AADL
Architecture Analysis & Design Language
(SAE AS-5506A Std)
An Introduction

G. Edward Roberts
eroberts@elparazim.com
Freely taken from

Software Engineering Institute | Carnegie Mellon

SAE AADL V2: An Overview

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Peter Feiler
phf@sei.cmu.edu

Sponsored by the U.S. Department of Defense
© 2010 by Carnegie Mellon University
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)

System User

Operator Error
Lag, proximity

Hardware Engineer

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

Runtime Architecture

Concurrency
Communication
iTunes crashes on dual-cores

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)
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 Text

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

Graphical

AADL Representation Forms

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>
MetaModel

e.g. For use with OSATE
Cooperative Engineering of Systems

AADL
- Application Software Runtime Architecture (task & communication)
  - Application Software Components (source code)
    - Java, UML, Simulink
- Computer Platform Architecture (processors & networks)
  - Hardware Components (circuits & logic)
    - VHDL

SysML
- Physical System Architecture (interface with embedded SW/HW)
  - Physical Components (mechanical, electrical, heat)
    - Modelica
- Operational Environment (People, Use scenarios)
  - UML

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

TOGAF

Phase D

Enterprise

SysML

ATL

System

AADL

ATL

“Electronics”

UML2

Acceleo

Software

RT Java 5

With Annotations

Implementations

ATL = Atlas Transform Language
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

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

Resource Consumption
- Bandwidth
- CPU time
- Power consumption

Reduced model validation cost due to single source model
UML2 MARTE Profile

Modeling and Analysis of Real-Time and Embedded Systems (MARTE)
MARTE Sections

- NFP (Non-Functional Properties) Modeling
- VSL (Value Specification Language)
- Time Modeling
- GRM (General Resource Modeling)
- Alloc (Allocation) Modeling
- GCM (Generic Component Modeling)
- HLAM (High-Level Application Modeling)
- DRM (Detail Resourced Modeling)
- GQAM (Generic Quantitative Analysis Modeling)
- SAM (Schedulability Analysis Modeling)
- PAM (Performance Analysis Modeling)
- RSM (Repetitive Structure Modeling)
NFP (Non-Functional Properties) Modeling
VSL (Value Specification Language)

```
<resource>
  uC: Controller
<end>

  procUtilz = (Su1, calc)

  <clockResource>
  p1 / procClock
  <end>

  <scheduler>
  {schedPolicy = FixedPriority}
  s1 / sysSched
  <end>

  <nfpContraint {kind = deferred}
  {contextSwitch = (8, us, msg) and schedUtilz = (5, percent)
  <end>

  <nfpContraint {kind = contract}
  {procUtiliz > (90, percent) ? clockFreq = (60, MHz) : clockFreq = (20, MHz)
  <end>

VSL Conditional Expression
Condition
If-True Expression
If-False Expression
```
Time Example

Example of sequence diagram

MOS stands for MessageOccurrenceSpecification

Note that red and blue annotations are not part of the UML notation.
GRM Example
SRM (Software Resource Modeling)
HRM Primitives

- **HwComputing**
  - "HwProcessor", "HwPLD", "HwASIC"

- **HwStorage**
  - "HwCache", "HwRAM", "HwROM", "HwDrive"
  - "HwMMU", "HwDMA"

- **HwDevice**
  - "HwDevice", "HwI/O"
  - "HwSupport"

- **HwTiming**
  - "HwClock", "HwTimer"

- **HwCommunication**
  - "HwBridge"
  - "HwArbiter"
  - "HwMedia", "HwBus"

- **HwLayout**

- **HwPower**
  - "HwPowerSupply"
  - "HwCoolingSupply"

kind: {Card, Channel, Chip, Port}
SAM (Schedulability Analysis Modeling)
PAM (Performance Analysis Modeling)
Repeative Structure Modeling (RSM)

- **SIMD unit**
  - 16 processors

- **Topology**
  - Toroidal 4x4 grid
  - Bidirectional connections
    - North-South
    - East-West
Semantically Consistent Architecture & Analysis Concepts

Architecture Meta Model

- Error Occurrence & Propagation Behavior
- Error Model Annex

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

© 2010 Carnegie Mellon University

6/30/2011 eroberts@elparazim.com Copyright 2011
OSATE Plug-in Extensions

OSATE
- Textual AADL, Graphical AADL
- XML/XMI AADL, AADL object model API
- AADL extension support

EMF
- XML/XMI, Metamodel
- Change notification
- Multi-file support

Eclipse
- Platform independence
- Extensible help
- Task & Problem Mgt
- Team support
- Plug-in development

AADL Front-end
- Text editor
- Object editor
- Graphical editor
- Text <-> XML
- Semantics

OSATE Extensions
- Analysis template
- Generation template
- AADL Semantic API

External Models
- External tools

Model Transformation
- Timing analysis (RMA)

Architecture Import
- Simulink/Matlab model
- Extraction via SVM

Architecture Export
- MetaH

Architecture Analysis
- Security level
- Data stream miss rate
- Latency

Architecture Consistency
- Required connectivity
- Model completeness profiles
- Connectivity cycles

Architecture Transform
- Conceptual architecture ->
- Runtime architecture
- Rate group optimization
- Port group identification

www.aadl.info

© 2004 by Carnegie Mellon University
6/30/2011
eroberts@elparazim.com Copyright 2011
AADL Toolset

• **OSATE – Open Source**
  – SEI developed, full language editing and semantic checking, multiple analysis plug-ins, Eclipse based, integrated text and graphical editing with TOPCASED

• **TOPCASED – Open Source**
  – Airbus led, 20 companies, Metamodeling Framework, AADL Graphics, AADL XML, model transformation, Behavior Annex, also will support UML, stable July 2007

• **STOOD - Commercial**
  – CASE toolset supporting UML, HOOD and AADL. Includes transformations between notations, document support. Works with OSATE, TOPCASED, and Cheddar

• **OCARINA – Open Source**
  – ENST AADL graphics and middleware generation and integration to AADL model of network distributed processors. Creates formal model of executive integrated in AADL. Generates to network protocols – CORBA, RT, FT

• **Fremont – Open Source, Consulting and Toolset support**
  – AADL to ACRS (process algebra), formal analysis of concurrent resources.
  – AADL to Charon, generation and integration of hybrid control systems.
  – AADL Architecture Simulator – integrates event driven and schedule driven

• **Generic Modeling Environment (GME) – Consortium**
  – Vanderbilt Univ, DARPA sponsored Metamodeling Framework, AADL capture and role based system security analysis, model transformation, integration.

• **CHEDDAR – Open Source**
  – Univ of Brest, advanced scheduling analysis toolset
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
AADL: Components and Connections
Standard Component Types

**Software Components**
- data
- subprogram
- subprogram group
- thread
- thread group
- process

**Execution Platform Components**
- Compute Hardware:
  - processor,
  - virtual processor
  - memory
  - bus
  - virtual bus

- Physical Environment:
  - device

**System Component**
- system
- abstract
- prototype
Software 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

**Subprogram Group** - represent subprogram libraries.
Execution Platform Components

**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
Type vs Implementation

1. Type represents the functional interface of the component, what is visible by other components.

2. The *implementation*, describes the contents of the component (subcomponents, properties, connections, etc.).

```
system type1
end type1;

system implementation type1.impl1
end type1.impl1;

system type2
end type2;

system implementation type1.impl2
end type1.impl2;
```

Note: Types have thin lines and implementations have thick lines.
Inheritance: type and impl

```
system type1
end type1;

system type2 extends type1
end type2;

system implementation type2.impl1
end type2.impl1;

system implementation type1.impl1
end type1.impl1;

system implementation type1.impl2 extends type1.impl1
end type1.impl2;

system implementation type2.impl2 extends type1.impl2
end type2.impl2;
```
System State Model

- **System offline**
  - `abort(system)`
  - `start(system)`
    - `ST ← 0`
- **System starting**
  - `ST ≤ startup_deadline`
    - `started(system)`
      - `assert ST ≤ startup_deadline`
- **System operational**
- **System stopping**
  - `stop(system)`
  - `stopped(system)`
  - `abort(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.

```
system type1
  properties
    Startup_Deadline => 0.5s;
  end type1;

system implementation type1.impl1
  properties
    Startup_Deadline => 0.5s;
  end type1.impl1;
```

This `Startup_Deadline` applies to only
Instances of this type

This `Startup_Deadline` applies to only
This instance
Property Sets

A named group of property types, property definitions, and property constant values.

```plaintext
property set mine is
  Length_Unit : type units ( mm,
    cm => mm * 10,
    m => cm * 100,
    km => m * 1000 );
  OnOff : type aadlboolean;
  Car_Length : type aadlreal 1.5 .. 4.5 units Length_Unit ;
  Speed_Range : type range of aadlreal 0 .. 250 units ( kph );
  Position : type record ( X: aadlininteger; Y: aadlininteger; );
  Max_Threads : constant aadlinteger => 256;
end mine;
```

This type declaration references a separately declared units type

This type declaration defines the units in place
Property Types

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; ... )
Predefined Properties

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
Timing_Properties Example

property set Timing_Properties is
...
Startup_Deadline: Time
applies to (processor, virtual processor, process, system);
...
end Timing_Properties;

Properties are available for:
Any namable model element, e.g.,
components, features, modes, connections,
flows, and subprogram calls

One can now see how one could extend these components with properties
From a different domain of interest (in an Annex)
Subcomponent

```
system CarSystem
end CarSystem;

system BrakingSystem
end BrakingSystem;

system implementation CarSystem.impl
subcomponents
  braking: system BrakingSystem;
end CarSystem.impl;
```
Subcomponent Array

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;
```

Single dimension, size still undefined

myCompute[1], myCompute[2], myCompute[3]
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
Thread Properties

Properties related to thread dispatch
• Dispatch_Protocol => periodic;
  One from previous slide (or user-defined)
• Period => 50ms;
  Required for periodic, sporadic, timed, and hybrid threads

Properties needed for thread scheduling
• Compute_Execution_Time => 45ms..50ms;
  The execution time range of this thread – upper bound is worst case execution time
• Deadline => 40ms;
  Optional, defaults to period
• Dispatch_Offset => 5ms;
  For periodic threads: indicate delayed dispatch relative to other periodic threads
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 States

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

Substates:
- Executing normally
- Blocked
- Preempted
- Suspended

State with associated code execution
State without code execution
What do we need to describe an Architecture?

Component

Feature
describes an interface of a component through which control and data may be provided to or required from other components.

Component

Connection
specify interaction between components at runtime.
Feature Types

*data port* - connection points for transfer of state data such as sensor data

*event port* - connection points for transfer of control through raised events that can trigger thread dispatch or mode transition.

*event data port* - connection points for transfer of events with data, i.e., messages that may be queued.

**Provided subprogram access** - entrypoints to code sequences in source text that is associated with a data type or a thread that can be called locally or remotely

**Subprogram parameters** - represent in and out parameters of a subprogram.

**Data component access** - provided and required access to shared data.

**Bus component access** - provided and required access to buses for processors, memory, and devices.
**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
Ports

```plaintext
process process1
features
  outport: out data port;
end process1;

thread thread1
features
  outport: out data port;
end thread1;

process process2
features
  inport: in data port;
end process2;

thread thread2
features
  inport: in data port;
end thread2;
```
Subcomponents and Connections

```
process implementation process1.impl
  subcomponents
    t1: thread thread1.impl;
  connections
    cn: data port t1.outport -> outport;
end process1.impl;

thread implementation thread1.impl
end thread1.impl;

process implementation process2.impl
  subcomponents
    t2: thread thread2.impl;
  connections
    cn: data port inport -> t2.inport;
end process2.impl;

thread implementation thread2.impl
end thread2.impl;
```
The **Semantic Connection** is from thread1.impl to thread2.impl
Port Typing

data Alpha_Type
  properties
    Source_Data_Size => 256 Bytes;
end Alpha_Type;

thread P
  features
    Data_Source : out data port Alpha_Type;
end P;
Pattern: Message Passing

data d end d;

thread receiver_node
features
  e : in event data port d;
end receiver_node;

thread sender_node
features
  s : out event data port d;
end sender_node;

process proc end proc;

process implementation proc.i
subcomponents
  t1 : thread sender_node;
  t2 : thread receiver_node;
connections
  event data port t1.s -> t2.e;
end proc.i;
## 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
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";
Connection Types

Timing: \textit{enumeration (sampled, immediate, delayed)} \Rightarrow \text{sampled}
Sampled Data Port Connection
Connection Type (cont.)
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)
Subprogram/Group Access

data d
d end d;

subprogram sp
features
  e : in parameter d;
  s : out parameter d;
end sp;

thread implementation node.i
calls
  {call1 : subprogram sp;
   call2 : subprogram sp;};
connections
  parameter e -> call1.e;
  parameter e -> call2.e;
  parameter call1.s -> s;
  parameter call2.s -> s;
end node.i;
	hread node
features
  e : in event data port d;
  s : out event data port d;
end node;
Execution Semantics: Example

data d end d;

subprogram sp
features
  e : in parameter d;
  s : out parameter d;
end sp;

thread node
features
  e : in event data port d;
  s : out event data port d;
end node;

data d end d;

subprogram sp
features
  e : in parameter d;
  s : out parameter d;
end sp;

thread implementation node.i
subcomponents
  var1 : data d;
  var2 : data d;

calls
{call1 : subprogram sp;
  call2 : subprogram sp;
  call3 : subprogram sp;};

call connections
  cnx1 : event data port e -> var1;
  cnx2 : parameter e -> call11.e;
  cnx3 : parameter var1 -> call12.e;
  cnx4 : parameter call11.s -> call13.e;

  cnx5 : parameter call11.s -> var2;
  cnx6 : parameter call12.s -> s;
  cnx7 : parameter call13.s -> s;
  cnx8 : event data port var2 -> s;
end node.i;

• the first incoming data in e is stored in var1
• the second incoming data is passed to call1
• call2 uses the first data, stored in var1
• call3 uses the output of call1
• the output of call1 is stored in var2
• the outputs of call2 and call3 are sent through s
• the content of var2 is then sent through s
Pattern: Remote Procedure Call

subprogram sp end sp;

thread server_node
features
  rpc : provides subprogram access sp;
end server_node;

thread client_node
features
  rpc : requires subprogram access sp;
end client_node;

thread implementation client_node.i
calls
  {call1 : subprogram access rpc;};
end client_node.i;

process proc end proc;

process implementation proc.i
subcomponents
  t1 : thread client_node.i;
  t2 : thread server_node;
connections
  subprogram access t2.rpc -> t1.rpc;
end proc.i;
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)
**Data Component**

```plaintext
data Address
   features
   getStreet : provides subprogram access;
   getCity : provides subprogram access;
end Address;

data implementation Address.others
   subcomponents
   street : data Base_Types::String;
   streetnumber: data Base_Types::Integer;
   city: data Base_Types::String;
   zipcode: data Base_Types::Integer;
end Address.others;
```
Pattern: Object Oriented

data class
    features
        public_method : subprogram;
    end class;

data implementation class.i
    subcomponents
        attribute : data;
        private_method : subprogram;
    end class.i;
Shared Data Access

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
Data Access

**Thread1**
- **features**
  - A: provides data access Data1;
  - B: required data access Data2;
- **end** Thread1;

**Thread2**
- **features**
  - C: requires data access Data1;
- **end** Thread2;
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_Physical_Access => (processor, memory);

Properties related to data transmission time
- Transmission_Time
- Latency

What can be transmitted over this bus
Supported protocols and QoS
What may be connected to this bus
**system** System1
end  System1;

**system implementation** System1.impl
features
  A: **provides bus access** Bus1;
subcomponents
  B: **processor** Processor1;
  C: **device** Device1;
  D: **memory** Memory1;
connections
  bus access A <-> B.E;
end System1.impl;

**processor** Processor1
features
  E: **requires bus access** Bus1;
end  Processor1;
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
- ...
Example Flows

Flow Specification
- Flow path F1: pt1 -> pt2
- Flow path F2: pt1 -> pt3

Flow Implementation of flow path F1
- Flow path F1: pt1 -> C1 -> P2.F5 -> C3 -> P1.F7 -> C5 -> pt2
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;
```

![Diagram of flow implementation]

```
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.FSsrc1 -> C1 -> cruise_control.brake_flow ->
   C2 -> throttle_actuator.FSnk1;
end CarSystem.impl;
Flow Concept in AADL

device brake_pedal
features
  brake_status: out data port bool_type;
flows
  Flow1: flow source brake_status;
end brake_pedal;

system cruise_control
features
  brake_status: in data port;
  throttle_setting: out data port;
flows
  brake_flow_1: flow path brake_status -> throttle_setting;
end cruise_control;

device throttle_actuator
Features
  throttle_setting: in data port float_type;
flows
  Flow1: flow sink throttle_setting;
end throttle_actuator;
Abstract Features

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;
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:

```adl
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:

Connection between two arrays:

```haskell
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_Next**
- **previous**
- **Neighbor = (next, previous)**
- **Cyclic Neighbor = (CyclicNext, CyclicPrevious)**
- **next, one_to_one**
Connection Patterns – Two Dimensions

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

*Bundling of connections reduces graphical clutter

* 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.
Feature Group as a plug

feature group xfer_plug
features
  Alpha : out data port Alpha_Type;
  Beta : in data port Alpha_Type;
end xfer_plug;

process A
features
  Produce : feature group xfer_plug;
end A;

feature group xfer_socket inverse of xfer_plug
end xfer_socket;
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
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
Binding

```plaintext
processor PowerPC
features
  Card_Connector : requires bus access VME;
end PowerPC;

thread Producer
end Producer;

processor implementation PowerPC.speed_350Mhz
properties
  Speed => 350Mhz;
end PowerPC.speed_350Mhz;

thread implementation Producer.Basic
properties
  Compute_Execution_Time => 0ms..10ms in binding ( PowerPC.speed_350Mhz );
  Compute_Execution_Time => 0ms..8ms in binding ( PowerPC.speed_450MHz );
end Producer.Basic ;
```
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
Modeling ARINC 653 Partitions

Partitions are assigned fixed time slots in the schedule.

```plaintext
virtual processor implementation RMA.impl1
  properties
    AllowedDispatchProtocol => (periodic);
    SchedulingProtocol => (RMS);
    DispatchProtocol => none;
end Partition.impl1;
processor implementation PPC.two
  subcomponents
    part1: virtual processor RMA.impl1;
    part2: virtual processor RMA.impl1;
  properties
    AllowedDispatchProtocol => (none);
    SchedulingProtocol => (FixedTimeline);
    FramePeriod => 90ms;
    SlotTime => 30ms;
    TimeSlot => (1) applies to part1;
    TimeSlot => (2,3) applies to part2;
end PPC.two;
```

- Fixed binding
- Scheduling slots
- Slot assignment to partitions
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 → TCP/IP → Ethernet
• The lowest level of the hierarchy is a bus component
Modeling Communication Channels

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

```
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_Proto => TDMA;
    Bandwidth => 1 GBpSec;
end Ethernet.two;
```

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)
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
Abstract Component Example

```
abstract implementation car.generic
subcomponents
    PowerTrain: abstract power_train;
    ExhaustSystem: abstract exhaust_system;
end car.generic;

abstract power_train
features
    exhaustoutput: requires bus access Manifold;
end power_train;

abstract exhaust_system
features
    exhaustManifold: provides bus access Manifold;
end exhaust_system;

system carRT extends car
end carRT;

system implementation carRT.impl extends car.generic
subcomponents
    PowerTrain : refined to process power_train;
    ExhaustSystem : refined to process exhaust_system;
end carRT.impl;
```
Prototype Consistancy

```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
Another Prototype Example

abstract flowComponent
prototypes
dt: data;
incoming: in feature;
features
insignal: in feature incoming;
outsignal: out data port dt;
end flowComponent;

data implementation signal.unit16
end signal.unit16;

process controller extends flowComponent ( dt => data signal.unit16,
incoming => event data port signal.unit16 )
end controller;
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;
Derived Modes in Detail

system implementation S.impl
  modes
    A initial mode; B mode;
    C mode; D mode; E mode;
    ... mode transitions here ...
  end S.impl
end system S1;

subcomponents
  subsys: system S1 in modes (A => X, B => Y, C => Y, D);
end system S1;

If S.impl is in mode ...
|       | ... then subsys is in derived mode ...
<table>
<thead>
<tr>
<th></th>
<th></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_Transition`.
  - 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
Package

provide a means to organize the descriptions by the use of namespaces

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

Provides a structure for organizing component type, component implementation, feature group types, and annex libraries into a separate namespace.

A defining package name must be unique in the global namespace.

Packages can be organized hierarchically by giving them nested package names. This hierarchy does not impose any restrictions on whether a package is accessible by other packages.

A defining package name consists of a sequence of one or more package identifiers separated by a double colon.
public and private

Packages have a public and a private section

```
package A
public
  system B
end B;
private
  system C
end C;
  system D
end D;
end A;

package E
public
  system F
end F;
end E;
```

Declarations in the public section are visible outside the package, i.e., names declared in the public part can be referenced by declarations in other packages.

Declarations in the private section are visible only within private section of the package, i.e., they cannot be referenced from the public section or from other packages.
Annex

enable the use of declarations written in another sublanguage.

The AADL core language is extensible through

1. property sets,
2. annex subclauses
3. annex libraries (standardized or user-defined)

Annex subclauses consist of annex-specific sublanguages whose constructs can be added to component types and component implementations.

Annex libraries are declarations of reusable annex specific sublanguage elements that are placed in AADL packages and can be referenced in annex subclauses.

You are basically free to define whatever language you want in these libraries.
Use of an Annex in Model

```plaintext
thread Collect_Samples
  features
    Input_Sample : in data port Sampling::Sample;
    Output_Average : out data port Sampling::Sample;
  annex Error_Model {**
    Model => Transient_Fault_Model;
    Occurrence => 10e-4 poisson applies to Transient_Fault;
  **};
end Collect_Samples;
```
Example Annex Extension

THREAD t
 FEATURES
   sem1 : DATA ACCESS semaphore;
   sem2 : DATA ACCESS semaphore;
 END t;

THREAD IMPLEMENTATION t.t1
 PROPERTIES
   Period => 13.96ms;
   cotre::Priority => 1;
   cotre::Phase => 0.0ms;
   Dispatch_Protocol => Periodic;

ANNEX cotre.behavior {**
 STATES
   s0, s1, s2, s3, s4, s5, s6, s7, s8 : STATE;
   s0 : INITIAL STATE;
 TRANSITIONS
   s0 -[]-> s1 { PERIODIC_WAIT };
   s1 -[]-> s2 { COMPUTATION(1.9ms, 1.9ms) };
   s2 -[ sem1.wait ! (-1.0ms) ]-> s3;
   s3 -[]-> s4 { COMPUTATION(0.1ms, 0.1ms) };
   s4 -[ sem2.wait ! (-1.0ms) ]-> s5;
   s5 -[]-> s6 { COMPUTATION(2.5ms, 2.5ms) };
   s6 -[ sem2.release ! ]-> s7;
   s7 -[]-> s8 { COMPUTATION(1.5ms, 1.5ms) };
   s8 -[ sem1.release ! ]-> s0;
 **};
 END t.t1;

COTRE thread properties

COTRE behavioral annex

Courtesy of cotre

6/30/2011 eroberts@elparazim.com Copyright 2011
Standard Annex

Annex Document A Code Generation [not normative] provides guidance for automatic generation and integration of runtime systems and application code in different implementation languages. It defines a standardized set of properties for recording mappings from the AADL model to source text and for automatic code generation.

Annex Document B Data Modeling [not normative] provides guidance on data modeling and how to map relevant data modeling information into an AADL model if desirable. It defines a standardized set of properties and basic data types in support of data modeling.

Annex Document C Error Model defines a standardized core language extension in the form of a sublanguage notation and properties the component to support annotating AADL models with safety-criticality and dependability related information of a system.

Annex Document D Behavior Model defines a standardized core language extension in the form of a sublanguage notation to specify the behavior of AADL components as AADL model annotations.

Annex Document E Mini Annexes
Error Annex Example

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

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

• Is this a “formal method”?
• What is the advantages and disadvantages of this over VDM?