Specifications & Safety

CSCI 420-02 Robotics

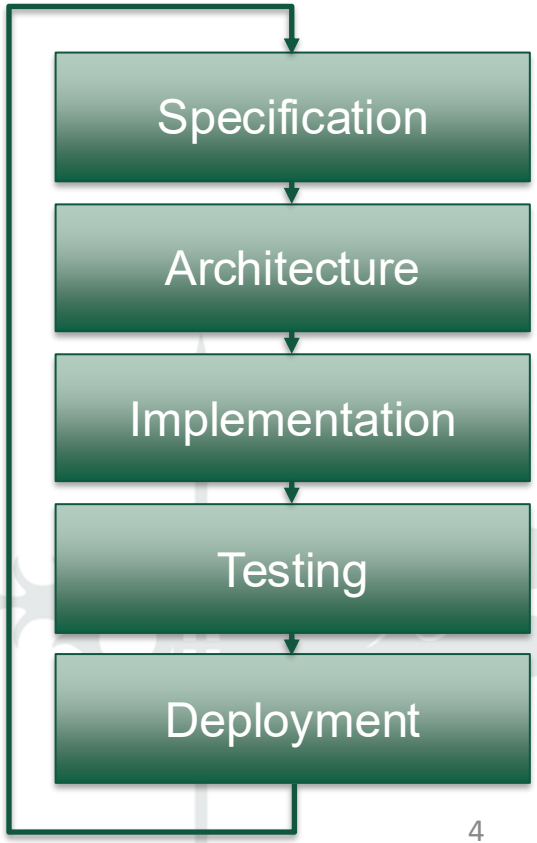


WILLIAM & MARY

CHARTERED 1693

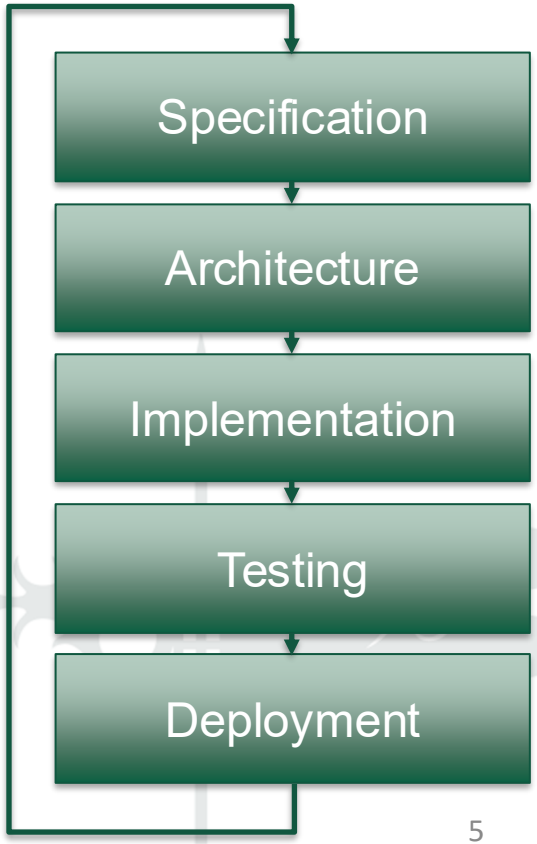
Software Lifecycle

- *What* do we specify?
- *How* do we know it is correct?

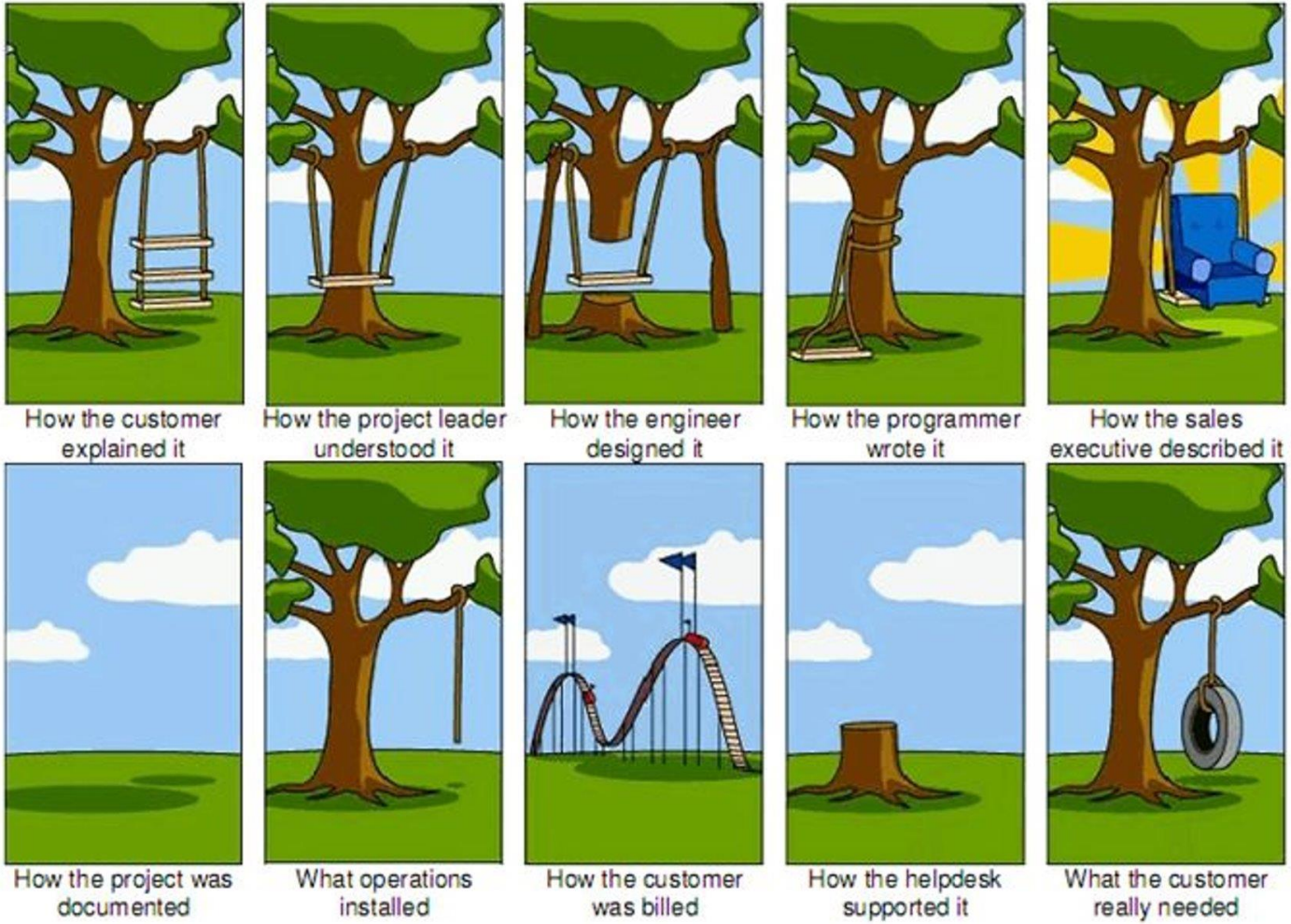


Software Lifecycle

- *What* do we specify?
- *How* do we know it is correct?
- How do we know it is *safe*?
  - Acceptable Risk
  - Within time/cost/capability
  - Treating system as a whole



https://sma.nasa.gov/sma-disciplines/system-safety



http://www.ricardoruizlopez.com/2012/09/10/what-would-you-like-software-specification/

Requirements vs Specifications

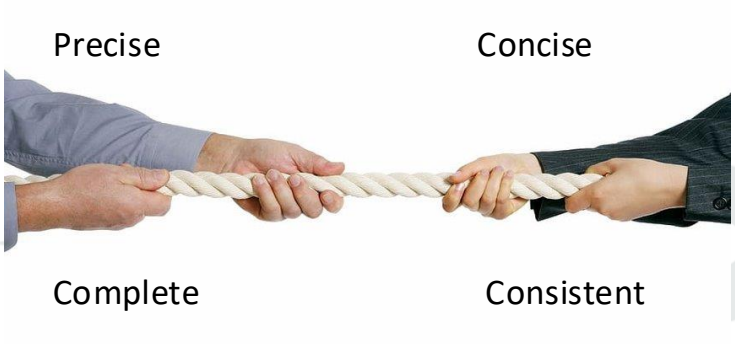
- Requirement
  - Broad description of goal – User stories
  - Not directly testable
- Specification
  - Formal, detailed description of what to do
  - Testable/checkable

Req vs Spec: Path-finding Robot

- System-level requirement
  - It must traverse an indoor environment
  - It must be able to travel from start to goal
- System-level specification
  - It must drive ≥1 m/s on tile floor at 10% grade
  - If a safe* path exists, robot must find it in 600s

Specification Goals

- Explain *what* to do not *how* to do
- Good specifications are:
  - Complete
  - Consistent
  - Concise
  - Precise



Ernst, *Specifications*, 2005

Example Requirements

- What are the automated vacuum's requirements?



Example Requirements



- What are the automated vacuum's requirements?
  - Clean common indoor floor types
  - Clean on a regular schedule
  - Navigate around furniture

Example Requirements

- What are the automated vacuum's safety requirements?



Example Requirements



- What are the automated vacuum's safety requirements?
  - Always return to base
  - Never stall the intake motor
  - Can be instantly disabled

Example Specifications

- What are the automated vacuum's specifications?



System vs Component Specs

- Vacuum as a whole:
  - Cleans up to 100sqft in less than 10 min
- Vacuum
  - Maintains constant suction of 10-15 cuft/min
- Sensing
  - Distance to objects $\pm$10mm at 60Hz
- Localization
  - Robot will map areas up to 500sqft in 5 min

System vs Component Safety

- Vacuum as a whole:
  - Cleans up to 100sqft in less than 10 min
- Vacuum
  - Maintains constant suction of 10-15 cuft/min
  - *Motor shuts off within 2s if it stalls*
- Sensing
  - Distance to objects $\pm$10mm at 60Hz
- Localization
  - Robot will map areas up to 500sqft in 5 min
  - *Specifications about sensing and localization interaction?*

Why do we need good specs?

- Mars polar lander (1999)
  - $165 million robot
  - Soil studies at Martian South Pole
  - Crash landed after software disengaged engine too early



Why do we need good specs?

- “A magnetic sensor is provided in each of the three landing legs to sense touchdown when the lander contacts the surface, initiating the shutdown of the descent engines”
- “The software—intended to ignore touchdown indications prior to the enabling of the touchdown sensing logic—was not properly implemented”
- [JPL Failure Report]



Paying attention to specs…

- Therac-25 - 1982-1987
  - Software race conditions caused massive overdoses in radiation. 3 injuries, 3 fatalities
- Ariane 5 - 1996
  - Re-used software from Ariane 4, specs not updated, crashed
- Mars Climate Orbiter - 1999
  - One component used metric units, another used imperial units - led to bad values, crashed
- Boeing 737 MAX - 2018, 2019
  - MCAS system over-corrected the plane's pitch. Two crashes totaling 346 fatalities

Specification Goals

- Explain *what* to do not *how* to do
- Good specifications are:
  - Complete
  - Consistent
  - Concise
  - Precise

How do we verify, validate, and enforce our specifications?

Ernst, *Specifications*, 2005

How can we design/check specs?

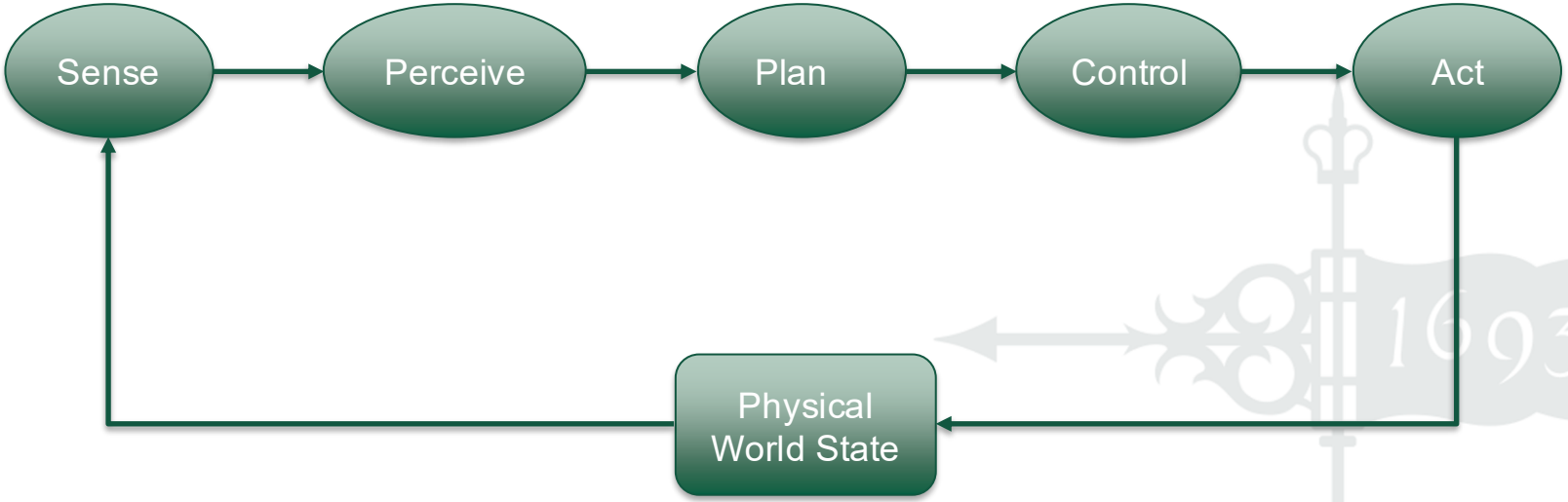
- If a safe* path exists, robot must find it in 600s

*What we mean by “safely” can depend on the robot, environment, etc. and must be rigorously specified

For example, recall in Lab 6 we added `safe_distance` as a tunable parameter
This altered the performance of the algorithm based on what threshold of safety was required

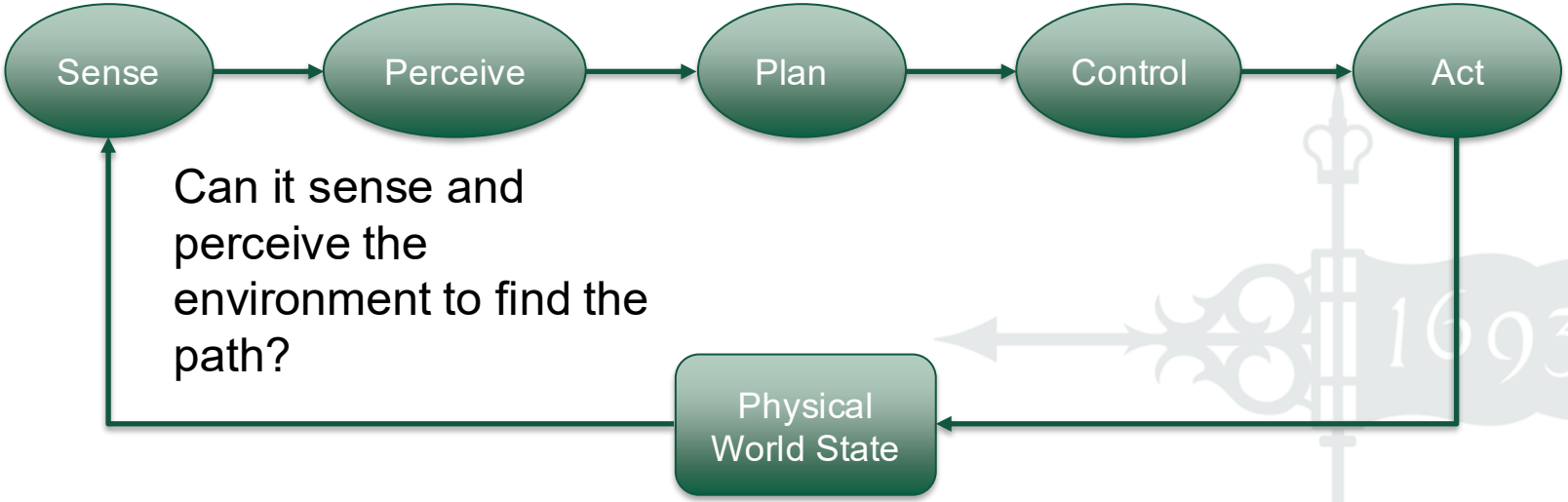
How can we design/check specs?

- If a safe* path exists, robot must find it in 600s



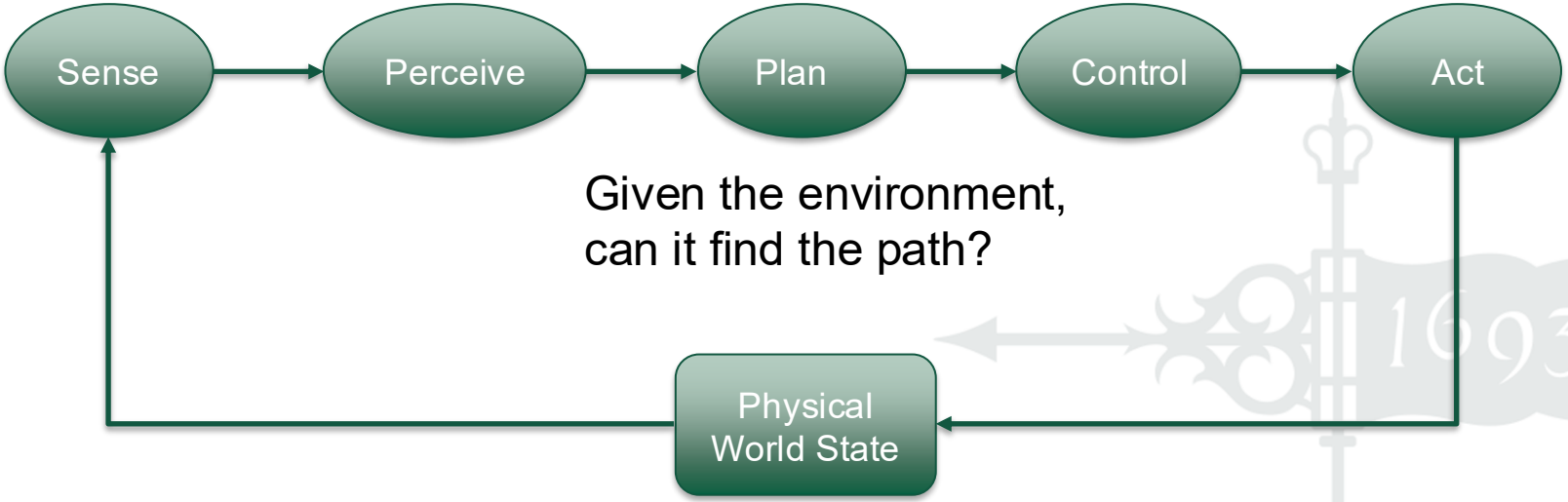
How can we design/check specs?

- If a safe* path exists, robot must find it in 600s



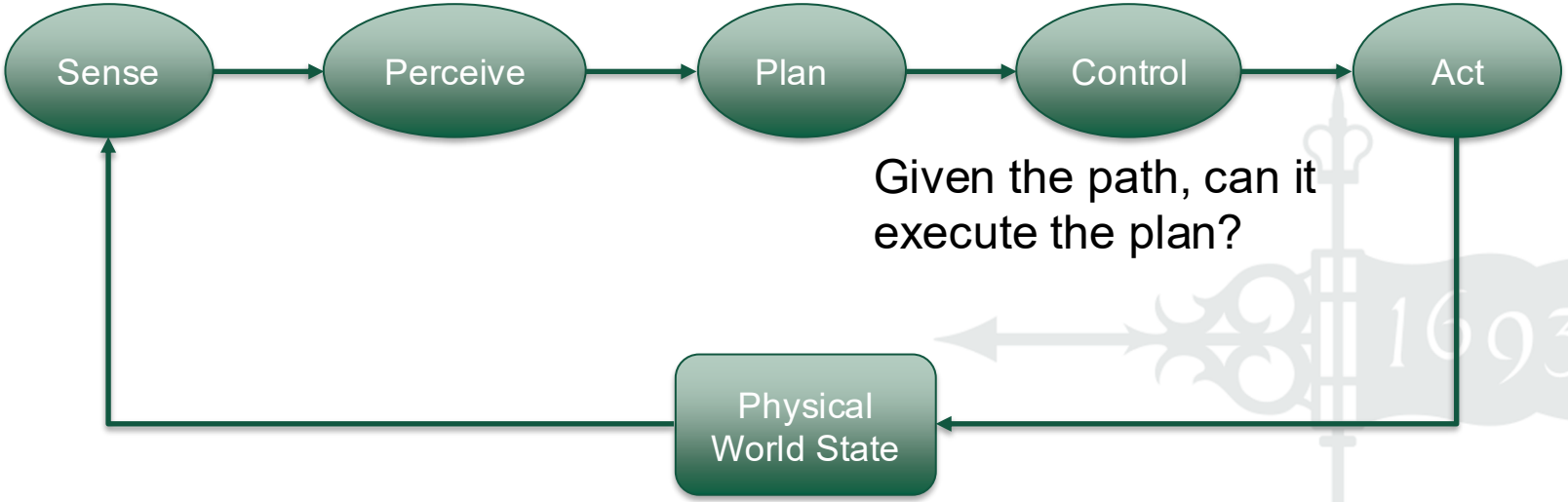
How can we design/check specs?

- If a safe* path exists, robot must find it in 600s



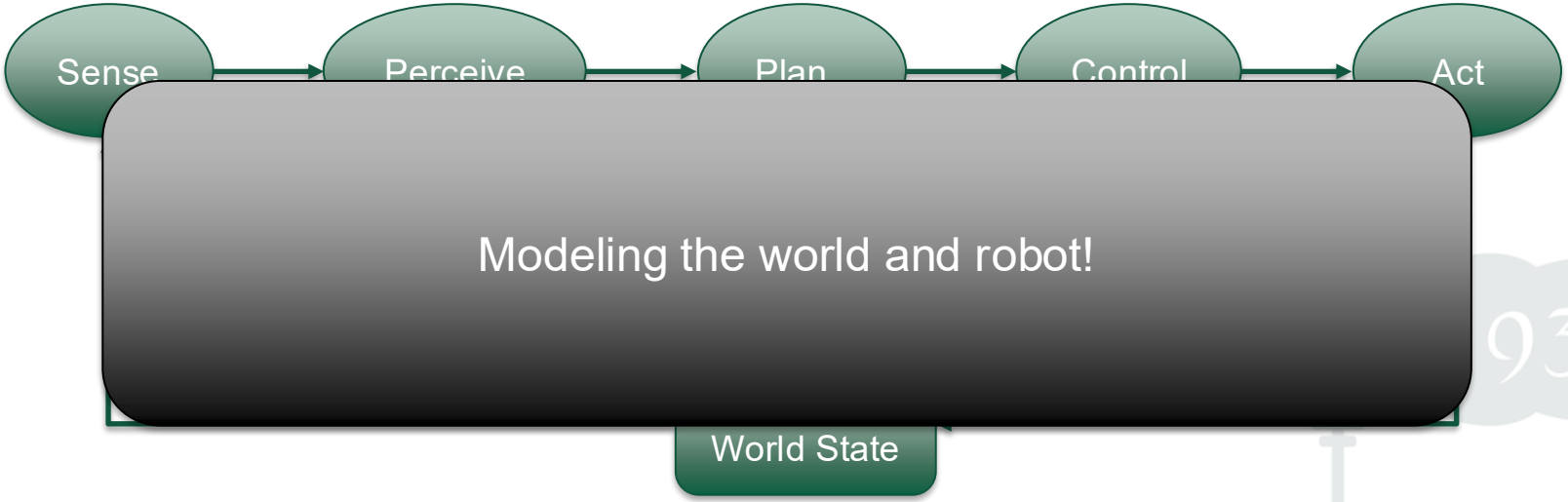
How can we design/check specs?

- If a safe* path exists, robot must find it in 600s



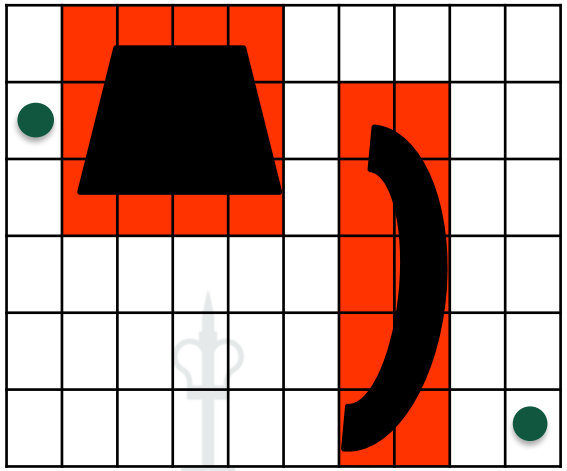
How can we design/check specs?

- If a safe* path exists, robot must find it in 600s



Modeling for Specifications

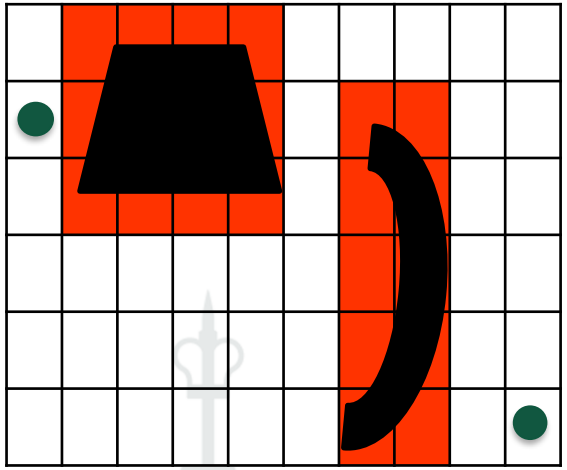
- Use abstractions!
- World
  - Grid world!
- Robot
  - A point that moves up/down/left/right



Modeling for Specifications

If a safe path exists, robot must find it

Given an environment, the robot must make a 2D grid and, if a path exists, the robot must be able to plan a path only moving on the grid



- Robot
  - A point that moves up/down/left/right

Modeling for Specification

- Given a model:
  - Test/check if the model meets the spec
  - Evaluate how well model captures the system
- What if model is too different from system?
  - False Positives: model violates, but not system
  - False Negatives: system violates, but not model

Modeling for Specification

- Given a model:
  - Test/check if the model meets the spec
  - Evaluate how well model captures the system
- What if model is too different from system?
  - False Positives: model violates, but not system
  - False Negatives: system violates, but not model

For safety, only tolerate false positives!

Using Specifications

- Verification
  - Analytically prove spec cannot be violated
- Validation
  - Gain empirical evidence spec is not violated
- Monitoring
  - While deployed, check spec is not violated

Verification

- *Prove* no violation
- Spec: thrust is always positive

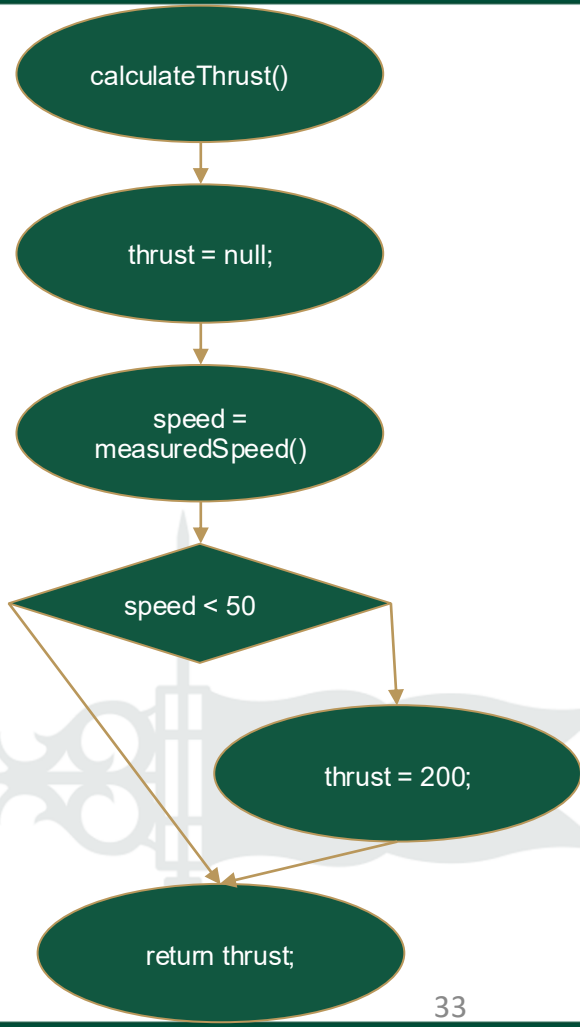
```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  return thrust; // Will thrust always be positive?
}
```

How do we build a model of this system?

Verification

- *Prove* no violation
- Spec: thrust is always positive

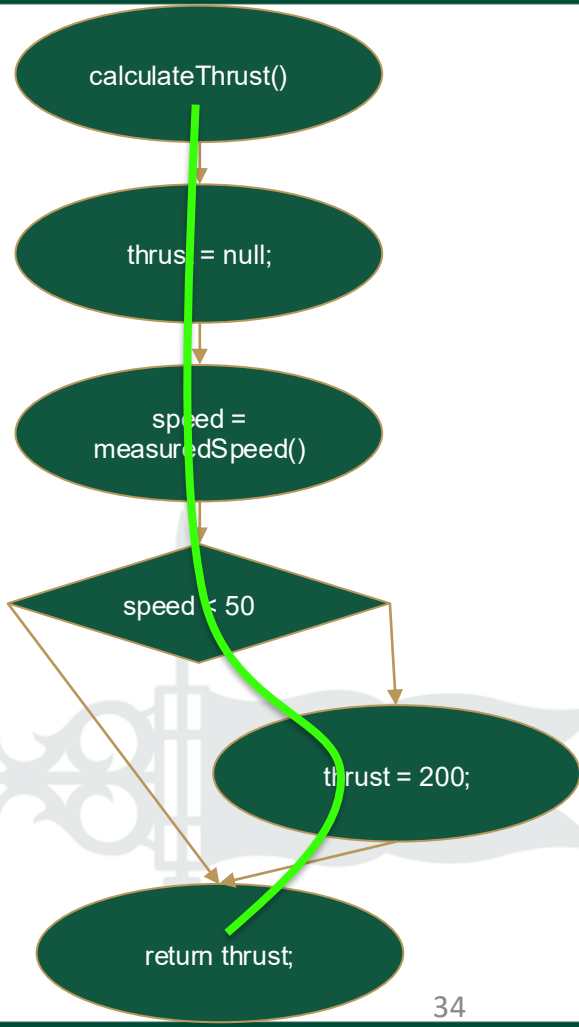
```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  return thrust; // Will thrust always be positive?
}
```



Verification

- *Prove* no violation
- Spec: thrust is always positive

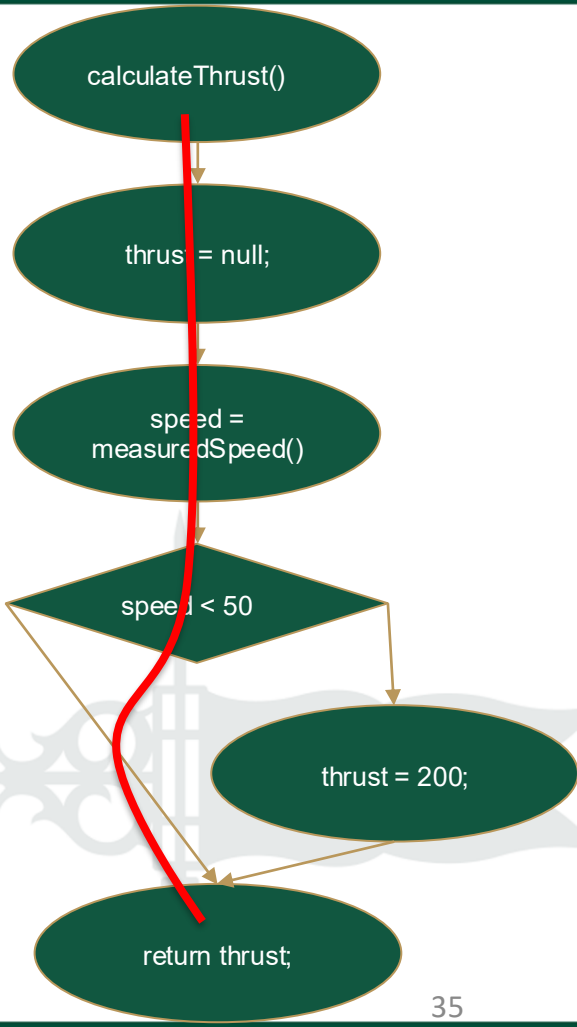
```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  return thrust; // Will thrust always be positive?
}
```



Verification

- *Prove* no violation
- Spec: thrust is always positive

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  return thrust; // Will thrust always be positive?
}
```

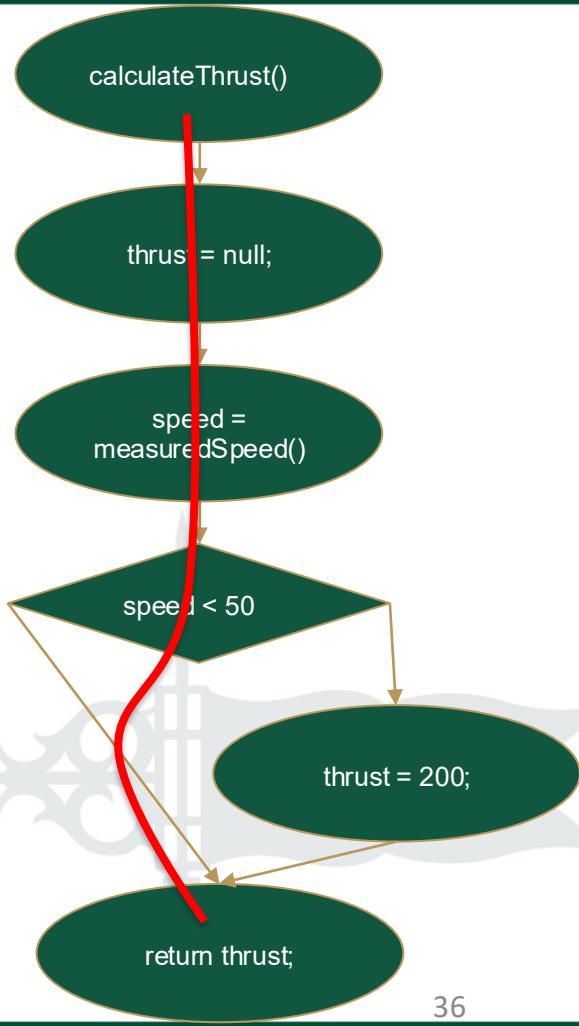


Verification

- *Prove* no violation
- Spec: thrust is always positive

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  return thrust; // Will thrust always be positive?
}
```

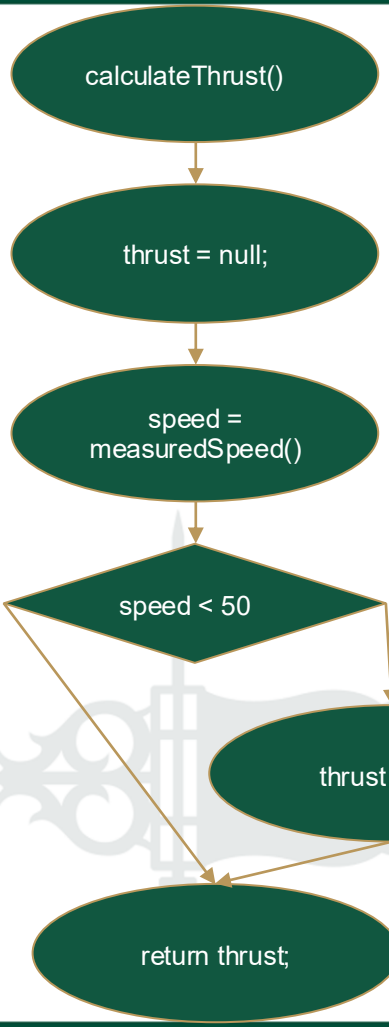
Any speed ≥ 50 = violation!



Verification

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  return thrust; // Will thrust always be positive?
}

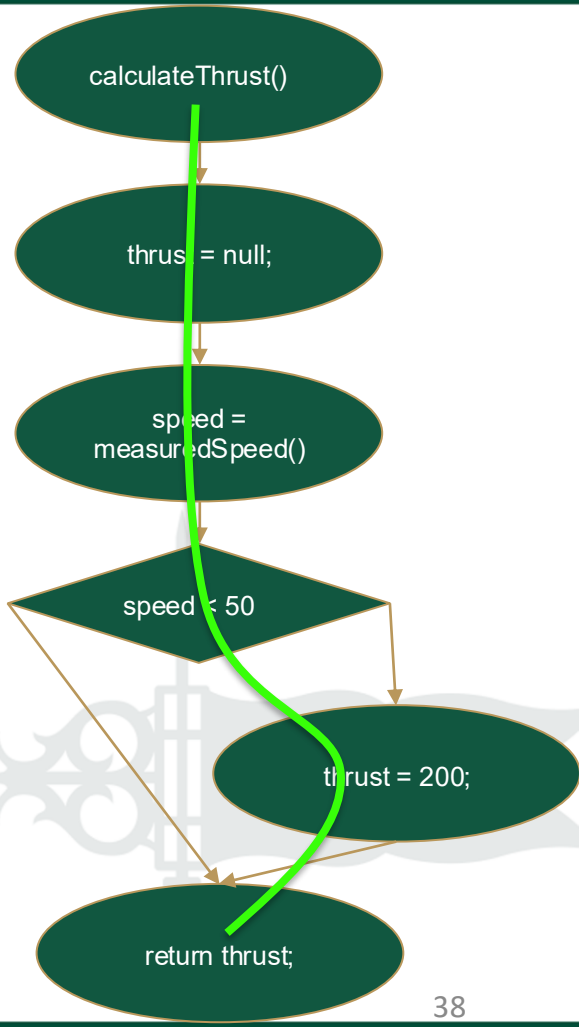
measureSpeed() {
  return min(45, max(0, sensor_value));
}
```



Verification

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  return thrust; // Will thrust always be positive?
}

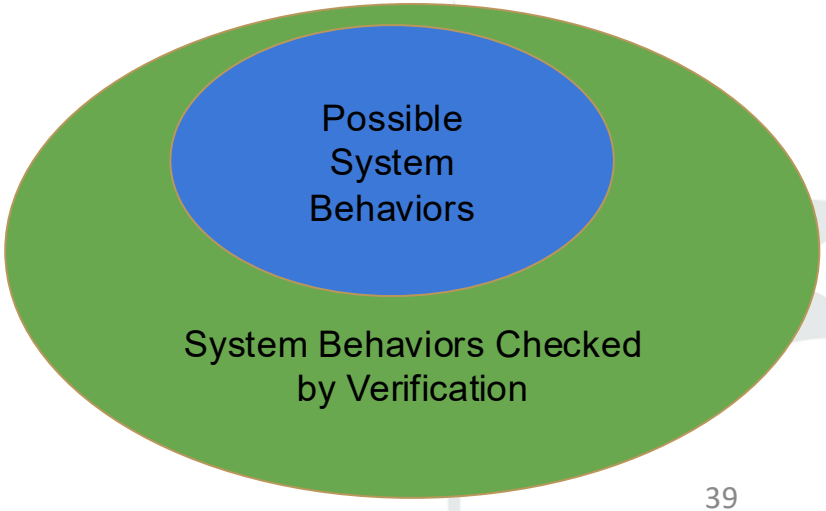
measureSpeed() {
  return min(45, max(0, sensor_value)); // Limited 0-45
}
```



Verification and Overapproximation

- “For every system behavior, does the spec hold”
- For this to be useful, must over approximate

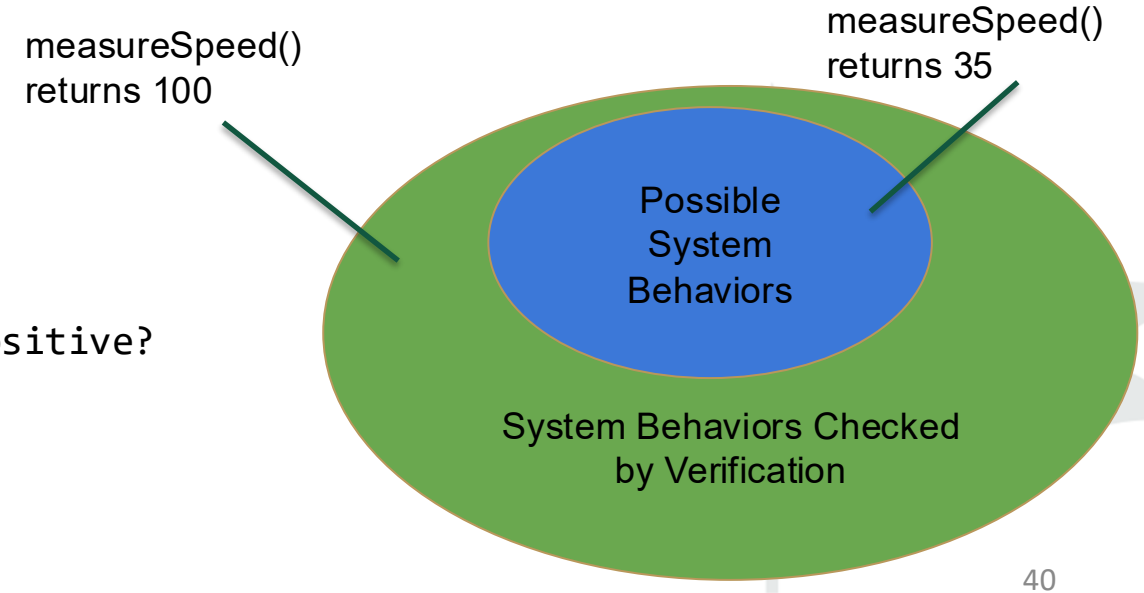
```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  // Will thrust always be positive?
  return thrust;
}
```



Verification and Overapproximation

- “For every system behavior, does the spec hold”
- For this to be useful, must over approximate

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  // Will thrust always be positive?
  return thrust;
}
```

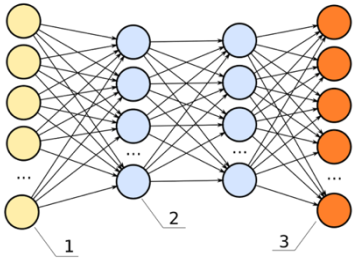


Verification of Complex Properties

- Machine Learning
  - Given range of inputs, output will be in X bound
- Control Plans
  - Temporal properties: once drone enters, it won't leave

Verification of Complex Properties

- Verification can be expensive!
  - Computation time to check all behaviors
  - Time to develop model of system
  - Time to handle false-positives





Validation

- Can we *test* that a spec is not violated?
- Spec: thrust is always positive

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  // Will thrust always be positive?
  return thrust;
}
```

Input (measuredSpeed)	Test Result
0	Pass
1	Pass
-1	Pass
2	Pass

Validation

- Can we *test* that a spec is not violated?
- Spec: thrust is always positive

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  // Will thrust always be positive?
  return thrust;
}
```

Input (measuredSpeed)	Test Result
	Fail
	Fail
	Fail

Validation

- Can we *test* that a spec is not violated?
- Spec: thrust is always positive

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  // Will thrust always be positive?
  return thrust;
}
```

Input (measuredSpeed)	Test Result
2147483647	Fail
Infinity	Fail
NaN	Fail

Validation

- Can we *test* that a spec is not violated?
- Spec: thrust is always positive

```
calculateThrust() {
  thrust = null;
  if (measureSpeed() < 50) {
    thrust = 200;
  }
  // Will thrust always be positive?
  return thrust;
}
```

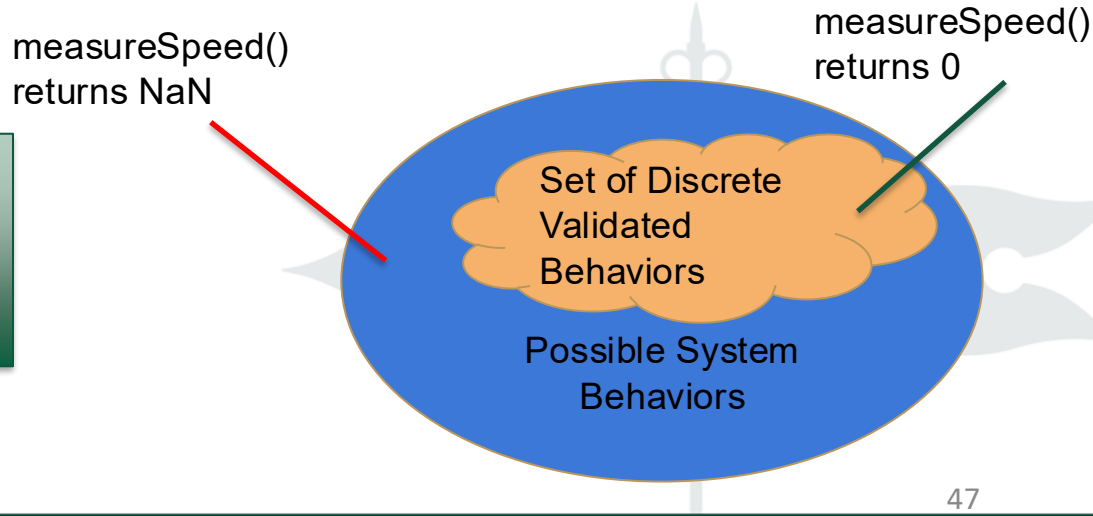
Input (measuredSpeed)	Test Result
2147483647	Fail
Infinity	Fail
NaN	Fail
50	Fail

Validation

- Validation provides util when tests fail
- Lack of failures doesn't mean none exist
- Success is evidence *for that input*

```
calculateThrust() {
```

It is important to design a robust test plan to exercise the system!

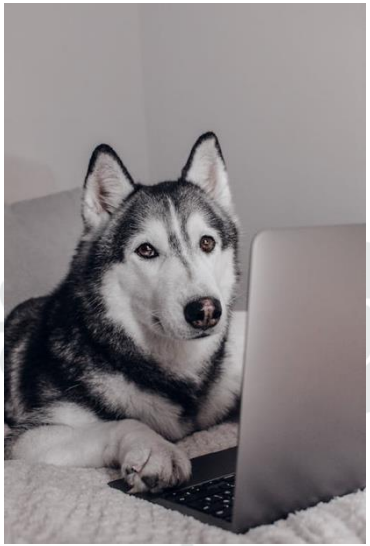


Runtime Monitoring

- Constantly check spec live in deployment
- If violated (or about to), intervene

Watchdog Timer:

- Constantly check if the system is "stuck" and intervene



Runtime Monitoring

- Constantly check spec live in deployment
- If violated (or about to), intervene

```
thrust_monitor() {
  thrust = calculateThrust();
  if (monitor_violated(thrust)) { // Specification violated!
    turnOffEngine();
  }
}
monitor_violated(thrust) {
  return !(thrust > 0)
}
```

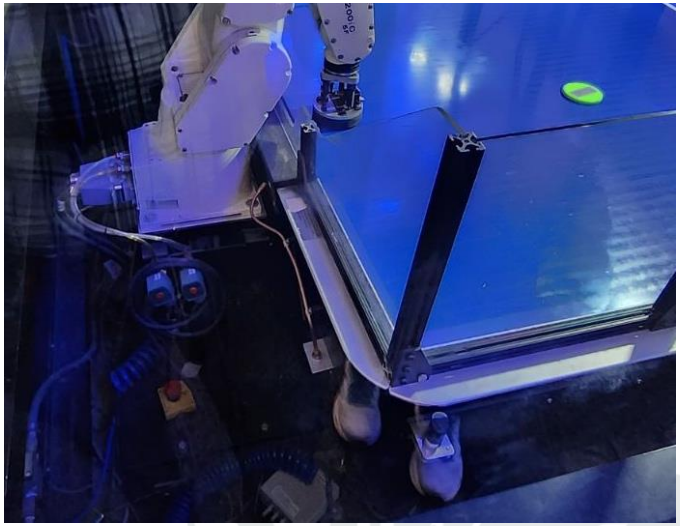
Runtime Monitoring

- Therac-25: radiation therapy system designed to emit controlled doses
- Software bugs caused it to emit lethal doses under certain (rare) conditions

- How could monitoring have helped?

Runtime Monitoring – ESTOP

- A dedicated, consistent way to stop, disarm, immobilize, or otherwise disable the robot
  - Industrial robotics
  - Nuclear robotics
  - Field robotics
  - Our robot vacuum



“Lab” 8 – Designing Specs

- Start from initial spec for go-to-goal robot
- Design a spec to handle *all possible scenarios*
- ***Before class on Thurs:***
  Blackboard submission on initial spec
- ***During class on Thurs:***
  we will work in groups to refine

Lab 8 – Scenario Description

Your robot serves the inside of ISC, delivering catering orders from the Starbucks in ISC3. The robot starts each mission at the Starbucks fully charged, loaded with the customer's order, and is sent on a go-to-goal mission to one of the rooms in ISC. The robot has a map of the static environment (walls, doors, elevators, etc.) but does not know the locations of any dynamic obstacles (tables, chairs, people, etc.). By connecting wirelessly to the building's network, the robot can contact the elevator/powered doors to wirelessly "push" any of the elevator/door's buttons. The robot must navigate through the building to the door outside of the customer, wait for the food to be delivered, then return to charge or receive the next order.

Lab 8 – Robot Commands

- Turn on
- Engage
- Disengage
- Emergency Stop
- Open Hatch
- Toggle Hatch Lock
- Set Geofence*: 2D
- Set Exclusion Zones* (obstacles): 2D
- Go to goal*: 2D location

*each floor can be a separate map

Lab 8 – Robot Capabilities

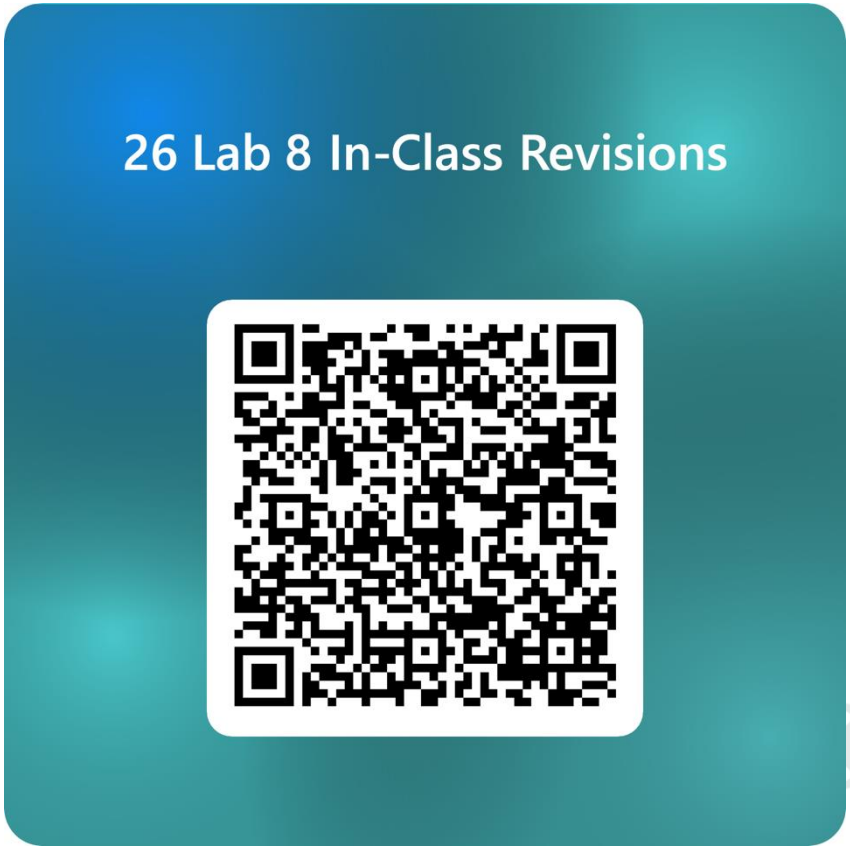
- LiDAR mounted on top (assume infinite precision)
- Bump sensor on entire front
- Non-slip rubber tires can reach 5m/s
- Robot can only rotate in place or move forward
- Battery capable of operating for at least 30 min
- Wireless capabilities to:
  - “push” buttons on elevator/door. Elevator/door responds with current state
  - Communicate to user/customer
- Top-mounted speakers
- Hatch that can lock/unlock/open to release order to customer

Lab 8 – Reqs and Specs

- The robot must always reach its goal, deliver the order, and return to base
- When the robot turns on, it is not engaged
- When e-stopped, the robot immediately stops until it is turned off/on again
- When disengaged, the robot safely stops operation
- The robot must never enter an exclusion zone or leave the geofence
- The robot must always preserve its ability to complete the mission
- The robot must never move while disengaged
- The robot must never run out of battery
- The robot must never collide with an obstacle at >2m/s
- The robot must immediately stop contact after collision
- If the goal cannot be achieved because of an exclusion zone/geofence, the robot must ask the user for an override

Lab 8 – Initial Questions (50%)

1. What happens if a GOAL command is sent after an ESTOP command?
2. What is the mission cruising speed of the robot?
3. How does the robot respond when it receives a new geofence that doesn't include the robot?
4. What happens when the robot approaches an obstacle on the way to the goal?
5. Given these requirements, describe one scenario you are unsure about designing a specification for (similar to the above).



https://forms.cloud.microsoft/r/g1DcBFPGrZ