Outline

• Why AADL
• AADL Language Overview
• Modeling Embedded Software
• Modeling Computer Systems
• Properties & Patterns
• Large Scale Systems
• Summary
An Overview of AADL V2

Cost & Time Reduction due to Early Fault Discovery

Sources:

Where faults are introduced
Where faults are found
The estimated nominal cost for fault removal
Mismatched Assumptions

System Engineer

- Physical Plant Characteristics
  - Lag, proximity

Control Engineer

- Measurement Units
  - Ariane 4/5
  - Air Canada

System Under Control

- Data Stream Characteristics
  - ETE Latency (F16)
  - State delta (NASA)

Compute Platform

- Distribution & Redundancy
  - Virtualization of HW
    - (ARPA-Net split)

Runtime Architecture

- Concurrency
  - Communication
    - iTunes crashes on dual-cores

Application Software

- Embedded SW System Engineer

Why do system level failures still occur despite fault tolerance techniques being deployed in systems?

SysML does not address Embedded Software System Architecture Issues
What is the AADL?

SAE International Architecture Analysis and Design Language (AADL) is a *standard* architecture modeling language, developed by and for the avionics, aerospace, automotive, and robotics communities.

Uses component-based notation for the *specification* of task and communication *architectures of real-time, embedded, fault-tolerant, secure, safety-critical, software-intensive systems*.

The language & associated tools are used to *model, analyze*, and generate embedded real-time systems

- **Tool-based** analysis in Eclipse framework
- A modeling infrastructure that supports *model-based engineering concepts*
- Based on 15 Years of DARPA funded *research* technologies
- First published Nov 2004 (*V1*) - revised standard Jan 2009 (*V2*)

* SAE International standard document AS 5506A (R)
An Overview of AADL V2

Model-Based Embedded System Engineering

- System Analysis
  - Schedulability
  - Performance
  - Reliability
  - Fault Tolerance
  - Dynamic Configurability

- System Construction
  - AADL Runtime System
  - Application Software Integration

- SAE AADL

- Document the Runtime Architecture
  - Abstract, but Precise

- Application Software
  -• Fault Tolerance
  -• Dynamic Configurability

- External Environment
- Execution Platform
  - GPS
  - DB
  - HTTPS
  - Ada Runtime
  - Devices
  - Memory
  - Bus
  - Processor

© 2010 Carnegie Mellon University
An Overview of AADL V2

Embedded Software System Architecture

- Application SW Runtime Architecture: SW packages running as communicating tasks
- Logical interface between software and physical system
- Physical system/environment Interface with embedded SW/HW
- Computer platform architecture: Processors & networks & runtime systems
- Deployment of software on computer platform
- Physical interface between computer platform and physical system
An Overview of AADL V2

Cooperative Engineering of Systems

Key elements of physical system are captured in AADL as component abstractions & properties relevant to embedded software system analysis.

© 2010 Carnegie Mellon University
Architecture-Centric Modeling Approach

Single Annotated Architecture Model

**Availability & Reliability**
- MTBF
- FMEA
- Hazard analysis

**Security**
- Intrusion
- Integrity
- Confidentiality

**Data Quality**
- Data precision/accuracy
- Temporal correctness
- Confidence

**Resource Consumption**
- Bandwidth
- CPU time
- Power consumption

**Real-time Performance**
- Execution time/Deadline
- Deadlock/starvation
- Latency

**Reduced model validation cost due to single source model**

© 2010 Carnegie Mellon University
Semantically Consistent Architecture & Analysis Concepts

Architecture Meta Model

Error Occurrence & Propagation Behavior

AADL Semantic Model
Meta model & semantic spec

Static SW Architecture
Packages, data, subprograms, abstract components

Runtime Architecture
Processes, threads, connections
Modal runtime configurations

Computer System & Platform
Processor, memory, bus, device system components

Component & Interaction Behavior
Behavior Annex

AADL Offers
• Domain concepts with strong semantics
• XMI-based interchange format
• Extensible domain model

Textual AADL
Graphical AADL
UML Profile via MARTE
Database Schema & Form-based Frontend
Import via XML/XMI interchange format

Safety Analysis
Reliability Analysis
Performance Analysis
Resource Analysis
Data Quality Analysis

Auto-propagation of Changes through regeneration

© 2010 Carnegie Mellon University
An Overview of AADL V2

Benefits of System Architecture Virtual Integration

Virtual System Integration

Low fidelity
Adequate confidence
Requirements Engineering
System Design
Software Architectural Design
Component Software Design

High-level AADL Model
Detailed AADL Model
Specify Model-Code Interfaces
Code Development

→ generation of test cases
← updating models with actual data

→
←

Integration Test
System Test
Acceptance Test
Strong confidence

→
←

Top-Level Verification Items

© 2010 Carnegie Mellon University
What is the AADL (SAE AS-5506A Std)?

A formal modeling language for describing software and hardware system architecture

Based on the component-connector paradigm

Key Elements:

- Core AADL language standard (V2-Jan, 2009, V1-Nov 2004)
  - Textual & graphical, precise semantics, extensible
- AADL Meta model & XMI/XML standard
  - Model interchange & tool interoperability
- Annexes Error Model Annex as standardized extension
  - Error Model Annex addresses fault/reliability modeling, hazard analysis
- UML 2.0 profile for AADL
  - Transition path for UML practitioner community via MARTE
AADL Working Group Annex Activities

Behavior Annex as AADL extension (in ballot)
- Concurrency behavior
- Validation of implementation

ARNIC 653 Annex (in ballot)
- Capture 653 architecture in standard way
- Define 653 architectural elements in AADL for analysis
- Enable generation to 653 O/S

Data Modeling Annex (in ballot)

Code Generation Annex (in review)

Error Model Annex (revision)
Outline

- Why AADL
- AADL Language Overview
- Modeling Embedded Software
- Modeling Computer Systems
- Properties & Patterns
- Large Scale Systems
- Summary
AADL: The Language

Precise execution semantics for components
- Thread, process, data, subprogram, system, processor, memory, bus, device, virtual processor, virtual bus

Continuous control & event response processing
- Data and event flow, synchronous call/return, shared access
- End-to-End flow specifications

Operational modes & fault tolerant configurations
- Modes & mode transition

Modeling of large-scale systems
- Component variants, layered system modeling, packaging, abstract, prototype, parameterized templates, arrays of components and connection patterns

Accommodation of diverse analysis needs
- Extension mechanism, standardized extensions
An Overview of AADL V2

AADL Representation Forms

**AADL Text**

```plaintext
thread data_processing
features
  raw_speed_in: in data port;
  speed_out: out data port;
properties
  Period => 20 ms;
end data_processing;
```

**XML**

```xml
<ownedThreadType name="data_processing">
  <ownedDataPort name="raw_speed_in"/>
  <ownedDataPort name="speed_out" direction="out"/>
  <ownedPropertyAssociation property="Period"
    <ownedValue xsi:type="aadl2:IntegerLiteral"
      value="20" unit="ms">
  </ownedValue>
</ownedPropertyAssociation>
</ownedThreadType>
```

**Graphical**

![Graphical representation of AADL thread]

© 2010 Carnegie Mellon University
AADL Language Elements

- AADL Language Elements
- core modeling
- engineering support
- infrastructure

- Components
  - Interactions
  - Properties

- Abstractions
  - Organization
  - Extensions

© 2010 Carnegie Mellon University
Component-Based Representation

Specifies a well-formed interface

*Component type* allow for multiple implementations with extensions

All external interaction points defined as *features*

Data and event *flows* through component, across multiple components

*Properties* to specify component characteristics

Components organized into system hierarchy

Component interaction declarations must follow system hierarchy
Some System Properties

AADL standard properties for systems include the following

System startup

• **Startup_Deadline** => 0.5s
  
  A property of type `Time` assigned (`=>`) a value of 0.5 seconds. Value is a floating point number with a time unit. Valid units are ps, ns, ms, s, h, m, d, etc.

Time to load programs, data into the system at startup

• **Load_Time** => 0.1s..0.15s
• **Load_Deadline** => 0.2s

  Two values indicating a time interval: Loading takes between 0.1 and 0.15 seconds.
Sample System Type

Throughout this module we create a model of a part of an automotive system to describe data flow and analyze correctness of data types. We start with a top-level system type for the car system.

Textual AADL

```aadl
package carPackage
  public
    system CarSystem
  end CarSystem;
end carPackage;
```

Note: Each declaration must be contained in a package.
Sample System Implementation

We now add a subsystem that handles braking

```plaintext
package carPackage
public
    system CarSystem
    end CarSystem;
    system BrakingSystem
    end BrakingSystem;
    system implementation CarSystem.impl
        subcomponents
            braking: system BrakingSystem;
        end CarSystem.impl;
end carPackage;
```
AADL: Components and Connections

Component type
- identifier
- component category
- prototype
- extends {component_type}
- features
- flow specification
- properties

Component implementation
- identifier
- extends {component implementation}
- refines type
- subcomponents
- connections
- call sequences
- modes
- flow implementation & end-to-end flows
- properties

Properties
- standard
- user defined

Property set
- property types
- property definitions
- property values

Component Category
- application
- platform
- composite

is one of

Package
- public
- component classifier
- private
- component classifier

Connections
- data
- event
- event data
- port group
- access

modes
mode transitions
mode configurations

Version 2

© 2010 Carnegie Mellon University
Outline

- Why AADL
- AADL Language Overview
- Modeling Embedded Software
- Modeling Computer Systems
- Properties & Patterns
- Large Scale Systems
- Summary
Application Software Components

**System** – hierarchical organization of components

**Process** – protected address space

**Thread group** – logical organization of threads

**Thread** – a schedulable unit of concurrent execution

**Data** – potentially sharable data

**Subprogram** – callable unit of sequential code
Process Components

A process component represents a protected virtual address space
- Address space boundaries are by default enforced at run-time
- A property setting allows to disable the protection

Contains executable program and data needed for execution and must be loaded into memory
- Process is stored in ROM
- Process is loaded at system startup
- Process may be unloaded when it is not active

A process must contain at least one thread subcomponent to be executable

Note: An AADL process does not have an implicit thread
Sample Process Properties

Process at run-time

• Runtime_Protection => false;  
  No run-time enforcement of address space protection

• Load_Time => 150ms..300ms;
• Load_Deadline => 500ms;  
  Time to load binary image into memory

• Startup_Execution_Time => 100ms..110ms
• Startup_Deadline => 200ms;  
  Time to start the process after loading, e.g., to create contained threads

Relationship to implementation in a programming language

• Source_Language => “C”;
• Source_Text => “navigation.c”;

© 2010 Carnegie Mellon University 26
Thread Components

A thread component represents a schedulable and executable entity in a system

- Concurrent tasks
- Active objects

Threads execute based on time or thread-external events

- Periodically every 50ms, e.g., a data sampling thread in a control system
- Process a message upon arrival with arbitrary arrival pattern, e.g., a thread in a camera processing image data when the shutter button is pressed

Threads are mapped onto operating system threads for execution

- One or more application threads per OS thread

Interacts with other threads through port connections, subprogram calls, and shared data access

Executes within the virtual address space of its enclosing process
Thread Dispatch Protocols

Periodic thread
• Periodic dispatch of threads, typically with hard deadlines

Aperiodic thread
• Dispatch based on events with arbitrary arrival patterns

Sporadic thread
• Dispatch based on events with a minimal time between dispatches

Background thread
• Dispatch once and execute until completion

Timed thread
• Dispatch based on events, or based on timeout if no events occur

Hybrid
• Dispatch based on events and periodically (combines periodic and aperiodic dispatch protocols)
Thread Properties

Properties related to thread dispatch
- Dispatch_Protocol => periodic;
  - Period => 50ms;

Properties needed for thread scheduling
- Compute_Execution_Time => 45ms..50ms;
- Deadline => 40ms;
- Dispatch_Offset => 5ms;

- One from previous slide (or user-defined)
- Required for periodic, sporadic, timed, and hybrid threads
- The execution time range of this thread – upper bound is worst case execution time
- Optional, defaults to period
- For periodic threads: indicate delayed dispatch relative to other periodic threads
Thread Example

A thread executes periodically to filter navigation data, e.g., from a GPS device.

```
thread NavDataFilter
  features
    rawData: in data port navData.raw;
    filteredData: out data port navData.filtered;
  properties
    Dispatch_Protocol => periodic;
    Period => 50ms;
    Deadline => Period;
    Compute_Execution_Time => 18..20 ms;

end NavDataFilter;
```
Thread States

- Initialize
- Inactive
- Activate
- Deactivate
- Finalize
- Awaiting Dispatch
- Recover
- Compute

Substates:
- Executing normally
- Blocked
- Preempted
- Suspended

State with associated code execution
State without code execution
Thread Fault Handling

Thread errors are classified as recoverable or unrecoverable

- **Recoverable error** (e.g., deadline overrun during computation)
  → Runtime system invokes recovery entry point

- **Unrecoverable error** (e.g., any error during recovery)
  → Thread is aborted

Errors are reported via the thread's error port and can be processed like any other event / message, for example by a fault monitoring component.
Ports and Connections

Ports – interaction points of a component to model directional transfer of data and control. Ports are declared as features in component types.

- **Data port**: non-queued data
- **Event port**: queued signals
- **Event data port**: queued messages

Feature group – aggregation of ports (and other features) into single connection point

Connections – connect ports in the direction of data/control flow; uni- or bi-directional
Some Port Properties

Queuing of events and messages

- **Required_Connection** => true;
- **Queue_Size** => 3;
- **Queue_Processing_Protocol** => FIFO;
- **Overflow_Handling_Protocol** => DropOldest;
- **Dequeue_Protocol** => AllItems;
- **Urgency** => 255;

Default: no connection needed

Handling of incoming event and message queues

To resolve conflicts if several queues are not empty

Frequency of data input and output

- **Input_Rate** => (  
  **Value_Range** => 1.0 .. 1.0;  
  **Rate_Unit** => PerDispatch;  
  **Rate_Distribution** => Fixed;  )

- **Output_Rate**

Mapping to variable in an implementation

- **Source_Name** => "brake_state";
## Port Connections

<table>
<thead>
<tr>
<th>Source Feature</th>
<th>Destination Feature</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data port</td>
<td>Data port</td>
<td>-</td>
</tr>
<tr>
<td>Event data port</td>
<td>Data port</td>
<td>Discard the event</td>
</tr>
<tr>
<td>Data port</td>
<td>Event port</td>
<td>Arrival of data triggers event, data is discarded</td>
</tr>
<tr>
<td>Event port</td>
<td>Event port</td>
<td>-</td>
</tr>
<tr>
<td>Event data port</td>
<td>Event port</td>
<td>Discard the data</td>
</tr>
<tr>
<td>Event data port</td>
<td>Event data port</td>
<td>-</td>
</tr>
<tr>
<td>Data port</td>
<td>Event data port</td>
<td>Arrival of data triggers event, data becomes message content</td>
</tr>
</tbody>
</table>

**Example:**

- In a control loop, a data port D of a data producer is connected to the data port of a consumer – D is also connected to a logging component’s event data port.
- A thread sends messages out of an event data port ED to another thread – ED is connected to a health monitor’s data port to periodically checks if there are new messages sent by sampling the sending port.
Connections Between Ports 1

An AADL port connection connects

- Two ports of subcomponents in the same component implementation ➔ Communication inside a component – identical port directions
- A port of a component implementation with a port of one of its subcomponents ➔ Communication with the outside – complementary port directions
- A port can have multiple outgoing connections (fan-out)
- Data ports can have one, other ports can have multiple incoming connections (fan-in)
- Connections can be bi-directional (<>)}
Port Timing

Input timing
- Data, events, and messages eventually processed by a thread, device, etc.
- Default behavior
  - Contents of ports frozen at thread dispatch time
  - Thread does not see new data/events before next dispatch

Output timing
- Data, events, and messages are produced by a thread, device, etc.
- Default behavior
  - Data available at completion time
  - Events and messages anytime during execution

Default timing can be overridden by properties
Feature Groups

Feature groups are collections of individual features* and nested feature groups such that

- Feature group can be connected as a single unit outside a component
- Individual features can be connected inside a component

* In addition to ports, AADL has access features and parameters. A component can declare that it provides access to a shared subcomponent or that it requires access to a subcomponent shared by another component. Subprogram components can have parameters. A feature group can contain all kinds of features.
Data Components

Data components can represent

- Data shared between several threads or subprograms
- Local data in a thread or subprogram
- The type of data exchanged through data and event data ports
- The type of subprogram parameters

AADL models should contain information about data that is relevant to analyses of the architecture, e.g.,

- Bandwidth analysis – size of data elements, frequency of data exchanges
- Model consistency – size, value ranges, and physical units of exchanged data

Note:

- Use of data components is optional in an architecture
- May be needed for analyses or code generation
- AADL is not a complete data modeling language (→ Data Modeling Annex)
A data component can be shared among several other components

- Data access features – to model required or provided access to a shared data component
- Access connections – to model access paths to the shared data component

Note:

- The data access symbol points away from the shared component. Data flow is indicated by the connection direction.
- Where possible use port connections to express intended data flow.
Data Consistency

Default consistency rules for port connections

- Data ports must have the same data type (if specified at both ends)
- Data implementations must be identical (if specified at both ends)
- Data implementation at the source end must implement the data type at the destination end

Configurable via connection property Classifier_Matching_Rule

- Default value: Classifier_Match (as above)
- Other values: Equivalence, Subset, Conversion

Make use of other properties that allow the architect to specify for pairs of data classifiers that

- Both are identical
- One is a subset of the other
- One is automatically converted into the other via the connection protocol
Flows

Logical flow of data and/or control through a sequence of components and connections.
Support analysis of data flow and control flow
Provide the capability of specifying end-to-end flows to support analysis such as

- End-to-end timing and latency
- Fault propagation
- Resource management based on operational flows
- Security based on information flows
- …
Flow Sources, Paths, Sinks

device BrakePedal
  features
    brake_event: out event data port;
  flows
    FSrc1: flow source brake_event;
end BrakePedal;

system CruiseControl
  features
    brake_event: in event data port;
    throttle_setting: out data port;
  flows
    brake_flow: flow path brake_event -> throttle_setting;
end CruiseControl;

device ThrottleActuator
  features
    throttle_setting: in data port float_type;
  flows
    FSnk1: flow sink throttle_setting;
end ThrottleActuator;
Flow Implementation

Flow through subcomponents and connections
Subcomponent flow in terms of its flow specification

brake_flow: flow path brake_event -> throttle_setting;

brake_flow: flow path brake_event ->
C1 -> data_in.F_di ->
C3 -> control_laws.F_cl ->
C5 -> throttle_setting;
End To End Flow Example

Flow from the brake through the cruise control to the throttle actuator

```
system CarSystem.impl
    subcomponents ...
    flows
        SenseControlActuate: end to end flow
            brake_pedal.FSrc1 -> C1 -> cruise_control.brake_flow -> C2 -> throttle_actuator.FSnk1;

end CarSystem.impl;
```
Subprogram and Subprogram Group

A subprogram component represents executable code
  • Is executed sequentially (concurrent execution expressed with threads)
  • Can be called from a thread or another subprogram
  • Can be called with parameters
  • Does not maintain internal state across calls, but may access shared data

A subprogram group represents a library of subprograms

Subprograms and subprogram groups can be shared across components

Note:
  • Use of subprograms is optional in an architecture
  • May be needed for analyses or code generation
Subprogram Access

A subprogram can be shared between components

- Subprogram access features – to model required or provided access to a shared subprogram
- Access connections – to model the path to the shared subprogram

Note:
- The access symbol points away from the shared subprogram
- Subprogram access connections are bidirectional
Subprogram Calls and Call Sequences

Calls can be to explicit or implicit subprogram instances
Calls to subprograms are organized in call sequences
Call sequences can occur in thread and subprogram implementations
A subprogram call executes the call sequence in the called subprogram once
A thread can have call sequences for initialization, finalization, activation, deactivation, computation, and recovery
Each thread dispatch executes the computation call sequence once
Subprogram calls can be local or remote
  • Local call – the subprogram executes in the context of the calling thread
  • Remote call – the subprogram executes in the context of another thread

Note:
  • Modeling of call sequences is optional
  • Useful as an intermediate representation for code generation
  • To model more complex control flows a language extension must be used (→ Behavior Annex)
Modes and Mode Transitions

Modes represent system configurations
- Subcomponents can be active or inactive in a mode
- Connections can exist in certain modes only
- Property values can depend on the component's mode

Modes can represent software states in threads and subprograms

Mode transitions represent configuration changes as reaction to events
- Triggered through ports (from outside or from a subcomponent)
- Triggered internally by implementation software
- Triggered internally in an execution platform component or a device

Example: In an avionics system, different components are active during different flight phases (takeoff, cruising, autopilot, landing)

Note: Modes are not intended for modeling detailed internal behavior of threads or subprograms (→ AADL Behavior Annex)
Modal Components

A system that can be connected to a fault monitoring component

system DualRedundant
features (three event ports as below)
modes
  nominal: initial mode;
  backup: mode;
  reinit: mode;
  t1: nominal -[Primary_fail]-> backup;
  t2: backup -[Init_restart]-> reinit;
  t3: reinit -[Primary_ok]-> nominal;
end DualRedundant;

© 2010 Carnegie Mellon University
Modes in the Component Hierarchy

Mode of a subcomponent can be *derived* from the mode of the containing component

- Subcomponent modes are declared as required modes, they are provided by the enclosing component
- Subcomponent may not contain mode transitions, subcomponent mode changes are driven by mode changes in the containing component
- Modal subcomponent declaration specifies how container modes map to subcomponent modes
- Mode of the container determines the subcomponent mode

With derived modes it is possible to specify synchronized mode transitions throughout all components in a subsystem
Derived Modes in Detail

system implementation S.impl
  modes
    A initial mode; B mode;
    C mode; D mode; E mode;
    ... mode transitions here ...
  subcomponents
    subsys: system S1 in modes (A => X, B => Y, C => Y, D);
end S.impl;

system S1
  requires modes
    X mode; Y mode; D mode;
end S1;

<table>
<thead>
<tr>
<th>If S.impl is in mode ...</th>
<th>... then subsys is in derived mode ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>None, S1 not active</td>
</tr>
</tbody>
</table>

Subsys active in mode A and its mode is X

Multiple modes can map to the same derived mode

S1 (and implementations) may not declare additional modes or mode transitions

Name mapping optional if names are identical
System Operation Modes

A system typically consists of multiple components that have modes

- Multiple instances of the same modal component
- Instances of different modal components

The overall system state is described by the collection of current modes of all modal components in the system – System Operation Mode (SOM)

The SOM changes whenever the mode of a component in the system changes

- A component receives an event that triggers a mode transition
- An internal mode of a thread or subprogram changes as a result of execution

If an event or message is sent out through a single port and triggers multiple mode transitions, then this is treated as a single SOM transition.

Similarly, transitions of derived modes and parent modes are treated as a single SOM transition.
Mode Transitions and Thread Execution

Upon activation/deactivation the runtime system invokes the activation/deactivation entry point of each thread that is activated or deactivated.

Mode transition timing

- Mode transitions inside threads take place at the next thread dispatch, i.e., the next execution is the thread is in the new mode.
- Mode transitions in hardware components happen immediately.
- Other mode transitions may change the set of active threads and can happen in two ways:
  - Emergency mode transitions happen immediately, deactivated threads are aborted if necessary.
  - Planned mode transitions allow critical threads finish execution and happen when their periods align.
  - A mode transition is marked as an emergency transition via property Mode_Transition_Response.
  - By default, mode transitions are planned.
Mode Transition and Thread Execution

Steps in a planned mode transition

1. Wait until periods of critical periodic threads and devices align
2. Disable connections that are not part of the new SOM
3. Read data that is flows via connections that are marked active during the mode transition
4. Deactivate threads that are not part of the new SOM – invoke their deactivation entrypoints
5. Activate threads that are part of the new SOM – invoke their activation entrypoints
6. Enable connections that are part of the new SOM
7. Wait until periods of critical threads align
8. Continue in the new SOM

Periodic threads and devices are marked as critical setting property Synchronized_Component to true
Outline: AADL Standard & MBE

• Why AADL
• AADL Language Overview
• Modeling Embedded Software
• Modeling Computer Systems
• Properties & Patterns
• Large Scale Systems
• Summary
Execution Platform Components and Devices

**Processor / Virtual Processor** – Provides thread scheduling and execution services

**Memory** – provides storage for data and source code

**Bus / Virtual Bus** – provides physical/logical connectivity between execution platform components

**Device** – interface to external environment
Two Aspects of Execution Platform Components

Represent hardware components with corresponding characteristics

- Processor timing, hardware clock period/jitter
- Bus transmission time, latency
- Memory capacity, access time, RAM/ROM
- Constraints on physical connectivity
- Power consumption
- Weight

Represent logical resources with corresponding characteristics

- Thread scheduling policy of a processor
- Communication protocol over a network connection modeled as a bus
- Transactional characteristics of a database modeled as a memory component

These two aspects are reflected in properties that are applied to the components

AADL provides the virtual processor / bus component categories that represent only the logical resource aspect of a processor / bus
Processor

As a hardware component

- Processors are computer-hardware
  - Include a CPU, memory, bus, etc
  - Include a hardware clock that can interrupt the processor
  - MIPS rating, size, weight

As a logical resource

- Processors schedule threads
  - Implementation of one or more scheduling policies
  - A periodic clock interrupt to drive periodic dispatching
- Processors execute software
  - Software to provide scheduling and other runtime system functionality

Threads are bound to processors for execution

Processors may

- Access memory and device components via buses
- Execute software associated with devices
Some Processor Properties

Logical Resource
Thread Scheduling Properties
• Scheduling_Protocol => RMS;
• Thread_Swap_Execution_Time => 1.0ms;
• Process_Swap_Execution_Time => 1.5ms;
• Thread_Limit => 16;
• Allowed_Dispatch_Protocol => (periodic, aperiodic);

Hardware Component
Clock Properties
• Clock_Period => 10ms;
• Clock_Jitter => 2ms;
Bus

As a hardware component
• A bus provides the physical connection between hardware components
  – Inside a hardware component, e.g., PCI bus in a PC
  – Between hardware components, e.g., a USB connection between a PC and a camera

As a logical resource
• A bus represents the protocol(s) by which connected components communicate

Components are connected to a bus with a bus access connection
A bus is shared by all components that access it
Some Bus Properties

Logical Resource
Constraints on transported content
- Allowed_Connection_Type =>
  (Port_Connection, Data_Access_Connection);
- Allowed_Message_Size => 0B..1KB;

Protocols and protocol properties *(see module 6 for details)*
- Provided_Virtual_Bus_Class => …;
- Provided_Connection_Quality_of_Service =>
  (OrderedDelivery);

Hardware Component
Constraints on physical connectivity
- Allowed_Philysical_Access => (processor, memory);

Properties related to data transmission time
- Transmission_Time
- Latency
A bus component can be shared among components in different subsystems

- Bus access features – to model required or provided access to shared bus
- Bus access connections – to model the path to the shared bus

Note: The bus access symbol points away from the shared component. It does not indicate the direction of data flow.
Memory Components

AADL memory components represent randomly accessible physical storage (e.g. RAM, ROM)

AADL memory may also be used to model complex permanent storage (e.g. disks, database)

Stores binary images of source text (i.e., code and data) and run-time data

Processes must be in memory at startup to be executed

- Stored permanently in ROM
- Loaded into RAM

Processors need access to memory

- Processor and memory are connected via a shared bus
- Memory is contained in the processor
Device Components

AADL device components represent elements that are not decomposed further in a model. Devices are characterized by their interface, their internal structure is not modeled:

- Typically physical components interfacing with the environment
  - Sensors and actuators
  - Standalone complex devices, e.g., GPS device, camera
- Interact with application components, e.g., via port connections
  - Camera sends video frames to an application thread for processing at a rate of 25 frames per second
- Often 3rd party components that include
  - The device hardware
  - A device driver
Some Device Properties

Execution of the device driver

- Dispatch_Protocol => periodic;
- Period => 50ms;
- Deadline => 50ms;
- Compute_Execution_Time => 15ms;
- Priority => 5;
- Time_Slot => (1);

Memory and processor binding for execution

- Allowed/Actual_Memory_Binding
- Allowed/Actual_Processor_Binding
An Overview of AADL V2

Bringing Application Software and Execution Platform Together

Application software relies on computational resources for execution of threads, and communication among threads and between threads and devices.

In a model, application software and execution platform often form independent system hierarchies.

AADL provides binding properties to describe how application software components are allocated to the execution platform.
Binding Properties

Application software

- Component – process, thread, data, device, subprogram
- Ports – data port, event data port
- Connections – port connection, data/subprogram access connection, ...

Execution platform

- Components – processor, memory, device
- Communication – bus, bus access connections

Map application software elements to execution platform elements using binding properties

- Actual_Processor_Binding – Specify which processor schedules and executes a thread or executes a (kernel mode) device driver
- Actual_Memory_Binding – Specify the memory components in which executable code (process components) and data (data component) reside
- Actual_Connection_Binding – Specify the communication channels that are used by logical connections
Handling of Processor Speed

Execution time expressed in absolute time

Modeling of processor-specific execution times

- **Use of binding-specific property values**
  
  ```
  Compute_Execution_Time => 700us..750us
  in binding PowerPC.Mhz350;
  Compute_Execution_Time => 600us..630us
  in binding PowerPC.Mhz450;
  ```

- **Execution time in terms of reference processor**
  
  ```
  Reference_Processor
  Scaling_Factor
  ```

- **Estimated or measured time for each processor**

- **Execution time scaled to processor speed differential**

- **Other processors indicate their speed relative to the reference processor**

- **Execution times are specified for this processor**
Virtual Processor

Logical resource similar to a processor without the hardware aspect
- Schedules and executes threads and other virtual processors
- May communicate with other components via ports
- May provide services
- Must be bound to a processor for execution and are scheduled like threads

Threads can be bound to a virtual processor for execution

A virtual processor can be bound to another virtual processor for execution

A hierarchy of virtual processors represents a hierarchy of virtual machines, each with its own scheduling policy (hierarchical scheduling)

Two ways to associate a virtual processor with a physical processor
- Bind a virtual processor to a (virtual) processor
  → Flexible binding via property associations, just like thread bindings
- Declare a virtual processor as a subcomponent of a (virtual) processor
  → Fixed binding via component containment
Hierarchical Scheduling

A sporadic server is scheduled as a fixed priority thread on a processor virtual processor implementation SporadicServer.impl properties

Allowed_Dispatch_Protocol => (sporadic);
Scheduling_Protocol => (FIFO);
Period => 50ms;
Execution_Time => 10ms;
Dispatch_Protocol => periodic;
end SporadicServer.impl;

processor implementation RMProcessor.impl properties

Allowed_Dispatch_Protocol => (periodic);
Scheduling_Protocol => (RMS);
end RMProcessor.impl;

system implementation SS.impl subcomponents

sserver: virtual processor SporadicServer.impl;
rmserver: processor RMProcessor;

properties

Actual_Processor_Binding => reference(rmserver) applies to sserver;
end SS.impl;
Modeling Partitioned Architectures

A partitioned architecture provides spatial and temporal partitioning of a physical execution platform.

A partition is a virtual machine where each partition has exclusive access:
- To a memory region
- To all other compute resources whenever it executes

All communication between partitions must be mediated by the partitioning operating system.

Partitioning forms a basis for sharing hardware resources between processes:
- Mixed criticality – to guarantee limited fault propagation
- Mixed security levels – to allow the operating system to enforce a system wide security policy

In the avionics domain, ARINC 653 specifies a partitioned execution environment.
Modeling ARINC 653 Partitions Containment Approach

Partitions are assigned fixed time slots in the schedule

```plaintext
virtual processor implementation RMA.impl

properties
  Allowed_Dispatch_Protocol => (periodic);
  Scheduling_Protocol => (RMS);
  Dispatch_Protocol => none;
end Partition.impl;

processor implementation PPC.two

subcomponents
  part1: virtual processor RMA.impl;
  part2: virtual processor RMA.impl;

properties
  Allowed_Dispatch_Protocol => (none);
  Scheduling_Protocol => (FixedTimeline);
  Frame_Period => 90ms;
  Slot_Time => 30ms;
  Time_Slot => (1) applies to part1;
  Time_Slot => (2,3) applies to part2;
end PPC.two;
```
We choose a layered approach, where layers are connected by bindings.

```plaintext
system implementation AppSystem.impl
  subcomponents
    app1: process AppPartition.one;
    app2: process AppPartition.two;
end AppSystem.impl;

system implementation VirtualPlatform.impl
  subcomponents
    part1: virtual processor RMA.impl;
    part2: virtual processor RMA.impl;
end VirtualPlatform.impl;
```
Virtual Bus

Logical component representing a protocol or virtual channel, similar to a bus without the hardware aspect

Connections can be bound to a virtual bus
A virtual bus can represent an communication channel on a shared bus
  - Portion of the bus bandwidth
  - Performance guarantees per channel

Virtual buses are bound to bus, virtual bus, processor, and device components – like connections
A hierarchy of virtual buses can represent a protocol hierarchy
  - E.g., HTTP $\rightarrow$ TCP/IP $\rightarrow$ Ethernet
  - The lowest level of the hierarchy is a bus component
An Overview of AADL V2

Modeling Communication Channels

Buses can be subdivided into a set of virtual channels, each with its own bandwidth guarantees

```plaintext
virtual bus VOIP
end VOIP;

bus implementation Ethernet.twoVOIP
  subcomponents
    channel1: virtual bus VOIP {Bandwidth => 300 MBpSec;};
    channel2: virtual bus VOIP {Bandwidth => 250 MBpSec;};
  properties
    Multiplexing_Protocol => TDMA;
    Bandwidth => 1 GBpSec;
end Ethernet.two;
```

Bandwidth provided by the virtual channels

How the channels are managed

Similarly, a virtual bus can be subdivided into sub-channels

Each (virtual) bus supports its own multiplexing protocol, e.g., cellular networks use

- time division multiplexing (TDMA), or
- code division multiplexing (CDMA)
Modeling Communication Protocols

There are three approaches to indicate that a component provides a protocol

1. Add a virtual bus subcomponent
2. Bind a virtual bus to the component
3. List a virtual bus classifier in the component’s `Provided_Virtual_Bus_Class` property

Example: A bus that provides both HTTP and HTTPS protocols

```plaintext
bus implementation Ethernet.Web
    subcomponents
      protocol_HTTP: virtual bus HTTP;
    properties
      Provided_Virtual_Bus_Class => HTTPS;
      Bandwidth => 1 GBpSec;
end Ethernet.Web;
```

Protocol via subcomponent
Allow multiple channels with the same protocol

Protocol via property
Outline: AADL Standard & MBE

• Why AADL
• AADL Language Overview
• Modeling Embedded Software
• Modeling Computer Systems
• Properties & Patterns
• Large Scale Systems
• Summary
Property Sets

Allow definition of properties, property types, and property constants
Logical grouping of properties, e.g., all related to resource budgeting

Property types
• Define a set of allowed values for a property

Property constants
• Define a named property value

Properties
• Are defined in a property set
• Have a type and optionally a default value
• Can have either a single value or a list of values
• Are applicable to certain named AADL model elements

```aadl
property set myProps is
  import declarations
  types, constants, properties
end myProps;
```
Property Types 1

Boolean – aadlboolean
String – aadlstring
Enumerations – enumeration ( literal₁, literal₂, ... )
Units – units ( unit₁, unit₂ => unit * factor, ... )
aadlinteger [lower_bound .. upper_bound] [units units]
aadlreal [lower_bound .. upper_bound] [units units]
range of number_type
classifier [ ( category₁, category₂, ... ) ]
reference [ ( named_element_kind₁, ... ) ]
record ( field_name₁: [ list of ] property_type₁; ... )
Pre-declared Property Sets

The AADL standard includes 7 pre-declared property sets which are available in every AADL specification

1. **Deployment_Properties** – Binding constraints and actual bindings of application software to execution platform components
2. **Thread_Properties** – Characteristics of active components (threads and devices): dispatching, concurrency, mode transition
3. **Timing_Properties** – Time related characteristics of active components; runtime system support for thread execution
4. **Communication_Properties** – Properties to specify connection topology and queuing characteristics
5. **Memory_Properties** – Properties related to memory as storage, and memory and device access
6. **Programming_Properties** – Properties to specify relationship between AADL model elements and elements of an implementation in a programming language or hardware description language
7. **Modeling_Properties** – Properties that relate to the model itself
Abstract Components

Abstract components represent components without a specific category.

Abstract types can have any kind of features.
Abstract implementations can have any kind of subcomponents.
Any component implementation can have abstract subcomponents.

Abstract components can be specialized into a concrete component category:
  • Extension of abstract classifiers
  • Refinement of abstract subcomponents

**Note:** The features and subcomponents of an abstract component restrict the valid concrete categories that can be assigned.
Extensions and Refinements

Define a new extended classifier based on an existing classifier
Allows incremental refinement of a model

Component extension
  • Component types
  • Component implementations

Feature group type extension

Applications
  • Add elements to a classifier
    • Features, subcomponents, connections, flows, etc.
  • Refine existing elements in a component
  • Add or override properties
Refinement Substitution Rules

**Classifier_Match**
1. Goto type
2. Select an implementation

**Type_Extension**
1. Goto type
2. Select an extension
3. Select an implementation (optional)

**Signature_Match**
1. Goto type
2. Select a type with a superset of features and flow specifications
3. Select an implementation (optional)
Prototypes – Consistent Refinement

Example: Type of data on a port

```
system GpsGeneric
  prototypes
    dt: data;
  features
    pos_1: out data dt;
    pos_2: out data dt;
end GpsGeneric;

system Gps
  extends GpsGeneric(dt=>PosData)
end Gps;

system GpsBasic
  features
    pos_1: out data;
    pos_2: out data;
end GpsBasic;

system GpsRef extends GpsBasic
  features
    pos_1: refined to out data PosData;
    pos_2: refined to out data OtherData;
end GpsRef;
```

No enforcement of consistency possible
Abstract Features

Placeholders for concrete features (port, access, parameter, feature group)
May specify the direction of data or control flow
In a complete model all abstract features are replaced with concrete ones
  • By refinement into the concrete feature
  • By providing the concrete feature in a prototype binding

Refinement

```
thread filter
  features
    raw: in feature;
    filtered: feature;
end filter;
```

```
thread filter1 extends filter
  features
    raw: refined to in event data port;
    filtered: refined to out data port;
end filter1;
```
Subcomponent Arrays

Indicate a multiplicity at a subcomponent declaration within a component implementation

- Multidimensional arrays allowed, dimension is fixed
- Array size can be specified in subcomponent refinement (but not changed)
- Sizes for all dimensions must be specified in one place

```plaintext
process implementation N_Version.generic
  subcomponents
    myCompute: thread Compute [];
    myVoter: thread Voter;
end N_Version.generic;

process implementation N_Version.triple
  extends N_Version.generic
  subcomponents
    myCompute: refined to thread Compute [3];
end N_Version.triple;
```
Feature Arrays

Indicate a multiplicity at a feature declaration within a component type

- Only one-dimensional feature arrays allowed
- Array size can be specified in a feature refinement (but not changed)
- Limited to features of threads, devices, and processors

AADL Syntax:

```
thread Voter
  features
    input: in data port [3];
    output: out data port;
end Voter;
```

A property Acceptable_Array_Size can be associated with a feature or subcomponent to constrain the size of an array.
Acceptable_Array_Size => 2..5;
Connecting Arrays

Determine semantic connections in the presence of component and feature arrays <<FIXME:

![Diagram of NVersion.triple and connection between arrays](image.png)

Connection between two arrays:

```c1: port myCompute.dat -> myVoter.input {
    Connection_PATTERN => (one_to_one);
};

myCompute[N].dat -> myVoter.input[N], N = 1,2,3```

In general, (1) determine the semantic connections without arrays and (2) apply the connection pattern to them.
Connection patterns – one dimension

Identity aka. One_To_One

next

Cyclic Neighbor = (CyclicNext, CyclicPrevious)

Neighbor = (next, previous)

next, one_to_one
Connection patterns – two dimensions

S[3,3]; D[3,3];
Port S.p1 -> D.p2;
Connection Sets

Property Connection_Set can be used if patterns are insufficient to express desired connectivity
Each connection is specified individually in the set

Connection_Set => (  
  (src => (1,1), dst => (1,2)),  
  (src => (1,1), dst => (2,2)),  
  (src => (1,2), dst => (1,3)),  
  (src => (1,3), dst => (2,2)),  
  (src => (2,1), dst => (2,2)),  
  (src => (2,2), dst => (2,3)),  
  (src => (3,1), dst => (2,2)),  
  (src => (3,1), dst => (3,2)),  
  (src => (3,2), dst => (3,3)),  
  (src => (3,3), dst => (2,2))  
);
Outline: AADL Standard & MBE

• Why AADL
• AADL Language Overview
• Modeling Embedded Software
• Modeling Computer Systems
• Properties & Patterns
• Large Scale Systems
• Summary
Package Example

package Aircraft::Cockpit

public

with Avionics::DataTypes;

AirDataType renames Avionics::DataTypes::AirData;

system MFD

features

  air_data: in data port AirDataType;

end MFD;

private

system implementation MFD.impl

subcomponents

  local_data: data AirDataType;

end MFD.impl;

end Aircraft::Cockpit;

Private imports, aliases, and classifiers are local to the private section

Imports, aliases, and classifiers from the public part are visible in the private section
Use of AADL Packages

AADL Packages as name spaces
  • Nested package names: edu::cmu::sei::MySystem::App1
  • Qualify references by package name: BaseTypes::uint16

Component libraries
  • Component types and implementations
  • Hardware & application SW & SW task libraries
  • Subsystem details in separate packages

Data dictionary
  • Data types
  • Domain information on data types

Interaction specifications
  • Port group type specifications

System configurations
  • Deployment
  • Parameterization
An Overview of AADL V2

Variants in System Families or Product Lines

Multiple interface variants
- AADL component types with extends

Multiple realizations
- Multiple AADL component implementations per type

Variation in component structure and communication
- Parameterized component implementations (AADL V2 Prototype concept)
- Dynamic variation through mode-specific subcomponents and connections

Source code variations
- Different source files as Source_Text property
- Conditional compilation flags as properties or property constants

Seed & calibration values
- As property values on data components
Refinement of Partial Architectures

Extending the component types & implementations
Modeling of System Configurations

```
package SystemConfigurations
public
  system mysystem
    properties
      SEI::WeightLimit => 11.0 kg;
    end mysystem;
  -- a system with the application as a parts list of subsystems with resource budgets
  -- and a platform with hardware parts with resource capacities
  system implementation mysystem.parts
    subcomponents
      Platform: system HardwarePlatform::ComputingPlatform.ThreeProcessorParts;
      ApplicationSystem: system appsSystems::EmbeddedApp.SubSystemParts;
    end mysystem.parts;
  -- the same system or parts with subsystems allocated to processors
  -- This may be an initial allocation or an allocation specified as part of requirements
  -- Given such an allocation we can analyze whether the budgets of the allocated subsystems exceed the capacity of the processor
  system implementation mysystem.allocatedparts
    extends mysystem.parts
    properties
      Actual_Processor_Binding => reference platform.missionprocessor1 applies to ApplicationSystem.DM;
      Actual_Processor_Binding => reference platform.missionprocessor1 applies to ApplicationSystem.PCM;
      Actual_Processor_Binding => reference platform.missionprocessor2 applies to ApplicationSystem.FM;
      Actual_Processor_Binding => reference platform.missionprocessor2 applies to ApplicationSystem.FD;
      Actual_Processor_Binding => reference platform.missionprocessor3 applies to ApplicationSystem.WAM;
      Actual_Memory_Binding => reference platform.missionprocessor2.membank1 applies to ApplicationSystem.FD;
      Actual_Memory_Binding => reference platform.missionprocessor1.membank1 applies to ApplicationSystem.PCM;
      Actual_Memory_Binding => reference platform.missionprocessor2.membank1 applies to ApplicationSystem.FM;
      Actual_Memory_Binding => reference platform.missionprocessor1.membank1 applies to ApplicationSystem.DM;
    end mysystem.allocatedparts;
```

Use of extends to specify configurations

Use of contained property association to keep deployment information in one place
Managing an Architecture Model

- Reference architecture and instances (NASA/JPL)
- Integrator and suppliers (SAVI)
An Overview of AADL V2

Early & Continuous Validation of Multi-tier System & SW Architecture through Virtual Integration (AVSI SAVI)

Aircraft: (Tier 0)

Aircraft system: (Tier 1)
- Engine, Landing Gear, Cockpit, ...
- Weight, Electrical, Fuel, Hydraulics,

LRU/IMA System: (Tier 2)
- Hardware platform, software partitions
- Power, MIPS, RAM capacity & budgets
- End-to-end flow latency

OEM & Subcontractor: Subsystem interaction validation
- Functional integration consistency
- ARINC 429 protocol consistency

Subcontracted software subsystem: (Tier 3)
- Tasks, periods, execution time
- Software allocation, schedulability
- Generated executables

Redundancy Logic Validation:
- Dual Flight Guidance System
- Nominal, distributed, logical fault, physical fault, asynchronous system

Additional Opportunities:
- Safety & security analysis
- Fault modeling & impact analysis
- What-if trade studies

Validation through Analysis Demonstration
- Propagate requirements and constraints
- Higher level model down to suppliers' lower level models
- Verification of lower level models satisfies higher level requirements and constraints

SEI created ROI of validation impact on rework reduction

© 2010 Carnegie Mellon University
Outline: AADL Standard & MBE

• Why AADL
• AADL Language Overview
• Modeling Embedded Software
• Modeling Computer Systems
• Properties & Patterns
• Large Scale Systems
• Summary
Conclusion

AADL has been used in a number of large-scale industrial pilot projects
  • Most recently: AVSI System Architecture Virtual Integration with Boeing, Lockheed Martin, Airbus, Suppliers, FAA, DoD, SEI (year 2 of 5)
  • Commercial AADL support through ElliDiss & MARTE subset as UML profile for AADL

AADL has become a research platform of choice for fast-tracking transition
  • Over 200 published papers in refereed conferences and journals
  • Wide range of analysis and code generation plug-ins and tools

AADL and other standards
  • Ongoing harmonization with SysML and MARTE