Outline

• Overview & Background
• Basic concepts illustrated
• Error Types
• Error Propagation
• Component Error Behavior
• Hierarchical composition of error behavior
• Error Behavior and Fault Management
• Dual Redundancy and Modes
• Error Behavior in Layered Architectures
• Summary
Terminology (Based on IFIP WG10.4)

Impairment is the inability to provide nominal behavior.

A fault is a root (phenomenological) cause of an error or a failure.

An error is the difference in state from a correct state.

A failure is a deviation in behavior from a nominal specification.

A propagation is the creation of a new error or failure due to an error or failure.

Observation
We use the term error in describing Error Model language constructs.
Background

Steve Vestal at Honeywell implemented an Error Model extension to MetaH in 1998.

SAE AS5506 Architecture Analysis and Design Language (AADL), a standard for describing embedded computer system (software and hardware) architectures, was issued November 2004 and revised in Nov 2009.

The AADL language and standard permit extensions and tailoring.

SAE AS 5506-1, AADL Annex Volume 1, was issued in 2006
  • Includes Error Model Annex as small variation of MetaH Error Model capability.

Revision of Error Model Annex in progress to improve its expressive power and semantics
  • Expected publication as part of AS-5506-3 Volume 3 in early 2012.
AADL Error Model Scope and Purpose

System safety process uses many individual methods and analyses, e.g.

- hazard analysis
- failure modes and effects analysis
- fault trees
- Markov processes

Related analyses are also useful for other purposes, e.g.

- maintainability
- availability
- integrity

Goal: a general facility for modeling fault/error/failure behaviors that can be used for several modeling and analysis activities.
Example: Integrated Safety, FMEA, Reliability

- Capture hazards
- Capture risk mitigation architecture
- Capture FMEA model

Error Model features permit checking for consistency and completeness between these various declarations.
Goal of the Error Model Annex Revision

A core set of reliability concepts and error types
Interaction of systems with nominal behavior and threats in the form of defects, faults, misbehavior, violated assumptions resulting in error propagations (hazards)
Anomalous and threat mitigation behavior of a system or a system component
Composability of error model in the system hierarchy

Error model representation improvements
• Add error semantics to stochastic communicating state machines
• Properties on all error model concepts
• Error types and attributes
• Clean separation of error propagation, error behavior, composition
• Explicit propagation and observation
Approach for Error Model Annex Revision

Utilize core AADL property sublanguage
Define hierarchy of error types (Ontology)
Focus on fault propagation
  • External fault interaction with other components
Focus on fault behavior inside component
  • How the component deals with its internal faults & incoming propagations
Focus on component hierarchy
  • System fault behavior in terms of subsystem fault behaviors
Focus on interaction between fault management specification &
management implementation
  • Detection (heart beat, output monitoring)
  • Containment (space partition, time partition)
  • Restoration action: Recovery/repair behavior (replacement, extrapolation)
    – Isolation higher level symptom decomposed into lower level causes
Error Characteristics to be Reflected

Error event & recovery/repair event

Intrinsic error: defect, fault, misbehavior

- Permanent, transient error (state)
- Internal & propagated
- Intentional & unintentional propagation
- Unknown, Undocumented, Unhandled
- Prevented errors (prevention hazards)
- Mitigated errors (mitigation hazards)
- Detected, reported, mitigated, propagated
- Explicitly propagated & observable errors
Error Model and the Architecture

Propagation of errors of different types along propagation paths between architecture components.

Component error behavior as transitions between states triggered by error events and incoming propagations.

Error flows as abstractions of propagation through components. Composite error behavior in terms of component error behavior states.
Error Model Annex Sublanguage

Error Model annex declarations

- Reusable annex library declarations
  - Error Types and Type Sets (V1 Error Model Types & Propagations)
  - Error Type Mappings (V1 In and Out Guards)
  - Error Behavior State Machines (V1 Error Model Implementations)
- Component specific annex subclauses
  - Error Propagations & Flows (V1 Propagations & In and Out Guards)
  - Component Error Behavior (V1 Abstract error model)
  - Composite Error Behavior (V1 Derived error states)
Outline

• Overview & Background
• Basic concepts illustrated
• Error Types
• Error Propagation
• Component Error Behavior
• Hierarchical composition of error behavior
• Error Behavior and Fault Management
• Dual Redundancy and Modes
• Error Behavior in Layered Architectures
• Summary
A Simple Error Model Illustrated

Collection of error types, used to specify error propagations and flows to support fault impact analysis such as Failure Mode and Effects Analysis (FMEA).

- Error propagations can represent hazards
- The propagation paths are determined by the architecture.
- Components can be the source or sink of error propagations, or pass errors on or transform them into different types of errors.
An Error Propagation View

Components can have incoming and outgoing error propagations.

Error propagations follow propagation paths determined by connections and bindings in the architecture or undocumented observable paths.

Error flows specify flow of errors through a component, or the component acting as propagation source or sink.
Component Error Behavior Specification

Components can have error behavior specified by an error behavior state machine

*Transitions* between states triggered by *error events* and *incoming propagations*.

Conditions for *outgoing propagations* are specified in terms of the *current state* and *incoming propagations*.

Detection of error states and incoming propagations is mapped into a reporting port in the system architecture model.

---

![Component Diagram]

- **Error propagation**
- **Error event**
- **Color: Different types of error**
- **Port/access point**
- **Error flow**
- **Propagation path**
- **Detection**
- **Detection report**
- **Binding**
- **Recover/repair event**
Compositional Abstraction of Fault Model

- Abstracted error behavior of FGS and subcomponents
  - Error behavior and propagation specification

- Composite error behavior specification of FGS
  - State in terms of subcomponent states

\[
[1 \text{ ormore}(\text{FG1.Failed or AP1.Failed}) \text{ and } 1 \text{ ormore}(\text{FG2.Failed or AP2.Failed}) \text{ or AC.Failed}] \rightarrow \text{Failed}
\]
Defining Error Types and Propagations

Error types are defined in the Error Model library

- **Error types**
- NoValue: *type*;
- BadValue: *type*;
- EarlyValue: *type*;
- LateValue: *type*;

Error propagations are defined for ports (features) and bindings of individual components

- **Inport1**: in propagation \{NoValue, BadValue\};
- **OutPort2**: out propagation \{NoValue, BadValue, LateValue\};

Error flows are specified between incoming and outgoing ports of components

- **Flow1**: error source Outport2 \{LateValue\};
- **Flow2**: error path Inport1\{NoValue, BadValue\} -> Outport2 ;

These types are independent. They can occur simultaneously.
Defining Error Behavior as State Machine

Error behavior state machines
- Defined in the Error Model library
- They define error/repair/recover events
  - BadValueEvent: error event;
  - NoValueEvent: error event;
  - LateValueEvent: error event;
- They define states and transitions
  - Operational: initial state;
  - BadValueState: state;
  - NoValueState: state;
  - LateValueState: state;
  - BadValueTransition: Operational -[BadValueEvent]-> BadValueState;
  - NoValueTransition: Operational -[NoValueEvent]-> NoValueState;
  - LateTransition: Operational -[LateValueEvent]-> LateValueState;

The error states are mutually exclusive. The occurrence of BadValue and LateValue has to be modeled by a separate state. Transitions occur one error event at a time.
Defining Component Error Behavior

Defined by a *Component Behavior* subclause on component types and implementations

- Specify conditions for outgoing propagations in terms of current state and incoming propagations
  - **Propagations**
    - BadValueState → Outport2{BadValue};
    - NoValueState → Outport2{NoValue};
    - LateValueState → Outport2{LateValue};

- Specify conditions for transitions in terms of incoming propagations & events
  - **Transitions**
    - Operational -[inport1{BadValue}]→ BadValueState;
    - Operational -[inport1{NoValue} or processor{NoService}]→ NoValueState;

- Specify error behavior conditions that are detected by the system/component and reported as event
  - **Detections**
    - NoValueState → self.DetectedNoValue;
Use of Error Types on Events and States

Think of state machine with tokens of different types

- Error types can be placed into a type hierarchy
- Error events generate events of different types
- Event type becomes type of state
- Propagation type (trigger condition type) becomes type of state

Use of independent error types can be listed as type sets, as was done for propagation specifications of the initial simple model.

Placing types into the same type hierarchy allows us to specify that combinations of error cannot occur.

Operational

Failed

In propagation:
- Inport1
- processor

Out propagation:
- Outport2

Error event:
- BadValue : type;
- NoValue : type;
- LateValue : type;
- NoService : type;
Typed Error Behavior State Machine

Compact state machine representation

- Types can be placed into type hierarchy
- Avoids replication of events, states, propagations for each error type
- Properties can be specified for each error type of an event

```
MyEvent: error event {MyType};
Operational: initial state;
Failed: state {MyType};
EventTransition: Operational -[MyEvent]-> Failed;
```

```
MyType: type;
NoValue: type extends MyType;
BadValue: type extends MyType;
LateValue: type extends MyType;
```

- **Transitions**
  - Operational-[Inport1{BadValue}]->Failed;
  - Operational-[Inport1{NoValue} or processor{NoService}]-> Failed{NoValue};
- **Propagations**
  - Failed{MyType}->[OutPort2;
Error Event Handling in Error States

Transition for all events in every state
- Implicit loopback vs. explicit specification

Event & in propagation coverage
- Default interpretation: stay in state if no outgoing transition triggering
- Potential for unhandled error event types

Diagram:
- Operational
- BadValueState, NoValueState, LateValueState
- Inport1: In propagation BadValue, NoValue
- processor: In propagation: NoService
- Outport2: out propagation BadValue, LateValue
- NoValueEvent
- LateValueEvent
Outline

• Overview & Background
• Basic concepts illustrated
• Error Types
• Error Propagation
• Component Error Behavior
• Hierarchical composition of error behavior
• Error Behavior and Fault Management
• Dual Redundancy and Modes
• Error Behavior in Layered Architectures
• Summary
Error Type Definitions & Type Hierarchy

Error types are defined as reusable elements in the Error Model library
  - NoPower: type;

Error types can be defined as subtypes of an existing error type
  - Single inheritance type hierarchy
  - Subtypes are mutually exclusive alternatives of a given error type
  - Subtype is acceptable match for super type

Example ServiceError
  - Service error is either service omission or service commission
Hierarchies of Independent Error Types

Independent error type hierarchies

- Below are some canonical error type hierarchies

Error types from different hierarchies can occur simultaneously

Additional error type hierarchies
Concurrency (Race Condition, Deadlock, Starvation)

Parameters to types
- Range property for OutOfRange
- Rate for Rate related
- Bound (k) for Bounded Item
- Sequence Omission and Bounded Item Omission Interval
Error Types & Error Type Sets

Error type declarations

ServiceError: type ;
Omission: type extends ServiceError;
Commission: type extends ServiceError;
Early: type extends TimingError ;
Late: type extends TimingError ;

An error type set represents the combination of error types that can occur or be propagated simultaneously.

• An error type set is defined as the product of error types.
• Example: an error propagation may involve both a late value and an incorrect value.

InputOutputError : type set {TimingError, ValueError};
StreamError : type set {TimingError, ValueError, SequenceError, RateError};

An error tuple represents a typed token instance

• Represents actual event, propagation, or state types

{LateValue + BadValue} or {LateValue}
Error Type Sets

Specification of error type (sets) on events, states, propagations

- **Error types**
- **ValueError:** \texttt{type};
- **TimingError:** \texttt{type};
- **NoValue:** \texttt{type extends} ValueError;
- **BadValue:** \texttt{type extends} ValueError;
- **EarlyValue:** \texttt{type extends} TimingError;
- **LateValue:** \texttt{type extends} TimingError;
- **ETS1:** \texttt{type set} \{ValueError, TimingError\};
- **E1:** \texttt{error event} \{ETS1\};
- **S:** \texttt{state} \{ValueError, TimingError\};
- **Trans1:** S \([-\{E1\}]> S1\);
- **Trans2:** S\{ValueError, EarlyValue\} \([-\{ETS1\}]> S2\);

**Error Type Set as Constraint**
\{T1, T2\} power set of two error type hierarchies
\{T1\} error type set of one type
\{T1+T2\} product type (one error type from each error type hierarchy)
\{T1,\} error type set with at least one element from type (hierarchy) T1
\{NoError\} represents the empty set

**Error type set: Powerset of types from two type hierarchies**

- Reference to named error type set
- Declaration of equivalent error type set constructor
- Transition triggered by E1

Constrain transition to \{EarlyValue\} \{BadValue\} \{NoValue\} or \{EarlyValue+BadValue\} \{EarlyValue+NoValue\} tuples of S
Formalized Error Type Specification

Service as sequence of service items

- Service S delivered by a system with a single user can be defined in terms of a sequence of service items, \( s_i, i = 1, 2, \ldots \) each characterized by a tuple \( vs_i, ts_i \) where \( vs_i \) is the value or content of service item \( s_i \) and \( ts_i \) is the time interval or instant of observation of service item \( s_i \).

- A service item is defined to be correct, i.e., have no error, iff: \( (vs_i \in SV_i) \land (ts_i \in ST_i) \) where \( SV_i \) and \( ST_i \) are respectively the correct sets of values and times for service item \( s_i \). \( SV \) represents expected range of values throughout the service and \( ST \) represent the expected duration of the service.

- Definition of service errors in the value and time domains as perceived by an omniscient observer of that service.
Service Related Error Specification

Service errors with respect to the service as a whole rather than individual service items

- **Service Omission** is perceived as a permanent fault in that no service items are provided after the point of failure.  
  \[ \forall j: (s_j \in S' \land s_j \in S \land ts_j = \infty) \]

- **Service Commission** is perceived as an impromptu service in that service items are provided before the point service is expected.  
  \[ \forall j: (s_j \in S' \land s_j \not\in S \land ts_j \in ST) \]

- Other forms of service error can be defined: early service start, late service start, early service termination, late service termination.

Value errors with respect to the value of an individual service item

- **Out Of Range** error indicating that the value is outside the expected range of values, a detectable error.  
  \[ s_i: vs_i \notin SV \]
Fault Mitigation Behavior

Based on these categories of error types we can specify desirable and undesirable fault and fault mitigation behavior. For example, we can define semi-consistent replicate value error behavior as Inconsistent Replicate Value error, where the value error of the replicate is a Corrupt Value error. The detection of the corrupt value by the recipient of the replicates ensures that all the non-corrupt values are identical with respect to the deployed error detection code.

- We can specify bounded omission behavior in that a system may omit some service items (Item Omission error), but if more than k items are omitted then all further items are omitted (Service Omission error).
- *Fail-silent Bounded Item Omission mitigation:* \[ \forall i, (ts_i \in ST_i) \lor \forall j \geq i, ts_j = \infty \lor \lor ((ts_i = \infty) \land (\exists j \in \{i + 1, i + k\}), ts_j \in ST_j) \]
Fail Silent Behavior

We can specify fail-silent behavior of a system and delivery mechanism by assuming that the delivery mechanism can only introduce semi-consistent replicate value errors and that the system producing the service items delivers correct values. This leads to the following fail-silent behavior with respect to the values:

- **Fail-Silent V<sub>FS</sub>**
  \[
  \forall i, \left( \forall u \in \{1, n\}, (v_{si} \in SV_i) \land ((v_{si}(u) = v_{si}) \lor \lor (v_{si}(u) \notin CV_i)) \right)
  \]

With respect to time a system service always produces service items on time or stops producing service items. Similarly, the delivery mechanism is assumed to deliver the service items with consistent fixed propagation delays or stops delivering them.

- **Fail-Silent T<sub>FS</sub>**
  \[
  \forall i, \left( (\forall j \geq i, \forall u \in \{1, n\}, t_{sj}(u) = \infty) \lor \lor (\forall u, v \in \{1, n\}, (t_{si}(u) \in ST_i(u)) \land (|t_{si}(u) - t_{si}(v)| \leq \Delta)) \right)
  \]
Error Model Libraries & Namespaces

Error Model libraries and AADL Packages

- An AADL package can contain one Error Model library declaration
- The Error Model library is identified and referenced by the package name

Error Model library represents a namespace for error types and type sets

- Error type and type set names must be unique within an Error Model library
- An Error Model library can contain multiple error type hierarchies
Extending the Error Type Hierarchy

Extend the error type hierarchy
• By adding subtypes anywhere in the type hierarchy

Accomplished by extending an Error Model library
• The library inherits all error types and type sets from the library being extended

Domain-specific aliases for error types
• Alias and aliased error type are considered equivalent

```plaintext
package MyErrorTypes
public
annex Error_Model {**
error types extends ErrorTypes with
  MissingData renames type ItemOmission ;
  ExtraData renames type ItemCommission ;
  WrongValue: type extends IncorrectValue;
  EstimatedValue: type extends IncorrectValue;
end types;
**};
end MyErrorTypes;
```
Matching of Error Types

Matching of error types

• Subtype acceptable match for a given super type
• Alias is an acceptable match for type
• Need for equivalence of error types in independent type hierarchies?
  – NoPower and NoService are declared alias of Omission
    • Either can be a match for Omission
    • Can they be matches of each other? yes
  – NoPower and NoService are declared subtypes of Omission
    • Either can be a match for Omission
    • Can they be matches of each other? no
Outline

• Overview & Background
• Basic concepts illustrated
• Error Types
• Error Propagation
• Component Error Behavior
• Hierarchical composition of error behavior
• Error Behavior and Fault Management
• Dual Redundancy and Modes
• Error Behavior in Layered Architectures
• Summary
Component Error Propagation

Error Flow:
Path P1.NoData -> P3.NoData
Source P2.BadData;
Path processor.NoResource -> P2.NoData

Incoming/Assumed
- Propagated errors
- Errors not propagated

Outgoing/Intention
- Propagated errors
- Errors not propagated

Bound resources
- Propagated errors
- Errors not propagated
- Propagation to resource

"Not" indicates that this error type is not intended to be propagated.
This allows us to determine whether propagation specification is complete.

Error propagation specification without error behavior state machine is possible.
Error Propagation

Error propagation is component-specific

- through component features & bindings
- declared in Error Model subclauses attached to component types and implementations
- The propagations for P1 and P3 specify unnamed type sets

system Subsystem

features

P1: in data port; P2: in data port; P3: out data port;

annex Error_Model {**

  error propagations

  use types ErrorTypes;

  P1: in propagation {NoData, ValueError};
  P2: in propagation {NoData};
  P2: not in propagation {BadValue};
  P3: out propagation {NoData, BadValue};
  P3: not out propagation {LateData};

processor: in propagation {NoResource};

end propagations; **};
Imported Error Model Libraries

Imported Error Model libraries make error types and type sets accessible

- The namespace of an error model library is made visible by a \texttt{use types}
  - Types and type sets can be referred to without package name qualification
- Error types and type sets from multiple libraries can be made visible
  - Conflicting imported references must be qualified
  - A locally defined alias can be defined for a conflicting reference
Fault Propagation & Transformation Calculus

Default mapping
- Any to all

Single in port to multiple out ports
- late → (value, *, late)

Multiple in ports
- (late, _) → (value, late) : wildcard
- (late, f) → (f, late) : variable

Overlapping rules
- (*, late) → (*, *)
- (*, f) → (*, f)

Conditional mappings (considered too complex)
- (f, g) → late, if f = late
- * , if f = value and g = value
- value, if f = g = value

Fault mapping rules
- * → late (source)
- early → * (sink)
- omission → omission (propagate)
- late → value (transform)
Error Flows

Error flow specifies the role of a component in error propagation

- The component may be a source or sink of a propagated error types
- The component may pass incoming types through as outgoing types
- The component may transform an incoming type into a different outgoing type
- By default all incoming errors of any feature flows to all outgoing features
- If flows are specified in the core model error flows may occur along those flows

Flows

F1: flow path p1 -> p3;

annex Error_Model {**
  error propagations
  ...
  flows
    es1: error source P3{BadData} ;
    es2: error source P3{NoData} ;
    es3: error sink P2{NoData};
    ep1: error path P2{BadData}->P3;
    ep2: error path processor -> P3 mapping ErrorModelLibrary::MyMapping;
  end propagations ; **};

The same propagation may be part of a flow source/sink and flow path.
A propagation may be a sink for one type and not for another
A path may be specified by naming a flow spec or by specifying the source and destination feature.

type mappings MyMapping
use types ErrorModelLibrary;
{BadData} -> {NoData} ;
{NoService} -> {NoData} ;
end mappings;
Interaction Composition

Along architecture propagation paths
Port and access connections

Along observable propagation paths
Propagation points and paths not explicitly represented in the architecture model

© 2011 Carnegie Mellon University
Propagation Paths are Determined by The Architecture

- a processor to every thread bound to that processor
- a processor to every connection routed through that processor
- a memory to every software component bound to that memory
- a memory to every connection routed through that memory
- a bus to every connection routed through that bus
- a device to every connection routed through that device
- a component to each of its required and provided subcomponents
- a component to everything that requires or provides it
- a component to every connection from any of its out features
- a connection to every component having an in feature to which it connects
- a subcomponent to every other subcomponent of the same process
- a process to 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
- a connection to 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
- an event connection to every mode transition that is labeled with an in event port that is a destination of that connection
Observable Error Propagation Points and Paths

Propagation points and paths not explicitly represented in the architecture model

Example: Co-located processors that are not connected via any bus, but affect each other through heat dissipation.
Consistency in Interaction Composition

Mismatched fault propagation and masking assumptions
Robustness to unexpected fault propagation
Propagation Between Hardware Components

Device, system, and bus have been used to model physical resources such as electrical power.

Access keyword for bus or data subcomponent as access connection end.
Two-way Propagation Along Deployment Path

Bi-directional error propagation between HW and SW

Propagation from Processor
- NoResource (Omission)
- DeadlineMiss (Late) resource

Propagation to Processor
- ETOverrun (WCET)
  - Dispatch rate overrun

NoData → Component A → P1
BadData → P1
ETOverrun → Processor
NoResource → Processor
MissedDeadline → Processor

BadData → P2
LateData → P2
NoData → Processor
ETOverrun → Processor
NoResource → Processor
MissedDeadline → Processor

No enforcement of WCET
Deployment Composition

Deployment of components to processors and virtual processors

- Processors and virtual processors are shared resources
- Propagation can be both ways
- Component error behavior specifies impact of binding propagation

Deployment of connections to transfer mechanisms

- Represented by bus, virtual bus, device, system
- Type transformation rules specify impact of binding propagation
Error Propagations Impacting Connections

• No Service: Network failed resulting in no data delivered
  • May be due to network failure or propagated error (e.g. power failure)

• Late Delivery: results in late data
  • On time data may become late (or data may have been sent late)

• Dropped Item: results in missing data sequence element
  • Network protocol may mask this error but it may result in late delivery

• Corrupted Item: data content may get corrupted by transport mechanism
  • Example: stuck bit in hardware
  • Example: compare in dual redundant network cannot identify corrupted replicate value
  • Tactics: detection and masking, fail-silent, propagation
Determining the Result Type

Default result rule for error type sets

- Contributor overrides source for common element type hierarchy
  - Union of error types for non-overlapping element types
- Multiple contributors applied in order
  - Inp1 and Inp2

By transformation rules

Explicit result tuple

- s{NoValue} –[inP1{ValueError}]-> OutP{NoValue}
- all -[inP1{BadValue}]-> ErrorPort{NoValue}
Use of Error Type Transformations

Acceptable mappings between error types

- Applied to connections
  - Source is the connection source
  - Contributor is binding propagation
  - Target is the connection destination

- Applied to error behavior state machines
  - Typed states, events, propagations

\[
\begin{align*}
{\text{MyType}} & \rightarrow {\text{NoValue}} \\
{\text{MyType}} & \rightarrow {\text{NoService}} \\
{\text{NoValue}} & \rightarrow {{\text{(BadValue, LateValue)}}} \rightarrow {\text{NoValue}} \\
{{\text{(BadValue, LateValue)}}} & \rightarrow {\text{BadValue}} \\
{\text{BadValue}} & \rightarrow {\text{LateValue}} \\
{\text{LateValue}} & \rightarrow {\text{LateValue}}
\end{align*}
\]

Type mapping and transformation rules

- Per error type
- Tuple based
Type Mappings and Transformations

Element type mapping and transformation

• Separate rules for each element type
• Mapping/transformation within the same type hierarchy
  • BadValue -> NoValue
  • BadValue-[NoValue]-> NoValue

Type tuple mapping and transformation

• The left hand type set constraint must match
• Mapping/transformation to any result tuple
  • {BadValue} -> {NoValue}
  • {ValueError, TimingError} -[{NoValue}]-> {NoValue}

Handling of corner cases

• Source is {NoError}: default is target = contributor
  – Example: state without type
• Contributor is {NoError}: default is target = source
  – Example: no network error propagation
• Target as Mask: target is unaffected by contributor

Overlap in mapping or transformation rules

• Only if same result type
• We do not assume ordering and overriding rules (as was done in York U. FPTC)
Properties on Propagations and Flows

Stochastic propagation behavior

• Occurrence probability & distribution on outgoing propagations
  – On flow sources to represent probability of an error event of a given type within the component resulting in a propagation
    • Occurrence probability can be specified for each error type
  – On flow sinks to represent the probability of an incoming propagation being masked
    • The same incoming propagation can also participate in a flow path for the same type or different types
    • Propagations that are not masked follow any flow path from the same incoming feature; if none are specified we have any to all flow paths
  – On flow path to represent probability of an incoming propagation of a specific type (or all types) following a given path
    • Different error types may have different Occurrence values
Propagation Probability and Flows

We have several kinds of probabilities

- **Probability of event occurrence**
  - Applied to flow sources to indicate the rate it generates events
  - Applied to out propagations if no flows are specified to represent the rate of out propagation
    - This represents the aggregate of flow source and paths

- **Probability of error propagation**
  - Applied to flow sinks to indicate the chance an incoming propagation is masked
    - The remaining errors are propagated on all paths
  - Applied to flow path to indicate the probability the error is propagated on the particular path
    - This allows different paths to have different propagation probabilities
Properties on Propagations, Events, Components

Hazards

• Severity: criticality level
• Description
• Likelihood
• Risk

Fault/failure

• Occurrence probability, distribution
• Persistency
• Duration
• Failure condition
Outline

• Overview & Background
• Basic concepts illustrated
• Error Types
• Error Propagation
• Component Error Behavior
• Hierarchical composition of error behavior
• Error Behavior and Fault Management
• Dual Redundancy and Modes
• Error Behavior in Layered Architectures
• Summary
Error Behavior State Machines

Reusable error behavior state machine (EBSM)

- Working, non-working states
- Named transitions with trigger conditions
- Destination branches for transitions with branch probabilities
- Error, recovery and repair events
- EBSMs can be refined through \textit{extends}
Simple Error Behavior State Machine

annex Error_Model {**
  error behavior Example
  events -- both events will have mode-specific occurrence values for powered,unpowered
    SelfCheckedFault: error event;
    UncoveredFault: error event;
    SelfRepair: recover event;
    Fix: repair event;
  states
    Operational: initial state ;
    FailStopped: state;
    FailTransient: state;
    FailUnknown: state;
  transitions
    SelfFail: Operational -> [SelfCheckedFault] FailStopped ** with 0.7, FailTransient ** with 0.3);
    Recover: FailTransient -> [SelfRepair] Operational;
    UncoveredFail: Operational -> [UncoveredFault] FailUnknown;
end behavior;
Error Behavior State Machine with Type Sets

annex Error_Model {**

error behavior Example

use types TypesLibrary;

events -- both events will have mode-specific occurrence values for powered,unpowered

    SelfCheckedFault: error event {(BITFault, NoServiceFault)};
    UncoveredFault: error event {UndetectedFault};

SelfRepair: recover event;

Fix: repair event;

states

    Operational: initial state ;
    FailStopped: state {BITFault, NoServiceFault};
    FailTransient: state {BITFault};
    FailUnknown: state {UndetectedFault} ;

transitions

    BITFail: Operational -[SelfCheckedFault{BITFault}]-> {FailStopped with 0.7, FailTransient with 0.3};
    NoServiceFail: Operational -[SelfCheckedFault{NoServiceFault}]-> FailStopped ;
    NoServiceStop: FailStopped -[SelfCheckedFault{NoServiceFault}]-> FailStopped {NoServiceFault};
    Recover: FailTransient -[SelfRepair]-> Operational;
    UncoveredFail: Operational -[UncoveredFault]-> FailUnknown;

end behavior;

Typed error events and error states.
The error type of the event is passed on to the state. Independent errors are held as type set.

BITFault: type;
StuckBit: type extends BITFault;
BadBlock: type extends BITFault;
NoServiceFault: type;
UndetectedFault: type;

Error events as transition condition without an error type indicator implies all types specified in the event declaration. The type of the target state is inferred from the event type when not explicitly specified.
Working and Non-working States

annex Error_Model {{
  error behavior Example
  events -- both events will have mode-specific occurrence values for powered,unpowered
    SelfCheckedFault: error event { BITFault, UncoveredFault};
    SelfRepair: recover event;
    Fix: repair event;
  states
    Operational: initial state;
    FailStopped: state;
    FailTransient: state;
    FailUnknown: state;
  transitions
    Fail: Operational -[SelfCheckedFault]-> {FailStopped with 0.7, FailTransient with 0.3};
    Recover: FailTransient –{SelfRepair}-> Operational;
    UncoveredFail: Operational -[UncoveredFault]-> FailUnknown;
  properties
    StateKind => working applies to Operational;
    StateKind => nonworking applies to FailStopped, FailTransient, FailUnknown;
end behavior;}}

StateKind property allows for consistency checking: recover/repair events cannot cause transition to non-working. Error events cannot cause transition to working.
Component Error Behavior Specification

Component-specific behavior specification

- Identifies an error behavior state machine
- Specifies transition trigger conditions in terms of incoming propagated errors or working condition of connected component
- Specifies propagation conditions for outgoing propagated errors in terms of states & incoming propagated errors
- Specifies detection events, i.e., conditions under which an event is raised in the component (core AADL model)

component error behavior

use types MyErrorLibrary;
use behavior MyErrorLibrary::SWStateBehavior;

transitions
  Operational-[port1{BadData}]->FailTransient;
  FailStopped-[port1{BadData}]->Mask;

propagations
  all –[2 ormore (Port1{BadData}, Port2{BadData},Port3{BadData})] -> Outport3{BadData};

detections
  FailedState –[] -> Self.Failed;

properties
  Occurrence => 0.0005 Poisson applies to BadDataFault;

end behavior;
Transition Conditions

• Failed{BadValue} – [condition expression] -> targetstate{tuple} | mask
  – For a typed source state a error type tuple constraint may be specified
  – The condition expression specifies under which incoming error propagation conditions or error behavior events the transition will occur
    • Disjunction (or) of alternatives (and) of events, propagations, or orless/ormore expressions
  – mask is an explicit specification that the component error state is not affected by incoming error propagations
    • The incoming error propagation may still affect outgoing error propagations as specified in a propagation condition
  – Multiple transition declarations for the same source and target state represent alternative conditions
  – The error type tuple of the target state
    • is optionally specified
    • Derived from the source type tuple and condition contributor tuple
      – By default rule or type transformation specification
Transition Trigger Conditions - 2

Error types in states and in conditions

- Single event or incoming propagation as condition
  - Pass through of same type (default)

  operational –[SelfCheckedFault]–> Failed;

  All –[port1{ValueError}]–> Failed;

- Multiple incoming propagations & state
  - Explicit specification of resulting error type being assigned to state

  Operational –[port1{NoData} and Port2{NoData}]–> Failed{NoData};

  Operational –[port1{BadData} or Port2{BadData}]–> Failed{BadData};

  ailTransient {BadData} –[condition on incoming error propagations ]–> mask;

Conjunctions on incoming propagations

Inport1{LateValue} vs. Inport1{LateValue} and inport2{EarlyValue} vs. Inport1{LateValue} and inport2{noerror}

noerror means that no error propagation occurs at a given time on a feature
Conditional Error Transitions

Uniqueness of triggered transition for multiple outgoing transitions

- Require uniqueness vs. condition ordering
- Atomicity of error events and error propagations
  - Example: error event triggers transition to Failed, while incoming error propagation transitions to TransientFailed
  - Event triggered transition specified with the transition requires that transition conditions for the same transition do not include the event
  - Different events are considered to occur at distinct times or atomic processing order is non-deterministic

Uniqueness of trigger (event or transition condition)

- Multiple events and/or transition conditions triggering the same transition
- Type sets allow combination of types to be assigned to a state
- Conflicting types within one type hierarchy may be resolved by type mapping rules or the triggers are processed atomically with events specified with transition taking precedence
Transient Errors

In error behavior state machine

- Probability that error event results in transient error behavior
  - Branching transition with branch probability
- Error state with transition triggered by recover event
  - Recover event has property to indicate time (range) and distribution over time (range)
    - Distribution over single value vs. range

Need for explicit characterization of propagation as transient?

- Transient nature of propagation in impact analysis
- Inferred from state machine vs. specification w/o state machine as part of error source specification
Presence & Absence of Incoming Propagations

Conjunctons on incoming propagations with single type

- Inport1{LateValue} vs. Inport1{LateValue} and inport2{EarlyValue} vs. Inport1{LateValue} and inport2{noerror}
- noerror means that no error propagation occurs at a given time on a feature

Conditions on incoming propagation type set tuples

- Inport1{LateValue} vs. Inport1{LateValue,BadValue}
- Tuple with just a LateValue means that no other error types have occurred
Outgoing Error Propagation Conditions

Specification of a condition under which the outgoing propagation will occur

- In V1 this was one of the roles of Guard_Out
- The condition can be based on the error behavior state and on incoming error propagations or absence of a propagation (noerror)
- The condition may be solely based on an error behavior state
- The condition may be solely based on incoming error propagations
- Operators for boolean condition are: and, or, ormore, orless
- Mask on individual features (guard_out mask applied to features)
Outgoing Error Propagation Conditions - 2

Error types in states and in conditions

• Single event or incoming propagation as condition

   FailedState{ValueError}-[]-> Port1 ; -- type inferred from state type
   all |-[Inport{BadData}]-> Port1{BadValue} ; -- type mapped into propagated type

• Multiple incoming propagations & state

   Failed{NoData}-[port1{NoData} and Port2{NoData}]-> Port1{NoData} ;
   Operational -[ port1{BadData} or Port2{BadData} ]->Port1{BadData} ;

• Complex conditions

   Operational-[Port1{noerror} and Port2{BadData} or Port1{BadData} and Port2{noerror}]->
   Outport1{BadData} ;
   all-[2 orless (Port2{noerror}, Port1{noerror}, Port3{noerror})]-> Voter{BadData} ;
   All-[2 ormore (Port1{BadData}, Port2{BadData},Port3{BadData}) ]->Outport3{BadData} ;

• Example of a port specific masking condition

   Operational -[(Port1{BadData} or Port2{BadData}) ]-> Outport2{mask} ;
Completeness and Consistency Checking

Coverage of error types in error propagation specification

- Utilizes error propagation and contained error specification
- Robustness and unintended error propagations

Consistency between error propagation, contained error, and error flow specifications and the error propagation conditions

- Interactions between source/sink, contained error declarations (not propagation), noerror, and mask
Error Detection and Error Recovery

Error detection and reporting

- Example: novalue as detectable persistent fault
- Self detection and reporting (internal to component)
- Detection of error propagation by recipient component

Error recovery with probability of success and failure

- Branch transitions

Component internal detection and reporting

```plaintext
detections
PersistentFaultState[]-> ErrorOutPort {NoValue} ;
```

External detection and reporting

```plaintext
detections
all -[Inport1{NoValue}]-> ErrorOutPort {NoValue} ;
```
Modeling of Repair Actions

Interaction with health/fault management architecture

• Specification of error detection in system model
• Identification of repair actions in error model

Represent different repair behavior

• Traceability to repair action in actual system architecture
• Failure of repair action
• Repair action with duration
• Repair agent as shared resource
• Repair parts as consumable resource
Error Detection

In V1 this was one of the roles of Guard_Event

- Specification of a condition under which a failure (error) is detected by the system and reported as event
- The condition may be a single error state or an incoming error propagation or a logical condition of combinations
  - The component detects that it is in an error state
  - The component detects when it receives error propagations
  - The detected error tuple can be reported through an event data port as value

All – [Port1{NoData} and Port2{NoData}] --> ErrorOutEventDataPort{NoData};
FailedState – [] --> Self.Failed;

Error detection allows diagnostics to be performed based on failures actually observed by a system component.
Outline

• Overview & Background
• Error Types and Error Propagation
• Component Error Behavior
  • Hierarchical composition of error behavior
• Error Behavior and Fault Management
• Dual Redundancy and Modes
• Error Behavior in Layered Architectures
• Summary
Hierarchical Modeling

A subsystem of components may have an explicitly associated error model.

The user may declare whether a subsystem error model

1. has a state determined by a user-specified function of the error states of the components (e.g. to model internal redundancy)

2. is an abstract error model to be substituted for the composition of the component models (e.g. to improve tractability of analysis)

The annex supports abstraction and mixed fidelity modeling.
Composite Error Behavior Specification

Composite error behavior specification for a component

- error behavior model in terms of the subcomponent error models
- Specifies conditions under which a composite state is the current state expressed in terms of the (current) error behavior state of its subcomponents.
- The logical expression has the operators or, and, ormore, orless.
- Must be consistent with abstracted error behavior specification

```
composite error behavior
use behavior qualified_error_state_machine_reference
composite states
[ 2 ormore (sub1{Operational}, sub2.Operational, sub3.Operational) ]-> Working ;
[ 1 ormore
end composite;
```

Refer to state vs. refer to working/no-working. Also refer to type in state.
Outline

• Overview & Background
• Error Types and Error Propagation
• Component Error Behavior
• Hierarchical composition of error behavior
• Error Behavior and Fault Management
• Dual Redundancy and Modes
• Error Behavior in Layered Architectures
• Summary
Component & Composite Model for Dual Redundant System

- Dual redundant flight guidance system
  - Redundant Flight Guidance (FG) and Auto Pilot (AP) channel
- Two operational modes
  - Critical: requires both channels active
  - Non-critical: one channel active, operator initiated active channel selection
- Composite error model for FGS
  - Basis for system level reliability analysis
Compositional Abstraction in AADL

- Flight guidance system (FGS) as component type
  - Abstraction describing externally observable interaction points and behavior
- Component implementation of FGS
  - Composition of and interaction between subcomponents
  - Multiple implementations represent variants
Two State Component Error Model

- State machine represents component failure
  - Failure event & Failed state
- Component-specific propagation conditions for FG and AP

```plaintext
Package ErrorModelLibrary
public
annex Error_Model {**
  error types
  NoValue :type;
  end types;
  error behavior Simple
  events
  Failure: error event;
States
  Operational: initial state;
  Failed: state;
Transitions
  BadValueTransition : Operational -[Failure]-> Failed;
  end behavior;
**};
End ErrorModelLibrary;

-- FG component
component error behavior
  use types ErrorModelLibrary;
  use behavior ErrorModelLibrary::Simple;
  propagations
    Failed -[]-> Outport{NoValue};
  end component;

-- AP component
component error behavior
  use types ErrorModelLibrary;
  use behavior ErrorModelLibrary::Simple;
  propagations
    Failed -[]-> Outport{NoValue}
    Operational –[ Inport{NoValue}]-> Outport{NoValue};
  end component;
```
Mode-specific Error Model Logic

• AC error behavior is sensitive to operational modes
  • Fail-stop behavior by not producing output (Omission error) under failure conditions

```plaintext
-- AC component
annex EMV2 {**
  component error behavior
    use types ErrorModelLibrary;
    use behavior ErrorModelLibrary::Simple;
    propagations
      Failed –[]-> OutPort{NoValue};
      Operational –[FromAP1Port{NoValue} or FromAP2Port{NoValue}] -> OutPort{NoValue};
  **} in modes (Critical);

annex EMV2 {**
  component error behavior
    use types ErrorModelLibrary;
    use behavior ErrorModelLibrary::Simple;
    propagations
      OutPort{NoValue} when
      Failed –[]-> OutPort{NoValue};
      Operational –[FromAP1Port{NoValue} and FromAP2Port{NoValue}] -> OutPort{NoValue};
  **} in modes (NonCritical);
```
Compositional Abstraction of Fault Model

• Abstracted error behavior of FGS and subcomponents
  • Error behavior and propagation specification

• Composite error behavior specification of FGS
  • State in terms of subcomponent states
  
  \[
  [1 \text{ ormore}(FG1.Failed, AP1.Failed) \text{ and } 1 \text{ ormore}(FG2.Failed, AP2.Failed) \text{ or AC.Failed}] \rightarrow \text{Failed}
  \]
Composite Error Behavior as Three State Model

- FGS state machine reflects failure in critical mode and non-critical mode
  - States are defined in terms of subcomponent states

```plaintext
-- Annex subclause for FGS system
annex Error_model {**
  composite error behavior
  use types ErrorModelLibrary;
  use behavior ErrorModelLibrary::ThreeState;
  composite states
  [ AP1.Operational and AP2.Operational and
    FG1.Operational and FG2.Operational and AC.Operational ]-> Operational;
  [ AC.Failed or 1 ormore (AP1.Failed, FG1.Failed) and 1 ormore (AP1.Failed, FG2.Failed) ]-> NonCriticalModeFailure;
end composite;
**};
```

annex Error_Model {**
  error behavior ThreeState
  States
  Operational: initial state;
  NonCriticalModeFailure: state;
  CriticalModeFailure: state;
end behavior;
**};
Availability of FGS

- Availability of FGS
  - Probability that FGS is not in Failed state
- Abstract Error Model
  - Failure event transitions to Failed
  - Failure event probability: FGS itself fails
  - Power input failure propagation affects FGS state
  - Input propagation does not affect state but possibly output propagation

Probability of Powered FGS failure $P(\text{FGS.Failed})$

$$P(\text{PowerInput(Failed)}) + P(\text{FGS.Failed})$$

$P(\text{FGS.Failed})$ derived from Composite behavior specification
Outline

- Overview & Background
- Error Types and Error Propagation
- Component Error Behavior
- Hierarchical composition of error behavior
- Error Behavior and Fault Management
- Dual Redundancy and Modes
- Error Behavior in Layered Architectures
- Summary
An Overview of the AADL Error Model Annex

Multi-layered Platforms

Connection Binding to buses, processors, devices, virtual processors, virtual buses

Platform layers exposed via virtual bus and virtual processor

Type transformation rules specify how binding propagations affect /augment nominal communication and error propagation between component A and component B
Composite Error Model of FGS

- Failed state of FGS determined by error states of subcomponents
  - Defined as Composite error behavior for FcnSystem
- Probability of Networked FGS failure $P(\text{NFGS.Failed})$
  - $P(\text{Network.Failed}) + P(\text{FGS.Failed})$
  - $P(\text{FGS.Failed})$ derived from Composite behavior specification
Dual Redundant Switch

• Availability of Abstracted Switch
  • 1 - Probability that Switch is in Failed state
    • \( P(\text{Switch.Failed}) = P(\text{Event.Failure}) + P(\text{InProp.NoPower}) \)

• Abstract Error Model
  • Failure event transitions to Failed
  • Failure event probability: Switch itself fails
  • Power input failure propagation affects Switch state
  • Communicated data does not affect Switch state
Switch Abstraction and Implemented_As

All replication logic is internal to Switch

Communicating components are not aware of redundancy logic

P(Network.f) = P(Nif1.f) + P(b1.f) * P(b2.f) + P(Nif2.f)

We can introduce two failed states for 1 channel failing and both channels failing

Power supply considerations not shown
CRC as Virtual Bus Protocol

- CRC to facilitate data corruption detection
  - Data packaged with CRC (encoder)
  - One or more detection points
  - Unpacking via Decoder

- Layering of CRC
  - Use within a network protocol to handle data corruption (fail-silent, fail-operational)
  - Use by application layer to compensate for corrupt data propagation
Use of CRC Within Switch

CRC encoding part of network protocol and not visible to application
Improving FGS Availability

- Two channel replication
- Probability of FGS failure $P(fgs.f)$
  - $P(fg1/ap1.f) \times P(fg2/ap2.f) + P(AC.f)$
  - $P(fgx/apx.f) = P(fgx.f) + P(apx.f) + P(Networkx.f)$
Improving FGS Availability

- Cross-over configuration
  - Four communication paths
  - Integrity checking on replicate input
Triple Redundant Cross-Checking Example

- Large example from Error Model V1
- Four replicates of subsystem
  - Three active and reconfiguration on failure of one
- Cross checking
  - Exchange outputs
  - Compare other outputs to own
  - If one other different, report its failure
  - If self different, then fail silent

SS1
SS2
SS3
SS4(spare)

System mode logic
**Triple Redundant Cross-Checking Example**

- Error propagation types in O/A/B
  - NoData (Item Omission)
  - BadData (Incorrect Value)
  - Byzantine (Inconsistent Value)
- Error propagation types on A/B_Failed
  - Incorrect Value
    - Item Omission (no event raised)
    - Item Commission (false event raised)
  - Item omission
    - Benign detection of omission (report event/non-event Freshness)
    - Comparison detection (item omission vs. bad value)
- Mode logic (incoming)
  - Item/service omission (No event due to source fail stop)
  - Inconsistent Value (different X-Failed value)
    - Bad X_Failed data from one source (Omission/commission)