AADL V3 Standard Discussions

Peter Feiler

Feb 2019

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Content

Roadmap

Packages and General Syntax

Interface Composition

Configuration Specification

Features, Connections, Flows

Property Language

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 conversion into Restructured Text (RST) in progress
  - Document split into sections
  - Revision of packages, component interface, implementation, sucomponent, configuration
- 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

- Stereotypes to specify applicability
- Simplified property value assignment (default, final, override)

Explicit deployment binding concept

- Binding points and binding declarations
- Resource types associated with binding points

Virtual platform support

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

Flows

- (virtual) platform flows
- Flow merge points

Nested component declarations

- Define nested components without explicit classifier

Key V3 Changes

Array support revision

- Exposure of index dimensions/sizes in interface

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

AADL Packages & Components

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Packages for Property and Type Definitions

Request for property sets with nested identifiers

- Allow property definitions and type definitions in packages

- Decision: Yes

Nested Packages

Package definitions have nested name paths

- Allow syntactic nesting of package declarations
- Qualified name of package is the combination of outer package names and defining package name

Decision: Yes

- Use <dot> as separator instead of ::
- Decision: Go with ::

Imported Namespaces

Import declaration

- Make other package namespace content visible in a given package
  - All content: Import packA::*; [alias for package name]
  - Specific definition: import packB::TypeX [**as** mine];
- Declare within a package
- Reference by defining name only
  - Qualify if local definition with same name (indicator to user)
  - Qualify if multiple imported definitions with same name
  - Alias can resolve multiple imported name conflicts

Decision: Yes including alias support

- Qualified name references are not required to be in listed in import declaration

Decision: Yes (Alexey,Jerome)

Replaces **with** *clause and* ***renames*** *declarations*

Public and Private Sections in Packages

Public/private sections lead to complex rules about portions of implementation definitions residing in public and portions in private section

Proposal

- Eliminate public and private sections in packages

Proposal

- Allow classifier definitions to be marked as **private**

Decision: Yes

Recommendation: file per package (multiple nested packages ok). File name = package name.

Question: name nesting reflected in name nesting

Make AADL Case Sensitive

Identifiers: yes for all identifiers

Keywords:

- Case sensitive – ~~all upper xor~~ **all lower**; allows for identifiers with mixed case (Yes)

Decision: Yes

Section keywords in Classifiers

Proposal

- Sections in arbitrary order: yes
- Eliminate sections with keywords
  - Revisit after nested components and **connection** keyword on connections

```
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;

thread interface filter is
insignal : in port;
outsignal : out port;
filterpath: flow path insignal -> outsignal;
#Period => 20 ;
end;
```

End keyword without Matching Name

Proposal

- Eliminate matching name after **end** keyword
  - For packages
  - For classifier definitions

  Recommendation: all but Brian

```
package PackC2
    type tt;
    interface mine is
        sig : in feature tt;
    end ;

    bus interface canbus end;

end ;
```

Classifier Naming

As in AADL V2

Component interface name

- Single identifier

Component implementation name

- <component interface identifier> <dot> <impl identifier>

Configuration name

- <component interface identifier> <dot> <config identifier>

Property Association

As before but with new syntax instead of *applies to*

[ModelElementPath] #<propertyname> => <property value>;

General form used in classifier

```
Thread interface T is
   Inp: in port;
   #Period => 50 ms;
   Inp#Data_Size => 6 Bytes;
 End;
```

In context of local declaration

```
Thread interface T is
   Inp: in port { #Data_Size => 5 Bytes;};
End;

System s.impl is
   P1: process ComputeProcess.impl {
        #Code_Size => 3.5 Kbytes;
        t1#Period => 20 ms;
        t2#Period => 10ms;
        };
End;
```

Component Categories

Category

- Once specified cannot be refined into another category
  - Binding better for mapping functions to implementation architecture
  - *May be useful for providing “implemented as”*
- Usage: interface, implementation, subcomponent
- Category must match

Component interface

- <category> and **interface** keyword
- Composable interface without category
  - Usage in interface composition
    - Content consistent with target category

```
interface sub
features
    name : in feature person ;
    surname : in feature person ;
end ;
process interface subsub
features
    p1 : port date ;
    p2 : port date ;
end ;
```

Nested Subcomponent Declarations

Nested components without explicit classifier

- Single instance of an unnamed classifier
- No interface enforcement at given level
- Reach down for connection declarations

Recommendation: proceed.   Think of this as pattern that needs to be satisfied by classifiers getting configured. Can we define implementations without an explicit type but identify path in nested structure. Name mapping of features

- Optional explicit interfaces for intermediate nested component declarations
  - Interface enforcement as design constraint?

```
system ControlSystem {
    sensing: device { sensedata: out port;};
    processing: {
        filter: thread {
            inp: in port;
            outp: out port;
        };
        control: thread {
            inp: in port;
            outp: out port;
        };
        filtercontrolconn: filter.outp -> control.inp;
    };
    actuating: device { inp: in port; };
    sensefilterconn: sensing.sensedata -> processing.filter.inp;
    controlactuateconn: processing.control.outp -> actuating.inp;
}
```

Optional semi-colon

Optional semi-colon for last in list of items

- List of properties in curly brackets (, vs ; as separator)
- List of nested subcomponents
- List of declarations in classifier (end as separator/terminator)
- Proceed

```
interface sub is
    name : in feature person;
    surname : in feature person ;
end ;
interface subsub is
    p1 : in port date ;
    p2 : in port date { #Data_Size => 3; };
    p1#Data_Size => 3;
end ;
```

```
tem ControlSystem {
 sensing: device { sensedata: out port; };
 processing: {
     filter: thread {
         inp: in port;
         outp: out port;
     };
     control: thread {
         inp: in port;
         outp: out port;
     };
     filtercontrolconn: filter.outp -> control.inp;
 };
 actuating: device { inp: in port; };
 sensefilterconn: sensing.sensedata -> processing.filter.inp;
 controlactuateconn: processing.control.outp -> actuating.inp;
}
```

AADL Interface Composition

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Composition of Interfaces

Objectives

- Definition of component interfaces by
  - Feature, flow, mode declarations and property associations
  - Extension of component interfaces through additional declarations in extension
  - Definition of component interfaces from previously defined composable interfaces
- Named interfaces as connection point

Approach

- Component interface declaration with *interface* keyword and optional component category
- Allow multiple component interfaces as part of extends
- Composition rules align with current extends rules
  - Local addition of elements in extension
- Named interface instances
  - Multiple instances of same interface replaces feature group concept in V2

Interfaces and Component Categories

Component interface

- <category> and **interface** keyword
  - *has implementations*
  - *referenced in subcomponent*
  - *Can be extended*
- **Interface** keyword without category (composable interface)
  - Usage in interface composition
    - Content must be consistent with target category

```
interface sub
features
    name : in feature person ;
    surname : in feature person ;
end ;
process interface subsub
features
    p1 : port date ;
    p2 : port date ;
end ;
```

Interface Extension

Extension and categories

- Defining interface and extended interface(s) must have same category or no category
- Extended interface can be an interface without category

Addition of features, flows, properties

Local refinement of inherited features in named interfaces

- Assign type when absent (primitive type or classifier)
- Override existing type with
  - Type extension
  - Any type

```
Interface Logical
 Temperature: out data port;
 AirPressure: out data port;
End Logical;

System interface mysys extends Logical
is
 Speed: out data port;
 Temperature => TemperatureData;
End;

System interface mysys1
is
  L1: Interface Logical{
    Temperature => TemperatureData;
  };
 Speed: out data port;
End;
```

Composition of Interfaces

Inherited content from multiple interfaces

- Cannot be in conflict (same as for local definitions)

```
interface Logical
is
temperature: out data port;
Speed: out data port;
End Logical;

interface Physical
is
Network: requires bus access CANBus;
End Physical;

interface s1 extends Logical
Onemore: out event port;
End s1;

interface s2 extends Logical, Physical
End s2;

interface s3 extends Logical, Physical
is
Onemore: out event port;
End s3;
```

Right: at most one with category and others composable

V2: Locally added feature cannot conflict with a feature inherited from Logical

V3: Feature from Logical and Physical cannot be in conflict

V3: Locally added feature cannot conflict with inherited features

Composition of Directional Interfaces

Interfaces with directional features may be included as original direction or as inverse direction for component at the other end of a connection

- This is the inverse of from feature groups

```
System interface Sender extends Logical, Physical
End;

System interface Receiver extends Physical, reverse Logical
End;
```

Composition of Named Interfaces

Objective: Handle multiple instance of same interface, e.g., voter taking input from multiple instances of same subsystem

- Individual features qualified by interface instance name
- Internally: interfaceinstancename . Featurename
- Externally: subcomponentname . interfaceinstancename . Featurename
- Connections between named interfaces

```
System interface sif1
    IFlog: interface Logical;
    IFphys: interface Physical;
End;
System interface voter
Source1: interface reverse Logical;
Source2: interface reverse Logical;
End;

System Top.impl is
Sub1: system sif1;
Sub2: system sif1;
Voter: system voter;

Conn1: connection Sub1.IFlog <-> Voter.Source1 ;
Conn2: connection Sub2.IFlog.temperature -> Voter.Source2.temperature ;
End;
```

Directionality of arrow on named interface: Bi-directional arrow for interface connection. Connections between directional features must be directional. Directional connection on bi-directional interface: no.

Connections between named interfaces (V2 feature group connections) or between features in an interface (reach down V2.2)

Use of Named Interfaces

Example of mapping output to ports in different named interfaces

```
Device sensor is
temperature: out data port;
Speed: out data port;
End;

System sys2
is
  L1: interface Logical;
  L2: interface Logical;
F1: flow L1.outp -> L2.inp;
End sys2;

System sys2.i1 is
  sub1: device sensor;
  conn1: sub1.temperature -> L1.temperature;
  conn2: sub1.temperature -> L2.temperature;
End;

System sys2.i2 is
 sub1: device sensor;
 sub2: device sensor;

  conn1: sub1.temperature -> L1.temperature;
  conn2: sub2.temperature -> L2.temperature;
End;
```

How to refer to flow inside Logical?
L1.p1#DataSize =>

sub1 output is mapped into a port in two different interfaces. These may be ports with the same name, or ports with different names.

Output from different sources to different interfaces. L1.temperature and L2.temperature receive different output.

Nested Interfaces

Works for composition of named interface instances

- Nested name scopes
- Effectively we have nested feature groups
- Deprecate feature groups in V3

```
Interface composite is
  L1: interface Logical1;
  PF: interface Physical;
End;

System interface Top is
  FG: interface composite;
  L2: interface Logical2;
End;
```

All features in single namespace

Unnamed interfaces share a name space (no nested name space)

```
Interface composite extends Logical1, Physical
End composite ;

System interface Top extends composite, Logical2
End top;
```

Name conflict between Logical1 and Logical2 feature temperature

Subcomponent Refers to Interface

Substitution of any component that is an extension of interface

- Only in implementation extensions (not in configurations)
- Allow multiple interfaces on right hand side (unnamed composite interface)

- Rules about connected port (port_connection property)

```
System interface Sensor extends Logical, Physical
End;
System interface Actuator extends reverse Logical, Physical
End;

System Actuator.impl
End;


System top.i is
  sub1: system Logical;
  sub2: system reverse Logical;
  conn1: sub1.temperature -> sub2.temperature;
End;

System top2.i extends top.i
is
  sub1 => Sensor;
  sub2 => Actuator.impl;
<connections to additional features>
End;
```

Assign a component classifier that supports the interface plus more

Composition of Interface Property values

Interface property values are inherited by the component

```
Thread Interface Logical is
temperature: out data port;
Speed: out data port;
#Period=> 10ms;
Speed#Rate => 5 mpd;
End;

Interface Physical is
Network: requires bus access CANBus;
#Period => 10ms; -- should this property be there?
End;

System s2 extends Logical, Physical
End;


System s3 extends Logical, Physical is
#Period=> 20ms;
End;
```

Component level property value

Feature level property value

One inherited assignment only: Yes

Multiple inherited assignments of same value: No

Subject to default, final, override rules

Composition of Interface Property Values - 2

Named interface composition

- Component level property values apply to component, not the named interface name space

```
Interface Logical is
temperature: out data port;
Speed: out data port;
#Myname => "peter";
End;

Interface Physical is
Network: requires bus access CANBus;
Properties
#Hisname => "peter";
End;

System s2 is
 L1: Interface Logical;
 P1: Interface Physical;
L1#DataSize => 30 Bytes;
End s2;
```

Component level property value

Myname and Hisname are s2 properties, not L1 and P1 properties.

Composition of Flows

Same rules as V2 extends
Flows in interfaces are only with respect to its features
The composite component may add flow specification for flows between features in different interfaces

```
Interface Logical
temperature: out data port;
Speed: out data port;
flows
 temp: flow source temperature;
End Logical;

System s2 implements Logical, Physical
flows
 spd: flow source speed;
End s2;
```

Can add flows for inherited features as was possible in V2

Composition of Modes

Only one source (same as **extends** of single classifier)

- Local additions as in V2
  - current std allows adding states in type extensions

Annex Composition

Configuration of annex specifications into an AADL model

- See configuration discussion

Composition of annexes from different interfaces

- Same Annex notation in two interfaces
  - Not allowed
- Local addition of annex
  - Follow annex rules for annex extension

Feature Name Mapping for Connections

Support for composition of independently developed subsystems or subsystem with different nested interface hierarchies

- Inline mappings (reach down multiple interface nesting levels)

```
Conn1: sub1.lfea1.fea2 -> sub2.rfea1;
Conn2: sub1.lfea1.fea3 -> sub2.rfea2.fea11;
Conn3: sub2.rfea2.fea12 -> sub1.lfea1.fea4;
```

Needs to be repeated for each pair of subcomponent instances

- Reusable equivalence mapping

```
map1: mapping ComponentType1 == ComponentType2 as
lfea1.fea2 == rfea1;
Lfea1.fea3 == rfea2.fea11
end mapping ;

Connx: sub1 -> sub2 mapping pckx::map1;
```

Name mapping between name scope hierarchies

Direction is inferred from connection declaration and feature direction.

Is reusable mapping needed? Alternative: use name mapping in a feature mapping (up/down) as a wrapper or in an enclosing component with mapping between them.

Use as Aggregate Port

Interface elements interpreted as elements of aggregate data

```
Device sensor is
temperature: out data port;
Speed: out data port;
End;

System sys2
is
  L1: aggregate Logical;
End sys2;

System sys2.i1 is
  sub1: device sensor;
  conn1: sub1.speed -> L1.speed;
  conn2: sub1.temperature -> L1.temperature;
End;
```

Use output rates etc on aggregate.

For implementation architecture use virtual bus as an aggregator. Its binding indicates over what part of the HW flow it stays aggregated.

Do we need aggregate port specifications?

Should this be a protocol issue?

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
- Configuration assignments
  - implementation to subcomponents
  - Types or classifier to features
  - Association of collections of final 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 type extension or any type
    Decision:
- Assignment of property values

Myport => MyDataType;
Same as configuration assignment syntax

Extension without feature addition:
Difference to interface configuration?

Component Implementation Extension

- Addition of subcomponents, connections, etc.
- Revision of existing subcomponents
  - Assign implementation for specified interface
  - Override existing implementation with extension
  - Override existing implementation with alternative
  - Assign interface extensions and their implementations

V2 type match allows implementation override

V2 type extension

Eliminate signature match and need for substitution rule specification
Decision:

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 “final” property values within existing component hierarchy
- Specify bindings
- Add flow specification
- Add annex subclauses

Configuration Assignment

Configuration assignment

- Elaborate and annotate subcomponent substructure
  - Annotate substructure with “final” property values, bindings, annex subclauses
  - Assign component implementation for subcomponent with interface
    - Explicit: it becomes the intended implementation
    - Via configurations: associated implementation
    - Cannot be for interface 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;
```

Replacement of interface by implementation or configuration

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;
```

Should we allow implementation extension as part of configuration assignment in a configuration specification? It potentially adds additional subcomponents

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;
```

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;
    annex EMV2 {** … **};
    #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;
```

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

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

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**
  - Assigned value is “final”
  - May override previously assigned “default” values

```
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;
```

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 property value assignment is allowed for any assignment target
  - Property value assignments in configuration specifications are “final”

```
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;
```

Unnamed Compositions

Unnamed composition as part of a subcomponent configuration

- Same rules as for composite configuration specification (Probably yes)

```
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 act as implicit composition.

Unnamed composition as part of a subcomponent declaration

- Same rules as for composite configuration specification (probably not)

```
system top.basic is
 Sub1: process proc.i , proc.safety;
 Sub2: process proc.security , proc.safety;
 end;
```

Implicit composition (unavoidable)

- Different assigned configurations may contain configuration assignment to same target component
- Same rules as for composite configuration specification

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;
```

```
subclause Top_emv2 for top
use types ErrorLibrary;
  …
End Top_emv2;
```

Example of separately stored annex subclause

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

Inherited annex subclauses based on **extends**

- Automatically included
- Extends override rules of annex apply

Separate extensions

- No conflicting declarations

New idea: mode specific configuration specification: for property assignment.

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  (need #Deadline? Limit value to assignment to deadline)
  xsub3.T1#Period => TaskPeriod;
  xsub3.T1#Deadline => TaskDeadline;
end;
```

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;
```

- Users are able to add “missing annotations”
  - Additional flows, error model specification, property values
  - User can declare extensions of parameterized configuration that contain the annotations
  - 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

Match&replace classifier/data type within a scope

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

```
Configuration FlightSystem.secure
    extends FlightSystem.TripleRedundant
is
  GPS *=> GPS.secure;
  Dlib::dt *=> Secure.securesample;

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

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

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

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

```
Package FS
Import mine::*;
System FlightSystem.TripleRedundant
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
```

Generic 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 (Do I still need the implementaiton specific configuration pattern specification?)

```
Configuration GPSsecure.config is
  Mine::Sensor *=> Sensor.Settings;
  Dlib::dt *=> Secure.securesa
  #Period *=> 50 ms;

  Mine::GPS *=> GPS.secure { #Period => 50 ms};
end;
Configuration Sensor.Settings extends Sensor.impl is
  #Period => 50 ms;
  reading#Data_Size => 20 Bytes;
end;
```

Set Period default value within scope for any component requiring period and does not have an explicit assigned value

Assign period as part of pattern. Why not define classifier that includes the property

- Assign configuration pattern to subsystems

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

Features, Connections, Flows

Peter Feiler

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Features

- Generic feature
  - No refinement into one of the other categories
  - No specific communication semantics
  - Can be directional
- Ports
  - Discrete message communication semantics
  - Consistent I/O timing
  - Clarification of “frozen”
  - In/out direction and connection declaration
- Data access
  - Syntactic read/write declaration
  - Connection direction reflects data flow
- Bus/Virtual Bus access
  - Connection direction from provides to requires (direction of icon)
- Subprogram (group) access
  - Provides/requires
- Subprogram parameter
  - As before

Features - 2

- Named interfaces
  - Replaces feature groups
- Binding points
  - Provides/requires resource type

Ports

Directional feature

- In, out, in/out

Predictable received value

- IPO semantics (received value not affected by new arrivals)

Default send/receive timing

- Completion/dispatch
- Explicit service calls
  - Timing spec via property
  - Received value at time of call

Queuing

- Receiving port

Shared queue

- Queue serviced by multiple receivers

Move port related service function definitions to code generation annex

Event, Data, Event Data Ports

Syntactic and semantic distinctions

- Event: no type, has receive queue
- Event data: message type, has receive queue
- Data: data type, Receive queue size of 1
  - Intended for sampling by periodic receiver, can be input to aperiodic receiver
  - By default does not trigger dispatch, but can when explicitly specified in property
- Event data, data, and event ports are sampled by periodic receivers
- No distinction between sender side data port and event data port
  - They can be connected to all 3 types of ports
  - Event port can only be connected to event port
- Cannot define as 'generic port' and configure in data type and queue size

Simplified Syntax

- p1: **in port** <data type>
- Event port: no data type vs. **event**

buzden commented on Jan 28, 2018

ARINC653 Ports

Denis: From time to time we run into a problem that ports in the AADL core standard are specified very precisely and (at the same time) in a very non-practical way.
In particular, obligatory *double buffering* (when one buffer of in event data port is filled by input events and the second one is filled with Receive_Input service call) does not allow to model adequately port-like communication when no such buffering occurs (e.g. ARINC653 ports, AFDX ports, etc.). *Default timings* like sending output at completion time is also a nice concept but does not model real-world facilities well.
At the same time, some aspects of ports are not defined well, like output buffering of out event data ports with non-default queue size.
Some time ago at least some people in the committee agreed that the current specification of ports in AADLv2.2 is not acceptable for the future, in particular for AADLv3, because of overspecification. I.e. in v3 we need more flexible and less restrictive specification for ports with which we at least should be able to model adequately modern communication facilities from the desired field (like ARINC653-ports, probably lower-level ports too).
So, what we suggest is at least to track in the v3 roadmap a special bullet regarding to ports specification harmonization.

No description about buffering.
Only description about IPO semantics
(received value not affected by new arrivals)

Data Access and Data Components

Data access

- Provides (read->in | write->out | inout) data access <data type>
- Requires (read | write | readwrite) data access <data type>
- Configuring data type
  - Optional data type: configuration assignment => useful to have **data** keyword
  - Declared data type:
    - substitution by any type (individual, configuration pattern)
    - Substitution by type extension

Data access connection between data access features only

- Data component to be declared as instance of data interface
  - Data interface as extension of data type
- Alternative: V2 access connection to data component. Yes

Other Access Features

Other access features

- Bus, virtual bus, subprogram, subprogram group
- Bus, virtual bus: in, out

Need for syntactic distinction? Yes. Optional classifier

- All have provides access and requires access
- Is classifier sufficient as distinction?
  - *Provides access <bus_classifier>*
  - Specify access feature without type/classifier but category (yes)

Access connection between access features only

- Component classifier must have access feature
  - Every interface must have explicit provides access feature
  - Built-in access feature
- Alternative: Access connection to component (as in V1/V2). Yes

Named Interfaces

Declaration as named feature in interface

```
System interface sif1
    IFlog: interface Logical;
    IFphys: interface Physical;
End;
System interface voter
Source1: interface reverse Logical;
Source2: interface reverse Logical;
End;
```

- **Reverse** direction
  - Configuration assignment of interface with reverse direction
    - Source1 => **reverse** MyLogical; yes
    - Require explicit interface classifier declaration as **reverse**
    - **reverse** from original declaration
- Allow unnamed interface composition (multiple interfaces) in named interface feature declaration?
  - FullIF: **interface** Logical, Physical; No. As for subcomponent.

Connections

Connections between subcomponents

- Directional: Source -> target
  - information flow (out -> in, provides read -> requires read, Requires write -> provides write)
  - Subprogram Access control flow (requires -> provides)

```
system conntop.i is
    sense: abstract sensor.i;
    processing: process control.impl;
    actuate: abstract actuator.i;
    hw : system hardwareplatform.impl;
    sensetocontrol: connection sense.outp -> processing.insignal;
    controltoactuate: connection processing.outaction -> actuate.inp;
end;
```

Keyword **connection** instead of keywords for types of connections
Connect is a verb: no

- Non-directional
  - between abstract features without direction
  - Conn1: **connection** comp1.fea1 <-> comp2.fea1;
- Bi-directional
  - Between in/out ports, read/write access
  - <-> vs. two separate directional connections
- Named interfaces
  - <-> implies direction inferred from interface element direction
  - -> implies all interface elements same direction (no)

Feature Delegation

Feature delegation down the component hierarchy

- Map feature of enclosing component to feature of subcomponent
  - Maps connection targets to lower level targets
  - Does not connect between components

```
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 ;
```

Separate **delegate** keyword
⇒ Use connection direction

Reach down of Connections

Reach down into nested named interfaces

- Connecting ports within an interface

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

process interface controlProcess is
controlIF: interface control;
end;
system interface conntop end;
system conntop.i is
    sense: abstract sensor.i;
    processing: process controlProcess.impl;
    actuate: abstract actuator.i;
    hw : system hardwareplatform.impl;
    sensetocontrol: connection sense.outp -> processing.controlIF.insignal;
    controltoactuate: connection processing.controlIF.outaction -> actuate.inp;
end;
```

- Mapping of named interface elements

```
process controlProcess.impl is
    dofilter: thread filter;
    docompute: thread compute;
    extin: mapping controlIF.insignal => dofilter.insignal;
    ftoc: connection dofilter.outsignal -> docompute.insignal;
    extout: mapping controlIF.outaction => docompute.outsignal ;
end;
```

Reach down Into Component Hierarchy

- In nested component without intermediate subcomponent features
- Consistent with mappings
  - For nested components
  - For subcomponents with implementations

```
system ControlSystem.i
is
    sensing : device {
        sensedata : out port ;
    } ;
    processing : process {
        inp: in port;
        filter : thread {
            inp : in port ;
            outp : out port ;
        } ;
        control : thread {
            inp : in port ;
            outp : out port ;
        } ;
        filtercontrolconn : connection filter.outp -> control.inp ;
        outp: out port;
        outmap: mapping outp => control.outp;
    } ;
    actuating : device {
        inp :  in port ;
    } ;
    sensefilterconn : connection sensing.sensedata -> processing.filter.inp ;
    controlactuateconn : connection processing.outp -> actuating.inp;
    reachdowncontrolactuateconn : connection processing.control.outp -> actuating.inp;
end ;
```

Feature, Connections and Modes

V2.2 Issue #24

- Connection is only active if both endpoints are active: no need to explicitly specify in modes for connection (already in V2.2)
- Connection not active even though endpoints are active: need in modes on connection (already in V2.2). Needed? Yes
- Mode specific visibility of features
  - V2.2: active component
  - V2.2: requires connection property
  - V2.2: property indicating input actively received (mode specific)
  - V3 discussion: dispatch trigger port specific active input port list

Flow Specifications and Sequences

Flow specification (same as V2)

- 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

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)
- 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)

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

Property Profile

Constraints between properties in profile
Literal value specific sub-profile

Definition of property profile

- List of property references that are part of a profile
- Other profiles can be listed in a profile
- Same property reference can be in multiple profiles

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

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

Usage

- Classifier specific property profile
- Profile assignment to classifier
  - Multiple configuration assignments
  - Unnamed profile
- Analysis specific property profile

```
device GPS
 use properties GPSProperties;
End GPS;
```

```
MyPackage::GPS => properties
#SecurityLevel, #SafetyLevel;
```

Property Profiles for Model Elements

- Identification of model element "type"
  - By key word
  - By Meta model element name
  - By enumeration type for core and each annex
    - Union of enumeration subtypes
- Granularity of model elements
  - Component categories
  - Feature categories
  - Association categories
  - Flow specifications
- Usage
  - Property definition
  - Profile assignment

```
property Period : applies to Thread;
```

```
Thread => properties #Period, #Deadline;
```

Property Association

- Property reference always with #

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

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

```
process interface subsub
features
    p1 : port date ;
    p2 : port date { #Size => 3; };
properties
    p1#Size => 3;
end ;
```

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 (.)
  - One value multiple modes?

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

Property value assignment in design space

- Assignment in interface or implementation
- Value override
  - in interface extension
  - Implementation, implementation extension

Property value assignment in configuration

- Assign only if not previously assigned
- At most once via configuration

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