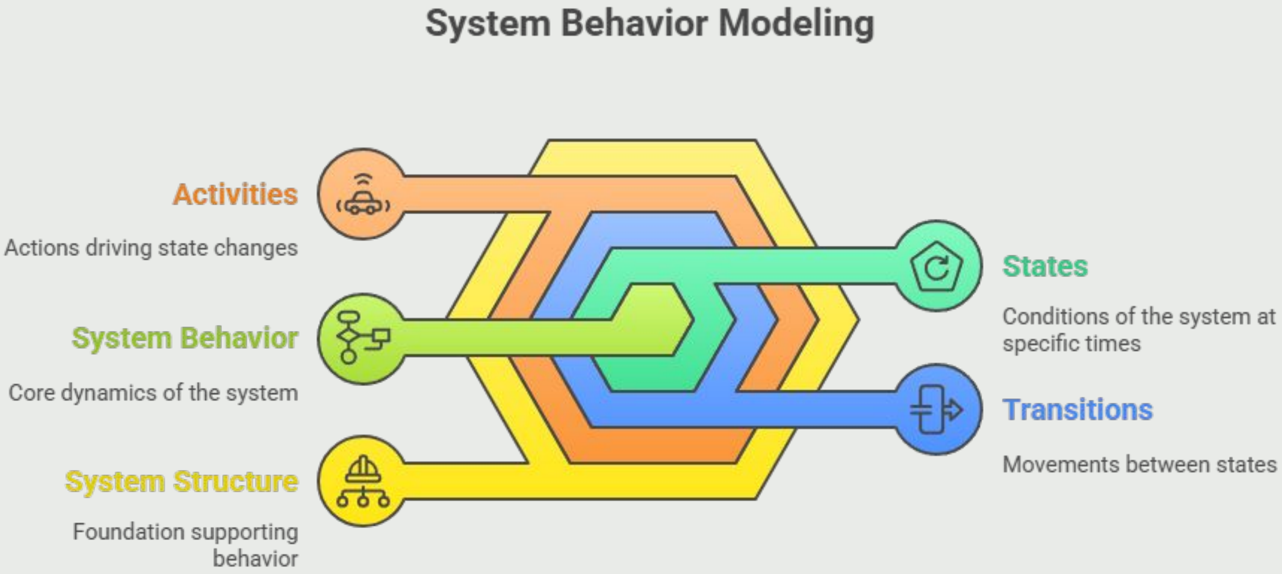
Behavioral Modeling Part 2
State Machines and Activities

Week 09

Behavioral Modeling Part 2

State Machines and Activities

Model how the system behaves over time using states, transitions, and structured activities — linked to the system structure.



Why Model Behavior Over Time?

Systems change over time — and modeling their behavior is key to understanding dynamic system behavior.

- **Structure** shows **what** a system *is*.
- **Behavior** shows **what** a system *does*, **when** it acts, and **how** it responds to change.

A behavior model helps us answer:

- *What triggers system behavior?*
- *What actions does the system perform?*
- *How does the system transition between different states over time?*

Modeling Behavior Complements Structure

SysML v2 provides rich behavior modeling that complements structure modeling.

- We model **Actions** → *What tasks are performed? How do they flow?*
- We model **States** → *What system states exist? How does the system transition between them?*
- We model **Flows** → *How is data or material transferred during behavior?*

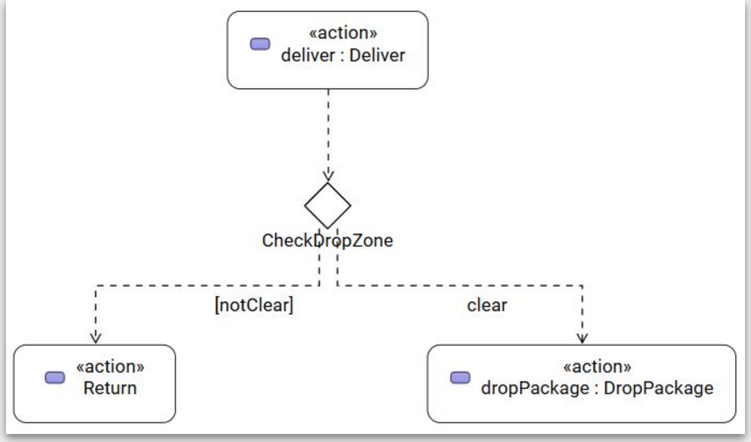
Together, these models provide a complete view of system dynamics.

Recap: Basic Actions Already Covered in Week 6

We have already introduced basic Action modeling:

- action def and action usages
- Basic sequencing of actions
- Simple activity-like behavior flows

This week, we will explore **advanced Action patterns** for richer behavior modeling.



Why Study Advanced Action Patterns?

Real systems do not behave as simple linear sequences of actions. They react to changing conditions, perform tasks asynchronously, and manage complex flows of behavior.

In SysML v2, advanced Action patterns allow us to model these richer dynamics:

- Actions can be explicitly performed by specific parts of the system.
- Behaviors can be interrupted or terminated in response to internal or external events.
- Parts may communicate using asynchronous message exchanges.
- Control structures such as branches, loops, guards, and parallel flows can be expressed clearly.

By using these patterns, we can model the true behavior of complex systems — not just their idealized task flows.

Perform Action Usage

A Perform Action usage specifies that a particular part of the system is responsible for executing an action. This allows us to clearly allocate behavior to structure — without requiring an explicit allocation relationship.

The perform keyword binds the execution of an action to a specific part usage.

For example:

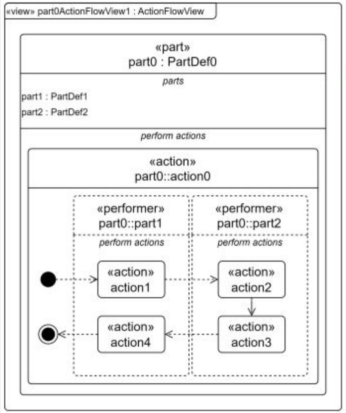
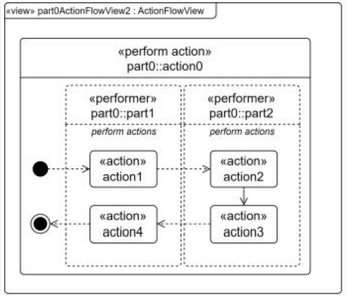
★ *The Flight Controller performs the "Compute Flight Path" action.*

★ *The Payload Manager performs the "Release Package" action.*

This pattern enables us to model which parts do what — bringing our behavior and structure models together.

Perform Action Usage

Element	Graphical Notation	Textual Notation

Perform Actions Swimlanes	
	
	Note. In View2, part0 has been elided.
		`package SwimLanes {`
		`part def Part0;`
		`part def Part1;`
		`part def Part2;`
		`part part0 : PartDef0 {`
		`perform action0;`
		`part part1 : PartDef1 {`
		`perform action0.action1;`
		`perform action0.action4;`
		`}`
		`part part2 : PartDef2 {`
		`perform action0.action2;`
		`perform action0.action3;`
		`}`
		`}`
		`action action0 {`
		`action action1;`
		`action action2;`
		`action action3;`
		`action action4;`
		
		`first start then action1;`
		`first action1 then action2;`
		`first action2 then action3;`
		`first action3 then action4;`
		`first action4 then done;`
		`}`
		`}`

Done Action vs. Terminate Action

SysML v2 distinguishes between "completion" of an action and "termination" of behavior:

A **Done Action** marks that an action usage has completed successfully.

- Successor actions of the containing behavior can proceed after this point.

A **Terminate Action** forces immediate termination of the enclosing behavior usage.

- No further actions or flows occur beyond this point.

 **Done Action usage:** done; → *Stop everything in this behavior — no continuation.*

 **Terminate Action usage:** terminate; → *This action is complete — normal flow continues.*

Understanding this distinction is critical for modeling realistic control of system behavior.

Send and Accept Actions

Many real systems communicate asynchronously — sending and receiving messages between parts or systems.

SysML v2 provides two key patterns to model this:

- **Send Action usage** — sends a message or flow of data to another part.
- **Accept Action usage** — waits for and accepts a specific message or data.

This enables modeling of asynchronous behavior — where parts interact without blocking the entire flow.

This is essential for modeling real-world systems such as control loops, event-driven systems, and distributed architectures.

Send and Accept Actions Example

Send Action	«send action» send1 **new** Picture() **via** displayPort	**item def** Picture; **port** displayPort; **action** send1 **send new** Picture() **via** displayPort;

Accept Action	«accept action» trigger1 scene : Scene **via** viewPort	**port** viewPort; **item def** Scene; **action** trigger1 **accept** scene : Scene **via** viewPort;

Control Nodes and Conditional Successions

Control Nodes and Conditional Successions allow us to model decisions, branches, merges, loops, and parallel flows in behavior models.

Control Nodes include:

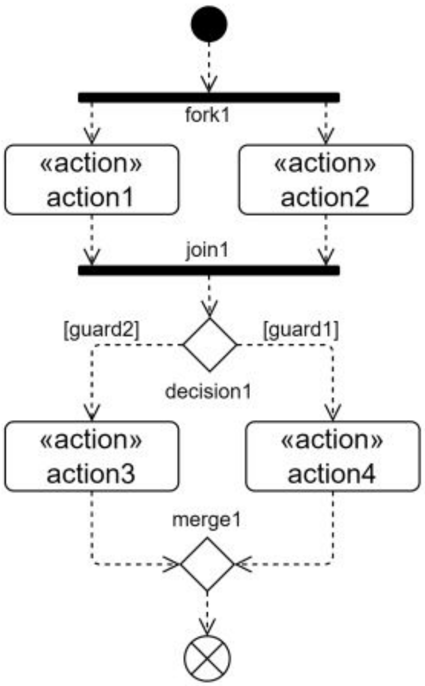
- **Initial node** — where behavior starts
- **Decision node** — where behavior branches based on conditions
- **Merge node** — where alternative branches converge
- **Fork node** — where parallel flows are initiated
- **Join node** — where parallel flows synchronize

Conditional Successions allow actions to be performed only if a specified condition (guard) is true:

Together, these patterns provide powerful constructs for modeling realistic control flow in complex behaviors.

Advanced Actions Example

Element	Graphical Notation	Textual Notation

Actions with Control Nodes		**first** start;
		then fork fork1;
		    then action1;
		    then action2;
		
		action action1;
		    then join1;
		action action2;
		    then join1;
		
		join join1;
		then decide decision1;
		    if guard2 **then**
		action3;
		    if guard1 **then**
		action4;
		
		action action3;
		    then merge1;
		action action4;
		    then merge1;
		
		merge merge1;
		then terminate;

What Is a State Machine in SysML v2?

A State Machine models the life cycle of a system or part — how it behaves over time as it transitions between different states.

Each **State** represents a distinct mode or condition of the system.

Transitions define how and when the system moves from one state to another, based on events, conditions, or time.

State Machines are used to model:

- Operational phases (e.g., *Standby*, *Takeoff*, *Cruise*, *Landing*)
- Modes of operation (e.g., *Nominal*, *Degraded*, *Emergency*)
- Life cycle states (e.g., *Initialized*, *Active*, *Terminated*)

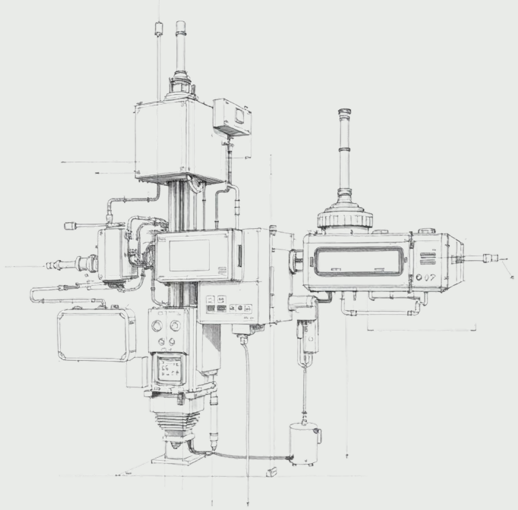
State Machines give us a powerful way to represent how the system evolves across time and responds to events.

State Definition and Usage

In SysML v2, a State is defined using state def, and used with state or exhibit state.

- A **State Definition** (state def) defines a named state that can be exhibited by parts or behaviors.
- A **State Usage** (state) places that state in a State Machine model.
- **Exhibit State Usage** (exhibit state) specifies that a part can exhibit a particular state as part of its behavior.

This separation between definition and usage enables States to be reused and consistently applied across the model.



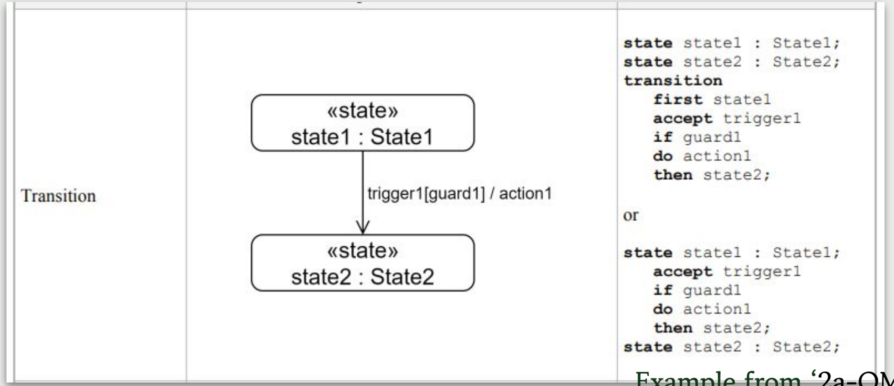
Transition Usage

A Transition defines how a system moves from one State to another — triggered by an event, condition, or time. In SysML v2, Transitions are expressed using **Transition Usages** between States.

A Transition may include:

- **Trigger** — the event or condition that causes the transition
- **Guard** — a condition that must be true for the transition to occur
- **Effect** — an optional action performed during the transition

Transitions are essential to capturing the dynamic behavior and responsiveness of the system.

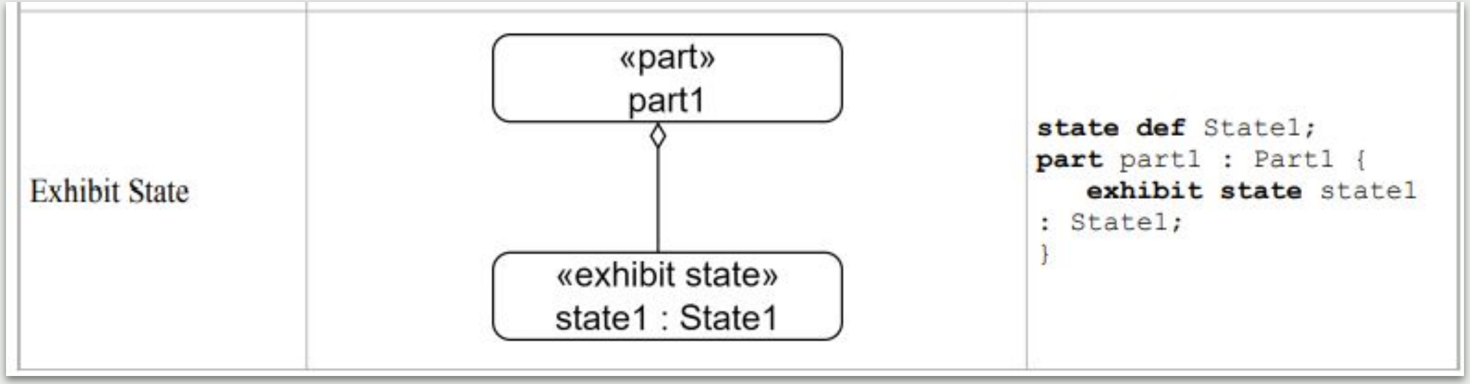


Example from ‘2a-OMG_Systems_Modeling_Language.pdf’ page 116

Exhibit State Usage

Exhibit State Usage specifies that a particular part exhibits a given State Machine as part of its behavior.

This is how we connect **State Models** to **Structure** in SysML v2 — without needing an explicit allocation.



Example from ‘2a-OMG_Systems_Modeling_Language.pdf’ page 116

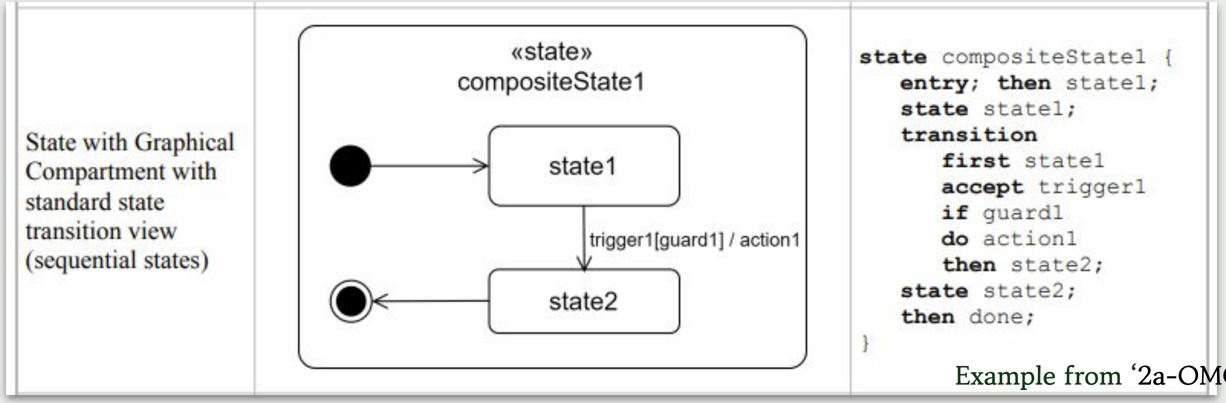
Composite State

A Composite State contains its own internal State Machine — allowing us to model hierarchical, nested behavior. This is useful when a system operates in **modes**, where each mode has its own internal states.

Key concepts:

- A **Composite State** is a State that contains a nested State Machine.
- The system must first enter the Composite State before any nested states become active.
- Transitions can occur both **into** and **out of** the Composite State, as well as **within** it.

Composite States help manage complexity — by breaking down behavior into nested levels of detail.



Example from ‘2a-OMG_Systems_Modeling_Language.pdf’ page 115

Parallel States

Parallel States allow a system to be in multiple active states at the same time — across different dimensions of behavior.

This is used to model behaviors that operate **concurrently**, such as:

- Monitoring sensors while flying
- Managing communications while navigating
- Performing health checks while executing a mission

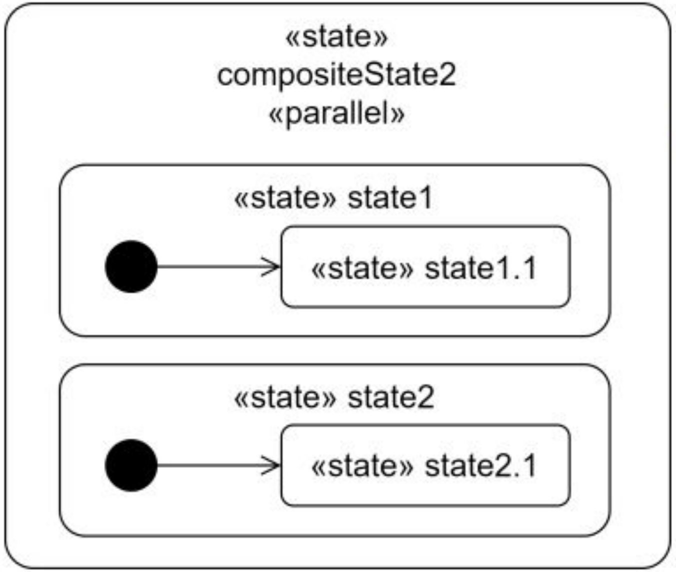
Key concepts:

- A Parallel State contains **Regions** — each with its own independent State Machine.
- All regions operate concurrently — the system is in one state from each region at the same time.
- Transitions can occur independently within each region.

Parallel States help model realistic multi-threaded behaviors in complex systems.

Parallel States

State with Graphical Compartment with standard state transition view (parallel states)




```
state compositeState2
parallel {
   state state1 {
      entry; then
'state1.1';
      state 'state1.1';
   }
   state state2 {
      entry; then
'state2.1';
      state 'state2.1';
   }
}
```

Example from ‘2a-OMG_Systems_Modeling_Language.pdf’ page 115

Actions Within States (Entry, Do, Exit)

In SysML v2, a State can include associated Actions — to model behavior that occurs on entry, during, and on exit from the State. These are called **Subactions**:

- **Entry Action** — performed when the State is entered
- **Do Action** — performed while the State is active
- **Exit Action** — performed when the State is exited

This enables precise modeling of behavior inside each State — not just transitions between States.

State with entry, do and exit actions.


```
state state1 : StateDef1
{
    entry action1;
    do action2;
    exit action3;
}
```

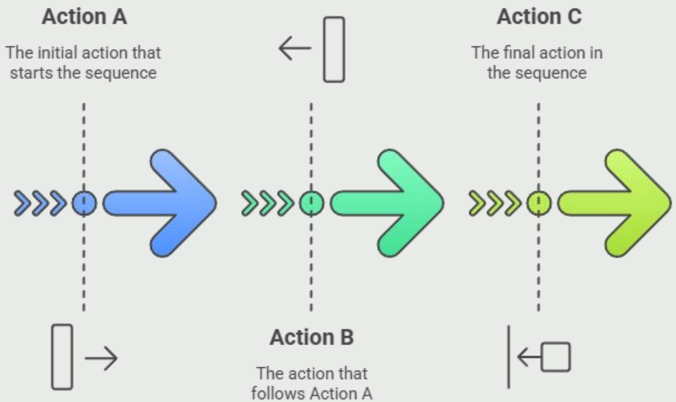
Example from '2a-OMG_Systems_Modeling_Language.pdf' page 115

What Is Control Flow?

Control Flow models the ordering of actions — specifying *when* actions should occur in relation to one another.

- A **Control Flow** defines a sequencing relationship between actions.
- The flow carries **control tokens**, not data.
- An action starts when **control is passed to it** from its predecessor.

Control Flow is used to model the logical flow of behavior — *what happens first, next, last.*

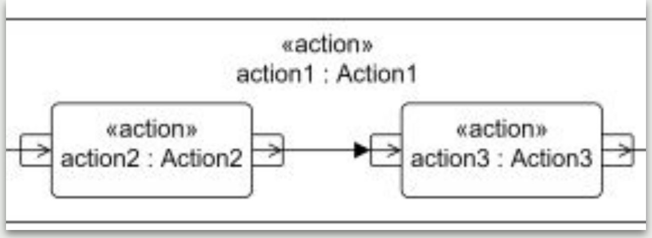


What Is Object Flow?

Object Flow models the movement of data, material, or information between actions.

- An **Object Flow** represents the passing of an object or value from one action to another.
- The flow carries **objects or data tokens** — not just control.
- The receiving action can use this data as input.

Object Flow is essential for modeling how information or material moves through a system's behavior.




```
    flow action2.output2
        to
action3.input3;
```

Example from '2a-OMG_Systems_Modeling_Language.pdf' page 88

Practice

Model State Machine and Actions for Vehicle System

In this practice, we will model a simple State Machine and associated Actions for a Vehicle System.

System context: *Electric Vehicle Controller* — manages vehicle operation and charging.

Example States: *Off, StartUp, Driving, Charging*

Example Actions: perform systemCheck, perform manageDrivePower, perform handleCharging

Now build this model in SysON:

- ➔ Define States and Transitions
- ➔ Add Entry/Do/Exit Actions
- ➔ Add Object Flows (optional)
- ➔ Use exhibit state to connect to vehicleController part

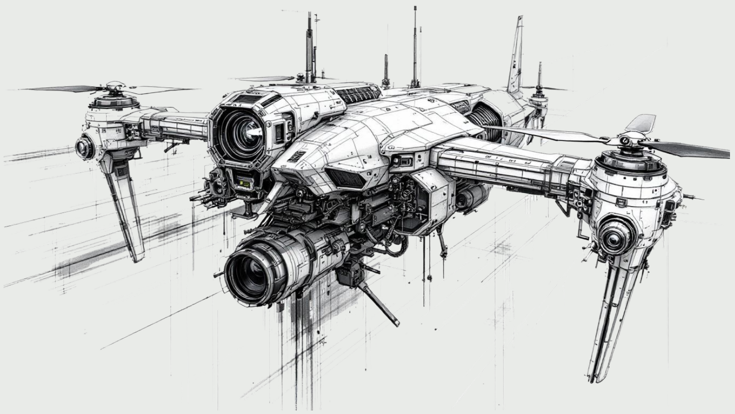
Student Project - System Behavior

Add Behavior to Your Drone Model

Apply what you learned this week to your **Drone System project**:

Define a State Machine for one key part of your Drone — e.g.:

- *Flight Controller*
- *Payload System*
- *Sensor System*



Include: 

- State Definitions
- Transitions with triggers
- Entry / Do / Exit Actions
- Use exhibit state to connect behavior to structure

Use **Control Flow** and **Object Flow** where appropriate.

Summary of Week 9

This week, we learned how to model **dynamic behavior** using advanced behavioral constructs in SysML v2:

Advanced Action Patterns:

- *Perform Action* — link behavior to parts
- *Done* and *Terminate* actions — control behavior flow and termination
- *Send* and *Accept* actions — model asynchronous communication
- *Control Nodes* and *Conditional Successions* — define rich control flow

State Machines:

- *State Definition* and *Usage*
- *Transitions* with triggers and guards
- *Exhibit State* — link States to parts
- *Composite* and *Parallel States* — model complex behavior
- *Actions within States* — entry, do, exit

Control Flow vs. Object Flow:

- Control Flow — sequencing behavior
- Object Flow — passing data between actions

QUESTION!