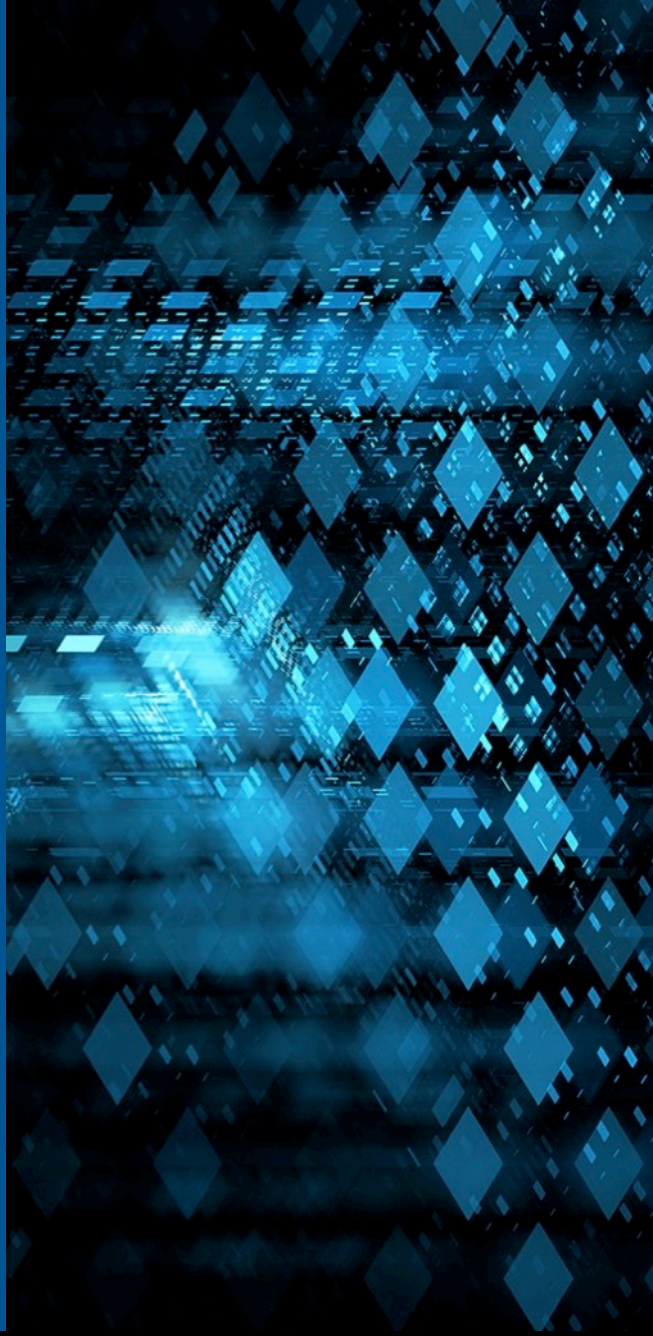
Features, Connections, Flows

Peter Feiler

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.

DM19-0166

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**

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)