Software Engineering Institute

Dependability Modeling with the Architecture Analysis & Design Language (AADL)

Peter Feiler (Software Engineering Institute)
Ana Rugina (LAAS-CNRS)

July 2007

CMU/SEI-2007-TN-043

Performance-Critical Systems Initiative

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

The Society for Automotive Engineers (SAE) recently published an Error Model Annex document (SAE AS-5506/1) to complement the SAE Architecture Analysis & Design Language (AADL) standard document (SAE AS5506) with capabilities for dependability modeling. The purpose of this report is to (a) explain the capabilities of the Error Model Annex and (b) provide guidance on the use of the AADL and the error model in modeling dependability aspects of embedded system architectures. The focus of the guidance is the creation of error model libraries and the instantiation of these error models on AADL architecture models. In that context, the report discusses modeling of error propagation, error filtering and masking, the interactions between error models and systems with operational modes, and modeling of repair activities.

1 Introduction

This report aims to show how the Error Model Annex [SAE-AS5506/1 2006] standard can be used in conjunction with the description capabilities of the Architecture Analysis & Design Language (AADL) standard [SAE-AS5506 2004] to add dependability-related information—such as fault and repair assumptions, error propagations, fault-tolerance policies, and voting—to an AADL architecture model. The resulting annotated model can then be used as an input to dependability analyses for fault forecasting during different phases of the development cycle.

Each dependability analysis requires specific dependability-related information from the model. This information may include

- fault assumptions
- repair assumptions
- fault-tolerance mechanisms
- stochastic parameters of the system (i.e., the occurrence of fault events and propagations)
- characteristics of phases in a phased-mission system

Depending on the analysis to be performed, the model will look different. For example, in the case of qualitative dependability analyses, no stochastic and timing properties are needed in the model. For a fault-tree analysis, repair assumptions do not need to be taken into account.

We assume that the reader is familiar with the concepts of fault-tolerance and dependability analysis. The reader is referred to the *Dependability Handbook* for detailed information on these topics [Arlat 1998]. This report is structured as shown in Table 1:

Table 1: Content in this Document

Section	Description of Content
2	Presents the scope of the AADL Error Model Annex
3	Identifies the constructs of the AADL core language relevant to error modeling and comments on the level of architectural detail necessary for dependability-oriented modeling
4	Explains mechanisms for reusing error models
5	Shows the architecture-dependent parts of the dependability-related information (i.e., the parts that cannot be reused across architectures)
6	Presents hierarchic error modeling options
7	Discusses issues related to dependability modeling for systems with operational modes
8	Explains how you can deal with maintenance and repair in AADL models
9	Presents the mechanisms that allow you to specify elements in the model of interest to specific dependability analyses

2 Dependability Modeling with the Error Model Annex

The Error Model Annex can be used to annotate the AADL model of an embedded system to support a number of the methods cited in SAE ARP4761, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* [SAE-ARP4761 1996]. An architecture specification containing error models may be subjected to a variety of analysis methods. For example, fault trees can be generated from specifications to assess safety, or Markov analyses can be applied to assess reliability and availability.

The error models of low-level components typically capture the results of failure modes and effects analysis (e.g., as failure modes and effects analysis as defined in SAE ARP 4761). The error models of the overall system and high-level subsystems typically capture the results of system hazard analysis (e.g., as hazard analysis as defined in SAE ARP 4761). Figure 1 illustrates the use of error models at different levels of the system hierarchy. Error models can also be associated with connections between components to characterize any fault behavior of component interactions, such as the transfer of data.

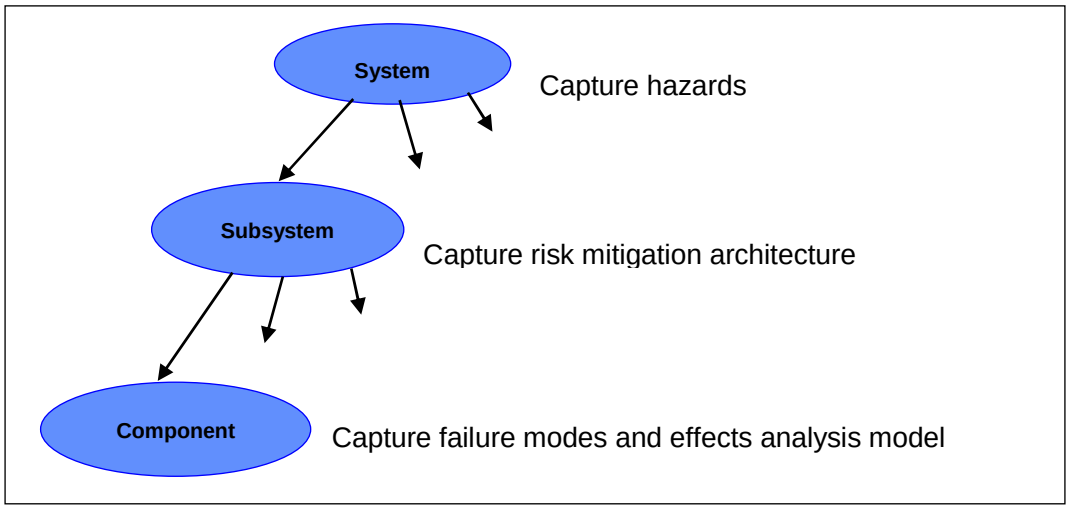


Figure 1: Error Models in System Hierarchy

The error behavior of a complete system emerges from the interactions between the individual component and connection error models. The system error model is a composition of the error models of its components where the composition is derived from the system hierarchy, the interactions between components, and the shared computing platform resources. For example, a component error model with probabilistic properties represents a stochastic automaton. The system error model represents the composition of the concurrent stochastic automata of the components in that system; it reflects error propagation between components based on the component dependencies in the AADL architecture model and the error management rules that are specified in error model annotations. Risk mitigation methods employed in embedded computer system architectures to increase safety, reliability, integrity, and availability are modeled by specifying how com-

ponents detect and mitigate errors in their subcomponents or in the components on which they depend.

The Error Model Annex supports mixed-fidelity modeling by annotating system components with different reusable error models. Mixed-fidelity modeling makes it easier to modify architecture specifications and automatically regenerate safety and reliability models at different levels of fidelity; it also enables improved traceability between architecture specifications and the generated models and analysis results.

You can define two kinds of reusable error models within an error model annex library: **basic error models** and **derived error models**. A basic error model declares a set of **error states** for a component or connection, together with **error state transitions** and properties to specify how the error state of a component changes due to **error events** and **error propagations**. For example, the error state of a component might change due to an internal fault, represented by an error event, or due to an error propagated into that component from some other component, represented by an error propagation.

In a derived error model, the **error state** of a component may be defined 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 erroneous states. In this case, error state transitions are not explicitly defined.

You annotate application system components and execution platform components through **error model annex subclauses**. These subclauses specify an error model from the library to be used for a component and component-specific **properties** of the error model, such as

- probability of occurrence of errors and error propagation
- logical guards that determine the effects those errors and error propagations have on component error states
- mappings of error states and error propagations in the error model into events on event ports of components
- mode transition conditions in terms of events through event ports, error states, and error propagations

You may use a basic error model as an abstraction for a given subsystem; a derived error model should be specified in terms of subcomponent and connection error models of that subsystem.

It is possible to check for consistency, completeness, and traceability between the error models of interacting components and between the error models of components and their subcomponents. This monitoring capability helps ensure a globally consistent and complete error model for the overall architecture. It also enables an integrated approach that ensures consistency and completeness between hazard analysis (HA) and failure modes and effects analysis (FMEA) and with the safety and reliability analyses that associate them together.

3 The AADL Architecture Model

To perform dependability analyses, you can describe a system's architecture in AADL and annotate this architecture model with error models containing relevant dependability-related information. AADL supports modeling of the embedded software system, the hardware platform, and the external environment as a set of interconnected application components mapped onto a set of interconnected execution platform components.

For dependability analyses, the architecture model does not need to be complete (i.e., the software does not need to be modeled to the level of threads and the hardware does not need to be described in terms of memory, processors, devices, and buses):

- The application software can be modeled using AADL system components; it can also be modeled to the level of partitions or that of processes and threads.
- The hardware platform can be modeled using AADL system components, or it can be modeled to the level of processors, memory, devices, and buses.
- The component may be defined at different levels of abstraction. For example, an AADL model with a real-time operating system or a bus type may represent a network including protocols. These execution platform component specifications can later be refined into models that provide the details of the implementation.
- Dynamic aspects of system architecture can be captured with the AADL mode concept. Different modes of a system or system component can represent different system configurations and connection topologies, as well as different sets of property values to represent changes in nonfunctional characteristics such as performance or fault occurrence.

It is only necessary to model the components that are of interest in the analysis (i.e., those for which the behavior in the presence of faults is considered). As a result, architecture models can be formed at early stages in the development process, when the architecture is not completely detailed. Later, they can be refined into a more detailed architecture representation for higher fidelity analysis.

The scope of the dependability analysis determines the aspects of the system to be modeled in AADL. The model may focus on representing the computing platform and the external environment, the embedded application software system, and an embedded application deployed on a particular execution platform. In the last case, the binding of the application system to the execution platform is expressed in AADL through a set of binding properties.

AADL supports the representation of end-to-end flows through the concept of a flow specification. End-to-end flows can be analyzed in the context of partially or fully complete AADL models. This flexibility allows for flow-related analyses to increase in fidelity as the architecture model is refined. Although not explicitly referenced in the Error Model Annex standard, end-to-end flow specifications can identify the relevant system components needed to document critical flows that must be considered in a reliability, availability, or fault-tree analysis.

You can find detailed information about the architecture description capabilities of AADL in *The Architecture Analysis & Design Language (AADL): An Introduction* [Feiler 2006] and the AADL standard document [SAE-AS5506 2004].

4 Reusable Error Models

In this section, we describe how to define error models that can be applied to a number of system components and tailored with component-specific information (i.e., error models that are reusable). We also illustrate how to associate such an error model to a component and provide examples of reusable error models.

4.1 ERROR MODEL DEFINITION

An error model is a state machine that can be associated with an AADL component or connection in order to describe its behavior in terms of logical error states in the presence of faults. Error models can be associated with (1) hardware components (processor, memory, device, and bus), (2) software components (process, subprogram, data, thread, and thread group), (3) composite components (system), and (4) connections.

An error model definition is divided into an error model type and an error model implementation. Elements declared in the error model type can be customized through component-specific properties, when an error model is associated with a component as an error model instance. Several error model implementations can correspond to the same error model type. Table 2 shows both an error model type declaration and an error model implementation declaration.

Table 2: Error Model Definition

```
error model Example1
features
ErrorFree: initial error state;
Failed: error state;
Fail, Repair: error event;
CorruptedData: out error propagation
   {Occurrence => fixed 0.8};
end Example1;

error model implementation Example1.basic
transitions
ErrorFree-[Fail]->Failed;
Failed-[out CorruptedData]->Failed;
Failed-[Repair]->ErrorFree;
properties
Occurrence => poisson 1.0e-3 applies to Fault;
Occurrence => poisson 1.0e-4 applies to Repair;
end Example1.basic;
```

The **error model** type `Example1` declares error states (i.e., `ErrorFree` and `Failed`), error events (i.e., `Fault` and `Repair`), and error propagations that can affect other components (i.e., `CorruptedData`). One **error state** (`ErrorFree`) is the initial state.

The **error model** implementation `Example1.basic` declares error transitions between states that are triggered by events and propagations. The **error model** instance is initially in the state `ErrorFree`. Due to a `Fault` error event, it becomes `Failed`. Then, after a `Repair`

error event it becomes `ErrorFree` again. While `Failed`, the component sends error propagations `CorruptedData`.

Both the error model type and the implementation can declare `Occurrence` **`properties`** for error events and error propagations. `Occurrence` **`properties`** specify the arrival rate (the language keyword is **`poisson`**) or occurrence probability (language keyword is `fixed`) of error events and outgoing error propagations. For the **`poisson`** arrival rate, the `Occurrence` property takes a single positive real value, which is the λ parameter in the exponential survival distribution $1 - e^{-\lambda t}$. For the fixed probability, the `Occurrence` property takes a single real value in the range [0.0, 1.0]. The `Occurrence` property can also have a user-defined distribution (indicated by the language keyword `nonstandard` and the distribution name) with one or more values. The Error Model Annex standard permits the `Occurrence` property to have literal expressions (i.e., μ, p, or 1-p).

If both the error model type and the error model implementation declare `Occurrence` **`properties`** for a same error event or error propagation, the property value declared in the error model implementation overrides the one declared in the error model type. The value declared in the error model type can be seen as a default value while the value declared in the error model implementation can be seen as an implementation-specific value (i.e., different implementations corresponding to the same type can declare different values for the `Occurrence` property of a same error event or error propagation). Either of these values can be replaced by a component-specific `Occurrence` value for each of the components with which the error model is associated (see Section 5.1).

Observations

- Elements declared in an error model definition can have slightly different meanings according to the dependability analysis to be performed. For example, a state can represent a failure mode identified in an FMEA analysis or a hazardous state identified in a hazard analysis.
- Although called **`error state`**, states can represent error-free states as well as error states.
- Although called an **`error event`**, this logical event may represent a repair event as well as a fault event.
- Some analyses do not involve probabilistic dependability measures; therefore, they do not require the definition of occurrence properties (i.e., `Occurrence` property declarations are optional).
- Note that error events and error propagations are logical events that may represent transient faults. They are not port events that are communicated through event ports. Error events can be declared in error models that can be associated with any kind of AADL component and connection, even with components that cannot communicate events through event ports. For example, one can associate an error model declaring error events and propagations with a memory component. Although memory components are not able to send events through ports, error events and states of the associated error model may be observed by the system; they may be mapped into port events (`Guard_Event`) and may specify a condition for transition to a different operational mode (`Guard_Transition`).

4.2 ERROR MODEL ANNEX LIBRARIES

Error model definitions like the one shown in Table 2 are meant to be reusable. They are defined as error model annex libraries separately from AADL component types and component implementations.

An error model annex library is declared as shown in Table 3. Several error model definitions are declared between the constructs **annex** `Error_Model {**` and `**};`.

Table 3: Error Model Annex Library

```
package My_ErrorModels
public
annex Error_Model {**
error model Example1
…
end Example1;

error model implementation Example1.basic
…
end Example1.basic;

error model Example2
…
end Example2;

error model implementation Example1.basic
…
end Example2.basic;
**};
end My_ErrorModels;
```

Error model annex library declarations can be placed in AADL packages or in the local (anonymous) namespace of an AADL specification. When declared in an AADL package, an error model can be referenced by the package name and the error model name from within error model annex subclauses of any component type or component implementation. When declared in the local namespace, the error model can be referenced by its name and can only be referenced within error model annex subclauses in component types and component implementations declared in the same local namespace.

Observations

- Different error models can be defined with the same name as long as the error model definitions are placed in different AADL packages.
- The AADL standard limits each AADL package to one annex library declaration for each annex. Consequently, all error model definitions in a package must be placed in the error model annex library declaration.

4.3 EXAMPLES OF ERROR MODEL DEFINITIONS

This section includes some examples of error model definitions that could be placed in a library and then applied to and customized in AADL models. We first define a simple error model for an

isolated system component (i.e., a system component whose errors do not affect other components and that is not affected by errors of other components). In that context, we illustrate how to model faults and repairs as well as transient and permanent faults. We then define an error model for system components that propagate errors and are affected by propagated errors.

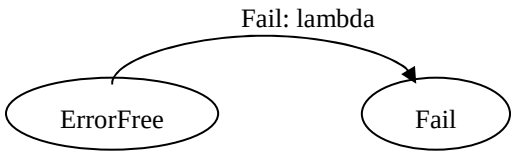
4.3.1 Fault and Repair Models for an Isolated Component

Table 4 shows a simple two-state error model definition that models faults in components. We refer to it as a **fault model**. It declares two error states, `ErrorFree` and `Failed`, and one error event, `Fail`. This error event triggers a transition between the two states. This error model definition does not declare any propagation, so it cannot influence the behavior of any components that interact with the component to which it is associated.

This fault model definition is simple for three reasons:

1. It takes failure into account.
2. The behavior of any component in the presence of faults can be described in terms of the `ErrorFree` and `Failed` states.
3. It declares a literal `Occurrence` property value for events that represent faults, which can be tailored for each component.

Table 4: Fault Model Definition for Isolated Component

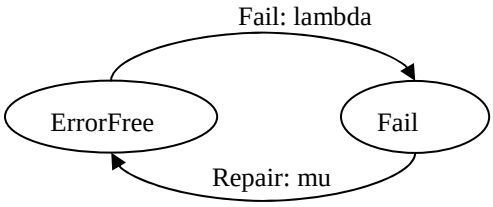



```
error model TwoStateFault
features
ErrorFree: initial error state;
Failed: error state;
Fail: error event {Occurrence => poisson lambda};
end TwoStateFault;

error model implementation TwoStateFault.general
transitions
ErrorFree-[Fail]->Failed;
end TwoStateFault.general;
```

This error model can be extended to include repair behavior as a **fault and repair model**. For this model, we add an event to represent that a component can repair itself or be repaired. This repair event is then used to specify a transition from the `Failed` to the `ErrorFree` state. A fault and repair model is illustrated in Table 5.

Table 5: Fault and Repair Model Definition for Isolated Component




```
error model TwoStateFaultRepair
features
ErrorFree: initial error state;
Failed: error state;
Fail: error event {Occurrence => poisson lambda};
Repair: error event {Occurrence => poisson mu};
end TwoStateFaultRepair;

error model implementation TwoStateFaultRepair.general
transitions
ErrorFree-[Fail]->Failed;
Failed-[Repair]->ErrorFree;
end TwoStateFaultRepair.general;
```

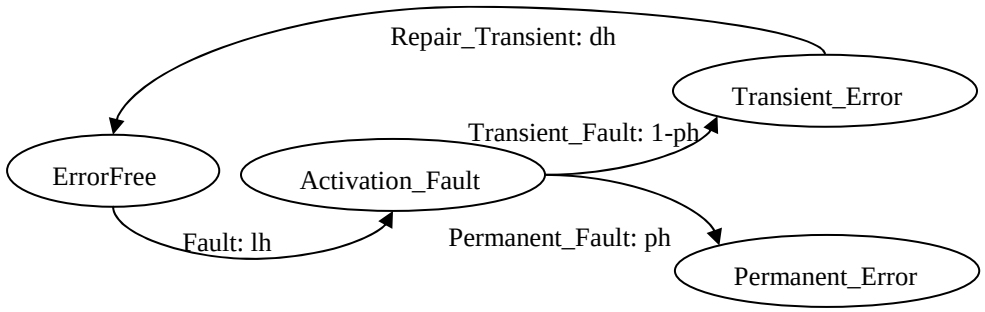
Observations

- Fault models can be used to represent fault information in dependency analyses that focus on faults, such as a fault tree analysis. The state machine represented by error states and error state transitions typically is without cycles (i.e., error events do not cause the error model to return to an error-free state). This allows the error model to be translated into fault trees.
- Fault and repair models can be used on system components that permit repair during the life of the system. Note that the state machine represented by error states and error state transitions is cyclical due to the fact that repair events may return the error model state to an error-free state. Fault tree analysis can still be performed, because the cycles of this state machine can be broken by distinguishing between fault events and repair events. To distinguish between event types, the error event should be tagged with a property (to indicate whether it is a fault or repair event) that can be interpreted by analysis tools.

4.3.2 Modeling Transient and Permanent Faults

Components, in particular hardware components, exhibit transient and permanent faults. Initially a component is in an `ErrorFree` state. Faults are activated with a specified rate, `lh`. A fault is permanent with a given probability (`ph`) and temporary with the complementary probability (`1-ph`). Errors caused by temporary faults disappear after a short period of time (`dh`). These behaviors are illustrated in Table 6.

Table 6: Fault Model with Transient and Permanent Faults




```
error model TransientPermanent
features
ErrorFree: initial error state;
Activation_Fault, Transient_Error, Permanent_Error: error state;
Fault: error event {Occurrence => poisson lh};
Permanent_Fault: error event {Occurrence => fixed ph};
Transient_Fault: error event {Occurrence => fixed 1-ph};
Repair_Transient: error event {Occurrence => poisson dh};
end TransientPermanent;

error model implementation TransientPermanent.general
transitions
ErrorFree-[Fault] -> Activation_Fault;
Activation_Fault-[Transient_Fault] -> Transient_Error;
Activation_Fault-[Permanent_Fault] -> Permanent_Error;
Transient_Error -[Repair_Transient] -> ErrorFree;
end TransientPermanent.general;
```

Observations

- We introduced an `Activation_Fault` state that allows the specification of a probability of fault occurrence separately from the probability that the fault is a permanent versus a transient fault. The same model can be specified without that intermediate state, when the modeler specifies the occurrences of the permanent and transient faults as separate probabilities.
- We modeled the transient fault as persisting for a short period of time, reflected in the `Transient_Error` state and the transitions between it and the `ErrorFree` state. If the modeler cares to model only the occurrence of a transient error, not its duration, the `Transient_Error` can be eliminated, and a transition can be defined from the `ErrorFree` state to itself.

4.3.3 Modeling Error Propagation

In many systems, failing components affect other components, because the components interact or one component is an execution platform resource that an application component is bound to for execution. Impact dependency information exists in an AADL model (see Section 5.2) and is used when the modeler specifies how errors are propagated and how propagated errors are handled for a system component (see Section 5.3).

In this section, we demonstrate how an error model shows that a component can propagate errors to other components and be affected by errors propagated from other components through **error propagations** (see Table 7)**.** The error propagation property enhances the error model definition shown in Table 4 on page 9 by declaring error propagations and referring to them in error state transitions. The property declares outgoing and incoming propagations through an **`in out`** propagation declaration. The **`in out`** propagation declaration is a shorthand for declaring an **`in`** propagation and an **`out`** propagation with the same name.

An error propagation out of a component is specified with an **`out`** error propagation declaration. An **`out`** error propagation occurs spontaneously and randomly according to the specified occurrence probability, when it is named in an error state transition and the current error model state of the component is the origin of the transition.

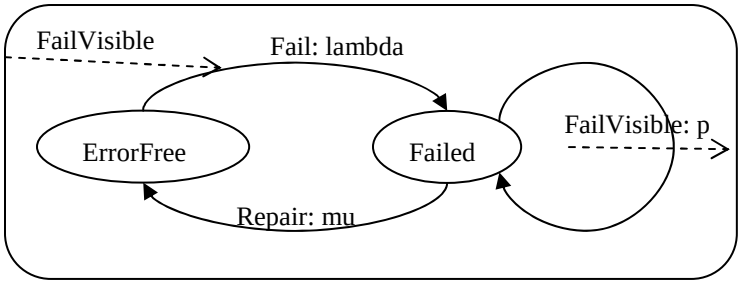
An **`in`** error propagation indicates that a component knows the propagations coming from other components by the specified name. The mapping of **`out`** error propagations of one component to an **`in`** error propagation of an impacted component is determined by name matching or explicitly specified as propagation guards for specific system components (see Section 5). The **`in`** propagation can be named in an error state transition to indicate that any error propagated from another component results in a transition to the destination state of that transition declaration.

In Table 7, we can assume that a component failure influences the behavior of components that depend on it. We make this action visible through the error propagation `FailVisible` that occurs with a given probability `p`. The `Occurrence` property only applies to the `FailVisible` **`out`** propagation. **`In`** propagations are the consequences of **`out`** propagations from other components; therefore, they do not need `Occurrence` properties.

The two supplementary transitions declared in the **`error model implementation`** specify respectively that

1. If the component is in the state `ErrorFree` and receives a `FailedVisible` **`in`** propagation, it goes to the state `Failed`.
2. The component remains in the state `Failed` when propagating out the `FailedVisible` propagation.

Table 7: Error Model Definition for Component with Error Propagation

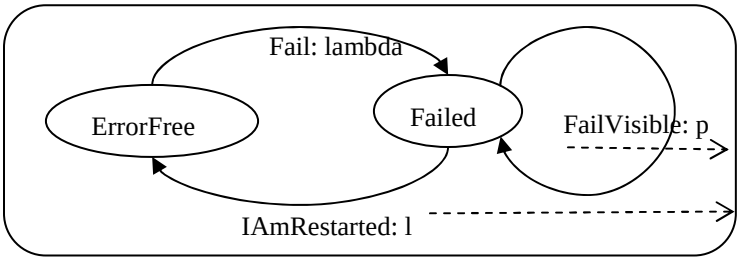



```
error model dependent
features
ErrorFree: initial error state;
Failed: error state;
Fail: error event {Occurrence => poisson lambda};
Repair: error event {Occurrence => poisson mu};
FailedVisible: in out error propagation {Occurrence => fixed p};
end dependent;

error model implementation dependent.general
transitions
ErrorFree-[Fail]->Failed;
Failed-[Repair]->ErrorFree;
ErrorFree-[in FailedVisible]->Failed;
Failed-[out FailedVisible]->Failed;
end dependent.general;
```

Table 8 shows an error model that models components that can observe failure of other components but their error state is not affected by error propagation. A component may fail and can be restarted to regain its `ErrorFree` state. **`Out`** propagations are used to notify when the component fails and when it is restarted.

Table 8: Error Model Definition with Error Observation




```
error model independent
features
ErrorFree: initial error state;
Failed: error state;
Fail: error event {Occurrence => poisson lambda};
FailedVisible: out error propagation {Occurrence => fixed p};
IAmRestarted: out error propagation (Occurrence => fixed l);
end independent;

error model implementation independent.general
transitions
ErrorFree-[Fail]->Failed;
Failed-[out IAmRestarted]->ErrorFree;
Failed-[out FailedVisible]->Failed;
end independent.general;
```

Observations

- In the models shown in Table 7 and Table 8, the `Fail` error event shows that a fault occurs in a component and is recognized as error. By defining a separate outgoing **`error propagation`**, we can represent that the error may not always be observed by another component or may be observed with a delay. We indicate those situations through an appropriate `Occurrence` property value of the **`error propagation`**.
- An **`error propagation`** reports an error state to other components. If their error states are affected, the other components will have a corresponding **`in`** propagation. Alternatively, those components may observe the **`out`** propagation through guards and take action based on the condition of the guard. (See Sections 5.4 and 5.5 for information about guards.)
- If the component fault is always and immediately visible to other components (i.e., the probability of occurrence of the propagation is 1), we could choose to declare `Fail` as an **`error propagation`** instead of an error event. However, the resulting error model would be less reusable because it includes an assumption that the failure is always visible.
- In general, the error state of a component is made visible to other components by declaring an **`error state`** transition that triggers an error propagation by naming an **`out`** propagation and has the same source and destination states (i.e., the propagation is sent out of the component but the component itself does not move to a different state). Figure 2 shows this mechanism. When the component is in `StateA`, it sends out an `I_Am_In_State_A` propagation and returns to `StateA`. It is noteworthy that the state is only visible if the **`out`** propagation occurs. If the **`out`** propagation occurs with a probability different from 1, the state may not be always visible.

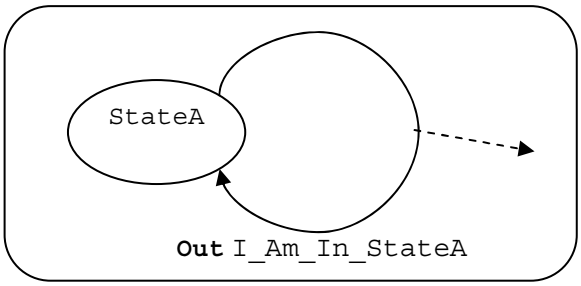


Figure 2: State Visible from Outside

The error model definition in Table 7 on page 13 can be considered to be more practical than the one in Table 4 on page 9, because it assumes that the AADL component to which it is associated interacts with other AADL components. The error model definition in Table 7 is general enough to be applied to any AADL component.

We can represent a more realistic behavior in the presence of faults by distinguishing different kinds of faults and consequences, as well as error detection mechanisms. The specification of an error model definition depends directly on the fault and repair assumptions considered in a given system and operational scenario. In the two following sections, we present error model definitions intended to describe the behavior of hardware and software components, respectively.

4.3.4 General Error Model for Hardware Components

In Figure 3, we define a general error model for hardware components. The behavior of the hardware component in the presence of faults is as follows:

1. Initially, the component is in `HW_ErrorFree` state.
2. Hardware faults (error event `HW_Fault`) are activated with a specified rate (`lh`) resulting in a transition to the `HW_Activation_Fault` state.
3. The fault is either permanent (error event `HW_Perm_Fault`), with a given probability (`ph`) triggering a transition to the `HW_Permanent_Error` state, or transient (error event `HW_Trans_Fault`), with the complementary probability (`1-ph`) triggering a transition to the `HW_Transient_Error` state.
4. The error caused by a permanent fault may be detected after some time (`th`), represented in Figure 3 as a transition triggered by error event `HW_Detection_Action` to the `HW_Detection_Action_End` state.
5. An error caused by a permanent fault is either detected (`HW_Perm_Fault_Detect`) with a given probability (`dh`) or not detected (`HW_Perm_Fault_Non_Detect`) with a probability (`1-dh`).
   - If the error is detected, the hardware component is repaired as represented by the `HW_In_Repair` state.
   - If the error is not detected (represented by the `HW_Error_Non_Detect` state), the failure is perceived after a certain amount of time (`fph`). This behavior is shown as error event `HW_Failure_Perceived` and triggers a transition the `HW_In_Repair` state, after which the hardware component is repaired.

6. The component repair from a permanent fault takes some time (`muh`), with a transition triggered by error event `HW_Repair_Perm` to the `HW_ErrorFree` state.
7. An error caused by a transient fault disappears after a short period of time (`tfh`). This behavior is shown as a transition triggered by error event `HW_Repair_Trans` to the `HW_ErrorFree` state.

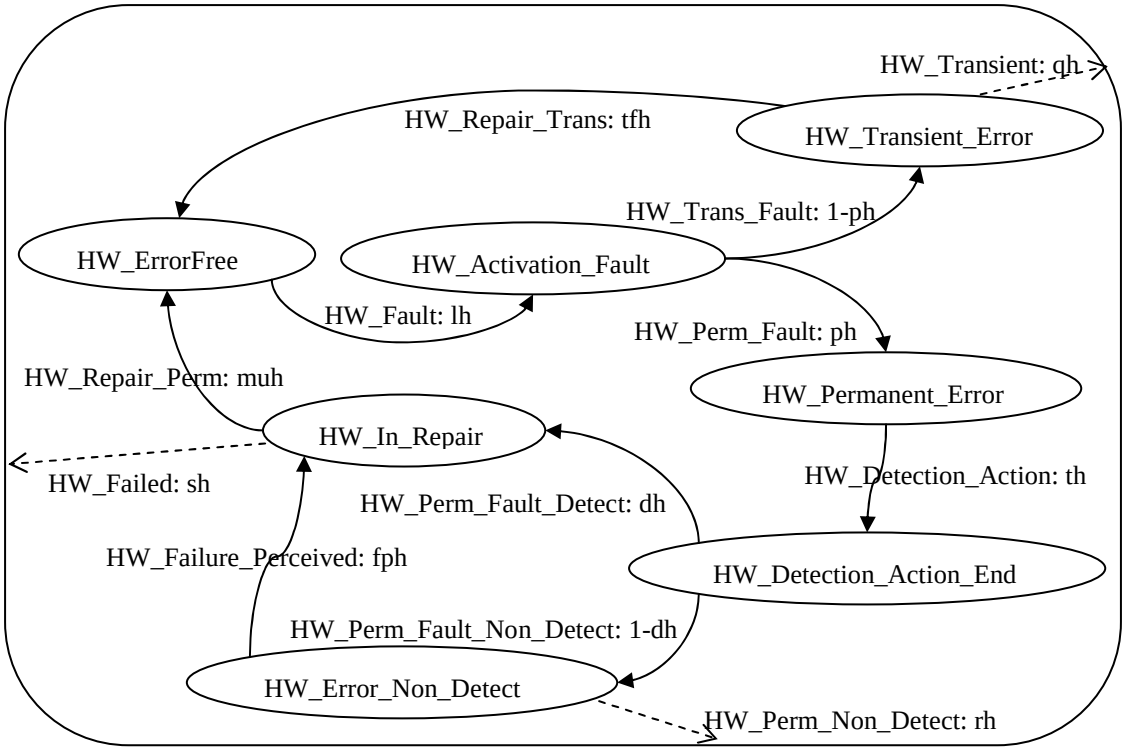


Figure 3: General Hardware Component Error Model

Some of the hardware states may influence other components of the architecture. Therefore, they must be made visible to the outside through outgoing error propagations (shown in Figure 3 as dashed lines with arrows). We assume that this influence exists for

- the transient error state

  The transient error state is observed by other components after a certain amount of time (`qh`) as represented by the outgoing error propagation `HW_Transient`.

- the state corresponding to a nondetected error

  The nondetected error state is observed by other components after a certain amount of time (`rh`) as represented by the outgoing error propagation `HW_Perm_Non_Detect`.

- the state where the hardware component needs repair

  The state representing a hardware component needing repair is observed by other components after a certain amount of time (`sh`) as represented by the outgoing error propagation `HW_Failed`.

Table 9 shows the error model definition corresponding to the specification depicted in Figure 3. Notice that we used the HW prefix for all error states, events, and propagations in order to ensure the readability of the model when several error models are associated with different system components. Also, propagation names are important, because **in** and **out** propagations declared in error model instances associated with interacting components are matched through their names or are named in guard specifications.

Table 9: Error Model Definition for a Hardware Component

```
error model forHardware
features
HW_ErrorFree: initial error state;
HW_Activation_Fault, HW_Transient_Error, HW_Permanent_Error,
HW_Detection_Action_End, HW_Error_Non_Detect, HW_In_Repair: error
state;
HW_Fault: error event {Occurrence => poisson lh};
HW_Perm_Fault: error event {Occurrence => fixed ph};
HW_Trans_Fault: error event {Occurrence => fixed 1-ph};
HW_Detection_Action: error event {Occurrence => poisson th};
HW_Failure_Perceived: error event {Occurrence => poisson fph};
HW_Perm_Fault_Detect: error event {Occurrence => fixed dh};
HW_Perm_Fault_Non_Detect: error event {Occurrence => fixed 1-dh};
HW_Repair_Trans: error event {Occurrence => poisson tfh};
HW_Repair_Perm: error event {Occurrence => poisson muh};
HW_Transient: out error propagation {Occurrence => fixed qh};
HW_Perm_Non_Detect: out error propagation {Occurrence=> fixed rh};
HW_Failed: out error propagation {Occurrence => fixed sh};end
forHardware;

error model implementation forHardware.general
transitions
HW_ErrorFree-[HW_Fault]-> HW_Activation_Fault;
HW_Activation_Fault-[HW_Trans_Fault]-> HW_Transient_Error;
HW_Activation_Fault-[HW_Perm_Fault]-> HW_Permanent_Error;
HW_Transient_Error -[HW_Repair_Trans]-> HW_Err_Free;
HW_Permanent_Error-[HW_Detection_Action]-> HW_Detection_Action_End;
HW_Detection_Action_End -[HW_Perm_Fault_Detect]-> HW_In_Repair;
HW_Detection_Action_End-[HW_Perm_Fault_Non_Detect]->
HW_Error_Non_Detect;
HW_Err_Non_Detect-[HW_Failure_Perceived]-> HW_In_Repair;
HW_In_Repair-[HW_Repair_Perm]-> HW_ErrorFree;
HW_Transient_Error -[out HW_Transient]-> HW_Transient_Error;
HW_Error_Non_Detect-[out HW_Perm_Non_Detect]->  HW_Error_Non_Detect;
HW_In_Repair-[out HW_Failed]-> HW_In_Repair;
end forHardware.general;
```

The **error model** definition shown in Table 9 can be associated to a hardware component and then customized by using particular values for Occurrence parameters. This definition models the fault and repair assumptions presented by Figure 3. If you want to consider additional assumptions, you can modify the model. For example, you can assume that errors might become visible outside the component when the component is in the state HW_Permanent_Err_State, even though the detection action has not taken place yet.

4.3.5 General Error Model for Software Components

We can consider the following behavior in the presence of faults for a software component. The error model is shown in Figure 4.

1. Initially, the component is in `SW_ErrorFree` state.
2. Faults (shown as error event `SW_Fault`) are activated with a specified rate, `ls`, leading to an `SW_Activation_Fault` state.
3. The error detection mechanisms need some time to detect an error (represented by the error event `SW_Detect_Action` with distribution `ts`), culminating in an `SW_Detection_Action_End` state.
4. An error can be detected (shown as error event `SW_Detected`) with a given probability `ds` or not detected (shown as error event `SW_Non_Detected`) with the complementary probability `1-ds`.
5. A detected error is processed during a certain amount of time (error event `SW_Handling` with distribution `pis`), triggering a transition to the `SW_Handling_End` state.
   - If the detected error is caused by a temporary fault (error event `SW_Error_Temp` with probability `1-ps`), its effects would be eliminated by the error detection mechanisms. Consequently, the component moves to the `SW_ErrorFree` state. (Note: It is assumed that all temporary faults can be eliminated.)
   - If the error is caused by a permanent fault (error event `SW_Error_Perm` with probability `ps`), the software would need to be restarted (`SW_In_Restart` state) to eliminate the effects of the error.
6. The effects of a nondetected error may disappear after a certain amount of time (error event `SW_Non_Detected_Disappear` with distribution `dis`) or may be perceived after a certain amount of time (error event `SW_Non_Detected_Perceived` with distribution `pcs`).
7. After recovery from a detected error due to a permanent fault or from a nondetected and perceived error, restart takes some time (error event `SW_Restart` with distribution `vs`).
8. Other components may observe the malfunctioning of the software component after a certain amount of time, once the `SW_In_Restart` state is entered. This behavior is shown as an outgoing error propagation `SW_Failed`.

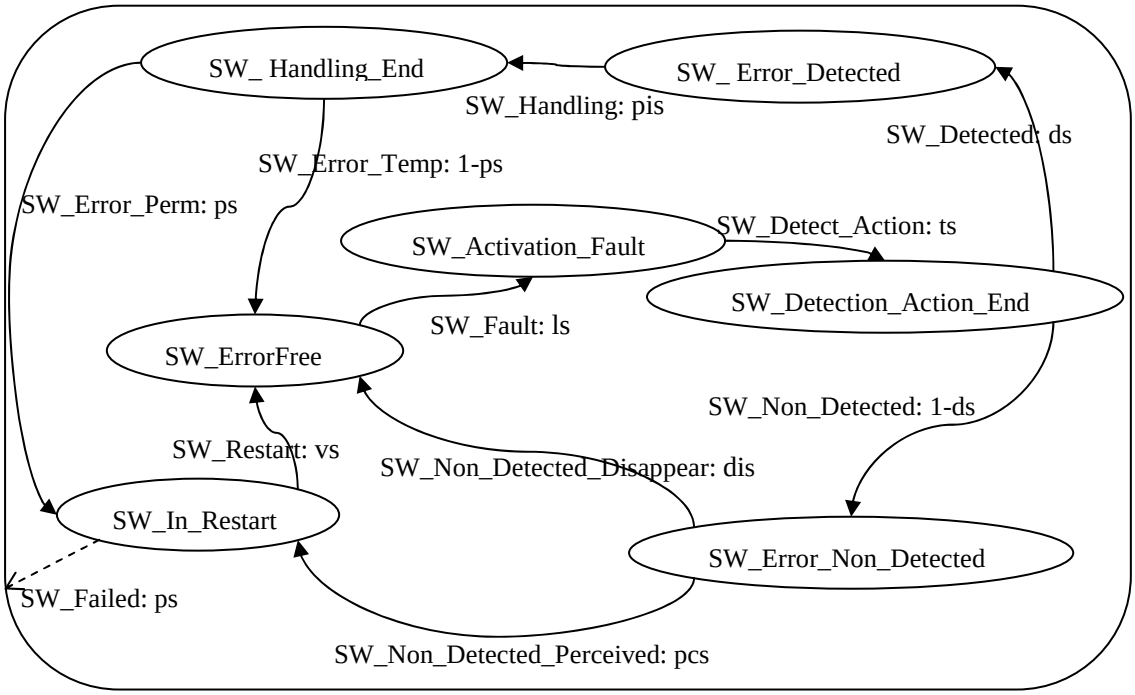


Figure 4: General Software Component Error Model

Table 10 shows the error model definition corresponding to the specification illustrated by Figure 4. As we did for the hardware error model definition, we tagged all states, events, and propagations with an `SW` prefix.

Table 10: Error Model for Software Component

```
error model forSoftware
features
SW_ErrorFree: initial error state;
SW_Activation_Fault, SW_Detection_Action_End, SW_Error_Non_Detected,
SW_Error_Detected, SW_Handling_End, SW_In_Restart: error state;
SW_Fault: error event {Occurrence => poisson ls};
SW_Detect_Action: error event {Occurrence => poisson ts};
SW_Detected: error event {Occurrence => fixed ds};
SW_Non_Detected: error event {Occurrence => fixed 1-ds};
SW_Non_Detected_Disappear: error event {Occurrence => poisson dis};
SW_Non_Detected_Perceived: error event {Occurrence => poisson pcs};
SW_Handling: error event {Occurrence => poisson pis};
SW_Error_Temp: error event {Occurrence => fixed 1-ps};
SW_Error_Perm: error event {Occurrence => fixed ps};
SW_Restart: error event {Occurrence => poisson vs};
SW_Failed: out error propagation {Occurrence => fixed ps};
end forSoftware;

error model implementation forSoftware.general
transitions
SW_ErrorFree-[SW_Fault]-> SW_Activation_Fault;
SW_Activation_Fault-[SW_Detect_Action]-> SW_Detection_Action_End;
SW_Detection_Action_End -[SW_Detected]-> SW_Error_Detected;
SW_Detection_Action_End -[SW_Non_Detected]-> SW_Error_Non_Detected;
SW_Error_Non_Detected-[SW_Non_Detected_Disappear]->SW_ErrorFree;
SW_Error_Non_Detected-[SW_Non_Detected_Perceived]-
>SW_In_Restart;SW_Error_Detected-[SW_Handling]-> SW_Handling_End;
SW_Handling_End -[SW_Error_Temp]->SW_ErrorFree;
SW_Handling_End -[SW_Error_Perm]-> SW_In_Restart;
SW_In_Restart-[SW_Restart]-> SW_ErrorFree;
SW_In_Restart-[out SW_Failed]-> SW_In_Restart;
end forSoftware.general;
```

4.3.6 Comparison of General Error Models for Hardware and Software Components

In the **error model** definitions shown in Table 9 and Table 10, we did not declare any **in** propagations because the matching between **in** and **out** propagations is done by matching names. Thus, we have to consider the architecture environment in order to choose names for **in** propagations. Section 5.1 shows how propagations are matched based on the AADL architecture model.

In the presence of faults, the error models for hardware and software components behave differently, as Table 11 shows.

Table 11: Error Model Handling Behavior

Aspect	Hardware Component Error Model	Software Component Error Model
The point at which faults are distinguished	Temporary and permanent faults are distinguished immediately following the fault activation during error handling.	Temporary and permanent faults are distinguished just prior to repair.
The result of permanent faults and perceived failures	Repair	Restart

5 System Architectures and Error Models

Using AADL, you can model systems as

- a hierarchical collection of interacting application system components
- a set of computing platform components
- a set of device components that represent the external environment

The application components are bound to the computing platform. Device components are logically connected to application components and physically connected to computing platform components.

Error models can be associated with application components, computing platform components, and device components, as well as with the connections between them. An error model associated with a component can be customized by setting component-specific values for the arrival rate or probability of occurrence for error events and error propagations declared in the error model type.

Interactions between the error models of different components are determined by interactions between components in the architecture model (i.e., connections and bindings). **`Out`** propagations are sent out of a component through all features connecting it to other components. Thus, **`out`** propagations have an effect on any receiving component that declares an **`in`** propagation with the same name.

You might need to model the handling of error propagations from multiple sources. In that case, you can use propagation filtering through voting mechanisms to control error propagations, which can be modeled by specifying filtering and masking conditions for propagations in an error model to a component.

In this section, we describe

1. how error models can be associated with components (Section 5.1)
2. the component dependencies through which errors can be propagated (Section 5.2)
3. error propagation across components in the context of a simple system model (Section 5.3)
4. the mechanisms to model masking and filtering of error propagations (Sections 5.4 and 5.5)

5.1 ASSOCIATION OF ERROR MODEL INSTANCES

You can choose an error model definition and associate an instance of it with an AADL component. Error model instances are declared for system components or connections through the `Model` property in an error model annex subclause, as shown in Table 12. The `Model` property names the error model implementation to be used. If the error model definition is in a different package than the component, the package identifier precedes the name of the error model. The

error model is associated with any component that is an instance of the component implementation containing the Model property.

The Model property also allows you to define a new error model without placing it in an error model annex library. However, this option is not recommended, because the error model would not be reusable (i.e., it cannot be instantiated anywhere else).

You might define component-specific Occurrence properties for events and outgoing propagations declared in the error model type definition through the **applies to** error clause. This clause allows you to customize error models (i.e., to specify component-specific Occurrence properties for the same error model associated with several different components). In Table 12, we specify that computer.personal propagates corrupted data with a higher probability than the default specified as part of the error model type definition declared in the library. The **applies to** error clause associates a component implementation-specific Occurrence property value with the error event CorruptedData.

Table 12: Error Model Instance for Component Implementation

```
system computer
end computer;

system implementation computer.personal
annex Error_Model {**
   Model => My_ErrorModels::Example1.basic;
   Occurrence => fixed 0.9 applies to error CorruptedData;
**};
end computer.personal;
```

Error model property values can also be specified for specific instantiations of components (subcomponent declarations) and connections (see Table 13). The **applies to** clause is used with a Model property to specify an error model instance for the CPU subcomponent. The **applies to** clause may specify a path to a subcomponent or connection (i.e., a dot-separated sequence of subcomponent names ending with a subcomponent or connection name to identify a component or connection recursively contained in the component implementation).

Table 13: Error Model Instance for Subcomponent

```
system computer
end computer;

system implementation computer.personal
subcomponents
  CPU: processor Intel.DualCore;
  RAM: memory SDRAM;
  FSB: bus FrontSideBus;
annex Error_Model {**
      Model => My_ErrorModels::Example1.basic applies to CPU;
   Occurrence => fixed 0.9 applies to error CPU.CorruptedData;
**};
end computer.personal;
```

Similarly, the **applies to** error clause may specify a path to an error model feature of a subcomponent or connection (i.e., a dot-separated sequence of subcomponent names ending with a

subcomponent or connection name and followed by the error model feature name). Table 13 shows the `Occurrence` property being associated with the `CorruptedData` error propagation of the `CPU` subcomponent.

Observations

- A component-specific `Occurrence` property value overrides values declared in the error model definition (type and implementation).
- Similarly, the **`applies to`** error clause can be used to declare all component-specific `Occurrence` property values in the error model annex subclause of the **`system implementation`**. Thus, component-specific and connection-specific error model information can be placed with either each component declaration or the root component of the system.

5.2 ERROR PROPAGATIONS BETWEEN COMPONENTS OF THE SYSTEM

Propagation of errors between components is determined by their interdependencies. Those dependencies are defined as **dependency rules** in the AADL architecture model and fall into four categories (see Section 5.2.1). A second set of rules, defined as **inheritance rules**, determine propagations when a component or connection does not have an error model (see Section 5.2.2).

5.2.1 Dependency Rules for Propagations

The first category of dependencies is **Shared Hardware Dependencies**; the rules in this category are due to the fact that application software components execute on hardware (see Table 14)**.** The binding of the application components and connections to the execution platform components is specified through `Actual_Processor_Binding`, `Actual_Memory_Binding`, and `Actual_Connection_Binding` properties that indicate the processor, memory, bus, and device that application components and connections are bound to.

Similarly, the binding of a server subprogram call is specified through an `Actual_Subprogam_Call_Binding` property and is treated as a connection in the dependency rules shown in (i.e., the remote call can be affected by the hardware over which the call is routed).

Table 14: Shared Hardware Dependency Rules for Propagations

Rule No.	Propagations may occur from	Propagations may occur to
D-1	Processor component	Every thread bound to that processor
D-2	Processor component	Every connection routed through that processor
D-3	Memory component	Every software component bound to that memory
D-4	Memory component	Every connection routed through that memory
D-5	Bus component	Every connection routed through that bus
D-6	Device component	Every connection routed through that device

The second category of dependencies is **Application Interaction Dependencies**. The rules in this category (see Table 15) are due to the fact that application components interact with each other through port-based communication (data ports, event ports, event data ports, and their respective connections), through access to shared data (provides and requires data access), and through calls on services provided by another component (server subprogram call bindings expressed through a `Actual_Subprogam_Call` property).

Table 15: Application Interaction Dependency Rules for Propagations

Rule No.	Propagations may occur from	Propagations may occur to
D-7	Application component	Each of the data components it has access to through **`provides`** and **`requires`** data access declarations
D-8	Shared component	All components that access it to through **`provides`** and **`requires`** data access declarations[1]
D-9	Application component	Every connection from any of its **`out`** ports
D-10	Connection	Every component having an **`in`** port to which it connects[2]
D-11	Application component	Any component via its outgoing connections
D-12	Client subprogram	Every server subprogram to which a call is bound
D-13	Server subprogram	Every client whose calls are bound to that server

The third category of dependencies is **Hardware Interaction Dependencies**; the rules in this category are due to the fact that execution platform components are connected to each other through shared access to buses (see Table 16). This is expressed in an AADL model by requires and provides bus access declarations and connections.

[1] As a consequence of rules D-7 and D-8, an application component can affect any component with which it shares access to a data component. If read and write access properties are specified for data access, the flow of information can be taken into consideration in determining the impact.

[2] As a result of rules D-9 and D-10, an application component can affect any connected component through its outgoing connections.

Table 16: Hardware Interaction Dependency Rules for Propagations

Rule No.	Propagations may occur from	Propagations may occur to
D-14	Component	Each bus that is accessed by a component through a bus access connection
D-15	Bus	Each component that accesses the bus through a bus access connection

As shown in Table 17, the fourth category of dependency rules addresses special cases in system architecture.

Table 17: Dependency Rules for Propagations to Address Special Cases

Rule No.	Propagations may occur from	Propagations may occur to
D-16	Subcomponent	Every other subcomponent of the same process
D-17	Process	Every other process that is bound to any common processor or memory—except for processes that are partitioned from each other on all common resources
D-18	Connection	Every other connection that is routed through any common bus, processor, or memory—except for connections that are partitioned from each other on all common resources
D-19	Event connection	Every mode transition that is labeled with an `in` event port that is a destination of that connection

Figure 5 illustrates propagation between error model instances based on application component interactions and execution platform bindings. The upper portion of the figure shows two components (A and B) connected by a port connection. Both the components and the connection have an error model instances. Error propagation occurs in the direction of the port connection flow. The lower portion of the figure illustrates that processors and the buses have error model instances as well. Error propagation can occur between these hardware components due to the bus connectivity. Finally, as the upward pointing arrows in Figure 5 show, error propagation can occur between the hardware and the application components and connections, due to their binding to the execution platform.

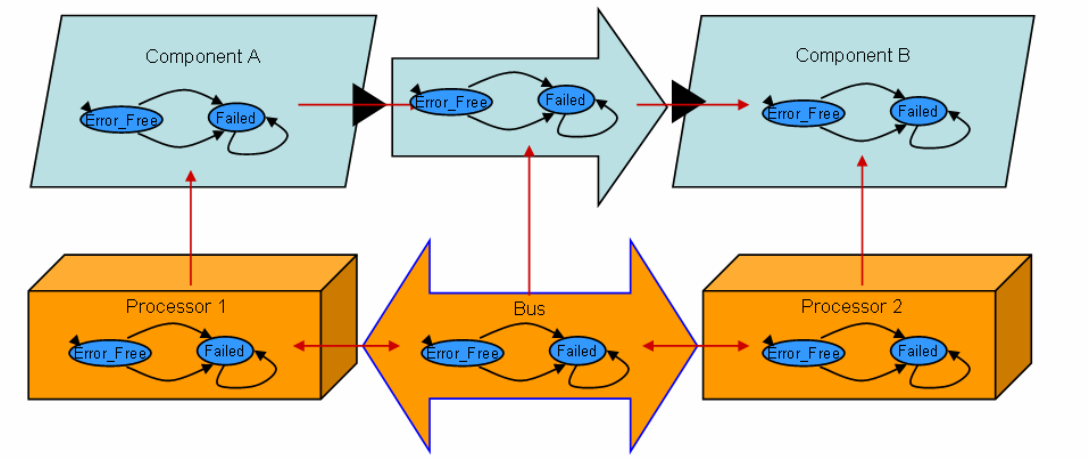


Figure 5: Execution Platform and Applications Error Propagation

5.2.2 Inheritance Rules for Propagations

A set of rules defines end-to-end error propagations (from one component error model to another one). These rules are designed to recognize that some AADL components will not have error model instances in a given architecture. Let us assume that a processor has an error model that declares out propagations but the threads bound to it do not have error model instances. In this case, no error model instance would be able to process those propagations. The set of inheritance rules for error propagation is shown in Table 18.

Table 18: Inheritance Rules for Error Propagation

Rule No.	Inheritance Rule Text
I-1	If the dependency rules define error propagations out of a component that does not have an associated error model but does have subcomponents with error models, then error propagations occur out of each subcomponent error model. Thus, where a component does not have an error model but its subcomponents do, error propagation occurs out of the subcomponents' error models and follows the dependency rules of the component.
I-2	If the dependency rules define error propagations out of a component that does not have an associated error model but does have a hierarchically containing component with an associated error model, then error propagations occur out of the error model associated with this containing component. Thus, where a component does not have an error model but its parent does, error propagation occurs out of the parent error model.
I-3	If the dependency rules define error propagations in to a component that does not have an associated error model but does have subcomponents with error models, then error propagations occur into each subcomponent error model. Thus, where a component does not have an error model but its subcomponents do, error propagation is passed on to the error models of the subcomponents.
I-4	If the dependency rules define error propagations in to a component that does not have an associated error model but does have a hierarchically containing component with an associated error model, then error propagations occur into the error model associated with this containing component. Thus, where a component does not have an error model but its parent does, error propagation is handled by the parent error model.
I-5	If the dependency rules define error propagations in to a semantic connection, then propagations occur to all ultimate destinations of that connection. Thus, where a connection does not have an error model, error propagation is passed from the component that is the origin of the connection to the component that is the destination.
I-6	If the dependency rules define error propagations in to a shared data component, then propagations occur to all other components that also share access to that data component.
I-7	Errors never propagate from an error model instance to itself.

5.3 ERROR PROPAGATION ACROSS ERROR MODELS

Let us assume that we have the following AADL architecture model: two system components are connected through a unidirectional data port connection that goes from `Component1` to `Component2`, as shown in Figure 6. We assume that the behavior of `Component2` depends on that of `Component1`, because `Component2` receives data from `Component1`. We also assume that the connection is perfect (i.e., it never fails). We associate error models to `Component1` and `Component2`, choosing to associate the same generic error model to both components, `dependent.general` as introduced in Table 7 on page 13.

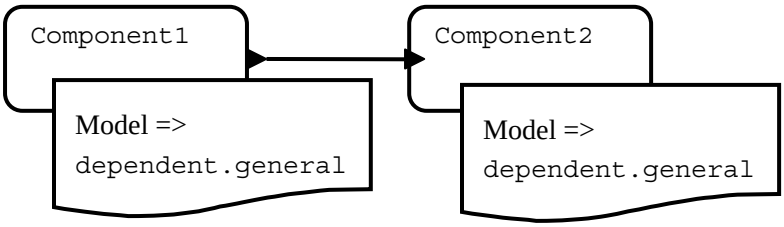


Figure 6: End-to-End Propagation

We can apply the dependency and inheritance rules from Sections 5.2.1 and 5.2.2 to the AADL model of Figure 6. Specifically, rules D-9, D-10, and I-5 apply to this architecture. `Component1` can propagate errors into the connection, and the connection can propagate errors into `Component2`. Also, propagations that go into a connection affect the ultimate destination of that connection if the connection itself does not have an error model instance. Consequently, outgoing propagations defined in the error model `dependent.general` and associated with `Component1` can affect the error model `dependent.general` associated with `Component2`, as the error model `dependent.general` declares incoming propagations whose names match those of the outgoing propagations. The name matching rule does not apply if a `Guard_In` condition is defined for an incoming propagation (see Section 5.4).

In short, due to the port connection from `Component1` to `Component2`, errors are propagated from the error model of `Component1` to the error model of `Component2`. Because port connections are directional, an error cannot be propagated from `Component2` to `Component1` unless there is also a port connection from `Component2` to `Component1`—even though the error model of `Component2` declares outgoing propagations that match incoming propagations in the error model of `Component1`.

As shown in Table 7 on page 13, the transition triggered by the `FailureVisible` **`in`** propagation in the **`error model`** `dependent.general` associated with `Component2` is a consequence of the transition that triggers the **`out`** propagation `FailureVisible` in the **`error model`** `dependent.general` associated with `Component1`. Both transitions occur according to the `Occurrence` property value of the **`out`** propagation in `Component1`.

If the `Occurrence` property probability value is not 1, `Component2` might be affected in two ways: (1) it might transition into the `Failed` state after `Component1` transitions into the

Failed state when the Fail event occurs in Component1, and (2) it might not observe the error of Component1 (i.e., there is a certain probability that the error in Component1 does not propagate and thus does not affect the behavior of Component2).

5.4 FILTERING OF INCOMING PROPAGATIONS

Propagations coming to a component can be filtered by using a Guard_In property. In the following subsections, we describe the role of this property and how it is used.

5.4.1 Role of a Guard_In Property

A Guard_In property allows you to

- unconditionally map the name of an incoming propagation and error state declared in a sender error model instance to a propagation name declared in the receiving error model
- conditionally map an incoming set of propagations and error states into a single (or a set of) **in** propagation(s)
- conditionally mask incoming propagations

5.4.2 Guard_In Property Application

A Guard_In property may be declared to apply to **requires** and **provides** data access features, incoming ports (data, event, or event data), and server subprogram features of a component. As a result, the Guard_In property is evaluated when error propagations occur into a component through those features (e.g., through an **in** data port or a shared data object or when there is a change in the error state of a sender component). Each component feature can only have one Guard_In property.

A Guard_In property consists of an ordered set of rules for incoming propagations. Each rule maps at least one outgoing (**out** or **in out**) propagation or error state from error models of connections or connected components (**when** clause) or an error state of the component itself to an incoming (**in** or **in out**) propagation; or, it specifies that the propagation or error state should be masked.

This mapping is illustrated in Figure 7. The Guard_In property is defined for ComponentA. The Guard_In rule is defined for the incoming error propagation InProp for port in1 of ComponentA (the outlined arrowed line in Figure 7). The rule condition is determined by examining the error states of ComponentA as well as the error states and outgoing error propagations of connections or connected components (shown as double-lined arrows for a component). The result of the guard rule can affect the error state transition if InProp is named as part of the transition condition (shown as solid arrow).

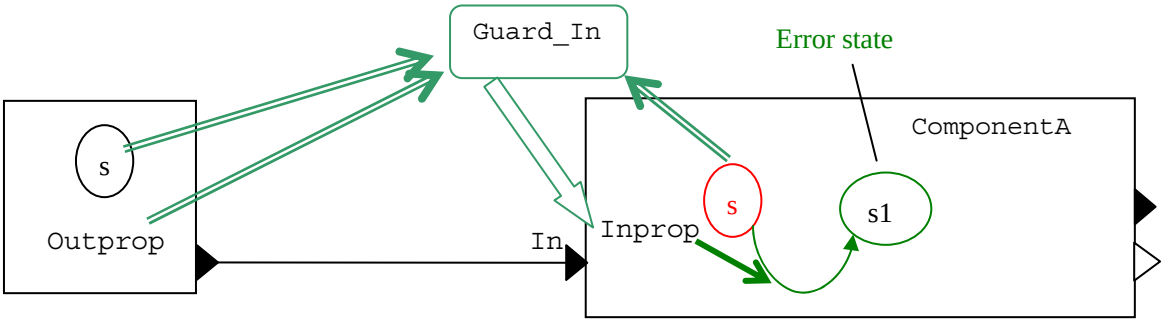


Figure 7: Guard_In Mapping

In Table 19, the `in_propagation_x` expression refers to a name of an incoming (**`in`** or **`in out`**) propagation declared in the error model of the component for which the rule is being specified. The **`when`** clause represents the guard expression to be evaluated to determine whether the rule applies.

Table 19: Guard_In Property Use

```
Guard_In =>
  in_propagation_1 when port_name_1[out_propagation_1],
  in_propagation_2 when port_name_2[error_state_1]
                        and port_name_3[error_state_2],
  mask when port_name_2[error_state_1] or
                port_name_3[error_state_2]

  applies to port_name_1;
```

The outgoing propagation(s) or error state(s) is specified in square brackets. If a reference does not include a bracketed outgoing propagation or error state, the initial error state is implied. If the connection does not have an error model, the outgoing propagation or error state in the error model of the component that is the origin of the connection is referenced. The single port reference for `in_propagation_1` in Table 19 represents an unconditional name mapping of the outgoing propagation name or error state of the origin into the **`in`** propagation name.

The **`when`** clause can

- reference a single incoming (**`in`** or **`in out`**) component port, **`requires`** data access, or server subprogram feature
- reference to an outgoing propagation or error state in the error model of the connection through the feature
  - If the connection does not have an error model, the outgoing propagation or error state in the error model of the connected component(s) is referenced.
- specify a condition that refers to multiple component features and an outgoing propagation or error state in the error model of the connection through the feature
  - This condition specifies how propagations and error states of origins of component interactions are handled. The condition may represent a voting protocol to determine whether the propagations and error states are masked or cause an error state transition through the named in propagation.

- include references to error states in the component for which the `Guard_In` mapping is defined
  - The references are included by referring to self and an error state of the component's error model, which allows filtering or masking of propagated errors based on the error state of the component that receives the propagations.
- contain the following logic operators: **`not`**, **`and`**, **`or`**, **`ormore`**, **`orless`**
  - The **`not`** operator has the highest precedence, followed by the **`and`** operator. The other operators have equal precedence and are evaluated from left to right in use, except where parentheses specify otherwise. A numeric literal appearing in an **`ormore`** or **`orless`** operator must be a positive integer.
- specify **`others`**
  - The **`when`** clause of the last rule might specify **`others`**, meaning that the rule applies if none of the previous rules apply.

`Guard_In` expressions are evaluated each time an error is propagated into a component through the feature for which the guard is specified. They are evaluated in the order of declaration until the first one is found whose Boolean error expression evaluates to TRUE. If a rule with `in_propagation` before the **`when`** clause evaluates to TRUE, the incoming propagation is considered to have occurred and caused a transition that names the incoming propagation. If a mask rule evaluates to TRUE, the propagation is suppressed and does not result in a transition between states for the receiving error model. If none of the guard rules evaluates to TRUE and there is no **`others`** clause, the specification is erroneous.

Observations

- The evaluation rules only require that guards be defined for those combinations of states and propagations that might occur for a specified system in the operational scenario being used for analysis.
- The AADL standard Error Model Annex specifications allow propagation filtering to be associated with component dependencies through port connections, shared access, and server subprogram calls. However, it does not support filtering or masking of error propagations through bindings to execution platform components.

5.4.3 Error Propagation Mappings

A `Guard_In` property maps outgoing propagation or error state names of components that it connects with the component to incoming propagations of the error model. Table 20 shows several examples of name mappings. In this table, it is assumed that the component connected through the port `Sensor1` has the **`error model`** `transientpermanent.general` associated, and the **`component`** connected through the **`port`** `Sensor2` has an **`error model`** associated with two outgoing propagations named `Error1` and `Error2`.

The first `Guard_In` property (see ① in Table 20) defines a mapping of an error state `Permanent_Error` in an error model associated with a connection or connected component that can impact the given component through `Sensor1` into the incoming error propagation `FailedVisible` of the error model of the impacted component `computer.personal`. This

guard is specified to apply only to connection or component error models reachable through `Sensor1`.

The second `Guard_In` property (see ② in Table 20) defines a mapping of several outgoing propagations `Error1` and `Error2` in an error model into an incoming error propagation `FailedVisible` declared in the error model associated with the impacted component `computer.personal`. This guard is specified to only apply to connection or component error models reachable through `Sensor2`.

Table 20: Error Propagation Mappings

```
system computer
features
Sensor1: in data port;
Sensor2: in data port;
end computer;

system implementation computer.personal
annex Error_Model {**
 Model => My_ErrorModels::transientpermanent.general;
 Guard_In =>                                    ①
  FailedVisible when Sensor1[Permanent_Error]
   applies to Sensor1;
Guard_In =>                                     ②
  FailedVisible when Sensor2[Error1,Error2]
   applies to Sensor2;
   **};
end computer.personal;
```

Observations

Error propagation mappings can be used for several purposes.

1. If the error models of two connected components or a component and the connection have error propagations with different names, the outgoing error propagation names of one model can be mapped to incoming error propagation names of the second model.
2. If the error model that is the origin of an error propagation has multiple error propagations to distinguish between different errors but the error model of an impacted component treats all errors the same way, the different types of outgoing error propagations can be mapped into the same incoming propagation. (For example, in Table 20, see the error model connected through `Sensor2`).
3. If the error model that is the origin of an error propagation does not have any outgoing propagations (i.e., has not been designed to explicitly propagate out error information), the dependent components can observe the error state of the originating component and map it into an incoming propagation. This is done in the example in Table 20 for the error model `transientpermanent.general`, which does not have any propagation.
4. Assume that an impacted component has two incoming features and the components that are connected through these features use the same error model definition, `dependent.general`. If the impacted component wants to distinguish between errors propagated through each feature, a separate guard can be defined for each component feature that maps the `FailedVisible` outgoing propagation into separate incoming propagations

of the impacted component error model. Each of these incoming propagations can trigger a transition to a different error state in the impacted component error model.

5.4.4 Error Propagation Filtering and Masking

The `Guard_In` property can specify filters for error propagations from other components that reflect voting protocols used to determine whether the impacted component should change its current error state. Using the same logic expressions, the `Guard_In` property can also specify the conditions under which error propagations are masked. The logic expression can name outgoing propagations, as well as error states of the components that can impact the given component. In addition, the logic expression can include conditions that reflect the current error state of the impacted component.

Table 21: Masking and Filtering of Error Propagations

```
system computer
features
Sensor1: in data port;
Sensor2: in data port;
end computer;

system implementation computer.personal
annex Error_Model {**
 Model => My_ErrorModels::dependent.general;
 Guard_In =>
 mask when (Sensor1[FailedVisible] and Sensor2[ErrorFree])
   or (Sensor1[ErrorFree] and Sensor2[FailedVisible]),
 FailedVisible when (Sensor1[FailedVisible] and
Sensor2[FailedVisible])
   applies to Sensor1, Sensor2;
   **};
end computer.personal;
```

Table 21 shows an AADL system component that has two **in** ports, `Sensor1` and `Sensor2`. We associate the error model `dependent.general` (detailed in Table 7 on page 13) with the component implementation. We declare a `Guard_In` property that applies to both **in** ports. We assume that the components connected to `Sensor1` and `Sensor2` both have associated error models `dependent.general`.

The `Guard_In` property describes the following propagation controls:

- When a `FailedVisible` propagation (declared as an **out** propagation in the error model associated with a component connected to a sensor port) comes through only one of the sensor ports, that propagation is masked. As a result, it will not have an impact on the `computer.personal` component implementation.
- When `FailedVisible` propagations (declared as **out** propagations in error models associated with components connected to the sensor ports) come simultaneously through both sensor ports, a `FailedVisible` propagation (declared as an in propagation in the error model associated with the `computer.personal` component implementation itself) occurs in the error model of the impacted component `computer.personal`.

Observations

- The Guard_In property can refer both to states and **out** propagations in its logic expression. In Table 21 (right hand side of the **when** clause, after the **or** operator), ErrorFree is a state while FailedVisible is an **out** propagation.
- The Guard_In property may reference an error state of a component that can impact another component. This flexibility permits the specification of error propagations between dependent components, when the error model of the origin component does not have outgoing propagations specified for error states. Notice that this action is equivalent to specifying a transition from an error state to itself with an **out** propagation label with an occurrence probability of 1 and naming the propagation in the Guard_In expression.
- The Guard_In property can include an error state of the impacted component in the logic expression, allowing the definition of conditions (under which propagations affect a component) that are dependent on the error state of the impacted component. For example, a component may respond to propagations from other components only if the impacted component is in the Transient_Error state.

5.4.5 Connection-Specific Filtering

Different features of a component may have distinct Guard_In properties through the declaration of Guard_In properties with an **applies to** clause to name the specific data access features, ports (data, event, or event data), and server subprogram features of a component to which the Guard_In property applies. The Guard_In property of a given feature is evaluated only for error propagations that occur into a component through that feature (e.g., through an **in** data port or a shared data object). It is forbidden to declare several Guard_In properties for the same feature. This prohibition ensures that only one Guard_In property is evaluated and that logic rules applying to a same feature do not conflict.

Observations

- Distinct Guard_In properties for different component features allow different name mappings to be specified for different incoming propagations.
- If distinct Guard_In properties that specify filter or masking conditions are specified for different features of a component, the same feature cannot be referred to in the different Guard_In properties.

5.5 FILTERING OF OUTGOING PROPAGATIONS

Error propagations out of a component can be filtered by using a Guard_Out property. The following subsections show the role of this property and how it is used.

5.5.1 Role of a Guard_Out property

A Guard_Out property allows you to specify pass-through of error propagations under the following conditions:

- unconditionally pass incoming propagations from different senders as outgoing propagations of the error model associated with the component whose implementation contains the Guard_Out property

- conditionally pass incoming propagations or error states as outgoing propagations of the error model associated with the component whose implementation contains the Guard_Out property
- conditionally mask incoming propagations from different senders

An incoming propagation is identified by naming an incoming (**in** or **in out**) port and an outgoing (**out** or **in out**) propagation of the connection or connected component.

5.5.2 Guard_Out Property Application

A Guard_Out property may be declared to apply to **provides** data access features, outgoing ports (data, event, or event data), and server subprogram features of a component. The Guard_Out property, consequently, is evaluated when error propagations occur into a component through the features specified in the guard condition (e.g., through an **in** data port or a data access feature or when there is an error state change in the error model associated with an origin error model). The Guard_Out property determines under what conditions the outgoing propagation occurs or is masked. Each component feature of the above-mentioned kinds can only have one Guard_Out property.

A Guard_Out property has a similar structure to a Guard_In property. A Guard_Out property consists of an ordered set of rules for outgoing propagations. Each rule maps error states of the component itself and outgoing (**out** or **in out**) propagations or error states from error models of connections or connected components (**when** clause) through an incoming (**in** or **in out**) port to an outgoing (**out** or **in out**) propagation; or, it specifies that they should be masked. The functioning of a Guard_Out property is illustrated in Figure 8, using the same symbols as in Figure 7 (on page 29).

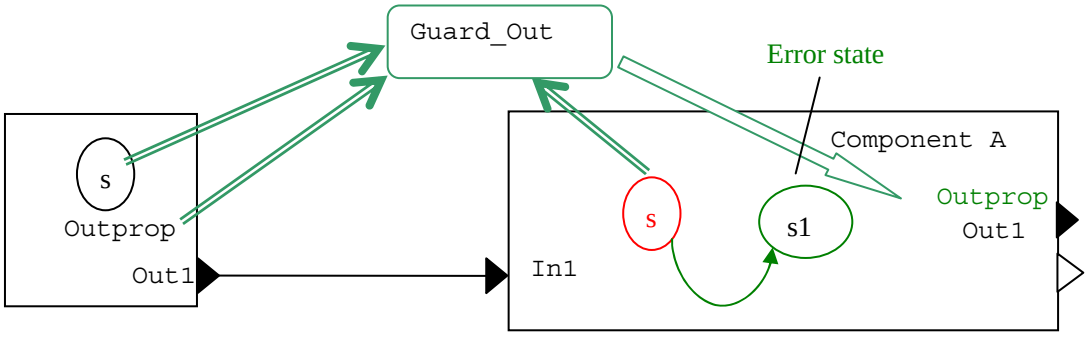


Figure 8: Guard_Out Mapping

In Figure 8, a Guard_Out property is defined for the outgoing propagation Outprop through port Out1 of ComponentA. This propagation is conditional on error states of ComponentA and on the error state or outgoing propagation Outprop of the component connected to ComponentA.

In Table 22, the out_propagation_*x* expression refers to a name of an **out** or **in out** propagation declared in the error model of the component for which the rule is being specified. The **when** clause represents the guard expression to be evaluated to determine whether the guard rule applies.

Table 22: Guard_Out Property in Use

```
Guard_Out =>
  out_propagation_1 when port_name_1[out_propagation_1],
  out_propagation_2 when port_name_2[error_state_1]
                     and port_name_3[error_state_2],
  mask when port_name_2[error_state_1] or
            port_name_3[error_state_2]
  applies to feature_Name_1;
```

The **when** clause can

- provide a single reference to an incoming (**in** or **in out**) component port, **requires** data access, or server subprogram feature and an outgoing propagation or error state in the error model of the connection through the feature
  - In this case, the **when** clause represents an unconditional name mapping or masking of the incoming propagation.
  - The outgoing propagation(s) or error state(s) are specified in square brackets, as shown in Table 22. If a reference does not include a bracketed outgoing propagation or error state, the initial error state is referred to implicitly. If the connection does not have an error model, the outgoing propagation or error state in the error model of the connected component(s) is referenced.
- specify a condition that refers to multiple component features and outgoing propagations or error states in the error model of the connection through the features
  - If the connection does not have an error model, the outgoing propagation or error state in the error model of the connected component(s) is referenced. This condition describes the circumstances under which incoming propagations and error states are passed through (by the component error model) or masked.
- include references to error states in the component for which the `Guard_Out` mapping is defined by including references to an error state of the component's error model
  - This action allows pass-through or masking of incoming error propagations to occur conditional on the error state of the component.
- contain the following logic operators: **not**, **and**, **or**, **ormore**, **orless**
  - The **not** operator has the highest precedence, followed by the **and** operator. The other operators have equal precedence and are evaluated from left to right except where parentheses specify otherwise. A numeric literal in an **ormore** or **orless** operator must be a positive integer.
- specify **others**
  - The specification of **others** means that the rule applies if none of the previous rules apply.

The `Guard_Out` expressions are evaluated each time an error is propagated into a component through the feature for which the guard is specified. They are evaluated in the order of declaration until the first one is found whose Boolean error expression evaluates to TRUE. If a rule with `out_propagation` before the **when** clauses evaluates to TRUE, the outgoing propagation is considered to occur and is propagated to connections or dependent components of the feature to

which the Guard_Out applies. If a **mask** rule evaluates to TRUE, the propagation is suppressed and does not result in a transition between states for the receiving error model. If none of the guard rules evaluates to TRUE, and there is no **others** clause, the specification is erroneous.

Observations

- The evaluation rules only require guards be defined just for those combinations of states and propagations that might occur for a specified system in the operational scenario being used for analysis.
- The current AADL error model annex specification allows outgoing propagation filtering to be associated with component dependencies through port connections, shared access, and server subprogram calls. However, it does not support filtering or masking of error propagations through bindings to execution platform components.

5.5.3 Error Propagation Pass-Through Mappings

The Guard_Out property can specify mappings of incoming propagations or error states from other components to outgoing propagations of the error model of components. This pass-through can be specified unconditionally or dependent on the error state of the component with the Guard-Out property.

Table 23 shows several examples of pass-through mappings. In these examples, it is assumed that

- the component connected through the port Sensor1 has the error model TransientPermanent.general
- the component connected through the port Sensor2 has an error model associated with two outgoing propagations named Error1 and Error2

The first Guard_Out property (see ① in Table 23) defines a pass-through mapping of an error state in a component error model into an outgoing error propagation of the error model. This guard is specified to only generate an outgoing propagation through port Output1.

The second Guard_Out property (see ② in Table 23) defines a mapping of several outgoing propagations in an error model (associated with a component that can affect the given component) into an outgoing error propagation of the error model of the affected component. This guard is specified to only generate an outgoing propagation through port Output2.

Table 23: Error Propagation Pass-Through Mappings

```
system computer
features
Sensor1: in data port;
Output1: out data port;
Sensor2: in data port;
Output2: out data port;
end computer;

system implementation computer.personal
annex Error_Model {**
 Model => My_ErrorModels::TransientPermanent.general;
 Guard_Out =>                                          ①
  FailedVisible when Sensor1[Permanent_Error]
   applies to Output1;
Guard_Out =>                                           ②
  FailedVisible when Sensor2[Error1,Error2]
      and self[Failed]
   applies to Output2;
   **};
end computer.personal;
```

Observations

Error propagation pass-through mappings can be used for several purposes:

- A component can observe the error states of another component or incoming connection and map them into an outgoing error propagation.
- A component can pass through error propagations from other components by mapping them into one of its own outgoing error propagations.
- The error propagations from other components can be passed on through one, several, or all outgoing features of the component. In other words, the component can route its observation of error propagations or error states from other components as one of its own outgoing error propagations.
- The error propagations from other components can be passed through when the component is in certain error states. In other words, pass-through can be conditional on the error state of the component performing the pass-through.
- The conditions for passing through incoming propagations may be different from the conditions under which the component itself handles incoming propagations. The latter conditions are captured in a `Guard_In`, whose in propagation is then named in a error state transition.
- If the condition for handling incoming propagations by the component and for passing the incoming propagations through is the same, it is better to trigger the outgoing propagation from a component error state instead of repeating the condition in a `Guard_In` and the `Guard_Out` property.

5.5.4 Pass-Through Filtering and Masking

The Guard_Out property can specify filters for error propagations from other components that reflect voting protocols used to determine whether the observed incoming propagations and error states should be visible to others. Using the same logic expressions, the Guard_Out property can also specify the conditions under which error propagations and error states are masked. The logic expression can name outgoing propagations and error states of the components that can affect the given component. In addition, the logic expression can include conditions that reflect the current error state of the affected component.

Figure 9 shows an example using the graphical AADL notation of architecture with a Guard_Out property on an **out** port. The out propagations sent through the SensorFailed **out** port of the computer.personal system implementation depend on **in** propagations arriving as inputs on the Sensor1 and Sensor2 **in** ports. The components connected to Sensor1 and Sensor2 are assumed to have error models dependent.general.

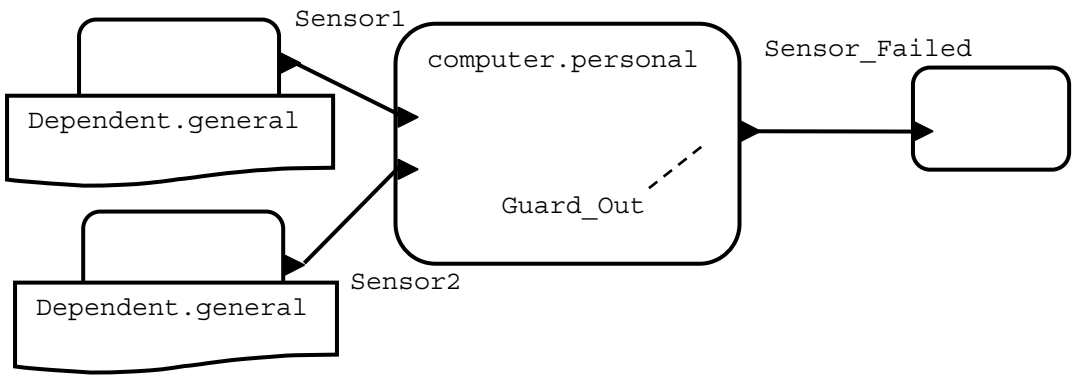


Figure 9: AADL Architecture with Guard_Out Property

Table 24 shows the AADL system component computer.personal (with its two **in** ports, Sensor1 and Sensor2, and one **out** port, SensorFailed). We associate the error model dependent.general (shown in Table 7 on page 13) with the component implementation. We declare a Guard_Out property that applies to the SensorFailed **out** port. The Guard_Out property describes the following propagation control.

- When a FailedVisible propagation (declared as an **out** propagation in the error model associated with a component connected to Sensor1 or Sensor2) comes through only one of the sensor ports, that propagation is masked. As a result, the masked propagation will not affect the computer.personal component implementation.
- When FailedVisible propagations (declared as **out** propagations in error model associated with components connected to Sensor1 or Sensor2 and as **in** propagations in the error model instance associated with the computer.personal component implementation itself) come simultaneously through both sensor ports, a FailedVisible propagation (declared as an **out** propagation in the error model associated with the computer.personal component implementation itself) is sent out through the **out** port SensorFailed.

Table 24: Guard_Out Example

```
system computer
features
Sensor1: in data port;
Sensor2: in data port;
SensorFailed: out data port;
end computer;

system implementation computer.personal
annex Error_Model {**
 Model => My_ErrorModels::dependent.general;
 Guard_Out =>
 mask when (Sensor1[FailedVisible] and Sensor2[ErrorFree])
   or (Sensor1[ErrorFree] and Sensor2[FailedVisible]),
 FailedVisible when (Sensor1[FailedVisible] and
Sensor2[FailedVisible])
   applies to SensorFailed;
   **};
end computer.personal;
```

Observations

- The `Guard_Out` property can refer both to states and **`out`** propagations. In Table 24, `ErrorFree` is a state while `FailedVisible` is an **`out`** propagation. Reference to error states allows us to observe component error models without outgoing propagations and conditionally produce outgoing propagations.
- The `Guard_Out` property can include an error state of the affected component in the logic expression, which allows us to define conditions under which propagations dependent on the error state of the impacted component actually affect a component. For example, a component may respond only to propagations from other components if the affected component is in the `Transient_Error` state.
- Distinct `Guard_Out` properties can be associated with different features of a component. Consequently, propagations depending on the same inputs that are sent out through different features can be different.

5.6 ERROR STATE PROPAGATION

In some dependability analysis scenarios, it is natural to think in terms of propagation of error states. Propagation can be achieved in two ways:

1. representation of guard conditions in terms of error states only
2. interpretation of error propagations as representing inferred error states

We will examine each modeling approach in detail.

5.6.1 Use of Error States in Conditions

You can restrict the conditions of `Guard_In` properties to refer only to error states of connection or connected component error models and the component's own error states. In other words,

`Guard_In` conditions do not refer to outgoing propagations of connections or connected components.

Similarly, you can restrict the `Guard_Out` conditions to refer only to error states of connection or connected component error models and the component's own error states. However, since out propagations are not named in `Guard_In` conditions, we have to reflect those `Guard_Out` conditions in a different way. `Guard_Out` conditions based on the component error states themselves become part of the `Guard_In` condition of the recipient component. `Guard_Out` conditions based on the error states of incoming connections or connected components can be mapped into a new `Guard_In` condition of the component with the `Guard_Out` property. This new **`in`** propagation results in an error state transition, and the error state being entered is observed by the `Guard_In` condition of the recipient component.

In summary, this modeling approach leads to error models that utilize error events, error states, and **`in`** error propagations but not **`out`** error propagations. It assumes that the error state of a component is immediately observable by other components.

5.6.2 Use of Inferred Error States

The Error Model Annex standard defines inferred error states as follows:

> *For each **out** or **in out** propagation defined in an error model, the inferred error states for that propagation are the set of error states named as a source state for any transition labeled with that propagation. Note that there may be more than one inferred error state for an error propagation name* [SAE-AS5506/1 2006].

This definition means that observing a propagation from another component or a connection can be interpreted as observing the error state that is the source of the transition triggering the propagation.

Figure 2 in on page 15 illustrates a patterned way of propagating an error state through an error propagation named in a transition from the error state to itself. In that case, naming the error propagation in a condition is equivalent to naming the error state with the loopback transition if the `Occurrence` probability of the propagation is 1.

`Occurrence` value less than one allow the model to reflect the fact that another component may observe the error state of a component with some delay or may not always observe the error state. The latter scenario is typically represented by an error state that is entered according to some distribution. The error state then has two outgoing transitions, one with fixed probability `p` and the second with fixed probability `1-p`. One is labeled with an **`out`** propagation, reflecting that an error is observed with probability `p`, while the other reflects that the error state remains unobserved.

Figure 10 illustrates the propagation of an observed fault. Notice that the **`out`** propagation is triggered by a transition from one error state to another error state. According to the definition of the Error Model Annex standard, the inferred error state is the error state from which the transition originates.

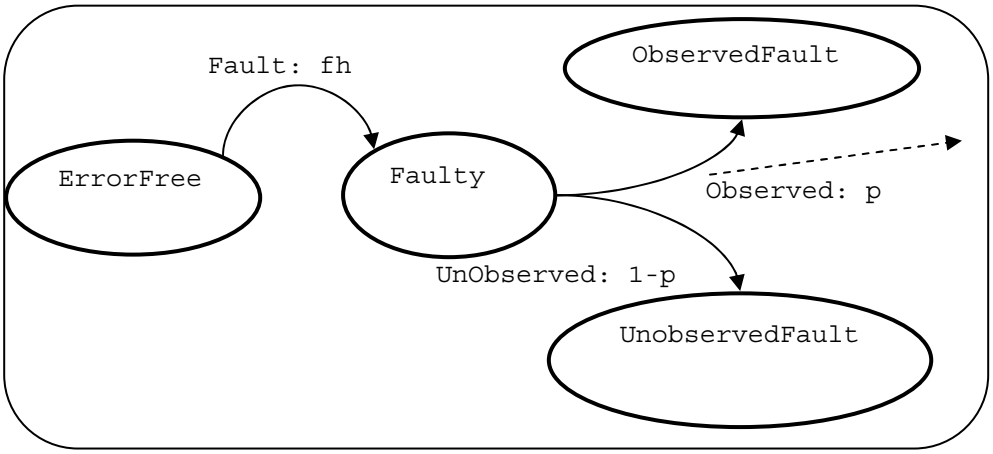


Figure 10: Observed Fault

In summary, the inferred error state approach utilizes error events, error states, **`in`** error propagations, and **`out`** error propagations. This approach allows users to define outgoing and pass-through error propagation conditions separately from incoming error propagation conditions. Furthermore, it can capture delayed and sporadic observation of faults in other components.

5.7 COMPARISON BETWEEN GUARD_IN AND GUARD_OUT

Sections 5.4 and 5.5 explained how to use the propagation control properties `Guard_In` and `Guard_Out`. Table 25 shows the similarities and differences of these two properties from a functional view and an input/output view.

You might use a `Guard_In` if the decision-making (i.e., voting, filtering) layer is placed at the input interface of a component. A `Guard_Out` might be more appropriate if the decision-making functionality exists as a component in its own right.

Table 25: Symmetry and Asymmetry between Guard_In and Guard_Out

	Guard_In	Guard_Out
Functional View	Applies to incoming features Its evaluation result has an impact on the internal behavior of the component that declares it.	Applies to outgoing features Its evaluation has an impact on components that depend on the component that declares it.
Input/Output View	**Input**: propagations from the component's environment **Output**: propagations to the component itself	**Input**: propagations from the component's environment **Output**: propagations to the component's environment

6 System Instance Error Models

An AADL architecture model is hierarchical. The level of detail it contains depends on the stage of the design and development process. For example, a system may initially be modeled as a partial model to the level of subsystems and later completed to the level of threads. Both partial and complete models can be instantiated to produce system instance models for system analysis.

Error models can be associated with components of a system model at any level of the component hierarchy. For example, an error model can be associated with the root-level system component to represent an abstracted error model of the system instance. An instance of this error model represents the system instance error model as a finite state stochastic automaton. Similarly, error models can be associated with each of the leaf components in the system hierarchy (i.e., individual application threads and hardware components). In this case, the system instance error model consists of the set of component error model instances and connection error model instances. The system instance error model represents a set of concurrent stochastic automata.

Error models can be associated with several levels of the system hierarchy at the same time. For example, an error model can be associated with an application thread, an enclosing application (sub)system, and the system as a whole. In this case, the error model higher in the system hierarchy is an abstraction of the contained error models. The AADL Error Model Annex standard offers two approaches for representing error model abstractions:

1. A basic error model represents the behavior of a component and its subcomponents in the presence of faults as an abstraction. (The error models presented in Sections 4 and 5 are basic error models.)
2. A derived error model represents the behavior of a component in the presence of faults as a function of the error states of its subcomponents.

Sections 6.1 and 6.2 discuss these two approaches.

6.1 ABSTRACTION WITH BASIC ERROR MODELS

A basic error model associated with a component or connection consists of an error model type and implementation identified by the `Model` property, any component-specific tailoring with the `Occurrence` property, and component-specific error propagation filtering and masking defined through `Guard_In` and `Guard_Out` properties.

A basic error model describes the behavior of a component in terms of error events intrinsic to the component, error states, error propagations from and to components this component interacts with, and error state transitions that are triggered by intrinsic error events or incoming error propagations and initiate outgoing error propagations. The behavior of a component in the presence of faults is defined without referring to any subcomponent. Consequently, a basic error model repre-

sents the behavior of a component or connection in the presence of faults as an abstraction independent of subcomponent error models.

6.1.1 When to Use Basic Error Models

You can use a basic error model as an abstraction if an existing AADL architecture model with associated error models for components and connections is very detailed and requires a high level of processing. In those instances, a tradeoff between the accuracy and the complexity of the model is necessary, and some of the model's details must be ignored to produce a lower-fidelity error model. The lower-fidelity error model can be achieved by associating a basic error model to a component containing subcomponents, so that the error models associated to the subcomponents are ignored by the analysis. Besides states, events, and transitions, the basic error model of a component includes **`in`** and **`out`** propagations that abstractly describe the actual error propagations between contained subcomponents and external component. Issues about the relationship between abstracted error models and higher fidelity error models are discussed in the work by Binns and Vestal [Binns 2004].

6.1.2 How to Use Basic Error Models

If both a component and its subcomponents have associated error models, you can use the `Model_Hierarchy` property to indicate whether the enclosing basic error model should be considered as the abstraction and the error models associated with subcomponents should be ignored by the analysis. This property allows you to annotate an AADL system model with error model information at different levels of the system hierarchy and choose the level of fidelity for analysis. Table 26 shows how to use `the Model_Hierarchy` property to specify that an error model is abstract.

In Table 26, the system implementation `computer.personal` contains two subcomponents: `hardware.nominal` and `software.nominal`. Error models are associated with each of the subcomponents. An error model is also associated to `computer.personal` itself. It is declared as **`abstract`**. Thus, the analyses will ignore the error models declared for subcomponents. In other words, an abstract error model at a higher level of the component hierarchy abstracts the details of the subcomponent error models away.

Note that the `Model_Hierarchy` property declaration is not mandatory, because it is implied by the association of a basic error model. Consequently, you have to add or remove error models and guard declarations in parents to change the fidelity at which models are analyzed.

Table 26: Abstract Error Model Specified Using Model_Hierarchy

```
system implementation hardware.nominal
annex Error_Model {**
   Model => forHardware.general;
   **};
end hardware.nominal;

system implementation software.nominal
annex Error_Model {**
   Model => forSoftware.general;
   **};
end software.nominal;

system implementation computer.personal
subcomponents
   HW: system hardware.nominal;
   SW: system software.nominal;
annex Error_Model {**
   Model => dependent.general;
   Model_Hierarchy => abstract; **};
end computer.personal;
```

6.2 DERIVED ERROR MODELS

A derived error model is a component error model whose error state is determined by a derived state mapping of subcomponent error states and error states and connected components' error propagations.

A derived error state mapping, described in Table 27, specifies the condition under which a component is in its error state in terms of

- error states of subcomponents
- error states or error propagations of connections or connected components

Table 27: Derived State Mapping Structure

```
Derived_State_Mapping =>
      ErrorFree when HW[ErrorFree] and SW[ErrorFree],
      Failed when others;
```

A derived error state mapping is illustrated in Figure 11, using the same symbols as in Figure 7 on page 29. The dashed double-line arrow indicates that subcomponent error states are inferred when an outgoing error propagation of a subcomponent is named. The inference rules are described in Section 6.2.2. A derived error model does not have error events or explicitly declared error state transitions.

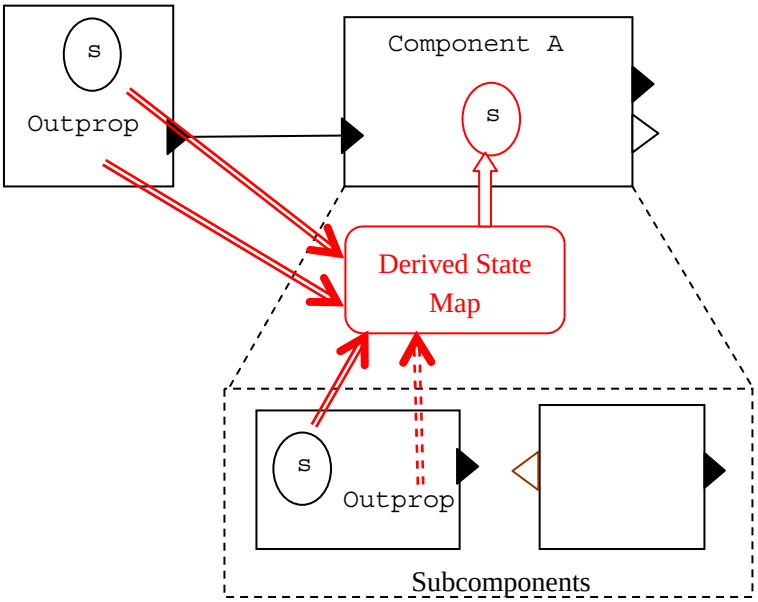


Figure 11: Derived Error State Mapping

6.2.1 When to Use Derived Error Models

You can use derived error models to express a high-level view of subcomponent error behavior through a (possibly) smaller set of error states and outgoing propagations. This expression is obtained through partitions that consist of sets of subcomponent error states that represent

- nominal behavior
- faulty states
- catastrophic states

This kind of partitioning is used in dependability evaluation tools. The `report` property can be used to identify the error state partitions of interest.

Note that in probabilistic dependability analyses (like those based on Markov chains and Petri nets), it is not mandatory to describe the system in terms of derived state-mapping expressions. These analyses allow deriving the global states from compositions of error states of subcomponents.

6.2.2 How to Use Derived Error Models

A derived component error model is declared through three error model properties:

1. a `Model` property that identifies an error model type
2. a `Derived_State_Mapping` property that defines the states of the derived error model as a function of
   a. subcomponent error states and outgoing propagations
   b. error states and outgoing propagations of connections

c. error states and outgoing propagations of connected components of a named incoming component feature

3. a `Model_Hierarchy` property with the value `Derived`

When a component error model is declared as derived, the `Model` property can refer to the error model type or error model implementation. If it refers to an error model implementation, only the error model type is considered. In the error model type, only error state and outgoing error propagations are considered to be part of a derived error model. The error states of the component are determined by evaluating the derived state-mapping logic expression.

A derived state mapping is declared using a `Derived_State_Mapping` property that contains a set of derived state mapping rules. If a component has a `Model_Hierarchy` property association of `derived`, a `Derived_State_Mapping` property must be declared for that component. The `Derived_State_Mapping` property is ignored if the `Model_Hierarchy` property does not have the value `derived`.

A `Derived_State_Mapping` property consists of several state-mapping rules, one for each error state of the component for which the `Derived_State_Mapping` is defined.

Table 28 shows how to use the `Derived_State_Mapping` property. The system implementation `computer.personal` contains two subcomponents: `hardware.nominal` and `software.nominal`. (Note: This architecture is the same as in Table 26 on page 44.) Error models are associated to subcomponents. The implementation `computer.personal` declares a `Derived_State_Mapping` property inside its error **`annex`** clause. This property specifies that the system is `ErrorFree` when both subcomponents are `ErrorFree` and `Failed` otherwise.

In the **`when`** clause of each state-mapping rule, names of subcomponents are followed by optional bracketed lists of error states or error propagations of the error models associated with the subcomponents. If no bracketed list follows a subcomponent name, the initial state of the subcomponent is inferred. For a named error propagation, the error state is inferred to be the source error state for error state transitions that refer to it.

The name of a port, data access feature, or server subprogram feature is also permitted in the **`when`** clause. There, the name refers to the error model associated with the connection made to that component feature. If there is no associated error model for a connection, this naming rule applies to the error model associated with the component that is the source of that connection.

The **`when`** clause can contain the following logic operators: **`not`**, **`and`**, **`or`**, **`ormore`**, **`orless`**. The **`not`** operator has the highest precedence, followed by the **`and`** operator. The other operators have equal precedence and are evaluated from left to right except where parentheses specify otherwise. A numeric literal appearing in an **`ormore`** or **`orless`** operator must be a positive integer.

Table 28: Derived State Mapping Property

```
system implementation hardware.nominal
annex Error_Model {**
   Model => forHardware.general;
   **};
end hardware.nominal;

system implementation software.nominal
annex Error_Model {**
   Model => forSoftware.general;
   **};
end software.nominal;

system implementation computer.personal
subcomponents
   HW: system hardware.nominal;
   SW: system software.nominal;
annex Error_Model {**
   Model => dependent.general;
   Model_Hierarchy => derived;
   Derived_State_Mapping =>
      ErrorFree when
         (HW[ErrorFree] and SW[ErrorFree]),
      Failed when others;
   **};
end computer.personal;
```

The **when** clause can contain the keyword **others** only in the final state-mapping rule. If no preceding rule is evaluated to TRUE, the final **others** rule (with **others**) is used to determine the current error state of the component.

The state-mapping rules are evaluated in the order of declaration until the first **when** clause evaluates to TRUE. The error state in that rule becomes the current state of the component. If no rules evaluate to TRUE and there is no **others** clause, the specification is erroneous.

Among the state-mapping expressions that define states of a set of components there must be no circular references. A circular reference could cause an error in which a subcomponent is shared by systems at different levels of the nesting hierarchy or in which two components communicate with each other and their states are defined by derived state mappings that name the features connecting the two components.

Observations

- The mechanisms of abstraction and derivation for hierarchical error models are convenient for enhancing the readability of the model. Where the model is too big to be processed, it may be necessary to abstract away error modeling details. Sometimes it may be convenient to use derived state-mapping expressions to make global states of the system visible. However, it is not mandatory to use these mechanisms in any model.
- It is worth noting that, unlike basic error models, derived error models cannot have component-specific `Occurrence` properties for their outgoing propagations. The occurrence probability of a derived outgoing propagation is determined by the error models that the propagation is derived from.

7 Operational Modes and Error States

Systems can be in various operational modes. An operational mode might represent a mission-phased mode of operation—such as takeoff, cruise, or landing in an avionics system. An operational mode might reflect a particular fault-tolerant system configuration, such as operating the primary or backup variant of a dual redundant system. An individual system component might have multiple levels of performance, such as algorithms with different levels of precision.

System operational modes are visible execution states of the embedded software system. They can be modeled by AADL modes. System components may encounter failures that cause them to go from an error-free state to an error state. These error states are logical states of a system component that may propagate to other system components. Component faults, error states, and error propagation can be by modeled by AADL error models. Both modes and error states, then, represent states of a system.

The difference between modes and error states lies primarily in their semantics. Error states result from occurrences of error events (e.g., faults or repair events), while modes represent operational states that may be totally independent of error events. Error states are not necessarily observable, but modes are always observable. For example, an error state might represent a component that is in an unobserved erroneous state because it has not yet failed at its interface.

Although they are different, modes in operational systems and error states in error models can affect each other. The embedded software system can observe component error states and might change to a different fault tolerant configuration in response. Consequently, a logical system state (the error state) is translated into an operational system mode. Similarly, changes in the operational system mode might affect logical error states (e.g., a repair action in the embedded software might cause a component to re-enter an error-free state).

In this section, we discuss the interaction between operational system modes, represented by AADL modes, and logical error states, represented by error models.

- In Section 7.1, we discuss the use of AADL modes and mode transitions to represent the dynamics of operational modes.
- In Section 7.2, we describe how changes in the error model of a system can be translated into events that the operational system can respond to.
- In Section 7.3, we discuss how the logic behind mode transitions, in particular logic that addresses fault tolerance, can be expressed in AADL.
- In Section 7.4, we present mechanisms that allow error models to reflect changes in operational states (i.e., respond to mode changes).
- In Section 7.5, we give some modeling examples to illustrate the interaction between modes and error states in actual systems.

7.1 MODELING OF OPERATIONAL MODES

Actual systems can be represented according to one of these scenarios:

1. Operational modes in phased-mission systems model configurations representative of different phases in a mission. For example, in the case of an aircraft model, one may distinguish between the takeoff, cruise, and landing phases. During each of these phases, the system would have a particular configuration with active components and connections. The fault management approach may be different during each phase.
2. Fault-tolerance modes model configurations due to the fault-tolerance strategy chosen for the system or for particular parts of the system. For example, a fault-tolerant duplex system may have two operational modes corresponding to (a) replica no. 1 delivers the service, replica no. 2 monitors replica no.1; and (b) replica no. 2 delivers the service, replica no. 1 monitors replica no.2.

7.1.1 Modes, Mode Transitions, and Events

A mode is a visible operational state of an AADL component. A component can have mode-specific properties and configurations of subcomponents and connections that are active in specific modes. Although components and connections can be part of more than one mode, they are in only one mode—the current mode—at any one time. Mode transitions represent dynamic operational behavior (i.e., switching between configurations and changes to system characteristics expressed through property values).

Modes and mode transitions can be specified for a component anywhere in the system hierarchy. Components with modes may contain subcomponents with modes. The AADL defines a **system operation mode** to be the set of current mode states of all modal components in the system.

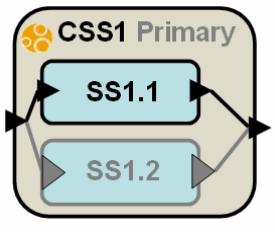
Mode transitions are triggered by events from outgoing (`out` or `in out`) event ports of subcomponents, by events from incoming (`in` or `in out`) event ports of the component with the mode transition, or by events raised local to the component. The state machine formed by modes and mode transitions in a component implementation is deterministic (i.e., from a given mode, only one mode transition is triggered at a time).

7.1.2 Application of Modes and Events

Mission-phased operational modes may apply to the whole embedded system or they may apply to a particular subsystem. This scope of applicability determines which components in the system hierarchy should be declared with modes. For each mode, this system component can be configured with different subsets of active subcomponents and connections and with variants of the same subcomponent. The subcomponents themselves may be modal; their mode selection can be represented either by different implementation variants or by mode declarations. Their selection is driven by events that originate from a system component that acts as operational mode manager.

Usually, phased-mission systems also need modes to represent fault-tolerance mechanisms. In AADL, a mode cannot have nested modes itself, but a component with modes can also have subcomponents that have modes. If the fault tolerant mode is reflected in a redundancy pattern [Feiler 2004], the application of this redundancy pattern to a system component with mission-phased op-

erational modes results in nested components. The redundancy component contains the fault tolerance modes, while the redundant application component contains the operational modes. An example of a redundancy pattern is shown in Figure 12. Any events that are passed into the application component to control its operational mode must be routed through the redundancy pattern to the redundant copies.




```
system implementation PrimaryBackupPattern.impl
 subcomponents
  primary: system sys in modes (Primarymode);
  backup: system sys in modes (Backupmode);
 connections
  inprimary: data port insignal ->
primary.insignal in modes (Primarymode);
  inbackup: data port insignal -> backup.insignal
in modes (Backupmode);
  outprimary: data port primary.outsignal ->
outsignal in modes (Primarymode);
  outbackup: data port backup.outsignal ->
outsignal in modes (Backupmode);
 modes
  Primarymode: initial mode;
  Backupmode: mode;
 end PrimaryBackupPattern.impl;
```

Figure 12: Dual Redundancy Pattern

Notice that events going through a named port cannot be distinguished because they do not have names or other identifying data. Thus, any event going through a port that is specified in a mode transition will trigger that mode transition. By using separate event ports, you can specify that different events can trigger different mode transitions.

If a mode transition lists multiple event ports, an event through any of the ports can trigger the transition. For mode transitions that name multiple event ports as their trigger condition, logic represented by the **or** operator is assumed. For dependability analyses, it is desirable to specify transition logic other than **or** logic. We will show in Section 7.2.1 how mode transition logic can be described through the `Guard_Transition` property of the error model.

7.2 GENERATION OF SYSTEM EVENTS

The AADL allows us to model logical error states separately from the operational mode of the running application. It also establishes a connection between the logical error states and the operational mode by using `Guard_Event` properties to translate logical error states into actions in the form of port events on the running system. Those events can initiate the dispatch of an aperiodic or sporadic thread and trigger a mode transition.

7.2.1 Role of a Guard_Event Property

The `Guard_Event` property links the error states and error propagations in the error model to port events in the AADL architecture model. It is intended to map error conditions into architectural events (i.e., to specify that an architectural event is raised depending on the error model log-

ic). In particular, this property can be used to specify that a "real" event is raised in the system according to error conditions detected by a voting protocol implemented in the component.

7.2.2 Guard_Event Property Application

A `Guard_Event` property is associated with an outgoing (**`out`** or **`in out`**) event port of the component that declares the property. The generated event will be sent out of the component through that port. A `Guard_Event` can also specify that an event is raised local to the component (e.g., an event that control mode transitions of the component itself.[3]) Such an event is not available outside the component, unless explicitly connected to an outgoing event port.

7.2.3 How to Use Guard_Event Properties

A `Guard_Event` property specifies that an event is generated through an outgoing event port. The outgoing event port can be named in a mode transition of the enclosing component, or the event can be routed to another component through an event connection. The ultimate destination of this event connection can be a thread event port that results in the dispatch of the thread, or it can be a mode transition if the event port is named in a mode transition. Consequently, a dispatch or mode transition will occur when a specific condition is detected in the system error model.

Table 29 shows how a `Guard_Event` property can be used. The example declares a system having two **`in`** data ports, `Sensor1` and `Sensor2`, and one **`out`** event port, `SensorsFailed`.

Table 29: Guard_Event Property

```
system computer
features
Sensor1: in data port;
Sensor2: in data port;
SensorsFailed: out event port;
end computer;

system implementation computer.personal
annex Error_Model {**
Model => My_ErrorModels::dependent.general;
Guard_Event =>
   Sensor1[FailedVisible] and Sensor2[FailedVisible]
   applies to SensorsFailed;
**};
end computer.personal;
```

The error model annex subclause associated with the system's implementation contains the error model `dependent.general` and a `Guard_Event` property declaration applying to the **`out event`** port `SensorsFailed`. The `Guard_Event` property specifies that an event will be raised and sent out of the component through the port `SensorsFailed` if error propagations `FailedVisible` arrive through `Sensor1` and `Sensor2`. The propagations referred to in the

[3] The ability to reference local events is addressed in errata to the AADL standard.

logic expression are declared as **in out** propagations in the type of the error model (dependent.general) of the system and named as **out** error propagation action in an error state transition or have a Guard_Out rule.

The event triggered by the Guard_Event property through the port SensorsFailed can be passed through an event port connection to other components. This event may then trigger a thread dispatch or mode transition, or it may be observed as an "alarm" event in the vent port queue of a health monitoring thread. In other words, the observation of FailedVisible propagations arriving at the computer system may cause thread dispatches or mode transitions, or those propagations may become an observable events in the application itself.

Notice that the AADL architecture model in Table 29 is the same as the one shown in Figure 9 on page 38 for Guard_Out properties. The difference between Guard_Out and Guard_Event properties lies in the type of event generated. In the case of the Guard_Out property, an (error) propagation is generated; for the Guard_Event property, a port event is generated.

A separate Guard_Event property must be defined for each event port and each event local to the component. The **applies to** clause identifies the event port by name and a local event by **self**.eventname. Note that local events do not have to be explicitly declared; they are just named in mode transitions. Local events can be connected to outgoing event ports by an event port connection whose source is **self**.eventname, making them externally visible.

The condition under which a port event occurs is specified by a logic expression that can refer to

- component features and outgoing propagations or error states in the error model of the connection through the features
  - If the connection does not have an error mode, the outgoing propagation or error state in the error model of the connected component(s) is referenced.
- error states declared in the error model associated with the component whose implementation contains the Guard_Event property
  - The special keyword **self** is used in this case (i.e., **self**[ErrorFree]).

Error propagations or error states can be listed in a bracketed list following the component feature name or the keyword **self**. If no bracketed list is given, it is assumed that the initial state of the connected component is referred to.

The logic expression can contain the following logic operators: **not**, **and**, **or**, **ormore**, **orless**. The **not** operator has the highest precedence, followed by the **and** operator. The other operators have equal precedence and are evaluated from left to right except where parentheses specify otherwise. A numeric literal appearing in an **ormore** or **orless** operator must be a positive integer.

Whenever any error propagation into a component error model occurs, every Guard_Event property associated with any of its **out** event ports is evaluated. If the associated expression evaluates to TRUE, the port event is raised. Either zero or one port event will be raised for each error propagation, depending on the value of the Guard_Event property at the time the error is propagated. For the purposes of error modeling and analysis, the latency between the error propagation and the raising of any resulting events is zero. If more than one Guard_Event expression eva-

luates to TRUE when an error propagation occurs, there is no defined order in which the events are considered to occur or be processed.

If the predefined standard **out** event data port `Error` (defined in Section 5.3 of the core AADL standard[4]) for a thread has a connection declared with that port as its source and if the thread has an associated error model, a `Guard_Event` error property association must be specified for the predeclared error port.

If an event port is an ultimate source for an event connection and there is no `Guard_Event` property associated with that event port, it should be assumed that an event could be raised for that port in any state relevant to the modeling and analysis scenario.

7.3 MODE TRANSITION LOGIC

AADL mode transition declarations name one or more event ports whose events can trigger a mode transition. By default, an event on any of the named ports can trigger the mode transition. When modeling fault tolerance, it is desirable to model the specific conditions under which the mode transition occurs, such as the use of a voting protocol.

7.3.1 Role of a Guard_Transition Property

The `Guard_Transition` property specifies the conditions under which a mode transition occurs. This property can be used as an advanced decision-making mechanism that might model voting protocols affecting the mode configuration of a component.

The `Guard_Transition` property supports

- Specification of conditions other than the default or condition on port events arriving at the named event ports of a mode transition. We refer to this as **Event-Based Mode Transition Condition**.
- Specification of conditions in terms of error propagations and error states in the error model under which a mode transition is expected to occur. We refer to this as **Error-Based Mode Transition Condition**.

We discuss each of these two scenarios in turn.

[4] The SAE AADL standard is available from the SAE as document AS5506. It can be ordered using this Web address: http://www.sae.org/servlets/productDetail?PROD_TYP=STD&PROD_CD=AS5506.

7.3.2 How to Specify Event-Based Mode Transition Conditions

Table 30 shows how a Guard_Transition property can be used as mode transition condition on the port events named in a mode transition. The example declares a system computer having two **in** event ports, Sensor1 and Sensor2. The system's implementation declares modes, System_OK and Sensors_Fail, and a mode transition that names both event ports Sensor1 and Sensor2. The Guard_Transition specifies that if the system is in mode System_OK, it will move to mode Sensors_Fail if port events arrive through ports Sensor1 and Sensor2. This overrides the default mode transition condition of Sensor1 or Sensor2.

Table 30: Event-Based Mode Transition Condition

```
system computer
features
Sensor1, Sensor2: in event port;
end computer;

system implementation computer.personal
modes
System_OK: initial mode;
Sensors_Fail: mode;
BecomeFailed: System_OK-[Sensor1, Sensor2] -> Sensors_Fail;
annex Error_Model {**
Guard_Transition =>
   (Sensor1 and Sensor2) applies to BecomeFailed;
**};
end computer.personal;
```

The error model annex subclause associated with the system implementation contains a Guard_Transition property declaration for mode transition BecomeFailed. The Guard_Transition properties specify that the mode transition occurs only when port events arrive through ports Sensor1 and Sensor2.

Port events can have one of three sources.

1. generated using Guard_Event properties on the **out** event ports that are at the origin of connections to ports Sensor1 and Sensor2

   In this case, the port event represents a port event that reports an error model state of the originator.

2. originated by an application thread that raises a port event through a Raise_Event system call

   In this case, the port event may represent application logic or an exceptional condition that the thread wants to report.

3. originated by a processor to which an application thread is bound

   In this case, the thread may have decided to pass this event on to the receiving component.

A separate `Guard_Transition` property must be defined for each mode transition. The mode transition is identified by name[5] in the **`applies to`** clause. The condition under which a port event occurs is specified by a logic expression that can refer to

- outgoing subcomponent event ports
- incoming features of the component for which the `Guard_Transition` is declared

The optional outgoing propagations or error states, which are specified in square brackets after the event port name, are not used when specifying mode transition conditions.

The logic expression can contain the following logic operators: **`not`**, **`and`**, **`or`**, **`ormore`**, **`orless`**. The **`not`** operator has the highest precedence, followed by the **`and`** operator. The other operators have equal precedence and are evaluated from left to right except where parentheses specify otherwise. A numeric literal appearing in an **`ormore`** or **`orless`** operator must be a positive integer.

Each time a port event occurs that might cause a mode transition, the logic expression of a `Guard_Transition` property that names event ports is evaluated. If the expression evaluates to TRUE, any mode transition labeled with that event port will occur.

7.3.3 How to Specify Error-Based Mode Transition Conditions

This section illustrates the use of `Guard_Event` for specifying error-based mode transition conditions.

A `Guard_Transition` property may specify a condition in terms of error states and error propagations under which a mode transition is to occur. This may result in the `Guard_Transition` condition having the value TRUE, even if a port event named in the mode transition has not been raised. This inconsistency is referred to in the Error Model Annex standard as "false mode transition" [SAE-AS5506/1 2006, Annex E.3.4.3 (10)]. To avoid this inconsistency, we recommend that the error state and propagation-based condition for a mode transition be specified through a `Guard_Event`.

Table 31 shows how a `Guard_Event` property can be used to specify an error-based mode transition condition. The example declares a system `computer` having two **`in`** data ports, `Sensor1` and `Sensor2`. The system's implementation declares modes, `System_OK` and `Sensors_Fail`, and a mode transition that names both event ports `Sensor1` and `Sensor2`. The `Guard_Event` specifies that if the system is in mode `System_OK`, it will move to mode `Sensors_Fail` if `FailedVisible` is propagated through port `Sensor1` and `Sensor2` (i.e., both originating components are in the `Failed` error state and this can be observed by the recipient).

[5] Erratum to the AADL standard provides the ability to name mode transitions.

Table 31: Error-Based Mode Transition Condition

```
system computer
features
Sensor1, Sensor2: in data port;
end computer;

system implementation computer.personal
modes
System_OK: initial mode;
Sensors_Fail: mode;
System_OK-[self.BeFailed] -> Sensors_Fail;
annex Error_Model {**
Guard_Event =>
   (Sensor1[FailedVisible] and Sensor2[FailedVisible])
   applies to self.BeFailed;
**};
end computer.personal;
```

7.4 MODE TRANSITIONS AND ERROR MODELS

AADL components of a system can be either active or inactive in a particular mode. For certain analyses, it may be interesting to consider distinct behaviors in the presence of faults for inactive and active components. To this end, the AADL Error Model Annex standard allows you to declare (optionally) an initial inactive state, in addition to the initial state, in an error model type. This initial inactive state is the initial state of the component if the component is inactive in the initial mode of the system. If no initial inactive state is declared, the initial state is used even if the component is initially inactive.

When the system configuration changes (i.e., when the component is activated or deactivated), the error model characteristics can change. This mode change can be in the error model. An error state transition labeled **`activate`** occurs when the component is activated at a mode transition while an error state transition labeled **`deactivate`** occurs when the component is deactivated at a mode transition. If a component is activated or deactivated at a mode transition but no transitions from the current state are labeled respectively **`activate`** and **`deactivate`**, its state does not change.

Table 32 shows an error model that declares an initial inactive state. It is an error-free state, just like the initial (active) state. We make the assumption that an inactive component does not fail. A component that is error-free when activated at a mode transition moves to an active error-free state and may fail while the component is active. An active failed component may be repaired to regain its active error-free state. If a failed component is deactivated, it moves directly to an inactive error-free state.

Table 32: Activate and Deactivate State Transitions

```
error model Modal
features
ON_ErrorFree: initial error state;
OFF_ErrorFree: initial inactive error state;
Failed: error state;
Fail: error event {Occurrence => poisson lambda};
Repair: error event {Occurrence => poisson mu};
end Modal;

error model implementation Modal.example
transitions
ON_ErrorFree-[deactivate]->OFF_ErrorFree;
OFF_ErrorFree-[activate]->ON_ErrorFree;

ON_ErrorFree-[Fail]->Failed;
Failed-[out CorruptedData]->Failed;
Failed-[Repair]->ON_ErrorFree;
Failed-[deactivate]->OFF_ErrorFree;
end Modal.example;
```

7.5 AADL MODEL EXAMPLES FOR SYSTEMS WITH MODES

This section presents a series of example models for systems that have modes and failure behaviors. All of the models can be used to drive dependability analyses. They show how the fault-related information is added to different AADL architecture models and how this fault logic can be related to the fault management mechanisms used in the system architecture.

The examples represent dual redundant systems and self-managing systems. The redundant components provide the same service. This service may be responding to server subprogram calls or processing a data stream, event stream, or event data stream. Those systems may be complete systems or may be components of larger systems. Modes are used to model fault management through reconfiguration at runtime. Error models are used to capture the fault behavior. Section 7.5.1 presents a dual-redundant system in a cold standby setup where the components are able to detect faults themselves. Section 7.5.2 presents the same dual-redundant system in a hot standby setup. Section 7.5.3 presents a system that observes its own faults and takes corrective actions itself (i.e., a self-managing system). Section 7.5.4 presents a dual-redundant system with a monitoring component (e.g., a health monitor). Section 7.5.5 presents a dual-redundant system with mutually informing components, and Section 7.5.6 presents a dual-redundant system with mutually observing components.

7.5.1 Cold Standby of Self-Observing Components

In this scenario, the component is assumed to be self-observing (i.e., able to report its own error states as port events). The redundant instances of the component reside inside system components whose mode determines which component instance and connection is active at a given time. In case of the cold standby pattern, only one component is active in a given mode and only the connections to and from the active component are active in the same mode. The cold standby pattern is shown in Figure 13 for a component processing a data port stream. The active component and

connections are shown in black, while the inactive component and connections are shown in gray. *CompP* and *CompB* do not communicate one with the other. The large, round-cornered rectangle on the left shows the *Primary* mode as active mode, in which component *CompP* is active. The large, round-cornered rectangle on the right it shows the *Backup* mode with component *CompB* active.

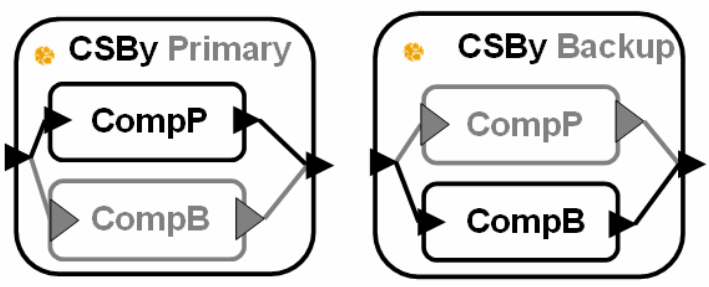


Figure 13: Cold Standby Pattern

Figure 14 shows the two modes (Primary and Backup) and an event port for each of the redundant components through which the component reports that it is in a failed state. The mode transition occurs when the active component reports a failure. In this model, it is assumed that the failure is recoverable (i.e., the component can perform its service when reactivated through a mode transition).

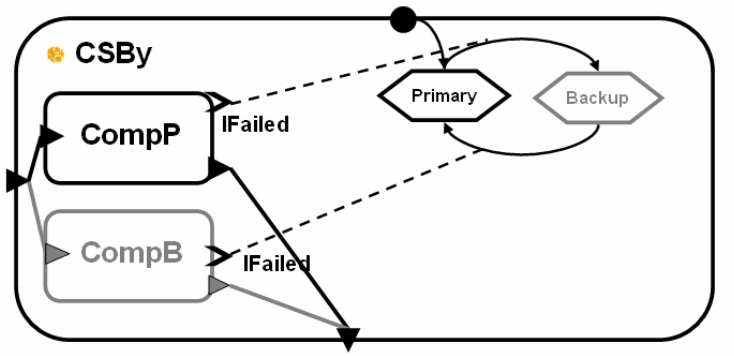


Figure 14: Cold Standby of a Self-Observing Component

Table 33 shows a part of the AADL architecture model (textual AADL) and the associated error model. A `Guard_Event` property is associated with the **`out`** event port involved in mode transitions. We use the `Modal.example` error model in this example; it takes into account the fact that components get deactivated and activated through mode transitions.

Table 33: Cold Standby of a Self-Observing Component

```
System computer
features
Input: in data port;
Output: out data port;
IFailed: out event port;
end computer;

system implementation computer.personal
annex Error_Model {**
   Model => Modal.example;
   Guard_Event => self[Failed] applies to IFailed;
**};
end computer.personal;

system CSBy
features
Input: in data port;
Output: our data port;
end CSBy;

system implementation CSBy.generic
subcomponents
CompP: system computer.personal in modes Primary;
CompB: system computer.personal in modes Backup;
connections
data port Input -> CompP.Input in modes Primary;
data port CompP.Output -> Output in modes Primary;
data port Input -> CompB.Input in modes Backup;
data port CompB.Output -> Output in modes Backup;
modes
Primary: initial mode;
Backup: mode;
Primary -[CompP.IFailed]-> Backup;
Backup -[CompB.IFailed]->Primary;
end CSBy.generic;
```

7.5.2 Hot Standby of Self-Observing Components

In this scenario the component is assumed to be self–observing (i.e., able to report its own error states as port events). The redundant instances of the component reside inside system components whose mode determines which component instance and connection is active at a given time. In case of the hot standby pattern, both components are active, but only one component's output is made available as output of the redundant system. The hot standby pattern is shown in Figure 15 for a component processing a data port stream. The active component and connections are shown in black, while the inactive connections are shown in gray. *CompP* and *CompB* do not communicate with each other. In the large, round-cornered rectangle on the left, the *Primary* mode is ac-

tive, in which component *CompP* output is sent out through an active connection, while the connection from *CompB* is inactive. In the large, round-cornered rectangle on the right, the *Backup* mode is active with component *CompB* output made available through an active connection, while the connection from *CompP* is inactive.

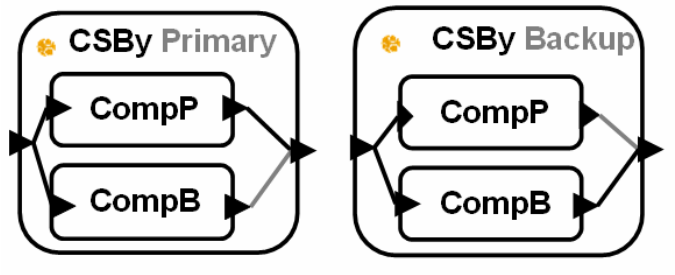


Figure 15: Hot Standby Pattern

Figure 16 shows the two modes (Primary and Backup) and two event ports for each of the redundant components. The component reports that it is in an error-free or in a failed state through the event ports. The mode transition occurs when one component reports a failure and the other component reports that it is error free. In this model, it is assumed that the failure is recoverable (i.e., the component can perform its service when reactivated through a mode transition).

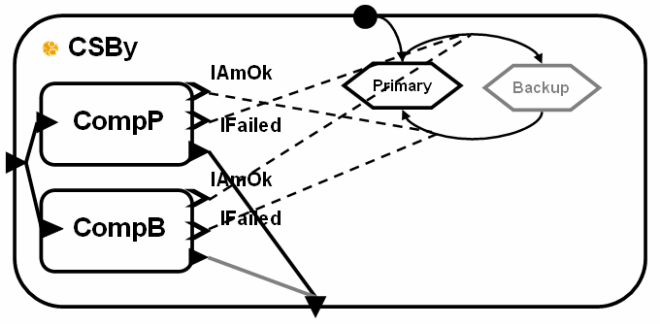


Figure 16: Hot Standby of a Self-Observing Component

Table 34 shows part of an AADL architecture model and associated error model. `Guard_Event` properties are associated with the `out` event ports involved in mode transitions. `Guard_Transition` properties define the mode transition condition with respect to the event ports and the mode transition specification with respect to the error states.

Table 34: Hot Standby of a Self-Observing Component

```
System computer
features
Input: in data port;
Output: out data port;
IAmOk: out event port;
IFailed: out event port;
end computer;

system implementation computer.personal
annex Error_Model {**
   Model => dependent.general;
   Guard_Event => self [ErrorFree] applies to IAmOk;
   Guard_Event => self [Failed] applies to IFailed;
**};
end computer.personal;

system CSBy
features
Input: in data port;
Output: our data port;
end CSBy;

system implementation CSBy.generic
subcomponents
CompP: system computer.personal;
CompB: system computer.personal;
connections
data port Input -> CompP.Input;
data port CompP.Output -> Output;
data port Input -> CompB.Input;
data port CompB.Output -> Output in modes Backup;
modes
Primary: initial mode;
Backup: mode;
ToBackup: Primary -[CompP.IFailed, CompB.IAmOk]-> Backup;
ToPrimary: Backup -[CompP.IAmOk, CompB.IFailed]-> Primary;
annex Error_Model {**
-- mode transition conditions
   Guard_Transition => CompP.IFailed and CompB.IAmOk
       applies to ToBackup;
   Guard_Transition => CompP.IAmOk and CompB.IFailed
       applies to ToPrimary;
**};
end CSBy.generic;
```

7.5.3 Self-Managing Components

In this scenario, the focus is on how a component is self managing (i.e., able to manage its own mode transitions). In other words, the component has several operational modes and will operate in different modes in error-free and error states. This structure differs from the previous two examples in which a system component was managing its subcomponents.

`Guard_Event` properties associated with outgoing event ports of the component are intended to report an error state to other components. This design would require that users route the event back to an incoming event port of the component to be named in a mode transition. Instead, we will associate the `Guard_Event` with an event local to the component, which is expressed by **`self`**`.eventname`.

We consider the following scenarios of a self-managing component.

- The component may be a thread (i.e., may cause failure if it is actively executing). The code of the thread itself may raise the event by calling on *Raise_Event*, or the event may be raised in one of the called subprograms. This scenario is illustrated on the left in Figure 17.
- The component may be a higher level system component (e.g., a process or system). In this case, the event may be raised by a subcomponent or the component itself. Where it is raised by the component, the event might be raised by the underlying runtime system or may represent an abstraction for a raised event in a partially specified system model, such as a model of major subsystems or partitions. This scenario is illustrated on the right in Figure 17.
- The fault of an application component may be detected by the execution platform (i.e., a processor, on behalf of the application component). This scenario is illustrated in the center in Figure 17.

Figure 17 shows a self-managing thread and system that have two modes. The modes are used to represent mode-specific property values, such as mode-specific execution time, and to allow for specification of mode-specific configurations of subcomponents and connections. The mode transition observes events raised by

- thread and system
- called subprogram or a subcomponent
- processor on which the thread executes

The raising of the event by the component itself is represented by a call to the *Raise_Event* system subprogram or by an event abstraction expressed by **`self`**`.eventname`. The figure also shows the thread and system observing an event raised by the processor (e.g., an event that is caused by a protected address space violation during the execution of the thread). Notice that the processor may report a fault through the exception handling mechanism of the language runtime system of the application. In that case, the application code of the thread may handle the exception and decide to raise a port event, as shown on the left.

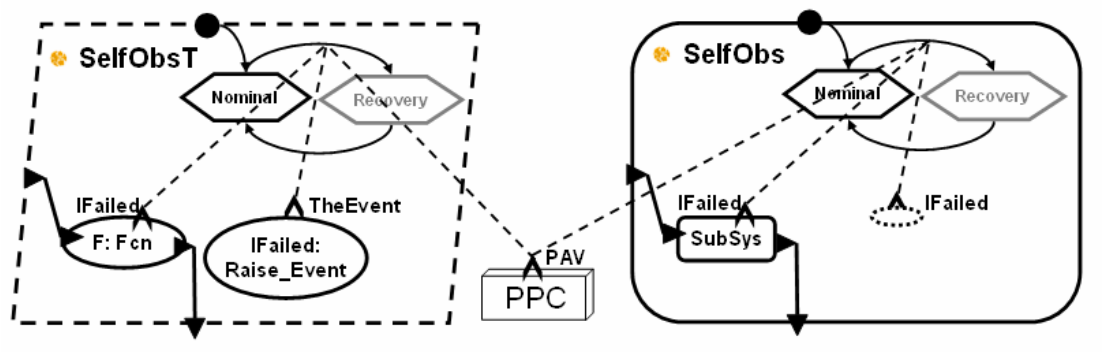


Figure 17: A Self-Managing Component

Table 35 shows a part of the AADL architecture model (textual AADL) and the associated error model for the self-managing thread and system examples depicted in Figure 17. The mode transition observes the thread's own raised events as well as events raised by the processor on which the thread executes. The raising of the event by the thread itself is represented by a call to the *Raise_Event* system subprogram. The mode transition refers to it by **`self`**`.eventname`. The `Guard_Event` property specifies under what conditions this event is raised. Notice that it is not required to explicitly specify the *Raise_Event* call unless its temporal order in the call sequence matters.

The fault may be identified by the subprogram *Fcn*. In that case, the subprogram call would report the event through an event port of the subprogram and the `Guard_Event` is part of the subprogram declaration. Events raised by the processor to which the thread is bound are named by the processor type. The system component follows the same pattern, but it does not include an explicit *Raise_Event* call.

Table 35: Self-Managing Component

```
thread SelfObsT
features
Input: in data port;
Output: out data port;
end SelfObsT;

thread implementation SelfObsT.personal
calls
f: subprogram Fcn;
IFailed: subprogram Raise_Event;
connections
data port Input -> F.Input;
data port F.Output -> Output;
modes
Nominal: initial mode;
Recovery: mode;
Nominal-[F.Ifailed, self.IFailed, PPC.PAV]-> Recovery;
annex Error_Model {**
    Guard_Event => self[FailedVisible] applies to self.IFailed;
**};
end SelfObsT.personal;

system SelfObs
features
Input: in data port;
Output: out data port;
end SelfObs;

system implementation SelfObs.personal
subcomponents
subsys: system;
connections
data port Input -> subsys.Input;
data port subsys.Output -> Output;
modes
Nominal: initial mode;
Recovery: mode;
Nominal-[subsys.Ifailed, self.IFailed, PPC.PAV]-> Recovery;
annex Error_Model {**
    Guard_Event => self[FailedVisible] applies to self.IFailed;
**};
end SelfObs.personal;
```

7.5.4 A Monitoring Component

In this scenario, we introduce a monitoring component that monitors the output of the component to determine whether the active component fails. The system modeled in Figure 18 consists of two identical active subcomponents and one monitoring component that makes decisions about the two subcomponents based on its observation of the output. Each of the active subcomponents can be in sender or receiver mode. In the sender mode, the component provides output, while in the receiver mode it is in hot standby (i.e., does processing but sends the output only to the monitor).

The *Monitor* component observes the output of the components. If the output of the active component is missing or bad, the monitor initiates a mode transition. If it detects the failure of both components, it waits for one of them to become operational and configures it through mode transition to send its output to other components. The controller initiates mode transitions by means of `Guard_Event` properties associated with its outgoing event ports.

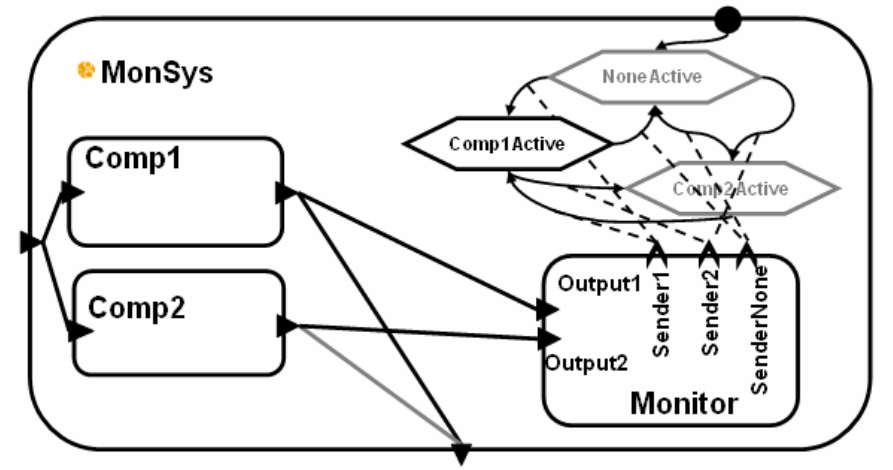


Figure 18: Monitoring Component

It is not necessary to associate `Guard_Transition` properties with mode transitions, as each mode transition names only one event port. Table 36 shows a part of the AADL architecture model (textual AADL) and the associated error model.

Table 36: Monitoring Component

```
System computer
features
Output: out data port;
end computer;

system implementation computer.impl
annex Error_Model {**
   Model => independent.general;
**};
end computer.impl;

system Monitor
features
Output1: in data port;
Sender1: out event port;
Output2: in data port;
Sender2: out event port;
SenderNone: out event port;
end Monitor;

system implementation Monitor.generic
annex Error_Model {**
   Model => independent.general;
   Guard_Event => Output1[FailedVisible] and Output2[ErrorFree]
        applies to Sender2;
   Guard_Event => Output2[FailedVisible] and Output1[ErrorFree]
     applies to Sender1;
   Guard_Event => Output2[FailedVisible] and Output1[FailedVisible]
     applies to SenderNone;
**};
end Monitor.generic;

system MonSys
features
Output: out data port;
end MonSys;

system implementation MonSys.generic
subcomponents
Comp1: system computer.impl;
Comp2: system computer.impl;
Monitor: system Monitor.generic;
connections
data port Comp1.Output-> Output in modes Comp1Active;
data port Comp2.Output-> Output in modes Comp2Active;
data port Comp1.Output->Monitor.Output1;
data port Comp2.Output->Monitor.Output2;
Modes
NoneActive: initial mode;
Comp1active: Mode;
Comp2Active: Mode;
NoneActive -[Monitor.Sender1]-> Comp1Active;
NoneActive -[Monitor.Sender2]-> Comp2Active;
Comp2Active -[Monitor.Sender1]-> Comp1Active;
Comp1Active -[Monitor.Sender2]-> Comp2Active;
Comp2Active -[Monitor.SenderNone]-> NoneActive;
Comp1Active -[Monitor.SenderNone]-> NoneActive;
end MonSys.generic;
```

7.5.5 Mutually Informing Components

In this scenario, we have two components that inform each other about their state. Each component decides whether it should be the active sender and determines whether it has encountered a fault or is in a state to operate normally. A component goes into a reboot state to repair itself and re-enter a normal operational state.

The system modeled in Figure 19 consists of two identical subcomponents. Each subcomponent can be in one of these modes: sender, receiver, reboot to become sender, or reboot to become receiver. When a subcomponent is in the sender mode, it provides the service expected from the system. If a failure occurs in a component, the component reports that fact to the other component as an event *IFailed* and goes into reboot mode. The other component must switch to sender mode, so that the expected service continues to be provided. Once the component has recovered, it transitions into receiver mode. If the second component fails as well, both components reboot.

When both components fail, the first one to complete its reboot goes into the sender mode. Similarly, during startup both components start in reboot and the first one to complete its initialization sequence (in this case assumed to be the same as used for reboot) will transition to the sender mode, while the second component transitions to receiver mode. The figure shows event ports named in mode transitions as dashed lines. It also shows locally raised events as dashed ovals. Naming of locally raised events in a mode transition is shown as a label on the mode transition.

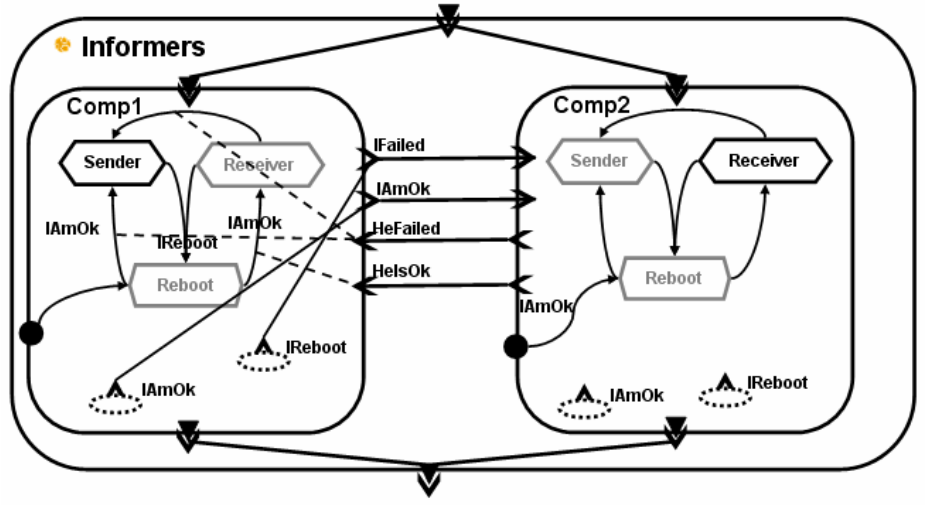


Figure 19: Mutually Informing Components

The logic behind the mode transitions is described in the error model subclause of the component. Table 37 shows a part of an AADL architecture model and associated error model. *Comp1* and *Comp2* have the same error model. Locally raised events are shown syntactically as `self.eventname`. Each locally raised event is also connected to an outgoing event port in order to report the event to the companion component.

The error model subclause contains two sets of declarations. The first is a set of `Guard_Event` declarations to represent under what error model conditions a local event is raised. The second set of `Guard_Transition` declarations that indicate under what port event conditions a mode transition occurs.

Table 37: Mutually Informing Components

```
system computer
features
IFailed: out event port;
IAmOk: out event port;
HeFailed: in event port;
HeIsOk: in event port;
Input: in event data port;
Output: out event data port;
end computer;

system implementation computer.impl
connections
event port self.IReboot -> IFailed;
event port self.IAmOk -> IAmOk;
modes
Sender: mode;
Receiver: mode;
Reboot: initial mode;
DoReboot1: Sender-[self.IReboot]->Reboot;
DoReboot2: Receiver-[ self.IReboot]->Reboot;
BeSender1: Reboot-[self.IamOk,HeFailed]->Sender;
BeReceiver: Reboot-[self.IAmOk,HeFailed]->Receiver;
BeSender2: Receiver-[ self.IAmOk,HeFailed]->Sender;
annex Error_Model {**
   Model => dependent.general;
   Guard_Event => self[ErrorFree]
     applies to self.IAmOk;
   Guard_Event => self[Failed]
      applies to self.IReboot;
   Guard_Transition => HeFailed and self.IAmOk
        applies to BeSender1;
   Guard_Transition => HeFailed and self.IAmOk
        applies to BeSender2;
   Guard_Transition => HeIsOk and self.IAmOk
     applies to BeReceiver;
   **};
end computer.impl;

system Informers
features
Input: in data port;
Output: out data port;
end Informers;

system implementation Informers.generic
subcomponents
Comp1: system computer.impl;
Comp2: system computer.impl;
connections
event data port Input -> Comp1.Input;
event data port Input -> Comp2.Input;
event data port Comp1.Output -> Output;
event data port Comp2.Output -> Output;
event port Comp1.IAmOk -> Comp2.HeIsOk;
event port Comp1.IFailed -> Comp2.HeFailed;
event port Comp2.IAmOk -> Comp1.HeIsOk;
event port Comp2.IFailed -> Comp1.HeFailed;
end Informers.generic;
```

7.5.6 Mutually Observing Components

In this scenario, we have two components that observe each other's output to determine whether the other component has failed. Based on this information and its own error state, each component decides whether it should be the active sender of output. A component goes into a reboot state to repair itself and be able to enter a normal operational state again. Different from the mutually informing component scenario, a component in this scenario does not require the cooperation of the other component to report its own error states. Instead lack of output and erroneous output allow the observing component to determine that the other component misbehaves. The mutually observing model is shown in Figure 20.

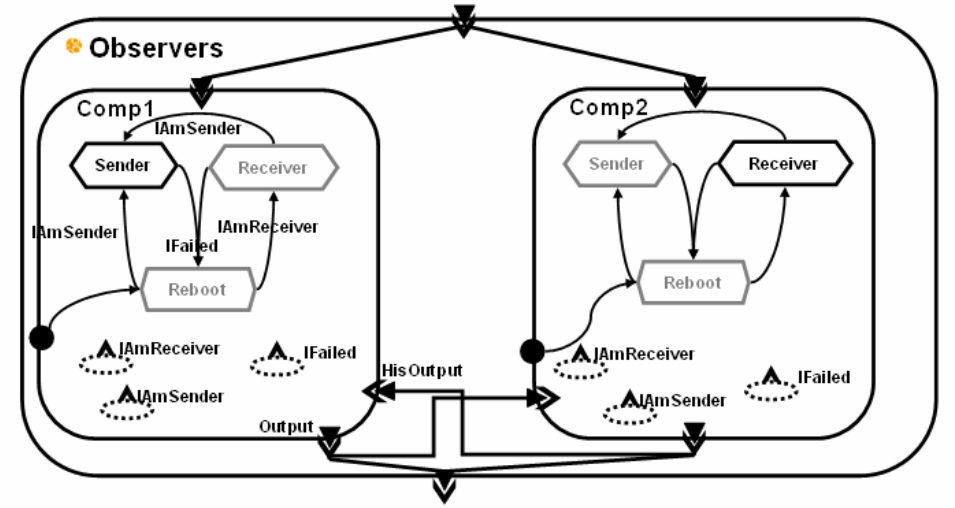


Figure 20: Mutually Observing Components

The AADL text description of the mutually observing component patterns is shown in Table 38. The model primarily differs in that the conditions for mode transitions are specified as guard events on the error propagation from the other component based on the exchange of component outputs.

Table 38: Mutually Observing Components

```
system computer
features
Input: in event data port;
Output: out event data port;
HisOutput: in event data port;
end computer;

system implementation computer.impl
modes
Sender: mode;
Receiver: mode;
Reboot: initial mode;
DoReboot1: Sender-[self.IFailed]->Reboot;
DoReboot2: Receiver-[self.IFailed]->Reboot;
BeSender1: Reboot-[self.IAmSender]->Sender;
BeReceiver: Reboot-[self.IAmReceiver]->Receiver;
BeSender2: Receiver-[self.IAmSender]->Sender;
annex Error_Model {**
   Model => dependent.general;
   Guard_Event =>
      self[Failed] applies to self.IFailed;
   Guard_Event =>
      HisData[FailedVisible] and self[ErrorFree]
        applies to self.IAmSender;
   Guard_Event =>
      HisData[ErrorFree] and self[ErrorFree]
     applies to self.IAmReceiver;
   **};
end computer.impl;

system Observers
features
Input: in data port;
Output: out data port;
end Observers;

system implementation Observers.generic
subcomponents
Comp1: system computer.impl;
Comp2: system computer.impl;
connections
event data port Input -> Comp1.Input;
event data port Input -> Comp2.Input;
event data port Comp1.Output -> Output;
event data port Comp2.Output -> Output;
event data port Comp1.Output -> Comp2.HisOutput;
event data port Comp2.Output -> Comp1.HisOutput;
end Observers.generic;
```

Observations

- The examples described in Sections 7.5.1 through 7.5.6 illustrate techniques for representing management of faults by reconfiguration of the system mode transition. This reconfiguration may be achieved by making the component itself modal or by the enclosing the component that is taking responsibility for reconfiguring its subcomponents and connections between them.
- Four of the examples (see Sections 7.5.1, 7.5.27.5.5, 7.5.3, and 7.5.5) illustrate the use of components that are able to discover their own faults. In contrast, the monitoring component and the mutually observing component examples (see sections 7.5.4 and 7.5.6) illustrate that component failure may be observed externally by monitoring the component's output. Bad output, no output, and untimely output are examples of observations that allow a monitoring component to draw the conclusion that a component has failed.
- All dual-redundant system examples use the same approach to fault tolerance, namely dual redundancy. Each of them, however, makes different assumptions, uses different tactics, and has different impact on the reliability of the system as a whole. For example, cold standby (see Section 7.5.1) assumes that the component does not maintain state. Hot standby (see Section 7.5.2) allows both components to maintain state independently. A monitoring component (see Section 7.5.4) observes output simultaneously with the output being made available to other components, resulting in occasional bad data to be passed on.
- In the mutually informing components example (see Section 7.5.5), the component detects its own faults and reports them immediately (`Failed` state); in the mutually observing components example (see Section 7.5.6), the component observes the fault of the other component. For mutually observing components, this observation may be delayed relative to the occurrence of the fault and is captured by the `FailedVisible` error propagation.
- Systems can be modeled at different levels of detail. The monitoring system and the mutually observing component examples (see Sections 7.5.4 and 7.5.6) explicitly model the occurrence of double faults and fault recovery.
- Error models provide a specification of faults and fault occurrence rates. They also provide specifications of desired fault management strategies—expressed in mode transition and event guards.

8 Modeling Maintenance and Repair

Maintenance dependencies need to be described when repair facilities are shared between components or when the maintenance or repair activity of some components has to be carried out according to a given order or a specified strategy (i.e., software can be restarted only if the hardware is available).

Components that are not dependent at the architectural level may become dependent due to the maintenance strategy. Thus, the AADL architecture model might need to be adjusted to support the description of dependencies related to the maintenance strategy. Because AADL error models interact only by propagations through architectural features (i.e., connections or bindings), the maintenance dependency between components' AADL error models must also be supported by the AADL architecture model. Consequently, besides the system architecture components, we might need to add an AADL architecture model component to describe the maintenance strategy.

Figure 21 in the area marked `a:` shows an example of AADL dependability model. In this architecture, *Component 3* and *Component 4* do not interact at the AADL architecture level, as there is no architecture-based dependency between them. However, if we assume that they share one repairman, the maintenance strategy has to be accounted for in the AADL error model of the system. Thus, it is necessary to represent the repairman at the AADL architecture model level, as shown in Figure 21 in the area marked `b:` in order to model explicitly the maintenance dependency between *Component 3* and *Component 4*.

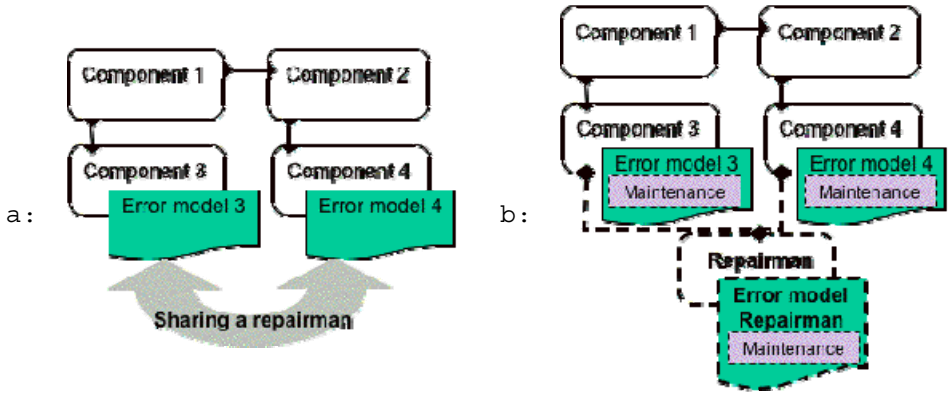


Figure 21: Maintenance Dependency

For the simplest repair strategy, the error model associated to the repairman should declare two error states—one representing a free repairman and the other representing a busy repairman. The transitions from one state to the other have to be triggered by propagations coming from the dependent components. If the repairman is busy and a component needs it, the component keeps sending the request until the repairman becomes free again. When the repairman is able to start repairing a component, the component sends a notification to that component. This error model is described in Table 39.

Table 39: Error Model for Shared Repairman

```
error model SharedRep
features
Free: initial error state;
Busy: error state;
RepairMe, Repaired: in error propagation;
StartRepair: out error propagation;
   {Occurrence => fixed 1};
end SharedRep;

error model implementation SharedRep.basic
transitions
Free-[in RepairMe]->Busy;
Busy-[out StartRepair]->Busy;
Busy-[in Repaired]->Free;
end SharedRep.basic;
```

9 Analysis Report Information

AADL architecture models can be enriched with dependability-related information (through error annex library and subclause declarations) and subjected to several dependability analyses. You need a mechanism to specify the error states and error propagations that are of particular interest for an analysis. The AADL Error Model Annex standard specifies a `Report` property that has to be associated to such states and **`out`** or **`in out`** propagations. The use of this property needs to be specified in the context of a specific analysis. For example, for dependability evaluation purposes, the `Report` property could be associated to up states. Table 40 shows how the `Report` property might be used.

Table 40: Report Property

```
system implementation mySystem.generic
annex Error_Model {**
   Model => dependent.general;
   Report => ErrorFree;
**};
end mySystem.generic;
```

10 Summary

The Society of Automotive Engineers AADL standard was designed to model embedded systems architectures. AADL architecture models can be annotated with information to support a wide range of architectural analyses. The AADL Error Model Annex standard defines a standardized language extension to the AADL to represent information for dependability analysis. Using the annex, you can define reusable error models that consist of

- a set of error states
- error state transitions
- error events and error propagations that trigger error state transitions

Error models are associated with components and connections and can have component- or connection-specific properties. These properties specify the rate of occurrence of error events and the probability of error propagation.

This report has shown how error models can be used to model various fault tolerance strategies through error propagation filtering and masking. It has discussed the interaction between the logical system states represented by the error model and the operational system modes of the running system. It also has illustrated how maintenance and repair activities can be represented in error models.

This report is the first in a series by the Carnegie Mellon® Software Engineering Institute to provide guidance in architecture modeling and analysis with AADL. Other reports will discuss additional dependability-related topics—such as fault tree analysis, reliability analysis, and redundancy and health-monitoring patterns to describe fault management support in the operational system—as well resource management, security analysis, data integrity, and safety-criticality.

References

[Arlat 1998]
Arlat, J.; Blanquart, J.P.; Costes, A.; Crouzet, Y.; Deswarte, Y.; Fabre, J.-C.; Guillermain, H.; Kaâniche, M.; Kanoun, K.; Mazet, C.; Powell, D.; Rabéjac, C.; & Thévenod, P. *Dependability Handbook*. Edited by J.-C Laprie. LAAS-CNRS Report 98-346, 1998.

[Binns 2004]
Binns, P. & Vestal, S. "Hierarchical Composition and Abstraction in Architecture Models." *Workshop on Architectural Design Languages at the 18th IFIP World Computer Congress.* Toulouse, France, August 27, 2004. http://www.laas.fr/FERIA/SVF/WADL04/slides/CONCORDE2-27-1100-VESTAL/BinnsVestalADLWorkshop.ppt

[Feiler 2004]
Feiler, P. H.; Gluch, D. P.; Hudak, J.; & Lewis, B. A. "Pattern-Based Analysis of an Embedded Real-time System Architecture." *Workshop on Architectural Design Languages at the 18th IFIP World Computer Congress*. Toulouse, France, August 27, 2004. http://www.laas.fr/FERIA/SVF/WADL04/slides/CONCORDE2-27-1100-LEWIS/AADLpatternstoulouse.pdf

[Feiler 2006]
Feiler, P. H.; Gluch, D. P.; & Hudak, J. J. *The Architecture Analysis & Design Language (AADL): An Introduction.* (CMU/SEI-2006-TN-011). Pittsburgh, PA: Software Engineering Institute Carnegie Mellon University, 2006. http://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html

[SAE-ARP4761 1996]
Society of Automotive Engineers. *SAE Standards: ARP4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.* December 1996. http://www.sae.org/technical/standards/ARP4761

[SAE-AS5506 2004]
Society of Automotive Engineers. *SAE Standards: AS5506, Architecture Analysis & Design Language (AADL)*, November 2004. http://www.sae.org/servlets/productDetail?PROD_TYP=STD&PROD_CD=AS5506

[SAE-AS5506/1 2006]
Society of Automotive Engineers. *SAE Standards: AS5506/1, Architecture Analysis & Design Language (AADL) Annex Volume 1*, June 2006. http://www.sae.org/technical/standards/AS5506/1

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Dependability Modeling with AADL	FA8721-05-C-0003

6. AUTHOR(S)

Peter Feiler and Ana Rugina

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2007-TN-043

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

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

13. ABSTRACT (MAXIMUM 200 WORDS)

The Society for Automotive Engineers (SAE) recently published an Error Model Annex document (SAE AS-5506/1) to complement the SAE Architecture Analysis & Design Language (AADL) standard document (SAE AS5506) with capabilities for dependability modeling. The purpose of this report is to (a) explain the capabilities of the Error Model Annex and (b) provide guidance on the use of the AADL and the error model in modeling dependability aspects of embedded system architectures. The focus of the guidance is the creation of error model libraries and the instantiation of these error models on AADL architecture models. In that context, the report discusses modeling of error propagation, error filtering and masking, the interactions between error models and systems with operational modes, and modeling of repair activities.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Error model, AADL, architecture analysis and design language, dependability analysis, model-based engineering	86

16. PRICE CODE

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