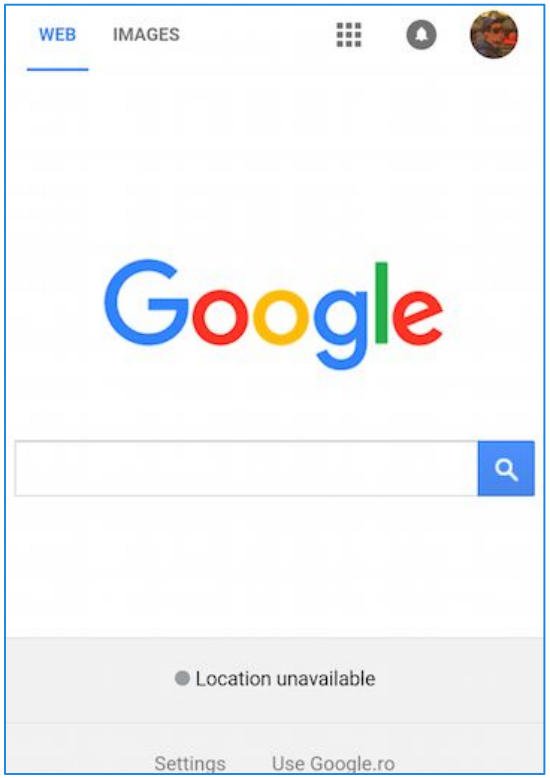
Architectural Modeling and Real-Time Systems

CSCE 742 - Lecture 20 and 21 - 11/20 and 27/2018

Our Society Depends on Software

This is software:



So is this:

Also, this:

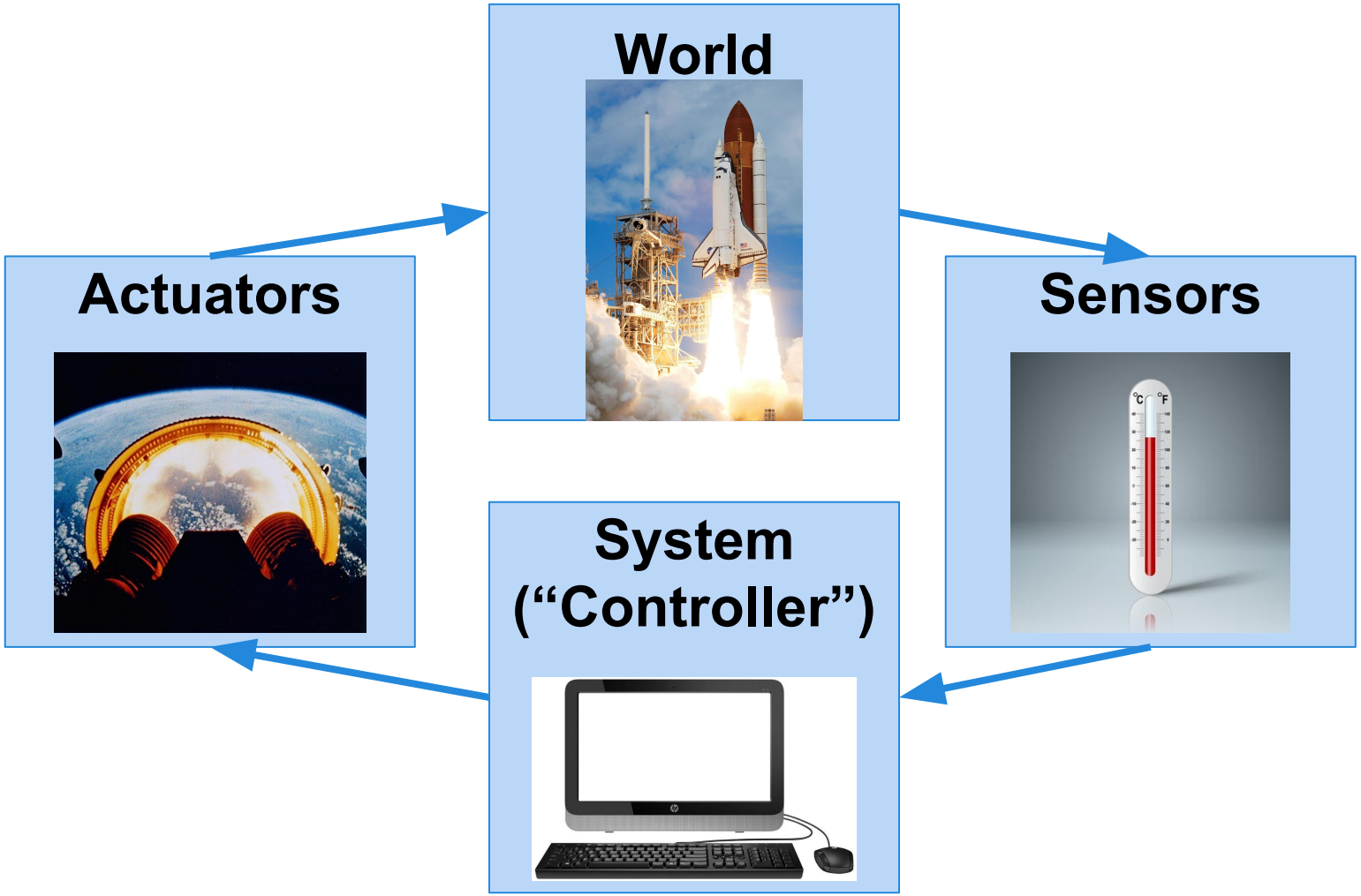
Real-Time Architectures

- A system is **real-time** if “correctness” depends not only on the logical answer, but the time it was performed.
  - **Hard or immediate real-time:** The completion of an operation after the deadline is useless.
    - Can cause a critical failure.
  - **Soft real-time:** Some lateness is tolerated, but may cause decreased service quality.
    - (ex: omitting frames in a video)

Traditional Domains of Concern

Recent Domains of Concern

Structure of Real-Time Systems



Key Concepts

- Repeatedly sample inputs, assess system state, and generate output.
- Sampling Rate: Rate at which the cycle takes place.
  - Goal is often to approximate *continuous assessment*
  - Sampling rate must be “fast enough”
    - (for good approximation)
  - Rate depends on speed of computations.

Key Concepts

- Perform bounded amount of work per cycle.
  - Need to process the next sample!
    - Need to finish within sample time.
- Longest time to process is called **worst-case execution time** (WCET).
- End-to-end latency is also important.
  - System -> Actuator -> Environment -> Sensor -> System
  - Controller must have an accurate view of the system
  - Actuator must react “fast enough” to influence the environment in time.

Concept: Tasks

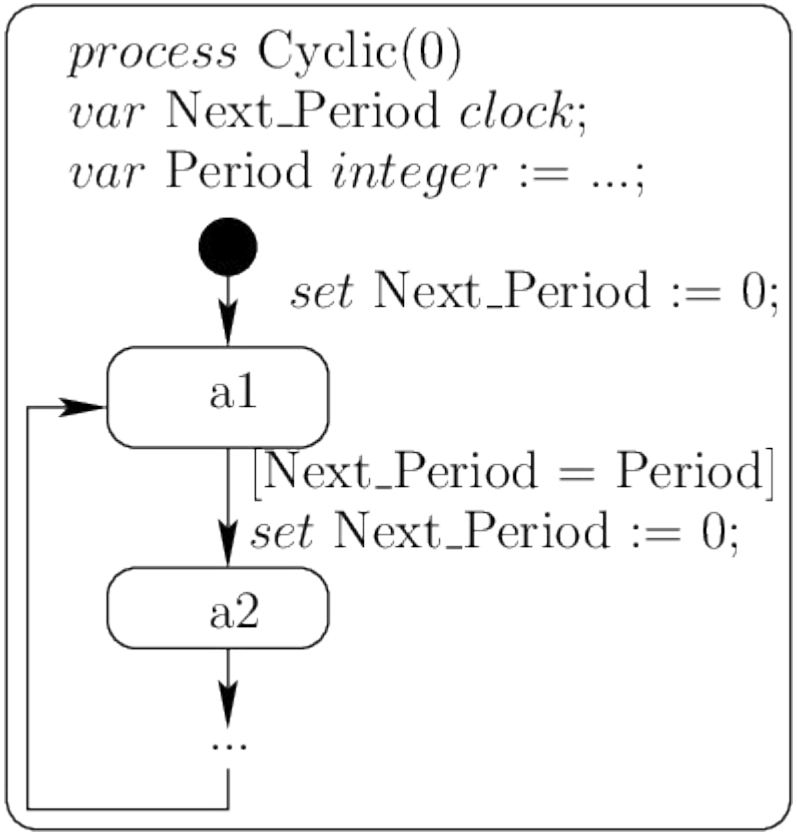
- Tasks are separately scheduled units of work
  - AKA: Threads!
- Real-time systems are split into independent, concurrent tasks.
- A **process** describes a set of **threads** that share an address space.
  - Each thread is a “task”.
  - May run at different sampling rates
  - (measured in cycles-per-second, Hz)
  - Threads need to be scheduled to run on CPUs.
  - May need to communicate dependencies.

Concept: Scheduling

- Algorithms create a guaranteed schedule or abort (no schedule can guarantee schedulability of system).
  - RMA: Rate-Monotonic (high throughput)
  - DMA: Deadline-Monotonic (finish critical first)
- Take as input:
  - Per task: period, WCET
  - Per processor: worst-case thread context switch, process context switch
  - Additional constraints and frame dependencies.
    - Frame dependencies = ordering constraints for tasks in a schedulable period (frame).

Scheduling

- Most tasks are polling (cyclic).
- May also have event-based tasks.
  - Called aperiodic tasks.
  - Require minimum delay between events to schedule.



End-to-End Latency

- Data flows through controller from inputs to outputs (back to environment).
  - May go through several intermediate tasks.
- **End-to-End Latency** describes the amount of time required for end-to-end flow.
  - Time to impact the environment and notice the change in the sensor readings.
- Frame dependencies change end-to-end latency.
  - **Mid-frame** communications have less latency.
  - **Phase-delayed** communications have more.

Process Communication

- Several mechanisms for communication between processes/threads.
- Logical views:
  - Message queue / pipe
  - Rendezvous
  - Remote procedure call
- Physical views:
  - Shared memory
  - Interrupt
  - Bus
  - Network

Polling or Event-Based?

- Do you ask sensors for a reading, or let sensors send you readings?
  - Polling… or event-based?
- Current system uses polling for most tasks.
  - Polling processes allow simpler scheduling.
  - Maintains periodicity of threads to be scheduled.
- However, polling can be very inefficient.
  - Monitoring user interface elements (key presses).
  - Polling rate must be high (no missed state changes).
  - Also want high rate to reduce latency.
  - But most of the time, it does nothing.
- But…events can cause missed deadlines.

Architecture Challenges

- Scheduling tasks is extremely hard.
  - Some tasks need to run more often than others, but have worse WCET.
  - Task A may need to run more often than task B, but can’t preempt task B.
- Do you poll sensors or wait for events?
- Algorithms must be fast!
- Hardware can be noisy or fail.
- Environmental input can fail to arrive as expected (too fast or too slow).

How do we design an architecture for this?

Architectural Modeling

Many Errors Stem from Architecture Issues

- Global variable used in different functions:
  - Issues: inconsistent values, concurrent accesses
  - Cause: Architecture Design (use of encapsulation)
- Use of COTS elements without validation:
  - Impact: do not fit with the environment, crash
  - Cause: No Validation of Components Integration
- Timing issues
  - Impact: deadlines not enforced, bad values
  - Cause: poor integration policy, lack of analysis
- These errors **could** be detected during design, but are detected instead during integration - incurring major cost!

Architecture Description Languages

- Language for viewing and analyzing “architectural” software concerns.
- Describes structure of system rather than (functional) implementation.
- Subject of much research in late 1990s and early 2000s.
  - Many academic ADLs
  - …and of course, UML
  - Wide range of ADLs for different kinds of software
  - Today: AADL (Architectural Analysis & Design Language) is one of the most common.

Architecture Description Languages

- ADLs ≈ Multi-model architecture notations
  - Overview paper: "A classification and comparison framework for Architecture Description Languages" by Medvidovic and Taylor
  - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.151.4061&rep=rep1&type=pdf
- We have seen one (UML) in some depth already in the viewpoints.
  - Doesn't have (agreed upon) semantics, though.
  - Difficult to use for analysis and architecture generation - more for human consumption.

Why use ADLs?

- Help us **understand** the architecture.
  - Designed for **analysis** and **reasoning** to draw conclusions about the performance of the model.
  - **Designed to ensure consistency between different system views.**
  - **Often designed to generate system skeleton.**
- Medium of communication.
  - Reduce the amount of information the reader needs to understand, and should structure the information.
- Help organize processes, teams, and deliverables.
- ADs can be built directly from ADLs.

Two Architecture World Views

- Bottom up (assembly):
  - Architecture description is assembled from viewpoints that describe different facets of architecture.
- Top down (generative):
  - Architecture specification (in ADL) is used to generate different viewpoints (via extraction) to present information of interest to stakeholders.
  - Can also generate implementation skeletons.

Language Focus and Applications

ADL	ACME	Aesop	C2	Darwin	MetaH
Focus	Architectural interchange, predominantly at the structural level	Specification of architectures in specific styles	Architectures of highly-distributed, evolvable, and dynamic systems	Architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings	Architectures in the guidance, navigation, and control (GN&C) domain

Rapide	SADL	UniCon	Weaves	Wright
Modeling and simulation of the dynamic behavior described by an architecture	Formal refinement of architectures across levels of detail	Glue code generation for interconnecting existing components using common interaction protocols	Data-flow architectures, characterized by high-volume of data and real-time requirements on its processing	Modeling and analysis (specifically, deadlock analysis) of the dynamic behavior of concurrent systems

ADL Features

- An ADL must provide the means to model:
  - Elements and their interfaces.
    - Interfaces are essential for demonstrating underlying semantics of the model.
  - Connectors.
  - Architectural configurations/topologies.
- ADLs often have tool support for:
  - Showing individual views.
  - Performing automated verification of properties or other analyses.
  - Performing model refinement.
  - Implementation generation.

Support for Modeling Elements

- **Interface:**
  - A element's interface is a set of interaction points between it and the external world.
  - Specifies services and constraints on their usage.
- **Types:**
  - Element types are abstractions that encapsulate functionality into reusable blocks.
  - Elements can be instantiated multiple times.
- **Semantics:**
  - High-level model of an element's behavior.
  - Needed to perform analysis, enforce constraints, ensure consistent mapping between abstractions.

Support for Modeling Elements

- **Constraints:**
  - A property of or assertion about a system or one of its parts, where a violation will reduce value.
  - Constraints needed to verify adherence to uses, enforce boundaries, and establish dependencies.
- **Evolution:**
  - The modification of (a subset of) a element's properties, e.g., interface, behavior, or implementation.
- **Non-Functional Properties:**
  - Properties that affect safety, security, performance, portability.

Support for Modeling Connectors

- Interface
  - A set of interaction points between the connector and the attached elements and other connectors.
  - Enable connectivity of elements and their interaction
- Types
  - Abstractions of element communication, coordination, and mediation decisions.
  - Makes coordination protocols reusable within and across architectures.
- Semantics
  - High-level model of a connector's behavior.
  - Enables element interaction analysis.

Support for Modeling Connectors

- Constraints
  - Ensure adherence to intended interaction protocols, establish intra-connector dependencies, and enforce usage boundaries.
  - I.e., limit on number of elements that use connector.
- Evolution
  - The modification of (a subset of) its properties, e.g., interface, semantics, or constraints on the two.
- Non-Functional Properties
  - Properties that represent (additional) requirements for correct connector implementation.

Support for Modeling Configurations

- Connected graphs of elements and connectors that describe structure.
  - Needed to determine whether appropriate elements are connected, that their interfaces match, that connectors enable communication, and that combined semantics are correct.
- Modeling enables assessment of concurrent/distributed behavior.
  - Deadlock/starvation, performance, reliability, security
- Enable analysis of architectures for adherence to design rules.
  - Too many direct communication links harm evolution

Support for Modeling Configurations

- Understandable Specifications
  - Model structural (topological) information with simple and understandable syntax. System structure should be clear from the configuration.
- Compositionality
  - A mechanism that allows architectures to describe software systems at different levels of detail.
  - Can show complex information in detail, or abstract it into a subelement that is modeled elsewhere.
- Refinement and Traceability
  - Enable correct and consistent refinement of architectures into executable systems and traceability of changes across levels of refinement.

Support for Modeling Configurations

- Heterogeneity
  - Facilitate development of large-scale systems.
    - Pre-existing elementsand connectors of varying granularity
    - Different formal modeling languages and programming languages
    - Varying operating system requirements
    - Different communication protocols
- Scalability
  - Provide developers with abstractions needed to cope with the issues of complexity and size.
- Evolvability
  - Incremental addition, removal, replacement, and reconnection in a configuration.

Support for Modeling Configurations

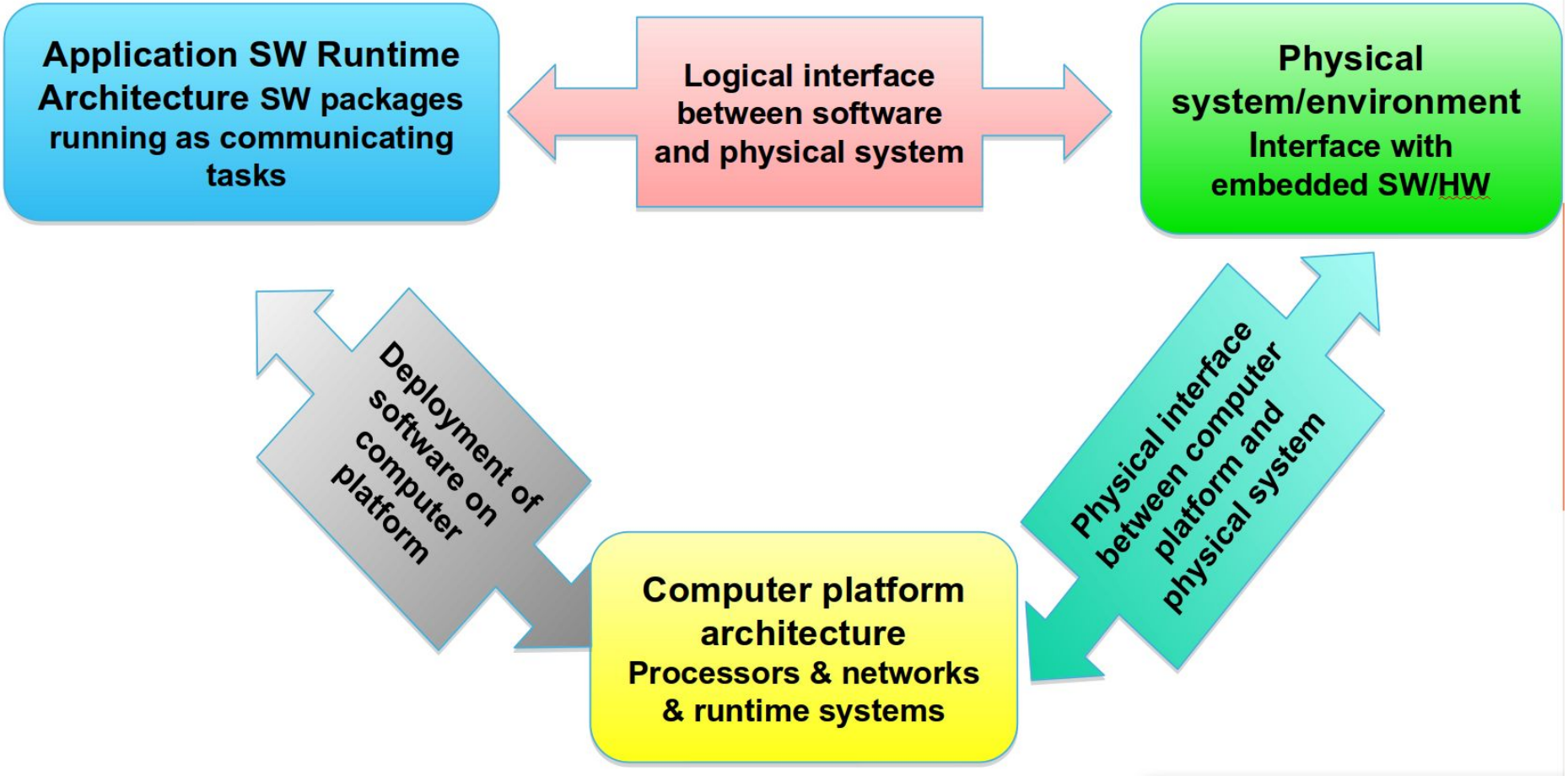
- Dynamism
  - Modifying the architecture and enacting those modifications while the system is executing.
- Constraints
  - Depict dependencies in a configuration complement those specific to individual elements and connectors.
- Non-Functional Properties
  - Used to select appropriate elements and connectors, perform analysis, enforce constraints, map architectural building blocks to processors, and aid in project management.

Architecture Analysis and Description Language (AADL)

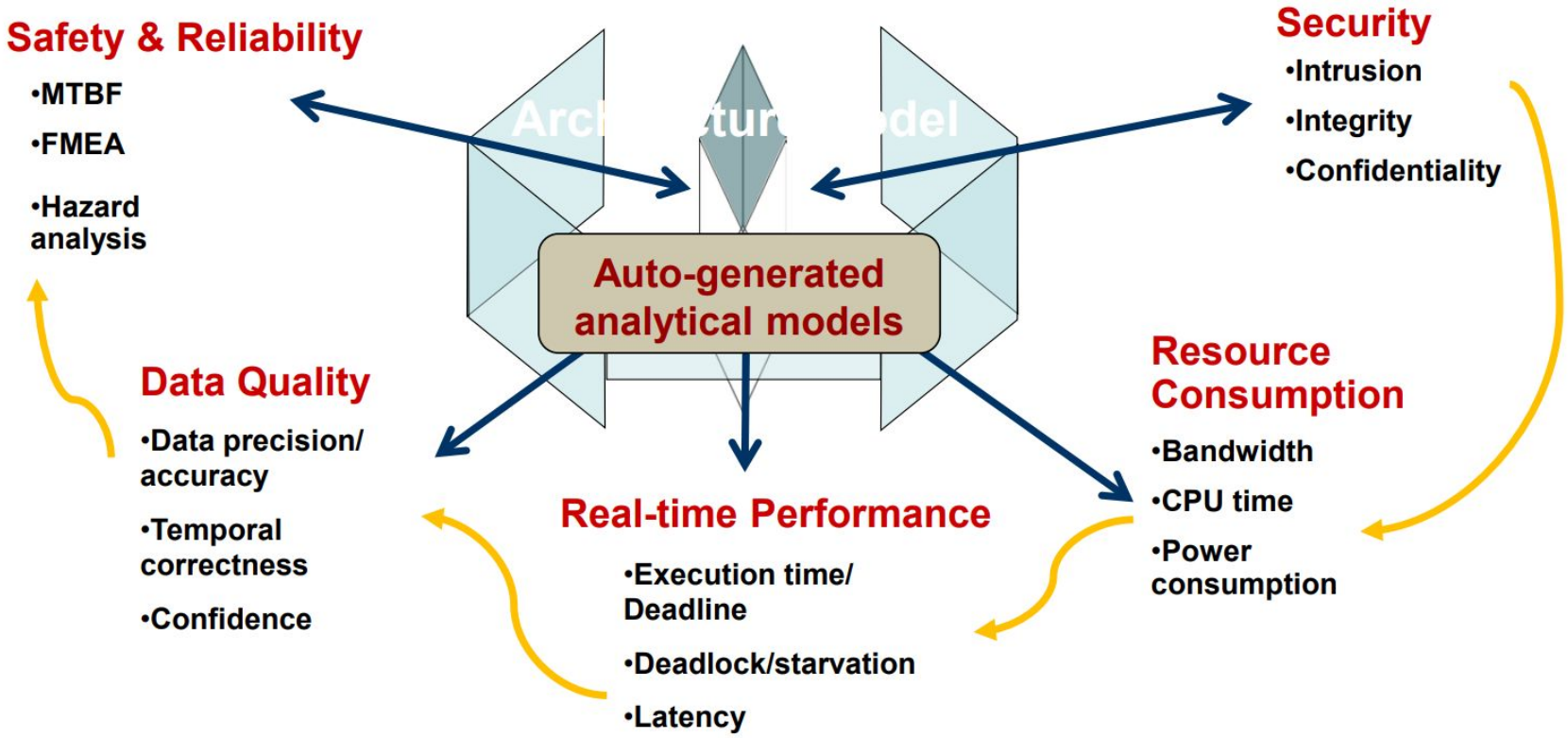
What is AADL?

- Architecture modeling language, developed for the embedded system communities.
- Uses component-based notation for the specification of task and communication architectures of real-time systems.
- Offers tool-based analysis in Eclipse framework (OSATE).

Embedded Architecture



Analysis Support

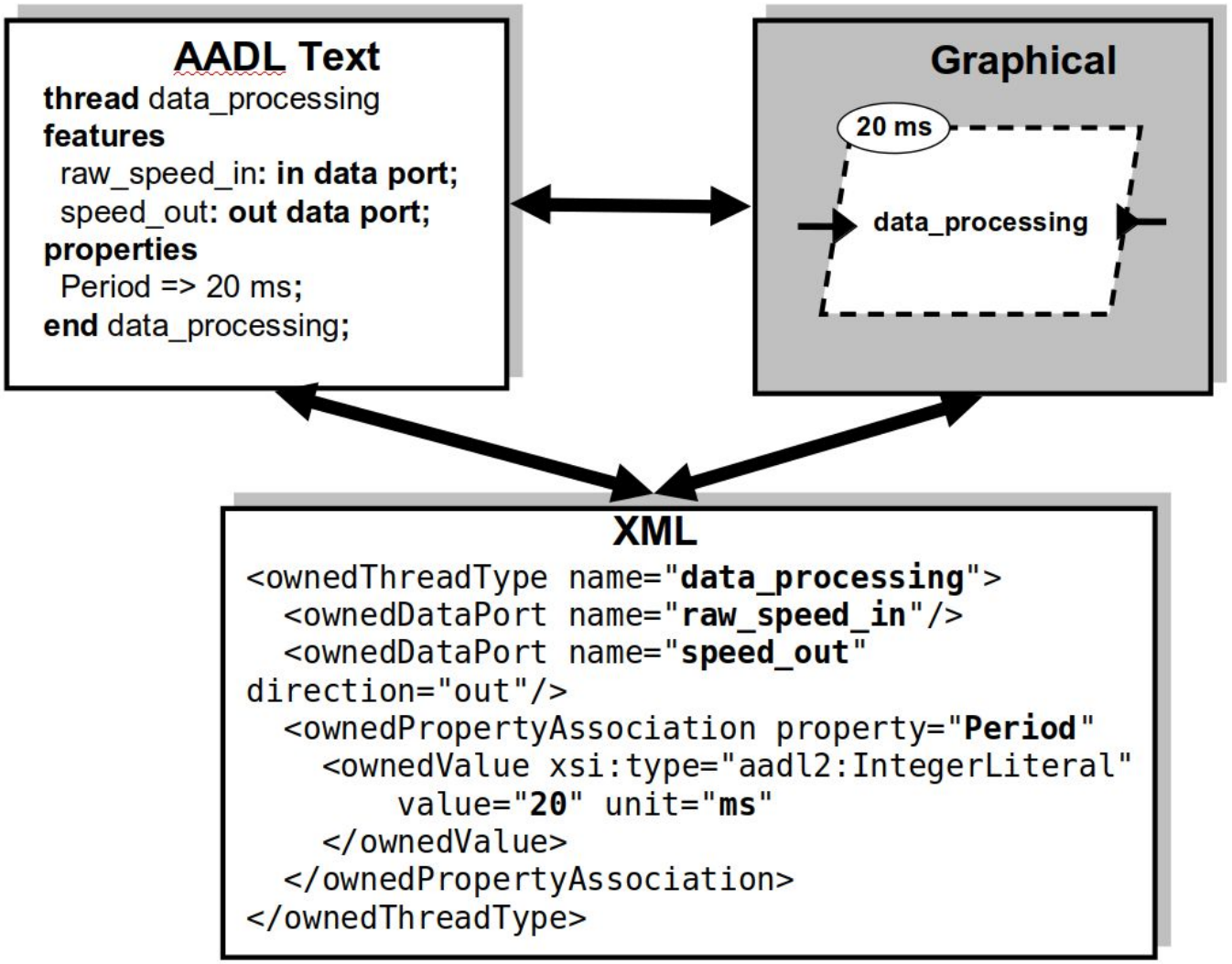


- Reduced model validation cost due to single source model.
- AADL offers an estensible domain model with strong semantics and an XML-based interchange format.

The AADL Language

- Precise execution semantics for components
  - Thread, process, data, subprogram, system, processor, memory, bus, device, virtual processor, virtual bus.
- Continuous control/event processing
  - Data and event flow, synchronous call/return, shared access.
  - End-to-End flow specifications.
- Operational modes/fault tolerant configs.
  - Modes & mode transition
- Modeling of large-scale systems
  - Component variants, layered system modeling, packaging, abstract, prototype, parameterized templates, arrays of components and connection patterns.

AADL Representation Forms



Component-Based Representation

- Specifies a well-formed interface.
  - **Component** type allow for multiple implementations with extensions.
  - All external interaction points defined as features.
  - Data and event **flows** through component, across multiple components.
  - **Properties** to specify component characteristics.
- Components organized into a hierarchy.
  - Component interaction declarations must follow system hierarchy.

Basic System Properties

- AADL defines standard properties for systems, including:
  - System startup:
    - `Startup_Deadline => 0.5s`
      - A property of type `Time`, assigned (`=>`) a value of 0.5 seconds.
    - Value is a floating point number with a time unit.
    - Time units include ps, ns, ms, s, m, h, d.
  - Time to load all programs and data into the system:
    - `Load_Time => 0.1s..0.15s`
      - Two values indicate a time interval (between 0.1 to 0.25 sec)
    - `Load_Deadline => 0.2s`

Example System: Car

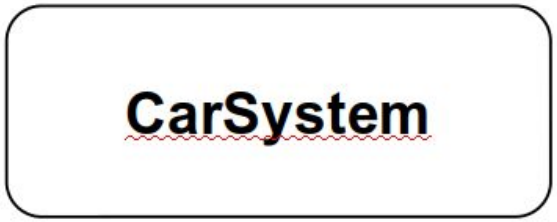
Textual AADL

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

All AADL classifiers are organized in packages

Declarations end with a semi-colon

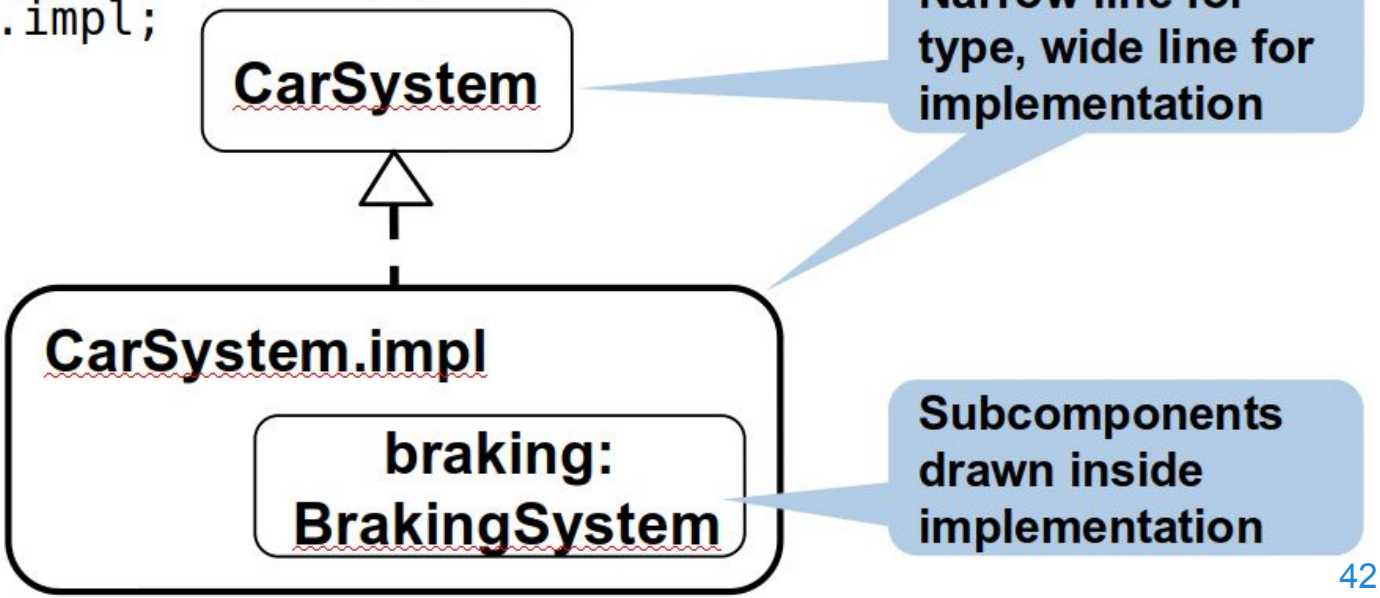
Graphical AADL



Note: Each declaration must be contained in a package

Example: Car with Braking Subsystem

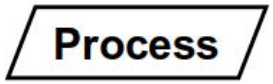
```
package carPackage
  public
    system CarSystem
    end CarSystem;
    system BrakingSystem
    end BrakingSystem;
    system implementation CarSystem.impl
      subcomponents
        braking: system BrakingSystem;
    end CarSystem.impl;
end carPackage;
```



Architecture Software Components

System – hierarchical organization of components

Process – protected address space



Thread group – logical organization of threads

Thread – a schedulable unit of concurrent execution

Data – potentially sharable data

Subprogram – callable unit of sequential code

Process Components

- Processes represent protected virtual address spaces.
  - Address space boundaries are enforced at run-time.
  - A property setting allows to disable the protection.
- Contains executable and data needed for execution and must be loaded into memory.
  - Process is stored in ROM
  - Process is loaded at system startup
  - Process may be unloaded when it is not active
- A process must contain at least one thread subcomponent to be executable.

Process Properties

- Process at run-time
  - `Runtime_Protection => false;`
    - No runtime enforcement of space protection.
  - `Load_Time => 150ms..300ms;`
    - Time to load image into memory.
  - `Load_Deadline => 500ms;`
  - `Startup_Execution_Time => 100ms..110ms`
    - Time to start process after loading (to create threads).
  - `Startup_Deadline => 200ms;`
- Relationship to implementation in a programming language
  - `Source_Language => “C”;`
  - `Source_Text => “navigation.c”;`

Thread Components

- Represents a schedulable and executable entity in a system.
  - Concurrent and Active tasks.
- Threads execute based on time or thread-external events.
  - Periodically every 50ms, e.g., a data sampling thread in a control system.
  - Process a message upon arrival with arbitrary arrival pattern, e.g., a thread in a camera processing image data when the shutter button is pressed.

Thread Components

- Threads are mapped onto operating system threads for execution.
  - One or more application threads per OS thread.
- Interacts with other threads through port connections, subprogram calls, and shared data access.
- Executes within the virtual address space of its enclosing process.

Thread Dispatch Protocols

Periodic thread (50ms)
- Periodic dispatch of threads, typically with hard deadlines

Aperiodic thread (A)
- Dispatch based on events with arbitrary arrival patterns

Sporadic thread (S:10ms)
- Dispatch based on events with a minimal time between dispatches

Background thread (B)
- Dispatch once and execute until completion

Timed thread (T:10ms)
- Dispatch based on events, or based on timeout if no events occur

Hybrid (H:10ms)
- Dispatch based on events and periodically (combines periodic and aperiodic dispatch protocols)

Thread Properties

- Properties related to thread dispatch
  - `Dispatch_Protocol => periodic;`
    - Any from previous slide.
  - `Period => 50ms;`
    - Required for periodic, sporadic, timed, hybrid threads.
- Properties needed for thread scheduling
  - `Compute_Execution_Time => 45ms..50ms;`
    - Execution time range of thread (upper is worst-case).
    - Optional, defaults to period.
  - `Deadline => 40ms;`
    - For periodic threads: indicate delayed dispatch relative to other periodic threads.
  - `Dispatch_Offset => 5ms;`

Thread Example: GPS Data Filtering

```
thread NavDataFilter
  features
    rawData: in data port navData.raw;
    filteredData: out data port navData.filtered;
  properties
    Dispatch_Protocol => periodic;
    Period => 50ms;

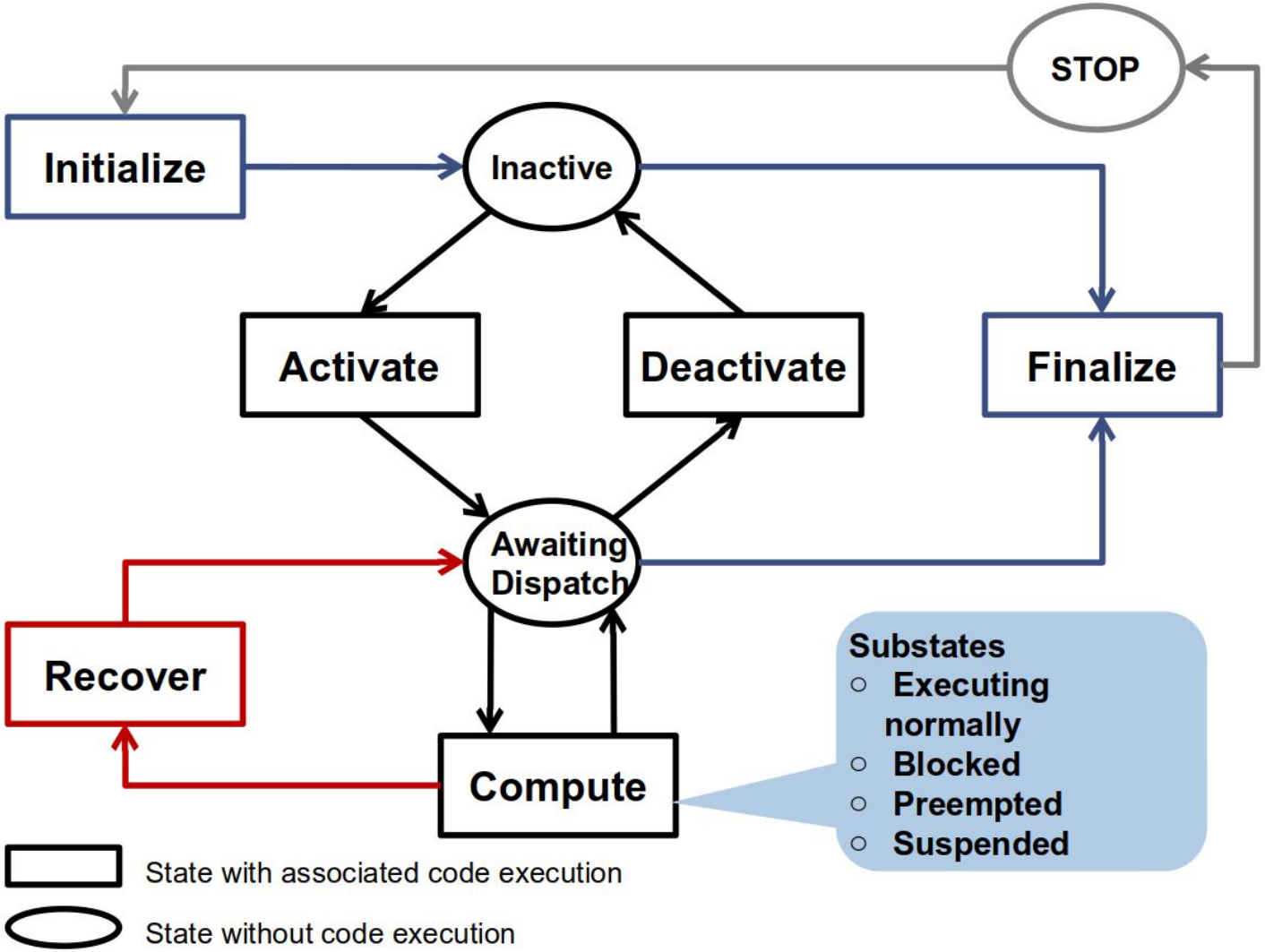
    Deadline => Period;
    Compute_Execution_Time => 18..20 ms;

end NavDataFilter;
```

Execute once every 50ms

Deadline and execution time needed by a scheduler

Thread States



Ports and Connectors

- Ports: interaction points
  - Model transfer of data and control.
  - Ports are declared as features in component types.
  - Data port: non-queued data.
  - Event port: queued signals.
  - Event data port: queued messages.
- Feature group: aggregates ports into a single connection point.
- Connections: connect ports in the direction of data/control flow.
  - uni- or bi-directional.

Port Properties

- Queuing of events and messages
  - `Required_Connection => true;`
    - Default: no connection needed.
  - `Queue_Size => 3;`
  - `Queue_Processing_Protocol => FIFO;`
    - Handling of incoming event and message queues.
  - `Overflow_Handling_Protocol => DropOldest;`
  - `Dequeue_Protocol => AllItems;`
  - `Urgency => 255;`
    - To resolve conflicts if queues are not empty.

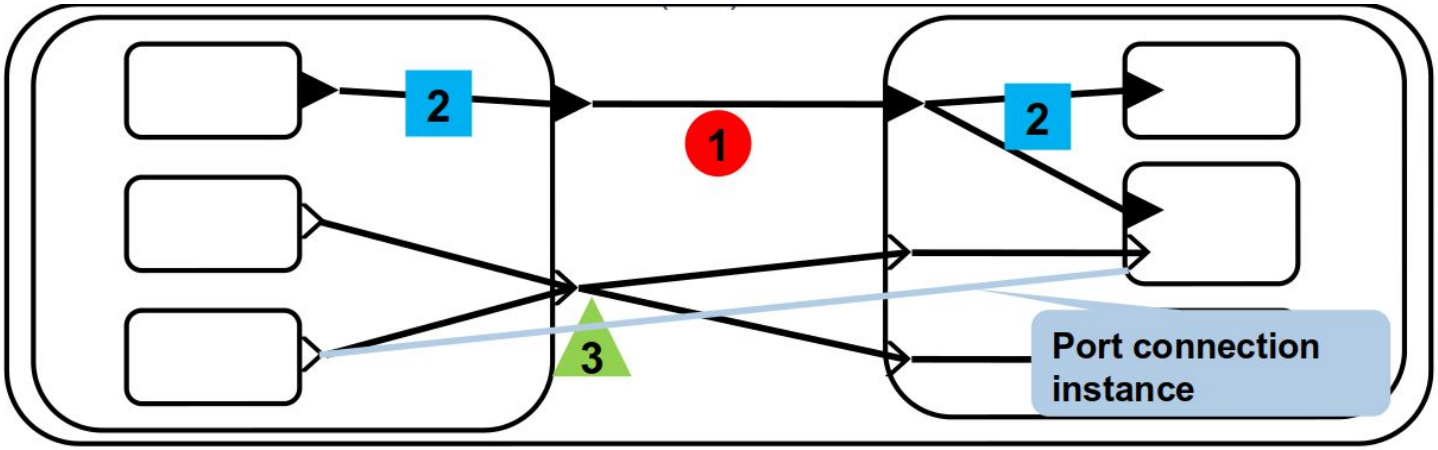
Port Properties

- Frequency of data input and output
  - ```
    Input_Rate => (Value_Range => 1.0 .. 1.0;
    Rate_Unit => PerDispatch;
    Rate_Distribution => Fixed;)
    ```
  - `Output_Rate`
- Mapping to variable in an implementation
  - `Source_Name => "brake_state";`
    ```

Connections Between Ports

An AADL port connection connects:

- Two ports of subcomponents in the same component implementation (1).
  - Communication inside a component – identical port directions.
- A port of a component implementation with a port of one of its subcomponents (2).
  - Communication with the outside – complementary port directions
- A port can have multiple outgoing connections (fan-out) (3).
- Data ports can have one, other ports can have multiple incoming connections (fan-in).
- Connections can be bi-directional (<->).



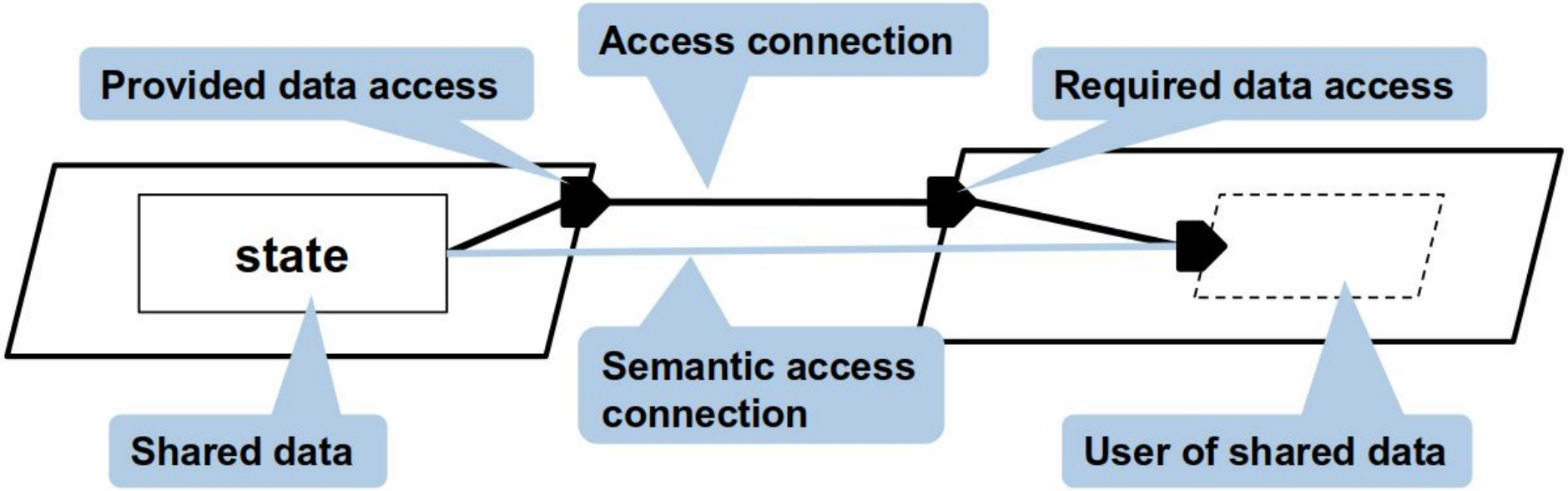
Data Components

- Data components can represent:
  - Data shared between threads or subprograms.
  - Local data in a thread or subprogram.
  - Type of data exchanged through data/event ports.
  - The type of subprogram parameters.
- AADL models should contain information about data that is relevant to analyses of the architecture:
  - Bandwidth analysis – size of data elements, frequency of data exchanges.
  - Model consistency – size, value ranges, and physical units of exchanged data.

Shared Data Access

A data component can be shared among components.

- Data access features: model required or provided access to a shared data component.
- Access connections: model access paths to the shared data component.
- Data ports must have same type, implementations of data must be identical.



Flows

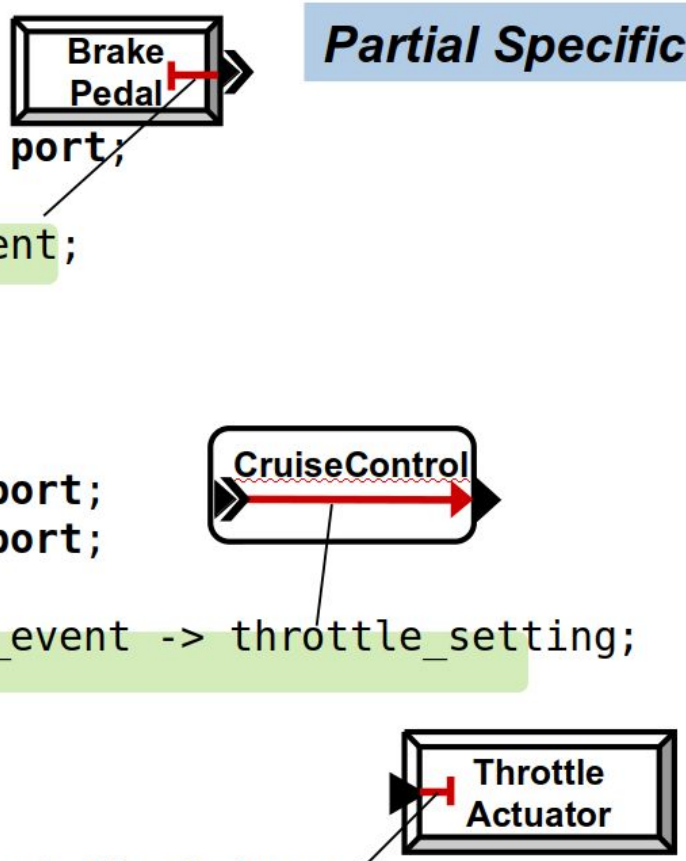
- Model logical flow of data and control through a sequence of components and connections.
  - Support analysis of data flow and control flow.
- Provide the capability of specifying end-to-end flows to support analyses:
  - End-to-end timing and latency.
  - Fault propagation.
  - Resource management based on operational flows.
  - Security based on information flows.

Flow Sources, Paths, Sinks

```
device BrakePedal
   features
      brake_event: out event data port;
   flows
      FSrc1: flow source brake_event;
end BrakePedal;

system CruiseControl
   features
      brake_event: in event data port;
      throttle_setting: out data port;
   flows
      brake_flow: flow path brake_event -> throttle_setting;
end CruiseControl;

device ThrottleActuator
   features
      throttle_setting: in data port float_type;
   flows
      FSnk1: flow sink throttle_setting;
end ThrottleActuator;
```



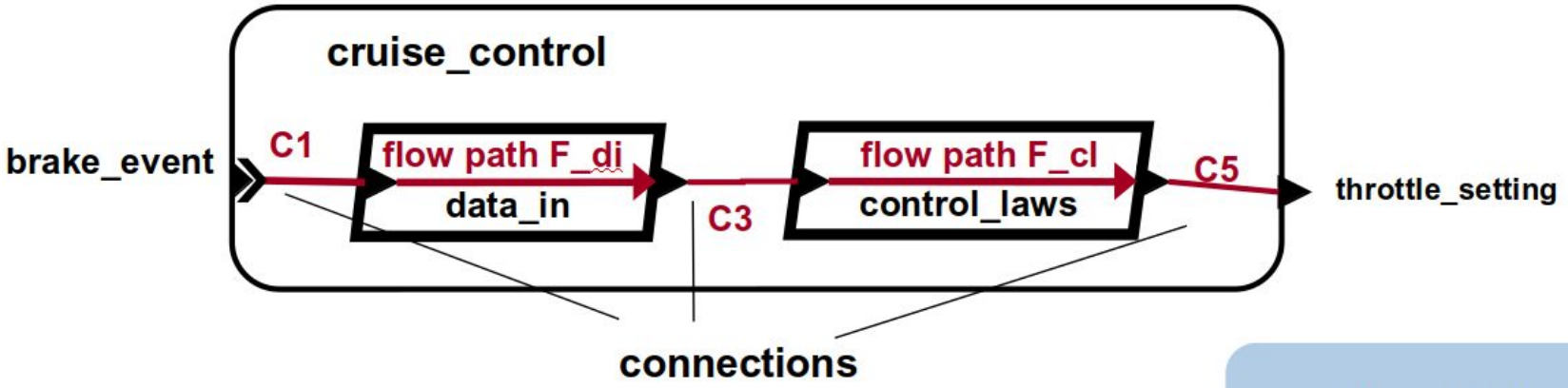
Flow Implementation

Flow through subcomponents and connections

Subcomponent flow in terms of its flow specification

Flow Path Specification

```
brake_flow: flow path brake_event -> throttle_setting;
```

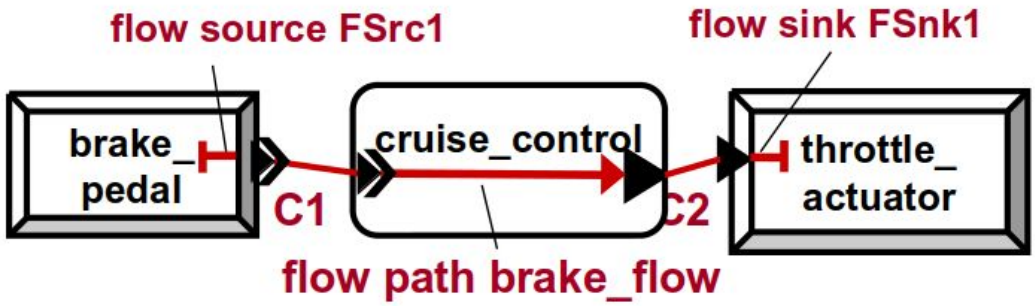


Flow Path Implementation

```
brake_flow: flow path brake_event ->
            C1 -> data_in.F_di ->
            C3 -> control_laws.F_cl ->
            C5 -> throttle_setting;
```

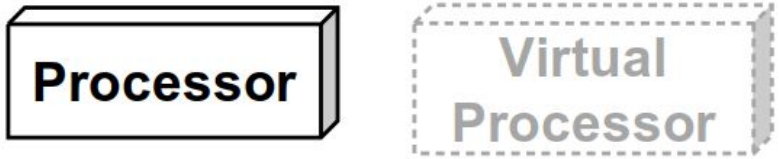
End to End Flow Example

```
system CarSystem.impl
  subcomponents
    ...
  flows
    SenseControlActuate: end to end flow
      brake_pedal.FSrc1 -> C1 -> cruise_control.brake_flow
-> C2 -> throttle_actuator.FSnk1;
end CarSystem.impl;
```



Execution Platform Components

Processor / Virtual Processor – Provides thread scheduling and execution services



Memory – provides storage for data and source code

Bus / Virtual Bus – provides physical/logical connectivity between execution platform components

Device – interface to external environment

Execution Platform Components

- Represent hardware components:
  - Processor timing, hardware clock period/jitter
  - Bus transmission time, latency
  - Memory capacity, access time, RAM/ROM
- Represent logical resources:
  - Thread scheduling policy of a processor.
  - Communication protocol over a network connection.
  - Transactional characteristics of a database modeled as a memory component.
  - Virtual bus/processor represent only logical aspects.
- These two aspects reflected in properties applied to the components.

Processor Components

- As a hardware component:
  - Processors include a CPU, memory, bus, a hardware clock that can interrupt the processor
  - Have a MIPS rating, size, weight
- As a logical resource:
  - Processors schedule threads
  - Processors execute software to provide scheduling and other runtime system functionality.
- Threads are bound to processors
- Processors may access memory and device components via buses, execute software associated with devices.

Processor Properties

- Logical Resource (Thread Scheduling)
  - `Scheduling_Protocol => RMS;`
  - `Thread_Swap_Execution_Time => 1.0ms;`
    - Cost of context switching
  - `Thread_Limit => 16;`
  - `Allowed_Dispatch_Protocol => periodic;`
  - `Source_Text => “Linux-rt.c”;`
    - File containing the runtime system.
- Hardware Component (Clock Properties)
  - Clock_Period => 10ms;
    - Time between two interrupts.
  - Clock_Jitter => 2ms;
    - Difference between interrupt handling start in multicore system.

Bus Components

- As a hardware component:
  - A bus provides the physical connection between hardware components.
  - Inside a hardware component, e.g., PCI bus in a PC.
  - Between hardware components, e.g., a USB connection between a PC and a camera.
- As a logical resource:
  - A bus represents the protocol(s) by which connected components communicate.
- Components are connected to a bus with a bus access connection.
  - A bus is shared by all components that access it.

Bus Properties

- Logical Resource
  - Constraints on transported content
    - `Allowed_Connection_Type => (Port_Connection, Data_Access_Connection);`
      - What can be transmitted over the bus?
    - `Allowed_Message_Size => 0B..1KB;`
  - Protocols and protocol properties
    - `Provided_Connection_Quality_of_Service => (OrderedDelivery);`
      - Supported protocols and QoS

Bus Properties

- Hardware Component
  - Constraints on physical connectivity
    - `Allowed_Physical_Access => (processor, memory);`
      - What may be connected to the bus.
  - Properties related to data transmission time
    - `Transmission_Time`
    - `Latency`

Memory Components

- Represent randomly accessible physical storage (e.g. RAM, ROM).
- May also be used to model complex permanent storage (e.g. disks, database).
- Processes must be in memory at startup to be executed.
  - Stored permanently in ROM, loaded into RAM.
- Processors need access to memory.
  - Processor and memory connected via a shared bus.
  - Memory is contained in the processor.

Bringing Software and Execution Platform Together

- Software relies on computational resources for execution of threads and communication among threads/between threads and devices.
- In a model, software and execution platform often form independent system hierarchies.
  - AADL provides binding properties to describe how software components are allocated to the execution platform.

Binding Properties

- Map software elements to platform elements using binding properties:
  - `Actual_Processor_Binding`
    - Specify which processor schedules and executes a thread or executes a device driver.
  - `Actual_Memory_Binding`
    - Specify the memory components in which executable code (process components) and data (data component) reside.
  - `Actual_Connection_Binding`
    - Specify the communication channels that are used by logical connections.

1. Open OSATE Framework
2. Go over Toy Example
3. Look at both textual and graphical representation.

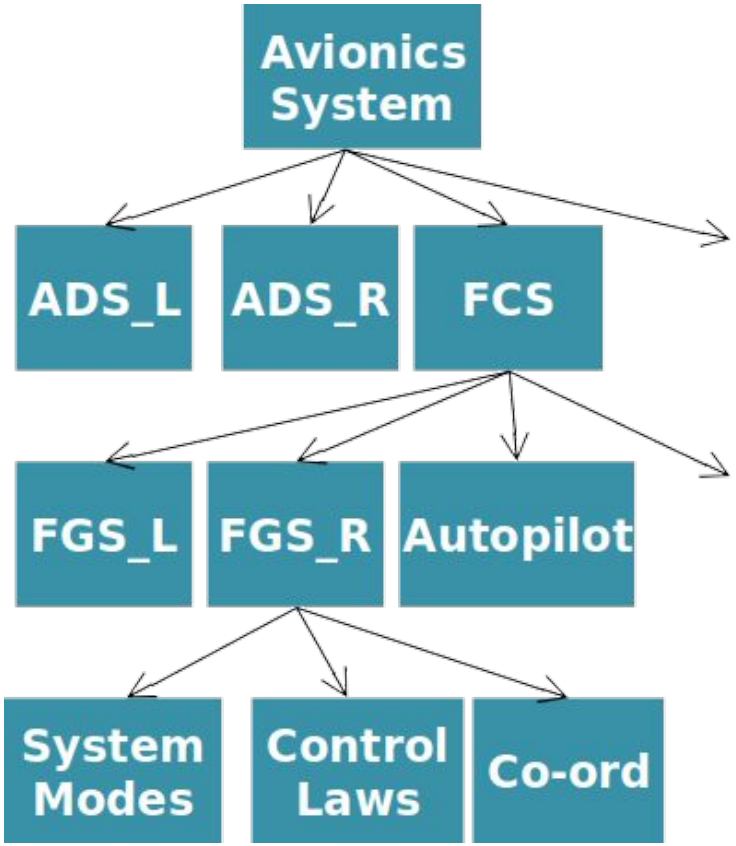
Compositional Verification Using AGREE

Compositional Verification

- Complex systems are usually designed as a hierarchical federation of elements.
  - As we descend the hierarchy, the **design** of some level becomes the **requirements** of elements at the next level of abstraction.
- We need to verify requirements and architectural design.
  - What can we **assume** about an element's input?
  - What do we **guarantee** about an element's output?
  - Do these assumptions and guarantees hold throughout the hierarchy?

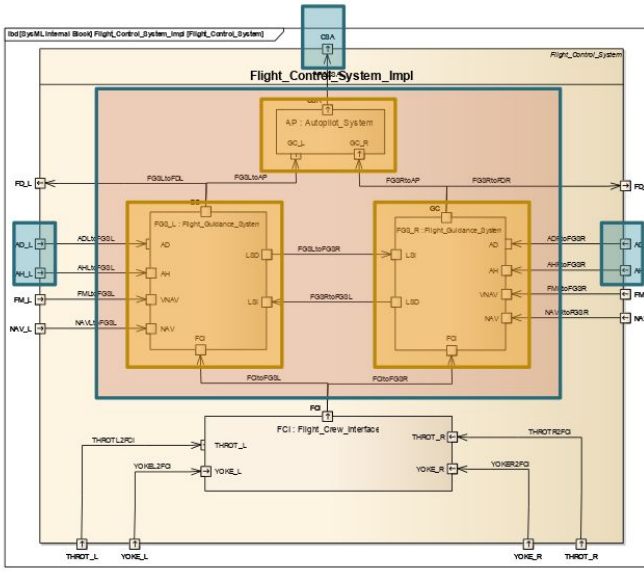
Hierarchical Reasoning

- Avionic System Requirement:
  - Under single-fault assumption, GC output transient response is bounded in time and magnitude
- Relies on:
  - Accuracy of air data sensors
  - Control commands from FCS
  - Mode of FGS
  - FGS control law behavior
  - Failover behavior between FGS systems
  - ….
  - Response of Actuators
  - Timing/Lag/Latency of Communications



Compositional Reasoning for Active Standby

- Want to prove a **transient response** property:
  - The autopilot will not cause a sharp change in pitch of aircraft.
  - Even when one FGS fails and the other assumes control
- Given assumptions about the **environment**:
  - The sensed aircraft pitch from the air data system is within some absolute bound and doesn't change too quickly
  - The discrepancy in sensed pitch between left and right side sensors is bounded.
- and guarantees provided by **elements**:
  - When a FGS is active, it will generate an acceptable pitch rate
- As well as **facts** provided by pattern application
  - Leader selection: at least one FGS will always be active (modulo one "failover" step)



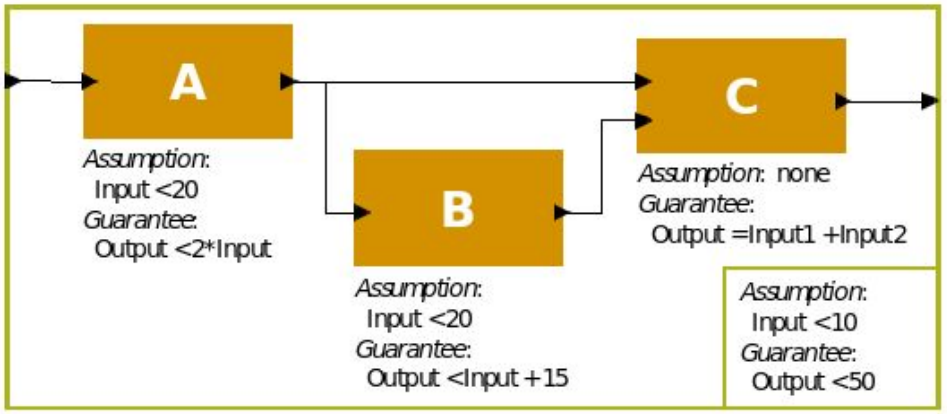

```
transient_response_1 :
assert true ->
abs(CSA.CSA_Pitch_Delta) <
CSA_MAX_PITCH_DELTA ;
transient_response_2 :
assert true ->
abs(CSA.CSA_Pitch_Delta -
prev(CSA.CSA_Pitch_Delta, 0.0))
< CSA_MAX_PITCH_DELTA_STEP ;
```

Contracts

- AGREE annotates AADL elements with:
  - **Assumptions** that elements make about the environment (input).
  - **Guarantees** that elements make about their output.
    - … As long as the assumptions are met.
- Each layer of the hierarchy is verified individually.
- AGREE attempts to prove the system-level guarantees in terms of the guarantees of its elements.

Contracts

Want to show that the system-level property holds, given the guarantees provided by the elements and the system assumption.



System-Level:

- `assume "System input range" : Input < 10;`
- `guarantee "System output range" : Output < 50;`

A

- `assume "A input range" : Input < 20;`
- `guarantee "A output range" : Output < 2*Input;`

B

- `assume "B input range":Input < 20;`
- `guarantee "B output range" : Output < Input + 15;`

C

- `guarantee "C output range": Output = Input1 + Input2;`

AGREE Language Basics

- Data types: integer, float, boolean
  - Integers can be 8-64 bit, signed or unsigned.
  - Float can be 32 or 64 bit.
- Variables can include AADL data classes.
  - Alarm offers Boolean `Is_Audio_Disabled`, integer `Log_Message_ID`
    - `Alarm.Is_Audio_Disabled => Alarm.Log_Message_ID > 3;`
- `prev(x)` uses the value of `x` from the last computational cycle.

AGREE Language Basics

- Equation statements create local variable declarations.
  - `eq ctr : int = prev(ctr + 1, 0);`
  - Variable that counts up from 0.
  - “Intermediate” expressions that prove guarantees.
- Properties allow specification of named Boolean expressions.
  - `property not_system_start_implies_mode_0 = not(OP_CMD_IN.System_Start) > (GPCA_SW_OUT.Current_System_Mode = 0);`
- Constants can be defined:
  - `const ADS_MAX_PITCH_DELTA: real = 1.0 ;`

1. Go over Toy Example
2. Execute AGREE
3. Examine Counterexample
4. Fix Model

Key Points

- Real-Time Systems have strict timing requirements, and tend to react to and influence an environment.
- Architecting them is difficult and requires careful task scheduling.
- Architectural Description Languages often have constructs for analyzing combinations of hardware and software with timing constraints.
  - AADL/AGREE designed for this purpose!

Next Time

- Service-Oriented Architecture

- Homework:
  - Reading Assignment 3 - Tonight!
  - Project, Part 4 - 12/06
  - Assignment 3 - 12/09