Candidate AADL Flow Enhancements

Presented at
SAE AADL Standards Committee Meeting
5-7 June 2017
Atlanta, GA

Presented by
steve.vestal@adventiumlabs.com
Adventium Labs
Minneapolis, MN

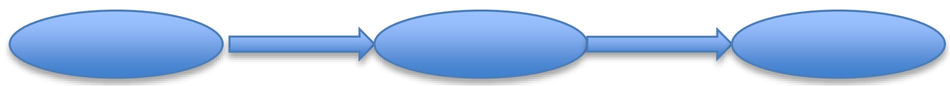
Distribution statement A. Approved for public release; distribution is unlimited.

Contents

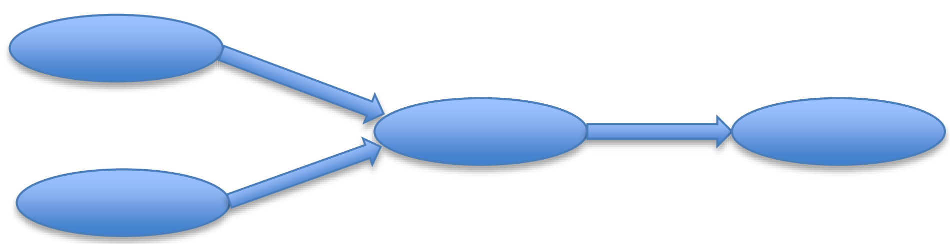
- Multi-input, multi-output flows
- Semantic models for flows
- Sequence diagrams $\leftrightarrow$ flow declarations
- Other comments

SISO Versus MIMO Flows

AADL 2.2 supports Single Input Single Output (SISO) flows



Flows through a system from boundary inputs to outputs are in general graphs (not necessarily acyclic). A tractable and desirable next step is Multiple Input and Multiple Output flow declarations.



Adventium LABS

Join/Fork Behaviors and Assertions

Examples of user-specified behaviors and assertions at joins/forks:

- Requires inputs from all incoming sub-flows
  - More generally, specified i1:i2:i3… ratio (with input queueing)
  - For event-driven joins, a declaration of behavior (e.g. AND vs OR)
  - For sampling at join inputs, an assertion to be verified
- Assert bounds on arrival skew between different inputs
- Assert arrival order constraints, e.g. data inputs arrive before a control input arrives
- Departure choice logic (possibly Behavior Annex features?)

MIMO Flow Declaration Ideas

- Idea 1: Additional rules and semantics for sets of linear end-to-end flows that declare paths through a tree.
  - Can be done in a non-standard way by a tool vendor near-term.
  - Must satisfy the property that they collectively form paths through a tree. Prefixes or postfixes will be duplicate sub-flows.

```
src1_to_sink: end to end flow
    dataSrc1.dataSrc -> inC1 -> dataFuse.dataPath1 -> outC -> dataSnk.dataSnk;
src2_to_sink: end to end flow
    dataSrc2.dataSrc -> inC2 -> dataFuse.dataPath2 -> outC -> dataSnk.dataSnk
        { FlowTreeProperty::FlowTree => (reference (e2eF1));};
```

Non-standard property to identify a set of linear flows that collectively form a tree flow. The set of flow declarations must be structured to form a set of paths through a tree from leaves to a common root.

MIMO Flow Declaration Ideas

Idea 2: Declare MIMO flow paths in subcomponent types.

- More clear, more concise, standard.
- Join/fork semantics declared by the subcomponent supplier without requiring collaboration with the system integrator, who declares the end-to-end flows.

```
process FuseData
features
     dataIn1: in data port;
     dataIn2: in data port;
     dataOut: out data port;
flows
     joinPath: flow path dataIn1, dataIn2 -> dataOut;
end FuseData;
```

Syntax change to allow multiple inputs or outputs in a flow path.

```
src1_to_join: end to end flow dataSrc1.dataSrc -> inC1 -> dataFuse.joinPath;
src2_to_join: end to end flow dataSrc2.dataSrc -> inC2 -> dataFuse.joinPath;
join_to_sink: join flow dataFuse.joinPath -> outC -> dataSnk.dataSnk;
```

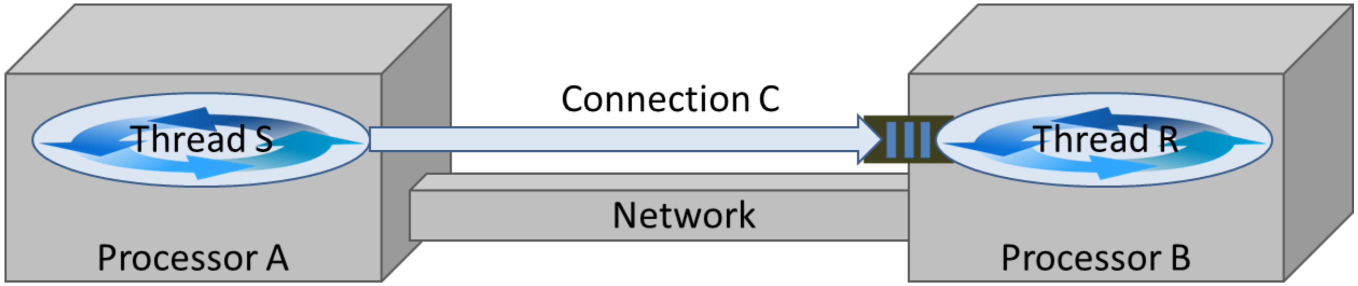
Need some added syntax and semantics to declare trees rather than sets of independent linear flows.

These ideas are starting points for several variations and combinations.

Flow Semantic Models

- Flow declarations are static model elements, e.g. text strings.
- A flow declares a run-time behavior where there are multiple run-time executions.
- Semantic models presented here are based on
  - Timed automata
  - Causal partial orders

A Simple Flow Declaration

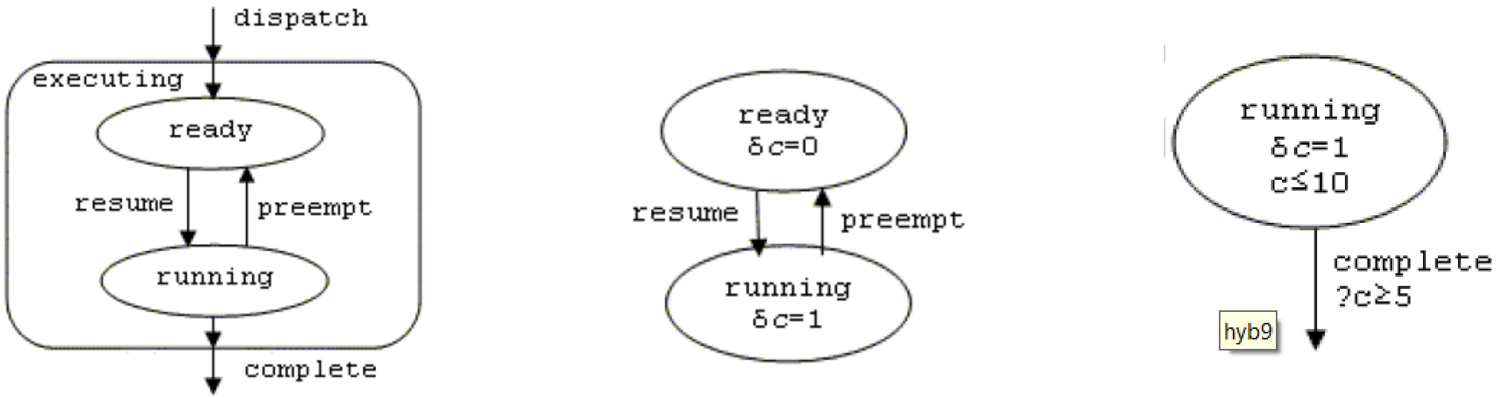



```
simple_flow: end to end flow S -> C -> R;
```

- How many executions of the flow occur when S and R have different periods?

- What set of events are part of each execution in the presence of
  - over/under sampling
  - mix of sampling and various queuing behaviors
  - traversing a mix of synchronous and asynchronous domains
  - MIMO flows

Timed Automata



From “SAE AS5506C Architecture Analysis & Design Language”

The AADL standard specifies thread semantics using a “concurrent hierarchical hybrid automata” model.

- Uses some linear hybrid automata features (e.g. stopwatches)
- Uses some hierarchical and rendezvous features (e.g. name scoping rules)

What follows is based on timed automata with some notational liberties (e.g. conjecture it is reducible to UPPAAL)

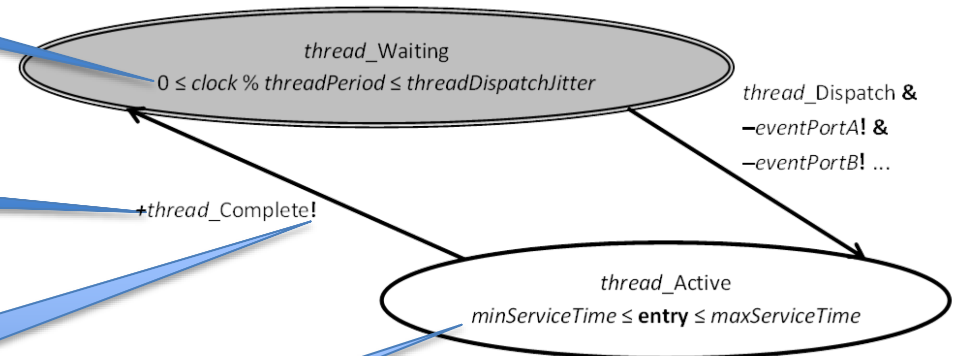
Notational Conveniences for AADL Behaviors

Notation for periodic and sporadic events

Distinguish sending from receiving transitions (+ vs -)

Distinguish wait-for-rendezvous from rendezvous-if-ready (? vs !)

Local latency bounds declared to get a simple, pure TA model (scheduling and contention not modeled)



A[illegible]mata pattern for AADL threads with Scheduling_Protocol => Periodic	
[illegible]	Fully qualified AADL name of the thread subcomponent
clock	Fully qualified name of the Reference_Time => subcomponent of the execution platform subcomponent to which the thread is bound
threadPeriod	Period => value for the thread
threadDispatchJitter	Clock_Jitter => value for the execution platform subcomponent to which the thread is bound
eventPortX ...	List of fully qualified names of in event ports for that thread specified as non-blocking read rendezvous events. If the Dequeue_Protocol => AllItems, then all_*eventPortA* labels are used.
minServiceTime	Lower bound on response time obtained from a schedulability analysis
maxServiceTime	Upper bound on response time obtained from a schedulability analysis

Timed Automata Pattern for Periodic Thread

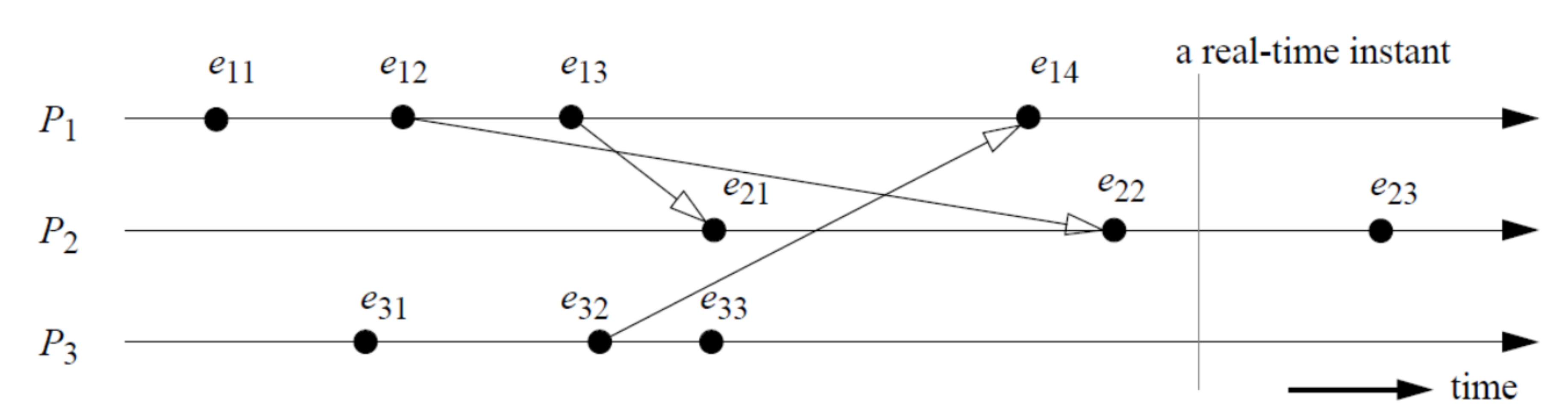
Timed Automata Pros and Cons

\+ Integrates real-time and complex discrete state behaviors

\- Analysis is computationally challenging

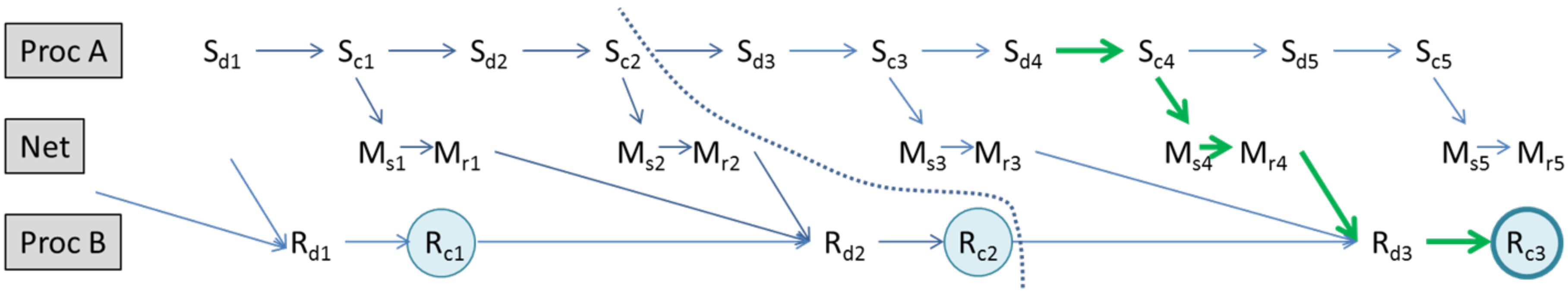
\- Event sequence generators/recognizers are challenging (e.g. constructing a recognizer TA for the negation of a TA and a deterministic TA from a nondeterministic TA are uncomputable; see Tripakis, "Folk Theorems on the Determinization and Minimization of Timed Automata")

Causal Partial Orders



Reinhard Schwarz and Friedmann Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grain," Report SFB 124-15/92, December 1992.

Proximal Cause Flow Execution Semantics



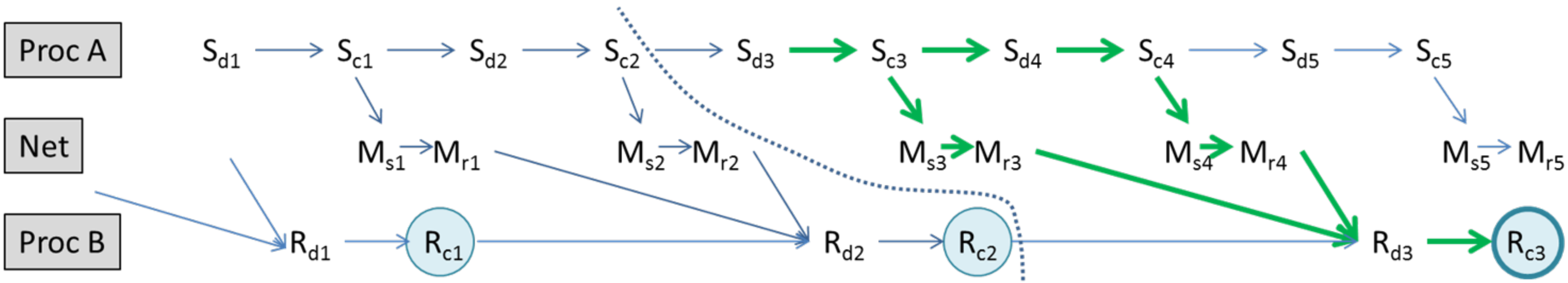
A flow execution ending in R_{c3} when Task R samples its most recent input.

- Definition: there is one flow execution for each <u>final event</u> in the executions of the <u>final component</u> in the flow declaration.
  - Example: There is one flow execution for each completion event of the final receiving thread, circled in the above illustration.

- The set of events in a flow execution ending at event E_i is a <u>subset</u> of the events that precede it in the partial causal order <u>AND</u> are not elements of any previous execution of that flow (what we call the proximal causal events for E_i).

- The set of events in a flow execution is filtered to reflect specific semantics, e.g. events may be removed due to under-sampling at an input port or loss at a bounded input queuing port. When in doubt, include the event ("causal" means "might cause").

With Input Queueing Behavior…



A flow execution ending in R_{c3} when Task R reads all queued inputs.

Partial Order Pros and Cons

+ Clear answer to, “What event sets are flow executions?”

+ Efficient partial order data structures and algorithms

– No integrated real-time semantics

– Generators/recognizers are still challenging due to timing, non-determinism, concurrency effects.

Flow Sequences

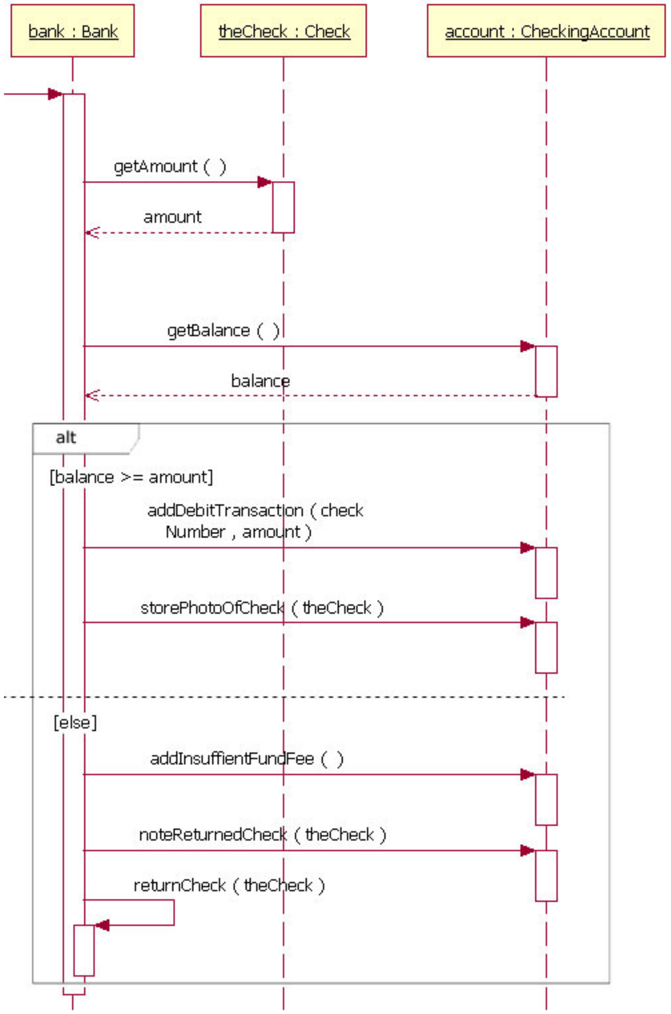
Sequence diagrams (UML, SysML) are widely-used to specify flows through components.

They may be a source of useful ideas.

Alignment with a widely-used notation may facilitate training and adoption.

Complete alignment seems unlikely due to complex sequence diagram features, e.g.

- Conditionals and loops
- Anonymous sources/sinks
- Complex cycles and self-messaging
- Diagram vs structure distinction



Other Comments

- "10.2(4) A flow implementation may refer to subcomponents without identifying a flow specification in the subcomponent. In this case, (a) an unnamed flow specification is assumed to exist from the destination port of the preceding connection to the source port of the succeeding connection if no flow specification is declared for the subcomponent, (b) it represents a separate flow for each of the flow specifications of the subcomponent."
  - Where the subcomponent appears as the first or last element in the flow declaration, part (a) will not be defined because there is no preceding or following connection in the flow declaration.
  - Part (b) at multiple points along an end-to-end flow declaration specifies a potentially combinatorically large number of individual end-to-end flows.

- Avoid structural and behavioral properties and declarations that are redundant and can conflict with other declarations when tools are unlikely to verify internal model consistency. For example, avoid adding properties that simplify analysis just because it is complicated to derive the behavior against which assertions are checked from "core" declarations – the tool may say OK when the core declarations do not satisfy the assertion. Redundant declarations that are not automatically checked for consistency only increase modeling effort, reduce readability, and introduce inconsistency.

- What does it mean for an AADL IDE to comply with the standard? All legality rules are checked. The standard may need to distinguish between tractably and non-tractably verifiable rules.

- Discussion topic: Is it useful to distinguish between properties that specify system structure and behavior versus properties that declare assertions to be verified?
  - Assertions are verified using data that can be automatically determined by analysis of "core" structural and behavioral declarations – they can be "model-checked" in a broad sense.
  - An assertion can be removed without changing the specified structure or behavior.
  - Constraints cannot be verified to hold by analyzing the model. They are properties that must be satisfied by design choices that are not captured in the model. They are a form of "core" structural and behavioral declaration.
  - Thought questions: Is a flow Latency an assertion or a constraint? Does this depend on the model use case and available tools?