AEROSPACE STANDARD

AS5506™/1	REV. A
Issued 2006-06 Revised 2015-09 Superseding AS5506/1	

(R) SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex

RATIONALE

The purpose of the annexes in this document are:

- **The ARINC653 annex**: provide modeling guidelines to specify avionics architectures, as the ones used by the ARINC653 standard. This annex replaces the existing ARINC653 AADL annex published in AS5506/2.
- **The Code Generation Annex**: define a binding between the AADL notations and existing programming languages used to develop safety-critical systems (such as Ada and C). This is a new annex to the AADL standard.
- **The Error Model Annex**: extend the core language in order to provide the ability to to specify error propagations and error behavior in the architecture. This annex replaces the existing Error-Model annex published in the first revision of AS5506/1.

The other annexes from the existing AS5506/1 document (Graphical AADL Notation, AADL Meta-Model and Interchange Formats, Language Compliance and Application Program Interface) are deprecated and not updated in this new revision.

The Architecture Analysis and Design Language (AADL) standard document AS5506B was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division and revised by SAE in September 2012. The Annexes presented herein have also been developed under the auspices of the SAE AS-2C Subcommittee.

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
SAE WEB ADDRESS: **http://www.sae.org**

SAE values your input. To provide feedback on this Technical Report, please visit
http://www.sae.org/technical/standards/AS5506/1A

FOREWORD

This standard was prepared by the Society of Automotive Engineers (SAE) Avionics Systems Division (ASD) Embedded Computing Systems Committee (AS-2) Architecture Description Language (AS-2C) subcommittee.

This AADL standard document replaces the initial Error-Model Annex specified in the Annex Collection Volume 1 AS5506/1 published in 2006. It also replaces the ARINC653 annex defined in AS5506/2 and introduces the Code Generation Annex.

This AADL standard document defines an Error Model language that extends the AADL core language with a state machine-based notation. This notation allows for specification of 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. The notation also allows for specification fault mitigation strategies expected to be implemented in the health monitoring and fault management component of the actual system – also known as Fault Detection, Isolation, and Recovery (FDIR). The actual design of this component is expressed in the AADL core model.

This AADL standard document contains an Annex (the ARINC653 Annex) that defines modeling guidelines that use and extend the AADL core language – as defined in AS5506B – to Integrated Modular Avionics Architectures (IMA), such as ARINC653.

This AADL standard document contains an Annex (Code Generation Annex) that defines the bindings between the core language – as defines in AS5506B – and existing programming languages (such as Ada or C).

INTRODUCTION

The SAE Architecture Analysis and Design Language (referred to in this document as AADL) is a modeling 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. The 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). The 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.

The 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.

Since the AADL supports multiple and extensible analysis approaches, it provides the ability to analyze the cross cutting impacts of change in the architecture in one specification using a variety of analysis tools. The AADL specification language is designed to be used with analysis tools that support the automatic generation of the source code needed to integrate the system components and build a system executive. Since the models and the architecture specification drive the design and implementation, they can be maintained to permit model driven architecture based changes throughout the system lifecycle.

This document includes three annexes to the SAE AADL standard AS5506B to extend the AADL core language with error modeling support, define modeling patterns to specify ARINC653 architectures and define a mapping between the AADL core language and existing programming languages such as Ada or C.

INFORMATION AND FEEDBACK

The website at http://www.aadl.info is an information source regarding the SAE AADL standard. It makes available papers on the AADL, its benefits, and its use. Also available are papers on MetaH, the technology that demonstrated the practicality of a model-based system engineering approach based on architecture description languages for embedded real-time systems.

The website provides links to three SAE AADL related discussion forums:

1. The SAE AADL User Forum to ask questions and share experiences about modeling with SAE AADL,

2. The AADL Toolset User Forum to ask questions and share experiences with the AADL Open Source Toolset Environment, and

3. The SAE Standard Document Corrections and Improvements Forum that records errata, corrections, and improvements to the current release of 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.

Questions and inquiries regarding the annexes contained in this document, working versions of annexes, and future versions of the standard can be addressed to info@aadl.info.

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_A1-ss.ss(pp)
!from Author Name yy-mm-dd
!discussion
text of discussion
```

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.

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

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
```

TABLE OF CONTENTS

1. SCOPE

(1) This document contains three annexes to the SAE AS5506B Standard - the SAE Architecture Analysis and Description Language.

(2) The first annex, the Error-Model Language extends the AADL core language with a state machine-based notation. This notation allows for specification of 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. The notation also allows for specification fault mitigation strategies expected to be implemented in the health monitoring and fault management component of the actual system – also known as Fault Detection, Isolation, and Recovery (FDIR). The actual design of this component is expressed in the AADL core model.

(3) The second annex, the ARINC653 Annex defines modeling patterns to use the AADL core language for the specification of Integrated Modular Avionics Architectures (IMA), as defined by the ARINC653 standard. It also introduces a dedicated property set to capture specific requirements of such architectures.

(4) The third annex, the Code Generation Annex, defines a mapping between the AADL core language and programming languages. It specifies, for each AADL component type, how to map it into executable code. As the AADL language targets safety-critical systems, the annex focuses on defining such a mapping for programming languages that are typically used to implement such systems. However, mapping rules and principles defined in this annex can be translated to other programming languages.

1.1 Purpose

(1) The purpose of this document is to supply the information needed to integrate the SAE AADL notation with tools that enable the production of high quality, safety-critical, real-time, embedded systems from AADL specifications. The material contained in this standard may be used to facilitate the development of graphical tools that may be used to generate graphical representations of AADL specifications. This document also includes the specification of the models needed to allow the contents of AADL specifications to be transformed into representations that are interchangeable with other tools. Finally, this document contains guidelines that supplement the transition between AADL models and software source code.

2. REFERENCES

(2) This section lists all of the pertinent and reference documents. The first section will identify the SAE documents. Followed by other documents grouped by publisher and information on where to obtain documents, when available.

2.1 Applicable Documents

(3) The following publications form a part of this document to the extent specified herein. The latest issue of SAE publications shall apply. The applicable issue of other publications shall be the issue in effect of the date of issue of this document. In the event of conflict between the text in this document and references cited herein, the text of this document takes precedence. Nothing in this document, however, supersedes applicable laws and regulations unless a specific exemption has been obtained.

2.1.1 SAE Publications

(4) Available from SAE International, 400 Commonwealth Drive, Warrendale, PA 15096-0001, Tel: 877-606-7323 (inside USA and Canada) or 724-776-4970 (outside USA), www.sae.org.

(5) SAE AS-5506B:2012, Architecture Analysis & Design Language (AADL), Sept 2012 [AS5506B].

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

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

(8) ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996 [ARP4761].

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

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

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

2.1.2 Radio Technical Commission for Aeronautics Publications

(12) Available from RTCA, Inc., 1828 L Street, NW, Suite 805, Washington DC 20036, Tel: 202-833-9339, www.rtca.org.

(13) DO-178B Software Considerations in Airborne Systems and Equipment Certification, December 1992 [DO178B].

(14) DO-254 Design Assurance Guidance for Airborne Electronic Hardware, April 2000 [DO254].

2.1.3 ARINC Standards

(15) Available from the ARINC Standards order. SAE-ITC, ARINC Industry Activities, 16701 Melford Blvd, Suite 120 – Bowie, MD 20715

(16) Avionics Application Software Standard Interface, Part 1, Required Services [ARINC653-P1-3]

2.1.4 Military Standards and Handbooks

(17) Available from the Document Automation and Production Service (DAPS), Building 4/D, 700 Robbins Avenue, Philadelphia, PA 19111-5094, Tel: 215-697-6257, http://assist.daps.dla.mil/quicksearch/

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

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

2.1.5 IEEE Publications

(20) Available from IEEE, 445 Hoes Lane, Piscataway, NJ 08854-1331, Tel: 732-981-0060, www.ieee.org.

(21) IEEE Standard 1003.1-2001 Information Technology - Portable Operating System Interface (POSIX). Institute of Electrical and Electronic Engineers [IEEE 2001].

(22) IEEE Standard 1003.13-1998 Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP). The Institute of Electrical and Electronics Engineers [IEEE 1998].

(23) IEEE Standard 1003.5b-1996 Information Technology - POSIX Ada Language Interfaces - Part 1: Binding for System Application Program Interface (API) - Amendment 1: Realtime Extensions. The Institute of Electrical and Engineering Electronics [IEEE 1996].

2.1.6 ISO Publications

(24) Available from ISO, 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tel: +41 22 749 01 11, www.iso.org.

(25) ISO/IEC 8652:1995 (E) International Organization for Standardization, Information Technology - Programming Languages - Ada [ISO 1995]

(26) ISO/IEC 8652:1995/COR.1:2001 International Organization for Standardization, Technical Corrigendum to Information Technology - Programming Languages - Ada [ISO 2001]

(27) ISO/IEC 9899:1999 International Organization for Standardization, Information Technology - Programming Languages - C [ISO 1999]

(28) ISO/IEC TR 15942:2000 International Standards Organization, Guide for the Use of the Ada Programming Language in High Integrity Systems [ISO 2000]

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

2.1.7 Applicable References

(1) 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. [LAP 1992]

(2) 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. [IFIP WG10.4-1992]

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

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

(5) 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. [SHARD 1994]

(6) 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. [Bondavalli 1990]

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

(8) 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. [DAM 2008]

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

(10) 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. [Miller 2005]

(11) 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. [Rugina 2007

Annex A ARINC653 Annex

A.1 Rationale

(1) This annex has been defined to support the modeling, analysis and automated integration of ARINC 653 and derived or similar partitioned architectures. It provides AADL architectural style guidelines and AADL defined ARINC 653 oriented properties to define a common approach to use AADL standardized components to express ARINC 653 architectures. Without the annex defined framework, modelers would need to define their own AADL ARINC 653 oriented properties and select their own approach to representing ARINC 653 with AADL components.

(2) This annex was made to help the system designers in the modeling of partitioned architectures, especially ARINC 653 compliant systems. AADL models can be checked and verified using various tools. Consequently, the modeling of partitioned architectures will help system designers to verify and validate their models.This document does not provide any guidance for the verification of AADL models except to provide a means for common specification.

(3) By providing a common framework for partitioned system expression, distributed development and common analysis tools are supported. Code generators to auto integrate ARINC 653 systems based on AADL ARINC 653 annex compliant models are supported. The code generation annex of the AADL provides sufficient information to map most of AADL/ARINC 653 modeling patterns (processes, inter and intra communication channels, shared data) into C or Ada code. However, configuration of ARINC 653 operating systems is not detailed in the code generation annex.

(4) The ARINC 653 standard contains several parts, defining required and extended services. This document provides the mapping between the AADL and the required services of the ARINC 653 standard.

(5) The current document provides a mapping for services defined in the ARINC 653 PART1 standard. Mapping of other services are beyond the scope of this annex.

(6) The avionics-specific terms used in the annex are defined below:

a. **Integrated Modular Avionics**: A shared set of flexible, reusable, and interoperable hardware and software resources that, when integrated, form a platform that provides services, designed and verified to a defined set of safety and performance requirements, to host applications performing aircraft functions.

b. **Module**: A component or collection of components that may be accepted by themselves or in the context of an IMA system. A module may also comprise other modules. A module may be software, hardware, or a combination of hardware and software, which provides resources to the IMA system hosted application.

c. **Application**: Software and/or application-specific hardware with a defined set of interfaces that when integrated with a platform(s) performs a function.

d. **Application software**: The part of an application implemented through software. It may be allocated to one or more partitions.

e. **Partition**: An allocation of resources whose properties are guaranteed and protected by the platform from adverse interaction or influences from outside the partition.

f. **Application-specific hardware**: Hardware dedicated to one application.

g. **Cabinet**: A physical package containing one or more IMA components or modules, that provides partial protection from environmental effects (shielding) and may enable installation and removal of those component(s) or module(s) from the aircraft without physically altering other aircraft ystems or equipment.

h. **Core Software**: The operating system and support software that manage platform resources to provide an environment in which an application can execute.

i. **Component**: A self-contained hardware or software part, database, or combination thereof that may be configuration controlled.

A.2 ARINC 653 partition management (ARINC 653 Module)

(7) In ARINC 653, partitions are managed by the core software with a dedicated kernel that ensures time and space isolation. It schedules partitions using a static timeline scheduling algorithm repeated at a given rate, the major time frame. Each partition has at least one time frame to execute its tasks (called processes in the ARINC 653 standard).

(8) The ARINC 653 core software and its associated physical processor core are modeled with AADL using the `processor` component. This approach is consistent with the AADL concept of a processor to include the operating evironment. The `processor` component is used to specifiy partition management properties which express its requirements.

a. The AADL `processor` component models the ARINC 653 core software. ARINC 653 modules contain partitions. In AADL, these partitions are modeled with AADL `virtual processor` components that are either contained in or bound to an AADL `processor` component. These `virtual processors` subcomponents model partitions runtimes.

b. ARINC 653 module time slots are modeled with the `ARINC653::Module_Schedule` property attached to an AADL `processor` component. The property is a list that defines window schedules. List elements contains the following attributes:

i. `Duration`: The time slot duration.

ii. `Partition`: reference to the partition (`virtual processor`) associated with this slot.

iii. `Periodic_Periodic_Start`: specifies if all periodic tasks should start at the beginning of the partition execution.

c. The `ARINC653::Module_Major_Frame` property is associated with an AADL `processor` component and specifies the major time frame of an ARINC 653 module.

d. The `Process_Swap_Execution_Time` property from the core language specifies the time needed to switch from one partition to another. The value represents the time required to clean a partition and activate another (cache flush, memory segments change).

e. The `ARINC653::Module_Version` and `ARINC653::Module_Identifier` properties aim at adding a version to the module (potentially similar to the one used in the system configuration) as well as a textual description or other comments.

f. The ARINC 653 health monitoring properties usable at the module level are covered in the health monitoring section (A.13).

ARINC 653 entity	AADL entity	Properties
Module	`Processor`	• ARINC653::Module_Major_Frame • ARINC653::Module_Schedule • ARINC653::HM_Error_ID_Levels • ARINC653::HM_Error_ID_Actions • ARINC653::Module_Version • ARINC653::Module_Identifier • Process_Swap_Execution_Time

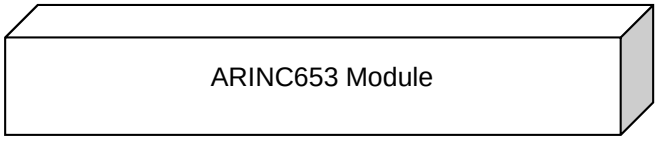


Figure 1 - Graphic representation of an ARINC 653 module (without partition runtime)

A.3 ARINC 653 partitions modeling

(9) An ARINC 653 partition conceptually consists of a separated address space and a specific runtime. This runtime manages resources within the partition and schedules ARINC 653 processes (that correspond to AADL `thread` components) that are executed in its address space. An ARINC 653 partition hosts the application software to be executed.

(10) Each ARINC 653 partition is represented by an AADL `process` component component bound to a `virtual processor` and one or several `memory` components. In AADL, `process` components are used to indicate address space protection for threads. The AADL `process` component and its association to a `memory` component model the partition address space. The AADL `virtual processor` component models the partition specific runtime environment provided by the core software.

a. The memory requirements of the ARINC 653 partition are specified by adding the `Data_Size` and `Code_Size` properties. They are added to the AADL `process` component that models the partition. In addition, `process` components are bound to the physical memory using the AADL `Actual_Memory_Binding` property.

b. The `virtual processor` component describes the partition-level scheduler and runtime requirements using AADL properties. Within each AADL `virtual processor` component, the property `Scheduling_Protocol` defines the scheduling policy used inside each ARINC 653 partition.

c. The `Actual_Processor_Binding` AADL property associates the components (`virtual processor` and `process`) that represent a partition. Each AADL `process` component (that contains the application software components) must be bound to an AADL `virtual processor` (that specified the partition runtime provided by the core software).

d. Partitions space isolation is specified by associating an AADL `process` component with one or several AADL `memory` components with the `Actual_Memory_Binding` property.

e. AADL `virtual processor` components (partition runtime) must be contained in or bound to AADL `processor` components (core software). In case one partition (AADL `virtual processor`) can be associated with several core modules (AADL `processor`), users must associate them using the property `Actual_Processor_Binding`.

f. The `Activate_Entrypoint_Source_Text` property specifies the name of a subprogram used to initialized the partition.

g. The Development Assurance Level (DAL) of the application software executed within a partition the `ARINC653::DAL` on its associated `virtual processor` component. It represents the DAL is the application software executed by the partition, not the DAL of the core software. Thus, all the software components contained in a partition should have a DAL value greater or equal to the one of the partition.

h. The `Thread_Swap_Execution_Time` specifies the potential cost of process switching inside a partition. When partition-level scheduler switches from one process to another, there is a potential switching time that should be taken in account for analysis. This property was designed to specify this overhead time so that system designers can specify scheduler overhead for each partition.

i. The `ARINC653::Error_Handling` property specifies the ARINC 653 process (AADL `thread` component) used to recover error raised at the partition level.

j. The `ARINC653::Partition_Id` defines an identifier that potentially corresponds to the one used by the underlying Operating System. The `ARINC653::Partition_Name` defines a partition name with a string that potentially uses natural language.

k. The `ARINC653::System_Partition` property indicates if the partition is an ARINC 653 system partition. A system partition is allowed to perform some specific operations (such as processing input/output)

l. The ARINC 653 health monitoring properties usable at the ARINC 653 partition level are covered in the health monitoring section (A.13).

ARINC 653 entity	AADL entities	Properties
Partition Runtime	`Virtual Processor`	• Scheduling_Protocol • Activate_Entrypoint_Source_Text • ARINC653::DAL • ARINC653::Partition_Name • ARINC653::Partition_Identifier • ARINC653::System_Partition • ARINC653::Error_Handling • ARINC653::HM_Error_ID_Actions • Thread_Swap_Execution_Time
Partition Software	`Process`	• Data_Size • Code_Size

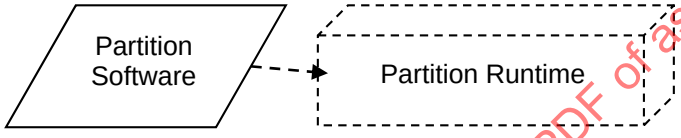


Figure 2 - Graphic representation of a partition and its association to partition runtime

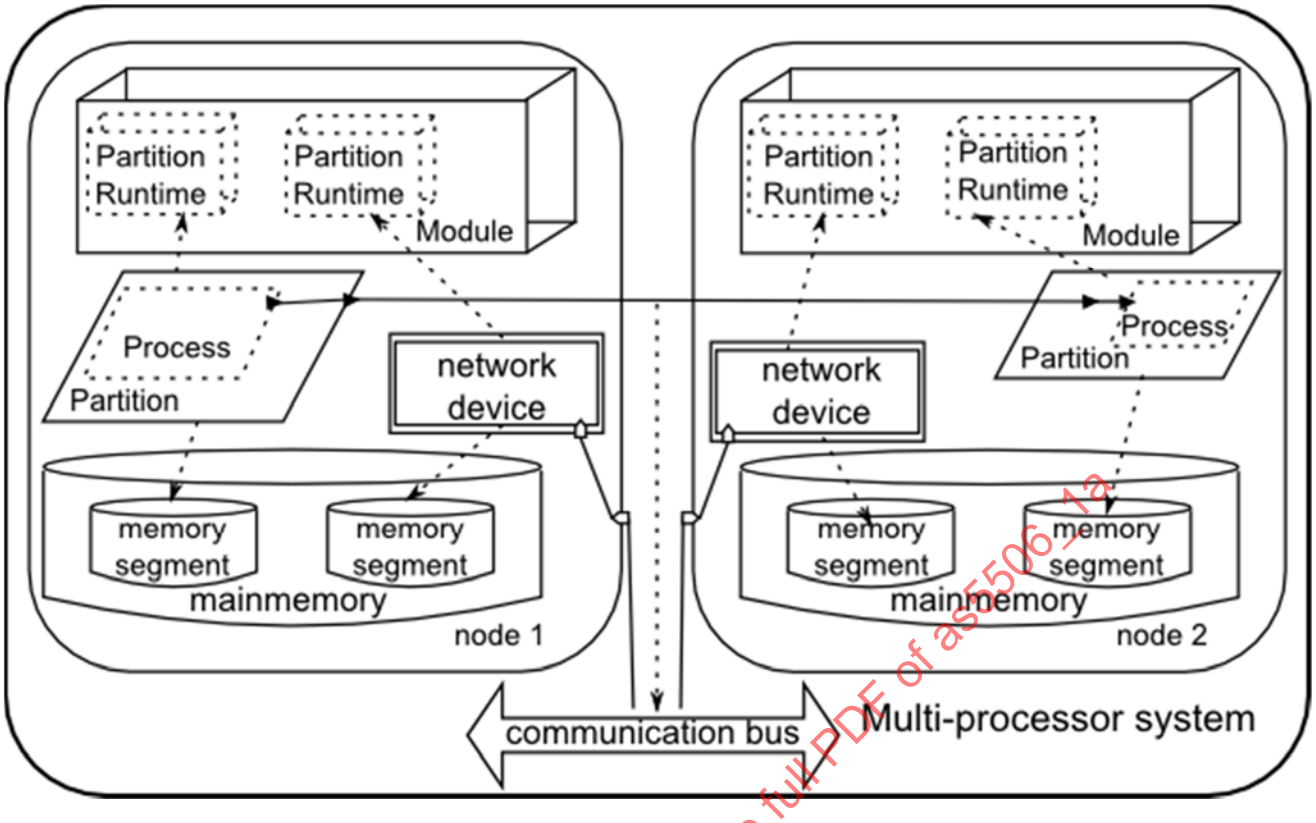
A.4 Multi processors architectures

(11) Multi-processor partitioned systems in ARINC 653 are represented by multiple physical processors, each with an ARINC 653 module defined to schedule the partitions on that processor. The AADL system component is a hierarchical component to integrate software and hardware components. It is used to model the multi-processor system with its partitions.

(12) Each node of the multi-processor is specified within an AADL `system` component. This top system component contains AADL `processor`, `virtual processor` and `process` with at least: one AADL `processor` (the ARINC module), one AADL `virtual processor` (the partition-level scheduler), one AADL `process` (the partition address space), one AADL `memory` component to bind it to, and one AADL `thread` within the AADL `process` component (the ARINC 653 process executed in a partition).

(13) A higher level AADL `system` component is used to contain multiple system components reflecting a multiprocessor system.

(14) Communication between ARINC 653 modules is modeled with event data ports and data ports. In ARINC653, it corresponds to inter-partitions communications through different processors. See section A.6 for modeling of inter-partitions communication channels.



A.5 ARINC 653 processes modeling

(15) ARINC 653 processes execute the application software that consists in code in their partition's address space. Each ARINC 653 process is associated with a set of requirements: entry point, stack size, period, priority, time capacity and deadline type. List of relevant properties properties are listed in the table below.

(16) ARINC 653 processes are mapped to AADL with the AADL `thread` component.

a. The ARINC 653 base priority concept is mapped to the AADL `Priority` property from the standard AADL property set. The meaning of priority is dependent on the scheduling algorithm used.

b. The ARINC 653 stack size concept is mapped in AADL using the `Stack_Size` property from the standard property set.

c. The ARINC 653 entrypoint concept is mapped in AADL using the `Initialize_Entrypoint` property from the standard property set.

d. The ARINC 653 period concept is mapped in AADL using the `Period` property from the standard property set.

e. The ARINC 653 deadline concept is mapped in AADL using the `Deadline` property from the standard property set.

f. The ARINC 653 time capacity concept is specified with AADL using the `ARINC653::Time_Capacity` property from the AADL standard property set

g. The ARINC 653 deadline type concept (soft or hard) is specified with the AADL enumeration `Deadline_Type` associated to an AADL thread. The value can be either `soft` or `hard`.

h. The ARINC 653 health monitoring properties usable at the ARINC 653 process level are covered in the health monitoring section (A.13).

i. The core AADL property `Dispatch_Protocol` specifies the type of process (periodic, aperiodic, sporadic).

ARINC 653 entity	AADL entity	Properties
Process	`Thread`	• ARINC653::HM_Error_ID_Actions • Code_Size • Data_Size • Heap_Size • Stack_Size • Initialize_Entrypoint • Dispatch_Protocol • Compute_Execution_Time • Deadline • Period • Priority • Deadline_Type • Time_Capacity

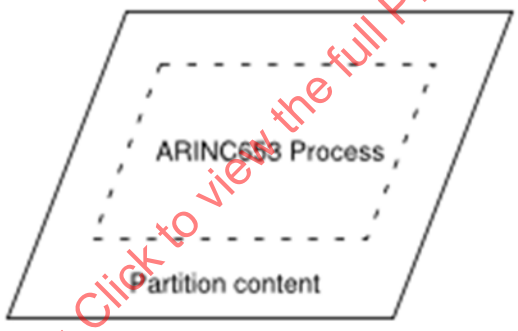


Figure 3 - Graphic representation of an ARINC 653 process inside a partition

A.6 ARINC 653 inter-partition communication modeling

(17) In ARINC 653, inter-partition communication represents data exchange across partitions (one partition sends data to other partitions). The ARINC 653 standard defines two kinds of inter-partition communication: queuing ports and sampling ports. ARINC 653 queuing ports store and queue each instance of data so that receivers can read each queued data element. ARINC 653 sampling ports keep the most recent value (latest data instance replaces prior values). Both ARINC 653 queuing and sampling ports have timing requirements (refresh of data, queuing policy, etc.).

(18) ARINC 653 inter-partition communications are specified using AADL `ports` connected between AADL `process` components.

(19) Identifiers for ARINC 653 inter-partitions may be defined in AADL in one of two ways:

a. Identifiers of the ARINC 653 sampling and queuing ports are derived from the name of the AADL `features` inside the AADL `process` component.

b. Identifiers are explicitly defined by the system designer using the AADL `Source_Name` property. In that case, the `Source_Name` property is added on AADL `data ports` or `event data ports`.

(20) ARINC 653 sampling ports are specified using AADL `data ports` inside an AADL `process` component, originating at an AADL `thread` component (ARINC 653 process).

a. The size of ARINC 653 sampling ports (AADL `data ports`) is deduced from the size the AADL `data` component associated with the port. It corresponds to the maximum message size option of an ARINC 653 sampling port.

b. The refresh period of ARINC 653 sampling ports is specified with the `ARINC653::Sampling_Refresh_Period` property. This property is only relevant on AADL `in data port` since the refresh period is only used on the receiver side.

(21) The modeling of ARINC 653 queuing ports is made with the declaration of `event data ports` in an AADL process component,

a. Size of ARINC 653 queuing ports (AADL `event data ports`) is deduced from the size of its associated `data` component and the port's queue size. The size of the queue (number of elements that can be stored) is specified with the property `Queue_Size`. It corresponds to the size used when the port is created.

b. The timeout of ARINC 653 queuing ports is modeled using the `ARINC653::Timeout` property. It is used for sending data (a sender process sending the data may be blocked until there is enough space to store its message) and receiving (a receiver process may be blocked until there is some data to read in the queue) data.

c. The modeling of queuing discipline of ARINC 653 queuing ports is achieved with the `ARINC653::Queueing_Discipline` property. Supplied values for this property are `FIFO` and `By_Priority`.

(22) In ARINC 653, ports are connected through channels. A channel connects one source port to at least one destination port (section 2.3.5.1 of the ARINC 653 standard). AADL can connect one source port to several destination ports (section 9.2.2 of the AADL standard). Thus, AADL connections fit with the semantics of ARINC 653 channels. In consequence, the modeling of ARINC 653 channels is specified by connecting one AADL [`event`] `data port` to at least another AADL `data port` or `event data port`.

ARINC 653 entity	AADL entity	Properties
Queuing ports	Connection of `event data ports` between `process` components	• Queue_Size • ARINC653::Queueing_Discipline • ARINC653::Timeout • Source_Name
Sampling ports	Connection of `data ports` between `process` components	• ARINC653::Sampling_Refresh_Period • Source_Name

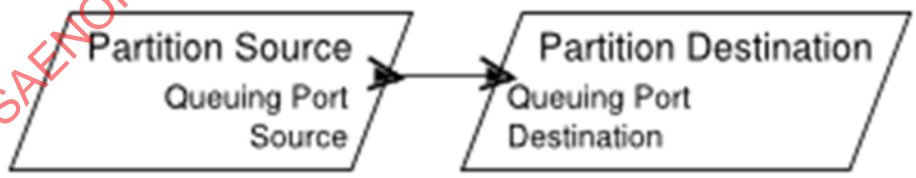


Figure 4 - Graphic representation of an ARINC 653 queuing port connected through two partitions

Figure 5 - Graphic representation of an ARINC 653 sampling port connected through two partitions

A.7 ARINC 653 intra-partition communication modeling

(23) In the ARINC 653 standard, processes contained within the same partition can exchange data using intra-partition communications. The ARINC 653 standard defines four mechanisms of intra-partition communications: buffer, blackboard, event and semaphore.

(24) ARINC 653 intra-partition communication is specified in AADL models using connections and data accesses between AADL `thread` components. Communication is considered as intra-partition communication as long as it is contained within the AADL `process` and involves only `thread` components located in the same `process`.

(25) ARINC 653 intra-partition ports identifiers may be defined in AADL in one of two ways:

a. The identifier can be deduced from the name of its corresponding AADL entity (shared `data` component, `[event] data port`).

b. System designers can specify another name for the port using the `Source_Name` property. This property is added to AADL ports or data and designate.

A.8 ARINC 653 buffers modeling

(26) ARINC 653 buffers provide a mechanism to exchange data across ARINC 653 processes located in the same partition. Values are put in a queue and the receiver receives each queued value.

(27) ARINC 653 buffers are modeled in AADL with `event data ports` in `thread` components

a. The numbers of elements in the queue is specified using the `Queue_Size` property associated with the AADL port.

b. The max size of an element in the queue is deduced from the data classifier associated with the AADL port.

c. The AADL `ARINC653::Queueing_Discipline` property indicates what kind of queuing protocol is used. This property is associated with an AADL `event data port.` Supplied values for this property are `FIFO` and `By_Priority`.

ARINC 653 entity	AADL entity	Properties
Buffers	Connection of `event data ports` between thread components located in the same process.	• Queue_Size • ARINC653::Queueing_Discipline • Source_Name

Figure 6 - Graphic representation of an ARINC 653 buffer communication channel

A.9 ARINC 653 blackboards modeling

(28) ARINC 653 blackboards enable data sharing across ARINC 653 processes located in the same ARINC 653 partition. Unlike ARINC 653 buffers, ARINC 653 blackboards do not keep each instance of the data and only store the most recent received value. Designers can specify a timeout when an ARINC 653 process reads data on a blackboard: a reading process can wait for data until a specified timeout.

(29) ARINC 653 blackboards can be specified with AADL using `data ports` or shared `data` components. The following paragraphs detail both modeling patterns.

(30) ARINC 653 blackboards can be specified with AADL using a shared `data` component across several AADL `thread` components.

a. When shared data is used to model an ARINC 653 blackboard, the size of the ARINC 653 blackboard is deduced from the size of the shared AADL `data` component.

b. The AADL `ARINC653::Timeout` property is associated with an AADL `data access` to specify the timeout used to read the ARINC 653 blackboard.

c. Modeling ARINC 653 blackboard usage by an ARINC 653 process is achieved by an AADL `data access` feature in an AADL `thread` component.

(31) ARINC 653 blackboards can also be specified with AADL using `data ports`. These AADL `data ports` are connected between AADL `thread` components located in the same partition.

a. When using AADL `data port` to represent ARINC 653 blackboard, the size of the specified ARINC 653 blackboard is deduced from the size of the `data` type associated with the AADL `data port`.

b. The AADL `ARINC653::Timeout` property is associated with the `data port` to model the timeout used to read a blackboard.

c. Modeling ARINC 653 blackboard sharing across several threads is achieved by connecting `data ports` between AADL `thread` components.

ARINC 653 entity	AADL entity	Properties
Blackboard	• `Data` component shared between `thread` components located in the same `process`. • `Data ports` connected between `thread` components located in the same `process`.	• ARINC653::Timeout • Source_Name

Figure 7 - Graphic representation of an ARINC 653 blackboard communication channel

A.10 ARINC 653 events modeling

(32) ARINC 653 events are used to synchronize ARINC 653 processes (AADL thread `components`) located in the same ARINC 653 partition (AADL `process` component). They support control flow between processes by notifying occurrences of conditions to awaiting ARINC 653 processes.

(33) ARINC 653 events are mapped using AADL `event ports` between processes.

a. The `ARINC653::Timeout` property is used on `in event ports` to model timeout used by an ARINC 653 process (AADL `thread` component) when it is waiting for the event.

ARINC 653 entity	AADL entity	Properties
Events	Connection of `event ports` between `thread` components located in the same process.	• ARINC653::Timeout • Source_Name

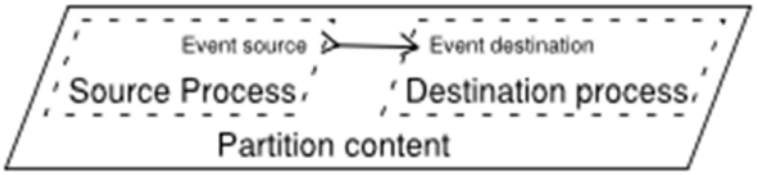


Figure 8 - Graphic representation of an ARINC 653 event communication channel

A.11 ARINC 653 semaphores modeling

(34) ARINC 653 semaphores synchronize ARINC 653 processes located in the same partition and provide controlled access to resources. Unlike ARINC 653 events, ARINC 653 semaphores can have a queuing discipline that defines queuing mechanisms for waiting processes.

(35) In ARINC 653, semaphores are used to protect resources from concurrent access. In AADL, resources are modeled using the `data` component. The use of shared data is achieved using `data access` features. The use of a semaphore is modeled using the property `Concurrency_Control_Protocol` with the value set to `Protected_Access`.

a. AADL shared `data` components that use ARINC 653 semaphores to avoid concurrent access must be contained in a process component in order to be shared with several threads. The access to the data is achieved with AADL `data access` features.

b. The AADL `ARINC653::Queueing_Discipline` property indicates what kind of queuing protocol is used to dispatch tasks waiting on the semaphore. This property is associated with an AADL `data access`. Supplied values for this property are `FIFO` and `By_Priority`.

ARINC 653 entity	AADL entity	Properties
Semaphore	Data inside `process` and `data access` for each `thread` that needs it.	• ARINC653::Timeout • ARINC653::Queueing_Discipline • Source_Name

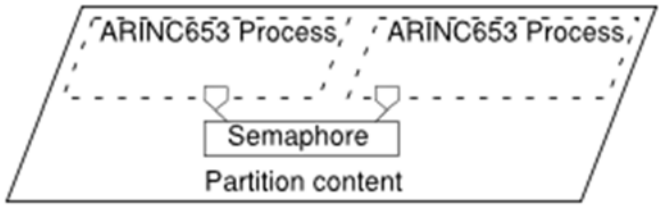


Figure 9 - Graphic representation of an ARINC 653 semaphore usage across two ARINC 653 processes

A.12 ARINC 653 memory requirements modeling

(36) System designers may want to model partitions isolation in different memory segments. To do that, `memory` components that represent physical memory contain several `memory` subcomponents (add other `memory` components as subcomponents). In this hierarchy of AADL `memory` components, the root `memory` component models the hardware memory whereas the AADL `memory` subcomponents model logical separation of the memory (memory segments where partitions store code and data).

(37) Requirements of `memory` component that models memory segments are specified with AADL using the `Base_Address` property (which correspond to the base address of the segment in the ARINC 653 architecture) and the `Memory_Size` property (which correspond to the size of the word in this memory component).

(38) Each AADL `process` component (part of the modeling of ARINC 653 partitions) is associated (bound) with an AADL `memory` component using the `Actual_Memory_Binding` property from the standard property set. It specifies the deployment of ARINC 653 partitions (AADL `process` component) on physical memory (AADL `memory` component).

(39) The ARINC 653 standard uses space isolation across partitions. Consequently, the binding rule between partitions and memory implies that an AADL `memory` component is bounded to exactly one partition (AADL `process`).

(40) ARINC 653 defines three memory access types: read, write and execute. ARINC 653 memory accesses and permissions are specified with AADL by associating the `Memory_Protocol` property with an AADL `memory` component.

(41) ARINC 653 requires that memory content is explicitly specified (if a memory contains code, data or both). This requirement is specified by adding the `ARINC653::Memory_Kind` property on an AADL `memory` component.

ARINC 653 entity	AADL entity	Properties
Memory Requirements	Describe memory requirements for `process` components and specify the allocation of partitions (AADL `process` component) on hardware memory (AADL `memory` component).	• Actual_Memory_Binding • Memory_Size • Base_Address • Memory_Protocol • ARINC653::Memory_Kind

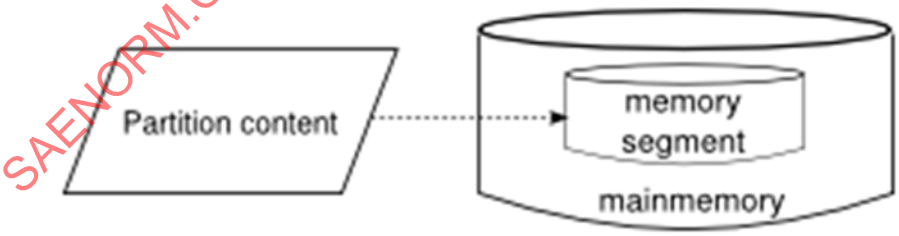


Figure 10 - Graphic representation of memory segments inside the main memory and process binding to memory segments

A.13 ARINC Health Monitor (HM) modeling

(42) In ARINC 653, errors can be managed at different levels: module level, partition level and process level. These levels correspond to the following AADL components: `processor`, `virtual processor` and `thread`. Moreover, this is the responsibility of the system integrator to indicate which error is handled at each level. The actual ARINC 653 enumerates faults that can be raised in each level of an ARINC 653 architecture.

(43) The AADL property `ARINC653::HM_ID_Levels` specifies the list of all potential errors that can be triggered within an ARINC 653 modules (AADL `processor`). This list contains information for these errors using an AADL record type (HM_Error_Level_Type) that specifies

a. the error identifier (using the `ErrorIdentifier` field)

b. the error description (using the `Description` field)

c. the containment level of the error (`Module_Level`, `Partition_Level` or `Processor_Level` – as defined by the `Error_Level_Type` AADL type.)

d. the ARINC 653 related error code (`ErrorCode`) specified by an `ARINC653::Supported_Error_Code` value.

(44) The AADL property `ARINC653::HM_Error_ID_Actions` specifies health-monitoring actions at each level. This property is added on AADL `processor` (ARINC 653 module level), AADL `virtual processor` (ARINC 653 partition) or AADL `thread` (ARINC 653 process). The `ARINC653::HM_Error_ID_Actions` specifies a list that contains informations for error handling:

a. the error identifier (using the `ErrorIdentifier` field) related to the action. This value is a reference to an existing identifier used by the `ARINC653::HM_ID_Levels` property

b. the action description (using the `Description` field) that details the action using natural language.

c. the undertaken action performed (using the `Action` field) to recover the error. This can be a textual description or a reference to the name of the procedure provided by the underlying ARINC 653 runtime.

(45) The ARINC 653 standard states that list of errors and actions are defined by the operating system provider. In consequence, users of the ARINC 653 annex can redefine the list of possible errors in the property set as in the property set values. To do so, users can extend the list of errors specified by the AADL type `ARINC653::Supported_Error_Code`.

(46) In addition, system designers may want to model errors detection and recovery mechanisms to analyze error impact or fault propagation. Modeling of faults or errors and their impact is beyond the scope of this annex. However, errors and recovery procedures can be specified using the Error Modeling Annex of the AADL. This annex provides a suitable semantics to model errors and their propagation in a layered architecture.

A.14 ARINC 653 modes/states modeling

(47) The ARINC 653 standard describes four operational modes for partitions (COLD START, WARM START, NORMAL and IDLE) as well as several states for the system (Module Init, Module Function, Partition Init, etc.). This list of states might be extended on a user and implementation basis.

(48) ARINC 653 modes and states could be mapped to AADL `modes`. To do so, the `State_Information` property provides a way to establish a connection between this two concepts. The `State_Information` is added to an AADL model and contains

a. An identifier (field `Identifier`) that references the related ARINC 653 mode identifier

b. A description (field `Description`) that describes the ARINC 653/AADL mode using natural language.

A.15 ARINC 653 application-specific Hardware & Device Drivers

(49) In ARINC 653, application-specific hardware is controlled by device drivers that reside in core software. The device driver can be located either at the kernel or partition level. Therefore, they can be located within main module functionalities or isolated in a distinct partition that has its own resources and services. In ARINC 653, such a partition is called system partition, because it provides dedicated services and functions to communicate with the hardware although it is contrained by time and space partitioning. Consequently, system designers and operating system providers can have different implementation strategies to implement device drivers in ARINC 653 systems.

(50) These implementation strategies impact system functionalities and performances. Our modeling patterns are designed for the specification of these two different implementation strategies so that system designers can precisely describe their system and analyze impact of their implementations strategies.

(51) Application-specific hardware (network interface, sensor, etc.) is specified with the AADL `device` hardware component. On the other hand, the device driver (software that controls the device) is described by adding an AADL `abstract implementation` component and associated with the AADL `device` component using `Device_Driver` property.

(52) The `abstract` component associated with the AADL `device` component contains all the necessary components to control the application-specific hardware. It is composed by AADL components (`thread`, `data`, etc.) with their properties and requirements (timing, memory, etc.). Modeling of device drivers internals describe the underlying operating system and thus, allow system designer to analyze their impact on the overall system (in terms of performance, latency, etc.)

(53) AADL `device` components must be associated with an AADL `processor` (an ARINC 653 module) or an AADL `virtual processor` (an ARINC 653 partition).

a. If the AADL `device` component is bound to an AADL `processor`, it means that the device driver resides in the ARINC 653 module.

b. If the `device` component is bound to an AADL `virtual processor`, it means that the device driver resides in a system partition and uses time and space isolation mechanisms of the partition. In this case, the partition is considered as a system partition.

These two binding mechanisms illustrate the different implementation strategies for device drivers in ARINC 653 systems and thus, ease the analysis of their impact on the overall architecture.

ARINC 653 entity	AADL entity	Properties
Device drivers	AADL `device` component. The associated device driver is described using the `Device_Driver` property of the standard property set. This `device` component is bound to AADL `processor` or AADL `virtual processor` components.	• Device_Driver • Actual_Processor_Binding

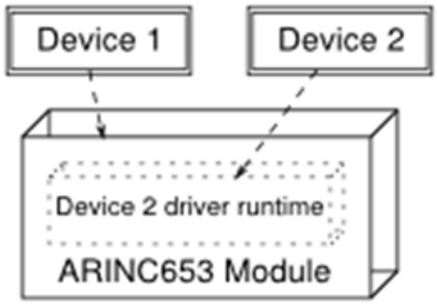


Figure 11 - Graphic representation of device modeling in ARINC 653 architectures with both different binding strategies

A.16 Summary of modeling rules

ARINC 653 entity	AADL entity	Properties
Module	`Processor`	• ARINC653::Module_Major_Frame • ARINC653::Module_Schedule • ARINC653::HM_Error_ID_Levels • ARINC653::HM_Error_ID_Actions • ARINC653::Module_Version • ARINC653::Module_Identifier • Process_Swap_Execution_Time • Scheduling_Protocol
Partition	`Virtual Processor`	• Scheduling_Protocol • ARINC653::DAL • ARINC653::Partition_Name • ARINC653::Partition_Identifier • ARINC653::System_Partition • ARINC653::Error_Handling • ARINC653::HM_Error_ID_Actions • Activate_Entrypoint_Source_Text • Thread_Swap_Execution_Time
	`Process`	• Data_Size • Code_Size
Process	`Thread`	• ARINC653::HM_Error_ID_Actions • Code_Size • Data_Size • Heap_Size • Stack_Size • Initialize_Entrypoint • Compute_Execution_Time • Deadline • Period • Priority • Proprity_Type • Time_Capacity • Dispatch_Protocol
Queuing ports	Connection of `event data ports` between `process` components	• Queue_Size • ARINC653::Queueing_Discipline • ARINC653::Timeout • Source_Name
Sampling ports	Connection of `data ports` between `process` components	• ARINC653::Sampling_Refresh_Period • Source_Name
Buffers	Connection of `event data ports` between threads components located in the same process.	• Queue_Size • ARINC653::Queueing_Discipline • Source_Name
Blackboards	• `Data` component shared between `thread` components located in the same `process`.	• ARINC653::Timeout • Source_Name

	• `Data ports` connected between `thread` components located in the same `process`.	
Semaphore	Data contained in a `process` component and shared between several `thread` components.	• ARINC653::Timeout • Source_Name • ARINC653::Queueing_Discipline
Events	Connection of `event ports` between `thread` components located in the same process.	• ARINC653::Timeout • Source_Name
Memory Requirements	Describe memory requirements for ARINC 653 process (AADL `thread` components) and specify the allocation of ARINC653 partitions (AADL `process` component) on hardware memory (AADL `memory` component).	• Actual_Memory_Binding • Memory_Size • Base_Address • Memory_Protocol • ARINC653::Memory_Kind
Device drivers	AADL `device` component. The associated device driver is described using the `Device_Driver` property of the standard property set. This `device` component is bound to AADL `processor` or AADL `virtual processor` components.	• Device_Driver • Actual_Processor_Binding
Mode	AADL `mode`. The mode can then represent an ARINC 653 mode.	• ARINC653::State_Information

A.17 ARINC 653 Property Set

property set ARINC653 **is**

Module_Major_Frame : **Time**

applies to (processor);

The Module_Major_Frame property specifies the major frame for the ARINC 653 module (AADL processor component).

Sampling_Refresh_Period : **Time**

applies to (data port);

The Sampling_Refresh_Period property indicates data arrival rate for an in data port. It corresponds to the concept of refresh time of ARINC 653 sampling port.

```
Supported_Error_Code: type enumeration
              (Module_Config, Module_Init, Module_Scheduling,
               Partition_Scheduling, Partition_Config,
               Partition_Handler, Partition_Init, Deadline_Miss,
               Application_Error, Numeric_Error,
               Illegal_Request, Stack_Overflow,
               Memory_Violation, Hardware_Fault, Power_Fail);
```

The Supported_Error_Code enumeration corresponds to the possible Error code that can be raised at the different levels of an ARINC 653 architecture (module, partition, process). The list of possible values is implementation dependent and can be modified by the system designer.

```
Supported_Memory_Kind : type enumeration
   (memory_data, memory_code);
```

The Supported_Memory_Kind enumeration describes possible content of an AADL memory component.

Memory_Kind : **list of (Supported_Memory_Kind)**

applies to (memory);

The Memory_Kind property describes the content of an AADL memory component.

Timeout : **Time**

applies to (event data port,data port,event port,access connection);

The Timeout property specifies the timeout used by an ARINC 653 process when sending/receiving a data. Depending on which component it is used, it could be useful for sender or receiver side.

```
Supported_DAL_Type : type enumeration
   (LEVEL_A, LEVEL_B, LEVEL_C, LEVEL_D, LEVEL_E);
```

The Supported_DAL_Type enumeration corresponds to the different Development Assurance Levels supported by the ARINC 653 standard.

DAL : **Supported_DAL_Type**

applies to (virtual processor, process, thread, subprogram);

The DAL property defines the Development Assurance Level of a component. It is associated to software component to capture their associated DAL. When applied to a virtual processor, this is a requirement for this partition and all software components associated by this partition must have at least the same or higher DAL value.

Module_Version : **aadlstring applies to** (processor);

The Module_Version property adds a description to the ARINC 653 module (textual comments using natural language).

Module_Identifier: **aadlstring applies to** (processor);

The Module_Identifier property specifies the ARINC 653 identifier for a module specifies with an AADL processor component.

```
Partition_Name: aadlstring applies to (virtual processor);
```

The Partition_Name property defines the name for a specific partition with natural language.

```
Partition_Identifier: aadlinteger applies to (virtual processor);
```

The Partition_Identifier property defines the identifier of the partition that potentially corresponds to the one used by the execution platform.

```
System_Partition: aadlboolean applies to (virtual processor);
```

The System_Partition property specifies if a given partition is operating as a system partition (performing I/O or other operations requiring special privileges).

```
Error_Handling: reference (thread) applies to (virtual processor);
```

The Error_Handling property specifies the ARINC 653 process (AADL thread) operating within the partition as the error handler. This ARINC 653 process is then supposed to receive notification of errors and take appropriate actions to recover them.

```
Error_Level_Type: type enumeration
                  (Module_Level, Partition_Level, Process_Level);
```

The Error_Level_Type type lists all potential error levels: in a module (AADL processor), partition (AADL virtual processor) or process (AADL thread).

```
HM_Error_ID_Level_Type: type record (
      ErrorIdentifier : aadlinteger;
      Description     : aadlstring;
      ErrorLevel      : ARINC653::Error_Level_Type;
      ErrorCode       : ARINC653::Supported_Error_Code;);
```

The HM_Error_ID_Level_Type property records all information related to a fault that may occur within a module. ErrorIdentifier is unique and is then re-used by the HM_Error_ID_Actions property. The Description provides a basic description using natural language. The ErrorLevel indicates at which level the error occurs while the ErrorCode designates the related error (such as scheduling error).

```
HM_Error_ID_Levels: list of ARINC653::HM_Error_ID_Level_Type
                                applies to (processor);
```

The HM_Error_ID_Levels property lists all errors that may occurs within a module using the HM_Error_ID_Level_Type type.

```
HM_Error_ID_Action_Type : type record (
      ErrorIdentifier : aadlinteger;
      Description     : aadlstring;
      Action          : aadlstring;);
```

The HM_Error_ID_Action_Type type lists all useful information related to error recovery. The ErrorIdentifier is the unique identifier for an error and is one included with the HM_Error_ID_Levels property. The Description is a textual description of the error using natural language. The Action is a string describing the action, potentially the name of a function used in the underlying ARINC 653 runtime to recover the error.

```
HM_Error_ID_Actions: list of ARINC653::HM_Error_ID_Action_Type
                 applies to (processor, virtual processor, thread);
```

The HM_Error_ID_Actions is a list of action used to recover an error, either at the module (AADL processor), partition (AADL virtual processor) or process (AADL thread) level.

```
State_Information_Type: type record (
      Identifier      : aadlinteger;
      Description     : aadlstring;);
```

The State_Information_Type type contains all information related to an ARINC 653 mode: an Identifier (that is potentially similar to the one of the ARINC 653 implementation) and a Description that uses natural language.

```
State_Information: ARINC653::State_Information_Type applies to (mode);
```

The State_Information property associates an ARINC 653 mode with an AADL mode.

```
Queueing_Discipline_Type: type enumeration (Fifo, By_Priority);
```

The Queueing_Discipline_Type type lists all potential queueing discipline on ARINC 653 communication mechanisms (such as ARINC 653 queueing ports or ARINC 653 buffers).

```
Queueing_Discipline: ARINC653::Queueing_Discipline_Type
                                applies to (port);
```

The Queueing_Discipline property reflects the queueing discipline of the ARINC 653 runtime for a given port. It indicates the dispatching policy for an ARINC 653 process (AADL thread component) waiting for a new value on a port.

```
Schedule_Window : type record (
   Partition : reference (virtual processor, processor);
   Duration  : time;
   Periodic_Processing_Start : aadlboolean;
);
```

The Schedule_Window type specifies a time slot of the ARINC 653 module. The Partition attribute represents the partition assigned to this slot. The Duration attribute capture the time allocated to the partition. The Periodic_Processing_Start specifies if all periodic tasks should start when the partition is activated.

```
Module_Schedule : list of ARINC653::Schedule_Window
  applies to ( processor, virtual processor );
```

The Module_Schedule property the schedule time slots in the module. This is a list of all time slots assigned for each partition.

```
Deadline_Type : enumeration (soft, hard)
  applies to ( thread );
```

The Deadline_Type property specifies the kind of deadline associated for a task. It corresponds to the same attribute associated to an ARINC 653 process.

```
Time_Capacity : Time
  applies to ( thread );
```

The Time_Capacity represents the time allocated to each task.

```
end ARINC653;
```

APPENDIX A - INFORMATIVE SECTION

This informative section illustrates ARINC 653 systems modeling by providing examples and validation examples.

A.18 System validation using the ARINC 653 annex

(54) The ARINC 653 standard defines an XML-style notation to describe system requirements and deployment of an ARINC 653 architecture.

(55) Here is a partial list of what could be validated from the AADL model:

a. Scheduling of each ARINC 653 module. AADL models describe partitions content, including threads (ARINC 653 processes) and intra-partition communication. Thanks to this finer-grain modeling, the user can verify partition-level scheduling, as well as the scheduling of the overall system.

b. Bus load. With AADL, system designers can precisely specify which connections use a bus. Thus, the user can verify the bus loading and if data might be delayed, due to various issues (overload, …)

(56) In general, modeling ARINC 653 systems with AADL would offer many benefits, because the language models the complete runtime system with its requirements, not just a subset of the system. Thus, using a model-based approach to design ARINC 653 systems with AADL can provide useful verification features.

(57) The definition of the verification rules is beyond the scope of this annex. Readers would refer to the toolset that supports this AADL ARINC 653 annex.

A.19 Example with one module

(58) The following example shows a system with two partitions. It shows the components involved in the modeling of ARINC 653 system and illustrates the mapping of ARINC 653 concepts to the AADL.

a. The first ARINC 653 partition is modeled with the AADL process `partition1_process` (which models the space isolation) bound to the AADL virtual processor `part1` (which models intra-partition runtime).

b. In the same way, the second partition is modeled with the process `partition2_process` and the virtual processor `part2`.

c. The scheduling algorithm used in each partition is specified in the `virtual processor` component that models the runtime within each partition.

d. The main memory (`mem`) is divided into two memory components (`part1em` and `part2em`). Each partition process (`partition1_process` and `partition2_process`) is bound to a memory component.

e. The ARINC 653 module (kernel) schedules the partitions on the processor and therefore provides the runtime at the processor level. In AADL, the `processor` concept includes the runtime environment. Therefore, the ARINC 653 kernel and the physical processor itself are specified with the AADL `processor` component. It contains properties (`ARINC653::Module_Schedule`, `ARINC653::Module_Major_Frame`) that describe scheduling requirements (major time frame, window slots and their allocation).

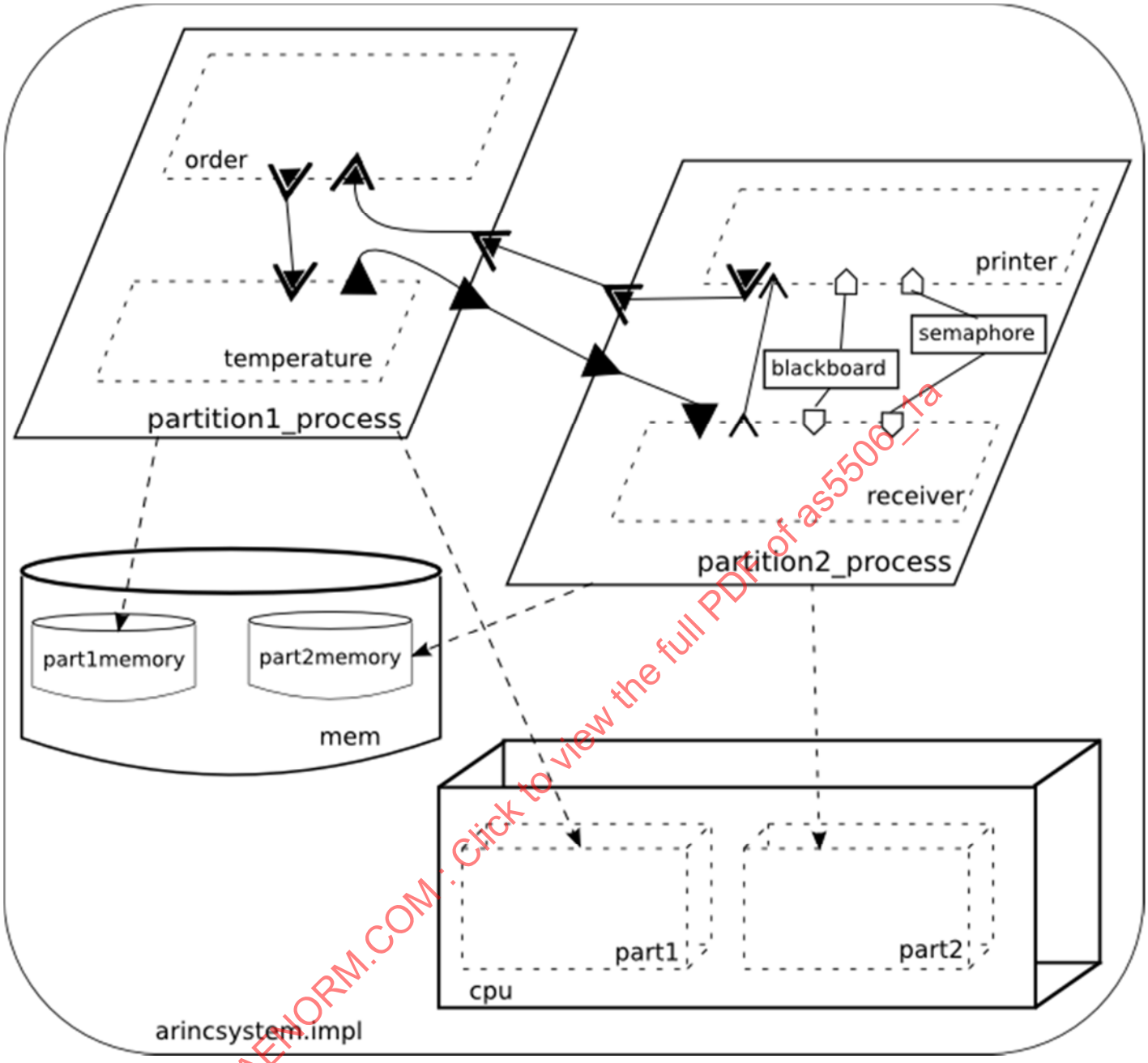
f. In the first partition (`partition1_process`), an intra-partition communication that uses ARINC 653 buffers is specified (AADL `event data ports`). This communication is made between AADL `thread` components representing ARINC 653 processes (order and temperature).

g. In the second partition (`partition2_process`), an intra-partition communication that corresponds to blackboards is added (AADL shared data component). This communication is made between the two threads of the partition (`printer` and `receiver`).

h. In the second partition (`partition2_process`), a communication using AADL `event ports` between two AADL `thread` components (ARINC 653 processes) is added.

i. A shared `data` component between the two threads (`printer` and `receiver`) of the second partition is added. The shared data is protected from concurrent accesses using ARINC 653 semaphores.

(59) There is the graphical representation of our example.



Graphic representation of our example

(60) There is the textual representation of our example. The textual representation includes properties on each component.

```
package arincexample1
public

with ARINC653;

data integer
end integer;

data ordercmd
end ordercmd;

data protected_data
```

```
properties
   Concurrency_Control_Protocol => Protected_Access;
end protected_data;

-- Now, declare the virtual processors that model
-- partition runtime.
virtual processor partition1_rt
properties
      Scheduling_Protocol => (RMS);
end partition1_rt;

virtual processor implementation partition1_rt.impl
end partition1_rt.impl;

virtual processor partition2_rt
properties
      Scheduling_Protocol => (RMS);
end partition2_rt;

virtual processor implementation partition2_rt.impl
end partition2_rt.impl;

subprogram  sensor_temperature_spg
end         sensor_temperature_spg;

subprogram  sensor_receiveinput_spg
end         sensor_receiveinput_spg;

subprogram  commandboard_receiveinput_spg
end         commandboard_receiveinput_spg;

subprogram  commandboard_printinfos_spg
end         commandboard_printinfos_spg;

-- Threads for the first partition
thread sensor_temperature_thread
features
   tempout: out data port integer;
   order: in event data port ordercmd;
properties
   Initialize_Entrypoint =>
classifier (arincexample1::sensor_temperature_spg);
   Priority => 42;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   Compute_Execution_Time => 10 ms .. 12 ms;
   Deadline => 40 ms;
end sensor_temperature_thread;

thread implementation   sensor_temperature_thread.impl
end                     sensor_temperature_thread.impl;

thread sensor_receiveinput_thread
features
   commandin: in event data port integer;
   order: out event data port ordercmd;
properties
   Initialize_Entrypoint =>
classifier (arincexample1::sensor_receiveinput_spg);
```

```
   Priority => 10;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   Compute_Execution_Time => 8 ms .. 10 ms;
   Deadline => 40 ms;
end sensor_receiveinput_thread;

thread implementation   sensor_receiveinput_thread.impl
end                     sensor_receiveinput_thread.impl;

--  Threads for the second partition
thread commandboard_receiveinput_thread
features
   temp: in data port integer;
   tempavg : requires data access integer {ARINC653::Queueing_Discipline => FIFO;};
   newavg: out event port;
   need_semaphore : requires data access protected_data  {ARINC653::Queueing_Discipline => FIFO;};
properties
   Initialize_Entrypoint =>
classifier (arincexample1::commandboard_receiveinput_spg);
   Priority => 42;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   ARINC653::Time_Capacity => 7 ms;
   Compute_Execution_Time => 5 ms .. 7 ms;
   Deadline => 40 ms;
end commandboard_receiveinput_thread;

thread commandboard_printinfos_thread
features
   ordersensor: out event data port integer;
   tempavg : requires data access integer  {ARINC653::Queueing_Discipline => FIFO;};
   newavg: in event port;
   need_semaphore : requires data access protected_data  {ARINC653::Queueing_Discipline => FIFO;};
properties
   Initialize_Entrypoint =>
classifier (arincexample1::commandboard_printinfos_spg);
   Priority => 43;
   Stack_Size => 100 Kbyte;
   Period => 20 ms;
   ARINC653::Time_Capacity => 6 ms;
   Compute_Execution_Time => 2 ms .. 6 ms;
   Deadline => 40 ms;
end commandboard_printinfos_thread;

-- Now, declare process that model partition address space
process partition1_process
features
   queueingin: in event data port integer
{Queue_Size => 4;
                           ARINC653::Timeout => 5ms;
                           ARINC653::Queueing_Discipline => FIFO;};
   samplingout: out data port integer;
end partition1_process;

process implementation partition1_process.impl
subcomponents
   temperature : thread sensor_temperature_thread.impl;
   order       : thread sensor_receiveinput_thread.impl;
```

```
connections
  bufferconnectionexample: port order.order -> temperature.order;
   c1 : port queueingin -> order.commandin;
   c2 : port temperature.tempout -> samplingout;
end partition1_process.impl;

process partition2_process
features
   queueingout: out event data port integer {ARINC653::Timeout => 10ms;};
   --  In the context of a event data port, the ARINC653::Timeout property
   --  is the timeout we used in the APEX functions.
   samplingin: in data port integer
{ARINC653::Sampling_Refresh_Period => 10ms;};
   --  The ARINC653::Timeout apply only to in data port. It is the refresh
   --  period for sampling ports.
end partition2_process;

process implementation partition2_process.impl
subcomponents
   receiver   : thread commandboard_receiveinput_thread;
   printer    :  thread commandboard_printinfos_thread;
   sem        : data protected_data;
   blackboard : data integer;
connections
   --  example of intra-partition communication with data ports (blackboards)
   blackboardconnection1: data access blackboard -> printer.tempavg;
   blackboardconnection2: data access blackboard -> receiver.tempavg;
   --  example of intra-partition communication with event port (events)
   eventconnectionexample: port receiver.newavg -> printer.newavg;
   c0 : port printer.ordersensor -> queueingout;
   c1 : port samplingin -> receiver.temp;
   c2 : data access sem -> receiver.need_semaphore {ARINC653::Timeout => 20 ms;};
   c3 : data access sem -> printer.need_semaphore {ARINC653::Timeout => 10 ms;};
end partition2_process.impl;

-- Main runtime

processor powerpc
end powerpc;

processor implementation powerpc.impl
subcomponents
   part1:   virtual processor partition1_rt.impl;
   part2:   virtual processor partition2_rt.impl;
properties
   ARINC653::Module_Major_Frame => 50ms;

ARINC653::Module_Schedule =>
    ( [Partition => reference (part1);
      Duration  => 10 ms;
      Periodic_Processing_Start => false;],
      [Partition => reference (part2);
      Duration  => 10 ms;
      Periodic_Processing_Start => false;],
      [Partition => reference (part1);
      Duration  => 30 ms;
      Periodic_Processing_Start => false;]
    );
end powerpc.impl;
```

```
-- Memory
memory partition1_memory
properties
   Base_Address => 0;
   ARINC653::Memory_Type => (Code_Memory);
end partition1_memory;

memory partition2_memory
properties
   Base_Address => 100;
   ARINC653::Memory_Type => (Code_Memory);
end partition2_memory;

memory main_memory
end main_memory;

memory implementation main_memory.impl
subcomponents
   part1mem: memory partition1_memory;
   part2mem: memory partition2_memory;
end main_memory.impl;

system arincsystem
end arincsystem;

system implementation arincsystem.impl
subcomponents
   mem            : memory main_memory.impl;
   cpu            : processor powerpc.impl;
   partition1_pr  : process partition1_process.impl;
   partition2_pr  : process partition2_process.impl;
connections
   samplingconnection: port partition1_pr.samplingout ->
 partition2_pr.samplingin;
   queueingconnection: port partition2_pr.queueingout ->
partition1_pr.queueingin;
properties
   --  bind partition process to their associated
   --  runtime (virtual processor)
   Actual_Processor_Binding =>
(reference (cpu.part1)) applies to partition1_pr;
   Actual_Processor_Binding =>
(reference (cpu.part2)) applies to partition2_pr;

   --  bind partition process to their address spaces
   --  (memory components)
   Actual_Memory_Binding =>
(reference (mem.part1mem)) applies to partition1_pr;
   Actual_Memory_Binding =>
(reference (mem.part2mem)) applies to partition2_pr;
end arincsystem.impl;


end arincexample1;
```

A.20 Example with two modules

(61) The following example illustrates the modeling of a distributed ARINC 653 system with two modules. This example illustrates the modeling of a communication between two ARINC 653 modules with AADL.

a. In the first module, three partitions are defined. One partition communicates (it sends data) ; the second does not communicate. The third contains the device driver for the network interface. Since the third partition contains a device driver it is considered in ARINC 653 to be a system partition.

b. Device drivers are specified using the Device_Driver property in the textual representation. Driver internal are not represented in the graphic version since there is no standardized way to represent properties in the graphic notation of AADL.

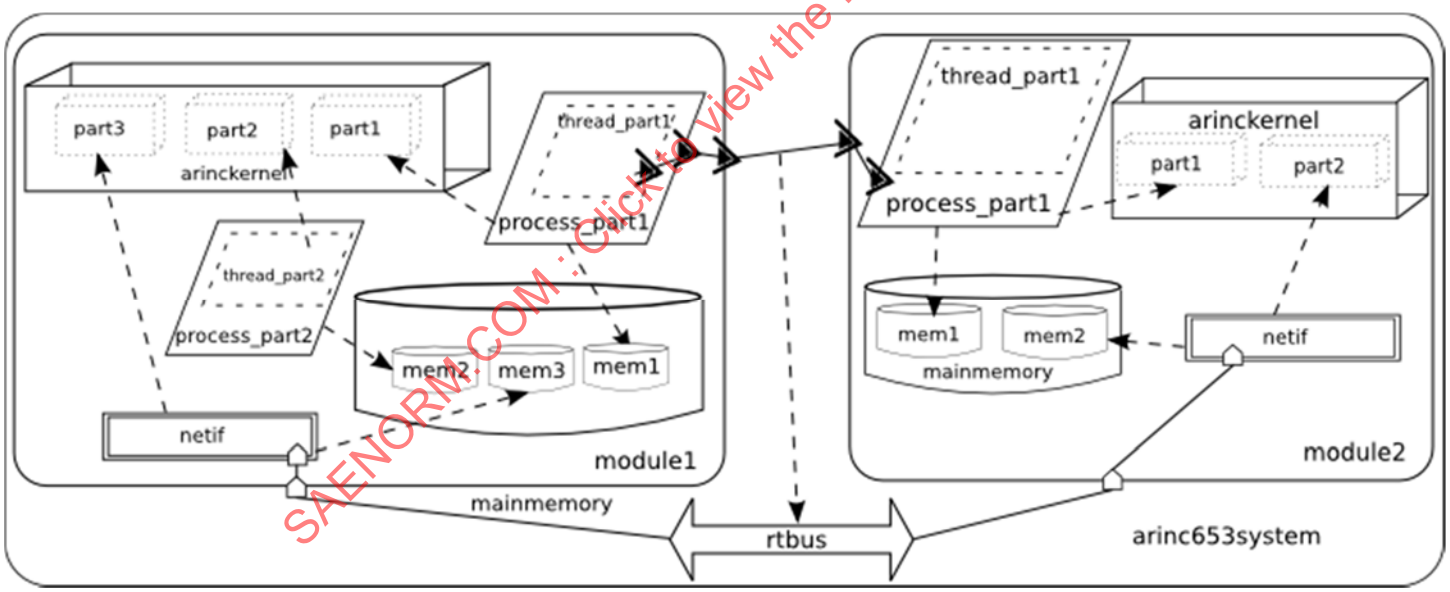
c. Each partition (even system partitions that execute device drivers) is bound to a part of the main memory (modeling of different address spaces).

d. The second module, contains two partitions : one communicates and the second contains the device driver for the network interface.

e. These two ARINC 653 modules communicate through AADL event data ports. It maps the concept of ARINC 653 queueing ports between two ARINC653 modules.

f. Notice that the ARINC 653 modules communicate across a bus named `rtbus`. The association between the bus and its driver is modeled with an `access` connection.

(62) There is the graphical representation of this example



(63) There is the textual representation of this example

```
package arincexample2
public
with ARINC653;

-- First, define generic component
data integer
end integer;

memory memchunk
end memchunk;
```

```
memory mainmemory
end mainmemory;

bus anybus
end anybus;

bus implementation anybus.i
end anybus.i;
thread network_driver_thread
properties
   Stack_Size => 4 Bytes;
   Code_Size => 10 Bytes;
   Period => 200 ms;
   Compute_Execution_Time => 5 ms .. 10 ms;
end network_driver_thread;

thread implementation network_driver_thread.i
end network_driver_thread.i;

abstract network_driver_partition
end network_driver_partition;

abstract implementation network_driver_partition.i
subcomponents
   thr : thread network_driver_thread.i;
end network_driver_partition.i;

device network_interface
features
   thebus: requires bus access anybus.i;
properties
Device_Driver => classifier (arincexample2::network_driver_partition.i);
end network_interface;

device implementation network_interface.i
end network_interface.i;

virtual processor partition_runtime
properties
      Scheduling_Protocol => (RMS);
end partition_runtime;

processor arinckernel
end arinckernel;

-- Then, we define the first module and its subcomponents.

processor implementation arinckernel.module1
subcomponents
   part1 : virtual processor partition_runtime
{ARINC653::DAL => LEVEL_A;};
   part2 : virtual processor partition_runtime
{ARINC653::DAL => LEVEL_B;};
   part3 : virtual processor partition_runtime
{ARINC653::DAL => LEVEL_C;};
properties
   ARINC653::Module_Major_Frame => 40ms;
```

```
ARINC653::Module_Schedule =>
    ( [Partition => reference (part1);
      Duration  => 10 ms;
      Periodic_Processing_Start => false;],
      [Partition => reference (part2);
      Duration  => 10 ms;
      Periodic_Processing_Start => false;],
      [Partition => reference (part3);
      Duration  => 20 ms;
      Periodic_Processing_Start => false;]
    );
end arinckernel.module1;

memory implementation mainmemory.module1
subcomponents
   mem1 : memory memchunk;
   mem2 : memory memchunk;
   mem3 : memory memchunk;
end mainmemory.module1;

thread module1_thread_part1
features
   sensorout : out event data port integer;
end module1_thread_part1;

-- Thread for the first partition

thread module1_thread_part2
end module1_thread_part2;

-- Thread for the second partition

process module1_process_part1
features
   sensorout : out event data port integer{Queue_Size => 1;
                                           ARINC653::Timeout => 5ms;
                                           ARINC653::Queueing_Discipline => FIFO;};
end module1_process_part1;

process implementation module1_process_part1.impl
subcomponents
   mythread : thread module1_thread_part1;
connections
   c0 : port mythread.sensorout -> sensorout;
end module1_process_part1.impl;

-- Process for the first partition

process module1_process_part2
end module1_process_part2;

process implementation module1_process_part2.impl
subcomponents
   mythread : thread module1_thread_part2;
end module1_process_part2.impl;

-- Process for the second partition

system module1_system
```

```
features
   thebus: requires bus access anybus.i;
   sensorout : out event data port integer;
end module1_system;

system implementation module1_system.impl
subcomponents
   netif : device network_interface.i;
   mainmemory : memory mainmemory.module1;
   cpu : processor arinckernel.module1;
   process_part1 : process module1_process_part1;
   process_part2 : process module1_process_part2;
connections
   c0 : port process_part1.sensorout -> sensorout;
   c1 : bus access thebus -> netif.thebus;
properties
   Actual_Processor_Binding   => (reference (cpu.part1)) applies to process_part1;
   Actual_Processor_Binding   => (reference (cpu.part2)) applies to process_part2;
   Actual_Processor_Binding   => (reference (cpu.part2)) applies to netif;
   Actual_Memory_Binding  => (reference (mainmemory.mem1)) applies to process_part1;
   Actual_Memory_Binding  => (reference (mainmemory.mem2)) applies to process_part2;
   Actual_Memory_Binding  => (reference (mainmemory.mem3)) applies to netif;
end module1_system.impl;

-- System that contain everything for the first module

-- Now, we declare the second module

processor implementation arinckernel.module2
subcomponents
   part1 : virtual processor partition_runtime;
properties
   ARINC653::Module_Major_Frame => 40ms;
ARINC653::Module_Schedule =>
    ( [Partition => reference (part1);
      Duration  => 20 ms;
      Periodic_Processing_Start => false;],
      [Partition => reference (part1);
      Duration  => 20 ms;
      Periodic_Processing_Start => false;]
    );
end arinckernel.module2;

thread module2_thread_part1
features
   sensorin : in event data port integer;
end module2_thread_part1;

process module2_process_part1
features
   sensorin : in event data port integer{Queue_Size => 1;
                                         ARINC653::Timeout => 5ms;
                                         ARINC653::Queueing_Discipline => FIFO;};
end module2_process_part1;

process implementation module2_process_part1.impl
subcomponents
   thread_part1 : thread module2_thread_part1;
connections
```

```
   c0 : port sensorin -> thread_part1.sensorin;
end module2_process_part1.impl;

memory implementation mainmemory.module2
subcomponents
   mem1 : memory memchunk;
   mem2 : memory memchunk;
end mainmemory.module2;

system module2_system
features
   thebus   : requires bus access anybus.i;
   sensorin : in event data port integer;
end module2_system;

system implementation module2_system.impl
subcomponents
   mainmemory     : memory mainmemory.module2;
   cpu            : processor arinckernel.module2;
   process_part1  : process module2_process_part1.impl;
   netif          : device network_interface.i;
connections
   c0 : port sensorin -> process_part1.sensorin;
   c1 : bus access thebus -> netif.thebus;
properties
   Actual_Processor_Binding   => (reference (cpu.part1)) applies to process_part1;
   Actual_Processor_Binding   => (reference (cpu.part1)) applies to netif;
   Actual_Memory_Binding  => (reference (mainmemory.mem1)) applies to process_part1;
   Actual_Memory_Binding  => (reference (mainmemory.mem2)) applies to netif;
end module2_system.impl;


-- Now, we declare the main system that contains both modules

system arinc653system
end arinc653system;

system implementation arinc653system.impl
subcomponents
   module1  : system module1_system.impl;
   module2  : system module2_system.impl;
   rtbus    : bus anybus.i;
connections
   conn1 : port module1.sensorout -> module2.sensorin;
   busaccess_module1 : bus access rtbus -> module1.thebus;
   busaccess_module2 : bus access rtbus -> module2.thebus;
properties
   Actual_Connection_Binding => (reference (rtbus)) applies to conn1;
end arinc653system.impl;
end arincexample2;
```

Annex C Code Generation Annex

C.1 Scope

(1) The AADL allows the end user to model Distributed Real-Time and Embedded systems. Such systems would run atop an AADL runtime, eventually built over a complete runtime system (Ada, Java), or a real-time operating system or RTOS (with a C/POSIX API, etc.). Services provided by the AADL runtime can be manipulated by model entities (like subprogram implementation). In this context, guidelines and specifications are required to build the code referenced by such entities.

(2) The purpose of this annex is to enable the mapping of AADL model entities onto programming languages, and the mapping of the AADL runtime services onto these languages.

(3) This document targets the Ada and C languages. Other languages are expected to be regular and to follow similar considerations. We retained Ada 2012 and C11 language specifications, as they are the latest revisions of these two ISO standards and use their corresponding short name, without mention of the actual version being used.

(4) This document only addresses how the user can insert its own code to interact with the AADL runtime and elements from the models. It details how to map AADL identifiers (section C.3**Error! Reference source not found.**), packages (section 0), data types (section C.5), subprograms (section C.6), AADL runtime services (section C.7)

(5) It addresses neither the actual implementation of the AADL runtime nor the use of actual underlying operating system services made by the AADL runtime.

(6) Different methods of implementation can be contemplated for supporting AADL concepts at source-code level. Among the solutions, one can pick: “manual coding” where the designer does AADL-to-code translation manually; “API based”, where the designer manipulates an API representing the AADL runtime; “compilation based”, where part of the code is generated from the AADL model. In this document, we aim at being generic to let implementation decide how to combine code and the AADL runtime, and perform specific optimizations.

(7) The objective of this document is to provide a portable definition of coding guidelines to map AADL concepts and model entities onto a programming language, so that source code can be included and manipulated by different tools.

C.2 Structure of the document

(8) The document is structured as follows: for each AADL concept, a mapping is proposed for both the Ada and C languages.

(9) The definition of the following rules obeys to the following rationale:

a. **Naming/Mapping**: one needs to name the AADL entities of its model in the target programming language. This is presented in section C.3.2 that lists naming conventions for mapping AADL identifiers, and in section 0 that lists how to map an AADL hierarchy of packages.

b. **Data consistency**: one needs to manipulate data types. This document relies on the Data modeling annex to provide support for data types. Section C.5 illustrates how to map AADL data types definition onto the target language.

c. **Functional routines execution**: one needs to execute subprograms. Section C.6 illustrates how AADL subprograms are mapped.

d. **Interaction within distributed environments**: one needs to interact with other entities. Section C.7 shows how to use AADL runtime services.

(10) This Annex document also completes for C and Ada the mappings of AADL runtime services defined in the AADL core document, or data types defined using the “Data Modeling Annex Document; and provides language-specific packages and files.

C.3 Naming conventions

(11) Depending on the mapping requirements, one need to map either component type/implementation onto language constructs, e.g. for mapping AADL data component types onto their corresponding data type definition in source code, or to map component classifier onto corresponding references (e.g. variables, etc). This is discussed in the next sections.

(12) AADL identifiers (such as name of subcomponents, component types or implementations) are converted into language identifiers. A method of implementation should use identifiers from the declarative model whenever possible, as these represent the user view on the model.

(13) If the target language supports a namespace mechanism, it is recommended to use this mechanism to avoid the use of the fully qualified name.

(14) To avoid name clashing with language-specific keywords or to follow language-specific syntactic rules, the following rules are defined:

C.3.1 Ada mapping

(15) To respect the Ada rules of identifier construction (AARM, 2.3 (2)), some characters that may appear in AADL entity names must be replaced. Here is a list:

- “.” must be replaced by underscores “_”.
- Two consecutive underscores “__” must be replaced by “_U_”.

(16) The string “AADL_” must prefix each identifier that clashes with an Ada keyword. If the concatenation provokes a clashing between identifiers, we add another concatenation of the same string until there is no longer name clashing.

(17) AADL string properties referencing to an Ada source code element (like `Compute_Entrypoint_Source_Text`) should be fully qualified Ada name.

C.3.2 C mapping

(18) To respect the C rules for identifiers (C RM 6.4.2.1), all identifiers derived from AADL identifiers are turned to lowercase.

(19) The string “`aadl_`” must prefix each identifier that clashes with a C keyword. If the concatenation provokes a clashing between identifiers, we add another concatenation of the same string until there is no longer name clashing.

(20) AADL string properties referencing to a C source code entity (like `Compute_Entrypoint_Source_Text`) should be the name of the corresponding entity.

(21) C requires special rules for mapping some constructs, these are listed after: enumeration type in section C.5.

(22) An implementation method is allowed to define additional rules in case of symbol conflicting with the ones defined by the underlying operating systems.

(23) Note: Such clash may occur if the implementation of the AADL runtime requires the use of C function from the C standard library (e.g. `read`), and the user defines also a subprogram called “read”.

C.4 Mapping of AADL packages

C.4.1 Ada mapping

(24) AADL packages and Ada packages differ in visibility rules. There are no nested packages in AADL, whereas Ada favor them. To avoid introducing unwanted visibility to parent package, AADL packages are mapped onto Ada packages in a flat hierarchy. AADL hierarchy is mapped onto a single name, where dots are replaced with one consecutive underscore. All identifiers are turned to lower case. For instance, the identifier `Foo::Bar` is mapped onto `foo_bar` and `Foo::Bar::Baz` onto `foo_bar_baz`.

C.4.2 C mapping

(25) AADL packages have no equivalent in C. AADL hierarchy is mapped onto a single name, where dots are replaced with two consecutive underscores. All identifiers are turned to lower case. For instance, the identifier `Foo::Bar` is mapped onto `foo__bar` and `Foo::Bar::Baz` onto `foo__bar__baz`.

C.5 Mapping data types components

(26) The AADL proposes several ways to define data types: either one uses the Data Modeling annex, or relies on AADL properties applied to data component types or implementation. See discussion in the Data Modeling Annex document, part of AS5506A document.

C.5.1 Mapping of Data_Model

(27) The Data Modeling annex of AADLv2 standard defines properties so that the system designer can define its own data. These properties are defined in the AADL property set `Data_Model`.

(28) An implementation method may provide a tool to map AADL data component type definitions onto the destination language.

(29) An implementation method may restrict the set of data types supported by the runtime to reflect actual hardware limitations (e.g. no floating point) or coding restrictions (e.g. no types of unbounded size).

C.5.2 Mapping of Base_Types

(30) The Data Modeling annex defines base types to be manipulated by the end user, in the AADL package `Base_Types`.

(31) The implementation method shall provide an implementation of the AADL package `Base_Types` in the destination language, or mechanism to generate it. The user may directly reference entities from this implementation in its own source code.

(32) An implementation method may restrict the content of `Base_Types` to the types whose properties are supported in the `Data_Model` property set.

(33) An Ada package mapped from `Base_Types` is provided for Ada in Appendix 1.1.

(34) A C header file mapped from `Base_Types` is provided for C in Appendix 1.2.

C.5.3 Mapping of data components

(35) Each data component denoting a basic type (Boolean, Character, Enum, Float, Fixed, Integer, String) is mapped to an equivalent data type in the target language. The name of the type follows rules on identifiers mapping. Its type derives from the properties of the AADL component.

(36) Mapping of the Based_Types package is provided in APPENDIX A.

(37) Complex data components (Array, Struct, Union) are mapped to the corresponding construct in the target language. The name of the type follows rules on identifier mapping. Its type derives from the properties of the AADL component.

a. The following array definition

```
data One_Dimension_Array
 properties
   Data_Model::Data_Representation => Array;
   Data_Model::Base_Type => (classifier (Base_Types::Integer));
   Data_Model::Dimension => (42);
 end One_Dimension_Array;
```

- would be mapped to

```
-- Ada
type One_Dimension_Array
      is array (1 .. 42)of Base_Types.Integer_Type;
/* C */
typedef base_types__integer_types one_dimension_array [42];
```

b. The following struct definition

```
data A_Struct1
properties
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type =>
(classifier (Base_Types::Float),
 classifier (Base_Types::Character));
  Data_Model::Element_Names => ("f1", "c2");
end A_Struct1;
```

- would be mapped to

```
-- Ada
type A_Struct1 is record
  F1 : Base_Types.Float;
  C2 : Base_Types.Character;
end record;
/* C */
typedef struct{
   base_types__float f1;
   base_types__character c2;
} a_struct1;
```

c. The following union definition

```
data A_Union1
 properties
```

```
    Data_Model::Data_Representation => Union;
    Data_Model::Base_Type =>
      (classifier (Base_Types::Float),  classifier (Base_Types::Character));
    Data_Model::Element_Names => ("f1", "f2");
  end A_Union1;
```

would be mapped to

```
  -- Ada
  type A_Union_1_Flag  is (F1_Flag, F2_Flag);

  type A_Union1 (A : A_Union_1_Flag) is record
     case A is
        when F1_Flag =>
           F1 : Float;
        when F2_Flag =>
           F2 : Character;
     end case;
  end record;
/* C */
typedef union {
   base_types__float f1;
   base_types__character f2;
} a_union1;
```

(38) C enum requires a special rule for mapping. C does not allow multiple use of the same enumerator. In this case, the name of the enumerator is prefixed by the name of the AADL data components type.

a. The following definition

```
data An_Enum
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("foo", "bar");
end An_Enum;
```

would be mapped to

```
-- Ada
  type An_Enum is (Foo, Bar);

/* C */
typedef enum { an_enum_foo, an_enum_bar } an_enum;
```

(39) Complex data components that reference other data components are mapped onto data structures of the target language: Ada record types (resp. C structure definitions). Each field defining identifier is mapped from the subcomponent name given in the data component implementation with the rules on identifiers mapping. The type of the field is the Ada (resp. C) type mapped from the data corresponding component. The name of the type follows rules on identifiers mapping. Its type derives from the properties of the AADL component.

C.5.4 Other cases

(40) The system designer may decide not to use the Data Modeling annex. In this case, he may use the `Source_Language`, `Type_Source_Name` and `Source_Text` properties to specify the definition of its data type in the target language. In this case, the user may reference directly this type, without further refinements.

a. The following definition

```
data C_Type
properties
  Source_Language => (C);
  Source_Text => ("types.h");
  Type_Source_Name => "the_type";
end C_Type;
```

it is expected the user provides a C source file named “types.h” where a type “the_type” is defined.

C.6 Mapping of AADL subprograms

(41) AADL subprograms denote library elements that may later be used by threads. In this section, we review mapping rules for subprograms without dependencies to the AADL runtime.

(42) We illustrate how the Ada mapping may incorporate C or Ada subprograms, and how the C mapping may incorporate C subprograms.

(43) The actual implementation of the mapped subprograms depends on the nature of the corresponding AADL subprogram component. Subprogram components can be classified in many ways depending on the value of the `Source_Language`, `Source_Name` and `Source_Text` standard AADL properties, the existence or absence of call sequences in the subprogram implementation. There are three kinds of subprogram components: empty subprograms, opaque subprograms and pure call sequence subprograms.

C.6.1 Ada mapping

(44) In this section, we discuss the mapping of AADL subprograms onto an Ada AADL runtime. AADL subprograms can be either Ada code, but also C or other formalisms.

(45) AADL subprograms are mapped onto Ada procedures. In case of data-owned subprograms, they are managed in the related generated package. The parameters of the procedure are mapped from the subprogram features with respect to the following rules:

- The name of the subprogram is derived from the name of the subprogram classifier
- The parameter name is mapped from the parameter feature name
- The parameter type is mapped from the parameter feature data type
- The parameter orientation is the same as the feature orientation (“in”, “out” or “in out”).
- The order of parameters is the same as the corresponding AADL subprogram component.

(46) **Empty subprograms** correspond to AADL subprogram types or implementations for which there is neither Source_Language nor Source_Name nor Source_Text values nor call sequences. Such kind of subprogram components is an empty placeholder for future implementation. It should evolve to one of the two other kinds in the production AADL model.

a. The AADL snippet below is an example of an empty subprogram:

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
```

b. An Ada mapping for this subprogram could be:

```
procedure sp (e : in message; s : out message) is
pragma Unreferenced (e…);
begin
   null;
end sp;
```

(47) **Opaque subprograms** are AADL subprograms for which the Source_Language property indicates the programming language of the implementation (C or Ada). The Source_Name property indicates the name of the target language subprogram implementing the AADL subprogram:

a. For Ada subprograms, the value of the Source_Name property is the fully qualified name of the subprogram (e.g. My_Package.My_Spg). If the package is stored in a file named according to the Ada compiler conventions, there is no need to give a Source_Text property for Ada subprograms. Otherwise the Source_Text property is necessary for the compiler to fetch the implementation files.

b. For C subprograms, the value of the Source_Name property is the name of the C subprogram implementing the AADL subprogram. The Source_Text is mandatory for this kind of subprogram and it must give one of the following information: the path (relative or absolute) to the C source file that contains the implementation of the subprogram; the path to one or more precompiled object file(s) that implement(s) the AADL subprogram; the path to one or more precompiled C libraries that implement(s) the AADL subprogram.

c. These properties can be used together, for example one may give the C source file that implements the AADL subprogram, an object file that contains entities used by the C file and a library that is necessary to the C sources or the objects. In this case, the code generation consists of creating a shell for the implementation code. In the case of Ada subprograms, the generated subprogram renames the implementation subprogram (using the Ada renaming facility).

d. Here is a mapping example

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
subprogram implementation sp.impl
properties
   Source_Language => Ada;
   Source_Name => “Repository.Sp_Impl”;
```

```
end sp.impl;
```

e. The generated code for the sp.impl component is:

```
with Repository;
-- ...
procedure sp_impl (e : in message; s : out message)
  renames Repository.Sp_Impl;
```

f. The code of the `Repository.Sp_Impl` procedure is provided by the designer and must conform to the `sp.impl` signature as defined in the architecture. The coherence between the two subprograms will be verified by the Ada compiler. The fact that the hand-written code is not inserted in the generated shell allows this code to be written in a programming language other than Ada. Thus, if the implementation code is C we have this situation:

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
subprogram implementation sp.impl
properties
   Source_Language => C;
   Source_Name => “implem”;
   Source_Text => “code.c”;
end sp.impl;
```

g. The `Source_Name` value is interpreted as the name of the C subprogram implementing the AADL subprogram.

h. Path information to actual source file can be added in `Source_Text`, or left under the control of the implementation method.

i. The generated code for the sp.impl component is:

```
procedure sp_impl (e : in message; s : out message);
pragma Import (C, sp_impl, “implem”);
```

j. This approach will allow us to have a certain flexibility by separating the generated signature from the hand-written code. We can modify the AADL description without affecting the hand-written code (the signature should not be modified of course).

(48) **Pure call sequence subprograms**: in addition to the opaque approach, which consists of delegating all the subprogram body writing to the user, AADL allows to model subprogram as a pure call sequence to other subprograms. Example:

```
subprogram spA
features
   s : out parameter message;
end spA;
subprogram spB
features
   s : out parameter message;
end spB;
subprogram spC
```

```
features
   e : in parameter message;
   s : out parameter message;
end spC;
subprogram spA.impl
calls {
   call1 : subprogram spB;
   call2 : subprogram spC;};
connections
   cnx1 : parameter call1.s -> call2.e;
   cnx2 : parameter call2.s -> s;
end spA.impl;
```

a. In this case, the subprogram connects together a number of other subprograms. In addition to the call sequence, the connections clause completes the description by specifying the connections between parameters. The pure call sequence model allows the generation of skeleton code referring to other subprograms: the calls in the call sequence correspond to Ada procedure calls and the connections between parameters correspond to the possible intermediary variables. The Ada code generated for the subprogram `spA.impl` is:

```
procedure spA_impl (s : out message) is
  cnx1 : message;
begin
  spB (cnx1);
  spC (cnx1, s);
end spA_impl;
```

b. Note that in case of pure call sequence subprograms, the AADL subprogram must contain only one call sequence. If there is more than one call sequence, it's impossible - in this case - to determine the relation between them.

C.6.2 C mapping

(49) In this section, we discuss the mapping of AADL subprograms onto a C AADL runtime. We review some options available.

(50) AADL subprograms are mapped onto C functions. In case of data-owned subprograms, they are managed in the related generated package. The parameters of the procedure are mapped from the subprogram features using the following rules:

- The parameter name is mapped from the parameter feature name previously generated using the data mapping rules in section C.5.
- The parameter type is mapped from the parameter feature data type
- The parameter orientation of the C subprogram follows the feature orientation: by copy for "in" parameters, by reference for "out" or "in out".
- The order of parameters is the same as the corresponding AADL subprogram component.

(51) **Empty subprograms** correspond to AADL subprograms for which there is neither `Source_Language` nor `Source_Name` nor `Source_Text` values nor call sequences. Such kind of subprogram components is empty placeholders for future implementation.

a. The AADL snippet below is an example of an empty subprogram:

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
```

b. An C mapping for this subprogram could be:

```
void sp (message e, message* s) {
   /* TO BE IMPLEMENTED */
}
```

(52) **Opaque subprograms** are AADL subprograms for which the Source_Language property indicates the programming language of the implementation (C or Ada). The Source_Name property indicates the name of the subprogram implementing the subprogram:

a. For C subprograms, the value of the `Source_Name` property is the name of the C subprogram implementing the AADL subprogram. The `Source_Text` is mandatory for this kind of subprogram and it must give one of the following information: the path (relative or absolute) to the C source file that contains the implementation of the subprogram; or, the path to one or more precompiled object files that implement the AADL subprogram; or, the path to one or more precompiled C library that implement the AADL subprogram.

b. These information can be used together, for example one may give the C source file that implements the AADL subprogram, an object file that contains entities used by the C file and a library that is necessary to the C sources or the objects. In this case, the code generation consists of creating a shell for the implementation code. Here is a mapping example

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
subprogram implementation sp.impl
properties
   Source_Language => C;
   Source_Name => “C_sp_impl”;
   Source_Text => “sp.c”;
end sp.impl;
```

c. The generated code for the sp.impl component is:

```
void sp_impl (message e, message* s) {
   C_sp_impl (e, s); /* Call user code */
}
```

(53) **Pure call sequence subprograms:** In addition to the opaque approach, which consist of delegating all the subprogram body writing to the user, AADL allows to model subprogram as a pure call sequence to other subprograms. Example:

```
subprogram spA
features
   s : out parameter message;
```

```
end spA;
subprogram spB
features
   s : out parameter message;
end spB;
subprogram spC
features
   e : in parameter message;
   s : out parameter message;
end spC;
subprogram spA.impl
calls {
   call1 : subprogram spB;
   call2 : subprogram spC;};
connections
   cnx1 : parameter call1.s -> call2.e;
   cnx2 : parameter call2.s -> s;
end spA.impl;
```

a. In this case, the subprogram connects together a number of other subprograms. In addition to the call sequence, the connections clause completes the description by specifying the connections between parameters. The pure sequence call model allows to generate complete code: the calls in the call sequence corresponds to C procedure calls and the connections between parameters correspond to the possible intermediary variables. The C code generated for the subprogram `spA.impl` is:

```
void spA_impl (message* s) {
   message cnx1;
   spB (&cnx1);
   spC (cnx1, &s);
}
```

b. Note that in case of pure call sequence subprograms, the AADL subprogram must contain only one call sequence. If there is more than one call sequence, it's impossible - in this case - to determine the relation between them.

C.6.3 Management of port variables

(54) Port variables represent a way to access the port of a thread or a subprogram. Those are the only place where user's code can interact with ports.

(55) To take into account multiple instances, the AADL runtime introduces a specific opaque type called an AADL context.

(56) One specific AADL context type is generated for each thread or subprogram component type or implementation. Its name is defined as follows

a. For Ada: `<component_identifier>_Context`

b. For C: `__<component_identifier>_context`

(57) This opaque type is a record whose members are the ports available in this particular context, but also access to data. The name of these members follows identifiers mapping rules defined in this annex.

(58) The `Code_Generation::Convention` property controls the generation of specific structures to manage port variables.

a. `Legacy` convention does not generate this information. It is the default value

b. `AADL` convention generates port contextual structure

(59) A method of implementation may infer the actual value of the `Convention` property based on the model information.

(60) The `Code_Generation` property set is present in APPENDIX C

C.7 Using AADL services inside subprograms

C.7.1 Using AADL ports to communicate

(61) AADL ports allow subprograms to send or receive events or data to other entities in the model. Ports are accessed through runtime services as defined in section A.9 of the standard AADLv2 document. These services are defined as AADL subprograms, but some implementation details exist, mostly in the way ports are defined.

(62) The following provides a solution to these implementation details in a way that is portable and consistent with the concept of port variables.

(63) AADL defines mechanisms to let subprograms interact with the environment through ports and access to data components. This section illustrates how to use these mechanisms

(64) One instance of the context type is passed as parameters to each user's subprograms that need to interact with ports. This parameter is used as parameter specifying the port variable to use.

a. Here is an example, using AADLv2

```
thread Operator_T
features
  Gear_Cmd: out event port;
properties
  Dispatch_Protocol  => Periodic;
  Period             => 10 Sec;
  Compute_Entrypoint_Source_Text => "On_Operator";
  source_text        => ("flight-mgmt.c");
end Operator_T;
```

and the corresponding C implementation

```
void on_operator (__operator_t_context *self) {
/* … */
__aadl_send_output (self->gear_cmd, &request);
}
```

(65) Note: in this example, the AADL convention is implicitly enforced as the subprogram implicitly requires access to port variable, and there is no way to deduce subprogram output parameters.

(66) An AADL model may connect `in` or `out` subprogram parameters to thread ports. In this case, there is no need to use explicitly AADL runtime services. A method implementation is allowed to generate directly the corresponding code from the AADL model if the subprogram follows the Legacy convention.

a. Let us consider the following AADL model

```
subprogram Do_Ping_Spg
features
  Data_Source : out parameter Simple_Type;
properties
  Code_Generation::Convention => Legacy;
end Do_Ping_Spg;

thread P
features
  Data_Source : out event data port Simple_Type;
end P;

thread implementation P.Impl
calls
Mycalls: {
  P_Spg : subprogram Do_Ping_Spg;
};
connections
  parameter P_Spg.Data_Source -> Data_Source;
 -- Out parameter of P_Spg is passed to the port
-- variable automatically
end P.Impl;
```

then, it is sufficient to implement Do_Ping_Spg this way

```
procedure Do_Ping_Spg (Data_Source : out Simple_Type);
```

and have the method of implementation generates the corresponding glue code to propagate the out parameter to the corresponding port variable.

Note: This allows the seamless integration of existing code in an AADL model, for instance for opaque subprograms.

C.7.2 Subprograms attached as entrypoints

(67) In some cases, the user may want to integrate source code as entrypoint attached to threads in event (data) ports using the `Compute_Entrypoint` standard property and its several derivations.

(68) The first parameter is the AADL context information, discussed in section C.7

(69) In the case of an `event port`, the signature of the corresponding subprogram adds no further parameter.

(70) In the case of an `event data port`, the signature has an additional unique parameter derived from the type of the port.

a. Here is an example, using AADLv2

```
thread HCI_T
features
  Stall_Warning  : in  event data port Ravenscar.Integer
    {Compute_Entrypoint_Source_Text => "Manager.On_Stall_Warning";};
  Engine_Failure : in  event      port
    {Compute_Entrypoint_Source_Text => "Manager.On_Engine_Failure";};
end HCI_T;
```

b. And the corresponding Ada and C declarations

```
-- Ada
package Manager is
   procedure On_Stall_Warning
     (Ctx: HCI_T_Context; Stall_Warning : Ravenscar_Integer);
   procedure On_Engine_Failure (Ctx : AADL_Context);

/* C */
void on_stall_warning
      (__hci_t_context ctx, ravenscar_integer stall_warning);
void on engine_failure (__hci_t_context ctx);
```

C.7.3 Accessing data components

(71) Data components can support subprogram access as a way to model accessors to its internal data. Corresponding user subprograms have additional parameters that represent the internal data to interact on. See the following example:

```
subprogram Update
features
   this : requires data access POS.Impl;
end Update;
subprogram implementation Update.Impl
properties
   source_language => Ada95;
   source_name     => “Toy.Update”;
end Update.Impl;

data Position_Internal_Type
properties
   Data_Model::Data_Representation => Integer;
end Position_Internal_Type;
```

```
data Position
features
   Update : provides subprogram access Update.Impl;
properties
   Concurrency_Control_Protocol => <..>;
end Position;

data implementation Position.Impl
subcomponents
   Field : data Position_Internal_Type;
properties
   Data_Model::Data_Representation => Struct;
end Position.Impl;
```

(72) This is mapped as the following in Ada

```
procedure Read (Field : in out POS_Internal_Type);
```

(73) The AADL runtime shall handle calls to this subprogram using the concurrency protocol mandated by the user. There is no need for explicit call to Get_Ressources and Release_Resources runtime services.

Note: an implementation method is allowed to support only a subset of allowed mechanisms, depending on the support from the target operating system.

C.7.4 Managing modes

(74) Mode changes are triggered through specific ports. A user subprogram may trigger one by sending an event to the corresponding port.

(75) A user subprogram may use the AADL runtime service `Current_System_Mode` to determine is current mode of operation.

APPENDIX A - MAPPING OF BASE_TYPES

(76) The AADL Base_Types package proposes a list of general utility data types.

Appendix 1.1.Ada mapping

(77) This package can be mapped onto Ada using the types provided by either the Standard package (types without size) or Interfaces as defined by the Ada 2012 reference manual.

(78) In order to support previous versions of Ada, an implementation method is allowed to select a different mapping that preserves the semantics of types.

```
with Interfaces;
   package Base_Types is
      type AADL_Boolean is new Standard.Boolean;
      type AADL_Integer is new Standard.Integer;
      type Integer_8 is new Interfaces.Integer_8;
      type Integer_16 is new Interfaces.Integer_16;
      type Integer_32 is new Interfaces.Integer_32;
      type Integer_64 is new Interfaces.Integer_64;
      type Unsigned_8 is new Interfaces.Unsigned_8;
      type Unsigned_16 is new Interfaces.Unsigned_16;
      type Unsigned_32 is new Interfaces.Unsigned_32;
      type Unsigned_64 is new Interfaces.Unsigned_64;
      type AADL_Natural is new Standard.Integer;
      type AADL_Float is new Standard.Float;
      type Float_32 is new Interfaces.IEEE_Float_32;
      type Float_64 is new Interfaces.IEEE_Float_64;
      type AADL_Character is new Standard.Character;
   end Base_Types;
```

Appendix 1.2.C mapping

(79) The AADL Base_Types package proposes a list of general utility data types. This package can be mapped onto C11 using the types provided by either stdint.h or stdbool.h standard header files.

(80) In order to support more ancient C compilers, an implementation method is allowed to select a different mapping that preserves the semantics of types, or one that is a superset of the intended type.

(81) For instance, Base_Types::Natural cannot be represented in C, as there is no native positive only numeric types.

```
/* C mapping of AADL package base_types */
#ifndef __BASE_TYPES_H__
#define __BASE_TYPES_H__

#include<stdint.h>
#include<stdbool.h>

typedef bool     base_types_boolean;

typedef int8_t   base_types_int8;
typedef int16_t  base_types_int16;
typedef int32_t  base_types_int32;
typedef int64_t  base_types_int64;

typedef uint8_t  base_types_uint8;
typedef uint16_t base_types_uint16;
typedef uint32_t base_types_uint32;
typedef uint64_t base_types_uint64;

typedef float    base_types_float32; /* As per IEEE 754 */
```

```
typedef double   base_types_float64;
typedef char     base_types_character;

#endif  /* __BASE_TYPES_H__ */
```

APPENDIX B - AADL RUNTIME SERVICES

(82) The AADL core standard defines two sets of runtime services in section C.9. Application runtimes services declare service subprograms that can be called by the application source text directly. The second set of services is concerned with services internal to the AADL runtime, and is not the subject of this annex.

(83) We recall that the following API is available to the user:

a. `Send_Output:` explicitly cause events, event data, or data to be transmitted through outgoing ports to receiver ports.

b. `Put_Value:` allows the source text of a thread to supply a data value to a port variable.

c. `Receive_Input:` explicitly requests port input on s incoming ports to be frozen and made accessible through the port variables.

d. `Get_Value:` allows access to the current value of a port variable.

e. `Get_Count:` determine whether a new data value is available on a port variable, and in case of queued event and event data ports, how many elements are available to the thread in the queue.

f. `Next_Value:` provides access to the next queued element of a port variable as the current value. A `NoValue` exception is raised if no more values are available.

g. `Updated` allows the source text of a thread to determine whether input has been transmitted to a port since the last `Receive_Input` service call.

(84) To favor optimization, code safety and reliability, the proposed mapping does not enforce any convention on the organization of the code: AADL API function may be placed in any relevant package. This allows for multiple implementation of the AADL API, with one dedicated instance per AADL model entities. The only recommended practice is to preserve the name of the functions and of its parameters.

APPENDIX C - CODE_GENERATION PROPERTY SET

(85) The following property set is defined to control various aspects of the code generation:

```
property set Code_Generation_Properties is
      Convention: enumeration (AADL, Legacy) => Legacy
             applies to (subprogram);
      --  Under the Legacy convention, no context information
      --  for port variables is generated.
      --  The AADL convention generates such information
      Parameter_Usage: enumeration (By_Value, By_Reference)
     applies to (data access, parameter);
      Return_Parameter: aadlboolean => false applies to (parameter);
  -- if true, out parameter is actually a return parameter
end Code_Generation_Properties;
```

Annex E Error Model Annex

Normative

E.1 Scope

(1) The objective of the Error Model language is to support qualitative and quantitative assessments of system reliability, availability, safety, security, survivability, robustness, and resilience, as well as assure compliance of the system design and implementation to the fault mitigation strategies specified in the annotated architecture model of the embedded software, computer platform, and physical system.

(2) This document defines a language for architecture error 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 subclauses*. 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 that consists of the embedded software runtime architecture, the distributed computer platform, and the physical system with which the embedded system interfaces. In that context the Error Model language defines a set of contracts 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 error, fault, failure, hazard, etc. drawing on the definitions in [ISO/IEC/IEEE 24765:2010] (see Section 2 for details). 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 error modeling at three levels of abstraction with the following concepts:

- Error propagation and flow: Focus on components as error sources, sinks, and paths and the impact of error propagation on other components or the operational environment. Connections can also be error sources, reflecting exceptional conditions in component interactions. Propagation paths are determined by AADL core model connections and bindings. Architecture error model specifications at this level of abstraction correspond to the Fault Propagation and Transformation Calculus (FPTC) [Paige 2009] and allows for technical risk assessment in the form of hazard identification, fault impact analysis, and stochastic fault analysis.
- Component error behavior: Focus on a system or subsystem 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 error 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 subsystems 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 error events, states, 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 language 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 *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 12* 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.

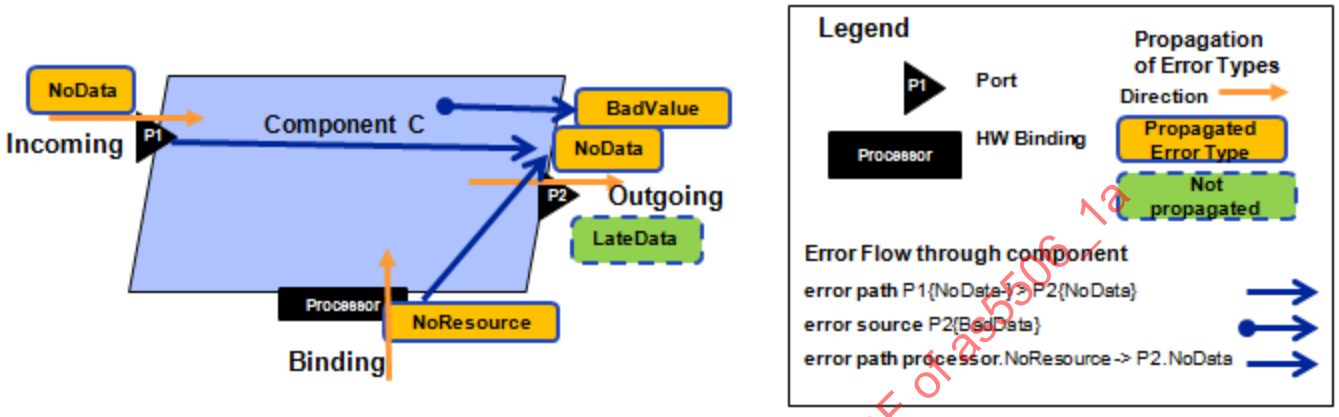


Figure 12 - Error propagations and error propagation flows

(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 13* 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.

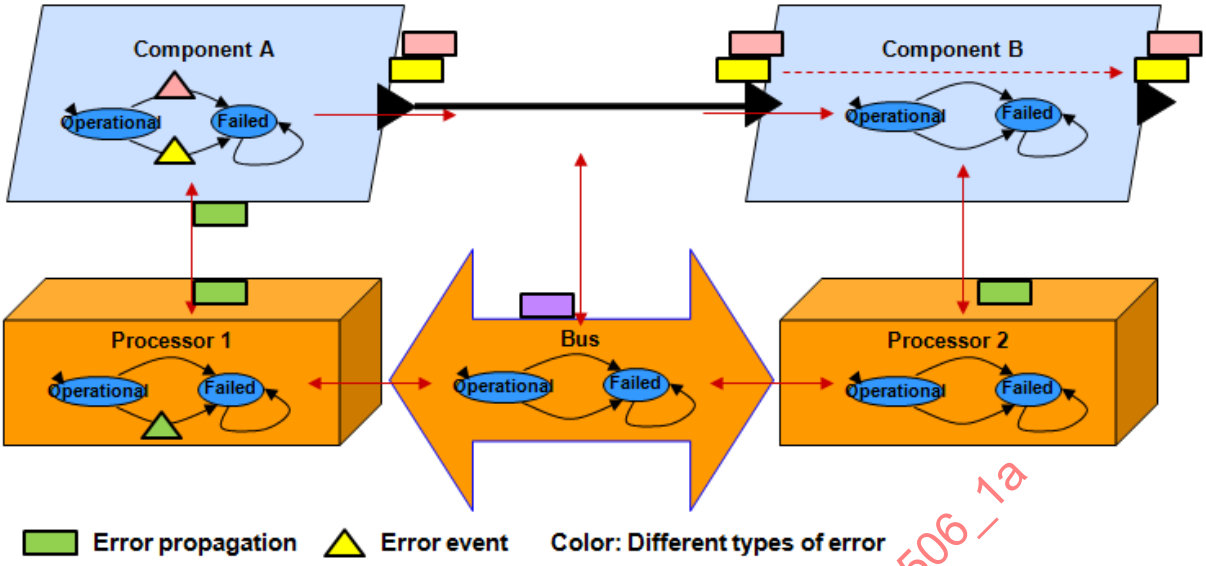


Figure 13 - Error propagation paths defined by architecture

(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 document can be used to specify the risk mitigation methods employed in embedded computer system architectures to increase reliability, availability, safety, security, survivability, robustness, and resilience. 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 language can be used to annotate the AADL model of an embedded system to support a number of the safety assessments and 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. These include Functional Hazard Assessments (FHA), Failure Mode and Effect Analysis (FMEA), Fault Tree Analysis (FTA), and Common Cause Analysis (CCA). FMEA is FMEA is an inductive reasoning (forward logic) technique on the propagation graph of a system from a single cause identifying all effects, while fault trees reflect deductive reasoning (backward logic) taking into account all possible contributors to an incident/accident.

(15) When annotated with occurrence probabilities architecture error models lead to stochastic reliability and availability analysis. In the latter case, not only error events, but also recover and repair events are taken into account. These analyses can be applied to the functional architecture, the conceptual system architecture, and to the result of mapping a functional architecture to a conceptual system architecture.

(16) In case of software, errors are in the requirement specification and design and we cannot assume zero defect. In other words, we have to assume a occurrence probability of one and address such errors with techniques such as analytical redundancy rather than replication. EMV2 allows modelers to characterize error sources and events as inherently design errors versus physical component errors with realistic occurrence probabilities.

(17) Security deals with confidentiality, integrity and availability of information and capabilities. While the focus of safety is to prevent losses – primarily life and property - due to unintentional actions by benevolent actors, the focus of security is to prevent losses – primarily information and resulting property damage - due to intentional actions by malevolent actors. The primary difference is in the intent and in the type of resulting damage. Security terms include vulnerability (security hazard), threat (error source and event), intrusion (instance of an error event resulting in an error state transition), and incident (effect of an intrusion on the error state of another component or the system). Security violations can affect safety and vice versa. The architecture error modeling capability of EMV2 allows both safety and security concerns to be analyzed and mitigated within the same framework.

(18) The architecture error modeling capability of AADL and EMV2 supports incremental development and compositional verification of systems. EMV2 annotations are attached to component classifier declarations that apply to all instances of these components. System components can have both an abstracted error behavior specification and a composite error behavior specification mapping abstract error behaviors of subcomponents and their interactions into the abstract specification. This contract-based compositional approach allows for scalable incremental analysis and verification of reliability, availability, safety, survivability, robustness, security, and resilience concerns – taking advantage of architecture abstraction.

(19) The rules defined in this annex assure that the EMV2 annotations in an architecture specification are consistent and complete. Representations for analysis, e.g., fault trees, Markov Chain, or Petri net models, are auto-generated from this architecture error model. Modifications to architecture specifications are propagated into these analytical models by automatically regenerating them.

(20) The Error Model language definition is organized into the following sections. Section E.2 introduces concepts and terminology used in this annex document. Section E.3 describes reusable Error Model libraries. Section E.4 and component-specific Error Model subclauses. Section E.5 introduces constructs to define error types, error type sets, and hierarchies of error types. Section E.6 describes a set of predeclared error types. Section E.7 introduces constructs in support of error propagation specification. Section E.8 introduces constructs to define reusable error behavior state machines with error and repair events, states, and transitions. Section 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 E.11 describes constructs to support specification of composite error behavior of components based on the error states of a set of subcomponents. Section E.12discusses how error propagation affecting connections is represented. Section E.13 introduces constructs that allow you to define reusable error type mappings and transformations. Section E.14 describes predeclared properties that apply to error model elements. Section E.15 discusses the interaction between the error behavior specification expressed by Error Model language constructs and the health monitoring and fault management capabilities in the actual system architecture. Section E.16 presents an example architecture model of a triple redundant system annotated with Error Model language specifications.

E.2 Concepts and Terminology

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

(2) *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.

- Error encompasses mistakes by humans resulting in incorrect design or code, defects in a process that can lead to incorrect design or operational system, the effect of incorrect system behavior, and a characterization of incorrect behavior as an indication of a failure. In other words, error is the most general and comprehensive term for dealing with architecture error modeling.
- In EMV2 we consistently use the term *error* as keyword for all EMV2 constructs. We do this 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.

(3) *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.

- Fault is a root (phenomenological) cause of an error that can potentially result in a failure, i.e., an anomalous undesired change in the structure or data within a component. A fault may cause that component to eventually not perform according to its nominal specification and result in malfunction or loss of function, i.e., result in a failure.
- In EMV2 we represent different types of faults as error types. In the error propagation abstraction, the presence of the fault in a component is expressed as an error source with the appropriate error type as the origin. In a component error behavior abstraction, a fault is expressed as an error event with an error type. An instance of an error event represents the activation of a fault, i.e., a failure.

(4) *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.

- Failure is a deviation in behavior from a nominal specification resulting in malfunction and loss of function, i.e., a component no longer functions as intended. This may be due to an activated fault within the component, due to error propagation from another component, or due to exceptional conditions when interacting with other components. The deviation can be characterized by type of failure, persistence, and degree of severity. The degree to which a failure affects nominal behavior is referred to as *severity* of the failure.
- In EMV2 we represent failures as occurrence instances of error sources and instances of error events. Error event instances cause transition to an error state, which represents the component failure mode. An error source identifies an outgoing error propagation including error type, reflecting that the failure mode of a component (error state) can affect components it interacts with. The propagation paths are determined by the AADL core model.

(5) *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 specifically refers to unsafe conditions. In this document we will use the term hazard to refer to any exceptional system state or exceptional condition on interacting system components or elements of the operational environment that potentially result in harm. We use the terms safety hazards and security hazard if such a distinction is beneficial. One or more hazards may be contributors to a catastrophic incident or accident (see [Leveson 2012]). A hazard may have one or more contributors, i.e., it is affected by one or more components in the system or environment.
- In EMV2 we represent hazards by a multi-valued property that can be associated with the error source, error state, and error propagation, i.e., to any element in the error propagation graph of a system.

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

(7) 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.

(8) 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.

(9) 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.

(10) 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.

(11) 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.

(12) 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.)

(13) 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).

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

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

(16) 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.

(17) 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.

(21) *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.

(22) *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.

(23) *Vulnerability* is defined as weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.

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 }*
    { error_type_mappings }*
    { error_type_transformations }*

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 (see `error_model_library_reference`). Its package name does not have to be listed in the **with** clause of the package containing the reference.

Semantics

(18) 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 E.10.3.

(19) 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.

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 language, 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 error_type_mappings_reference ; ]
    [ use mappings error_type_mappings_reference ; ]
    [ use behavior error_behavior_state_machine_reference ; ]
    [ error_propagations ]
    [ component_error_behavior ]
    [ composite_error_behavior ]
    [ connection_error_behavior ]
    [ propagation_paths ]
    [ 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 }* ;

-- adapted from AS5506B 11.3
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

(N3) 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).

(N4) The mode identifiers in the `in modes` statement of an Error Model language subclause must refer to modes in the component type or component implementation for which the annex subclause is declared.

(N5) The **use types** clause makes the defining identifiers of error types, error type sets, and aliases from the listed Error Type libraries accessible to an Error Model subclause, i.e., allowing them to be referenced without qualification. If identifiers from different Error Type libraries listed in a **use types** clause are identical, their reference must be qualified. Note that the qualifying Error Type library must be listed in the **use types** clause.

(N6) 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.

(N7) 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.

(N8) 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.

(N9) 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.

(N10) 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.

(N11) 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.

(N12) 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.

(N13) 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.

(N14) 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.

(N15) 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.

(N16) 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 mappings reference.
- **Use mappings** clause: replace the type mappings 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) If a component type or implementation has mode-specific Error Model subclauses, then all subclauses must reference the same error behavior state machine.

Semantics

(20) An Error Model subclause allows component types and component implementations to be annotated with error events to represent activation of component faults, recover and repair events to represent initiation and completion of a recover or repair activity, incoming and outgoing error propagations through component features as well as through bindings to platform components, error behavior expressed as a state machine whose transitions are triggered by error events, recover events, repair events, and incoming error propagations, as well as 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.

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

(22) 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.

(23) 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.

(24) 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.

(25) 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.

(26) 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.

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

(28) 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.

(29) 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.

(30) 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.

(31) 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.

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

- 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 instance hierarchy determines the property value.
- 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.

(33) 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:

- 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 instance hierarchy determines the property value.
- 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.

(34) 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.

(35) 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.

(36) 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.

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

(1) In this section we introduce the concepts of *error type*, *error type* product, *error type set*, and *error type hierarchy*. 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 two or more error types that can occur simultaneously. For example, the combination of `OutOfRange` and `LateDelivery` expressed as error type product `OutOfRange * LateDelivery`. The elements of the type product can be in any order.

(4) An *error type set* is defined as a set of unique elements, i.e., error types and error type products. For example, an error type set may be defined as consisting of the elements *OutOfRange* 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 represents the union of the two type sets.

(6) Error types can be organized into a type hierarchy of subtypes by declaring the error type as subtype of another error type. No error type may be its own subtype. 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 language standard (see Section E.6). They can be extended with user defined error types or renamed with aliases.

Syntax

```
error_type_library ::=
  error types
  [ use types error_type_library_list ; ]
  [ 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_set_definition ::=
  defining_error_type_set_identifier : type set error_type_set ;

error_type_set_alias ::=
  defining_error_type_set_alias_identifier renames type set error_type_set_reference ;

error_type_set ::=
   { type_set_element { , type_set_element }* }

type_set_element ::=
  error_type_or_set_reference | error_type_product

error_type_product ::=
  error_type_reference ( * error_type_reference )+

error_type_set_or_noerror ::=
   error_type_set | { 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_reference | error_type_product }
```

Naming Rules

(N17) 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.

(N18) 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.

(N19) 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.

(N20) 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.

(N21) 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.

(N22) 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.

(N23) 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.

(N24) The **use types** clause makes the defining identifiers of error types, error type sets, and aliases from the listed Error Type libraries accessible to an Error Model subclause, i.e., allowing them to be referenced without qualification. If identifiers from different Error Type libraries listed in a **use types** clause are identical, their reference must be qualified. Note that the qualifying Error Type library must be listed in the **use types** clause.

(N25) Any Error Type library of the **extends** list must not be listed in the **use types** clause.

(N26) The optional qualifying Error Model library reference of an error type or type set reference must adhere to Naming Rule (N2) in Section E.3, i.e., its package name does not have to be listed in the **with** clause of the package containing the reference.

Legality Rules

(L5) 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.

(L6) An error type cannot be the subtype of more than one other error type.

(L7) For two error types in an error type set, one must not be a subtype of the other.

(L8) An error type set cannot contain two elements representing the same error type or an error type and its alias.

(L9) An error type set cannot contain two elements representing the same error type set or an error type set and its alias.

(L10) If two elements of an error 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 type.

(L11) If two elements of an error type set are error type products with the same number of element types and whose element types are from the same type hierarchies, then the element type of one must not be a subtype of the other.

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. Two error types that are part of the same type hierarchy cannot both occur in an error type product. For example, an error propagation may propagate an error of type `IncorrectValue` or of type `OutOfRange`, but not simultaneously since both error types are part of the `ValueError` type hierarchy.

(18) An *error type product* represents combinations of error types that can occur simultaneously. For example, the combination of `ValueError` and `LateDelivery` can be expressed as error type product `ValueError * LateDelivery`. The elements of the error type product can be in any order, 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 of each element type. In our example, the product includes late delivery of out of range, out of bounds, and incorrect value errors. Note that different elements of an error type product cannot be from the same type hierarchy.

(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 it 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 specify that error propagations propagates errors that are a combination of value and timing errors `{ValueError * TimingError}` (see Section 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 with 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 (see Section E.14.1).

(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.

(26) An Error Type library can introduce error types, error type sets, and aliases based on existing Error Type libraries without including their definitions in its namespace. This is accomplished by not using **extends** and by using qualified references to these Error Type libraries. For example, a user can define aliases for predeclared error types without extending the original Error Type library and only those aliases are part of this Error Type library.

Type System of Error Types

(27) Error types and error 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:

(28) $t_1 < t$ 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.

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

(30) 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 $t_1 \leq t_2$.

(31) 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 $\forall\, u_j, u_k \in U \mid j \neq k \wedge r(w_k) \neq r(u_j)$.

(32) 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, $w_1*w_2*.*w_n$. U is *contained* in W, $U \subseteq W$ if-and-only-if each element of U is a subtype of the corresponding element in W; $U \subseteq W$ iff $\forall\, u_j \in U\ \exists\, w_k \in W \mid u_k \leq w_j$.

(33) 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 $\exists\, ts_i \in TS \mid E \leq ts_i$.

(34) 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 $\forall\, E_j\, E_k \in TS \mid E_j \not\subset E_k \wedge E_k \not\subset E_j$.

(35) 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 $\forall\ ts_1 \in TS_1\ \exists\ ts_2 \in TS_2 \mid ts_1 \subseteq ts_2$.

E.6 A Common Set of Error Types

(1) This section introduces a common set of error types that are predeclared as a standard Error Type library called *ErrorLibrary*. These predeclared error types can be used to characterize error propagations, error events, and error states. Occurrence instances of error events typically map into a small number of error propagation types that are common across domains and are reflected in this library, e.g., omission, value, and timing related errors.

(2) Users can adapt the predeclared error types by defining aliases 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 types and subtypes.

(3) Users can introduce their own error type libraries to define domain and component-specific error types, e.g., to characterize overheating of a component. Such error event instances then trigger transition to specific error states and result in error propagations of predeclared or user-defined error types.

(4) The error propagation follows propagation paths defined in the core AADL model (port connections, shared data access, subprogram service calls, execution platform bindings) as well as propagation paths defined in Error Model subclauses (see Section E.7.3). This allows us to map a large number of software component faults into a limited number of error propagation error types. In the case of software components this assumes that they operate in a fault container (process with runtime enforced address space protection, or partition with both space and time partition enforcement). 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] defines 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 as a sequence of service items provided by a component through its features or via bindings. We then give definitions of service errors in the value and time domains as they actually occur independent of whether they have been detected by the system. *Note that intended detection by the system is specified in the component error behavior specification (see SectionE.10.2) and processing of the detected error is modeled in core AADL and the AADL Behavior Annex.*

(7) A *service* S is defined as a sequence of *n service items* s_i with $n > 0$. 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 delivery time.

(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 sets V_i and time sets D_i 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) V represents the expected range (or sets) of possible values delivered by a service, i.e., $\forall i, (V_i \in V)$. D represents the expected range (or sets) of possible service operation times, 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, *concurrency* errors with respect to the service as a shared resource, *authorization* errors with respect to the service providing controlled access to information, and *authentication* errors with respect to establishing the identity of the service. 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.

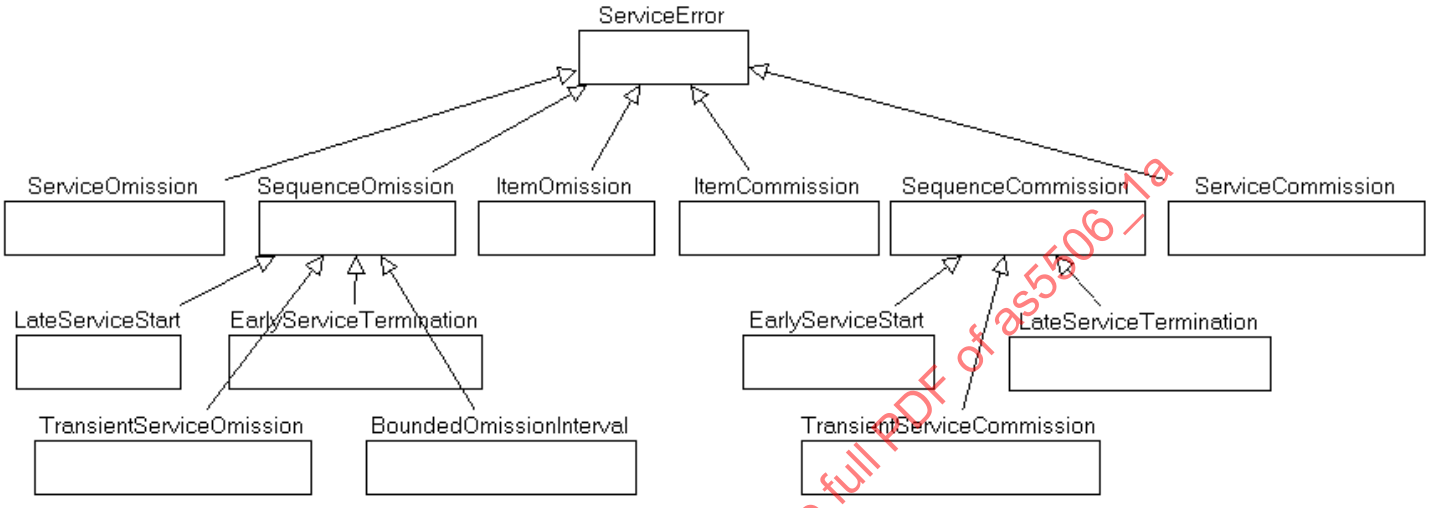


Figure 14 - Service error type hierarchy

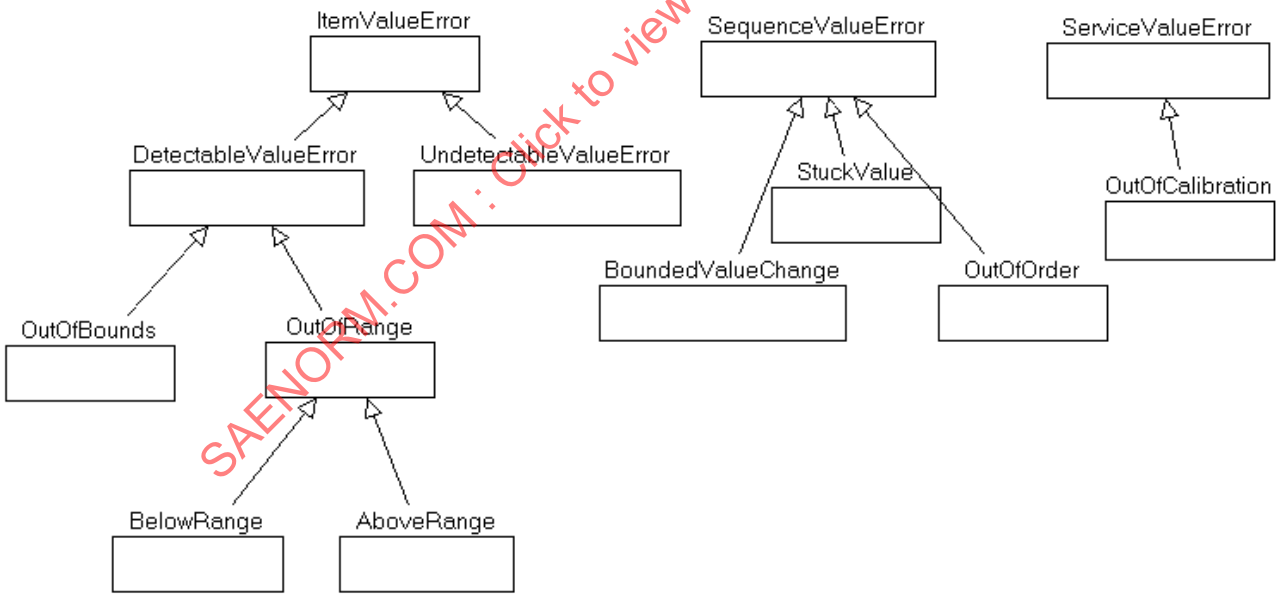


Figure 15 - Value related error type hierarchies

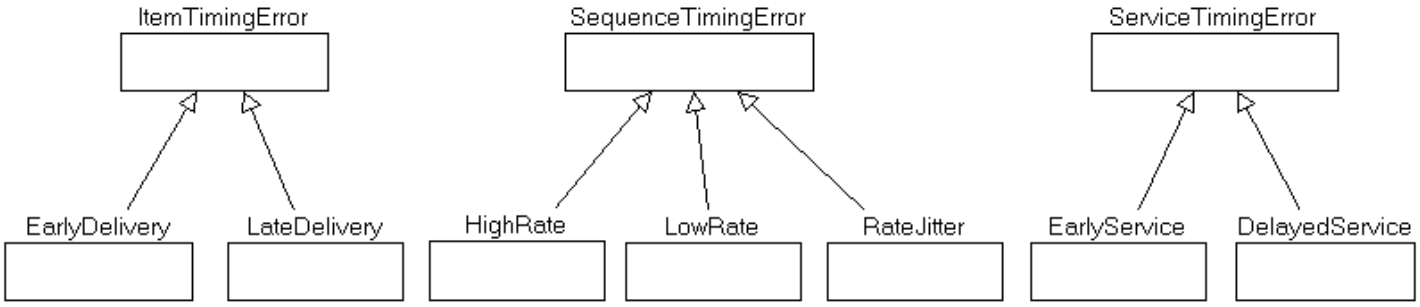


Figure 16 - Timing related error type hierarchies

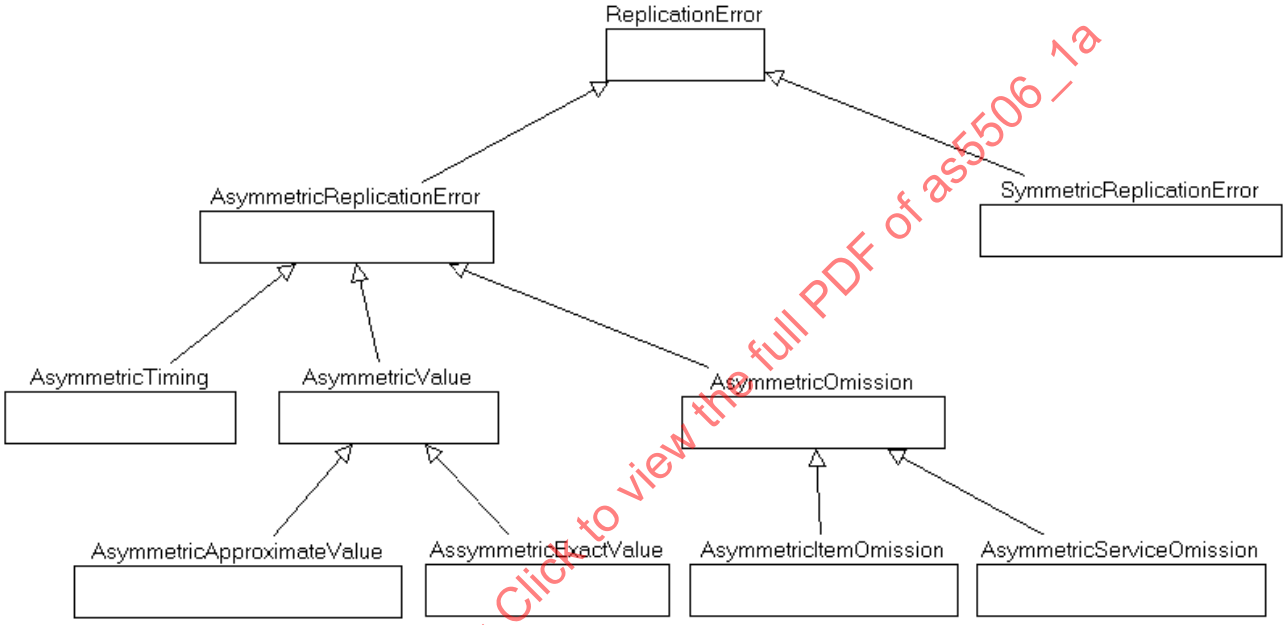


Figure 17 - Replication related error type hierarchies

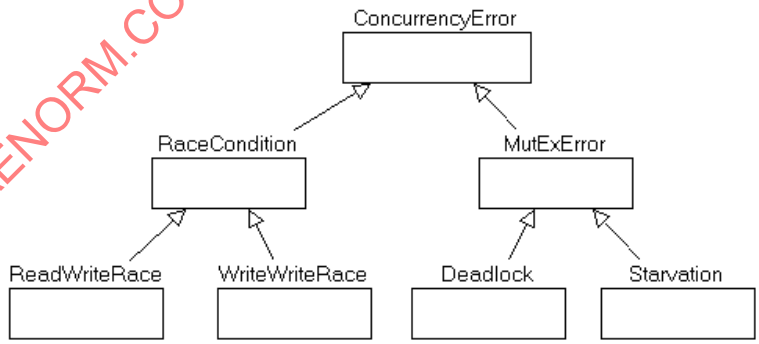


Figure 18 - Concurrency related error type hierarchy

```
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;

--authorization and authentication errors
AuthorizationError: type;
AuthenticationError: type;
```

```
end types;
**};
end ErrorLibrary;
```

E.6.1 Service Related Errors

(1) *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.

(2) *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.*

(3) *Item Omission* [ItemOmission] represents an error where one service item is not delivered.

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

(4) *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, cyclic redundancy check (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 service omission detection is based on a bounded sequence omission condition.

(5) *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: $\exists i \mid \forall j < i \mid s_j = \varepsilon$

(6) *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: $\exists i \mid \forall j > i \mid s_j = \varepsilon$

NOTE: Often early service termination detection is based on a bounded sequence omission condition.

(7) *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: $\exists\ [s_i\ s_{i+k}] \subset S \mid \forall\ s_j \in [s_i\ s_{i+k-1}] \mid s_j = \varepsilon \wedge s_{i+k} \neq \varepsilon$

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

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

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

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

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

(11) *Late Service Termination* [LateServiceTermination] represents an error where extra service items are provided after the service end time.

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

(1) 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.

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

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

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

(4) *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: $\exists a_i \in A \mid v_i \notin V_i \wedge B(v_i)$

(5) *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: $\exists a_i \in A \mid v_i \notin V_i \wedge \neg B(v_i)$

(6) 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.

(7) 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*.

(8) *Out Of Range* [OutOfRange] represents an error where a service item value falls outside the range of expected values for the service. 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 $v_i < min(V)$
NOTE: In practice a detection mechanism checks for the minimum and maximum expected value for the whole service.

(9) *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: $\exists s_i \in S \mid v_i \notin V_i \wedge O(v_i)$

(10) *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.

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

(12) *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.*

(13) *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: $\exists s_i \in S \mid \forall j > i: v_j = v_i$

(14) *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* $i \neq j$

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

(16) *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

(1) 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.

(2) *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: $\exists s_i \in S \mid \delta_i \notin D_i$

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

(4) *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: $\exists s_i \in S \mid \delta_i < D_i$

(5) *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: $\exists s_i \in S \mid \delta_i > D_i$

(6) *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: $\exists s_i \in S \mid r_i \notin R_i$

(7) *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: $\forall\ s_i \in S \mid r_i < R_i$

(8) *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: $\forall\ s_i \in S \mid r_i > R_i$

(9) *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: $\exists\ s_i \in S \mid r_i \notin R$

(10) *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: $\forall\ s_i \in S \mid (\delta_i + TD) \in D_i$

(11) *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: $\forall\ s_i \in S \mid (\delta_i + TD) \in D_i \wedge TD > 0$

(12) *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

(1) 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.

(2) 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 $\forall k \in [1,n] \mid v_i(k) \in V_i \wedge \delta_i(k) \in D_i$.

(3) 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.

(4) 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.

(5) 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: $\exists\ s_i \in S\ /\ \exists\ k \in [1,n] \mid v_i(k) \notin V_i$

(6) 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: $\exists\, s_i \in S \;/\exists\; k \in [1,n] \mid v_i(k) > V_i + h \vee v_i(k) < V_i - h$

(7) 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: $\exists\, s_i \in S \;/\exists\; k \in [1,n] \mid a_i(k) = \varepsilon$

(8) 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: $\exists\, s_i \in S \;/\exists\; j,k \in [1,n],\ \delta_i(j) - \delta_i(k) > \Delta$

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

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

(10) 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: $\exists\, s_i \in S \;/\forall\; k \in [1,n] \mid a_i(k) = \varepsilon$

(11) 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: $\exists\, s_i \in S \;/\forall\; k \in [1,n],\ \notin D_i$

(12) 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

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

E.6.6 Authorization and Authentication Related Errors

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

(2) Authentication-related errors [`AuthenticationError`] are related to authentication of services (roles, agents), of information, and of resources.

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;
```

E.7 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

(N27) 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 E.4 Naming Rule (N3)).

Semantics

(2) An **error propagations** section of an Error Model subclause consists of error propagation, error containment, 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.7.3). 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.7.1 Error Propagation and Error Containment Declarations

(1) 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 not to be propagated.

Syntax

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

error_containment ::=
    error_propagation_point :
      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
```

Naming Rules

(N28) 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 (recursively) exist in the feature group type of the feature group.

(N29) Propagation points referenced in an error propagation or error containment declaration must exist within the namespace of the subclause that contains the reference.

(N30) 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.

(N31) 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.

Legality Rules

(L12) For each error propagation point (feature, binding, user-defined propagation point) there must be at most one error propagation declaration and at most one error containment declaration.

(L13) 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.

(L14) The feature reference of error propagations for data components and bus components must be an access feature reference or the keyword *access*.

(L15) The binding reference of error propagations for software components must only contain the keyword **processor**, **memory**, **connection**, or **binding**.

(L16) The binding reference of error propagations for virtual bus, virtual processor, and system components may include the keywords **processor**, **memory**, **connection**, **binding**, and **bindings**.

(L17) The binding reference of error propagations for processor, memory, bus, and device components must only contain the keyword **bindings**.

(L18) 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.

(L19) 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. The common set of error types defined in the predeclared Error Type library *ErrorLibrary (see Section E.6)* provides a checklist of error types to be considered.

Semantics

(2) An error propagation declaration specifies that errors of the specific types are propagated into or out of a component through a feature, binding, or user-defined 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.

(3) An *error containment* declaration allows the modeler to explicitly specify, which error types are 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 expects an error of this type to not be propagated to it.

(4) 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.

(5) 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.

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

(7) 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. The keyword **binding** is used to identify the binding point of a functional architecture component to a system architecture component (expressed by the predeclared *Actual_Function_Binding* property). 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.

(8) An error propagation may occur between two components that do not have a connection or 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.7.3).

(9) 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 to be propagated or whether the Error Model specification is incomplete and unspecified error types may be propagated.

(10) Incoming error propagation and containment declarations represent assumptions a component makes about components interacting with it, while outgoing error propagations and containment declarations represent guarantees. These assumption and guarantees acts as contracts that can be verified compositionally according to consistency rules defined in Section E.7.3).

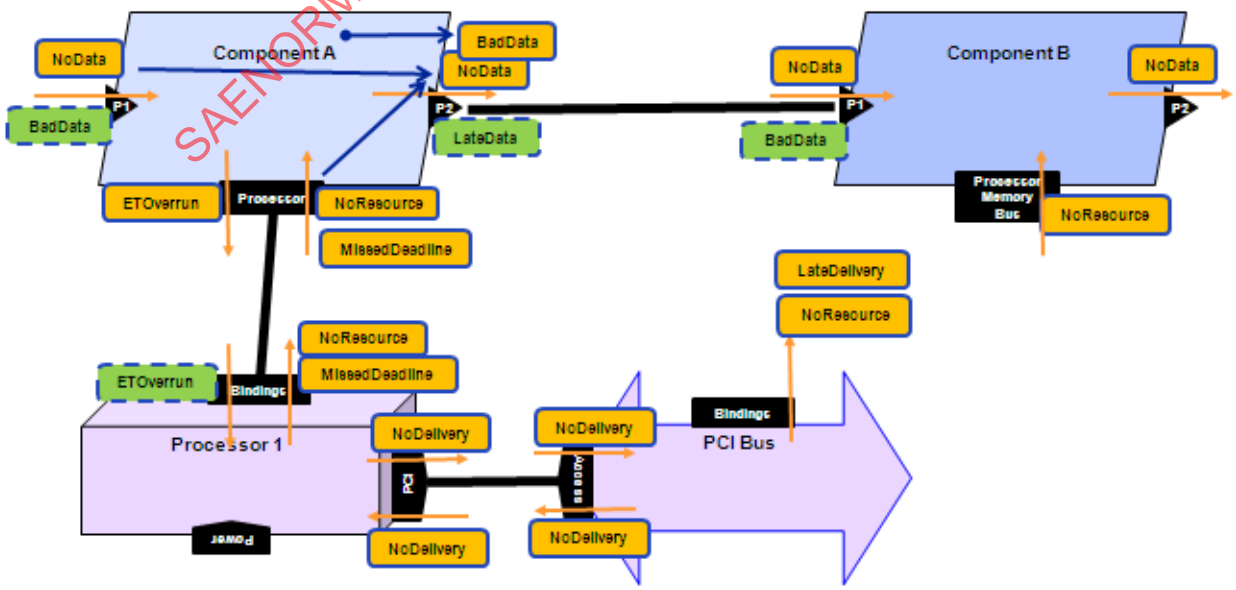


Figure 19 - Error propagations and flows between software and hardware components

(11) *Figure 19* shows two software components and two hardware components with error propagations along various error propagation paths between components (see SectionE.7.3) and error flows through a component (see Section E.7.2). Examples of error propagation paths shown are a port connection, a bus access connection, and a processor binding. In the case of the port connection between component A and component B component A intends to propagate `BadData` errors and `NoData` errors and not propagate `LateData` errors, while component B expects `NoData` errors, does not expect `BadData` errors, and is silent on `LateData` errors. In the case of the processor binding `NoResource` and `MissedDeadline` errors are shown as propagating to the software and `ETOverrun` error is shown as propagating overrun of execution time budget as software error to the processor. Examples of shown error flows for component A to be the source of `BadData` errors. It also shows an error flow from incoming `NoData` to outgoing `NoData` as well as an error flow mapping a processor `NoResource` error to a `NoData` error in the software component A.

E.7.2 Error Flow Declarations

(1) 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 | all )
   [ effect_error_type_set ] [ when fault_source ] [ if fault_condition ] ;

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

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

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

fault_condition ::= string_literal;
```

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

```
failure_mode_description ::= string_literal;
```

Naming Rules

(N32) The defining identifier of an error flow must be unique within the namespace of the Error Model subclause in which it is defined.

(N33) Referenced error propagation points must have been declared as error propagation declarations for the same component as the error flow.

(N34) 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.

(N35) 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.

(N36) 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

(L20) The direction of error propagations 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.

(L21) 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.

(L22) 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

(C2) 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.

(C3) All error types or error type products of an incoming error propagation must be contained in the type set of at least one error path or error sink. All error types or error type products of an outgoing error propagation must be contained in the type set of at least one error path or error source.

(C4) If a component type includes core AADL flow specifications, then error flow specifications must exist for each of the flow specifications.

Semantics

(2) An error source declaration may include a specification of the failure mode or event (expressed by a **with**) as well as a condition of the system (expressed by **if**) that results in the outgoing propagation identified by the error source declaration (first level effect).

(3) 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.

(4) *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 *component error behavior* specification of a component must be consistent with its error flow specifications (see Section E.9 Consistency Rules (C17)-(C21)).

(5) An outgoing error propagation of a component feature or binding may represent an error 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.

(6) An incoming error propagation through a component feature or binding may represent an error sink as well as an error 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.

(7) 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.

(8) 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.

(9) 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.

(10) The keyword **all** indicates that an error flow specification applies to incoming or outgoing error propagations. In the case of an error source the component is an error source for all outgoing error propagations. In the case of an error sink, the component is an error sink for all incoming error propagations. 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 section E.10.1) are declared.

E.7.3 Error Propagation Paths and User-defined Propagation Points and Paths

(1) 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, users can declare propagation points and paths 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

(N37) The defining identifier of a *propagation point* identifier must be unique within the namespace of the subclause for which the propagation point is defined.

(N38) 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.

(N39) 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 20* later in this section):

(L23) The error type set of the outgoing error propagation must be contained in the error type set of the incoming error propagation.

(L24) The error type set of the incoming error containment declaration must be contained in the error type set of the outgoing error containment declaration.

(L25) The direction of the error propagation or error containment for the source must be outgoing and for the destination must be incoming.

Consistency Rules

(C5) 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.

(C6) 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.

(C7) 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

(2) 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 process, thread group, or thread to every other process, thread group, or thread that is bound to any common virtual processor, processor or memory (Note that address space boundary is enforced at the process level, i.e., two threads inside the same process may affect each other beyond the specified error propagation points).
- 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.

(3) 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.

(4) 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 20* 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 error *propagation* of the *same type or nothing* is specified for the given error type.
- 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 to be *silent* on the propagation of a known error type and the source to indicate *propagation* or also be *silent*.

(5) These consistency rules are reflected in the legality and consistency rules earlier in this section.

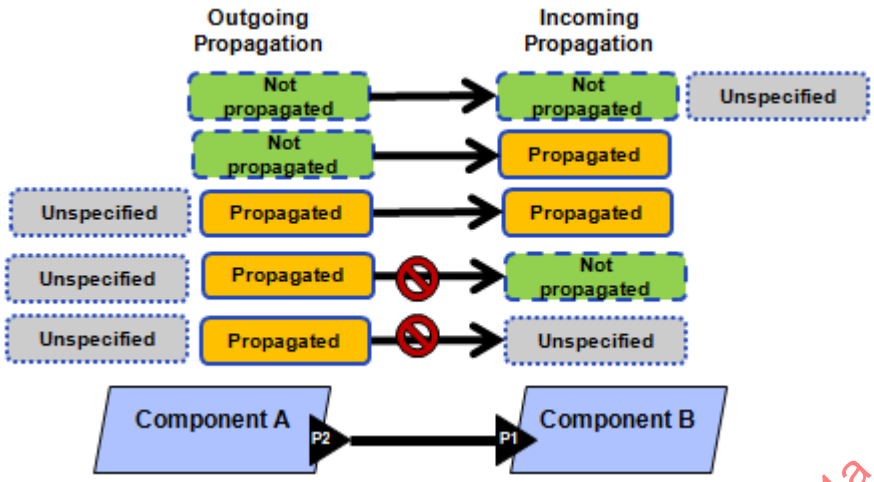


Figure 20 - Consistent and inconsistent error propagation paths

(6) A component may specify that it intends to contain errors of a given type. This is under the assumption that the component is implemented according to specification. A component may unintentionally propagate an error although it was declared to be contained. A destination component is robust to such unintentional error propagations if it expects such error types to be propagated.

E.8 Error Behavior State Machines

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

(2) The error behavior state machine can be defined as a typed token state transition system similar to a Colored Petri net, resulting in a more compact representation. This is accomplished by associating error types and type sets with error 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.8.3 elaborates on typed error behavior state machines.

(3) A component can show nominal behavior represented by one or more *working* states, or it can show anomalous behavior, represented by one or more *nonworking* states. A component can transition from a working state to a nonworking state as result of an activated fault (*error event*) or due to the propagation of 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 (section E.9) and a *composite error behavior* subclause (section 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 error_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 ]
    [ if 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
    [ when recover_event_initiators ]
    [ if error_event_condition ] ;

event_initiators ::=
  ( 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 ::=
```

```
    [ defining_error_transition_identifier : ]
    error_source_state -[ error_condition ]->
     ( error_transition_target | error_transition_branch ) ;

error_source_state ::=
    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 ]
    | [ in ] 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 |
    ( [ property_set_identifier :: ] real_property_identifier )
```

Note: the property name is interpreted as a symbolic label whose value is determined by its property value associated with the transition, or will be supplied separately to an analysis tool.

Naming Rules

(N40) 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.

(N41) 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.

(N42) 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.

(N43) 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 E.4 Naming Rule (N5)).

(N44) 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.

(N45) 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.

(N46) 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.

(N47) The behavior event reference of an error condition trigger must identify an error event, recover event, or repair event in the namespace of the error state machine containing the reference.

(N48) The incoming error propagation reference of an error condition trigger must identify an error propagation in the component that contains the error condition expression. The keyword **in** is used to qualify the error propagation reference, if it conflicts with a defining identifier of an error state or error behavior event.

(N49) 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.

(N50) 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.

(N51) 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

(L26) 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.

(L27) 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.

(L28) 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**.

(L29) The optional `error_type_set` of a *source* error state must be contained in the error type set specified with the defining state declaration.

(L30) The optional `error_type_set` of a *target* error state must be contained in the error type set specified with the defining state declaration.

(L31) 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.

(L32) 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.

(L33) 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 declaration in an Error Model library.

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 (section E.9) and *composite error behavior* specifications (section 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.8.1 Error, Recover, and Repair Events

(1) The Error Model language distinguishes between three kinds of error behavior events: error events, recover events, and repair events. An error event instance represents fault activation within a component and will result in a state transition to an error state that represents the resulting failure mode and in an outgoing error propagation. Recover events represent recovery from a nonworking state to a working state. This is used to model recovery from transient errors. Repair events represent longer duration repair action, whose completion results in a transition back to a working state.

(2) 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 represents out of range values and a separate error event represents late delivery of data.

(3) An error event can be annotated with 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 a transition.

(4) An error event may be annotated with the system condition (expressed by **if**) that results in the activation of the fault. This condition is specific to the component and may be expressed in terms of component properties, features, and state.

(5) 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 declared as part of the component error behavior declaration in the Error Model subclause specified for a component type or component implementation. In this case the value applies to all instances (subcomponents) of the classifier. 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).

(6) 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. The probability of an error type product is determined as the product of the type product element probabilities.

(7) 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.

(8) 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 an error state. By default it has an occurrence probability of 1.

(9) 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.

(10) 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.

(11) 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 internal event. If multiple initiators are listed, any of them can represent the initiation.

(12) 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.8.2 Error Behavior States and Transitions

(1) 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.

(2) 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.

(3) 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 (see Section E.9).

(4) 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 by declaring the target as **same state**.

(5) 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.

(6) 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.

(7) 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.

(8) 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.

(9) 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. Optionally, the user can explicitly indicate that the other error propagation points have **NoError**.

(10) 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.

(11) 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.

(12) The set of outgoing error behavior transitions 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 E.9.

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

(14) 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 E.9.

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.8.3 Typed Error Behavior State Machines

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

(2) 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.

(3) 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.

(4) 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.

(5) 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.

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

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

(8) 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* type instance 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 (see paragraph 44).

(9) 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 E.13 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.

(10) 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.

(11) 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;
  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;
```

E.9 Predeclared Error Behavior State Machines

(1) The Error Model language includes a set of predeclared reusable error behavior state machines. These state machines can be used in component error behavior and composite error behavior declarations and refined with component-specific information, such as occurrence probability of error and recover events, and with component-specific error behavior events and error behavior transitions.

Syntax

```
-- state machine for simple FailStop behavior
error behavior FailStop
events
Failure : error event ;
states
Operational : initial state ;
FailStop : state ;
transitions
FailureTransition : Operational -[ Failure ]-> FailStop ;
end behavior ;

-- state machine for Degraded then FailStop behavior
error behavior DegradedFailStop
```

```
events
Failure : error event ;
states
Operational : initial state ;
Degraded: state;
FailStop : state ;
transitions
FirstFailure : Operational -[ Failure ]-> Degraded ;
SecondFailure : Degraded -[ Failure ]-> FailStop ;
end behavior ;

-- state machine for Failure with Recovery behavior
error behavior FailAndRecover
events
Failure: error event ;
Recovery: recover event;
states
Operational: initial state;
Failed:  state;
transitions
FailureTransition : Operational-[Failure]->Failed;
RecoveryTransition :Failed-[Recovery]->Operational;
end behavior;

-- state machine for Degraded with Recovery and FailStop behavior
error behavior DegradedRecovery
events
Failure : error event ;
Recovery: recover event;
states
Operational : initial state ;
Degraded: state;
FailStop : state ;
transitions
FirstFailure : Operational -[ Failure ]-> Degraded ;
RecoveryTransition: Degraded -[ Recovery ]-> Operational;
SecondFailure : Degraded -[ Failure ]-> FailStop ;
end behavior ;

-- state machine for Transient and Permanent failure behavior
error behavior PermanentTransientFailure
events
Failure: error event ;
Recovery: recover event;
states
Operational: initial state;
FailedTransient:  state;
FailedPermanent:  state;
transitions
failtransition: Operational-[Failure]->(FailedTransient with EMV2::TransientFailureRatio,
FailedPermanent with others);
RecoveryTransition : FailedTransient-[Recovery]->Operational;
end behavior;


-- state machine for Failure with Recovery behavior where recovery can fail
error behavior FailRecoveryFailure
events
Failure: error event ;
```

```
Recovery: recover event;
states
Operational: initial state;
Failed:  state;
FailStop: state;
transitions
FailureTransition : Operational-[Failure]->Failed;
RecoveryTransition : Failed-[Recovery]->(Operational with EMV2::RecoveryFailureRatio, FailStop
with others);
end behavior;
```

Semantics

(2) The predeclared error behavior state machines are declared in the Error Model library called *ErrorLibrary*. These state machines can be used in component error behavior and composite error behavior declarations through the **use behavior** clause.

(3) Users of these state machines can specify component-specific property values for the *OccurrenceDistribution* of error events. Users of this state machine can also introduce component-specific error behavior events and error behavior transitions that are triggered by error behavior events and by incoming error propagations.

(4) The *FailStop* error behavior state machine represent error behavior of components whose failure occurrence (error event) results in *FailStop* error state, i.e., results in service omission behavior from an error propagation perspective.

(5) The *DegradedFailStop* error behavior state machine represent error behavior of components whose first failure occurrence (error event) results in *Degraded* error state, and a second error event in *FailStop* error state.

(6) The *FailAndRecover* error behavior state machine represent error behavior of components that have a failure and with specified probability are able to recover to full operation.

(7) The *DegradedRecovery* error behavior state machine represent error behavior of components that operate in degraded mode after one failure, are able to recover from *Degraded* to *Operational* with specified probability, and go into FailStop if a failure occurs while in *Degraded* error state.

(8) The *PermanentTransientFailure* error behavior state machine represent error behavior of components that permanent and transient failures. Users can specified component-specific values for the proportion of transient failures (*TransientFailureRatio*). Users can also specify the recovery occurrence and duration probability to transition back to *Operational*.

(9) The *FailRecoveryFailure* error behavior state machine represent error behavior of components that have a failure and with specified probability are able to recover to full operation, whose recovery may fail and result in *FailStop*. Users can specified component-specific values for the proportion of recoveries that fail (*RecoveryFailureRatio*).

Examples

(10) The following example makes use of the predeclared error behavior state machines. The sensor device has a component-specific occurrence probability, and the actuator device has a component-specific error event, transition, and occurrence probabilities, as well as a duration before recovery from the transient failure is complete.

```
package EBSMExampleUse
public
with EMV2;
device sensor
features
sensorReading: out data port;
annex EMV2 {**
use behavior ErrorLibrary::PermanentTransientFailure;
properties
   EMV2::OccurrenceDistribution => [ ProbabilityValue => 1.0e-4;
                       Distribution => fixed; ] applies to Failure;
**};
```