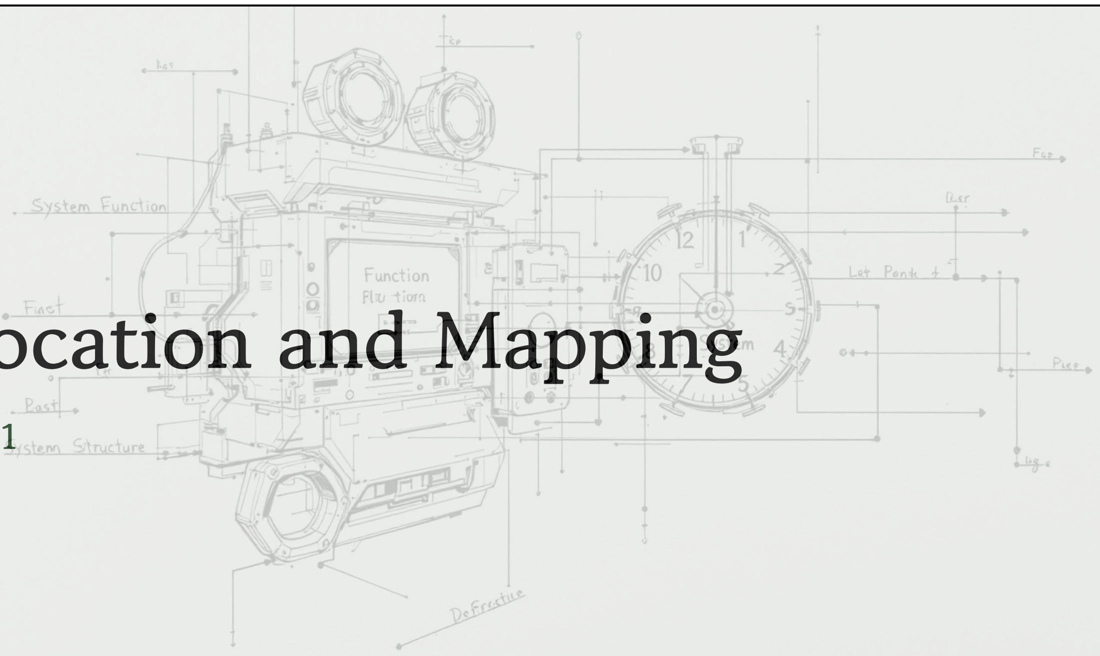
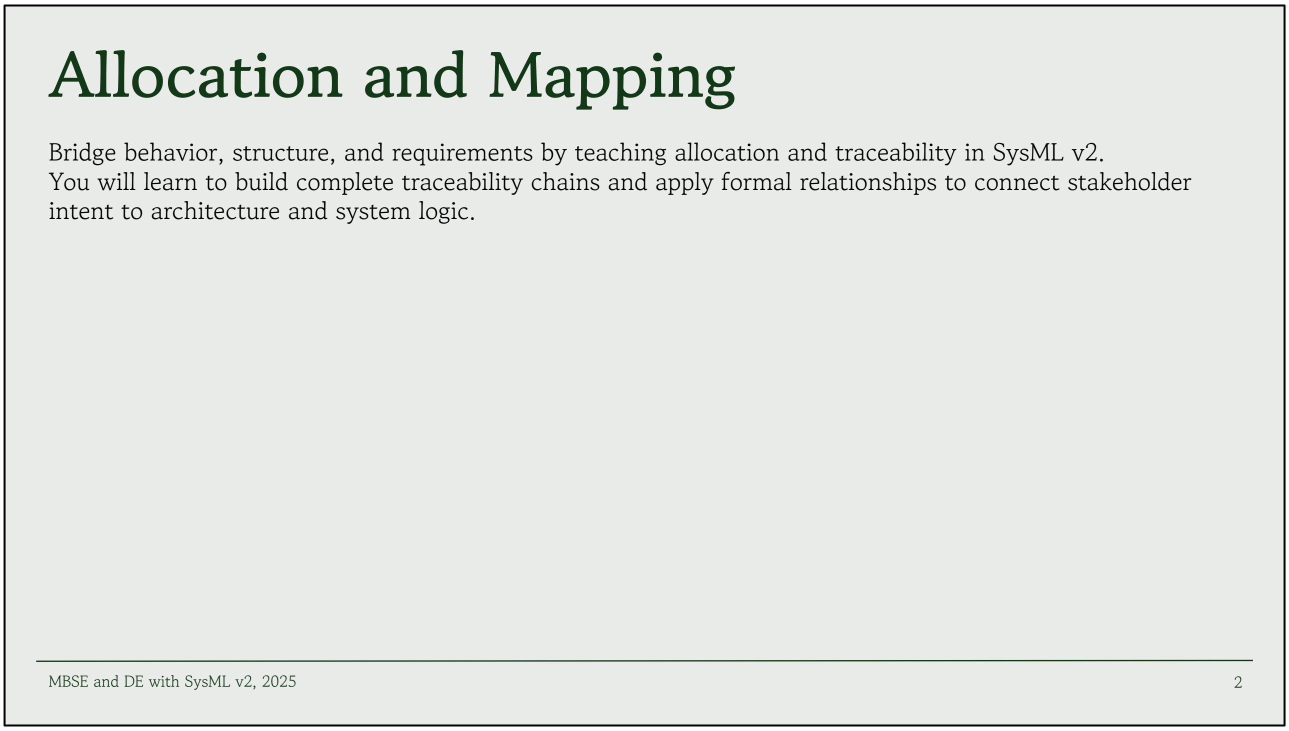


Allocation and Mapping

Week 11



Allocation and Mapping

Bridge behavior, structure, and requirements by teaching allocation and traceability in SysML v2.
You will learn to build complete traceability chains and apply formal relationships to connect stakeholder intent to architecture and system logic.

Welcome to **Week 11**. This week is an important milestone in our modeling journey.
Up to this point, we have modeled:

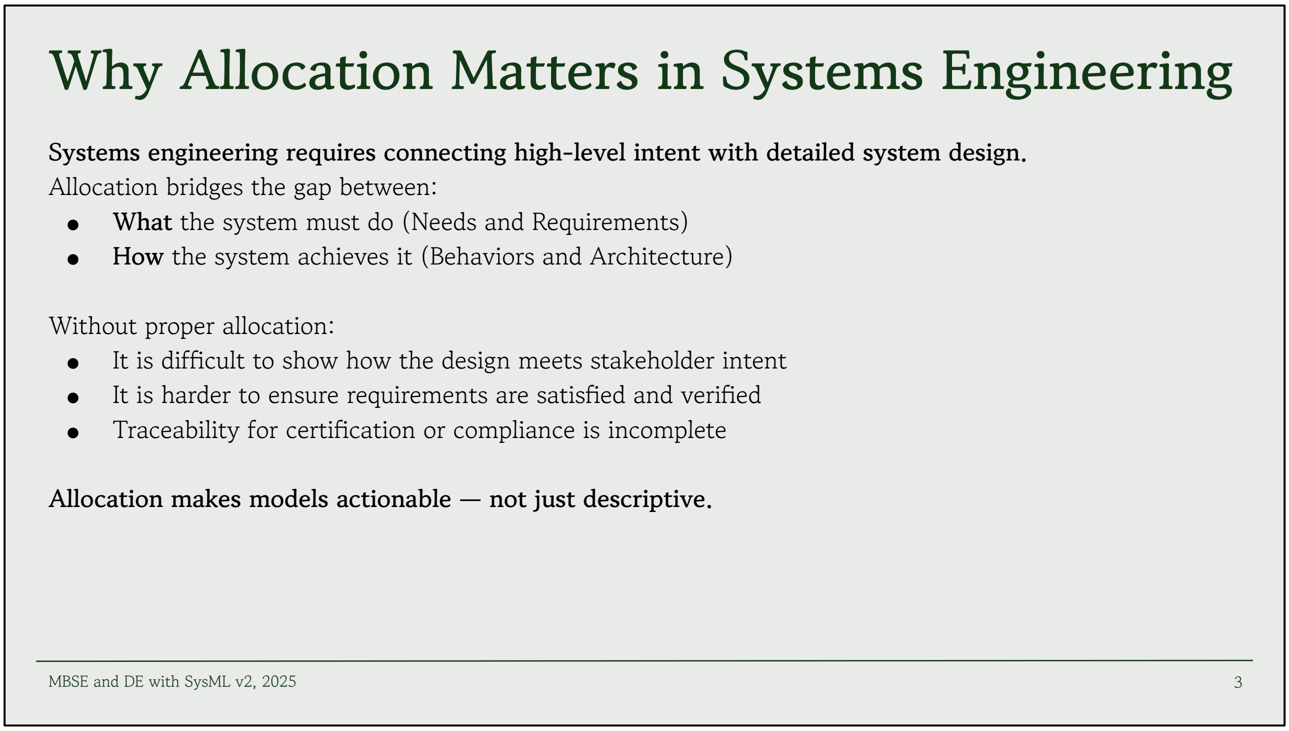
- **Structure** — the architecture of the system
- **Behavior** — what the system does
- **Requirements** — what the system must achieve
- **Constraints** — how the system is analyzed and validated

Now, we will learn how to formally **connect these elements** — to build full **traceability chains** through the model.
This is the core of **model-based systems engineering**:

- How do we trace from **Stakeholder Needs** to **Requirements**?
- How do we show which **Behaviors** satisfy which Requirements?
- How do we allocate those Behaviors to specific **Parts** of the system?
- How do we trace to **V&V** (Verification & Validation)?

This week, we will introduce the key **SysML v2 relationships** — refine, satisfy, verify — and learn how to use them properly.

By the end of this session, you will know how to build **complete, traceable models** — linking intent to architecture to system logic.



Why Allocation Matters in Systems Engineering

Systems engineering requires connecting high-level intent with detailed system design.
Allocation bridges the gap between:

- **What** the system must do (Needs and Requirements)
- **How** the system achieves it (Behaviors and Architecture)

Without proper allocation:

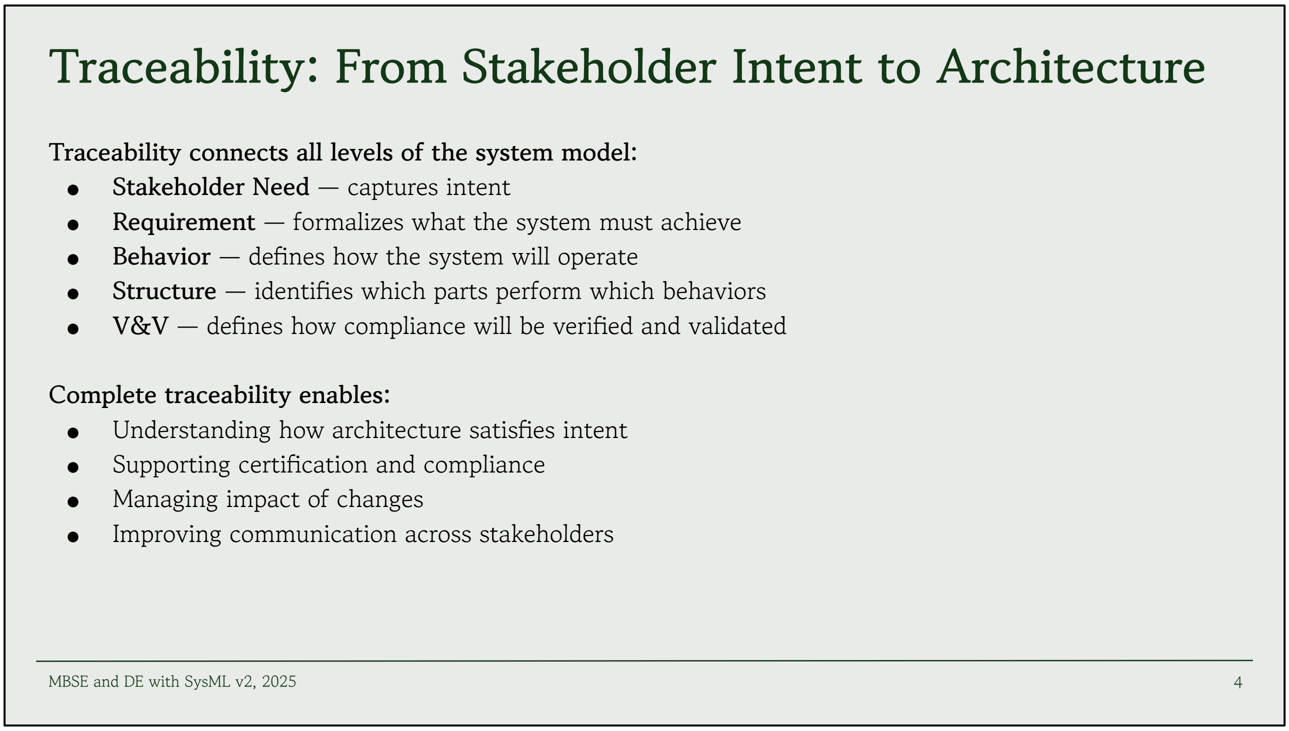
- It is difficult to show how the design meets stakeholder intent
- It is harder to ensure requirements are satisfied and verified
- Traceability for certification or compliance is incomplete

Allocation makes models actionable — not just descriptive.

Why Model Allocation and Traceability?
Why do we care about **allocation**?
In systems engineering, we begin with high-level **Stakeholder Needs** and **Requirements** — the *intent* of the system.
But ultimately, the system must be implemented in a **real architecture** — specific Parts, Behaviors, and Constraints.
Allocation is how we formally connect these two worlds.
Without proper allocation, the model remains **descriptive** — we can describe requirements and describe the architecture, but we cannot show **how the architecture satisfies the requirements**.
Allocation allows us to:

- Trace from **Needs** to **Requirements** to **Behaviors** to **Parts**
- Show **how requirements are implemented**
- Support **certification** and **compliance** by providing clear traceability
- Enable **verification and validation** by connecting Requirements to testable elements

This is why **Allocation matters** — it makes the model an **actionable engineering tool**, not just documentation.



Traceability: From Stakeholder Intent to Architecture

Traceability connects all levels of the system model:

- **Stakeholder Need** — captures intent
- **Requirement** — formalizes what the system must achieve
- **Behavior** — defines how the system will operate
- **Structure** — identifies which parts perform which behaviors
- **V&V** — defines how compliance will be verified and validated

Complete traceability enables:

- Understanding how architecture satisfies intent
- Supporting certification and compliance
- Managing impact of changes
- Improving communication across stakeholders

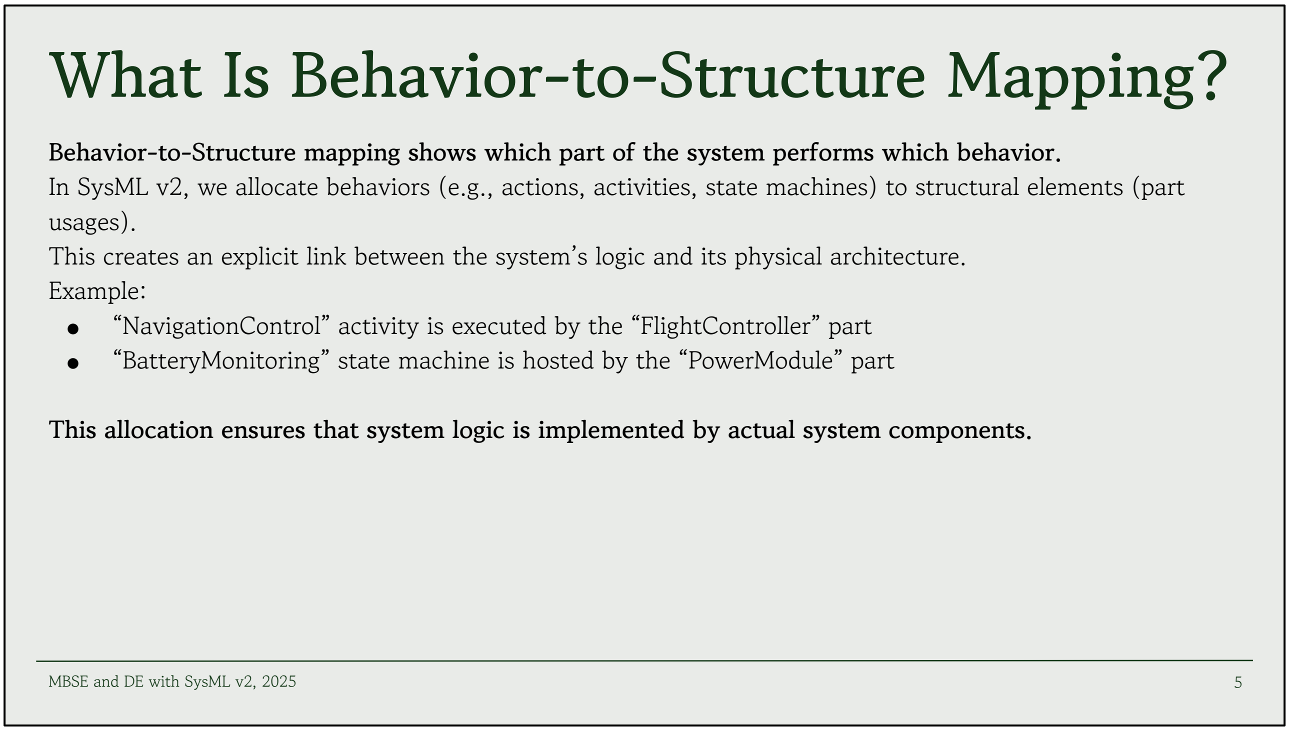
Why Model Allocation and Traceability?
Traceability is a foundational principle in model-based systems engineering.
A well-structured model provides a complete **traceability chain** — from **Stakeholder Intent** all the way down to **system implementation and verification**.
The chain typically looks like this:

- **Stakeholder Need** — what the customer or user wants
- **Requirement** — what the system must do to satisfy that need
- **Behavior** — how the system will operate to fulfill the requirement
- **Structure** — which **Parts** perform the required behavior
- **V&V** — how we will confirm that the requirement is met

When we maintain this traceability:

- We can clearly explain how the system satisfies the stakeholder's goals
- We can support **certification and compliance** processes
- We can manage **impact of changes** — if a requirement changes, we know what parts of the architecture are affected
- We improve communication — making it easier for engineers, managers, and customers to understand the system

This is why this week's focus is so important — learning to **build complete traceability chains** makes your model truly useful for engineering.



What Is Behavior-to-Structure Mapping?

Behavior-to-Structure mapping shows which part of the system performs which behavior.
In SysML v2, we allocate behaviors (e.g., actions, activities, state machines) to structural elements (part usages).
This creates an explicit link between the system's logic and its physical architecture.
Example:

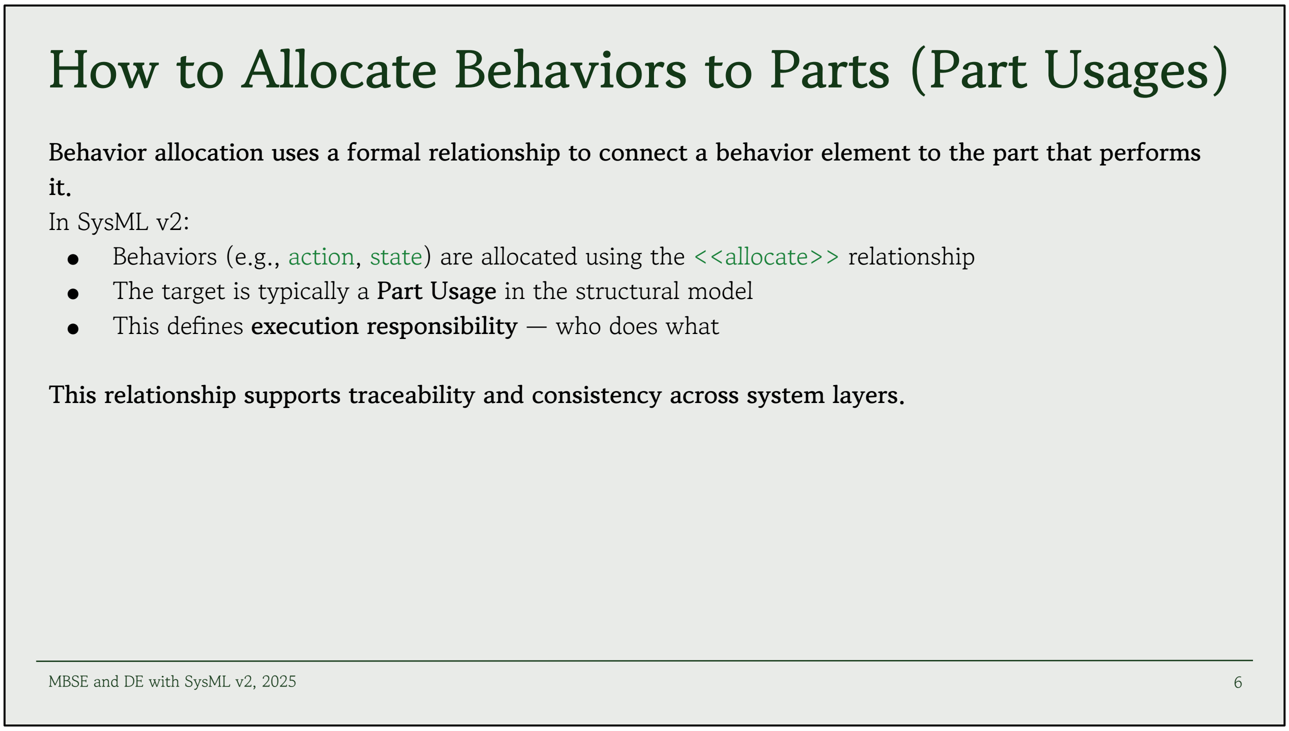
- "NavigationControl" activity is executed by the "FlightController" part
- "BatteryMonitoring" state machine is hosted by the "PowerModule" part

This allocation ensures that system logic is implemented by actual system components.

Behavior-to-Structure Mapping
Behavior-to-Structure mapping is one of the key techniques we use to turn behavior models into implementable architecture.
In SysML v2, **behaviors** can be defined independently — such as Actions, or State Machines.
But for the system to be real, these behaviors must be performed by something — some **Part** of the system.
Behavior-to-Structure mapping tells us: **Which part performs which behavior?**
This is called **allocation** — and it's modeled by linking a behavior to the **part usage** that is responsible for executing it.
For example:

- We might model a behavior called navigationControl as an action.
- Then we allocate it to the part usage flightController, meaning that this component is responsible for executing that behavior.

This mapping creates a **clear bridge** between the logic (what the system does) and the architecture (who does it).
Without this mapping, we cannot trace behavior to physical system implementation.
This step is essential for traceability, architecture review, and downstream deployment to software or hardware.



How to Allocate Behaviors to Parts (Part Usages)

Behavior allocation uses a formal relationship to connect a behavior element to the part that performs it.

In SysML v2:

- Behaviors (e.g., action, state) are allocated using the <<allocate>> relationship
- The target is typically a **Part Usage** in the structural model
- This defines **execution responsibility** — who does what

This relationship supports traceability and consistency across system layers.

Behavior-to-Structure Mapping

Now that we know what behavior-to-structure mapping means — let's see how it's done in SysML v2.

SysML v2 provides a formal modeling relationship called <<allocate>>.

This is how we express that a **behavior** — such as an action or state machine — is performed by a **part usage** in the system structure.

Here's the general idea:

- You define a behavior, such as an action called navigationLogic
- You identify a structural part, such as a part usage named flightController
- You then create an allocate relationship from the behavior to the part

This tells the model (and any downstream engineering tools):

The flightController is responsible for executing the NavigationLogic behavior.

This mapping allows us to:

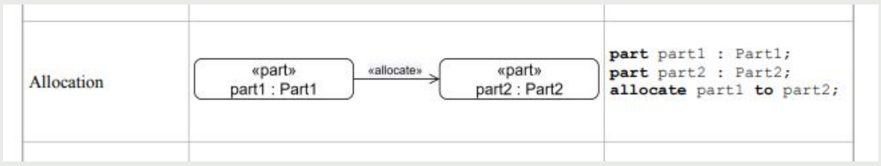
- Trace execution responsibilities clearly
- Check if all system functions have been allocated
- Support later tasks like software mapping, hardware partitioning, or testing

Allocation is not just documentation — it creates a formal, queryable, and analyzable connection in the model.

Good Practices for Behavior Allocation

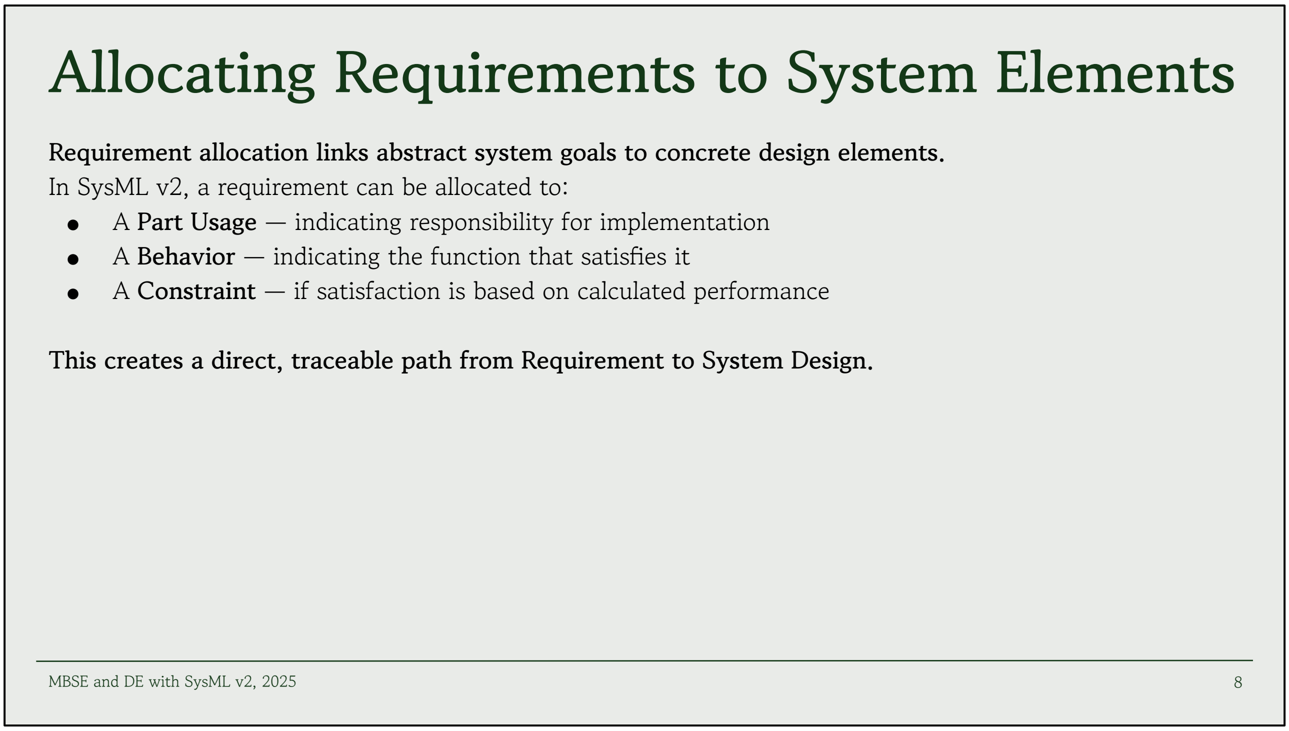
Effective allocation improves model clarity, traceability, and future implementation.
✔ Allocate every major behavior to a responsible part
✔ Keep allocation one-to-one where possible — avoid ambiguous mappings
✔ Use meaningful naming for behaviors and parts
✔ Group related behaviors under the same controller or subsystem
✔ Visualize allocations early to identify gaps or overloads

Allocation is not just traceability — it defines design intent.


Allocation
«part»
part1 : Part1
«allocate»
«part»
part2 : Part2

```
part part1 : Part1;
part part2 : Part2;
allocate part1 to part2;
```

Behavior-to-Structure Mapping
Let’s close this segment with some **best practices** for allocating behaviors to parts.
✔ **Allocate every major behavior** — don’t leave behaviors floating in the model. If a behavior exists, it should be allocated to something that performs it.
✔ Keep the allocation **unambiguous** — ideally, one behavior maps to one executing part. If multiple parts execute a shared behavior, be explicit about roles or use composite behaviors.
✔ Use **meaningful names** — a well-named behavior and a well-named part make the allocation readable and maintainable.
✔ Group related behaviors — for example, allocate all obstacle avoidance behaviors to the SensorController, or all flight logic to the FlightController. This helps reinforce modular design.
✔ Visualize the allocations — especially in early stages — to detect gaps (unallocated behaviors) or overloads (too many behaviors allocated to one part).
Most importantly, remember: **allocation expresses design decisions**.
It’s not just documentation — it shows **who does what** in the system.
That’s why it’s essential for implementation, simulation, testing, and review.



Allocating Requirements to System Elements

Requirement allocation links abstract system goals to concrete design elements.
In SysML v2, a requirement can be allocated to:

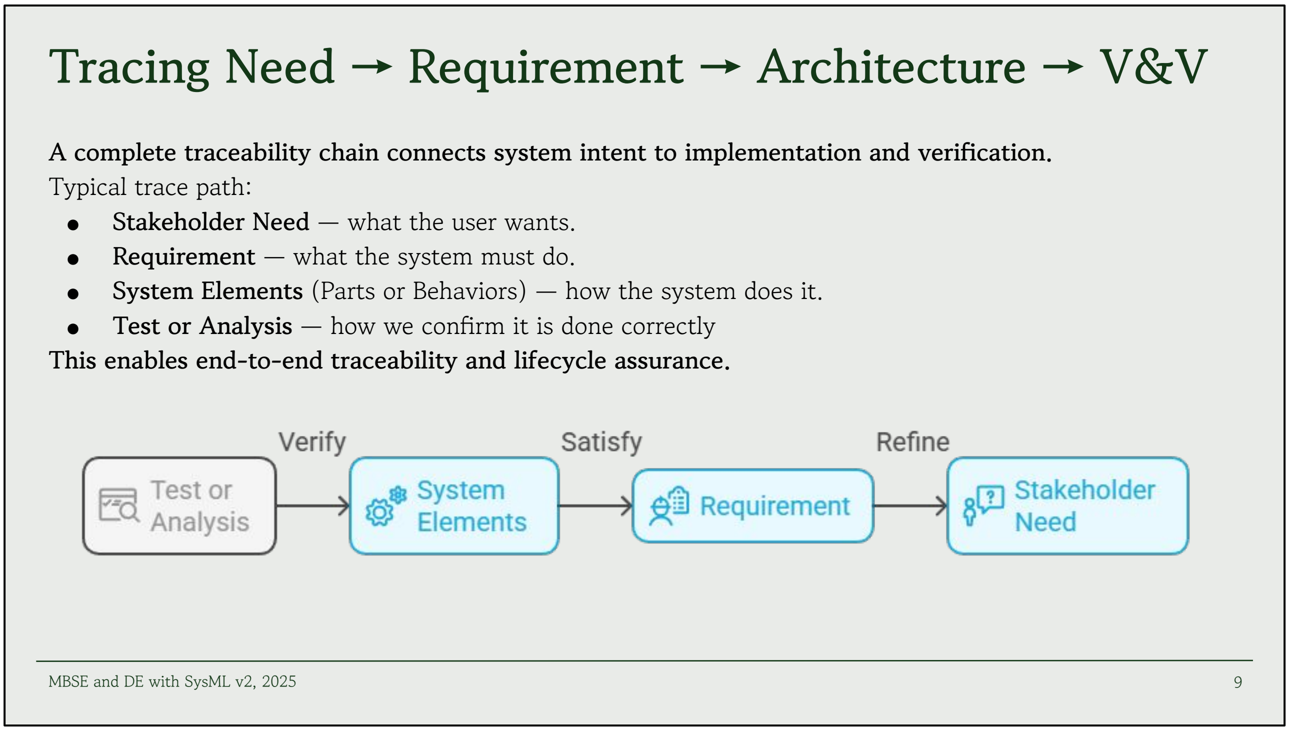
- A **Part Usage** — indicating responsibility for implementation
- A **Behavior** — indicating the function that satisfies it
- A **Constraint** — if satisfaction is based on calculated performance

This creates a direct, traceable path from Requirement to System Design.

Requirement Allocation and Traceability
Just like we allocate **behaviors** to structure, we also allocate **requirements**.
Requirement allocation is the process of linking an abstract requirement to something **concrete** in the system model — something that is responsible for satisfying it.
In SysML v2, a requirement can be allocated to:

- A **Part Usage** — for example, if the FlightController must support autonomous navigation
- A **Behavior** — if the system must execute a specific function to meet the requirement
- A **Constraint** — if the requirement is performance-based, like “must fly for 30 minutes”

This allocation provides a **clear ownership path**:
Who (or what) is responsible for fulfilling the requirement?
These relationships are not just traceable — they are **queryable**, **verifiable**, and can be automatically checked during model validation or simulation.
Once we allocate a requirement to system elements, we can begin to model how it will be **satisfied**, **verified**, and **refined**.

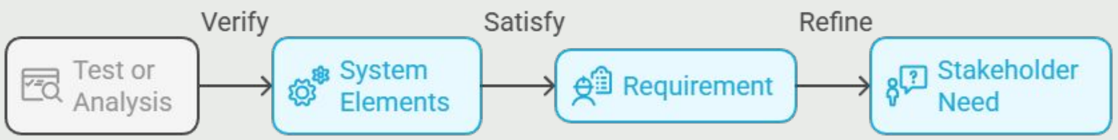


Tracing Need → Requirement → Architecture → V&V

A complete traceability chain connects system intent to implementation and verification.
Typical trace path:

- **Stakeholder Need** — what the user wants.
- **Requirement** — what the system must do.
- **System Elements** (Parts or Behaviors) — how the system does it.
- **Test or Analysis** — how we confirm it is done correctly

This enables end-to-end traceability and lifecycle assurance.



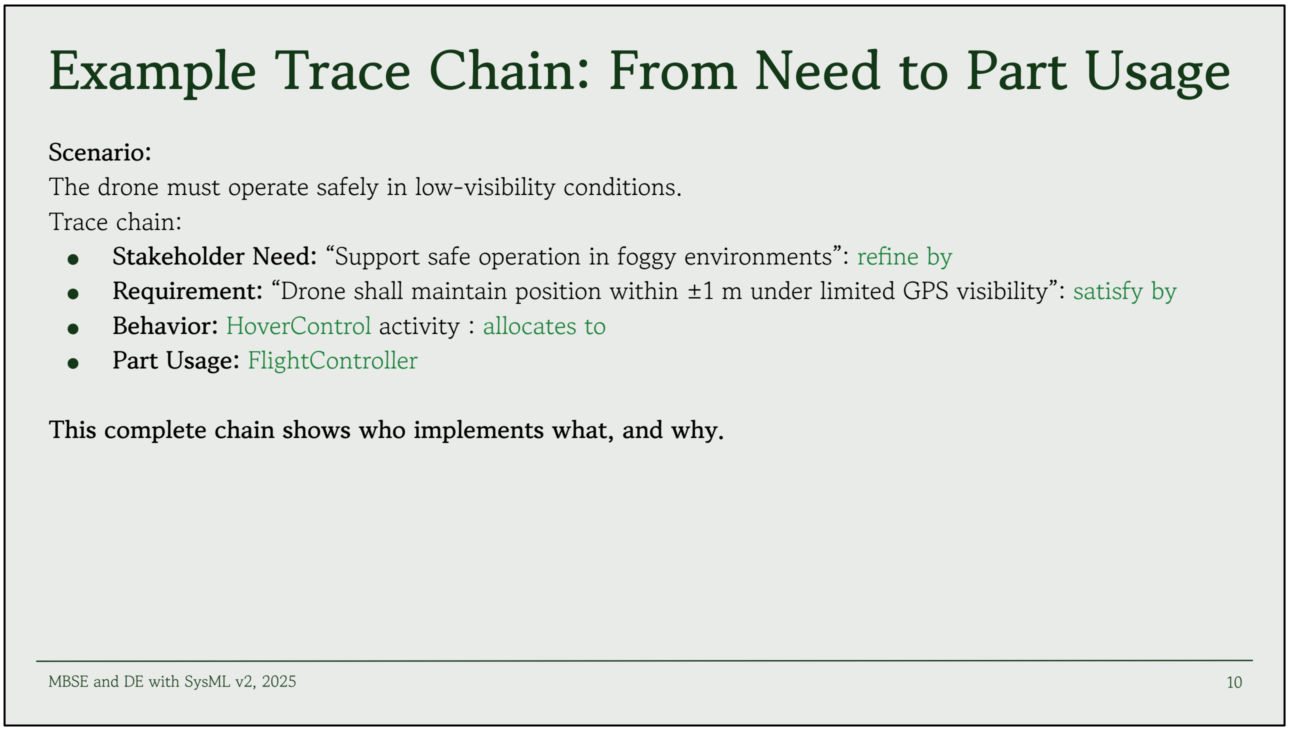
Requirement Allocation and Traceability
Now we're ready to look at the **full traceability chain** — from abstract intent to concrete realization and verification.
It typically follows this pattern:

1. Start with a **Stakeholder Need** — for example, "The drone should fly safely in wind conditions."
2. Create a **Requirement** that refines that need — for instance, "The drone must maintain heading within ±5° under 10 m/s wind."
3. Link the Need to the Requirement using the refine relationship.
4. Identify which **Parts** or **Behaviors** of the system satisfy that requirement — and use the satisfy relationship to connect them.
5. Link to the **Test Case** or **Analysis Model** that will confirm the requirement is met — using the verify relationship.

This chain is what enables traceability throughout the system lifecycle.
It allows us to answer critical questions:

- Where did this requirement come from?
- How is it implemented?
- How will we test it?
- What happens if it changes?

By modeling these relationships explicitly, we create a **complete verification path** — aligned with good systems engineering and certification practices.



Example Trace Chain: From Need to Part Usage

Scenario:
The drone must operate safely in low-visibility conditions.
Trace chain:

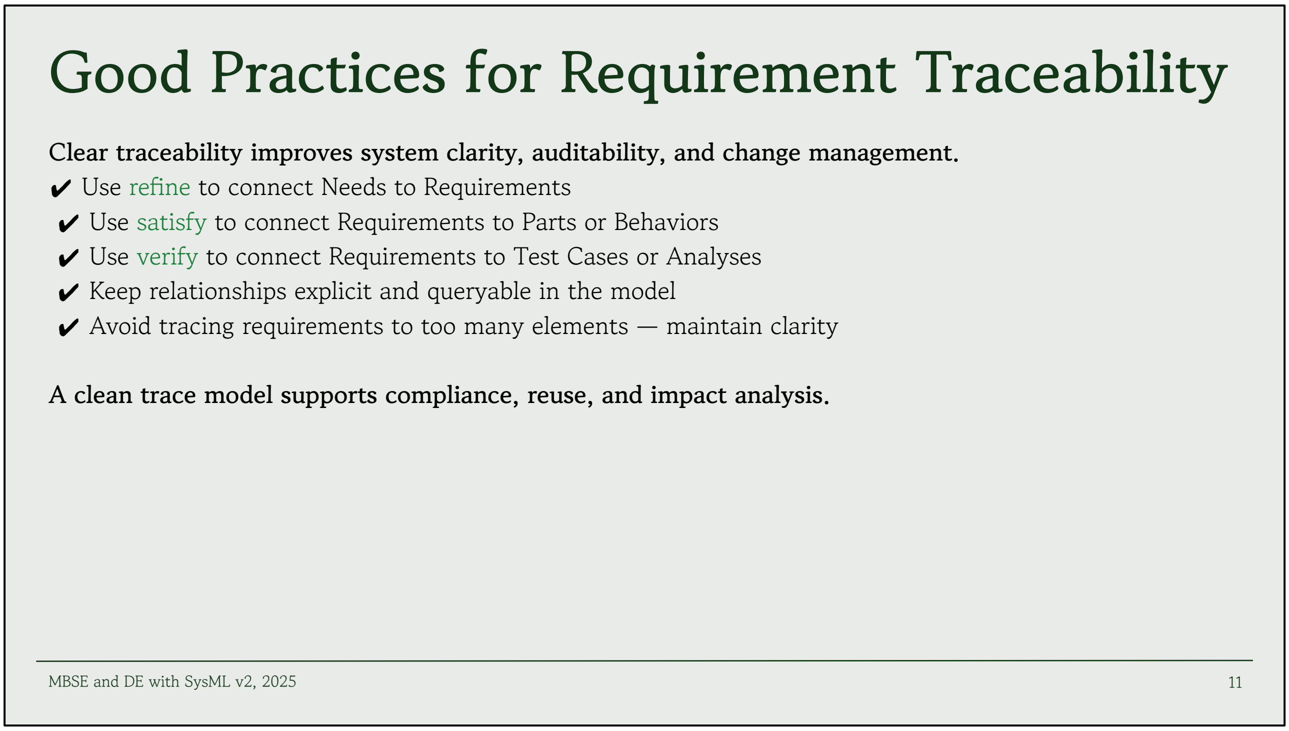
- **Stakeholder Need:** "Support safe operation in foggy environments": refine by
- **Requirement:** "Drone shall maintain position within ±1 m under limited GPS visibility": satisfy by
- **Behavior:** HoverControl activity : allocates to
- **Part Usage:** FlightController

This complete chain shows who implements what, and why.

Requirement Allocation and Traceability
Let's make the traceability chain more concrete with an example.
Imagine a **Stakeholder Need**:
"The drone should support safe operation in foggy environments."
We refine this into a **Requirement** that is more specific and measurable:
"The drone shall maintain position within ±1 meter under limited GPS visibility."
This requirement is satisfied by a specific **Behavior** in the system: An action called hoverControl that uses internal sensors to maintain position when GPS is weak.
That behavior is then **allocated** to a structural element: The flightController part — which actually performs the hover stabilization logic.
Now we have a full trace chain: Need → Requirement → Behavior → Part Usage
This traceability ensures that we can:

- Justify the architecture from the user's perspective
- Support design reviews and certification
- Identify impact if the requirement changes
- Provide clear ownership and accountability across system elements

This is the kind of trace you'll build in your own models.



Good Practices for Requirement Traceability

Clear traceability improves system clarity, auditability, and change management.
✔ Use refine to connect Needs to Requirements
✔ Use satisfy to connect Requirements to Parts or Behaviors
✔ Use verify to connect Requirements to Test Cases or Analyses
✔ Keep relationships explicit and queryable in the model
✔ Avoid tracing requirements to too many elements — maintain clarity

A clean trace model supports compliance, reuse, and impact analysis.

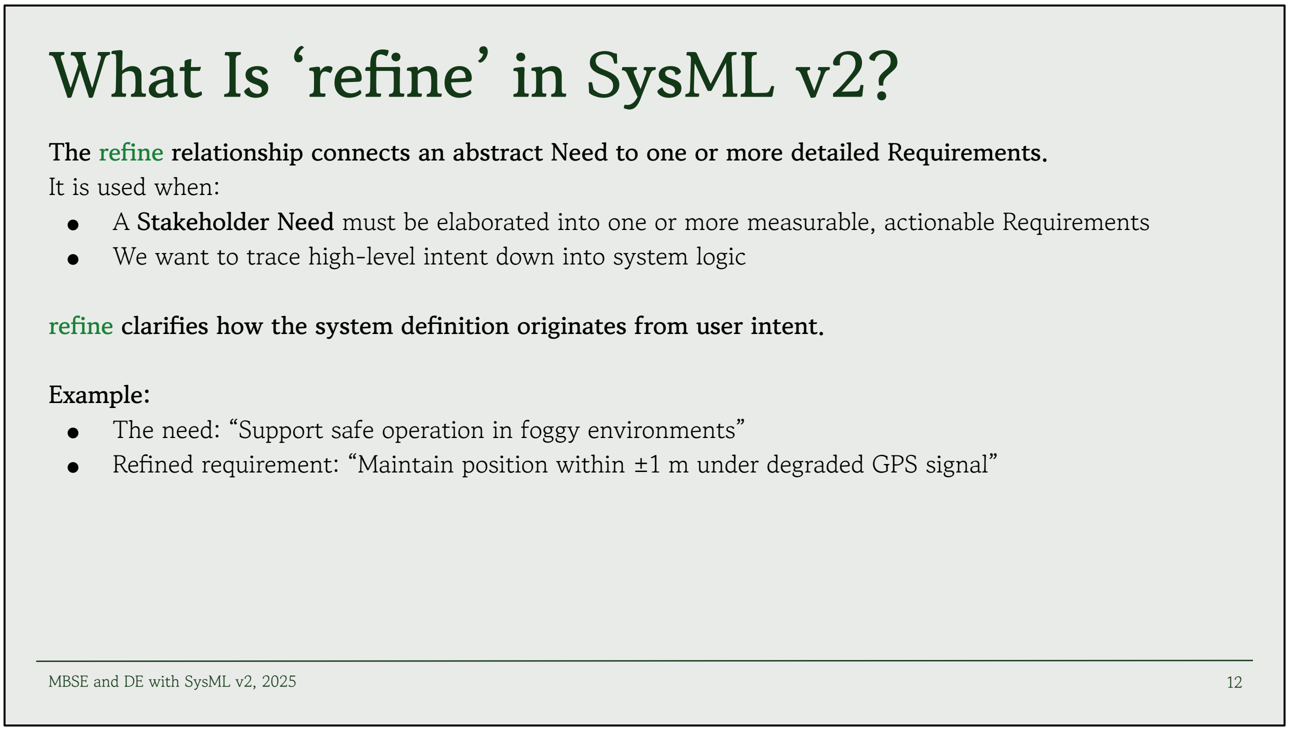
Requirement Allocation and Traceability
To make your models strong and maintainable, it's important to follow **good practices** for requirement traceability.
✔ First, always use the proper **SysML v2 relationships**:

- refine to go from **Stakeholder Need ➝ Requirement**
- satisfy to go from **Requirement ➝ Part or Behavior**
- verify to go from **Requirement ➝ Test or Analysis**

✔ Make sure each relationship is **explicit in the model** — not just in your head or on a whiteboard. Use trace links that are visible and navigable in SysON.
✔ Avoid tracing a single requirement to too many elements — that often means the requirement is too vague or the allocation is too scattered. Try to maintain clear accountability.
✔ When you trace well, you gain several benefits:

- You can clearly show how the system meets stakeholder goals
- You can support **compliance and audits**
- You can easily perform **impact analysis** when requirements change
- You can even reuse requirement structures in future projects

Traceability is not overhead — it's part of making the system **understandable, defensible, and future-proof**.



What Is ‘refine’ in SysML v2?

The refine relationship connects an abstract Need to one or more detailed Requirements.
It is used when:

- A **Stakeholder Need** must be elaborated into one or more measurable, actionable Requirements
- We want to trace high-level intent down into system logic

refine clarifies how the system definition originates from user intent.

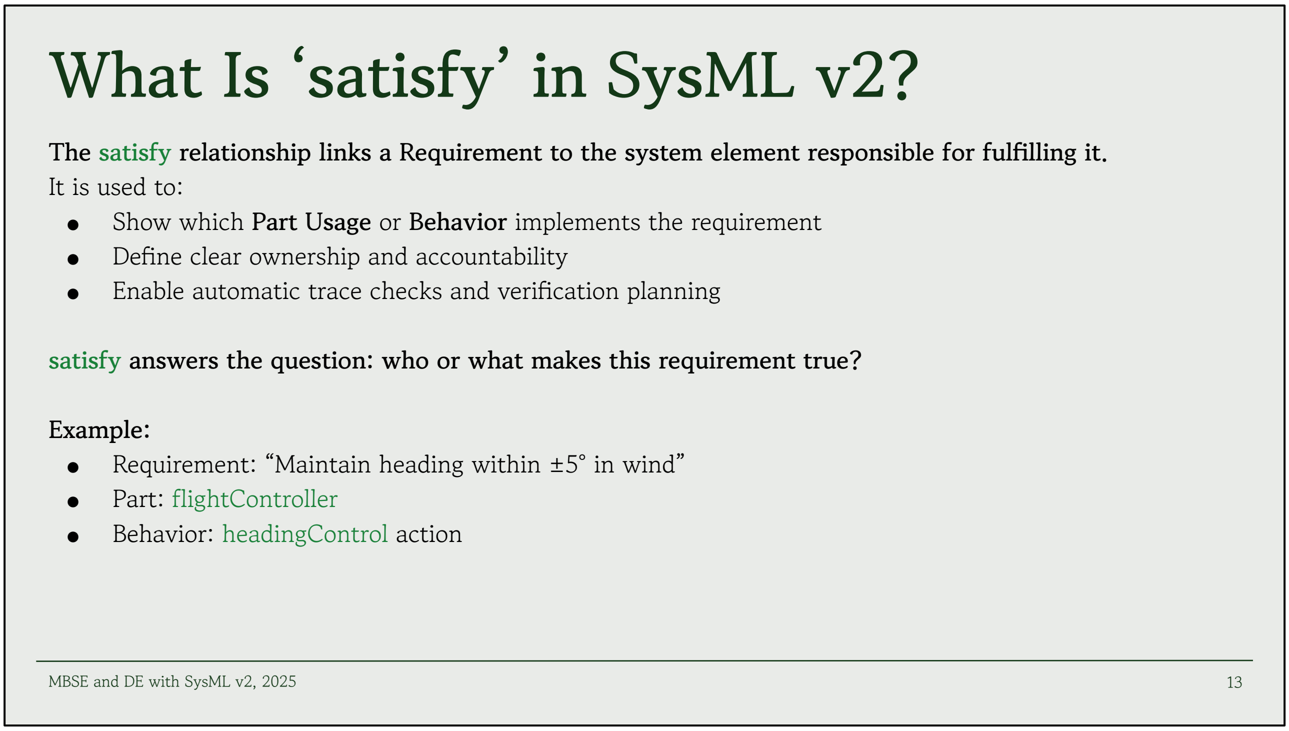
Example:

- The need: “Support safe operation in foggy environments”
- Refined requirement: “Maintain position within ±1 m under degraded GPS signal”

SysML v2 Relationships: refine, satisfy, verify
Let’s begin with the refine relationship — the first formal traceability link in SysML v2.
When we receive Stakeholder Needs — such as "The drone must be safe in fog" — they are often vague or abstract.
Before we can implement or test anything, we need to turn that need into something concrete.
That’s where refine comes in. The refine relationship links a **Need** to one or more **Requirements** that express it in a more precise, measurable way.
For example:

- The need: “Support safe operation in foggy environments”
- Refined requirement: “Maintain position within ±1 m under degraded GPS signal”

refine helps create a **logical decomposition of intent** — it tells the model how requirements came from stakeholder input.
It also improves **traceability** and helps us manage change.
If the need changes — we know which requirements are affected.
If a requirement is deleted — we can ask, “Did we lose traceability to a need?”
This is the **first step in building a traceability chain** — from user goals to system design.



What Is ‘satisfy’ in SysML v2?

The satisfy relationship links a Requirement to the system element responsible for fulfilling it.
It is used to:

- Show which **Part Usage** or **Behavior** implements the requirement
- Define clear ownership and accountability
- Enable automatic trace checks and verification planning

satisfy answers the question: who or what makes this requirement true?

Example:

- Requirement: “Maintain heading within ±5° in wind”
- Part: flightController
- Behavior: headingControl action

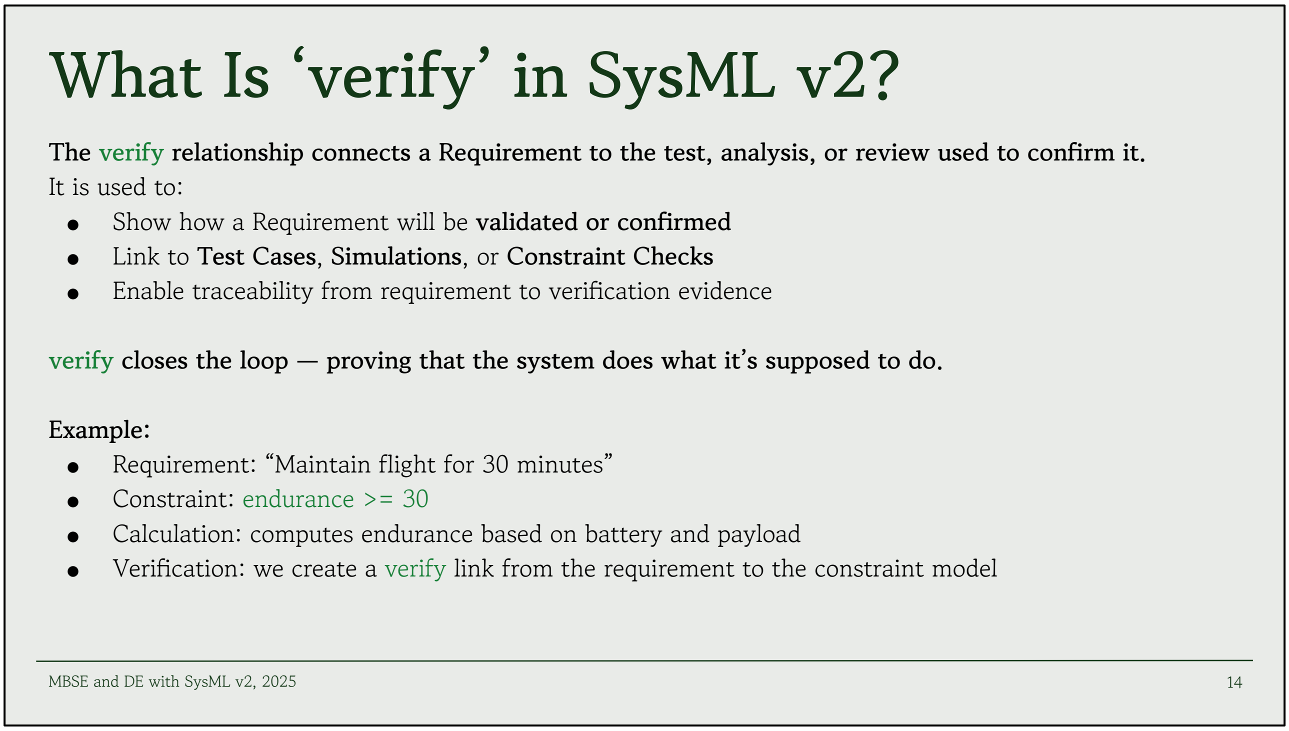
SysML v2 Relationships: refine, satisfy, verify
The satisfy relationship tells us **how a requirement is implemented**.
Once we define a Requirement, we need to know what part of the system is **responsible** for fulfilling it.
That’s what satisfy models: It links the **Requirement** to the **Part Usage** (structural) or **Behavior** (functional) that fulfills it.
For example:

- Requirement: “Maintain heading within ±5° in wind”
- Part: flightController
- Behavior: HeadingControl action

Both flightController and HeadingControl may be connected to the Requirement using satisfy.
This relationship has multiple benefits:

- It gives **clear accountability** — who is responsible
- It supports **change impact analysis** — if we modify a requirement, we can instantly see what parts of the system are affected
- It supports **completeness checks** — we can query all unsatisfied requirements

In a well-built model, every Requirement should have a corresponding satisfy link — otherwise we don’t know how it’s implemented.



What Is ‘verify’ in SysML v2?

The verify relationship connects a Requirement to the test, analysis, or review used to confirm it.
It is used to:

- Show how a Requirement will be **validated or confirmed**
- Link to **Test Cases**, **Simulations**, or **Constraint Checks**
- Enable traceability from requirement to verification evidence

verify closes the loop — proving that the system does what it’s supposed to do.

Example:

- Requirement: “Maintain flight for 30 minutes”
- Constraint: endurance >= 30
- Calculation: computes endurance based on battery and payload
- Verification: we create a verify link from the requirement to the constraint model

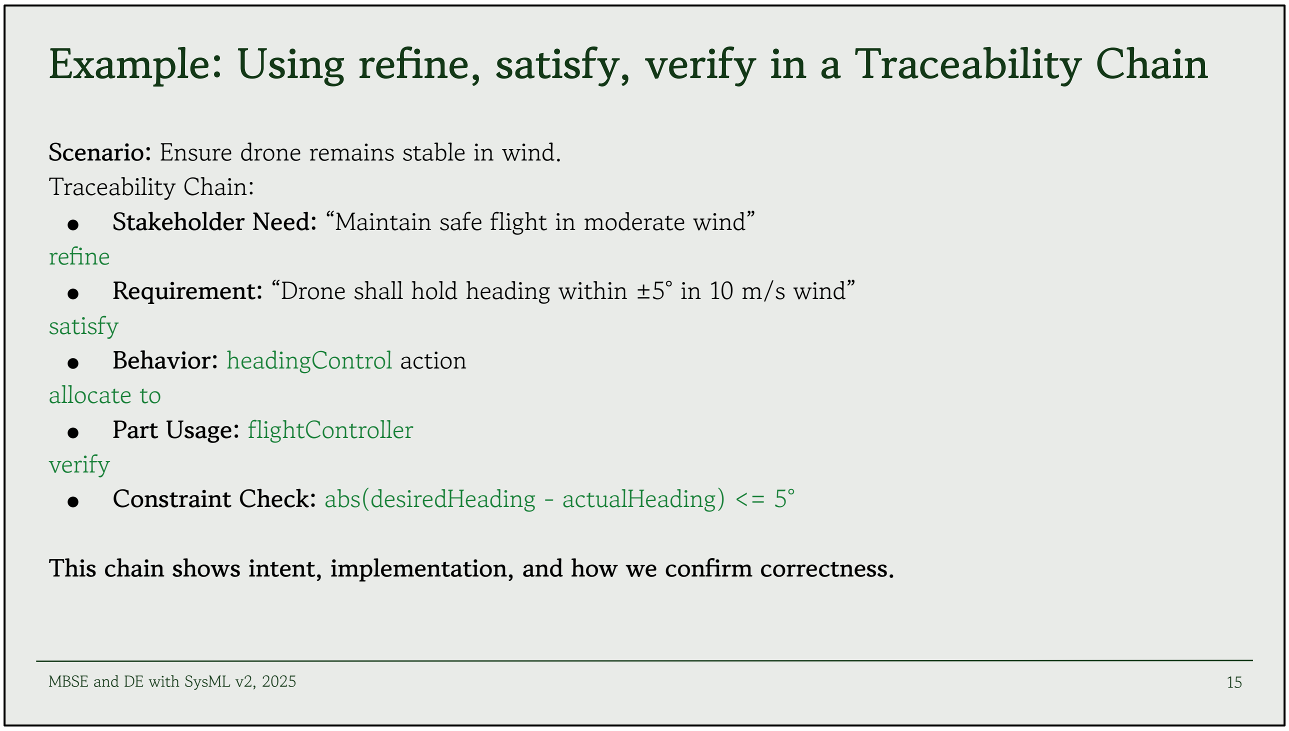
SysML v2 Relationships: refine, satisfy, verify
The final relationship in the core traceability trio is verify.
Where refine and satisfy tell us what the requirement means and how it is implemented, verify tells us **how we will prove it**.
Verification can take many forms in SysML v2:

- A **test case** executed on the actual system
- A **simulation** model result
- A **constraint** or **calculation** that confirms system performance

For example:

- Requirement: “Maintain flight for 30 minutes”
- Constraint: endurance >= 30
- Calculation: computes endurance based on battery and payload
- Verification: we create a verify link from the requirement to the constraint model

The verify relationship ensures that every requirement has an associated **evidence path** — a way to confirm it has been met.
It also supports automated **completeness checks** — so we can detect unverified requirements early.
Ultimately, verify is what gives our model **engineering credibility** — it's not just a diagram, it's a verifiable system description.



Example: Using refine, satisfy, verify in a Traceability Chain

Scenario: Ensure drone remains stable in wind.
Traceability Chain:

- **Stakeholder Need:** “Maintain safe flight in moderate wind”

refine

- **Requirement:** “Drone shall hold heading within ±5° in 10 m/s wind”

satisfy

- **Behavior:** headingControl action

allocate to

- **Part Usage:** flightController

verify

- **Constraint Check:** abs(desiredHeading - actualHeading) <= 5°

This chain shows intent, implementation, and how we confirm correctness.

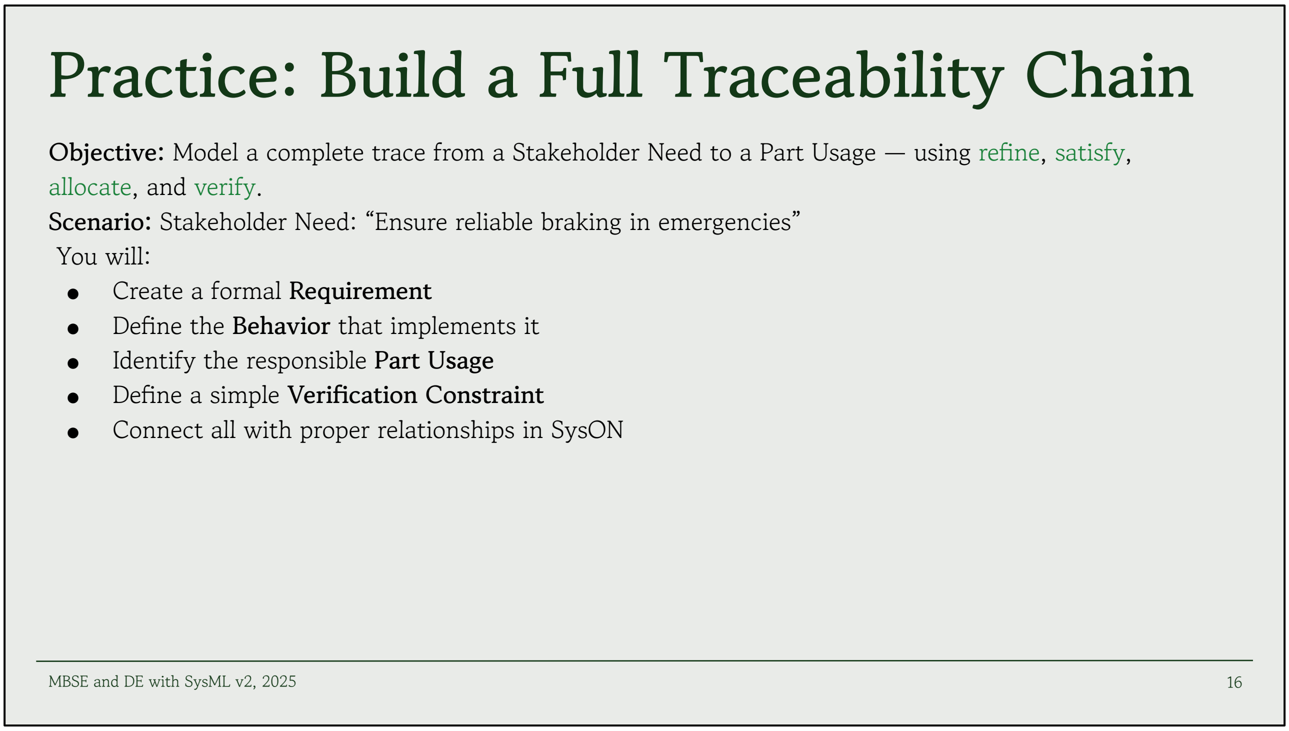
SysML v2 Relationships: refine, satisfy, verify
Let’s walk through a full example using all three relationships: refine, satisfy, and verify.
Start with a **Stakeholder Need**:
“The drone should maintain safe flight in moderate wind.”
That’s refined into a **Requirement**:
“Drone shall hold heading within ±5° in 10 m/s wind.”
We now have a measurable system objective.

Next, we define a **Behavior** — headingControl — that ensures this behavior is executed correctly.
We **allocate** that behavior to a **Part Usage** — the flightController.
Then we use satisfy to formally state that the Flight Controller implements this Requirement.
Finally, we define a **Constraint** to verify the heading error is within the limit:
abs(desiredHeading - actualHeading) <= 5°

We create a verify relationship from the Requirement to this Constraint.
This complete traceability chain tells us:

- Where the requirement came from
- What it means
- Who implements it
- And how we know it’s correct

This is the kind of trace you will build in your projects — and it’s what makes SysML models **auditable, testable, and engineerable**.



Practice: Build a Full Traceability Chain

Objective: Model a complete trace from a Stakeholder Need to a Part Usage — using refine, satisfy, allocate, and verify.
Scenario: Stakeholder Need: “Ensure reliable braking in emergencies”
You will:

- Create a formal **Requirement**
- Define the **Behavior** that implements it
- Identify the responsible **Part Usage**
- Define a simple **Verification Constraint**
- Connect all with proper relationships in SysON

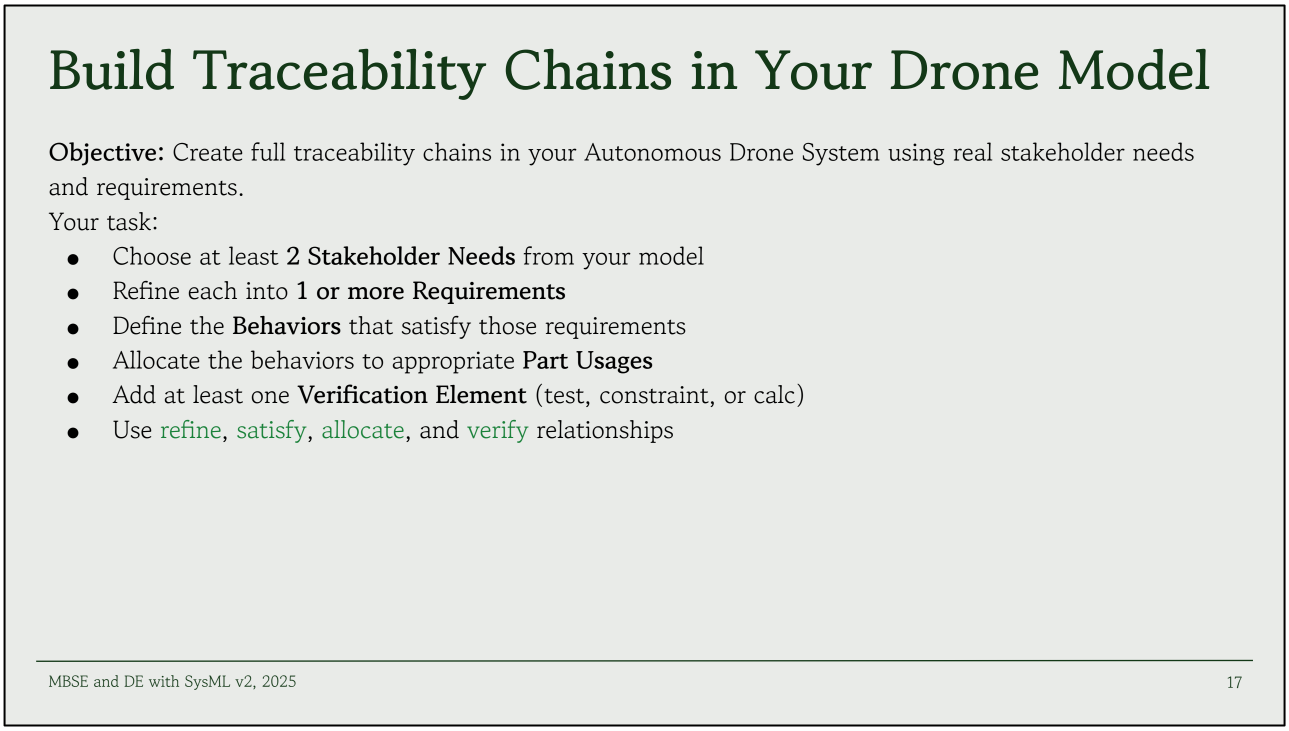
Traceability View and Navigation in SysON
In this exercise, you will build a **complete traceability chain** — hands-on in SysON.
We start from a **Stakeholder Need**:
“Ensure reliable braking in emergencies.”
You will now:

1. Create a formal Requirement that refines this need — something measurable like “Braking system shall decelerate vehicle from 60 to 0 km/h within 3 seconds.”
2. Define a Behavior (e.g., an action called emergencyBrakeLogic)
3. Allocate that behavior to a specific Part Usage — maybe brakeController
4. Write a simple verification Constraint — something like deceleration >= 5 m/s^2
5. Link the Requirement to:
    - The Behavior with satisfy
    - The Part with allocate
    - The Constraint with verify

In the end, you will have:

- One connected Need
- One Requirement
- One Behavior
- One Part
- One Verification element

And all formally linked in the model.
This mirrors what you’ll do in your **drone projects** — so take your time and ensure the links are clean, complete, and queryable.



Build Traceability Chains in Your Drone Model

Objective: Create full traceability chains in your Autonomous Drone System using real stakeholder needs and requirements.

Your task:

- Choose at least **2 Stakeholder Needs** from your model
- Refine each into **1 or more Requirements**
- Define the **Behaviors** that satisfy those requirements
- Allocate the behaviors to appropriate **Part Usages**
- Add at least one **Verification Element** (test, constraint, or calc)
- Use refine, satisfy, allocate, and verify relationships

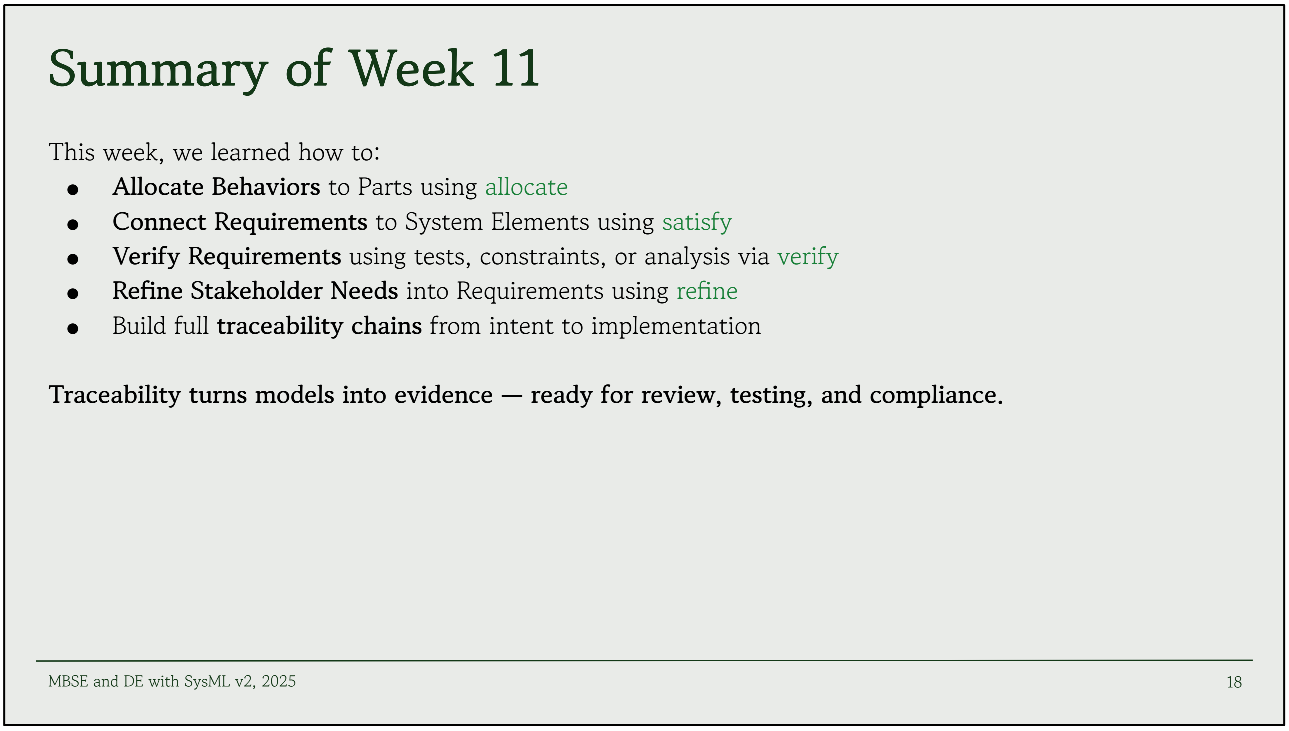
For this week's homework, you'll apply what we've practiced — directly in your own drone system model.

Use the formal relationships we studied:

- refine (Need → Requirement)
- satisfy (Requirement → Behavior or Part)
- allocate (Behavior → Part)
- verify (Requirement → Verification)

Try to create a **clear, traceable path** from stakeholder intent to implementation and verification.

This assignment will help you lock in the principles of **model accountability and verification** — and it builds a solid foundation for the next phase of system analysis.



Summary of Week 11

This week, we learned how to:

- **Allocate Behaviors** to Parts using allocate
- **Connect Requirements** to System Elements using satisfy
- **Verify Requirements** using tests, constraints, or analysis via verify
- **Refine Stakeholder Needs** into Requirements using refine
- Build full **traceability chains** from intent to implementation

Traceability turns models into evidence — ready for review, testing, and compliance.

Let’s summarize what we’ve covered this week.
We started by learning how to **allocate behaviors** to the structural parts that perform them — this brings logic and architecture together.
Then we studied how to **allocate requirements** to system elements, creating a link between what the system must do and who is responsible for doing it.
We looked at the core **traceability relationships** in SysML v2:

- refine — to link Needs to Requirements
- satisfy — to link Requirements to Behaviors or Parts
- verify — to link Requirements to their validation and verification evidence

These relationships make our models **structured, traceable, and defensible**.
We also explored how to **navigate and visualize** these trace links in SysON — allowing full traceability from the top-level stakeholder need all the way down to the part and test that fulfill it.
Finally, we applied this knowledge in a **hands-on exercise** — building your own traceability chain.
In your course projects, this traceability will help you connect your model to real system intent, and prepare your designs for deeper analysis and validation in the weeks ahead.

QUESTION!