ACVIP Perspective on Safety Analysis Using AADL

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Document Marking

This material is based upon work funded and supported by the US Army Development Command Aviation and Missile Center under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

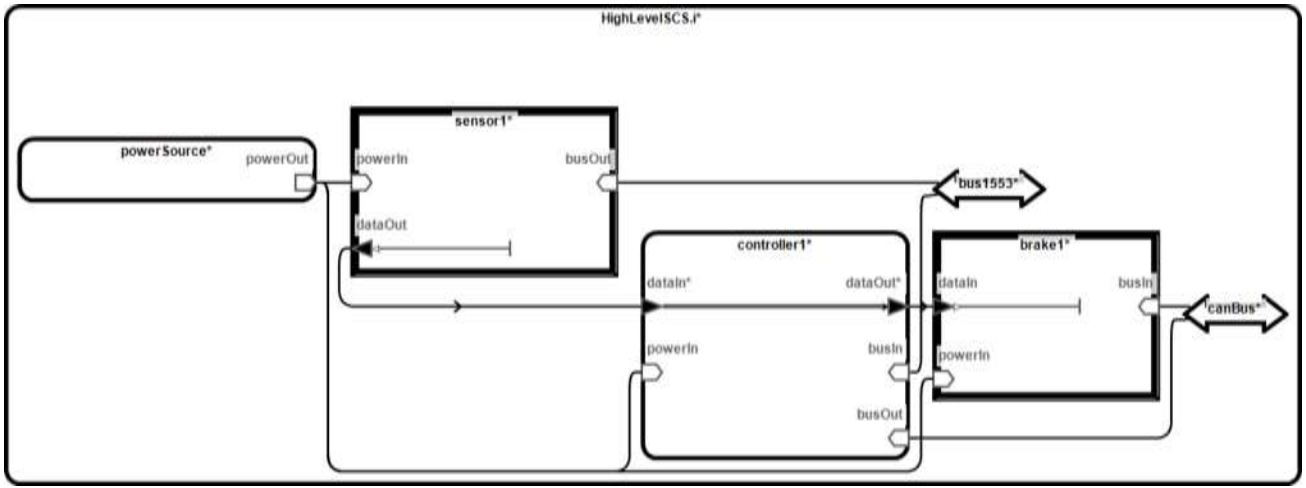
DM22-0111

Document Marking

DM22-0111

Previously …

In the previous video, “ACVIP Perspective on AADL,” the basic ideas behind the Architecture Centric Virtual Integration Process (ACVIP) were illustrated using a small model of a control system. The AADL model is shown below.



In this video, we describe the system’s design for handing errors. The error annex also has facilities for modeling the system’s safety hazards, which feed early safety analyses.

Viewpoints

This presentation was developed with two viewpoints in mind:

1. Model Creators
   - Who will use the model I create?
   - What information must be in my model to support product development?

2. Model Users
   - What does each part of the model contribute to the analyses?
   - How does this model help produce a product?

ACVIP Safety Analysis in Context

During Acquisition	A modeling plan is developed that specifies the models needed to provide sufficient information to support conclusions reached during analysis.
During Analysis	The use of AADL and OSATE traverse the models to produce evidence that can be used to guide decision making.
During Product Implementation	The evidence from model analysis directs corrections in existing designs and provides a basis for comparisons to previous and future analysis results to determine whether the design is closer to meeting all constraints.

Safety Analyses

Safety analyses are conducted at the system level and are often are conducted very early in development, using high-level requirements, or very late in development, using more detailed requirements after implementations.

Safety-critical systems are becoming increasingly software reliant. However, the complexity of this software has become a major source of defects with potentially fatal consequences. To address these issues, the SAE Architecture Analysis & Design Language (AADL) standard has been extended with an Error Model Annex to support architecture fault modeling and automated safety analysis.

AADL models are intended to be applied early and provide more structure than early requirements. Error model creators use the Error Ontology provided in AADL to define the flows of erroneous data in the system. Users of the model will implement these structures in the system.

In particular, we will illustrate the use of hazard descriptions to support fault tree analysis and others. First, we show the types of error modeling and then we apply them.

AADL Analysis in OSATE

Analysis	Analysis Description
Bus Load *	The bus load analysis looks at the connections and virtual buses bound to buses in a system and checks that the bus has the capacity to carry the necessary data.
Latency *	The latency analysis is performed on AADL models that include end-to-end flows, and it calculates minimum and maximum latency taking into account a wide range of latency contributors.
Power Requirements	Electrical power analysis framework supports the concept of power suppliers and power consumers. A power transmission system is used to move electrical power from suppliers to consumers.
Resource Allocation (Bound)	Analysis adds up memory capacity and keeps track of how many components are expected to have capacity property values and how many have the appropriate property value assigned.
Resource Budgets (Not Bound)	The intent of the resource budget analysis is to provide resource budgeting support early in the development lifecycle.
Fault Tree Analysis (FTA)	FTA is a deductive, failure-based approach used to support the probabilistic risk assessment of systems. FTA starts with an undesired event, such as failure of a main engine, and then determines (deduces) its causes using a systematic, backward-stepping process of identifying constituent components and their failure events (and probabilities). In determining the causes, a fault tree is constructed as a logical illustration of the events (and their relationships) necessary and sufficient to result in the undesired event.
Connection Consistency	The objective of this analysis is to check that there is a physical connection between the hardware components related to the source and destination of a connection.
Binding Constraints	The objective of this plug-in is to check that a model conforms to various constraints specified in binding properties.

* Can be run on the SCS model presented in these slides

Motivation for Using Error Modeling

The content of the AADL error model directly supports many of the architecture analyses that we will apply. These early analyses address the earliest parts of

Fault => Error => Failure

AADL Error Model Constructs		FHA	FTA	FMEA
Error Flows	Error propagation	X	X	X
	Error source	X	X	X
	Error path			X
	Error sink		X	X
Error Behaviors	Error states		X	
	Error transitions		X	
	Error events	X	X	X
	Composite error model		X	
Properties	Hazard records	X		

FHA = Fault hazard assessment

FTA = Fault tree analysis

FMEA = Failure mode effect analysis

System Error Model

The Error Annex (EMV2) in AADL defines constructs with 2 different scopes:

- Component scope
- Composite scope

```
system implementation bscu_subsystem.generic
    subcomponents
        mon: process monitor.i;
        cmd: process command.i;
    connections
        pedaltocmd: port pedal -> cmd.pedalvalue;
        brakecmd: port cmd.brake -> mon.brake;
        brakecmd_ext: port cmd.brake -> cmd_brk;
        skidcmd_ext: port cmd.skid -> cmd_skid;
        skidcmd: port cmd.skid -> mon.skid;
        isvalid: port mon.valid -> valid;
    annex EMV2 {**
        use types error_library;
        use behavior error_library::simple;

        error propagations
            pwr: in propagation {NoPower};
            valid: out propagation {NoValue};
            flows
                nopwr: error path pwr {NoPower} -> valid {NoValue};
        end propagations;

        component error behavior
            transitions
                t1: Operational -[pwr {NoPower}]-> Failed;
            propagations
                p1: Failed -[]-> valid {NoValue};
        end component;

        composite error behavior
            states
                [mon.failed or cmd.failed]-> Failed;
                [mon.operational and cmd.operational]-> Operational;
        end composite;

        properties
            ARP4761::hazards => ([crossreference => "ARP4761 figure L4 page 215";
                    failure => "Failure of the BSCU, either from the monitor, the command or both";
                    phases => ("all");
                    description => "Failure of a BSCU";
                    FailureConditionClassification => Major;
                    QualitativeProbabilityObjective => Probable;
                    comment => "Would be critical if two subsystem (primary and redundant) are deffective";
                ]) applies to Failed;
            EMV2::OccurrenceDistribution => [ProbabilityValue => 3.3e-5; Distribution => Fixed;] applies to Failed;
    **};
end bscu_subsystem.generic;
```

Basic Error Model (EMV2) Concepts

Error

An incorrectness that results from executing a fault and perhaps will cause a failure

Error Type

A classification of errors

Error propagation

The error flows from one element to another

```
annex EMV2 {**
    use types error_library;
    use behavior error_library::simple;

    error propagations
        pwr: in propagation {NoPower};
        valid: out propagation {NoValue};
        flows
            nopwr: error path pwr {NoPower} -> valid {NoValue};
    end propagations;

    component error behavior
        transitions
            t1: Operational -[pwr {NoPower}]-> Failed;
        propagations
            p1: Failed -[]-> valid {NoValue};
    end component;

    composite error behavior
        states
            [mon.failed or cmd.failed]-> Failed;
            [mon.operational and cmd.operational]-> Operational;
    end composite;
```

Basic Error Model (EMV2) Concepts

Error source

The location of the fault that produces the error

Error sink

The location at which the error is “handled”

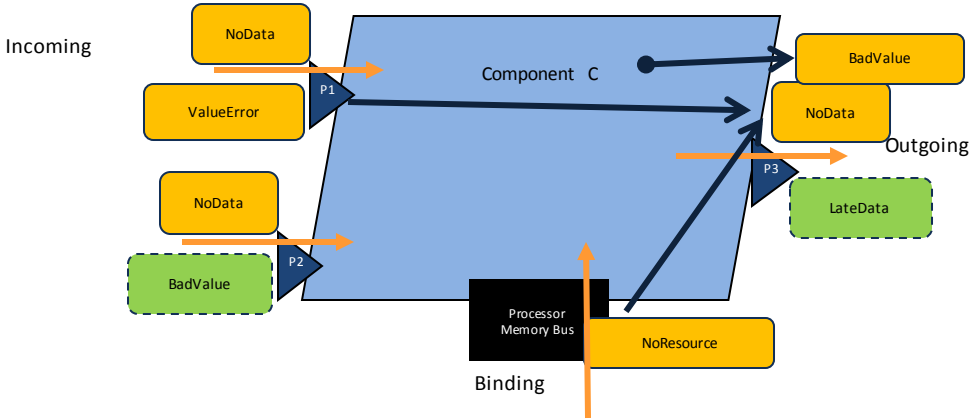
```
process command
    features
        brake: out data port common::command.brake;
        skid: out data port common::command.skid;
        pedalvalue: in data port common::command.pedal;
end command;

process implementation command.i
    annex EMV2 {**
        use types error_library;
        use behavior error_library::simple;

        error propagations
            pedalvalue: in propagation {NoService};
            brake: out propagation {NoValue};
            skid: out propagation {NoValue};
            processor: in propagation {SoftwareFailure, HardwareFailure};
            flows
                nopedal: error sink pedalvalue {NoService};
                noskid: error source skid {NoValue};
                nobrake: error source brake {NoValue};
                platformerr: error sink processor {SoftwareFailure, HardwareFailure};
        end propagations;
```

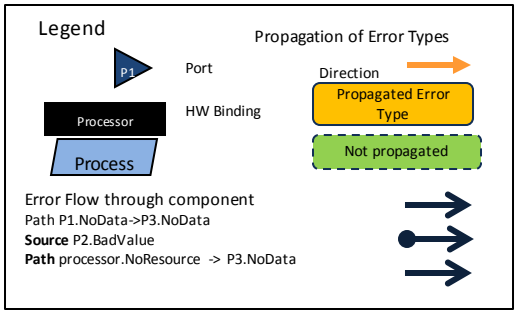
Component Error Models

Each architectural element should have representations of both the nominal and error behavior of the element.

A component may be a source of an error or a sink of an error or may pass an error through the component.

Error flows are supported by AADL components, such as error path, source, and sink. Error sink is an AADL component that is a final destination of an error.



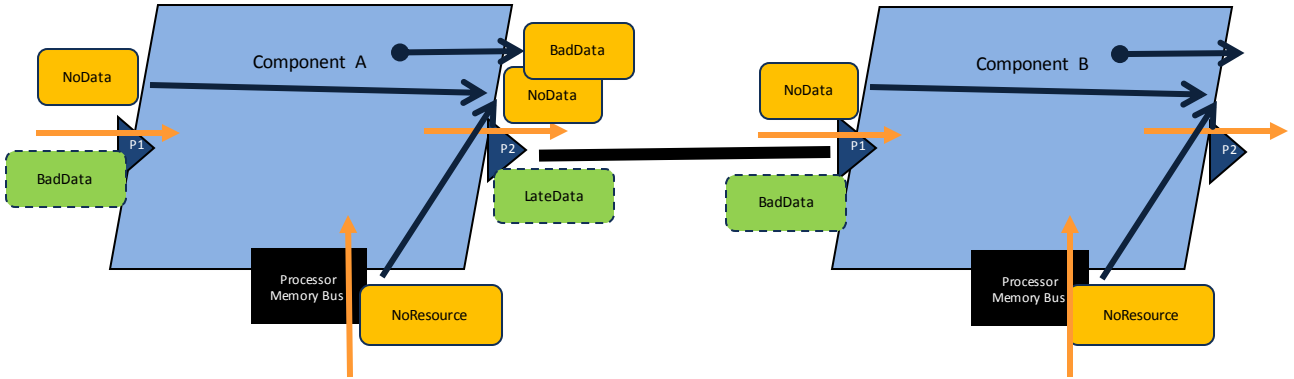


Composite Error Model - 1

The Model Creator composes a set of components, including both nominal and error models.

The Model User can follow nominal flows and trace the corresponding error flows using error paths and nominal connections.

As the models evolve both nominal and error sources, sinks, and paths are assigned types. Flows across components are checked for compatibility.



Composite Error Model - 2

The composite error model captures the propagation interactions of two or more components.

The state of the composition is defined in terms of the error states of the composed components.

The SCS is a composition of the sensor, controller, and actuator. Its very simple error behavior is represented below:

```
composite error behavior
--build behavior for all pieces of the system
--Example of component failure results in system FailStop
states
          [1 ormore( simpleSensor.FailStop and
          simpleSensor2.FailStop,
simpleActuator.FailStop,
          simpleController.FailStop)]-> FailStop;
end composite;
```

ACVIP, when Powered by AADL, Enables System-Level Analysis Earlier than Traditional Development Methods

We continue to use the example from our previous video by adding a model of the error behavior of each component.

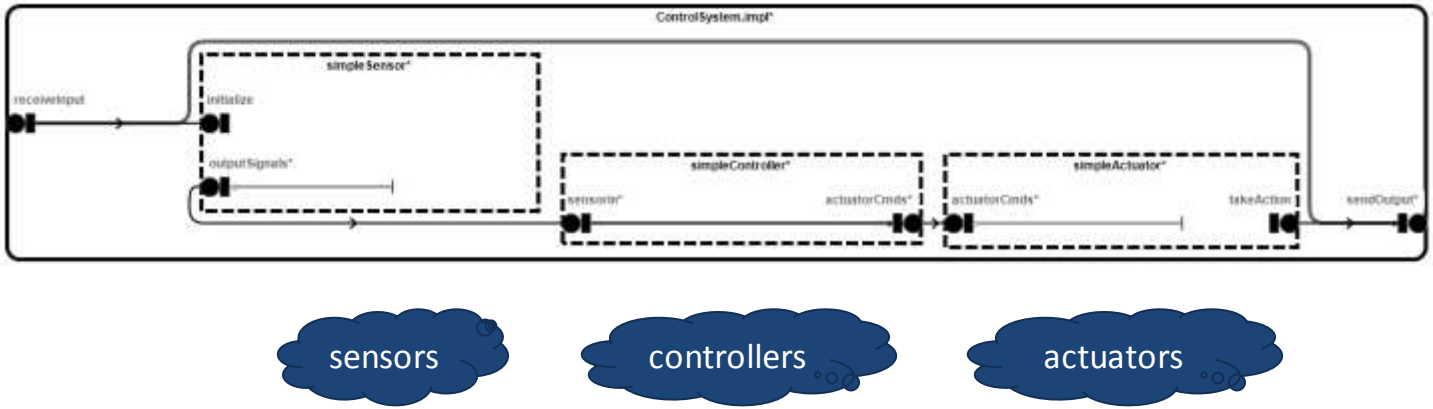
This will enable the application of several standard analyses, such as functional hazard assessment, fault impact analysis, and others. The AADL models have less detail than later models but sufficient detail to compute the relationships needed to support reasoning.

The model of the hazards that may be present in the running system are the basis for much of this analysis.

Context

We will use a portion of our Simple Control System (SCS) model.

The constructs used here focus on error modeling. Some features have been omitted for pedagogical purposes.



Sensor.aadl

The definition of an AADL architecture element is divided into two parts: specification and implementation.

The definition of sensor used here shows the sensor as the source of sensor data. Out ports and bus accesses allow this data to flow to other architectural elements.

The property values, given in the implementation section, specify how the flows behave.

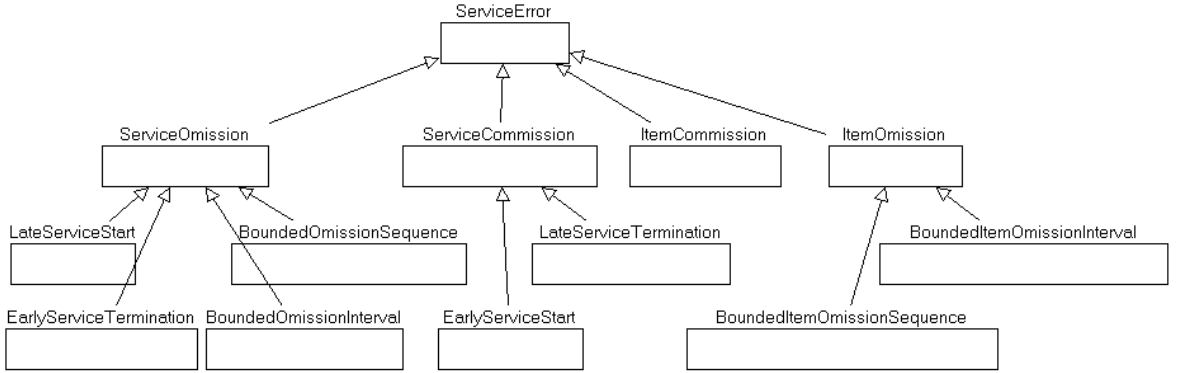
```
device SimpleSensor
features
dataOut: out data port DataDictionary::SensorData;
busOut: requires bus access Infrastructure::bus1553.i;
powerIn: requires bus access Infrastructure::power.generic;

flows
sensorFlow: flow source dataOut;
end SimpleSensor;

device implementation SimpleSensor.i
properties
dispatch_protocol => periodic;
period => 50ms;
latency => 3ms..5ms applies to sensorFlow;
SEI::WeightLimit => 0.2kg;
SEI::powerBudget =>4.0W applies to powerIn;
end SimpleSensor.i;
```

Annex devoted to error modeling

In addition to being a source of sensor data, the sensor might also be the source of error.

For example, if the sensor is expected to produce data at a specific rate and if that data is not produced at that rate, that is an error ~~an error~~ of omission. The program logic needed to handle that issue is different from the flow for nominal data.

The Error Model version 2 (EMV2) provides the capability to model those conditions, including a taxonomy of error types. The image below presents one of four available families of error types.



A Basic Error Annex for an Architectural Element

```
annex EMV2{**
use types ErrorLibrary;
use behavior ErrorLibrary::FailStop;
error propagations
dataout : out propagation {ServiceOmission,ValueError};

flows
f2 : error source dataout {ServiceOmission,
ValueError};
end propagations;
component error behavior
events
serviceommission:error event;
valueerror:error event;
transitions
Operational-[serviceommission]-> FailStop;
Operational-[valueerror]-> FailStop;
end component;
**};
```

Error Library

The Error Library contains error type definitions and reusable state machines of interacting error states.

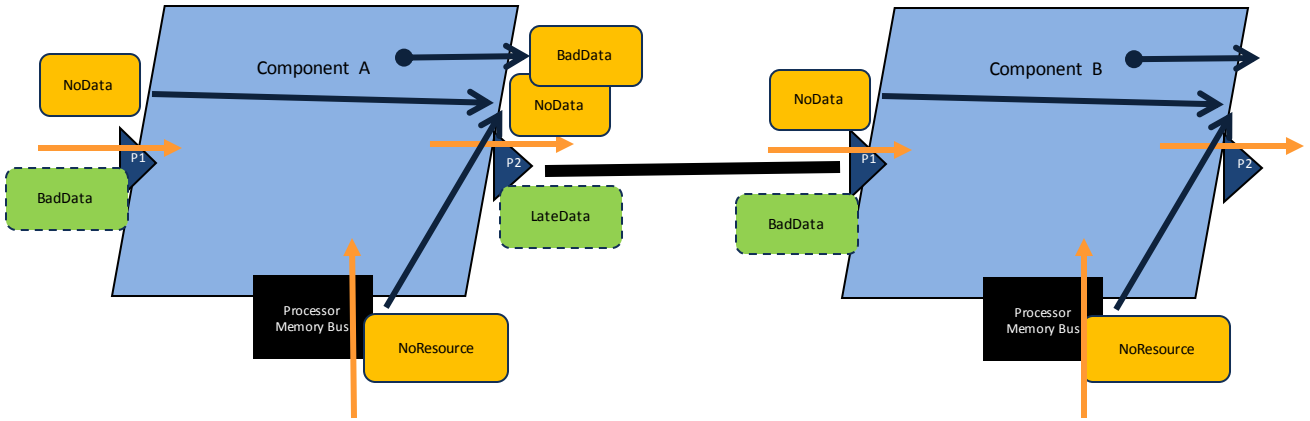
A simple one shows the following:

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

This is used in a specification as:
use behavior ErrorLibrary::FailStop
```

Composite Error Model

The Model Creator composes components, including the error models.

The Model User can follow nominal flows and trace the corresponding error flows using error paths and nominal connections.



Composite Error Model

The composite error model captures the propagation interactions of two or more components.

The state of the composition is defined in terms of the error states of the composed components.

The SCS is a composition of the sensor, controller, and actuator. Its very simple error behavior is represented below:

```
composite error behavior
--build behavior for all pieces of the system
--Example of component failure results in system FailStop
states
          [1 ormore( simple Sensor.FailStop and
          simpleSensor2.FailStop,
simpleActuator.FailStop,
          simpleController.FailStop)]-> FailStop;
end composite;
```

Error Propagations

An error not handled within a component flows to a connected component. Those flows occur through ports defined in the specification.

These propagations participate in error flows.

```
error propagations
dataout : out propagation {ServiceOmission, ValueError};

flows
f2 : error source dataout {ServiceOmission, ValueError};
end propagations;
```

Error States

This portion of the error model defines a state machine for sequencing error behavior.

In particular, moving from one state to another is defined in this machine.

At this time, there is no link between the mode defined in the core and the error machine in the error model.

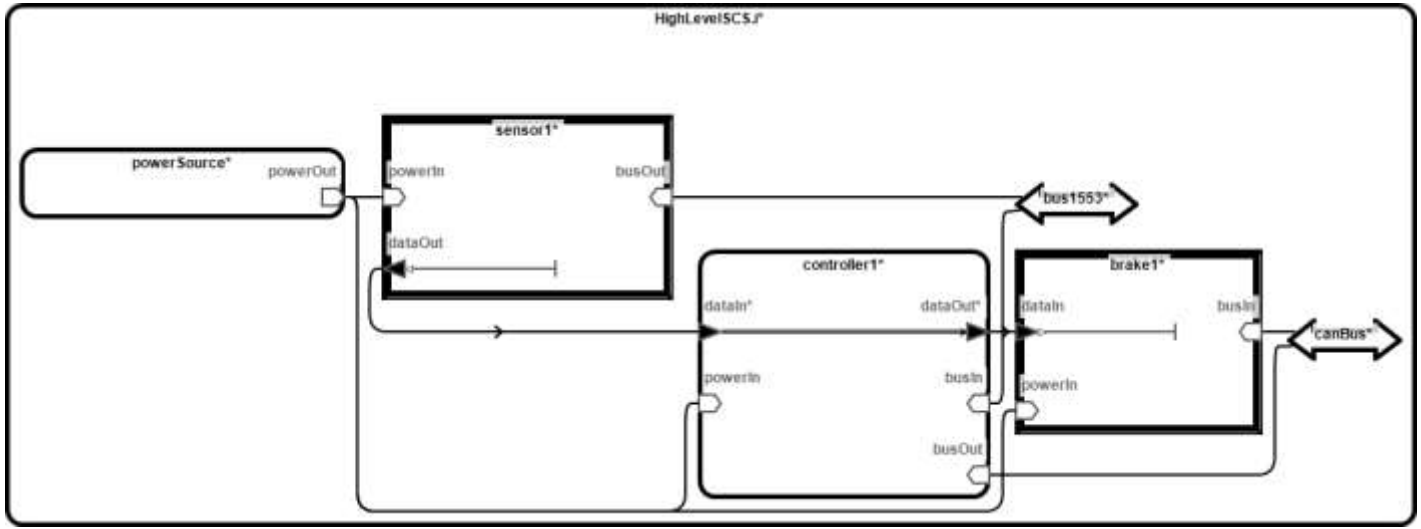
```
component error behavior
events
serviceommission:error event;
valueerror:error event;
transitions
Operational-[serviceommission]-> FailStop;
Operational-[valueerror]-> FailStop;
end component;
```

System Model

In a real-life project, each component might be purchased from a different vendor and multiple contractors assemble the components into one system.

The constructs used here focus on system runtime behavior. Some features have been omitted for pedagogical purposes.

- **Component connection:** models component interactions, control flow, and/or data flow (e.g., exchange of message or remote call [RPC]).
- **Bus:** models a data exchange mechanism between components.



Predictable Multidisciplinary Analysis

The Model Creator must propose a model that will support the execution of analyses on different types of attributes to ensure that the system's attributes stay within acceptable bounds.

The Model Creator must also conduct the independent verification & validation (IV&V) process. IV&V is a comprehensive review, analysis, and testing (software and/or hardware) performed by an objective third party to confirm (i.e., verify) that the requirements are correctly defined and to confirm (i.e., validate) that the system correctly implements the required functionality and security requirements.

For example

1. FTA analysis identifies that sensor 1 is more prone to errors compared to sensor 2.
2. The Model Creator decides to swap a sensor 1 to sensor 2 to decrease probability of system error.
3. The Model Creator needs to run analysis to check if other parts of the system have been affected.

Let's review analysis focused on error behavior in next few slides and the models needed to run them.

Fault Hazard Assessment (FHA)

Component	Error Model Element	Hazard Title	Description	Crossreference	Failure	Failure Effect	Operational Phases	Risk	Severity	Likelihood	Comment
Root system	"ServiceOmission on ef0"		"No readings due to sensor failure"	""	"Loss of sensor readings"		"all"		Hazardous	Remote	"Becomes major hazard; if no rdeundant sensor"
Root system	"ServiceOmission on ef0"		"Incorrect readings due to sensor failure"	""	"Loss of sensor readings"		"all"		Hazardous	Remote	"Becomes major hazard; if no rdeundant sensor"
Root system	"ServiceOmission on ef0"		"No readings due to sensor failure"	""	"Missing sensor readings"		"all"		Hazardous	Remote	"Becomes major hazard; if no rdeundant sensor"
SimpleSensor1	"ServiceOmission on f1"		"Failed digital airspeed sensor does not provide any values"	"AIR6110 page 35 figure 17"	"Loss of sensor"	"Unstable control"	"all"	"Unstable control"	3	F	"Sensor used in all operational modes"
SimpleSensor2	"ServiceOmission on f1"		"Failed digital airspeed sensor does not provide any values"	"AIR6110 page 31 figure 10"	"Loss of sensor"	"Unstable control"	"all"	"Unstable control"	3	F	"Sensor used in all operational modes"
SimpleActuator	"Incident on to_throttle"		"Failed actuator does not provide control actions"	"AIR6110 page 35 figure 17"	"Loss of actuator"	"Throttle is stuck at last position"	"all"	"Unstable control"	3	F	"Actuator used in all operational modes"
SimpleController	"ServiceOmission on f1"		"Failed digital controller does not provide any values"	"AIR6110 page 10"	"Loss of controller"	"Unstable control"	"all"	"Unstable control"	1	A	"Controller used in all operational modes"

Hazards

```
EMV2::hazards =>
        ([      crossreference => "";
                failure => "Loss of sensor readings";
                phases => ("all");
                severity => ARP4761::Hazardous;
                likelihood => ARP4761::Remote;
                description => "No readings due to sensor failure";
                comment => "Becomes major hazard, if no redundant sensor";
                        ])
                        applies to ef0;
                emv2::occurrencedistribution => [ probabilityvalue => 0.15; Distribution =>Fixed;] applies to ef0.BadValue;
```

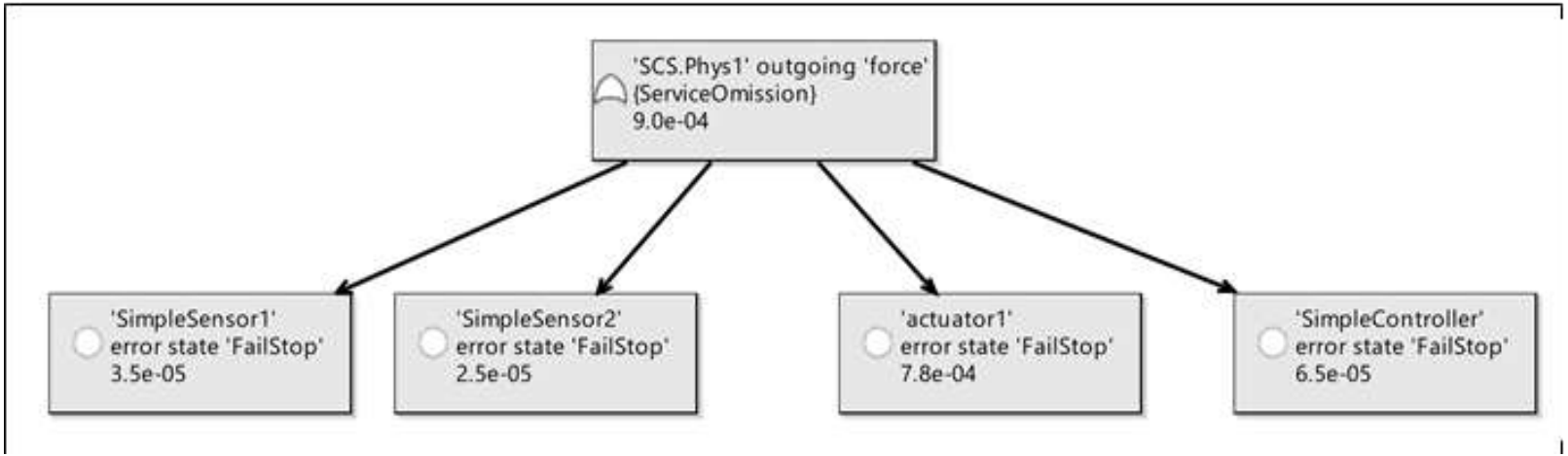
Fault Impact Analysis

Component	Initial Failure Mode	1st Level Effect	Failure Mode	second Level Effect	Failure Mode	third Level Effect	Failure Mode	4th Level Effect	Failure Mode
simpleSensor	{ServiceOmission}	{ServiceOmission} outputSignals -> simpleController.sw.iop:sensor1in	simpleController.sw.iop {ServiceOmission} [FlowPath]	{ServiceOmission} senseout -> simpleController.sw.app:samplingin	simpleController.sw.app {ServiceOmission} [FlowPath]	{ServiceOmission} controlout -> simpleController.sw.iop:controlin	simpleController.sw.iop {ServiceOmission} [FlowPath]	{ServiceOmission} actuatorout -> simpleActuator:actuatorCmds	simpleActuator {ServiceOmission} [Masked]

Tracing Error Propagations

```
annex EMV2 {**
   use types SysArchErrorModelLibrary;
      error propagations
         sensor1in.required: in propagation
            {ErrorLibrary::ServiceOmission,
             ErrorLibrary::ValueError};
         actuatorCmds.provided: out propagation
             {ErrorLibrary::ServiceOmission,
              ErrorLibrary::ValueError,
       SysArchErrorModelLibrary::ControllerFailed};
         sensor2in.required: in propagation
             {ErrorLibrary::ServiceOmission,
              ErrorLibrary::ValueError};
      flows
         DCS_singletier1_sen1_error_flow: error path
        sensor1in.required -> actuatorCmds.provided;
         DCS_singletier1_sen2_error_flow: error path
        sensor2in.required -> actuatorCmds.provided;
```

Fault Tree Analysis (FTA)



System Level Failure Condition

```
composite error behavior
   states
      [1 ormore (SimpleSensor1.FailStop,
         SimpleSensor2.Failstop,
         SimpleActuator.FailStop,
         SimpleController.FailStop)]-> FailStop;
end composite;
```

Summary

It is important to think about error handling, especially in safety critical systems.

The ACVIP perspective can be applied to AADL error models before implementation has started.

ACVIP methodology encourages to run analysis multiple times.

AADL provides variety of predefined error types and analysis; however, error model is extensible, and you can derive new error types that more closely fit your domain.

Summary

Achieving ACVIP methodology without AADL is possible, however doing so introduces multiple complications:

- With SysML, each user needs to write their own analysis and there's no way to validate those outside of the organization. With AADL, analysis comes prebuilt, and so it could be easily shared between contractors.

- If the model changes, SysML analysis will need to be rewritten to adjust for the new model structure. However, AADL analysis are robust enough to handle model changes and do not require any additional work to run.