AADL V3 Standard Discussions

Peter Feiler

June 2019

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM19-0625

AADL v3 Roadmap

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Overall Strategy

AADL V2.2

- New AADL V2.2 errata: https://github.com/saeaadl/aadlv2.2
- OSATE issue reports: https://github.com/osate
- Long term support (LTS) for OSATE 2.x

AADL V3

- Working slides
  - https://github.com/saeaadl/aadlv3/tree/master/SAEAADLV3
  - Issues: https://github.com/saeaadl/aadlv3/issues
- New draft standard document
  - Document split into sections
  - Revision of packages, component interface, implementation, sucomponent, configuration
  - Document conversion into Restructured Text (RST) to come
- Prototype implementation started
  - https://github.com/saeaadl/AadlV3Prototype

Migration Path to V3

Instance model representation with minimal changes

- Most analyses operate on instance model
- Documented API

Declarative model

- Translation from V2.2 to V3

Key V3 Changes

Packages and General Syntax

- Import of namespaces
- Property definitions in packages
- Private classifiers and property definitions
- Simpler syntax: no section keywords, no matching end identifier
- case sensitive

Composition of Component Interfaces aka. component type

- *Extends* of multiple interfaces
- Interface without category
- Eliminates need for feature group type

Configuration Specification

- Finalize design
- Configuration assignment of subcomponents with implementation, features with classifier/type (Replaces *refined to)*
- Assign final property values to any model element
- Annotate with bindings, annexes, flows
- Configurations are composable
- Parameterized configuration limits choice points (Replaces *V2 prototype)*

Key V3 Changes

Unified type system

- Single type system for properties and data types
- Records, lists, sets, maps, unions
- International System of Units

Properties

- Simplified property value assignment (default, final, override)

Explicit deployment binding concept

- Binding points and binding declarations
- Resources associated with bindings

Virtual platform support

- Virtual memory
- Connectivity between virtual bus, processor, memory

Flows

- (virtual) platform flows
- Flow graphs

Nested component declarations

- Define nested components without explicit classifier

Key V3 Changes

Connections

- Distinguish feature mappings
- Reach down of connection declarations
  - Into named interfaces (aka feature groups)
  - Into subcomponent hierarchy

No more category refinement

- Abstract component to other component
- Abstract feature to other features

Modes

Annex improvements

AADLv3

Table of Contents

AADLv3 table of contents

Goal: Provide an incremental view of AADL concepts

Objective is 4 – x ? Parts, < 60 pages each

Introduction to AADL, Lead: P. Feiler

Part 1: Syntax and general concepts, Lead: P. Feiler

Part 2: Static semantics, Lead: J. Hugues

Part 3: Dynamic semantics, Lead: J. Hugues

Part 4: Property sets & MoC configurations

Allow per part ballot to allow for early prototyping

AADLv3 table of contents

Part 1: Syntax and general concepts

BNF for the core language

AADL concepts: component category, types, implementation, features, bindings, flows, modes…

Type system (L. Wrage)

Property language definition

AADLv3 table of contents

Part 2: Static semantics, i.e.

Goal is to simply present the static structure of an AADL model

Component categories description

New component category: virtual memory

Architecture modeling through containments, connections and bindings

AADLv3 table of contents

Part 3: Dynamic semantics

Default semantics of component category and features

Separates the canonical property-less semantics (P3) from the parametric property-driven semantics (P4)

Rationale is: better to define a per-objective semantics using AADLv3 configuration sets than a long collection of properties that are hard to relate.

AADLv3 table of contents

Part 4 : Property sets & MoC configurations

Catalogue of AADL properties

Parametrization of default semantics through properties

Presentation of specific Model of Computation (e.g. scheduler), or support for verification objectives (e.g. flow latency)

Use v3 configuration sets for specific Model of Computation

Walkthrough: an example

Question: how do I model for scheduling analysis ?

Part 1 defines the general syntax of AADL core

Part 2 defines concepts e.g. threads, virtual processors

Part 3 defines the general state machine of threads, ports

Part 4 introduces properties for scheduling and introduces specific task models as configuration set

If you're familiar with AADL, Part 4 should provide all required information and examples.

Note: to some extend, Part 4 could be read as a collection of annex document, covering flow analysis, scheduling, memory, data modeling, etc.

AADL Configuration Specification

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Architecture Design & Configuration

Architecture design via extends, refines to evolve design space (V2)

- Revise and add to existing architecture design structure
- Add/revise annotation of property values, bindings, annexes

Configuration specification

- Elaborate but do not change architecture structure only expand leaf nodes
- Configuration assignments assign classifiers
  - To subcomponents and features
  - Assignments of classifiers are additive
  - Via configurations associate collections of property values, bindings, annexes to given architecture substructure

Composition of configuration specifications

Parameterized configuration specification

- Subcomponent configuration assignment via parameter only

Evolution of System Design

Component Interface Extension

- Addition of features, flows, etc.
- Assignment of types/classifiers to existing features
  - Assign missing type
  - Override with any type
- Assignment of property values

Myport => MyDataType;
Same as configuration assignment syntax

Component Implementation Extension

- Addition of subcomponents, connections, etc.
- Refinement of existing subcomponents
  - Refine to implementation or configuration

Configuration of a System Design

Configuration Specification elaborates and annotates component hierarchy

- Associated with an implementation/interface via **extends**
- Configuration assignment assigns
  - implementation or configuration to subcomponent
  - Data type or classifier to feature
- Assign property values within existing component hierarchy
- Specify bindings
- Add flow specification
- Add annex subclauses

Design Refinement

Configuration assignment in implementation extensions

- Effectively a refined to but with reach down
  - Assign implementation/configuration for specified interface
  - Override existing implementation with extension
  - Assign interface extensions and their implementations
    - Only for direct subcomponent as it may need to add connections

```
System Top.refined extends top.basic
is
Sub1 => x.i;
Sub2 => y.i;
end;
```

```
System top.basic is
 Sub1: system x;
 Sub2: system y;
End;
```

Configuration of a System Design

Configuration assignment in configuration

- Elaborate and annotate subcomponent substructure
  - Annotate substructure with property values, bindings, annex subclauses
  - Assign component implementation for subcomponent
    - Assigned classifier interface must not be an interface extension
    - Explicit implementation: it becomes the intended implementation that cannot be overwritten
    - If subcomponent was declared with implementation assignment cannot be of an implementation extension

```
configuration Top.config_L1 extends top.basic
is
Sub1 => x.i;
Sub2 => y.i;
end;
```

```
System top.basic is
 Sub1: system x;
 Sub2: system y;
End;
```

Explicit implementation: at least one configuration assignment must have an explicit implementation of the subcomponent only has a type. Otherwise additional configuration assignments can expand design with implementation extensions.

Configuration of a System Design

- Assign configurations for subcomponent with implementation
  - Configurations for ancestor implementation or interface are ok

```
configuration Top.config_L1 extends top.L1impl
is
Sub1 => x.i2;
Sub2 => y.performance;
end;
```

```
System top.L1impl is
 Sub1: system x.i;
 Sub2: system y;
```

```
System x.i is
 xsub1: process subsubsys;
 xsub2: process subsubsys;
```

```
System x.i2 extends x.i is
 xsub3: process subsubsys;
```

```
System y.i is
 ysub1: process subsubsys;
 ysub2: process subsubsys;
```

```
configuration y.performance extends y.i is
 xsub1#Period => 20 ms;
```

Configuration Across Multiple Levels

- Reach down configuration assignments
  - Left hand side resolved relative to classifier being extended

```
configuration Top.config_Sub11 extends top.L1impl
is
  Sub1.xsub1 => subsubsys.i;
  Sub1.xsub2 => subsubsys.i;
end;
```

```
System top.L1impl is
 Sub1: system x.i;
 Sub2: system y.i;
```

```
System x.i is
 xsub1: process subsubsys;
 xsub2: process subsubsys;
```

No ordering assumption in configuration assignments, i.e., second assignment cannot reach into the implementation assigned by the first. Handed by nested assignments (next slide)

Nested Configuration Assignment

- Nested configuration specification
  - Used to configure an assigned classifier
  - Left hand side resolved relative to enclosing extended or assigned classifier

```
System x.l2 extends x.i is
xsub1 => subsubsys.i;
 xsub2 => subsubsys.i;
```

```
configuration Top.config_Sub1 extends top.basic
is
  Sub1 => x.i {
    xsub1 => subsubsys.i;
    xsub2 => subsubsys.i;
  }
end;
Sub1 => x.l2
```

```
System top.basic is
 Sub1: system x;
 Sub2: system y;
```

```
System x.i is
 xsub1: process subsubsys;
 xsub2: process subsubsys;
```

  - Nested configuration for existing subcomponent classifier

```
configuration Top.config_Sub11 extends top.L1impl
Is
Sub2 => {
    ysub1 => subsubsys.i;
    ysub2 => subsubsys.i;
    @EM {* … *};
    #Period => 20 ms
  };
end;
```

Shorter target paths

Annex assignment without explicit configuration specification

Property assignment without target path

Assignment of Configuration Specifications

Specification and use of separate subsystem configurations

- Configuration of subsystems

```
Configuration x.config_L1 extends x.i is
  xsub1 => subsubsys.i;
  xsub2 => subsubsys.i;
end;
Configuration y.config_L1 extends y.i is
  ysub1 => subsubsys.i;
  ysub2 => subsubsys2.i;
end;
```

- Use of configuration as assignment value

```
Configuration Top.config_L2 extends top.basic is
  Sub1 => x.config_L1;
  Sub2 => y.config_L1;
end;
Configuration Top.config_L1L2 extends top.L1impl is
  Sub1 => x.config_L1;
  Sub2 => y.config_L1;
end;
```

Implementation associated with configuration is assigned to the target subcomponent if the original assignment is an interface

Implementation associated with configuration must be the same **or an ancestor** of the original implementation

Configuration of Property Values

Specifying a set of property values

- Property value assignment to any component in the
  - subcomponent path resolvable via the classifier referenced by **extends**
  - Should property value assigned in configuration always be final or should users specify final assignment explicitly and we have a design rule?

```
Configuration Top.config_Security extends Top.config_L2
is
  #myps::Security_Level = L1,
  Sub1#myps::Security_Level = L2,
  Sub1.xsub1#myps::Security_Level = L0,
  Sub2#myps::Security_Level = L1
end;

Configuration Top.config_Safety extends Top.config_L1
is
  #myps::Safety_Level = Critical,
  Sub1#myps::Safety_Level = NonCritical,
  Sub2#myps::Safety_Level = Critical
end;
Configuration x.config_Performance extends x.i
is
  xsub1 => subsubsys.i {
   #Period = 10ms,
   #Deadline = 10ms }
end;
```

=> is changeable
= is final

A configuration specification may only annotate property values or it may also configure and annotate other items.

Composition of Configurations

Combine multiple configurations into new configuration specification

- Define configuration with multiple extends
- Multiple configuration assignments to same subcomponent

Rules

- Associated interfaces must be the same
- Associated implementations must have a single extends lineage
  - The implementation associated with the composite: most descendant
- Only one assigned property value is allowed for any assignment target
  - Two property associations with the same value ok?
  - Local assignment may override or should conflict be error?

```
Configuration Top.config_L2 extends top.config_L1, Top.config_Sub1, Top.config_Sub2 end;

Configuration Top.config_L22 extends Top.config_Sub1, Top.config_Sub2 end;


Configuration Top.config_SafeSecure extends Top.config_L2, Top.config_Safety,
  Top.config_Security end;

Configuration Top.config_SafetySecurity extends Top.config_Security, Top.config_Safety end;
```

Multiple Refinements

Multiple assignments as part of a subcomponent configuration

```
Configuration Top.config_L2 extends top.basic is
  Sub1 => x.config_L1;
  Sub1 => x.security;
  -- shorthand:  Sub1 => x.config_L1, x.security;
  Sub2 => y.config_L1;
end;
```

Multiple assignments to same target within same configuration or by separate configurations.

- Different assigned configurations may contain configuration assignment to same target component
- Associated interfaces must be the same
- Associated implementations must be same or ancestors of explicitly assigned implementation
  - Explicitly assigned in subcomponent declaration
  - Explicitly assigned by one of the configuration assignments
    - If derived from configuration users can add an implementation extension through a ocnfiguration
- Only one property value assignment is allowed for any assignment target
  - Property value assignments in configuration specifications are “final”

Multiple Configuration Assignments

Multiple assignments as part of a configuration

```
Configuration Top.config_L2 extends top.basic is
  Sub1 => x.config_L1;
  Sub1 => x.security;
  -- shorthand:  Sub1 => x.config_L1, x.security;
  Sub2 => y.config_L1;
end;
```

- Different assigned configurations may contain configuration assignment to same target component and may do so at different levels of the hierarchy
- Associated interfaces must be the same
- Associated implementations must be same or ancestors of explicitly assigned implementation
  - Explicitly assigned in subcomponent declaration
  - Explicitly assigned by one of the configuration assignments
    - If implementation is derived from collection of configurations (deepest in extends lineage) users can add an implementation extension through a configuration
- Only one property value assignment is allowed for any assignment target
  - Property value assignments in configuration specifications are “final”
  - Alternative: Rules about override order as we have for implementations and implementation extensions

Composition of Flow Configurations

Adding in end to end flows

- End to end flows may be declared in a separate classifier extension
- No conflicting end to end flow declarations

```
System Top.flows extends top.basic
is
  Sensor_to_Actuator: end to end flow sensor1.reading -> … -> actuator1.cmd;
End;

Configuration Top.config_full extends Top.config_L2, Top.flows end;
```

- Flow specs for end-to-end flow targets may be declared in separate configurations
- Flow implementations for intermediate flow targets may be declared in a separate configurations

```
configuration X.flowspec extends X
is
  outsource: flow source outp;
End ;
configuration X.flowsequence extends x.i
is
  outsource => flow subsub1.flowsrc -> … -> outp;
End;
```

Configuration/composition of Annex Subclauses

Adding in annex specifications

- Annex subclauses may be declared in a separate classifier extensions
- Different annex specifications may be added

```
System Top_emv2 extends top is
Annex EMV2 {**
  use types ErrorLibrary;
  …
**};
End Top_emv2;
```

```
Configuration Top_emv2 extends top
@e {* use types ErrorLibrary; *};
End Top_emv2;
```

Use @ consistent with reference path usage

```
Configuration Top.config_full extends Top.config_L2, Top.flows, Top_emv2 end;
```

Inherited annex subclauses based on classifier **extends**

- Automatically included
- Extends override rules of annex apply

Separate extensions

- No conflicting declarations

Parameterized Configuration

Explicit specification of all choice points

- Configuration of subcomponents via configuration parameters only
  - Assignment of formal parameter to one or more subcomponents
- No direct configuration assignment to subcomponents by user
- Substitute the type of the parameter specification

```
Configuration x.configurable_dual(replicate: system subsubsys) extends x.i is
  xsub1 => replicate;
  xsub2 => replicate;
end;
```

Configuration parameter classifier must the same or an ancestor of the assignment target

Similar to V2 prototype but we map parameter to targets instead of requiring all targets to reference prototype

Usage

- Supply parameter values

```
Configuration Top.config_sub1_sub2 extends top.i
is
  Sub1 => x.configurable_dual( replicate => subsubsys.i );
end;
Configuration x.configured extends x.configurable_dual( replicate => subsubsys.i )
end;
```

Configuration parameter actual must match
- an implementation/configuration of the specified interface
- a configuration of the specified implementation or its ancestor or interface

Explicit Specification of Candidates

- Explicit list of candidates

```
Configuration x.configurable_dual(securityProperties: system {
subsubsys.sec1, subsubsys.sec2 } ) extends x.i is
  xsub1 => securityProperties;
  xsub2 => securityProperties;
end;
```

Property Values as Parameters

Explicit specification of all values that can be supplied to properties

- Values that can be used for different properties of the same type
- Values for specific properties

```
Configuration x.configurable_dual(TaskPeriod : time ,
    TaskDeadline : #Deadline) extends x.i is
  xsub3.T1#Period = TaskPeriod;
  xsub3.T1#Deadline = TaskDeadline;
end;
```

Specify value time vs. property to be assigned

Xsub2.T1 must exist in x.i

Usage: Supply parameter values

```
Configuration Top.config_sub1_sub2 extends top.i is
  Sub1 => x.configurable_dual(
    TaskPeriod = 20ms, TaskDeadline = 30 ms );
end;
```

Via configuration specification as parameter

- Collections of property value assignments
  - Consistent set of property values
- Explicitly specified collections to choose from

```
Configuration x.configurable_dual1(securityProperties: system subsubsys.i ) extends x.i is
  xsub1 => securityProperties;
  xsub2 => securityProperties;
end;
Configuration x.configurable_dual2(securityProperties: system { subsubsys.sec1, subsubsys.sec2 } )
extends x.i is
  xsub1 => securityProperties;
  xsub2 => securityProperties;
end;
```

Complete Configuration

- Finalizing choice points of an existing implementation or configuration

```
Configuration Top.config_L0() extends top.basic end;
```

Annotating Parameterized Configurations

- Users should be able to add “missing annotations”
  - Additional flows, error model specification, property values
  - User can declare extensions of parameterized configuration that contain the annotations
    - Configuration assignments can reach into component with parameterized configuration but can only add missing property values, flows, EM
  - User can compose multiple such annotations into the configuration
    - As new configuration or as part of each usage

```
Configuration Top.L0_Security extends Top.config_L0
is <security properties> end;
Configuration Top.L0_Safety extends Top.config_L0
is <EMV2 subclause for Top> end;
```

Configuration Assignment Patterns

Assignment of configuration to classifiers (all instances of classifiers)

Match&replace classifier/data type within a scope

- Match classifier in subcomponents and features, data types in features

```
Configuration FlightSystem.secure extends FlightSystem.TripleGPS is
 all(GPS) => GPS.secure;
 all(Dlib::dt) => Secure.securesample;

  #Period *=> 20 ms;
end;
```

Assign GPS.secure for all subcomponents with interface GPS within scope of FlightSystem.TripleGPS

Assign type Secure.securesample for all features with type dt within scope of FlightSystem.TripleGPS

Period for all elements within scope of associated implementation that require a Period

```
Package FS
Import mine::*;
System FlightSystem.TripleGPS
is
 gps1: device GPS;
 gps2: device GPS;
 gps3: device GPS;
End;
End;
```

```
Package mine
Device interface GPS
is
  inp1: in data port Dlib::dt;
  outp1: out data port Dlib::dt;
End;
Device GPS.secure is
```

Reusable Configuration Patterns

Match&replace within the scope the configuration pattern is assigned to

- Match classifier or primitive type in subcomponents and features
- Configuration without extends can be

```
Configuration GenericPattern is
  all(Mine::Sensor) => Sensor.Settings;
  all(Dlib::dt) => Secure.securesample;
  all(Mine::GPS) => GPS.secure { #Period => 50 ms};
  all(Your::GPS) => { #Period => 50 ms};
end;
Configuration Sensor.Settings extends Sensor.impl is
  #Period => 50 ms;
  reading#Data_Size => 20 Bytes;
end;
```

- Assign configuration pattern to subsystems

```
Configuration AvionicsSystem.Dual is
  FlightSystem1 => FlightSystem.primary, FlightSystem.secure;
  FlightSystem2 => FlightSystem.primary, GenericPattern ;
  BackupFlightSystem => FlightSystem.backup, SimpleGPS.config;
```

Assignment of Configurations to All Instances

Configurations as annotations for all subcomponents of a given classifier

- Example: EMV2 configuration for a classifier
- Assign annotations individually

```
Configuration AvionicsSystem.Dual is
  FlightSystem2 => FlightSystem.primary;
  all(Mine::Sensor) => Sensor.emv2;
  all(Mine::GPS) => { @em {* … *};};
```

- Specify collection of EM annotations

```
Configuration FlightSystemEMV2 is
  all(Mine::Sensor) => Sensor.emv2;
  all(Mine::GPS) => { @em {* … *};};;
end;
```

- Assign configuration pattern to subsystems

```
Configuration AvionicsSystem.Dual is
  FlightSystem2 => FlightSystem.primary, FlightSystemEMV2 ;
```

```
Configuration Sensor.emv2 extends
Sensor
@em {* use types ErrorLibrary; *};
End Top_emv2;
```

AADL: Bindings and Resources

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Bindings between System Hierarchies

AADL supports a (primary) containment hierarchy

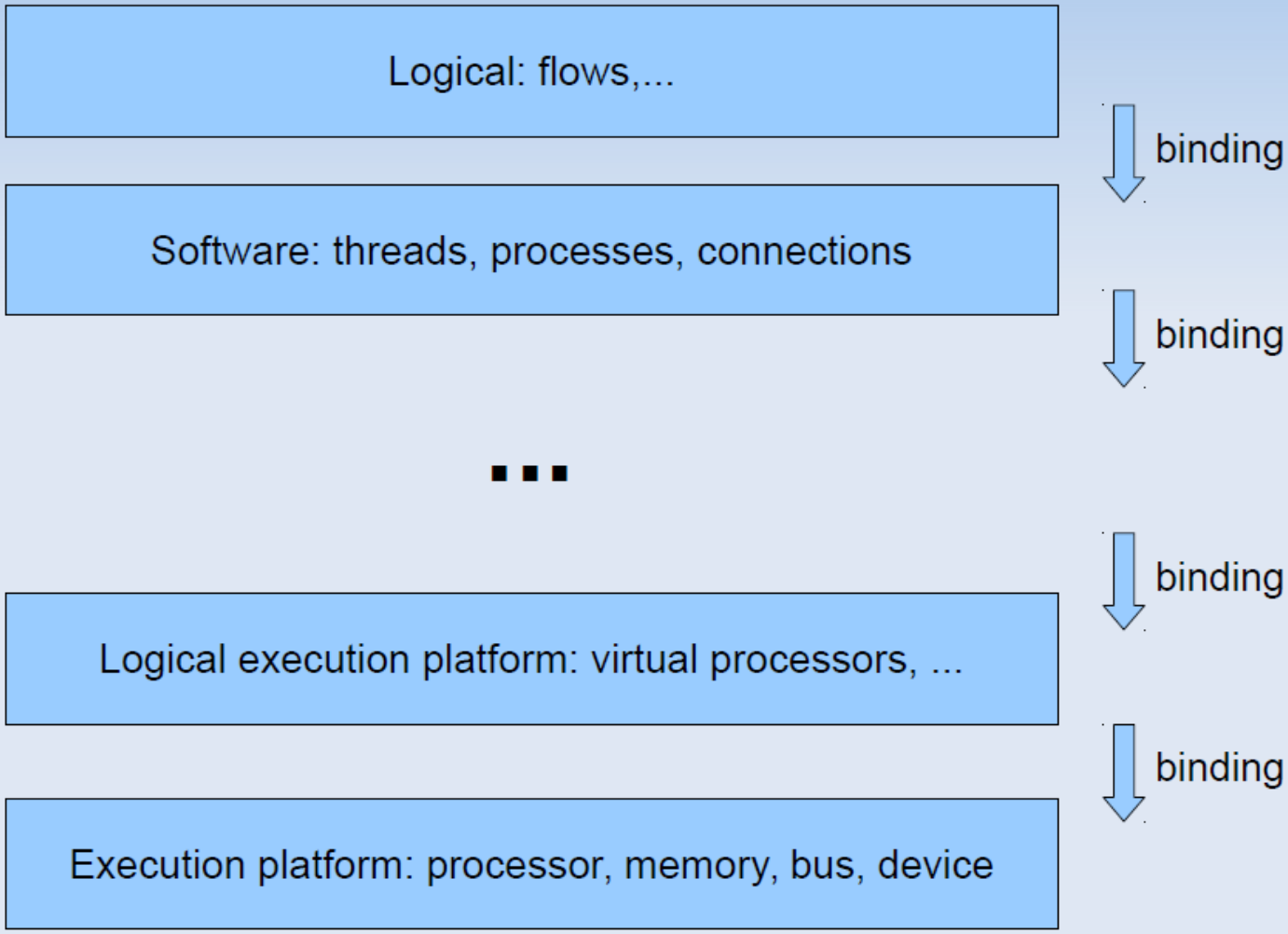
Semantic connections represent flow between and within subtrees

- Managed interaction complexity by requiring connections up and down the hierarchy to restrict arbitrary connectivity
- Note: for subprogram calls we offer both a connection and a mapping specification

Deployment bindings (aka. allocations) are a mapping from elements of one subtree to elements of another subtree

- The subtrees represent different virtual machine layers with the lower layer typically representing resources to the higher layer
- Bindings represent resource allocation

Multi Layer Architecture



Issues in Current Binding Approach

Bindings are currently expressed by properties

Binding related properties are not distinguishable from others

- Properties that express bindings
- Properties that relate to bindings

EMV2 propagation paths are derived from bindings

- binding points currently are identified by special keyword

Binding properties reach down the instance containment hierarchy

- A primary driver for introducing contained property associations

Resource Flow and Resource Allocation

Resource flow within an architecture: follows interface rules

- Continuous: Electricity, fluid flow, … Discrete: data samples, messages
- Directional flow with continuous characteristics: producer –> consumer
- Resource type represented as type (or abstract component type)
  - Annotations for discrete or continuous flow
- Fan out/in of flow “volume”
  - multiple features & multiple connections from one feature

Resource allocation/binding: Across different architecture layers

- Resource usage that needs to be allocated/scheduled
- SW to computer platform
- Logical to physical
- Resource capacity and demand as provides and requires features
  - Feature acts as binding point: resource type as classifier
  - Multiple

Binding Specification Proposal

Binding Types

Binding points

- Properties, constraints

Binding instances

- Specify source and target of binding

Binding Types

Define binding type

- User defined name, sets of source and target types

```
Binding ProcessorBinding : union( thread, thread_group, process) ->
union (  processor, virtual processor )
```

Predefined binding types

- *FunctionalBinding* to bind elements of a functional architecture to elements of a physical architecture.
- *ThreadBinding* to bind a thread to a processor, or to a virtual processor which in turn is the source of a processor binding to ultimately a processor.
- *CodeBinding* to bind source code associated with processes or threads to memory components.
- *DataBinding* to bind data components as well as source code data including stacks and heaps.
- *ConnectionBinding* to bind a connection to a flow sequence in hardware platform, or a virtual hardware platform whose elements are ultimately bound to a physical hardware platform.

Binding Instances

Deployment binding

- Binding source and target are components
  - Explicit binding type specification
  - Binding point identified by binding type

```
Configuration AS.deploymentconfig extends AS.systemconfig is
Binding1 : ProcessorBinding binding Appsys.sub.proc.thread1 ->
platform.node.cpu1;
Binding2 : ProcessorBinding binding Appsys.sub.proc.thread2 ->
platform.node.cpu2;
```

Binding Point Specification

As **provides/requires binding** feature

- As abstraction for targets contained inside a system
- Similar to access features

One component can have multiple binding points

- Binding points of different types
  - Need/provision of different resources, e.g., at system level
- Binding points of same type
  - E.g., subsets of total resource capacity

Binding Point Specification

Explicit in features section:

- Directional features to be used as source or target of binding
- Identify type of binding
  - Binding type

```
thread task1 is
RequiredMemory: Requires binding Storage;
RequiredCycles: Requires binding Cycles;

System LinuxBox is
ProvidedDisk: Provides binding Storage;
ProvidedCycles: Provides binding Cycles;
end;

type Cycles : union (thread) -> union {processor, virtual processor };
```

Visibility of Binding Points

How far down can the allocation declaration reach

- Parameterized configuration as boundary for external use

Map binding point at configuration interface to component(s) in implementation that manage or represent resource

```
System ASplatform
ComputeCycles: provides binding #ProcessingCycles;
  Storage: provides binding #cache;
End ASPlatform;
ASplatform.boundconfig configures ASPlatform.impl {
ComputeCycles -> cpu,
Storage -> Cachememory
};
```

Connection Bindings

Currently: sequence of target elements

Connection acts as binding point

- Propagation identifies connection by name

```
connections
Conn1: port sub1.p1 -> sub2.p1 Requires XferBandwidth;
Conn2: abstract sub1.fe1 -> sub2.fe1 Requires WattsPerHour;
```

Platform flow sequence as binding target

- Expressed by end to end flow declaration
- Source and destination of ETE flow must match binding target of connection source and destination
- Each element of the flow has binding point of matching type

Virtual bus as binding target

- Virtual bus itself needs to be bound to a sequence of items => ETE flow

Flow sequence as closed platform configuration binding point

- How to expose platform internal ETEF as external binding point? => access to virtual bus that is bound to ETE flow

Qualified and Quantified Resources

Resource specification

- Via property on binding point or providing/requiring component
- Quantified via numeric type
- Qualified via enumeration type

```
thread task1
RequiredCycles: Requires binding Cycles {#Resources::ProcessingCycles => 200
MIPS;};
thread task2
RequiredInstructionSet: Requires binding IntelX86 {#Resources:: InstructionSet =>
X86;};


Processor IntelX86 is
  #Resources::ProcessingCycles => 1200 MIPS;
  #Resources:: InstructionSet => X86;

Property InstructionSet : enumeration ( X86, ARM, RISC) applies to Processor,
virtual processor, system, binding point;

Property ProcessingCycles : real units CycleUnits applies to Processor, virtual
processor, system, binding point;
```

Binding Type Multiplicity

1-to-n binding Alternatives (*):

- example – multiple cores

```
Binding MultiCoreBinding :
  union( thread, thread_group, system ) ->* union ( processor)
```

Bind to virtual processor (resource) or enclosing system:

- It schedules multiple cores

Flows

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Flow Specifications and Sequences

Flow specification

- Flow source, sink, path
- For features and element in named interface features

Flow implementation

- Assignment of flow sequence to flow specification

```
interface control is
insignal: in port;
outaction: out port;
processflow: flow path insignal -> outaction;
end;

process control.impl is
    dofilter: thread filter;
    docompute: thread compute;
    extin: mapping insignal => dofilter.insignal;
    ftoc: connection dofilter.outsignal -> docompute.insignal;
    extout: mapping outaction => docompute.outsignal ;
    processflow => flow dofilter.filterpath -> ftoc -> docompute.computepath ;
end;
```

End to end flow sequence

```
controltoactuate: connection processing.outaction -> actuate.inp;
etef: end to end flow sense.reading -> sensetocontrol-> processing.processflow -> controltoactuate -> actuate.action;
```

Flow Sequence Specification

Currently (V2)

- Alternating component.flowspec and connection
- Alternating component and connection
  - Flow spec inferred from connection end points
  - Flow related property inferred from value assigned to component

Additional flexibility

- Component.flowspec sequence only
  - Infer connections
- Connection sequence only
  - Infer component and flow spec
- Reach down for components without flow spec
  - E.g., nested subcomponents
- End to end flow starting and endpoint
  - Assignment of flow sequence as for flow spec
  - We infer the connection and component flow spec instances

Flows at Platform Level

- Flow sequence as target of connection binding

Flow Graphs

Objective: Forward and backward traceability

- Forward: variation in latency/age at all end points
- Backward: variation in latency/age from all contributing sources
- Auto-generate from flow specs and connections
  - As we do for propagation graphs

Fan-in/out logic for each component (Merge point semantics)

- Fan in across ports
  - Flow path with multiple inputs (AND)
  - Separate flow paths as alternatives (OR)
  - Pre and post conditions on input/output
- Interpretation of BA logic
  - Input on several ports triggers dispatch
  - Fan in at single port with multiple incoming connections
- Fan out to multiple ports
  - All vs. alternative (Not needed) The fan-in takes care of everything. John Hatcliff discussion on canonical)

Flow Patterns

- End to end flow spec (endpoints only)
  - Etef1: **flow** sys.proc.thread1.fsrc -> * -> sys.proc2.thread4.snk
  - Endpoint spec as reference down the hierarchy
  - Infer all possible paths
  - Etef2: **flow all**(GPS.fsrc) -> * -> **all**(Displays.view)
  - Infer all instances, i.e., all paths between any GPS and DIsplay
- Flow impact
  - Impact : **flow** sys.gps.signalsrc -> *
  - **All(**FlowSpec) -> *
- Flow contributors
  - Effector : **flow** * -> sys.actuator.cmd
  - Effector : **flow** * -> sys.actuator.cmd

AADL V3 Property Language

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Property Definitions

Define in packages

Utilize unified type system

- No more **aadlinteger**, …
- Record, list, set, map
- Union of types:
- Integration of proposed Units system (ISO, SysML)

Identify assignment targets (V2 **applies to**)

- No need to list enclosing categories for **inherit**
- Component categories
- Specific classifiers
- Other model elements
- Use type system to express model element types, classifiers

Property Association

- Property reference always with #

```
process interface LocatorProcess
is
#Period => 20;
end;
```

- Properties on classifier elements
  - Directly attached
  - Via model element reference (aka contained property association)

```
interface subsub
is
    p1 : in port signal ;
    p2 : in port date { #Size => 3; };

    p1#Size => 3;
end;
```

Property Values

Property value can be overridden many times in V2

- As part of definition
- Inherited from enclosing component
- Inherited from interface (ancestor)
- Inherited from implementation (ancestor)
- Inherited from subcomponent definition
- Multiple layers of contained property associations

Property Values in V3

Assignment in interface or implementation

- modifiable assignment **=>**
- final assignment **=**

Value determination potential options

- Property on component
  - Interface and interface extension (#Size => 1 on classifier)
  - Implementation, implementation extension (#Size => 1 on classifier)
  - Configuration, configuration extension (#Size => 1 on classifier)
  - Local subcomponent(**{** #Size => 2; **}** and sub1#Size => 2;)
  - Configuration assignment (#Size => 1 on classifier)
  - Configuration assignment nested {#Size => 2; }
  - Contained property associationouter overrides inner (reach down)
- Values on model elements (features, connections, etc)
  - Local (**{** #Size => 2; **}** and feat1#Size => 2;)
    - Interface and interface extension
    - Implementation and implementation extension
    - Configuration and configuration extension
  - Configuration assignment (feat1#Size => 1 in classifier)
  - Contained property association outer overrides inner (reach down)

Property Values in V3

Simple approach

- Component and element properties specified as part of “spec”
  - Defined through classifiers during design
  - #P for instances of classifier
  - {} or identifier#P for all directly contained model elements except subcomponent.
  - Override rules according to extends hierarchy of interface, implementation
- Configurations finalize a design
  - Final property value assignment in configuration specification
  - Reach down property associations in implementations
    - Containment with first element a subcomponent

Property Values in V3

Conflicting assignment

- Composition of interfaces
  - #P assigned values: only one or must be the same value
- Multiple assignment through reach down & multiple configurations
  - Assignment is final: only one or must be the same value

Scoped Default Property Values in V3

V3: Scoped value assignment

- #Period *=> 20ms;
- Scope of configuration, implementation, or interface with assignment
- Used if no value assigned explicitly for contained model element
- Replaces **inherit** in V2

Property Association in Annexes

Syntax in context of an annex

- FailStop#Ocurence => 2.3e-4;
- ^Process[1].thread2@Failstop#Occurrence => 2.3e-5;
  - ^ escape to core model as context
  - @ enter same annex type as original
  - @(BA) enter specified annex: if we have annex specific properties in the annex rather than core we may not need this
  - [x] array index

Mode specific property value assignment #8

- Currently: => 2.3e-5 **in modes (**m1), 2.4e-4 in modes (m2);
- => { m1 => 2.3 , m2 => 2.4 };
- Event#Occurrence.m1 =>
- See also error type specific property value and binding specific value
  - Use map type: mode, error type, binding target as key
  - Syntax for identifying map key in path (.)

Property Set

Definition of property set

- List of property references
- Property set can be listed as element of a set
- Same property reference can be in multiple sets

```
Periodic : properties {
  Dispatch_Protocol => constant Periodic,
  Period, Deadline, Execution_time
};
```

```
GPSProperties :  properties {
  Period, GPSPropertyset::Sensitivity,
GPSPropertyset::Hardening
};
```

Usage: Analysis specific property set

- Must be present for analysis
- Analysis supporting multiple fidelities
  - Minimum, maximum set
  - Precondition vs. validation

AADL 3
Type System and Expression Language

Lutz Wrage

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Type System Unification

Unification of type systems and expression languages (Peter, Lutz*, Alexey, Brian, Serban)

- Data Components
- Property Types
- Classifiers
- Annexes
  - Resolute, AGREE
  - Data Modeling
  - EMV2
  - BA, BLESS
- ReqSpec
- Scripting languages (Python)

Current Composite Types

AADL 2.2

Property types

- Range of
- List of
- Record

Data implementations

No operations available except

- List append (+=>)
- Boolean operations

Property expressions provide syntax for literals

ReqSpec adds expressions, uses type inference

Current Usages of Types

Application data that occurs in the modeled system
- Data subcomponents
  - Shared data
  - Local variables in threads and subprograms
- Data communicated via data and event data ports

Information about the modeled system and individual components
- Properties

Mixture of models and properties
- Component classifiers and model elements as properties
  - Bindings
  - Specify constraints, e.g., `Required_Virtual_Bus_Class`

Additions in annexes
- Resolute: sets
- EMV2: error types and type sets, error types can have properties
- BLESS

Type System / Expression Language Goals

Provide types for

- Properties
- Features, e.g., data ports
- Data components
- Error types(?)

Support

- Specification of dependencies / constraints between properties
- Selecting model elements in configurations
- Structural analysis of instance models
  - Similar to Resolute
- Requirement specification
  - Similar to ReqSpec

Do we need structural analysis of declarative modes?

Type System Unification Approach

Base types

- Numeric, Boolean, String
- Enumeration, Unit
- Category (`thread`, `processor`, etc.), Classifier, Model Element
- Range of Numeric (`Compute_Execution_Time => 10ms .. 15ms`)

Composite types

- List (ordered sequence of arbitrary length)
- Set (unique elements)
- Record (named fields) / Union (named alternatives)
- Tuples (unnamed fields)
  - Convenient for multiple return values from a function
- Map
  - Modal and binding specific property values in AADL 2.2 are (almost) maps
  - Error type specific property values
- Array (ordered sequence of fixed length)

- Bag (?), ~~Graph~~

Type System Unification Approach

Properties on types

- Information about representation
  `integer {data_size => 16bit}`
- Range of valid values
  `integer {range => 10 .. 20}`

Useful for code generation and analyses the looks at data size (in memory or on a bus)

Properties are ignored for type checking purposes

User Defined Types

Users can create named types

- `type byte: int { range => 0 .. 255 }`
- `type otherByte: byte { data_size => 16bit }`

- ```
  type sensed: record {
    value: integer
    timestamp: integer
  }
  ```

```
type sensed2: record {
  value: integer
  timestamp: integer
}
```

Is a type name just a shorthand, or is it a new type?

- Structural equality as the default
- “Opaque” types would complicate the expression language
```

Subtypes

Subsets of numeric types (or enumerations?)

- Range constrained Numeric
  e.g., `integer [100 .. 120]`
- Could be considered special syntax for a property on a type
  e.g., `integer {range => 100 .. 200}`

Subset constraints are difficult to maintain for expressions

- Simple assignments are easy to check
- 2 * integer[100 .. 120] results in integer[200 .. 240]
- Sin(integer[100 .. 120]) results in (not quite) real[-1.0 .. 1.0]

Type checking should ignore range constraints, maybe except for simple assignments

Type Extension

Type extension

- Exists for classifiers (including data components in AADL 2)
- Records
  - Add fields
- Unions:
  - Add fields to one or more variant(?)
  - Add variants
- Add properties to any type
  - `byte` is subtype of `integer`
  - Then: `list of byte` is subtype of `list of integer`
- Change property values?
- “refinement”?
- Should there be a complete type hierarchy with something like `Object` as the root?

Do we really need type extension for data types?

Expression Language: Literals

Numbers, strings, boolean true/false as in AADL 2

- Automatic conversion from integer literal to real value

Range literals

- In OSATE we run into a lexer limitation: `2..3` vs. `2 ..3`

Enumeration and unit literals

- Qualified name: <package>.<enum type>.<enum literal> e.g., `myenums.signaltype.RED`
- Need to import enumeration and unit literals in order to use them

Collections

- Mirror declaration
- `list (1,2,3);`
- `record (intfiled = 1, boolfield = true);`

Expression Language: Operations 1

Boolean
- And, or, not, …

Numeric values
- +, -, *, /, div, mod

Ranges
- Union, intersection, contains

Enumerations
- Assign a numeric value to enumeration elements?
- Consider them ordered?

Units
- Get conversion factor

Strings, List
- Append, substring, …

Records
- Access a field value

Union
- Access field depending on variant tag

Expression Language: Operations 2

Set

- union, intersection, contains

Generic collection operations

- forall, exists, filter, fold
- Look for inspiration in existing collection library and copy

Classifiers

- Extends, get extended, get all extending, …
- → methods defined in the AADL meta-model

Named elements

- Get name, get classifier, get all subcomponents, …
- → methods defined in the AADL meta-model

Prototype Implementation

Expression Annex for AADL2

Implemented

- Most types
- Some type checking
- Subset of expressions
- Initial expression evaluation

- No units yet

Measurement Units

Represent a (physical) quantity as a number with a dimension

- Length, Time, Mass, Force

Dimension has associated measurement units

- Length – **m**eter (SI base unit)
- Time – **s**econd (SI base unit)
- Mass – **k**ilo**g**ram (SI base unit)
- Force – **N**ewton (Derived: $1\ N\ =\ 1\ \frac{kg \cdot m}{s^2}$)

Different unit systems

- SI vs. Imperial
- Non-physical quantities, e.g., bit, byte
- Other: minute, day, year; rpm, angle, …

Users must be able to define new units

Unit Definition 1

Defining dimensions and corresponding measurement units

- Dimension as variation of enumeration types
  - `type LengthU: unit (cm, m = 100 * cm, …)`
  - `type TimeU: unit (s, ms = s / 1000, …)`
  - `type USLengthU: unit (in, ft = 12 * in, …)`
- Similar to AADL2
- Similar to compound type declarations (records, lists, etc.)

Literals with units

- `100 ms`
- `12 [ms]`

Type declarations with units

- `type LengthType: real unit LengthU`
- `type LengthType: real [LengthU]`

Unit Definition 2

Property definition

- Value is a physical quantity
  - `property distance: real unit USLengthU`
  - `property distance: real [USLengthU]`

  - `distance => 2.5 [in]`

- Value is a unit, e.g., to document the unit of the data on a data port
  - `property dataUnit: LengthU`

  - `dataUnit => [m]`

Standard Metric Prefixes

Metric prefixes

- Base 10: **c**enti, **m**illi, micro **µ**, **d**eka, **k**ilo, **M**ega
- Binary: **Ki** (2^{10}), **Mi** (2^{20}), **Gi** (2^{30})
- These are case sensitive, one is a greek letter
- Not distinct from units: **m**eter vs. **m**illi

Convenient to use them with any unit without repeatedly defining the conversion factor.

Use syntax to separate metric prefix and unit name

- `1 [k'g], 12 [m's], 640 [Ki'byte]`

Only with base units

- If ms is defines as derived (ms = s/1000) the `1 [k'ms]` should not be valid

Unit Expressions 1

Avoid units names such as `KBytesps` (as we have in AADL 2)

Allow expressions for derived units

- `[k'g * m / s^2]`

Unit expressions are written in `[]`

- `speed == 12 [m/s]`

Simple unit may be written with or without `[]`

- `latency == 10 m's` or `latency == 10[m's]`

Allow only multiplication, division, and exponentiation

Defining a derived unit type

- **`type`** `ForceU =` **`unit`** `(N = [k'g * m / s^2])`

Unit Expressions - 2

Convert between numbers and quantities

- `val x = 1` x is an integer
  `val y = (x + 1)[s]` y is an integer with a unit: 2s
  `val z = y in [ms]` z is an integer: 1000

Calculation with units

- `10 N / 2.5 k'g == 4.0 [m / s^2]`

Unit Definitions and Usage

Derived units with unit expressions

- `type MassU: unit (g)`
- `type SpeedU: unit (LengthU / TimeU)`
- `type ForceU: unit (N = k'g * m / s^2, …)`

Type declarations with units

- `type SpeedT: real [SpeedU]`
- `type ForceT: real [ForceU]`
- `type OtherSpeedT: real [LengthU / TimeU]`

Property definition

- `property speedUnit: Speed`
- `speedUnit => [m/s]`
- `property force: ForceT`
- `speed => 2.5 [k'g * m / s^2]`

Backup

Old slides

Data Subcomponents

Interface + data type

Generic component + data type as property

Type System Usage

Port types

- P1: **in port** Temperature;

Data components

- DataObject: **data** Personel_Record;
  - Subtype substitution/match (Type_Extension)

Properties

- Property definitions reference types

Data Annex

- Characterization via properties vs. partial specification
- **Data** personel_record { Data_Representation => Struct; };
- Personel_Record: **type record** () { Source_Name => PersonnelRecord;};

- Personel_Record: **refined to type record** (first: string; last : string;);

Representation of Transferred Data

Example

- BodyTemperature: **type** integer [30..50 C] **units** TemperatureUnits;
- P1: **out port** BodyTemperature;

Is unit included in transferred data or is a unit assumed?

Non-zero reference point for transferred value

Transfer representation may be different from in memory representation

Alternatives:

- Protocol specification
  - As virtual bus
  - Mapping into bit representation (see 429 protocol example in SAVI demo)
- Assumed unit as property on port

Representation of Types

Example

- BodyTemperature: **type** integer [30..50 C] **units** TemperatureUnits;
- P1: **out port** BodyTemperature;

Digital representation

- Base_Type property in Data_Model
- Associated with type or with port

Physical representation

- Dynamic behavior
- Specified as part of type or specific to each use site
  - Associated with feature

AADLv3
Part 2 – Static Semantics

Jerome Hugues

About Part 2 semantics

Goal is to present the static structure of an AADL model

Architecture modeling through containments, connections and bindings

Component categories description

New component category: virtual memory

Textual description + legality rules + default interpretation of legality rules

Rule of thumb: any AADL tool **must** support these rules: they are the core of the AST analysis and name resolution mechanisms

Part 2 organization

For each component category

- Design intent: purpose of this component category, interaction with other categories
- Legality rules: allowed features categories, subclauses, etc.
- Semantics: text-based interpretation of the legality rules *only*

No section on properties -> moved to P4 for clarity

If list of properties per category required, could be provided as appendix to the document to facilitate cross-referencing

Component category

4 categories

- Software: `thread, thread group, process, data, subprogram, subprogram group;`
- Execution platform: `memory, virtual memory, processor, virtual processor, bus, virtual bus, device`
- Generic: `abstract`
- Composite: `system`

Change from v2:

- Suppression of call sequences (use subcomponents instead)
- Introduction of virtual memory
- Update to match AADLv3 new concepts for features, bindings

On model completeness

Must be clear on the requirements on model completeness, and prevent overspecification

> e.g. in v2, 13.3 (5) “All software components for a process must be bound to memory components that are all accessible from every processor to which any thread contained in the process is bound. “

For code generation, this brings no value for some RTOS (e.g. RTEMS, FreeRTOS), but is required for others (e.g. ARINC653)

$\Rightarrow$ Model completeness is a per-objective issue

Objective: what is my objective when modeling? What type of credits to I want to gain from this modeling activity?

On component categories

Question: should we split Execution Platforms (virtual bus/processor) from Hardware elements?

- Rationale: VB/VP/VM control executions of software, using resources provided by hardware elements, required by software elements.
- They act as a resource broker
  - Virtual Bus: communication protocols, message arbitration
  - Virtual Processor: scheduling policy
  - Virtual Memory: ? Security e.g. red/write/execute

Memory vs Virtual Memory (June 16)

- Separation of physical and logical concerns
  - Binding of logical to physical
- Memory
  - Storage with binding points
  - Need for representing different section of memory addresses: binding points have properties to indicate base address and range (size)
  - Memory binding point on devices can model device registers without requiring memory subcomponents.
- Virtual memory roles
  - Represent logical addresses that are mapped to addresses in different components in the platform
  - Logical resource with capacity/budget
  - Logical containment regions or segments of address space
- Memory as system (platform) subcomponents
  - Subcomponents as binding points
  - Memory system architecture with connectivity via bus
  - Platform with memory and processor

Virtual Memory component category

Question: what is the static semantics of virtual memory ?

- Define the logical view of memory, a process perspective ?
- Binding VM to device/system to capture flash storage?

Or is data/process sufficient to achieve the same goal?

- Data being structured memory, process being protected memory

Most of the semantics of VM seems driven by properties

- Stack/BSS/Heap/Code memory only ?
- Read/write/execute ?

Use cases:

- Binding Virtual Memory to Memory (Hardware) as a deployment
- Binding Process/Thread/Data/Subprogram to VM ?
- VM inside VM ?

Virtual memory and composition

Ultimately:

- A process (e.g. software partition) is bound to both VM and VP
- VP is then bound to a processor, VM to a memory
- And finally processor and memory are connected

Otherwise, architecture is inconsistent

But if some of these bindings are not done, model is “just” incomplete w.r.t. some verification objectives

Role split between virtual processor and processor (June 16)

Virtual Processor :- OS + scheduler configuration, security policies, health monitoring, contribution to fault propagation, etc.

Processor :- physical CPU (chip, core, etc.), contributes to fault propagation, ?

For code generation purpose, having only a virtual processor makes sense: you target an OS, the actual physical CPU is irrelevant, and deployment to this physical one depend on the deployment strategy

For safety analysis, one may want to have threads (functions) and processors. Virtual processors may or may not be required depending on its contribution to errors propagation/containment.

Definition of Virtual Processor

AADLv2.2 6.2 (1)

“A virtual processor represents a logical resource that is capable of scheduling and executing threads and other virtual processors bound to them. Virtual processors can be declared as a subcomponent ~~of a processor~~ or another virtual processor, i.e., they are implicitly bound to the ~~processor or~~ virtual processor they are contained in. Virtual processors can also be declared separately, that is as a subcomponent of a system component, and explicitly bound to a processor or virtual processor. “

Processor, device as systems

System are composite elements.

Processors and devices are also systems when we open the box.

Refining a processor (black box) into system (white box) violates type system (no type promotion), natural modeling step that must be supported

Using Implemented_As does not work: no feature connection, breaks mode or fault propagation

Processor, device as systems

Proposal to extend the definition of processor

AADLv2.2 6.1 (1)

"A processor is an abstraction of hardware and software that is responsible for scheduling and executing threads and virtual processors that are bound to it. A processor also may execute driver software that is declared as part of devices that can be accessed from that processor. Processors may contain memories and may access memories and devices via buses."

AADLv3

"A processor is a hardware system that is responsible for providing access to resources it controls to threads and virtual processors. A processor may control and give access to subcomponents: internal processing cores, memory elements (e.g. cache, flash storage) or internal devices."

Containment clarifies propagation of faults, mode changes, etc (loss of processor -> loss of all subcomponents, loss of subcomponents degrade processor capability), preserves modularity and type system

Processor, device as systems

Similarly, for devices

AADLv2.2 6,6 (1)

“A device component represents dedicated hardware within the system, entities in the external environment, or entities that interface with the external environment. [..] Devices may internally have a processor, memory and software ~~that can be modeled in a separate system declaration and associated with the device through the Implemented_As property~~.”

AADLv3

“A device component represents dedicated hardware within the system, entities in the external environment, or entities that interface with the external environment. [..] Devices may internally have a processor, memory and software **modeled as subcomponents**”

Rationale similar to processor: use containment to ease failure analysis, preserve types, etc.

Virtual bus/bus

No major change for the moment

Wait for security discussion to see if improvements are needed

e.g. composition of protocols (cyphering, authentication)

Patterns to capture execution of protocol code ?

- Attach an entrypoint on the bus to capture the code to run ? .. but not sufficient if producer?/consumer uses different APIs
- Use subprogram access so that a VB can requires subprograms provided by a VP (OS/library) ?

About data types

(Pending text on types system)

About call sequences

Change highlight: suppression of call sequence

Rationale: CS breaks symmetry in components hierarchy

```
--  AADLv2
thread implementation P.Impl
calls
   Mycalls: { S : subprogram Spg; };
connections
   parameter S.I -> I;
end P.Impl
```

```
--  AADLv3
thread implementation P.Impl
subcomponents
   S : subprogram Spg;
connections
   parameter S.I -> I;
end P.Impl
```

Directly reuse rules for subcomponents

Subprogram instances reside in attached process memory space

A Method of Implementation may decide to duplicate this subprogram (inlining) or have one instance per process

General updates

Other updates (in progress)

Features category to be reflected (pending completion of P1)

Link with types (e.g. parameter)

Access to data

Subprograms as accessors to data component types as in v2

AADlv3
Part 3 – Dynamic semantics

Jerome Hugues

AADLv3 table of contents

Part 3: Dynamic semantics

Default semantics of component category and features

They provide high-level rules for processing an AADL model and interpret its behavior

This behavior is later refined on a per-property basis in P4

Rule of thumb: an AADL tool **must** support these rules in the case they process the AADL model for one verification objective

Status update June 2019

Document split from AADLv2.2 document

Must revisit the definition of the hybrid automata

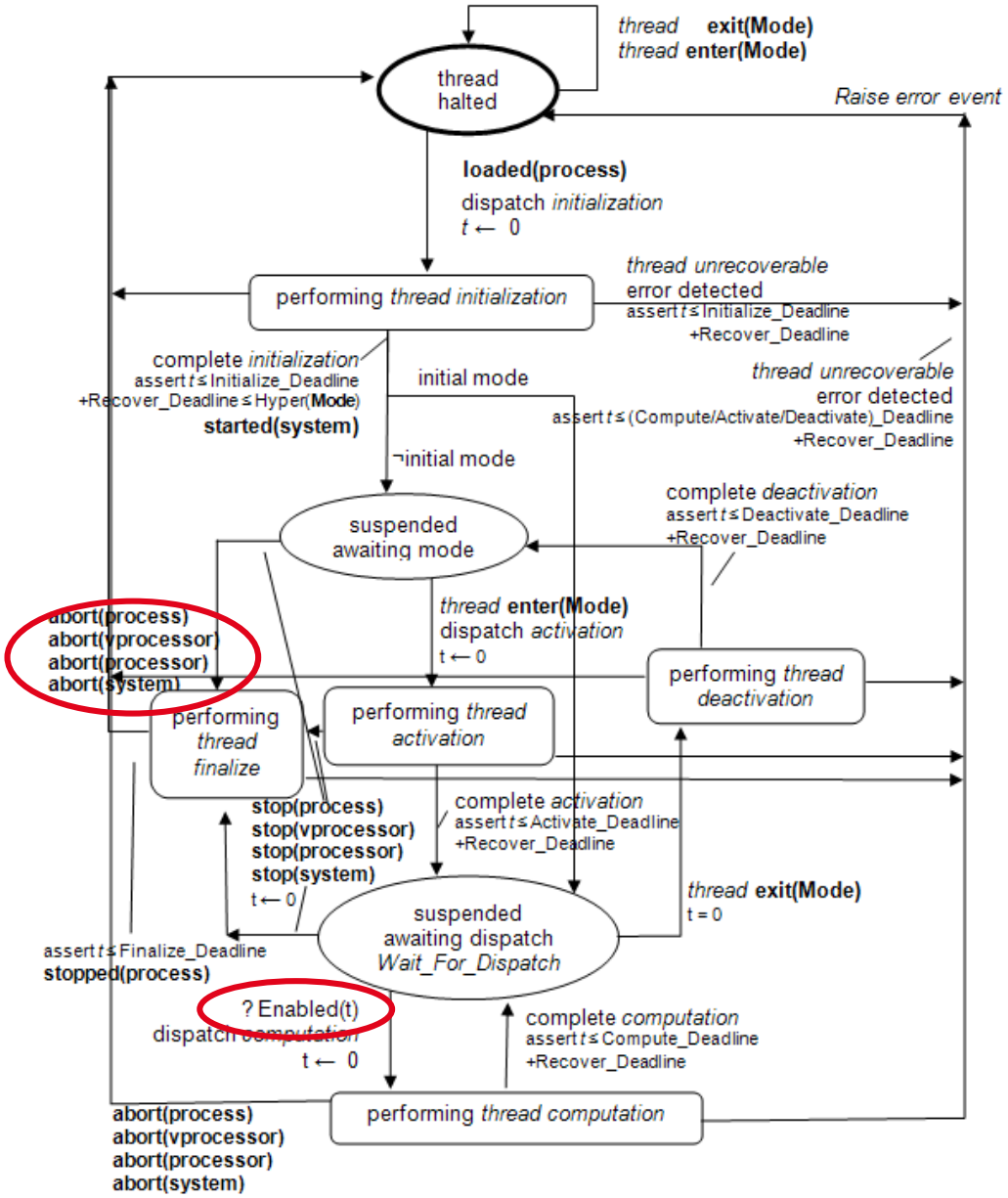
For the moment,

- informal states combined with RTS call (abort, await_dispatch, etc.)
- No clear rules for composing automata, it is implicit that automata at one level “controls” subcomponents
  - How to clarify this part?

Thread automata

Most complex one

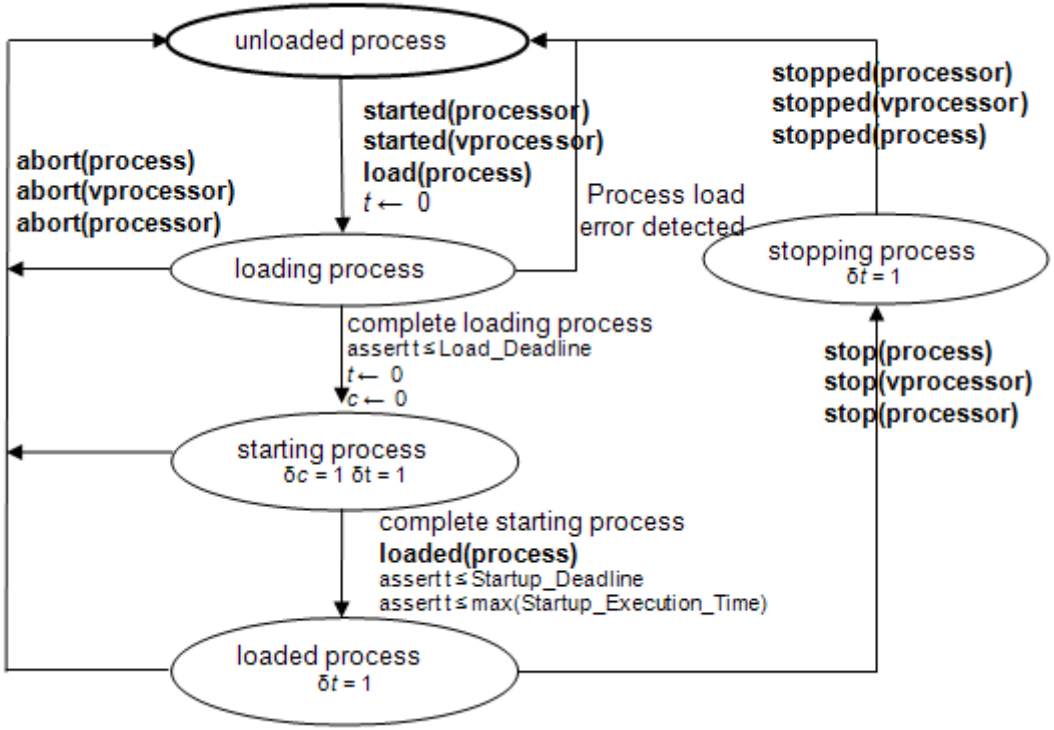
- RTS calls emerge from thread, control other automata
- *but* ?Enabled and dispatch ‘implemented’ by (virtual) processor scheduler
- State when thread blocked on resources as part of “performing thread computation” ?



Automata

Process on top of OS abstraction.

Initialization of OS vs. initialization of process (user code)

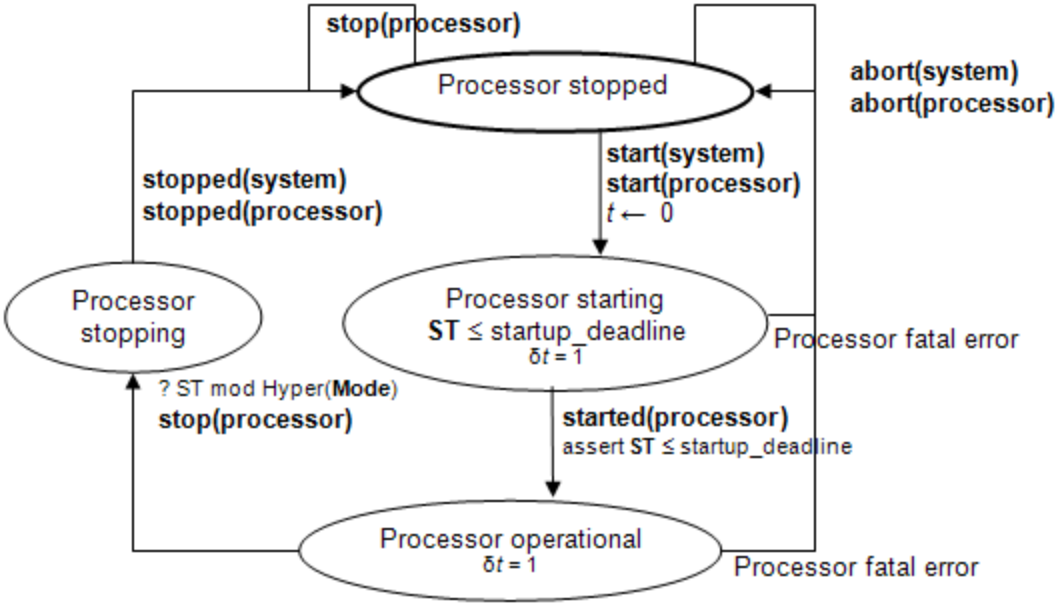


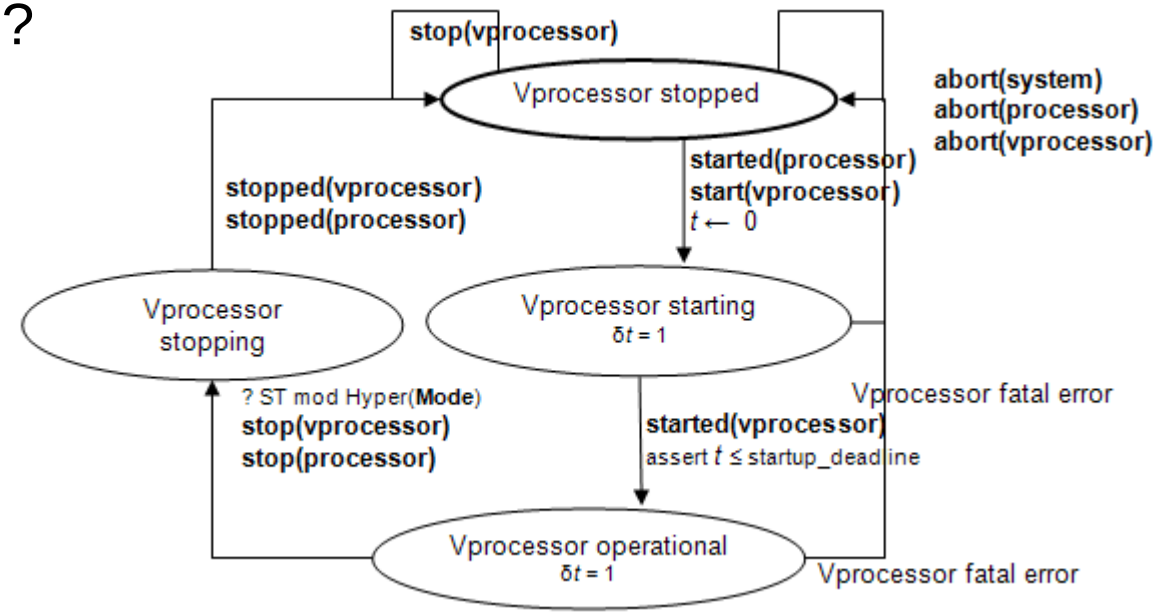
Automata

Start up of associated devices? Hidden in corresponding driver thread?

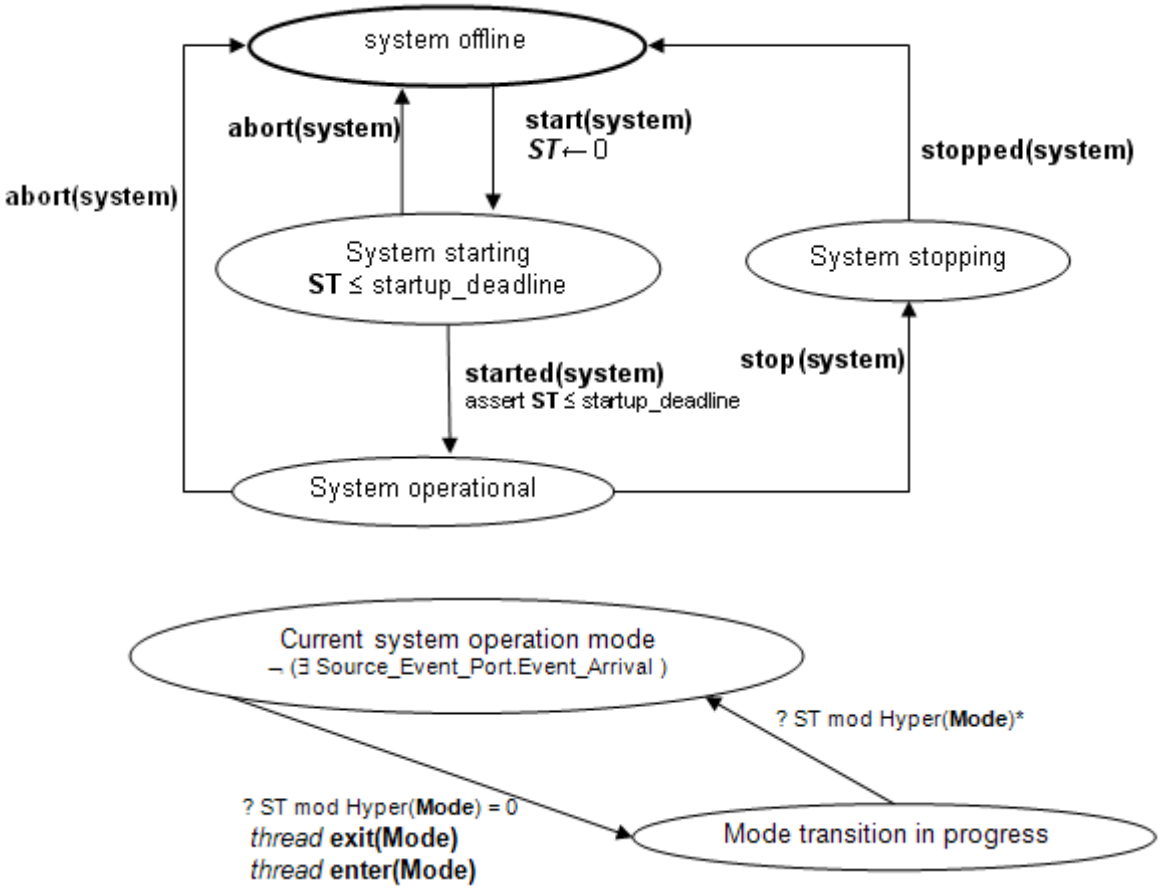
Bus initial handshaking?

How far do we want to go?





Automata



AADlv3

Part 4 – MoC

AADLv3 table of contents

Part 4 : Property sets & MoC configurations

Rationale: a MoC is a particular usage of properties e.g. Synchronous profile, mono core, multi core, etc. Goal is to document most common ones

Define relevant subsets, starting with AADLv2 property sets

Use configuration sets to compose them. Ideally, should be orthogonal