contains the reference.

(N3) Error propagations and error containments are identified by their *error propagation point*, i.e., by a feature reference, by the keyword identifying an access or binding point, or by a propagation point declared in the Error Model subclause. This reference may be

(N4) For each error propagation point there must be at most one incoming and one outgoing error propagation, and at most one incoming and one outgoing error containment declaration.

(N5) References to an *error type* or *error type set* of an `error_type_set` statement in an error propagation declaration or error containment declaration must exist in the namespace of one of the Error Type libraries listed in the **use types** clause of the containing Error Model subclause.

(N6) 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

(L1) 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 *error containment* declaration for the same feature reference or binding reference.

(L2) The feature reference of error propagations for data components and bus components must be an access feature reference or the keyword *access*.

(L3) The binding reference of error propagations for software components must only contain the keyword *processor, memory, connection*, or *binding*.

(L4) The binding reference of error propagations for virtual bus, virtual processor, and system components may include the keywords processor, connection*, binding,* and *bindings*.

(L5) The binding reference of error propagations for processor, memory, bus, and device components must only contain the keyword *bindings*.

(L6) 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.

(L7) For incoming features there must be at most one incoming error propagation declaration and at most one incoming error containment declaration. For outgoing features there must be at most one outgoing error propagation declaration and at most one outgoing error containment declaration. Binding error propagation points are considered both incoming and outgoing.

Consistency Rules

(C1) For a feature or binding all possible propagated error types must be included in the error type set of an *error propagation* declaration or an *error containment* declaration. Any error type that is not explicitly specified is referred to as *unspecified* error propagation type. The common set of error propagation types provides a checklist of error types to be considered.

Semantics

(4) 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.

(5) An *error containment* 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., contain 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.

(6) Error propagations follow the flow direction of features. 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 propagation may also occur along bus access connections.

(7) 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.

(8) 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.

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

(10) An error propagation may occur between two components that do not have a connection of binding relationship in the core AADL model. 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 this case, propagation points and their connections can be introduced within the Error Model subclause (see Section E.8.3).

(11) Error containment 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.

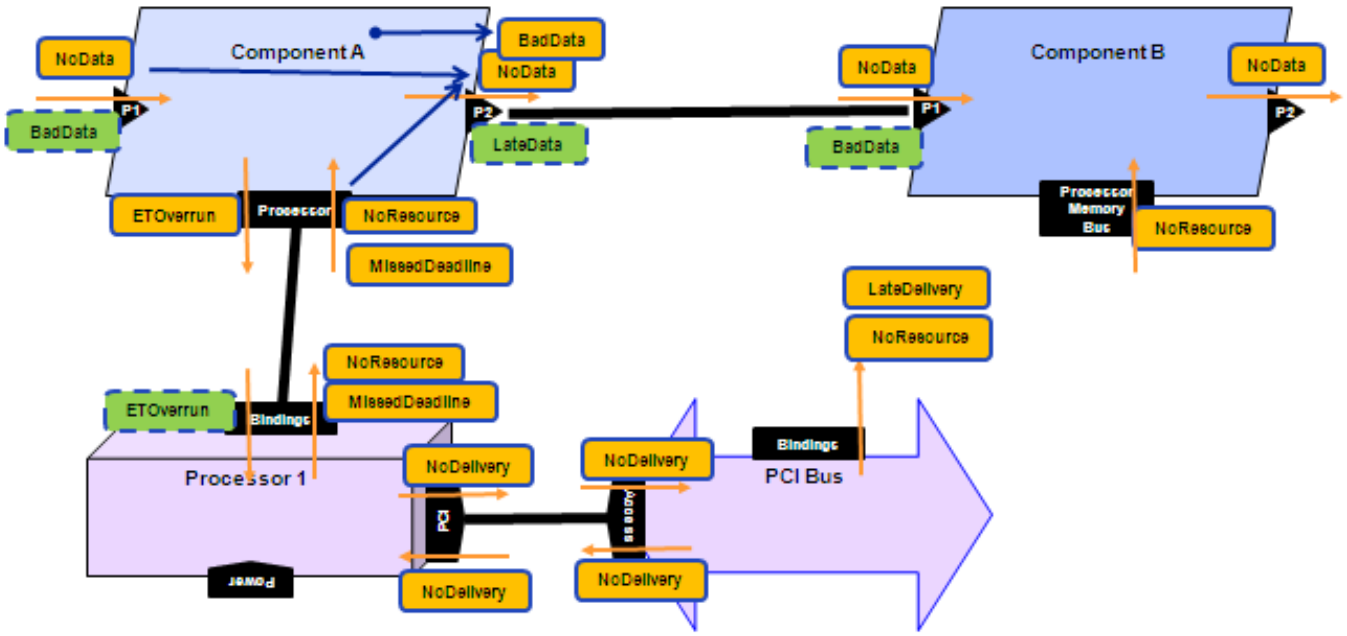

Figure 11 Error Propagations and Flows between Software and Hardware Components

(12) Figure 11 shows two software components and two hardware components with error propagations along various error propagation paths between components (see Section E.8.3) and error flows through a component (see Section E.8.2). 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.8.2 Error Flow Declarations

(13) 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 | all )
   [ effect_error_type_set ] [ when fault_source ] ;

error_sink ::=
  defining_error_sink_identifier :
    error sink ( incoming_error_propagation_point_reference | all ) [ error_type_set ] ;

error_path ::=
  defining_error_path_identifier :
    error path
      ( incoming_error_propagation_point_reference | all ) [ error_type_set ] ->
      ( outgoing_error_propagation_point_reference | all )
        [ target_error_type_instance ] ;

fault_source ::=
  ( error_behavior_state [ error_type_set ])
  | error_type_set | description

description ::= string_literal;
```

Naming Rules

(N1) The defining identifier of an error flow must be unique within the namespace of the Error Model subclause in which it is defined.

(N2) Referenced error propagation points must have been declared as error propagation declarations for the same component as the error flow.

(N3) 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 of the containing Error Model subclause.

(N4) References to an *error type* of a `target_error_type_instance` statement in an error path declaration must exist in the namespace of one of the Error Type libraries listed in the **use types** clause of the containing Error Model subclause.

(N5) 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 of the containing Error Model subclause.

Legality Rules

(L1) 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.

(L2) The `error_type_set` specified for an incoming error propagation point as part of the error flow declaration must be contained in the error type set specified as part of the incoming error propagation declaration.

(L3) The `error_type_set` specified for an outgoing error propagation point as part of the error flow declaration must be contained in the error type set specified as part of the outgoing error propagation.

Consistency Rules

(C1) For each incoming error propagation there must be at least one error path or one error sink referring to it as incoming error propagation point. For each outgoing error propagation there must be at least one error path or one error source referring to it as outgoing error propagation point.

(C2) All error types of an incoming error propagation must be contained in the type set of at least one error path or error sink. All error types of an outgoing error propagation must be contained in the type set of at least one error path or error source.

Semantics

(14) If no error flows are specified, then 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.

(15) *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.

(16) 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.

(17) 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.

(18) 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 contained, or an error of one type gets contained while an error of another type becomes an outgoing propagation of the same or a different type.

(19) The error type set of an error source declaration specifies that the component is the source of error types or type products that are contained in the error type set. If an error source declaration does not include the optional error type set, then the type set of the referenced error propagation point determines the error types or type products.

(20) The error type set of an error sink specifies that the component is the sink for all incoming errors of error types or type products contained in this error type set. If an error sink declaration does not include the optional error type set, then the type set of the referenced error propagation point determines the error types or type products.

(21) An error path maps incoming error types or type products that are contained in the error type set of the incoming error propagation point to the target error type instance of the outgoing error propagation point. If the optional target error type instance is not specified, then the target error type or type product is determined by the type mapping set

referenced in the **use mappings** clause. If no **use mappings** clause is specified, then the incoming error type or type product instance becomes the target error type instance. If the optional error type set of the incoming error propagation points is not specified, then the error type set of the error propagation point determines the incoming error types or type products, for which the error path applies.

(22) The keyword **all** indicates that an error flow specification applies to incoming or outgoing error propagations. In the case of an error source all outgoing error propagations of the specified type set become an error source. In the case of an error sink, all incoming error propagations of the specified type set are contained. In the case of an error path, an incoming error propagation can be mapped to all outgoing error propagations, all incoming error propagations can be mapped to a single outgoing error propagation, and all incoming error propagation can be mapped to all outgoing error propagations. The latter is the default interpretation if no error flows or outgoing propagation conditions (see Annex Section E.10.1) are declared.

E.8.3 Error Propagation Points and Paths

(23) 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, propagation points and paths may be declared to represent error propagation paths between components for which there is no connection or binding relationship in the AADL core model.

Syntax

```
propagation_paths ::=
propagation paths
   { propagation_point }*
   { propagation_path }*
end paths ;

propagation_point ::=
  defining_ propagation_point_identifier : propagation point ;

propagation_path ::=
    defining_observable_propagation_path_identifier :
     source_qualified_propagation_point ->
       target_qualified_propagation_point ;

qualified_propagation_point ::=
    ( subcomponent_identifier . )+ propagation_point_identifier
```

Naming Rules

(N1) The defining identifier of a *propagation point* identifier must be unique within the namespace of the subclause for which the propagation point is defined.

(N2) The defining identifier of a *propagation path* identifier must be unique within the namespace of the Error Model subclause for which the propagation path is defined.

(N3) The *qualified propagation point* reference in a propagation path 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 propagation path declaration.

Legality Rules

The following matching rules apply to error propagations on the source and destination of error propagation paths between components (see Figure 12 later in this section):

(L1) The error type set of the outgoing error propagation must be contained in the error type set of the incoming error propagation.

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

(L3) The direction of the error propagation or error containment 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 *error containment* 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 *error containment* 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*, *error containment* declaration, and any unspecified error propagation type.

Semantics

(24) 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 binding (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.

(25) Error propagation paths between propagation points are declared as propagation paths in the error model subclause of a component. These are propagation paths between propagation points of two subcomponents. The referenced subcomponent can be any subcomponent in the component hierarchy of the component with the propagation path declaration.

(26) Error propagation and error containment 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 12 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 as error type or the destination is silent regarding a known error type. i.e., it has not specified an error propagation or error containment 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 (L1) through (L3) and (C1) through (C3) of Section Annex E.8.3 for details of these consistency rules.

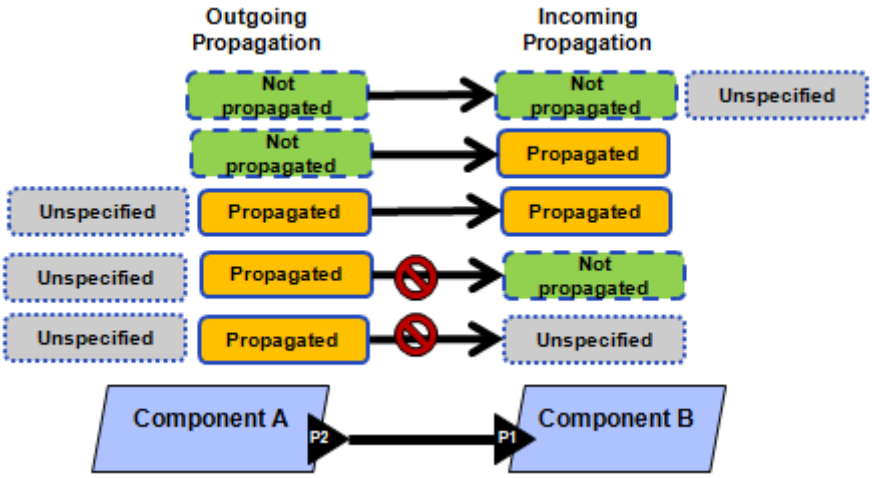

Figure 12 Consistent and Inconsistent Error Propagation Paths

(27) 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.

Annex E.9 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 behavior events, states and error propagations to specify acceptable types of tokens. The current state, when typed, is represented by a type instance. Section Annex E.9.3 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 an error (*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.10) and a *composite error behavior* subclause (Annex E.11).

(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.

(5) *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
    [ use types error_type_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 ;

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 ] ;

error_event_condition ::= string_literal
```

Note: error_event_condition states what condition must be met for the event to trigger. Example: temp > Max_Temperature (Above_Range error type). This is currently expressed as string value. The intent is to support the Constraint Annex notation in its place.

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

event_initiation ::=
  when ( initiator_reference { , initiator_reference }* )

initiator_reference ::=
  mode_transition_reference | port_reference | self_event_reference
```

Note: event_initiation allows the modeler to specify the event (from a port or a component event source) or mode transition in the AADL core model that initiates recovery or repair.

```
repair_event ::=
  defining_error_behavior_event_identifier : repair event
    [ repair_event_initiation ] ;

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_source -[ error_condition]->
     ( error_transition_target | error_transition_branch ) ;

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

error_transition_target ::=
    ( target_error_state_identifier [ target_error_type_instance ] )
    | same state

error_transition_branch ::=
```

```
    ( error_transition_target with branch_probability
     { , error_transition_target with branch_probability }* )

error_condition ::=
  error_condition_trigger
  | ( error_condition )
  | error_condition and error_condition
  | error_condition or error_condition
  | numeric_literal ormore
     ( error_condition_trigger { , error_condition_trigger }+ )
  | numeric_literal orless
     (error_condition_trigger { , error_condition_trigger }+ )


error_condition_trigger ::=
    error_behavior_event_identifier [error_type_set ]
    | incoming_error_propagation_point [ error_type_set_or_noerror ]
    | subcomponent_identifier . outgoing_error_propagation_point [
error_type_set_or_noerror ]

branch_probability ::=
  fixed_probability_value | others

fixed_probability_value ::=
    real_literal |
    ( [ package_identifier :: ] real_property_constant_identifier )
```

Note: the property constant name is interpreted as a symbolic label.

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) 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. This qualification is optional if the referenced error behavior state machine is declared in the same Error Model library as the reference.

(N4) 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 (see also Section Annex E.4 Naming Rule (N3)).

(N5) 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 of the containing Error Model subclause.

(N6) References to an *error type* or *error type set* of an `error_type_set` statement in a transition declaration must exist in the namespace of one of the Error Type libraries listed in the **use types** clause of the containing Error Model subclause.

(N7) 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.

(N8) The behavior event reference of an error condition trigger must identify a defining error event, recover event, or repair event in the namespace of the error state machine containing the reference.

(N9) The incoming error propagation reference of an error condition trigger must identify an error propagation in the component that contains the error condition expression.

(N10) The subcomponent reference of an error condition trigger must identify a subcomponent in the namespace of the component implementation containing the error condition expression. The outgoing error propagation reference must identify an error propagation in the referenced subcomponent.

(N11) 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.

(N12) 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_condition_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_condition_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 `error_condition_trigger` of an outgoing error propagation must only refer to error behavior events and incoming error propagation points.

(L7) The `error_condition_trigger` of an error detection can refer to error behavior events, incoming error propagation points, and outgoing error propagations of subcomponents.

(L8) The optional `error_type_set` of a *source* error state must be contained in the error type set specified with the defining state declaration.

(L9) The optional `error_type_set` of a *target* error state must be contained in the error type set specified with the defining state declaration.

(L10) 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.

(L11) 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

(6) 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.

(7) An error behavior state machine specification can be reused by associating it with components in *component error behavior* specifications (Annex E.10) and *composite error behavior* specifications (Annex E.11).

(8) An error behavior state machine can be defined as a typed token state transition system by associating error types as type sets with error events and state. This leads to a more compact error behavior specification.

E.9.1 Error, Recover, and Repair Events

(9) 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.

(10) 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

(11) 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.

(12) 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.

(13) 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).

(14) 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.

(15) 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.

(16) A recover event may be used to model transient error behavior of a component in that it represents the trigger to return from a nonworking state to a working state. A *DurationDistribution* property indicates a distribution over a time range as the length of time the component transiently stays in a non-working state. By default it has an occurrence probability of 1.

(17) 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.

(18) A duration property and an occurrence property characterize the repair event. A *DurationDistribution* property indicates a time range reflecting the duration of a repair as well as the distribution over the duration time range. An *OccurrenceDistribution* 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.

(19) A recover or repair event declaration may include a **when** clause to specify the initiator of the event in the core AADL model. This initiator can be a mode transition, an event from a port, or a component event source (**self.**<eventname>). If multiple initiators are listed, any of them can represent the initiation.

(20) 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 failure to complete recovery or repair. The probability specified for each branch indicates the probability of success or failure.

E.9.2 Error Behavior States and Transitions

(21) 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.

(22) 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 component is erroneous, i.e., malfunctioned or has lost its function. A component can have one or more working states and one or more non-working states.

(23) 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.

(24) 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. An error behavior transition can also specify that an error state does not change when a transition condition is satisfied.

(25) A transition can be a branching transition with multiple target states. Once the transition is taken, one of the specified target states is selected according to a specified probability with fixed distribution. 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.

(26) 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.

(27) 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.

(28) 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 *port1* and *port2*.

- If an alternative transition condition specifies a single error propagation point, e.g., `port1{BadValue},` by itself, then all other incoming error propagation points must not have a propagation present. If the alternative transition condition specifies `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., `port1{BadValue}` **or** `port2{BadValue}`, then the transition condition is satisfied if *port1* has an error propagation present and *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 `1` **ormore** `(port1{BadValue},port2{BadValue})`, then the condition is satisfied if error propagations are present on either port or on both ports.

(29) 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 **NoError**.

(30) Separately declared error behavior events within the same components or different components are considered to occur separately. If they occur at the same time then an arbitrary occurrence order is assumed. An error behavior specification may specify one error event or combinations of error events as a transition condition.

(31) Simultaneous occurrence of errors of more than one type is modeled by a typed error event with an error type product of more than one element type. For example, an error event declared with `{BadValue*LateLate}` represents *BadValue* and *LateValue* occurring simultaneously.

(32) 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.10.

(33) A transition can be declared as a steady state transition. In this case the source error state remains the current state and, if typed, its error type token remains the same.

(34) 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, error containment declarations, and error flow specifications for the component. The consistency rules expressing this can be found in Section Annex E.10.

Example

```
package RecoverErrorModelLibrary
public
annex EMV2 {**
error behavior Example
  events
    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.9.3 Typed Error Behavior State Machines

(35) A typed error behavior state machine represents a typed token state transition system with instances of error types or type products.

(36) 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 instance.

(37) 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 the current state includes a type instance that is contained in the specified type set.

(38) The set of type instances 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 instance 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 type instance for the current state, effectively transitioning between the sub-state representing the original type instance and the sub-state representing the new type instance.

(39) 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 type instances of the current state the transition applies.

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

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

(42) The error type instance 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 instance.
- If the target state of a transition does not have a target error type declared, then type transformation rules associated with the error behavior state machine via **use transformations** are used to determine the *target* error type.
- If no target error type or type transformation rules are specified then default rules apply.

(43) Type transformation rules determine the target error type of the state by matching the *source* error type with the *source* element of the transformation rules and the *contributor* error type with the *contributor* element of the transformation rules (see Section Annex E.12 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.

(44) The default rule to determine the *target* error type is as follows:

- If the source state of the transition is not typed, then the target error type of the 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.
- If the source state is typed and the error event is not typed, then the target error type is that of the source state.
- If neither the source state nor the error event is typed, then the target error type must be explicitly specified in the transition declaration.
- If the source state is typed and the contributor is typed, then the *target* error type is determined by the contributor error type. In the case of a type product it consists of union of the type product 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 *tut* is determined from the source error type token *tos* and the contributor error type token *toc* as follows: $\forall i: tot_i \in tot \mid tot_i \in toc \vee (tot_i \in tos \wedge (\forall toc_k \in toc \mid r(tot_i) \neq r(toc_k)))$. For multiple typed contributors the rule is applied to each in order.
- Otherwise, target error type is undefined.

(45) 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.

Example

```
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
    Fault: error event {MyFault};
    SelfRepair: recover event;
   states
    Operational: initial state ;
    FailTransient: state {DetectedFault};
    FailPermanent: state {MyFault};
  transitions
     SelfFail: Operational -[Fault{DetectedFault}]->
        (FailPermanent with 0.7, FailTransient with 0.3);
     Recovery: FailTransient –[SelfRepair]-> Operational;
     UncoveredFail: Operational -[Fault{UndetectedFault}]-> FailPermanent;
  end behavior;
 **};
end TypedErrorModelLibrary;
```

Annex E.10 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.11 defines composite error behavior specifications.

(2) 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 (additional) 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.

(3) A system cannot diagnose, recover from, or repair a failed component until the error 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 to specify the way that the system reports a detected component error as event or event data via ports.

Syntax

```
component_error_behavior ::=
    component error behavior
      [ use transformations type_transformation_set_reference ; ]
      [ events { error_behavior_event }+ ]
      [ transitions { component_specific_error_behavior_transition }+ ]
      [ propagations { outgoing_propagation_condition }+ ]
      [ detections { error_detection }+ ]
      [ mode mappings { error_state_to_mode_mapping }+ ]
    end component;

outgoing_propagation_condition ::=
  [ defining_outgoing_propagation_identifier : ]
  ( error_source | all )
  -[ [ error_condition ] ]->
  propagation_target ;

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

error_detection ::=
  [ defining_error_detection_identifier : ]
  ( error_source | all )
    -[ [ error_condition ] ]->
    error_detection_effect ;

error_detection_effect ::=
    ( port_identifier | component_event_reference ) !
    [ ( error_code_value ) ]
```

NOTE: We are considering adding information about the actual error type being detected to infer the existence of an error type. For example, detection of bounded omission sequence to determine service omission.

Example syntax: ***when*** `BoundedOmissionSequence => ServiceOmission`

```
component_event_reference ::=
    self . event_or_event_data_source_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 [ target_error_type_instance ]
    in modes ( mode_name { , mode_name }* ) ;
```

Naming Rules

(N1) 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 subclause that contains the component error behavior specification. They must not conflict with component feature identifiers representing error propagation and error containments (see Section E.8.1 (N2)).

(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 of the containing Error Model subclause.

(N3) The error behavior event reference of an error event condition in a transition declaration, outgoing propagation condition declaration, or error detection declaration 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.

(N4) The identifier of an error behavior state reference in a component specific transition, outgoing propagation condition, 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.

(N5) The identifier of an error propagation point reference of a propagation target declaration must exist in the error propagations clause of the component that contains the component error behavior specification.

(N6) The port identifier in a detection event must exist in the namespace of the component containing the component error behavior specification.

(N7) The type transformation set reference in the **use transformations** clause 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.

(N8) 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.

(N9) The enumeration identifier referenced as error code value must exist as enumeration literal in the data type of the port.

(N10) The error type references in the `target_error_type_instance` must exist in the namespace of one of the Error Type libraries listed in the **use types** clause of the containing Error Model subclause.

(N11) The modes referenced in the **in modes** clause must exist in the namespace of the enclosing component classifier.

Legality Rules

(L1) The **use transformation**s statement overrides 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 `target_error_type_instance` of the propagation target in an outgoing propagation declaration must be contained in the error type set specified with the defining error propagation point declaration and must not be contained in an outgoing error containment declaration.

(L4) The optional `target_error_type_instance` of the error state to mode mapping declaration must be contained in the error type set specified with the defining error state declaration.

(L5) The optional `error_type_set` of an error condition trigger in an outgoing propagation declaration must be contained in the error type set specified with the defining incoming error propagation declaration and must not be contained in an incoming error containment declaration.

(L6) The optional `error_type_set` of an error condition trigger in an error detection condition expression must be contained in the error type set specified with the defining incoming error propagation declaration and must not be contained in an incoming error containment declaration.

Consistency Rules

(C1) If an error event occurs in a given error behavior state and no outgoing transition names the error event then the error event is considered unhandled. For typed error events there must be at least one transition trigger whose type set contains each element of the error event type set.

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

(C3) For the current error behavior state there must be one unique outgoing transition. For typed error behavior states there must be a unique transition for each type instance.

(C4) For each typed error behavior state there must be at least one incoming transition with a type instance for each element of the error behavior state type set.

(C5) For each outgoing propagation there must be at least one outgoing propagation condition. All error types in the outgoing propagation type set must be contained in at least one outgoing propagation condition target type set.

(C6) The error types of each incoming error propagation must be covered by outgoing propagation condition declarations.

(C7) The result of evaluating the outgoing propagation declarations for an outgoing error propagation must result in at most one propagated error type.

(C8) The result of evaluating the error detection declarations must be at most one detected error.

(C9) If a component type or component implementation has modes, then each mode must be specified in at least one state to mode mapping.

The next set of rules address consistency of component error behavior with error propagation and error flow declarations.

(C10) The incoming error type set of all error sink declarations must be covered by outgoing propagation condition declarations resulting in *{noerror}* as target.

(C11) All error paths must be covered by outgoing propagation condition declarations, where the incoming error path element and its type are listed in an error condition trigger and the outgoing error path element and its type are specified as propagation target.

(C12) All error sources must be covered by outgoing propagation condition declarations, where the outgoing type set must be contained in the propagation target type set of at least one outgoing propagation condition, and the failure mode specified in the *when* statement of the error source must be referred to by at least one error condition trigger.

(C13) All error sinks must be covered by error detection declarations, where the incoming type set must be contained in the type set of an error condition trigger referring to the incoming error propagation of the error sink.

(C14) All error paths that transform error types must be covered by error detection declarations, where the incoming type set must be contained in the type set of an error condition trigger referring to the incoming error propagation of the error path.

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 an error behavior state or an incoming error propagation, is actually detected by a system component and reported as an event or event data (message).

E.10.1 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 contained for all or for specific outgoing error propagation points. 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, the error behavior state of a component becomes observable to other components as a result of error propagation.

(7) 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.

(8) 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 empty condition expression indicates that the outgoing error propagation occurs whenever the component is in the specified error behavior state (and error type).

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

(10) An outgoing propagation declaration can also explicitly specify the error type to be propagated. If this propagated error type is not specified, type transformation rules or default rules are used to determine the propagated error type.

(11) 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 clause **(noerror)**.

(12) 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, error containment declarations, and error flow specifications for the component.

Examples

```
component error behavior
events
     ResetEvent: recover event;
transitions
     -- incoming service omission gets reflected in a transition to Failed
     tfail: Operational -[Failure or inp{ServiceOmission} ]-> Failed;
     -- reset will be successful with 0.8 probability, and fail with 0.2 probability
     treset: Failed -[ ResetEvent ]->  (Operational with 0.8, Failed with 0.2);
propagations
     -- When in operational state the processing unit passes through Value Errors
     -- {ValueError} acts as a filter constraint. All incoming error types
     -- that match the constraint are passed through to the outgoing port
     Operational -[ inp{ValueError}]-> outp;
end component;
```

E.10.2 Error Detections

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

(14) 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.

(15) 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.

NOTE: A detection condition could also refer to an outgoing propagation of a subcomponent.

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

(17) An error detection declaration can also explicitly specify the error code used to represent the error state or error propagation that has been detected.

(18) 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.

(19) 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`.

(20) The error detection may also be to be made known as an event or data event within the component using the **self** clause. It can then be referenced as a mode transition trigger or as an endpoint of a connection.

E.10.3 Operational Modes and Failure Modes

(21) 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.

(22) 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.

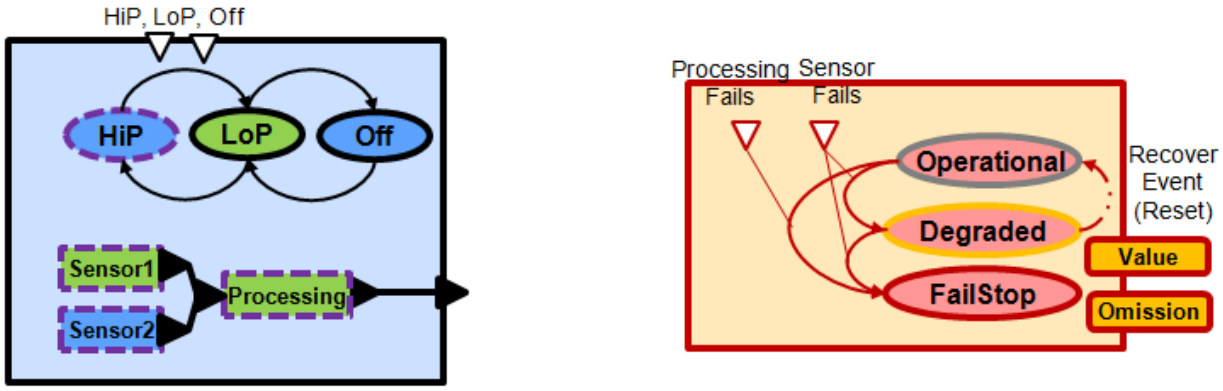

Figure 13 Operational Modes and Failure Modes

(23) Figure 13 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.

(24) Figure 13 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.

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

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

(26) By doing so we superimpose the error behavior states onto the mode state machine as shown in Figure 14. 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.

(27) 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.

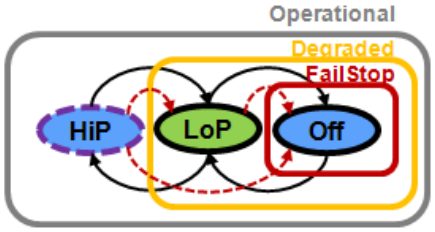

Figure 14 Superimposed Error Behavior States

(28) From this mapping we can derive a composite operational and failure mode state diagram shown in Figure 15. It reflects different configurations due to loss of component function.

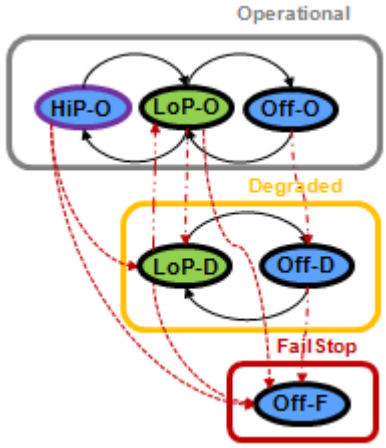

Figure 15 Composite Operational and Failure Mode State Diagram

Annex E.11 Composite Error Behavior

(1) 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
      states { composite_error_state }+
end composite;


composite_error_state ::=
    [ defining_composite_error_state_identifier : ]
  [ ( subcomponent_state_expression | others ) ]->
    composite_state_identifier [ target_error_type_instance ]  ;


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

```
  | composite_state_expression or composite_state_expression
  | numeric_literal ormore
     (composite_state_element { , composite_state_element }+ )
  | numeric_literal orless
     (composite_state_element { , composite_state_element }+ )

composite_state_element ::=
    subcomponent_error_state [ error_type_set ]
    | in incoming_error_propagation_point [ error_type_set_or_noerror ]

subcomponent_error_state ::=
  { subcomponent_identifier . }+ error_behavior_state_identifier
```

Naming Rules

(N1) References to an *error type* or *error type set* of a composite error state clause must exist in the namespace of one of the Error Model libraries listed in the **use types** clause of the containing Error Model subclause.

(N2) The identifier of the first subcomponent reference of a `subcomponent_error_state` 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.

(N3) The error behavior state reference of a `subcomponent_error_state` must exist in the namespace of the error behavior state machine associated with the last subcomponent, i.e., with its classifier.

(N4) The incoming error propagation point reference of a `composite_state_element` must exist in the namespace of the component that contains the composite error behavior specification.

Legality Rules

(L1) The `error_type_set` of the error behavior state reference in a `composite_state_element` declaration of must be contained in the error type set specified with the error behavior state or error propagation being referenced.

(L2) A composite error behavior specification must contain at most one `composite_error_state` declaration with an **others** condition.

Consistency Rules

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

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

(C3) For each composite state there must be a corresponding error state in the component error behavior specification for the same component.

(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.

(C5) The logic expression of the composite error behavior declaration must be consistent with the fault behavior logic of the subcomponents.

(C6) The incoming and outgoing error propagations of the enclosing composite component must be consistent with the error propagations of the subcomponents whose feature is connected to the composite component feature of the propagation.

(C7) Error sources, sinks, and paths declared for the composite component must be consistent with the error flows through the subcomponents. For example, an error source of the composite component must be backward traceable to an error source of one of the subcomponents.

Semantics

(2) Composite error behavior clause specifies error behavior states of the composite component in terms of its subcomponent error behavior states as well as in terms of external error sources. 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. The **others** keyword can be used to define one composite component error behavior state as representing all remaining subcomponent error state conditions not covered by the other declarations.

(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

```
use behavior ErrorLibrary::BabblingSM;
composite error behavior
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.12 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 16.

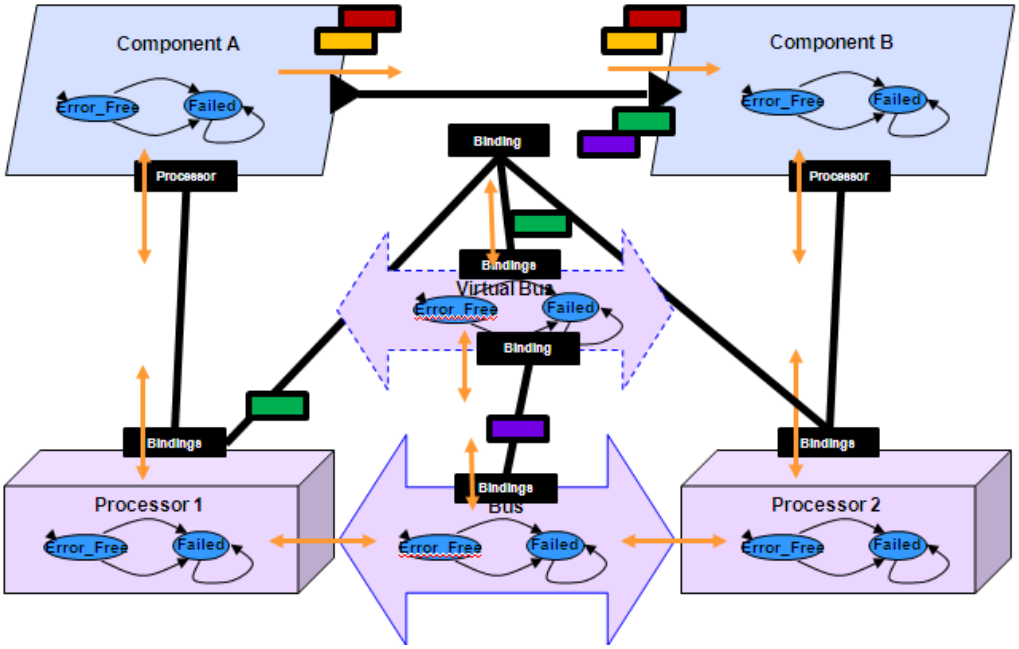

Figure 16 Error Propagation and Connections

(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

```
connection_error_behavior ::=
connection error
  [ use transformations type_transformation_set_reference ; ]
  { connection_error_source }*
end connection;


connection_error_source ::=
  defining_error_source_identifier :
    error source ( connection_identifier | all )
   [ effect_error_type_set [ when ( fault_source_error_type_set | description ) ] ] ;
```

Naming Rules

(N1) The reference to the type transformation set must exist in the Error Model library and must be qualified with the package name of the Error Model library, if the reference and the type transformation set reside in different packages.

(N2) 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 of the containing Error Model subclause.

(N3) 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. Users can specify an error type or type set to represent the effect this has on the communication, e.g., ValueError. Using **when** the user can also specify the error type or type set representing the original error (failure mode), e.g., measurement unit mismatch.

(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, first the connection error source and then the incoming propagations from the connection binding.

Annex E.13 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 type instance of the error propagation to a different type or type product. Second, the target error type of a transition may be determined by the type instance of the source and that of the transition trigger, i.e., by the type instance of the triggering event or incoming error propagation. For example, the type instance of the source may stay the same or may change due to the type instance contributed by the error event or transition condition. Third, the type instance of an error propagation along a connection may be affected by the error propagation 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 an error type or a type product matching a type set constraint is mapped into another type instance. A type mapping set can be associated with error flow paths.

(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* type instance that satisfies a constraint and a *contributor* type instance that satisfies a second constraint is mapped into a *target* type instance. 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 ]
```

```
   { error_type_mapping }+
  end mappings;


error_type_mapping ::=
    source_error_type_set -> target_error_type_instance ;

type_transformation_set ::=
  type transformations defining_type_transformation_set_identifier
  [ use_error_types ]
   {error_type_transformation }+
  end transformations;

error_type_transformation ::=
   ( source_error_type_set_or_noerror | all )
   -[ [ contributor_error_type_set_or_noerror ] ]-> target_error_type_instance ;

type_mapping_set_reference ::=
  [ package_reference :: ] type_mapping_set_identifier

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.

(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 type mappings or type transformations declaration.

(N4) 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 or must be qualified by an error type library name.

(N5) References to an *error type* in a `target_error_type_instance` statement must exist in the namespace of one of the Error Model libraries listed in the **use types** clause or must be qualified by an error type library name.

Consistency Rules

(C1) A type mapping set must define a unique set of mappings, i.e., every given *source* type instance must result in exactly one target type instance. This means that the *source error type constraints* of two type mapping rules must not intersect.

(C2) A type transformation set must define a unique set of transformations, i.e., for any given *source* and *contributor* type instance at most one transformation rule must apply.

(C3) A type mapping set must cover all possible *source* error types or type products of the entity the type mapping set is applied to.

(C4) A type transformation set must cover all possible *source* and *contributor* error types or type products of the entity the type mapping set is applied to.

Semantics

(4) Type mapping sets are used with error flow paths to change a type instance into another type instance.

(5) A type mapping set specifies a set of mapping rules of *source* error types or type products into *target* type instances. It specifies the mapping of a type instance that is contained in the *source* error type set into another type instance.

(6) Type transformation sets are used with transitions and with connections to indicate how a *source* type instance is combined with a *contributor* type instance, i.e., a transition trigger or binding propagation of a connection, is changed into a *target* type instance. If multiple contributors 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.

(7) The rules of a type transformation set specify how a *source* type instance that satisfies a constraint and a *contributor* type instance that satisfies a second constraint is mapped into a *target* type instance. A type transformation set can be associated with a component to be used on connections and as default transformation on transitions and outgoing propagation conditions.

(8) A type transformation rule with an empty contributor indicates that the rule applies to any contributor including no error propagation, i.e., the transformation is determined by the source.

(9) A type transformation rule with **all** as source indicates that the rule applies to any source including no error propagation, i.e., the transformation is determined by the contributor.

Example

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

Annex E.14 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 Annex E.9). Alternatively, a repair action can be explicitly represented in the Error Behavior model as a separate state.

(4) Section Annex E.10.3 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 error state. 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 appropriate actions to compensate for the failure, e.g., by using input from an alternative source, or by reporting the detection to a health monitor.

(6) 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.