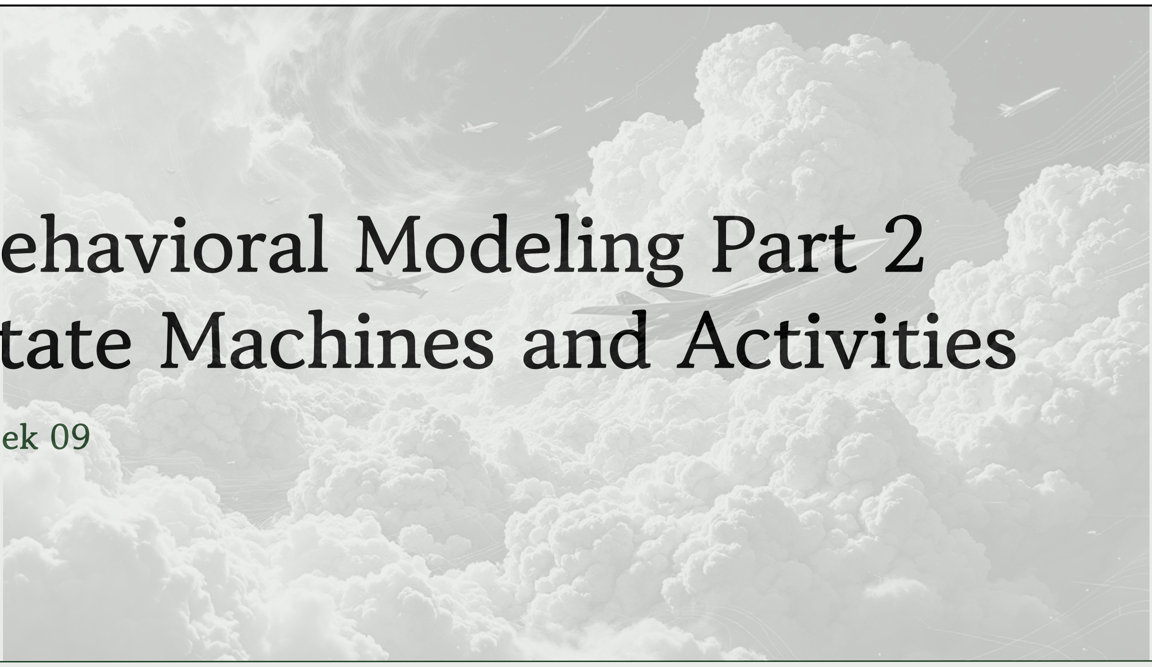
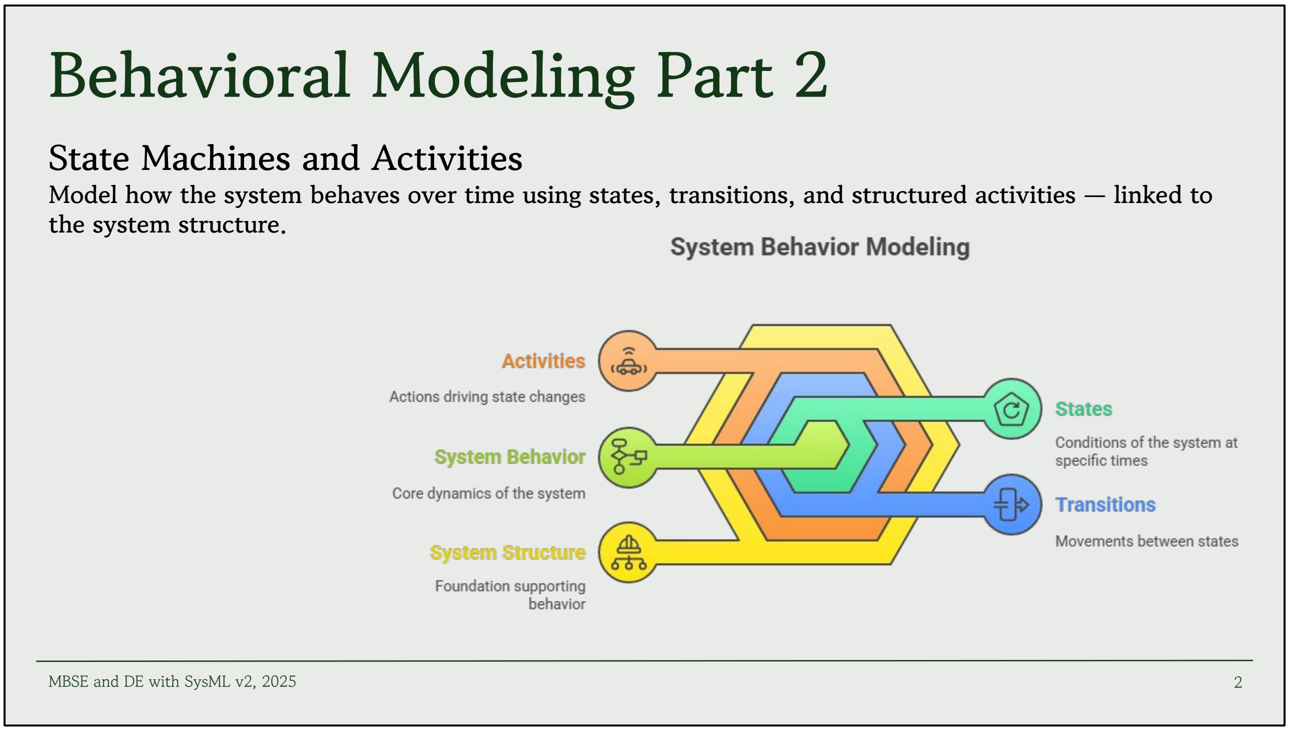


Behavioral Modeling Part 2 State Machines and Activities

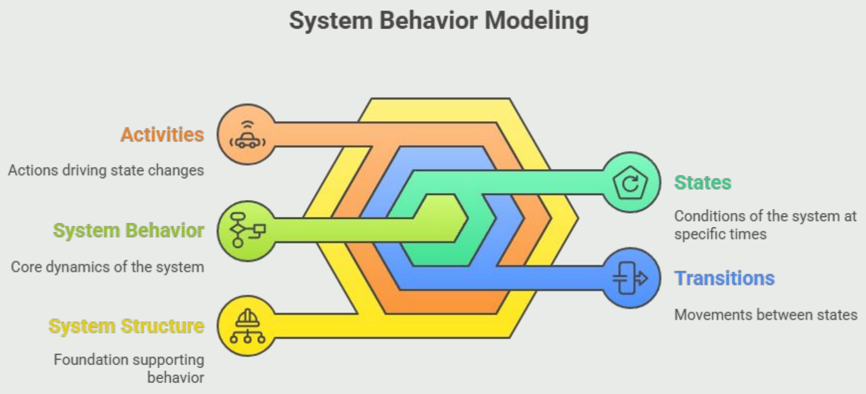
Week 09



Behavioral Modeling Part 2

State Machines and Activities

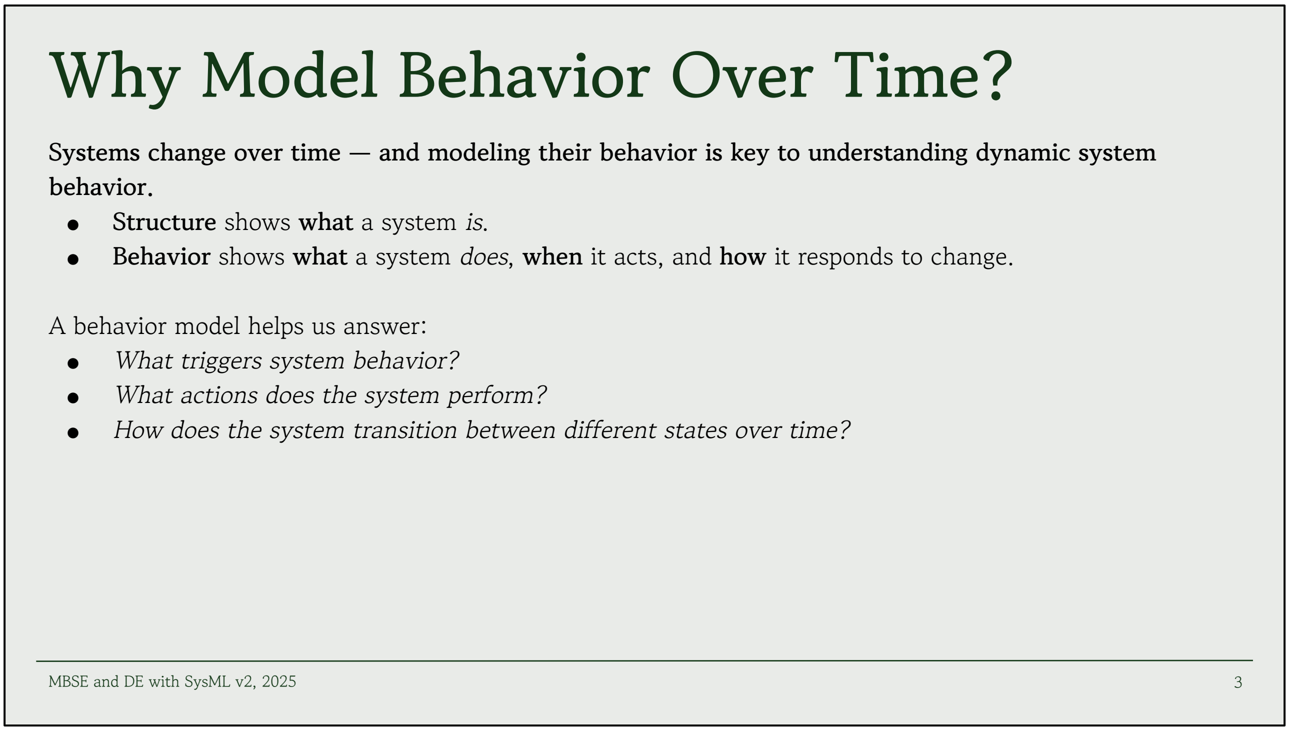
Model how the system behaves over time using states, transitions, and structured activities — linked to the system structure.


System Behavior Modeling
Activities
Actions driving state changes
System Behavior
Core dynamics of the system
System Structure
Foundation supporting behavior
States
Conditions of the system at specific times
Transitions
Movements between states

Welcome to **Week 9** — we are continuing our journey into **Behavioral Modeling**.
In Week 6, we introduced basic **Actions** and simple flow-based behavior.
This week, we go deeper:

- We will model **behavior over time** using **States** and **State Transitions**.
- We will explore more advanced **Action Patterns** — such as **Perform Action**, **Send/Accept Actions**, **Terminate Actions**.
- We will see how behavior links to system structure — and how to model **Behavior Life Cycles** for parts.

Our goal: learn how to model both **dynamic behavior** and **system life cycles** — using **State Machines** and **Activities**.



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?*

Why Model Behavior Over Time?

Systems are not static — they **change and react** as they operate. While a structural model tells us *what components exist and how they are connected*, it does not tell us *how the system behaves over time*.

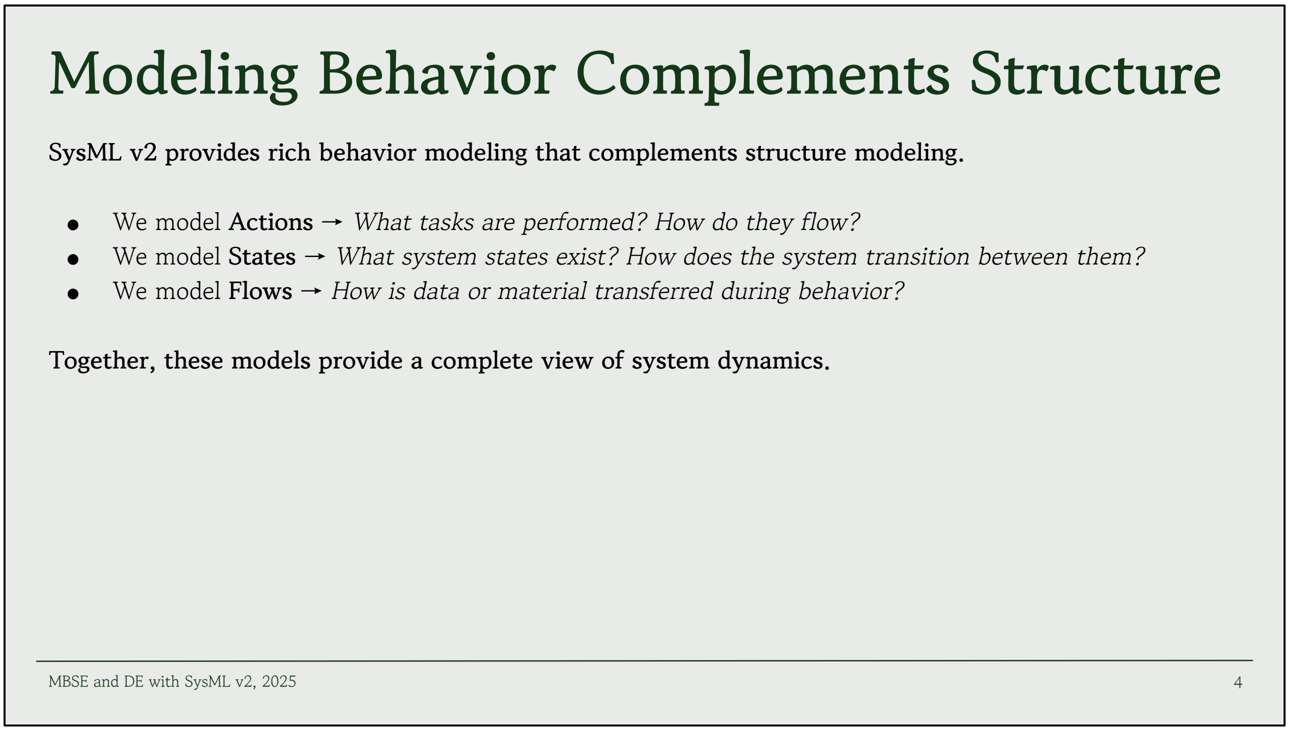
This is where **behavior modeling** is essential:

- We model **triggers** — what causes behavior to start.
- We model **actions** — what tasks the system performs.
- We model **state transitions** — how the system evolves across different phases of operation.

For example:

- How does a drone transition from *Standby* to *Takeoff* to *Cruising* to *Landing*?
- What happens when a fault is detected mid-flight?

This temporal understanding is critical for engineering reliable systems.



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.

Why Model Behavior Over Time?

In SysML v2, **Behavioral Modeling** is designed to work **together with Structure Modeling**.

➔ Structure shows *what parts exist* and *how they are connected*.

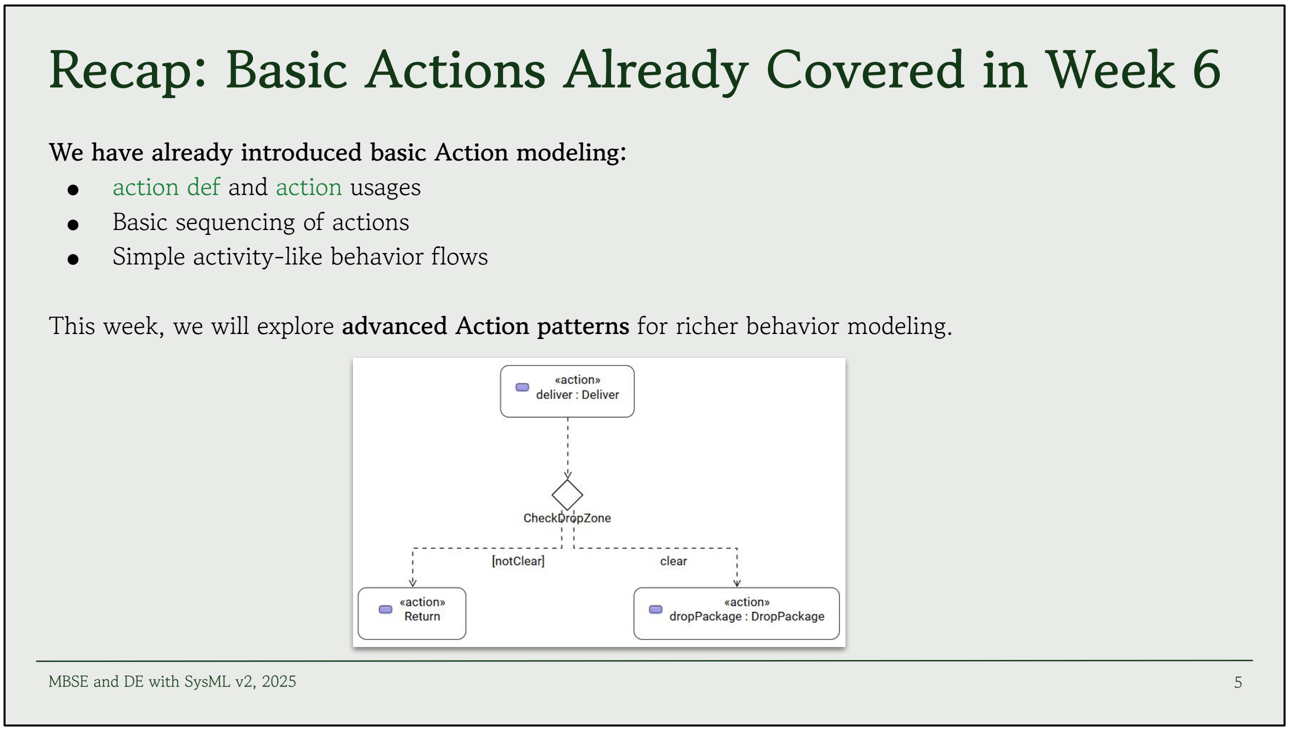
➔ Behavior shows *what parts do* and *how their behavior unfolds over time*.

Specifically:

- **Actions** model the **flow of tasks** — such as starting motors, processing data, sending signals.
- **States** model the **life cycle of the system** — which phases it goes through, how it transitions between them.
- **Flows** model **how things move** — such as data, energy, materials, commands.

When we combine these, we get a **full understanding of system dynamics** — both *what it is* and *what it does*.

This is critical in systems like our **Autonomous Drone** — where time, tasks, and state transitions are central to safe and effective operation.

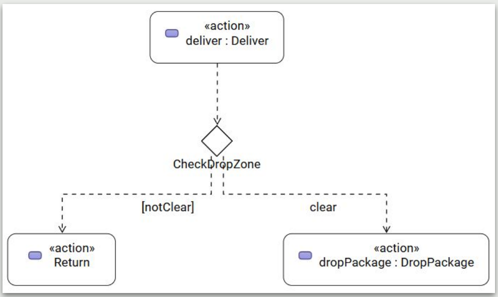


Recap: Basic Actions Already Covered in Week 6

We have already introduced basic Action modeling:

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

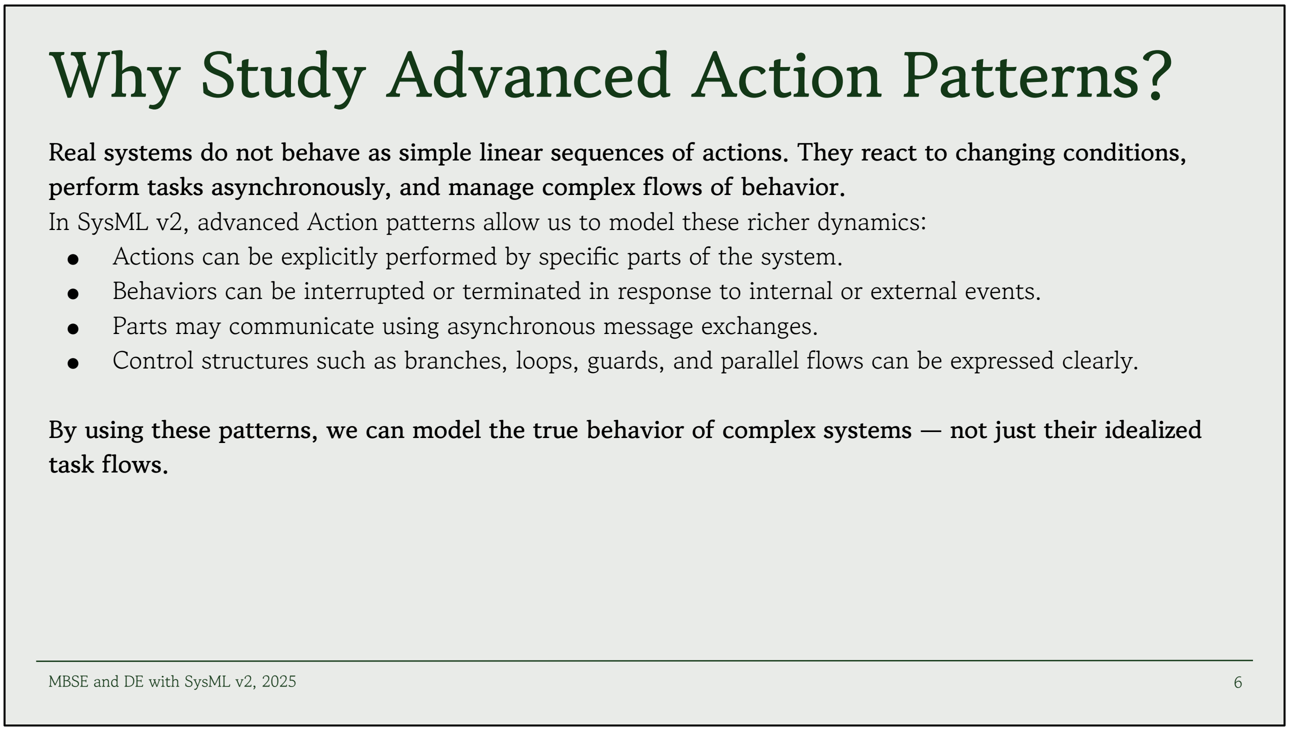
This week, we will explore **advanced Action patterns** for richer behavior modeling.



Advanced Action Patterns in SysML v2.
Before we dive into this week's new material, let's recap what we've already covered in **Week 6**:
👉 We introduced action def — to define reusable **Actions**.
👉 We used action to model **Action usages** in a behavior flow.
👉 We modeled **basic sequencing of Actions** — simple "do A → do B → do C" patterns.
This gave us the foundation to model **basic functional flows** — like **Takeoff**, **Navigate**, **Land**.
Now — this week — we will go further.
👉 We will explore **more advanced Action patterns**:

- Perform Actions → linking Actions to Parts
- Terminate Actions → to model stopping behavior
- Send/Accept Actions → to model asynchronous interaction
- Control Nodes → for conditional and parallel flows

These will enable us to model **richer, more realistic behaviors** for our systems.



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.

Advanced Action Patterns in SysML v2.
When we first introduced Actions in **Week 6**, we focused on basic flows: simple sequences of tasks.
However, real systems are rarely this simple.
Consider an **Autonomous Drone**:

- The **Flight Controller** performs actions continuously, not just in sequence.
- A **Terminate Action** may be needed if a fault is detected.
- The **Drone** sends and receives **messages** to ground control asynchronously.
- Its behavior includes **conditional flows** — "If GPS lock is lost, enter Recovery Mode."
- It may run **parallel behaviors** — such as navigating and monitoring battery health simultaneously.

Advanced Action patterns in SysML v2 give us tools to model these real-world dynamics.
This week, we will learn how to use these patterns to create **robust, realistic behavioral models**.

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.

Advanced Action Patterns in SysML v2.

One of the most important advanced Action patterns is **Perform Action usage**.

In real systems, we care not only *what actions happen*, but also *who performs them*.

In SysML v2, perform provides a very clean way to specify this — without needing an explicit allocation.

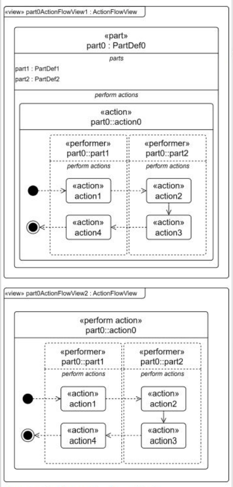
This states that the part usage flightController performs the action computeFlightPath.

👉 This is very useful in our **Drone model** — for example:

- The **Flight Controller** performs *Compute Flight Path*.
- The **Sensor Suite** performs *Collect Data*.
- The **Payload Mechanism** performs *Release Package*.

We will use **Perform Action** a lot in our hands-on modeling — it’s simple but very expressive.

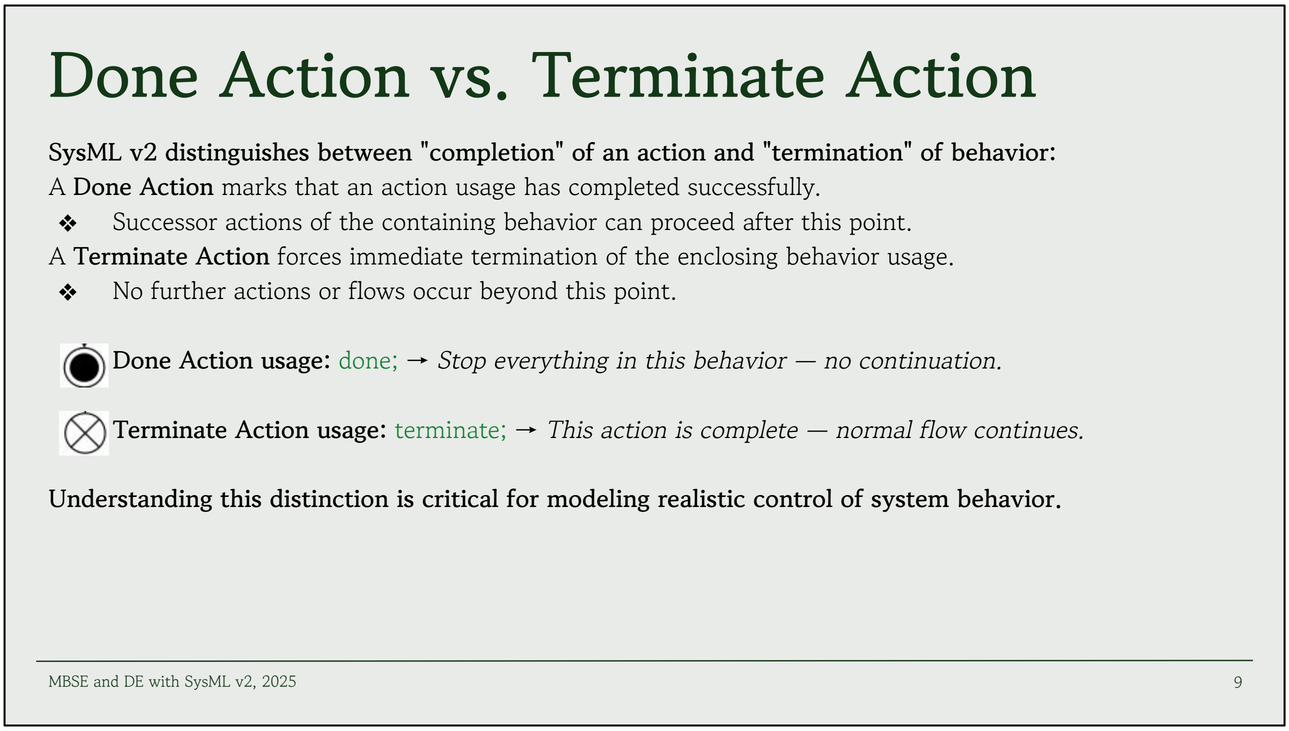
Perform Action Usage

Element	Graphical Notation	Textual Notation
Perform Actions Swimlanes	 «view» part0ActionFlowView1 : ActionFlowView «part» part0 : PartDef0 parts part1 : PartDef1 part2 : PartDef2 perform actions «action» part0::action0 «performer» part0::part1 perform actions «performer» part0::part2 perform actions «action» action1 «action» action2 «action» action4 «action» action3 «view» part0ActionFlowView2 : ActionFlowView «perform action» part0::action0 «performer» part0::part1 perform actions «performer» part0::part2 perform actions «action» action1 «action» action2 «action» action4 «action» action3 **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;
   }
}
```

Advanced Action Patterns in SysML v2.
Example of Perform Action in both Graphic Notation and Textual Notation from '2a-OMG_Systems_Modeling_Language.pdf' page 90



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.*

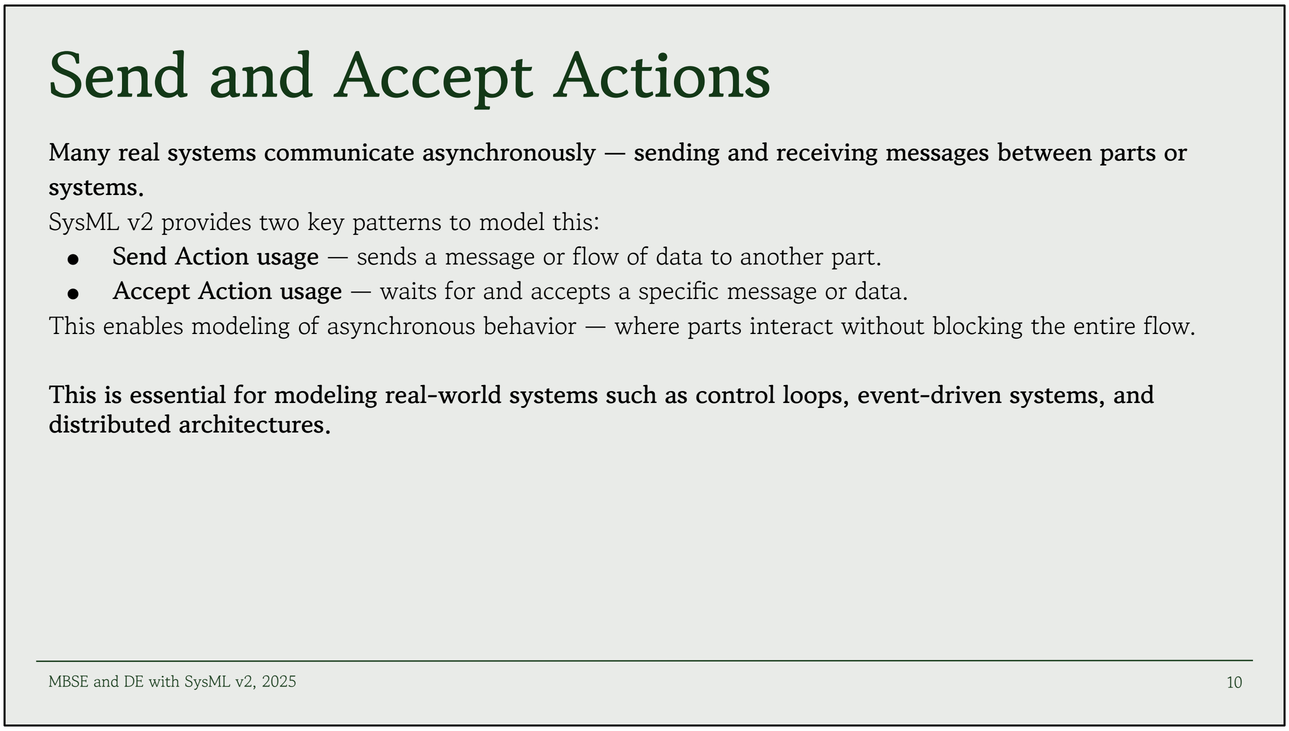
Understanding this distinction is critical for modeling realistic control of system behavior.

Advanced Action Patterns in SysML v2.
It is very important to model the **end of behavior** precisely.
SysML v2 gives us two distinct patterns:

- **Done Action** — marks that the *current action* is finished.
  - No further actions will execute.
  - Example: if a *Critical Battery Fault* occurs, we might use **Terminate** to stop the entire *Mission Behavior* immediately.
- **Terminate Action** — tells the *entire enclosing behavior* (such as a state behavior or an activity) to stop immediately.
  - Successors can proceed.
  - Example: after a *Scan Area* action completes, we may proceed to *Report Data.*

Remember:

- **Done** = stop entire behavior now
- **Terminate** = end of one action, flow continues



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.

Advanced Action Patterns in SysML v2.

In real systems — especially cyber-physical or distributed systems — parts often **send and receive messages**.

Example:

➔ The Drone's **Flight Controller** sends telemetry to Ground Control.
➔ The Ground Control sends updated mission commands to the Drone.

Send Action usage:

➔ Sends a message or data — does not wait for a response.
➔ The sender can continue its behavior.

Accept Action usage:

➔ Waits for a specific message or data — resumes behavior when message is received.

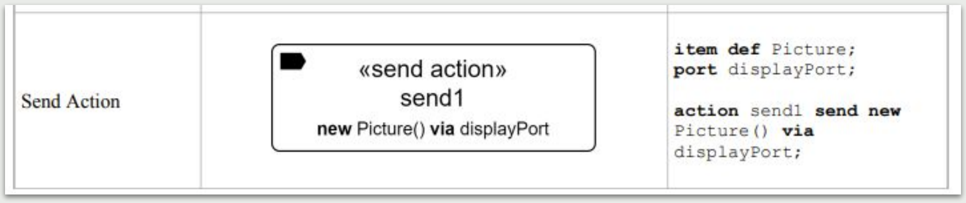
Key point:

- This models **asynchronous behavior** — not everything happens in a strict sequence.
- Parts of the system can be working in parallel, communicating via messages.

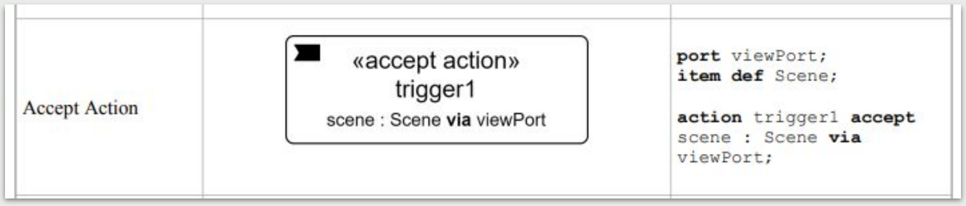
We possible use **Send/Accept Actions** in our **Drone model** to model:

- **Telemetry messages**
- **Mission command updates**
- **Event-driven responses**

Send and Accept Actions Example

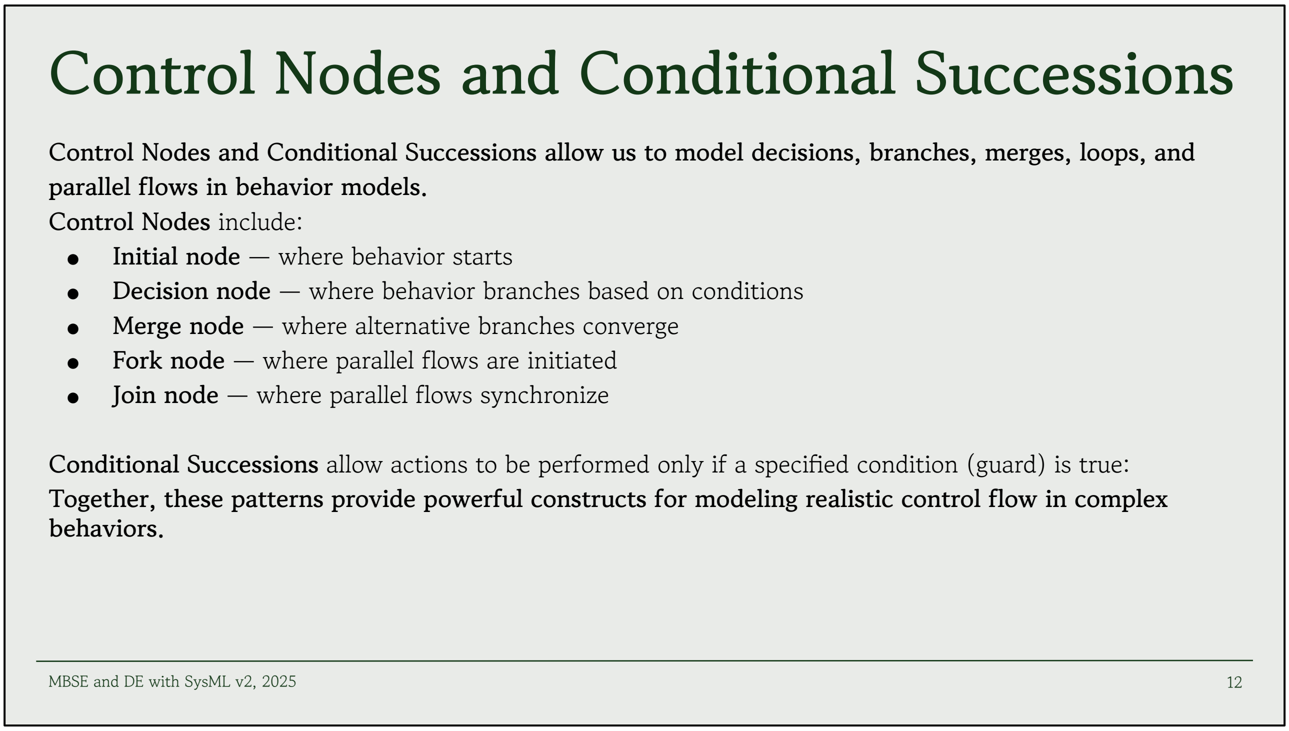


Send Action	«send action» send1 **new** Picture() **via** displayPort	`item def Picture;` `port displayPort;` `action send1 send new Picture() via displayPort;`



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

Advanced Action Patterns in SysML v2.
Example of Send and Receive Action in both Graphic Notation and Textual Notation from '2a-OMG_Systems_Modeling_Language.pdf' page 97



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 Action Patterns in SysML v2.
So far, we've looked at simple **sequences** of actions. But real behavior is rarely just a straight line.
We need ways to model:

- ❖ **Branches** — different paths based on conditions
- ❖ **Loops** — repeating actions
- ❖ **Parallel flows** — multiple actions happening concurrently
- ❖ **Merges** — converging paths

Control Nodes give us these capabilities:

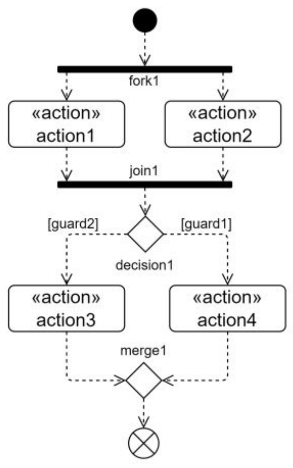
- **Decision Node** — like an if in programming
- **Fork Node** — run multiple flows in parallel
- **Join Node** — wait for all parallel flows to finish
- **Merge Node** — bring together branches
- **Initial/Final Node** — mark start and end of behavior

Conditional Successions allow us to attach a **guard** — a condition that must be true for the flow to proceed:
Key takeaway:

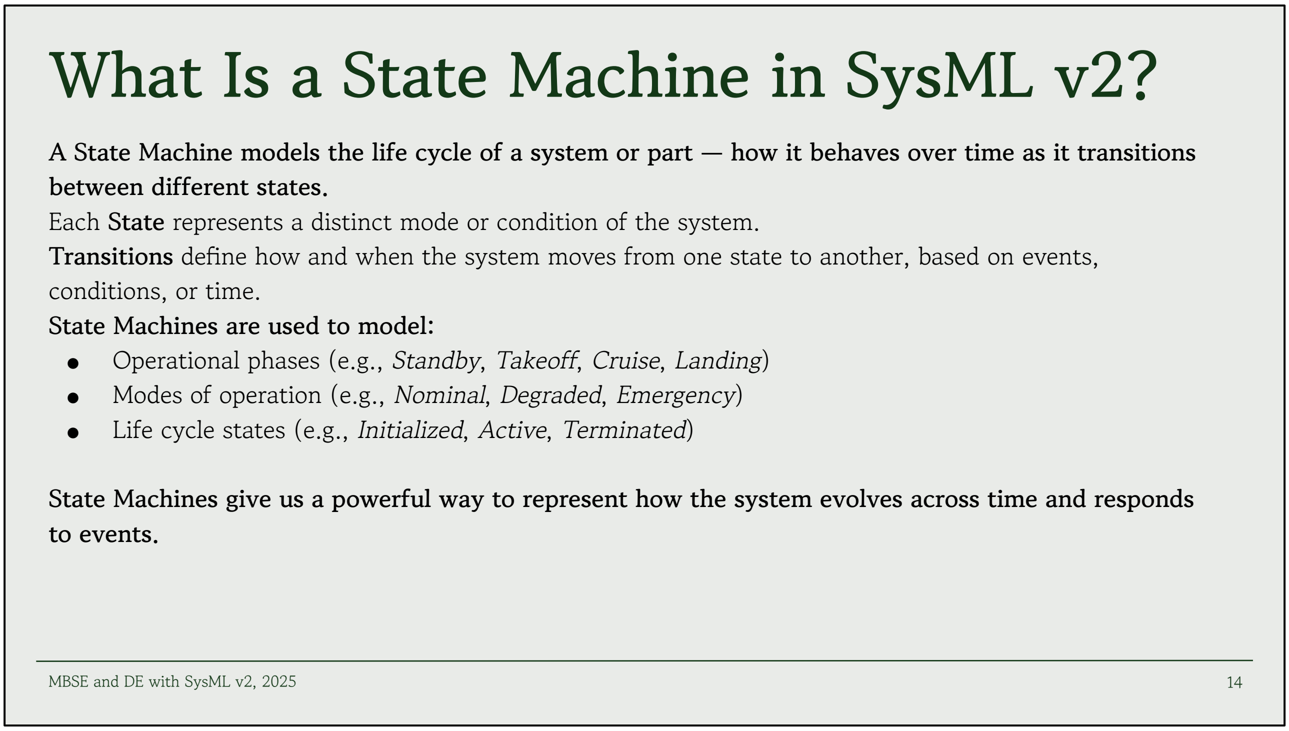
- ★ These constructs allow us to model **rich, realistic control flow** — beyond simple sequencing.
- ★ We will use them to build **robust Drone behavior models** — with guards, safety checks, and parallel monitoring.

Advanced Actions Example

Element	Graphical Notation	Textual Notation

Actions with Control Nodes	 fork1
	«action» action1
	«action» action2
	join1
	[guard2]
	[guard1]
	decision1
	«action» action3
	«action» action4
	merge1
		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;

Advanced Action Patterns in SysML v2.
Example of some advance actions in both Graphic Notation and Textual Notation from '2a-OMG_Systems_Modeling_Language.pdf' page 92



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.

Modeling Behavior with States
A **State Machine** is one of the most important behavior modeling tools in SysML v2.
👉 It allows us to model the **life cycle** of a system or part:

- What are the **different states** it can be in?
- **How does it transition** between states?
- What **events, conditions, or timing** cause those transitions?

States capture important modes of operation:
👉 Example — for an **Autonomous Drone**:

- *Standby*
- *Takeoff*
- *Cruise*
- *Delivery*
- *Return to Base*
- *Landing*

Transitions tell us *how and when* the drone moves between those states.
👉 Example — *Landing* transition might occur if:

- The delivery mission completes
- An operator command is received
- A battery fault occurs

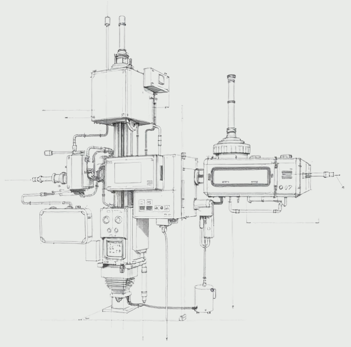
State Machines make it easy to visualize and analyze this dynamic behavior.
👉 They complement Actions — giving us both *what the system does* and *how its life cycle unfolds over time*.

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.



Modeling Behavior with States

Just like in other parts of SysML v2, **States** follow the **Definition and Usage** pattern:

➢ **state def** defines what a State *is*: state def Cruise;

➢ **state** places that State *in a State Machine*: state Cruise;

➢ **exhibit state** links a **part** to the fact that it can exhibit a State: exhibit state flightState : flightControllerState;

This gives us a lot of flexibility:

👉 We can define a common set of States (e.g. *Standby*, *Cruise*, *Landing*).

👉 We can use them in multiple State Machines.

👉 We can specify that *different parts* exhibit certain States.

For example:

👉 The **Drone** exhibits *Flight States*.

👉 The **Payload System** exhibits *Delivery States*.

👉 We will see examples of **exhibit state** shortly — it is how we "bind" States to **structure** (parts).

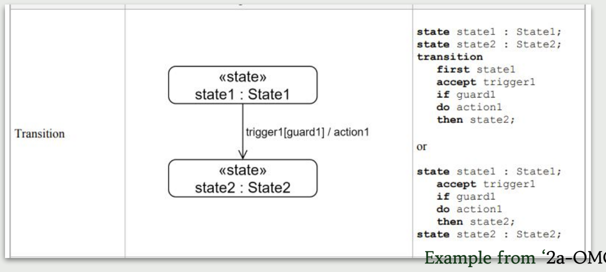
Transition Usage

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.



```
state state1 : State1;
state state2 : State2;
transition
   first state1
   accept trigger1
   if guard1
   do action1
   then state2;

or

state state1 : State1;
   accept trigger1
   if guard1
   do action1
   then state2;
state state2 : State2;
```

Example from '2a-OMG_Systems_Modeling_Language.pdf' page 116

Modeling Behavior with States

Transitions are the heart of a **State Machine** — they define *how and when* the system moves between States.

Each Transition can specify:

👉 **Trigger** — What event causes the transition?

👉 **Guard** — Is there a condition that must be true?

👉 **Effect** — Is any Action performed during the transition?

Key point:

- Without Transitions, a State Machine would just be a static list of States.
- **Transitions give it life** — showing how the system behaves dynamically in response to events.

Real-world example — Autonomous Drone:

- When *takeoffCommandReceived*, transition from *Standby* to *Cruise*.
- If *lowBatteryDetected*, transition from *Cruise* to *Landing*.
- If *MissionCompleted*, transition from *Delivery* to *ReturnToBase*.

We will build these kinds of State Machines for our Drone model.

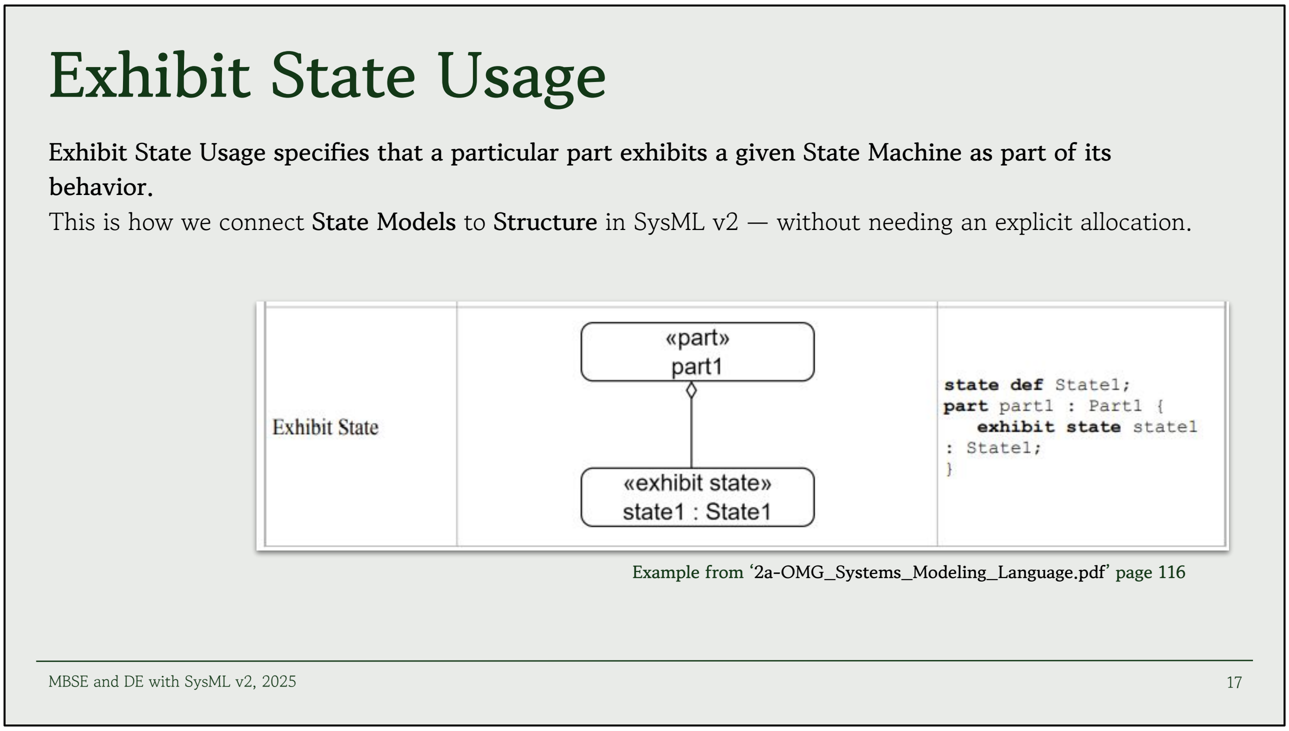
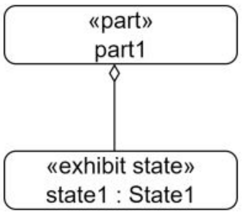


Exhibit State Usage

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.

Exhibit State		`state def State1;` `part part1 : Part1 {` `exhibit state state1 : State1;` `}`

```
state def State1;
part part1 : Part1 {
    exhibit state state1
: State1;
}
```

Example from '2a-OMG_Systems_Modeling_Language.pdf' page 116

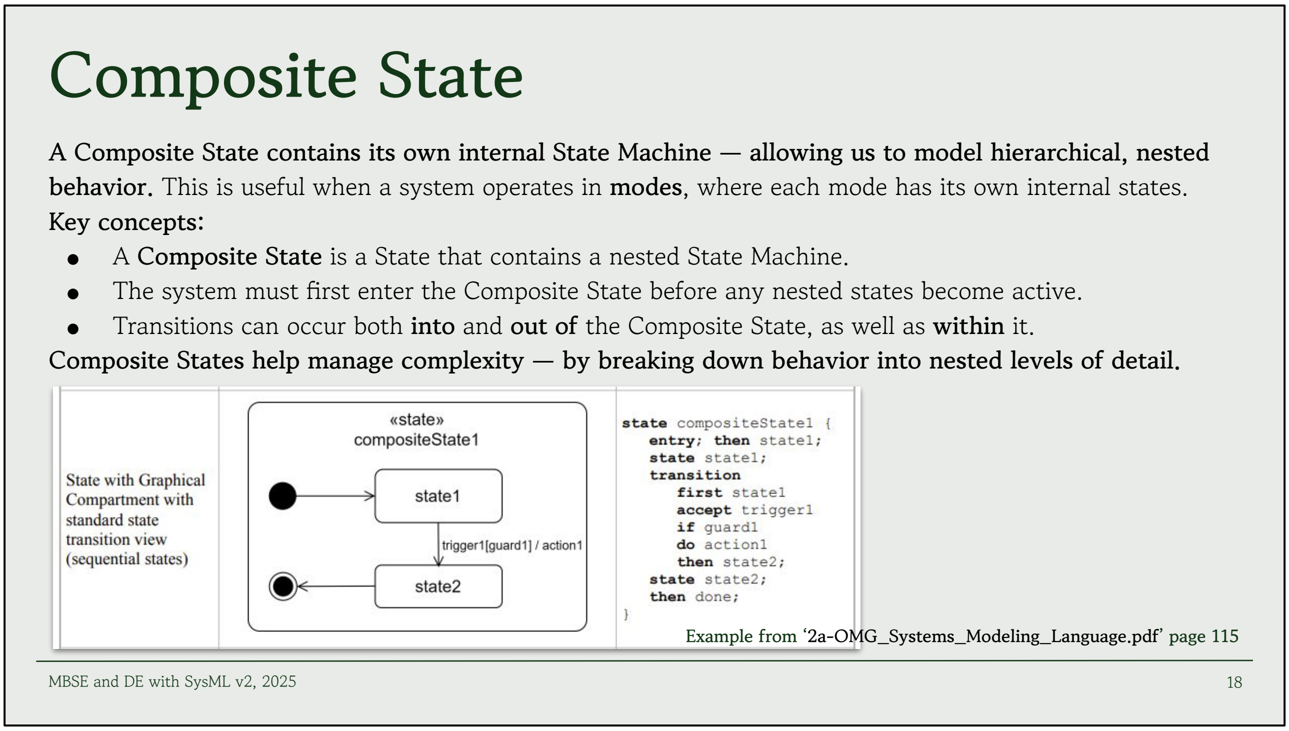
Modeling Behavior with States
We've now seen how to define **States** and how to model **Transitions** between them.
But how do we connect a **State Machine** to the **system structure** — so that a specific **part** behaves according to the State Machine?
We use **Exhibit State Usage**:

```
part fligetColler: flightController {
  exhibit state flightState  : flightControllerState;
}
```

This says:

- ➔ The part flightController exhibits (follows) the State Machine FlightStates.
- ➔ No need for an **allocation** — this is a direct, clean relationship.

Example in our Drone system:
👉 The flightController exhibits the *Flight States* (Standby, Takeoff, Cruise, Landing).
👉 The payloadSystem could exhibit *Delivery States* (Idle, Preparing, Delivering, Securing).
Benefits of this approach:
✅ Keeps structure and behavior **modular and reusable**.
✅ Makes it **clear which parts own which behaviors**.
✅ Works perfectly with the **Definition and Usage** style of SysML v2.



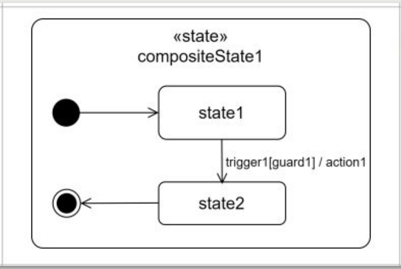
Composite State

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.

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



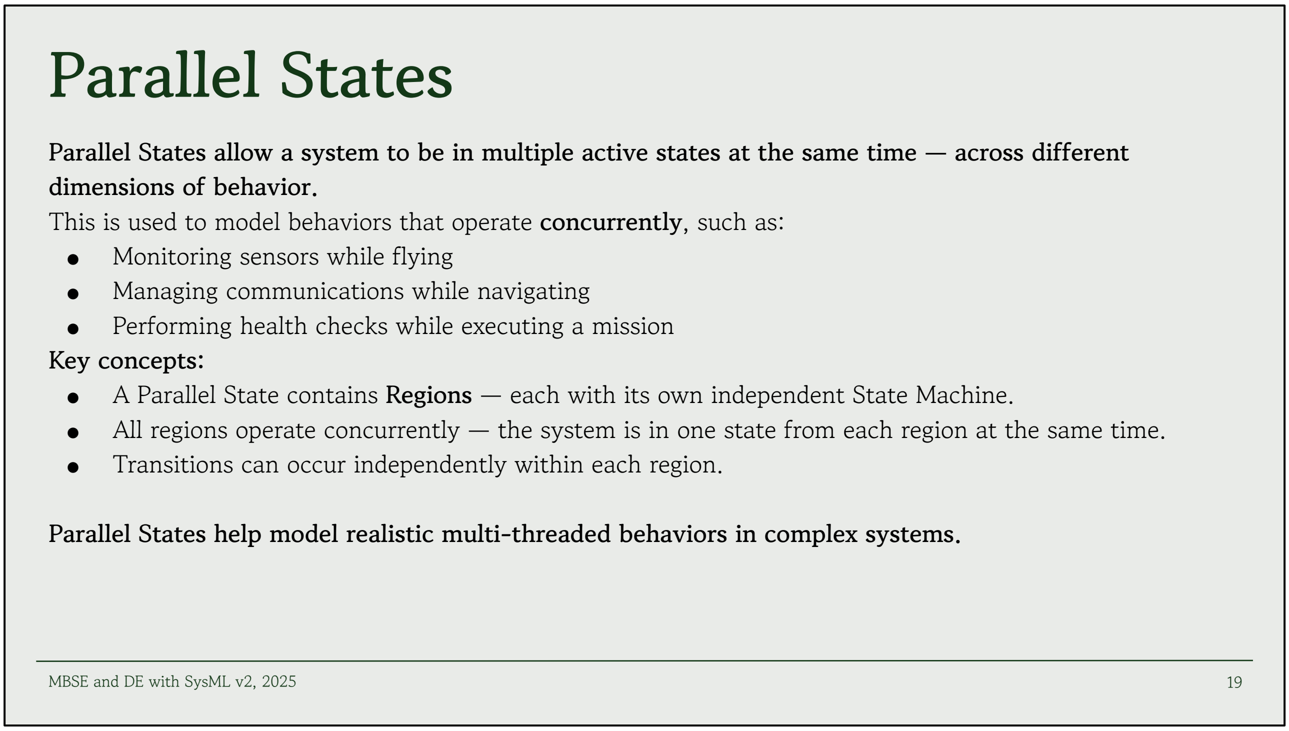
```
state compositeState1 {
   entry; then state1;
   state state1;
   transition
      first state1
      accept trigger1
      if guard1
      do action1
      then state2;
   state state2;
   then done;
}
```

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

Modeling Behavior with States
In real systems, **hierarchical behavior** is very common.
For example, when the Drone is in *Cruise* state, it may be:

- **Navigating**
- **Monitoring sensors**
- **Handling small adjustments**

This is where **Composite States** are powerful:
👉 A Composite State contains its own **internal State Machine**.
👉 This allows us to model **nested behavior** — cleanly and modularly.
When the system enters *Cruise*, the **internal states** start operating.
Benefits of Composite States:
✅ Manage complexity — break big State Machines into **modular pieces**.
✅ Reflect real-world systems — which often operate in **modes with internal behavior**.
✅ Support **reusability** — Composite States can be defined once and used in multiple contexts.
We will see how this applies to our **Drone**: *Cruise* may be a Composite State with *Navigate*, *Avoid Obstacles*, *Update Path*, etc.



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.

Modeling Behavior with States

In many real systems, different aspects of behavior operate **in parallel** — they are not strictly sequential.

For example, an **Autonomous Drone**: While in *Cruise*, it must:

- **Monitor sensors** continuously
- **Maintain communications** with ground control
- **Navigate** the flight path

These behaviors happen **at the same time** — not one after another.

Parallel States enable us to model this.

👉 We define **Regions** — each region has its own **independent State Machine**.

👉 The system can be in **one state per region at the same time**.

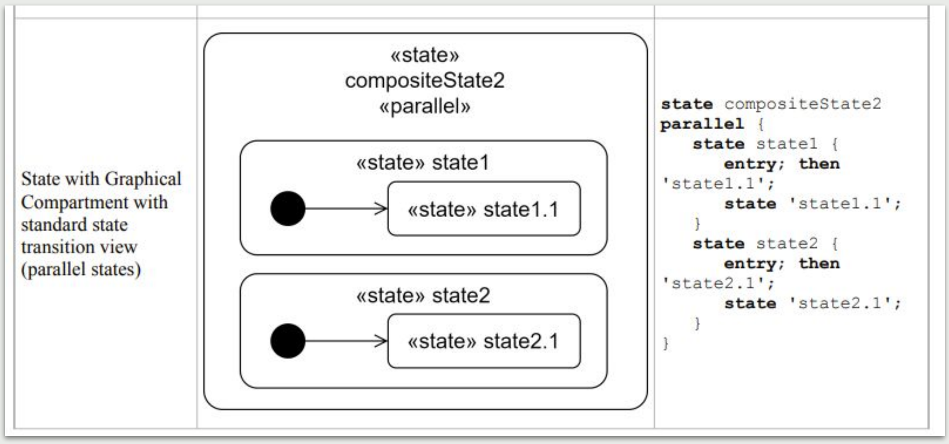
Benefits:

✅ Models realistic **concurrent behavior**.

✅ Matches how real cyber-physical systems operate.

✅ Keeps State Models **modular and scalable**.

Parallel States


«state»
compositeState2
«parallel»
«state» state1
«state» state1.1
«state» state2
«state» state2.1

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

Modeling Behavior with States

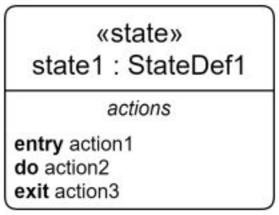
Actions Within States (Entry, Do, Exit)

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

Modeling Behavior with States

It is important to understand that **States themselves** can have behavior — not just the Transitions between them.

In SysML v2, a **State usage** can include:

- **Entry Action** — what happens when the State is entered
- **Do Action** — what happens *while the State is active*
- **Exit Action** — what happens when leaving the State

Typical usage in real systems:

Entry → Initialize navigation systems

Do → Continuously maintain flight path

Exit → Shutdown navigation, prepare for landing

Why this matters:

✅ Helps model **continuous behavior** that happens inside States

✅ Matches real system behavior — e.g., *Monitoring sensors during Cruise*

✅ Supports **clean separation of concerns**:

- *Transitions* handle *when* we move between States
- *State subactions* handle *what happens inside each State*

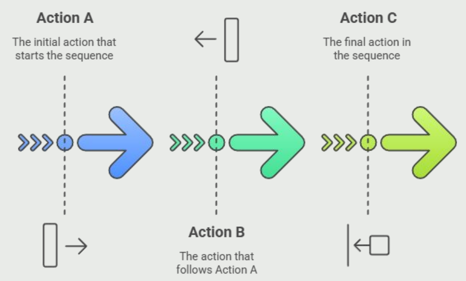
We will practice this — it is very useful in real-world modeling!

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.*


Action A
The initial action that starts the sequence
Action C
The final action in the sequence
Action B
The action that follows Action A

Control Flow vs Object Flow

Control Flow is the most basic type of flow in behavior models.

- It defines *ordering* — not data flow.
- A **Control Flow** tells us:
  - After actionA completes → start actionB.
  - After actionB completes → start actionC.

No data is passed — just **control tokens** (think of it as "permission to execute next").

Syntax example: actionA -> actionB -> actionC;

This simply says: First do actionA, then actionB, then actionC.

Typical uses:

- Sequencing steps in a process
- Modeling mission phases
- Defining ordered behaviors within States

In our Drone system: *Initialize Sensors* → *Start Navigation* → *Begin Cruise* → *Monitor Battery*.

Key point:

★ **Control Flow** models *when* things happen — but not *what data flows between them.*

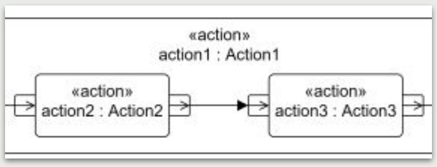
★ That is the role of **Object Flow** — which we will cover next.

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

Control Flow vs Object Flow

While **Control Flow** models *when* actions occur, **Object Flow** models *what data or material* flows between actions.

In other words — **Object Flow** models *information flow*.

This says: After collectSensorData executes, the resulting SensorData object is passed to analyzeData.

Typical uses:

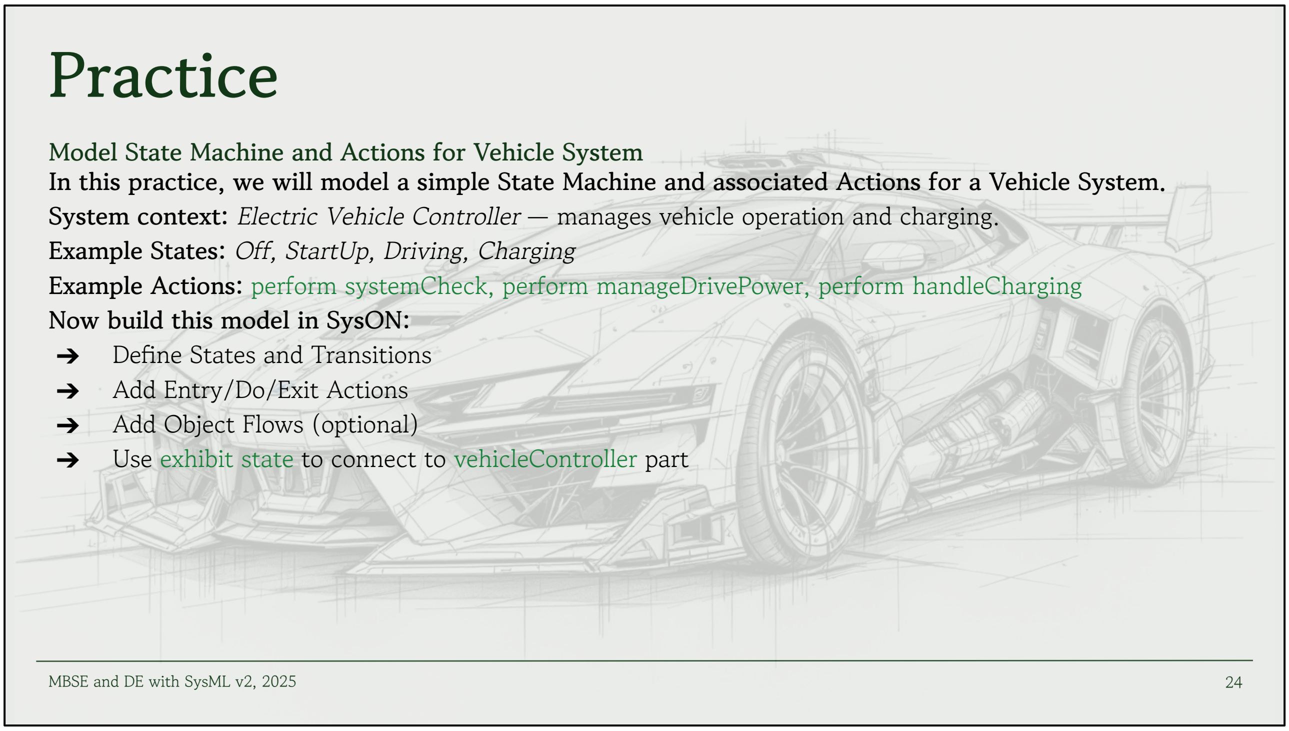
- Passing **Sensor Data** from one processing step to the next
- Passing **Control Commands** between subsystems
- Moving **Material** (like Payloads) between physical parts

Object Flow is critical when:

- The *data itself* drives behavior.
- The receiving action needs specific *input*.

Summary:

- ★ **Object Flow** models how *information moves*.
- ★ **Control Flow** models how *behavior progresses*.
- ★ Both are needed to model **realistic, complete behaviors**.



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:

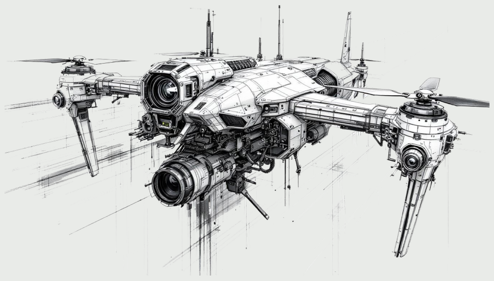
- ➔ Define States and Transitions
- ➔ Add Entry/Do/Exit Actions
- ➔ Add Object Flows (optional)
- ➔ Use exhibit state to connect to vehicleController part

Practice and Application: Vehicle System Behavior
Now let's do a simple but complete **Practice** — modeling behavior for a **Vehicle System**.

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.

Project Integration — Drone System
Your Homework this week is to apply these patterns to your **Drone System project**.

- Pick one **key part** — like the **Flight Controller** or **Payload System**.
- Define a **State Machine** for it.

Use:
✅ *State Definitions*
✅ *Transitions with triggers and guards*
✅ *Entry, Do, Exit Actions*
✅ *Exhibit State* to link to the part
Also model **Control Flow** and **Object Flow** inside your behavior where it makes sense.
Your goal: make your Drone model **come alive** — not just structure — but **dynamic behavior**.

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!