17-423/723: Designing Large-scale Software Systems

Design for Robustness

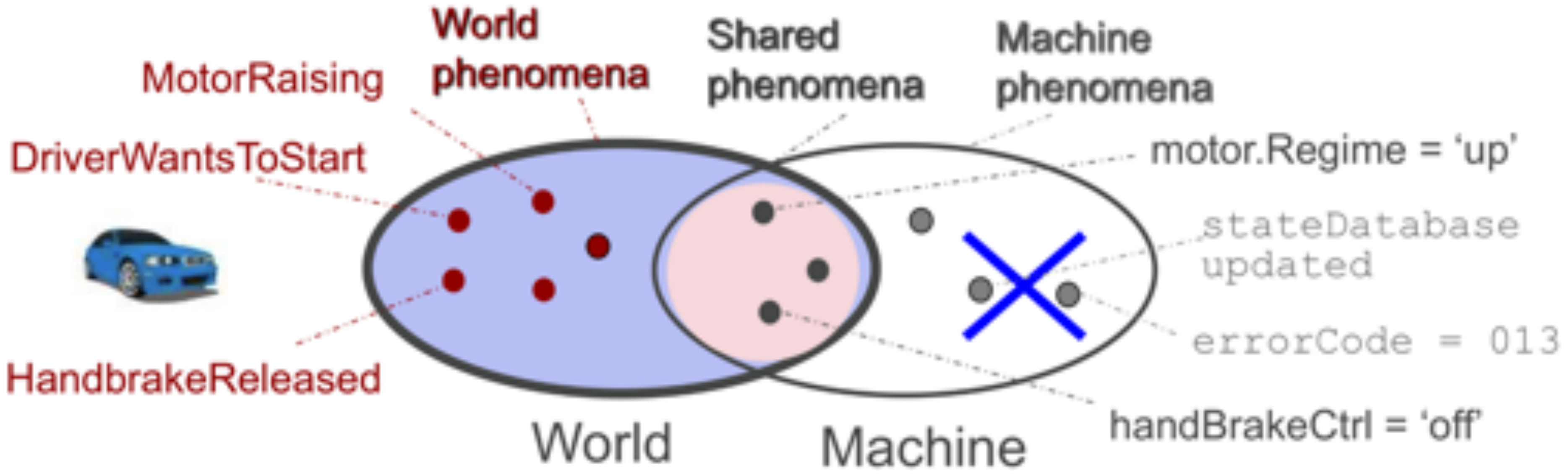
Mar 25 & 27, 2024

Leaning Goals

- Understand different ways in which a system may fail to meet its requirements and quality attributes
- Specify robustness as a quality attribute of a system
- Describe the differences between robustness, fault-tolerance, resilience, and reliability
- Apply fault tree analysis to identify possible root cause of a system failure
- Apply HAZOP to identify possible component failures and their impact on the system
- Apply design patterns for improve the robustness of a system

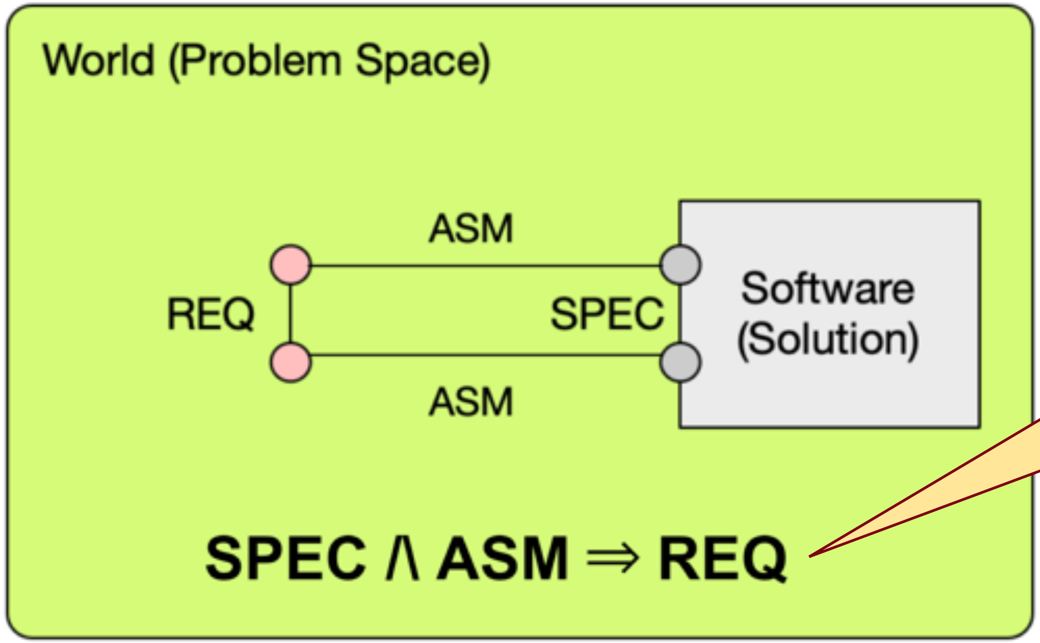
What can possibly go wrong with my system?

Recall: World vs. Machine



- **Shared phenomena**: Interface between the world & software
- Software can influence the world **only through the shared interface**
- Beyond this interface, we can only **assume** how the entities in the world will behave

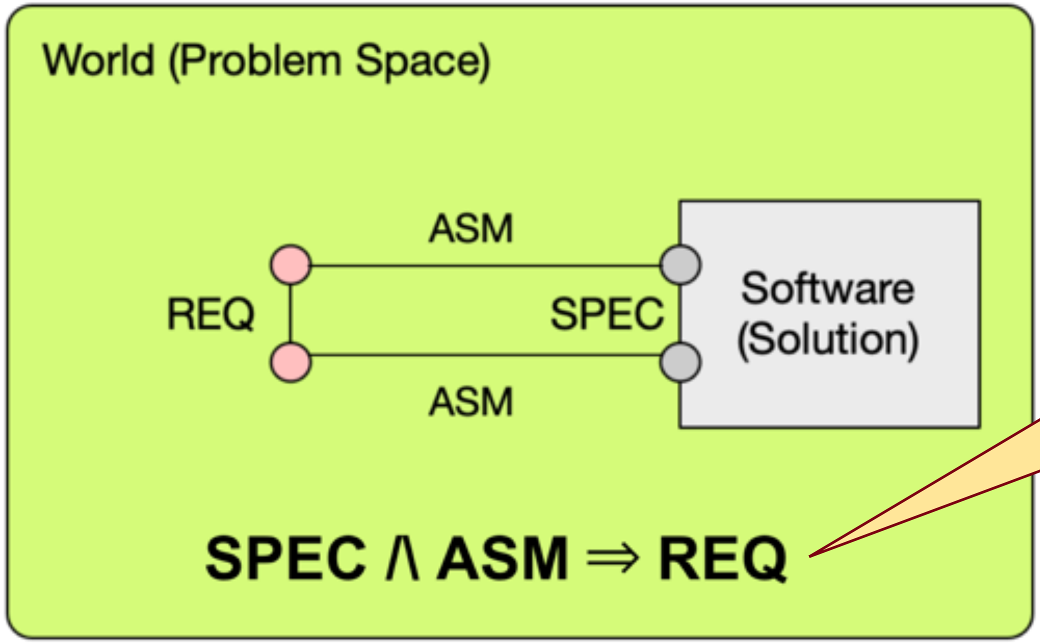
Recall: Satisfaction Argument



"If my software is implemented correctly (SPEC) and the world behaves as assumed (ASM), then the system should fulfill its requirement (REQ)"

- **Requirement (REQ):** What the system must achieve, in terms of desired effects on the world
- **Specification (SPEC)**: What software must implement, expressed over the shared interface
- **Domain assumptions (ASM)**: What's assumed about the world; bridge the gap between REQ and SPEC

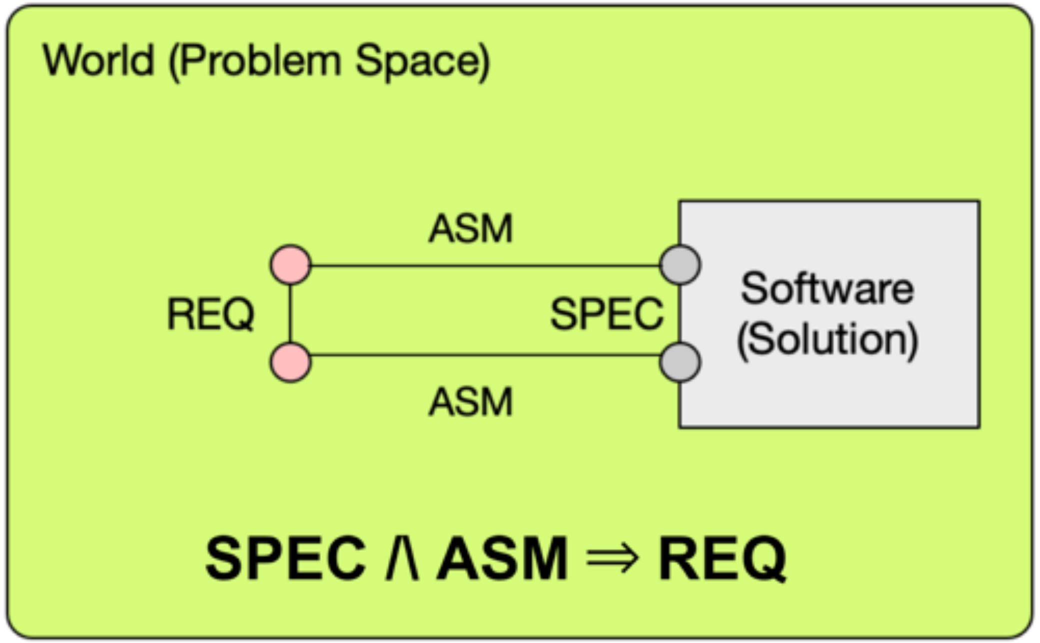
What can go wrong in my system?



"If my software is implemented correctly (SPEC) and the world behaves as assumed (ASM), then the system should fulfill its requirement (REQ)"

- **Q. What are some ways in which the system may fail to satisfy this argument?**

What can go wrong in my system?



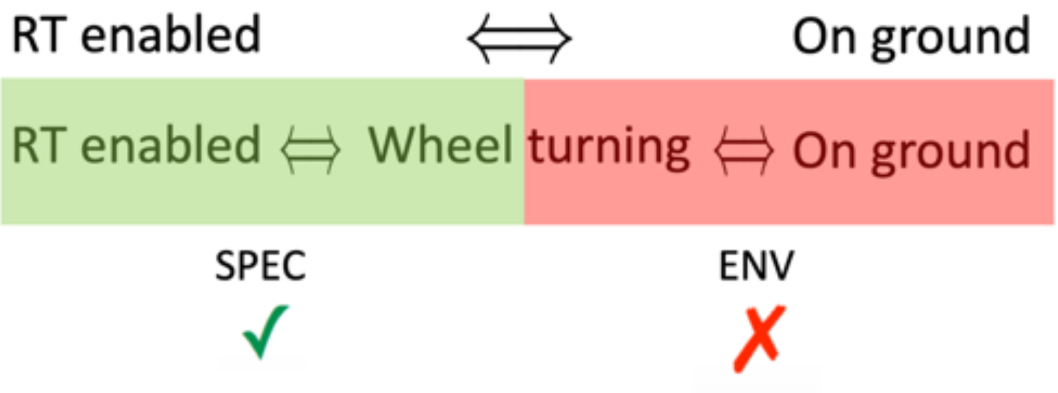
- Missing or incorrect specifications (SPEC)
- Violated specifications, due to bugs or faults in software (SPEC)
- Missing or incorrect assumptions (ASM)
- Missing or incorrect requirements (REQ)

Example: Lane Keeping Assist

Q. What can go wrong?

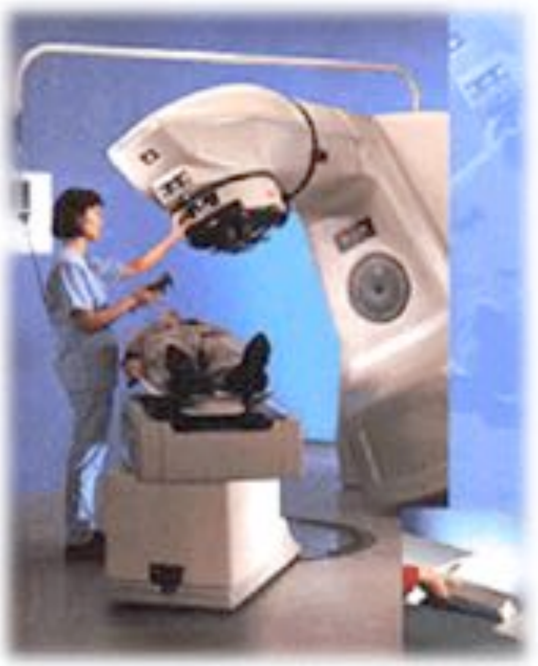
- **Requirement (REQ):** The vehicle must be prevented from veering off the lane.
- **Assumptions (ENV):** Sensors are providing accurate information about the lane; driver responses on time when given a warning; steering wheel is functional
- **Specifications (SPEC):** Lane detection accurately identifies the lane markings; controller generates correct steering commands to keep the vehicle within lane

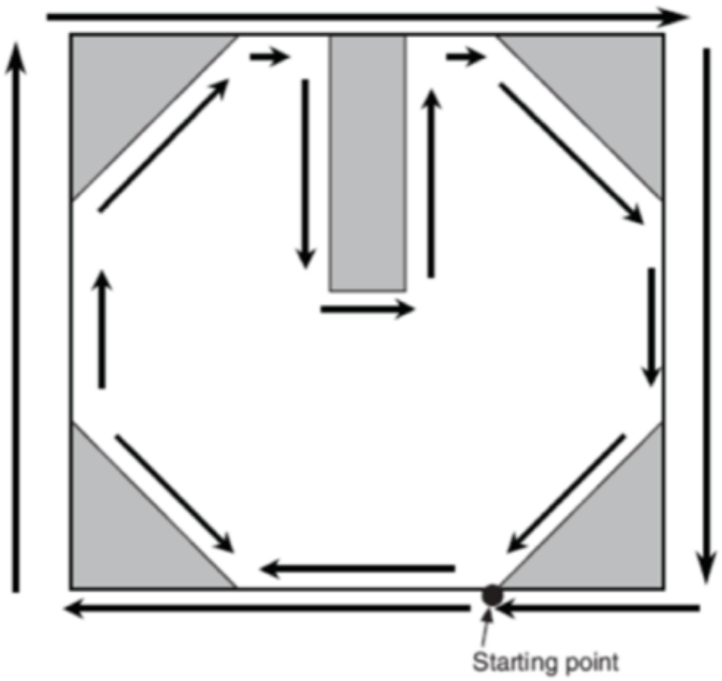
Recall: Lufthansa 2904 Runway Crash (1993)



- **Reverse thrust (RT):** Decelerates plane during landing
- What was required (REQ):
  RT is enabled if and only if plane is on the ground
- What was implemented (SPEC):
  RT is enabled if and only if wheel turning
- What was assumed (ENV):
  Wheel is turning if and only if it's on ground
- But runway was wet due to rain
  - Wheel failed to turn even when on ground
  - **Assumption (ENV) was incorrect!**
  - Pilot attempted to enable RT, but it was overridden by the software
  - Plane went off the runway and crashed

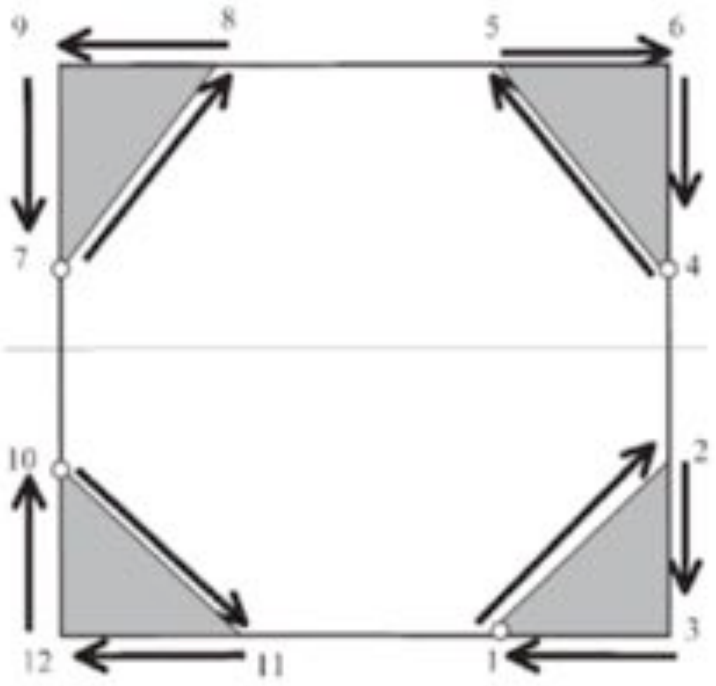
Example: Panama City Hospital (2000)





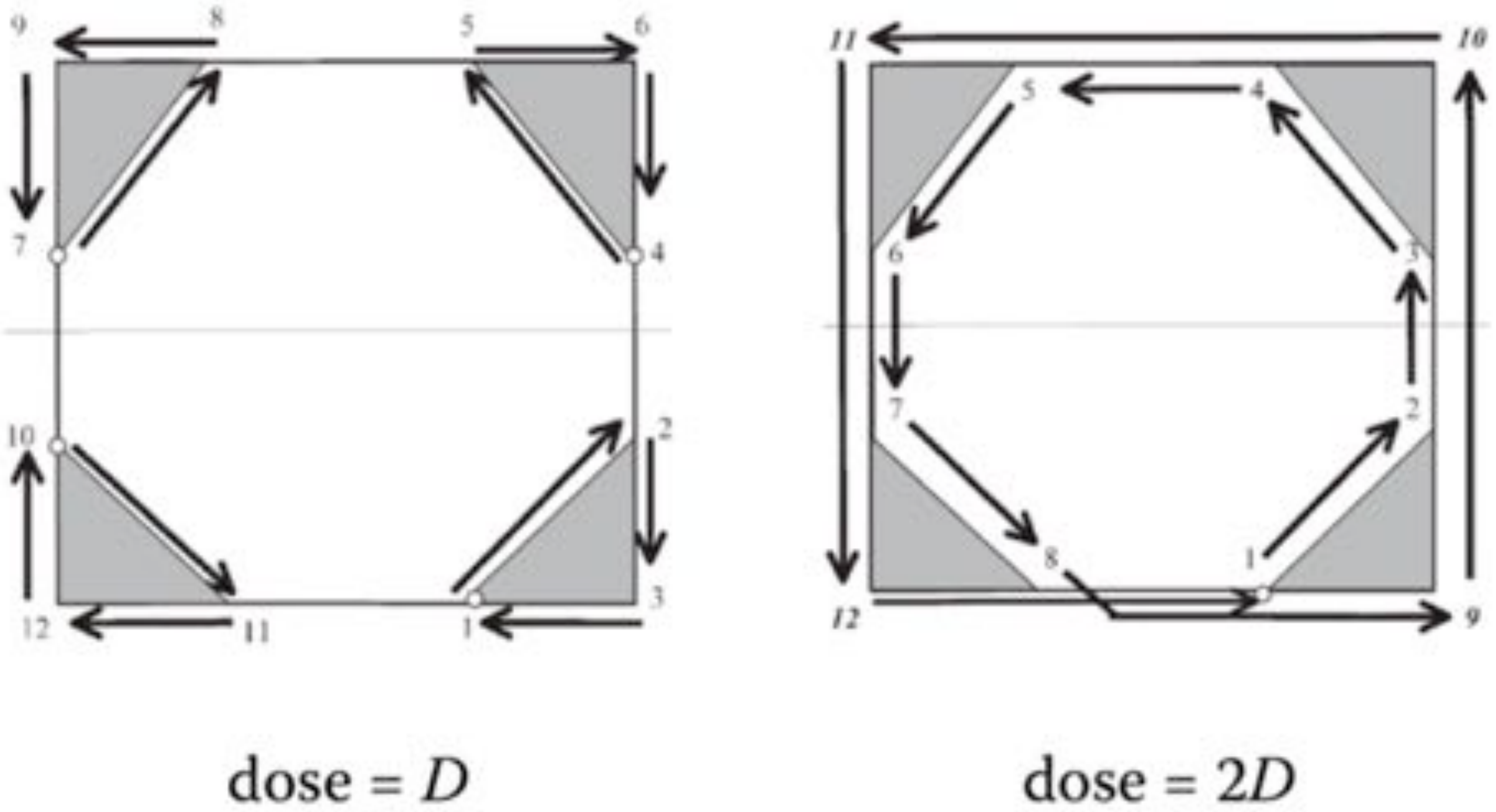
- Therapy planning software by Multidata Systems
- Theratron-780 by Theratronics (maker of Therac-25)
- **Shielding blocks**: Inserted into beam path to protect healthy tissue
- Therapist draws block shapes; software computes amount of radiation dose

Example: Panama City Hospital



dose = D

Example: Panama City Hospital



21 patients injured; 8 deaths

Blame the user or software?

- Lawsuits against the software company and hospital staff
- **Multidata Systems:**
  "Given [the input] that was given, our system calculated the correct amount, the correct dose. And, if [the staff in Panama] had checked, they would have found an unexpected result."
- Three therapists charged & found guilty for involuntary manslaughter; barred from practice for several years

Being robust against possible failures

- No system will ever be “correct”
- The environment will often behave in unexpected ways, violating assumptions (ASM)
- Software will have bugs and the underlying hardware will sometimes fail; specifications (SPEC) will be violated
- Even when these abnormal events occur, we want our systems to behave in an acceptable manner
  - Even if a user makes a mistake, this should not lead to a safety disaster
  - An off-by-one error should not lead to an entire rocket crashing
  - Even if some of the servers shutdown, the system should continue to provide critical services
- How do we design systems to be robust against such failures?

Robustness

Robustness

- The ability of a system to provide an **acceptable level of service** even when it operates under **abnormal conditions**
- Acceptable level of service: Quality attribute (typically of high importance) to be preserved, such as:
  - Safety: “No unsafe level of radiation delivered to the patient”
  - Performance: “The 95^{th}-tile response to client requests is at most 200ms”
  - Availability: “The patient record database is available 99% of the times”
- Abnormal conditions: An event or a condition that is outside of an expected, normal behavior, such as:
  - “The nurse deviates from the treatment instructions”
  - “The sensor provides an image with a significant amount of blur”
  - “The database is unresponsive and fails to store new appointments”

Robustness

- The ability of a system to provide an **acceptable level of service** even when it operates under **abnormal conditions**
- Acceptable level of service: Quality attribute (typically of high importance) to be preserved
- Abnormal conditions: An event or a condition that is outside of an expected, normal behavior
- **Q. Does this remind of you another quality attribute?**

Robustness

- The ability of a system to provide an **acceptable level of service** even when it operates under **abnormal conditions**
- Acceptable level of service: Functional requirement or quality attribute (typically of high importance) to be preserved
- Abnormal conditions: An event or a condition that is outside of an expected, normal behavior
- **Recall**: Scalability is the ability to handle growth in the amount of workload while maintaining an acceptable level of performance
  - Scalability can be thought of as one specific type of robustness!

Related Concepts

- Fault-tolerance: Ability of a system to provide acceptable service even when one or more of its components exhibit a faulty behavior
  - Typically about internal faults within a system
  - In this class, robustness covers both internal & external faults
- Resilience: Ability of a system to recover from an unexpected failure
  - Focus is on recovery instead of prevention
- Reliability: Ability of a system to provide acceptable level of service over a period of time
  - Typically measured as a "mean time between failures" (MTBF); e.g., 1 system failure over 1000 hours
  - Robustness is necessary to achieve reliability

Specifying Robustness: Good & Bad Examples

- The radiation therapy system should never deliver more than a maximum amount of radiation no matter what the nurse inputs
- The autonomous vehicle must operate even under a severe weather
- The scheduling app must process appointments even if the connection to the central database is lost
- Amazon must provide provide a response time less than 100ms even when the amount of concurrent customers exceeds 2 million
- The package delivery drone should never drop a package at a wrong location
- The autonomous vehicle must avoid hitting a pedestrian even if an object detection model fails to recognize it

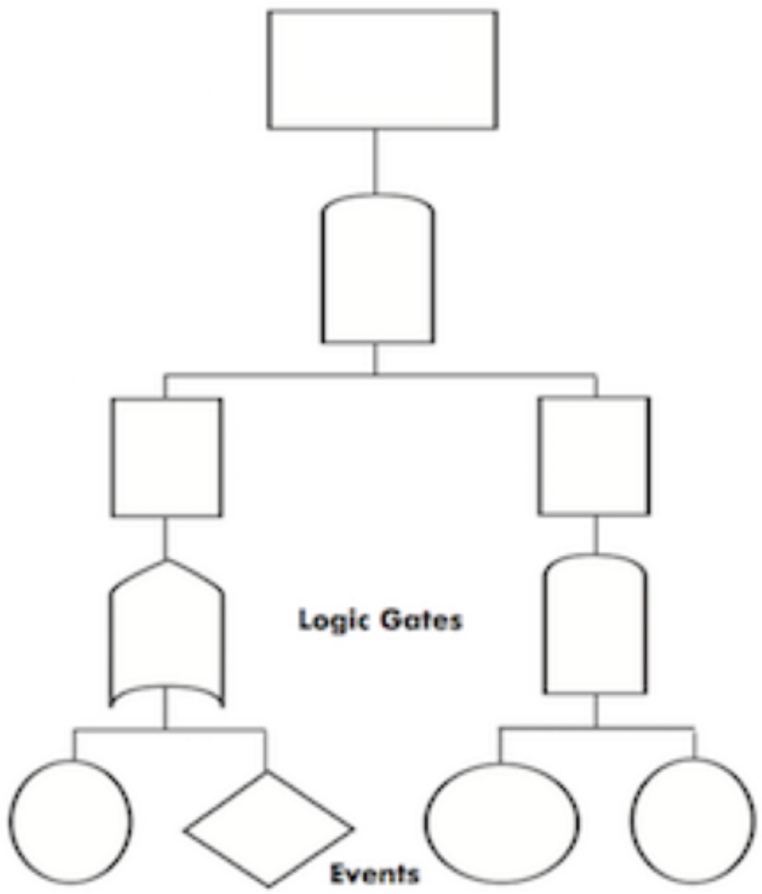
Failure Analysis

Failure Analysis

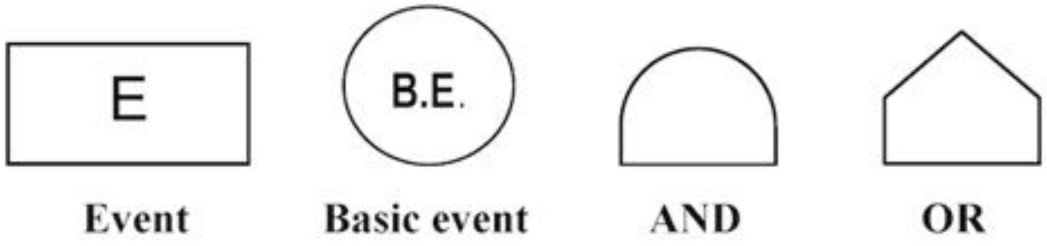
- *What can possibly go wrong in my system, and what is potential impact on system requirements?*
- Systematically analyze a design and identify different scenarios in which the system may fail to satisfy its requirements
- A number methods, developed and routinely applied in many engineering disciplines
  - **Fault tree analysis (FTA)**
  - **Hazard and operability study (HAZOP)**
  - Failure mode & effects analysis (FMEA)
  - Why-because analysis
  - ...

Fault-Tree Analysis (FTA)

- **Fault tree:** Specify relationships between a system failure (i.e., requirement violation) and its potential causes
  - Identify sequences of events that result in a failure
  - Prioritize the contributors leading to the failure
  - Inform decisions about how to (re-)design the system
  - Investigate an accident & identify the root cause
- Often used for safety & reliability, but can also be used for other types of QAs (e.g., poor performance, security attacks…)



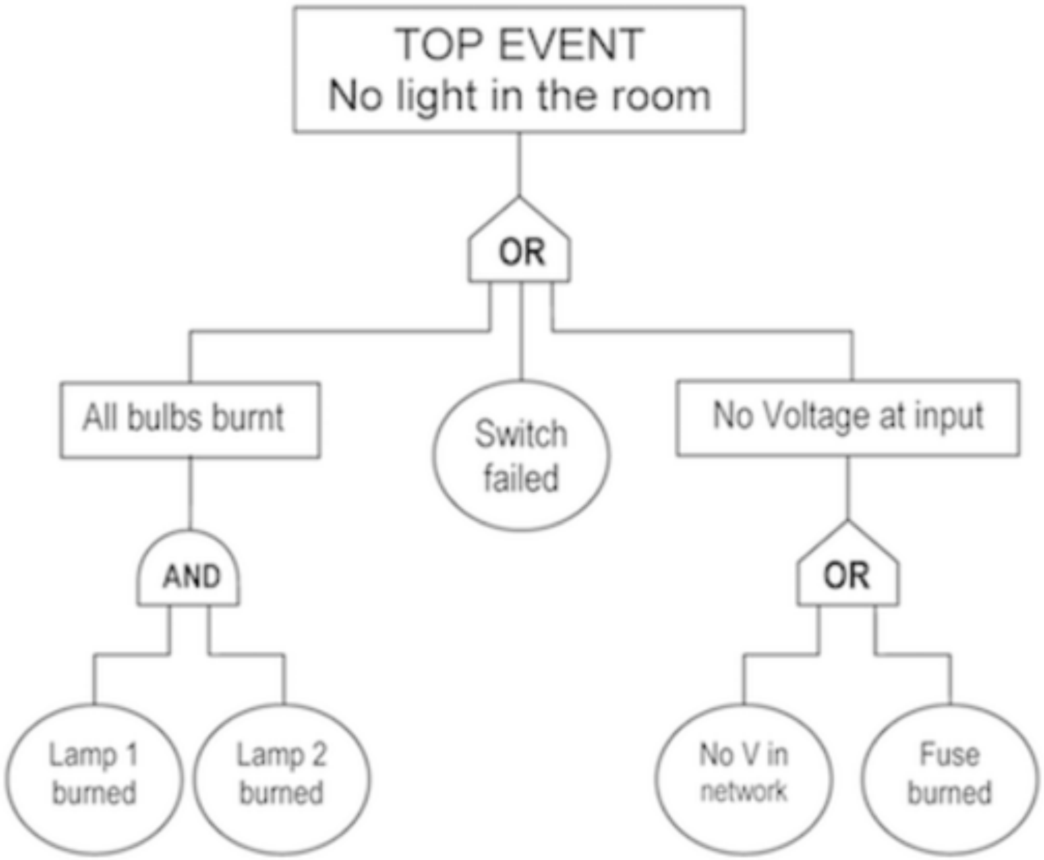
Elements of Fault Trees



- Event: A fault or an undesirable event
  - Non-basic event: An event that can be explained in terms of other events
  - Basic event: No further development or breakdown; leaf node in the tree
- Gate: Logical relationship between an event & its immediate subevents
  - AND: All of the sub-events must take place
  - OR: Any one of the sub-events may result in the parent event

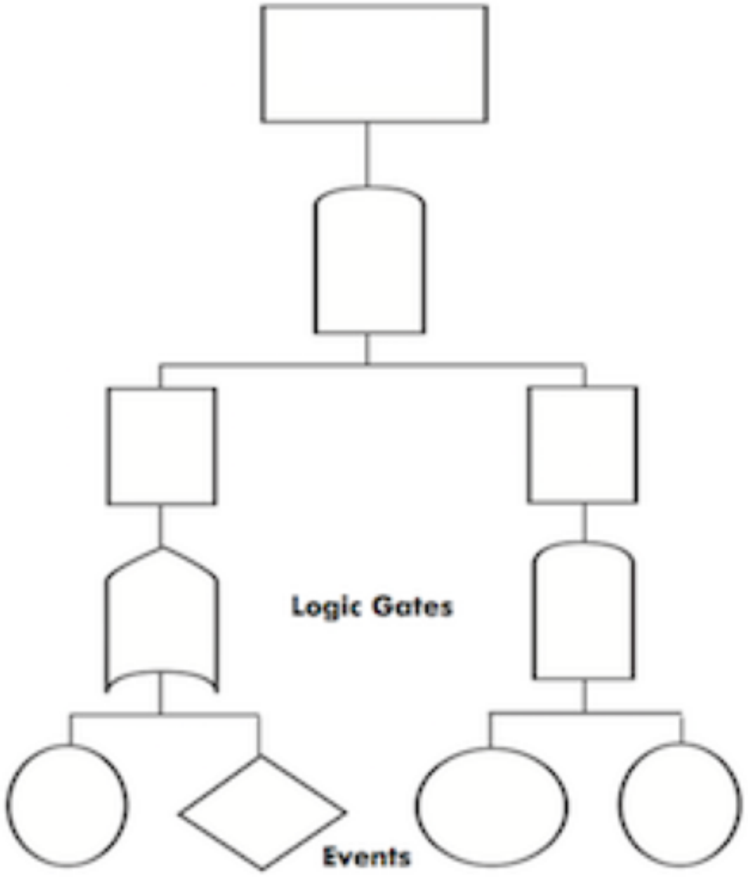
Elements of Fault Trees

- Every tree begins with a TOP event (typically a requirement violation or a hazardous event)
- Every non-basic event is broken into a set of child events and connected through an AND or OR gate
- Every branch of the tree must terminate with a basic event

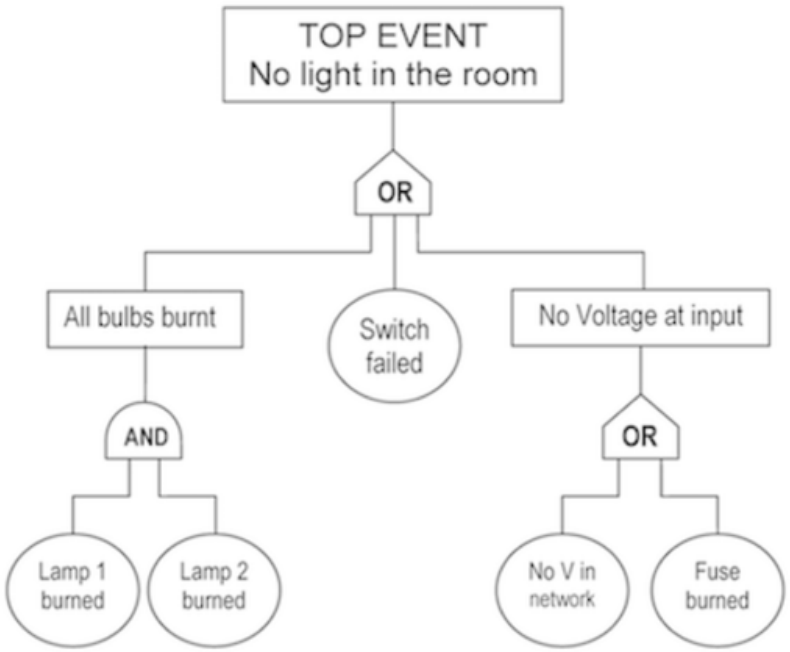


What can we do with FTA?

- **Qualitative analysis**: Determine potential root causes of a failure through **minimal cut set analysis**
- **Quantitative analysis**: Compute the probability of a failure based on the probabilities of the basic events



Minimum Cut Analysis



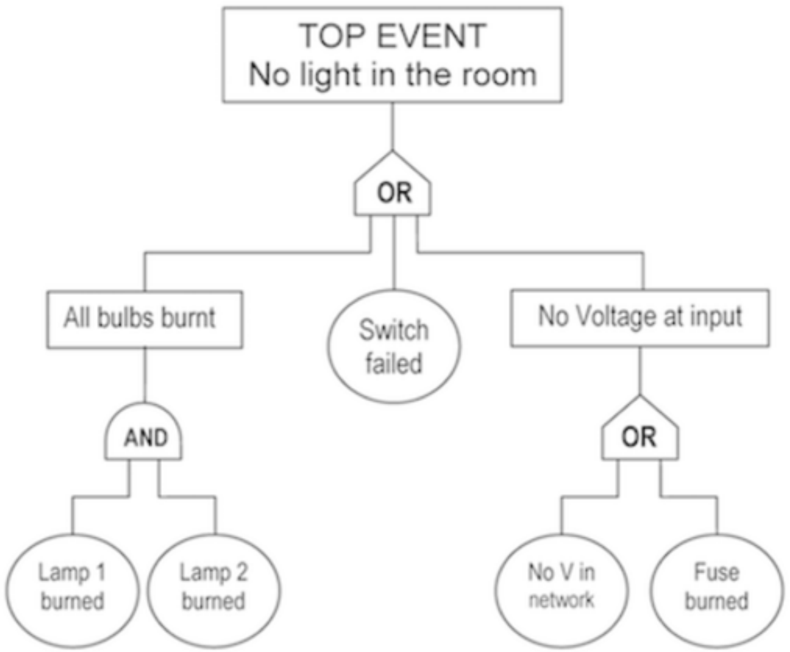
Minimal cut sets = {

??

}

- **Cut set**: A set of basic events whose simultaneous occurrence is sufficient to guarantee that the TOP event occurs.
- **Minimal cut set**: A cut set from which a smaller cut set cannot be obtained by removing a basic event.

Minimum Cut Analysis

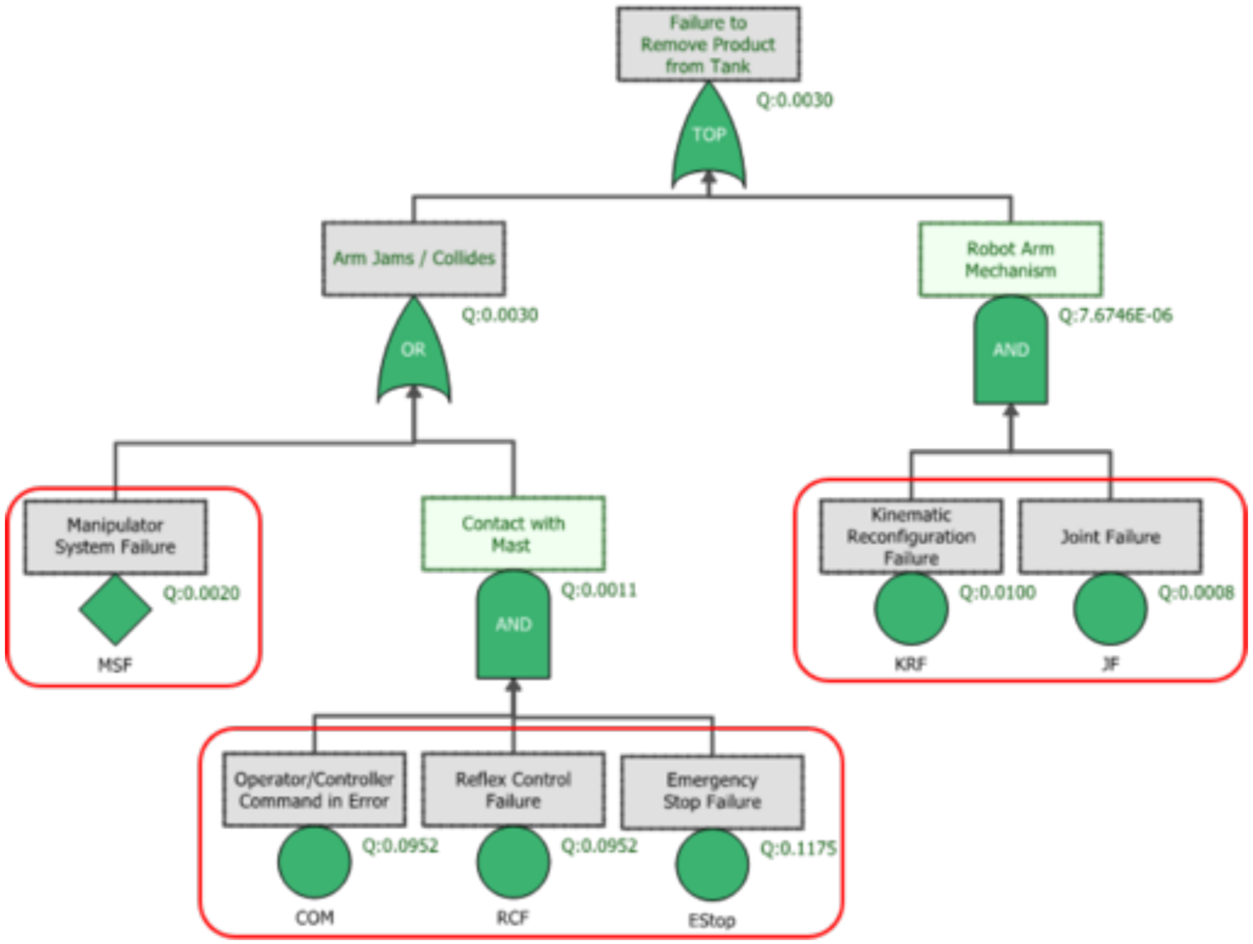


Minimal cut sets = {
{Lamp 1 burned, Lamp 2 burned},
{Switch failed},
{No V in network},
{Fuse burned}
}

- **Cut set**: A set of basic events whose simultaneous occurrence is sufficient to guarantee that the TOP event occurs.
- **Minimal cut set**: A cut set from which a smaller cut set cannot be obtained by removing a basic event.

Failure Probability Analysis

- To compute the probability of the top event:
  - Assign probabilities to basic events (based on data analysis or domain knowledge)
  - Apply probability theory to compute probabilities of intermediate events through AND & OR gates
- Alternatively, compute the top event probability as a sum of prob. of minimal cut sets
- **Q. This is difficult to do with software – why?**

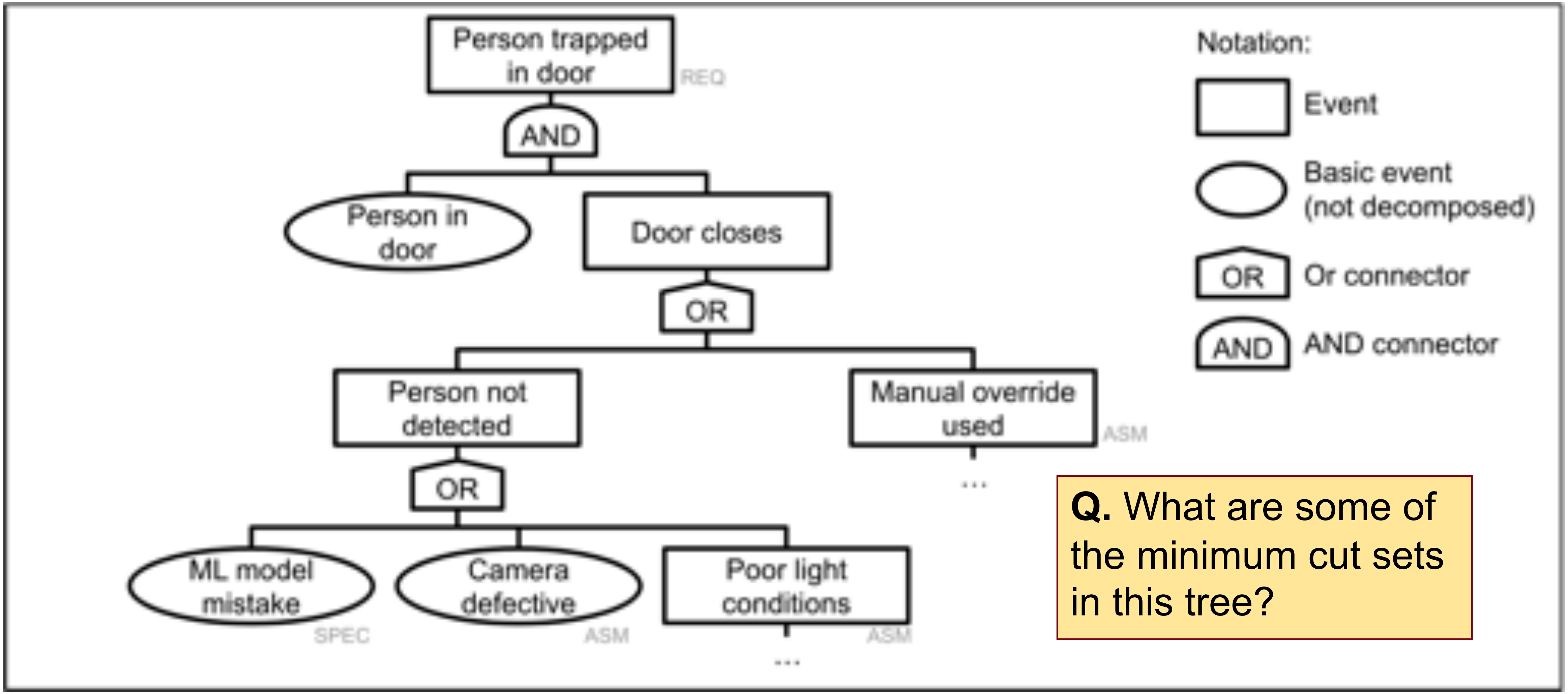


Example: Autonomous Train

Example: Autonomous Train

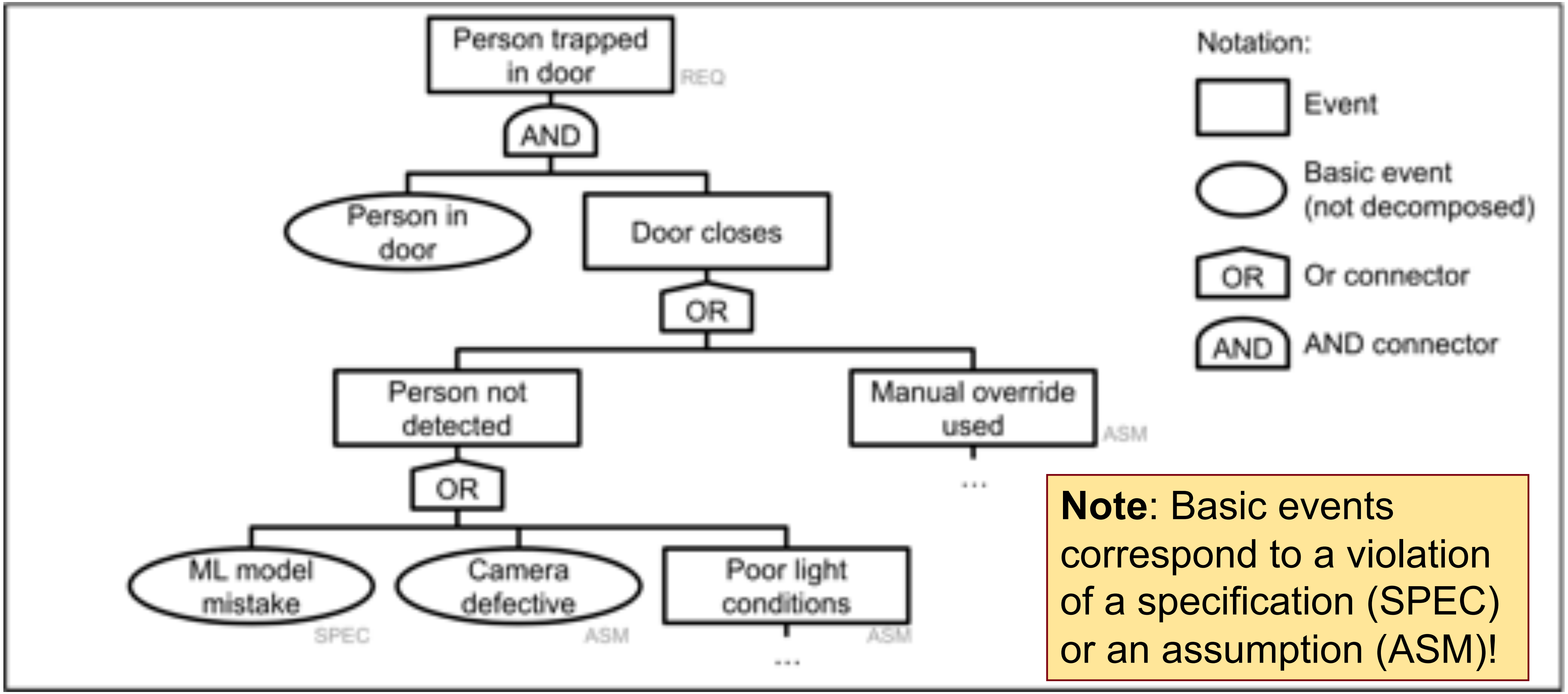
- **Requirements**: The train shall not depart all doors are closed. The train shall not trap people between the doors.
- Train uses a vision-based system to identify people in the door
- Use a fault tree to identify possible ways in which the person may be trapped in a door.

FTA Example: Autonomous Train



Q. What are some of the minimum cut sets in this tree?

FTA Example: Autonomous Train

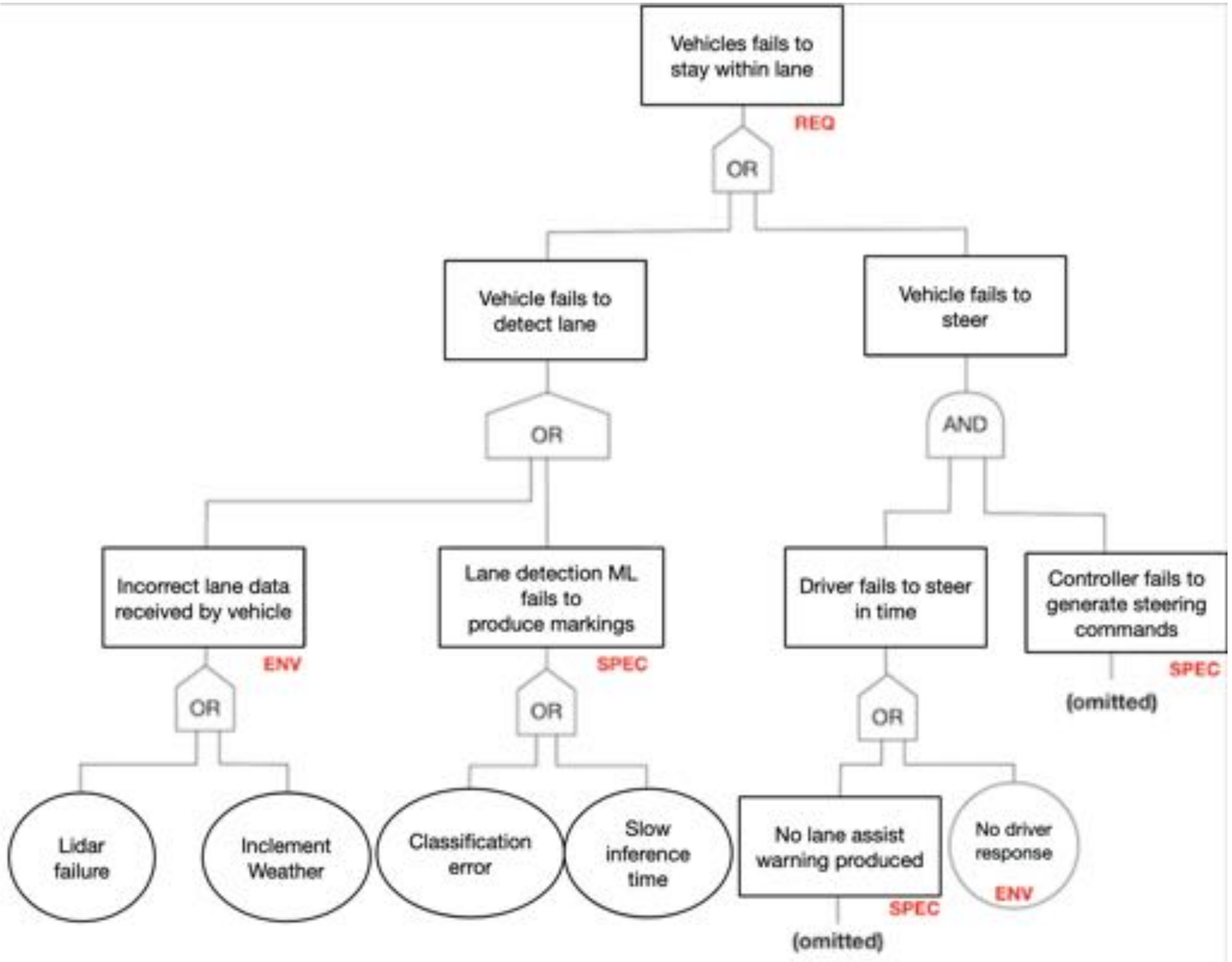


Note: Basic events correspond to a violation of a specification (SPEC) or an assumption (ASM)!

FTA Exercise: Lane Keeping Assist

- **Requirement**: The vehicle must be prevented from going off the lane.
- Use the failure to satisfy this as the TOP event
- Perform FTA to identify possible causes of this failure

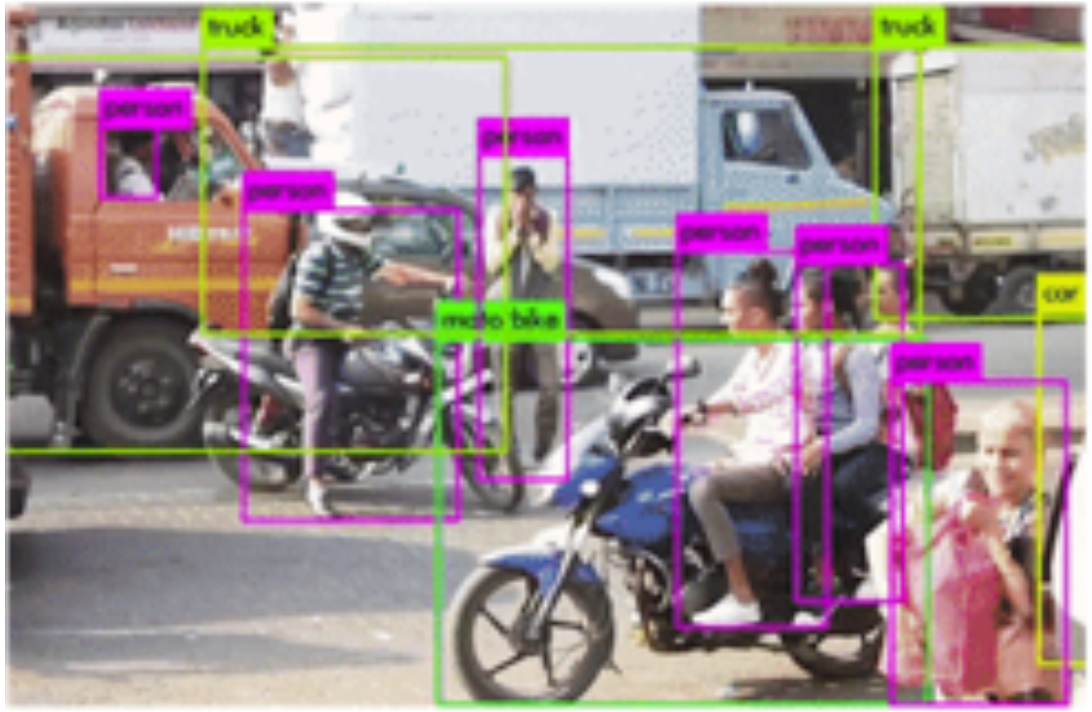
FTA Exercise: Lane Keeping System



FTA: Caveats

- In general, building a “complete” tree is impossible
  - There are probably some faulty events that you missed (i.e., ”unknown unknowns”)
- Domain knowledge is crucial for improving coverage
  - Talk to domain experts to identify important and common basic events for your application domain
- FTA is still very valuable for risk reduction!
  - Forces you to think about & explicitly document possible failure scenarios
  - A good starting basis for designing mitigations (more on this in the next lecture)

Hazard and Operability Study (HAZOP)



Guide Word	Meaning
NO OR NOT	Complete negation of the design intent
MORE	Quantitative increase
LESS	Quantitative decrease
AS WELL AS	Qualitative modification/increase
PART OF	Qualitative modification/decrease
REVERSE	Logical opposite of the design intent
OTHER THAN / INSTEAD	Complete substitution
EARLY	Relative to the clock time
LATE	Relative to the clock time
BEFORE	Relating to order or sequence
AFTER	Relating to order or sequence

- **Goal**: Identify hazards and component faults through systematic, pattern-based inspection of component functions

HAZOP

- HAZOP is a **bottom-up** method to identify potential failures: It starts from individual components
  - FTA is a **top-down** method: It starts from a top-level failure and links it to component-level faults
- HAZOP process:
  - For each component, specify the expected behavior of the component (SPEC)
  - Use a set of **guide words** to generate possible deviations from expected behavior
  - Analyze the impact of each generated deviation: Can it result in a system-level failure?

Guide Word	Meaning
NO OR NOT	Complete negation of the design intent
MORE	Quantitative increase
LESS	Quantitative decrease
AS WELL AS	Qualitative modification/increase
PART OF	Qualitative modification/decrease
REVERSE	Logical opposite of the design intent
OTHER THAN / INSTEAD	Complete substitution
EARLY	Relative to the clock time
LATE	Relative to the clock time
BEFORE	Relating to order or sequence
AFTER	Relating to order or sequence

HAZOP Example: Emergency Braking (EB)

Guide Word	Meaning
NO OR NOT	Complete negation of the design intent
MORE	Quantitative increase
LESS	Quantitative decrease
AS WELL AS	Qualitative modification/increase
PART OF	Qualitative modification/decrease
REVERSE	Logical opposite of the design intent
OTHER THAN / INSTEAD	Complete substitution
EARLY	Relative to the clock time
LATE	Relative to the clock time
BEFORE	Relating to order or sequence
AFTER	Relating to order or sequence

- Component: Software controller for EB
  - Expected behavior (SPEC): If the ego vehicle is too close to the leading vehicle, generate a maximum amount of braking to prevent collision

HAZOP Example: Emergency Braking (EB)

Guide Word	Meaning
NO OR NOT	Complete negation of the design intent
MORE	Quantitative increase
LESS	Quantitative decrease
AS WELL AS	Qualitative modification/increase
PART OF	Qualitative modification/decrease
REVERSE	Logical opposite of the design intent
OTHER THAN / INSTEAD	Complete substitution
EARLY	Relative to the clock time
LATE	Relative to the clock time
BEFORE	Relating to order or sequence
AFTER	Relating to order or sequence

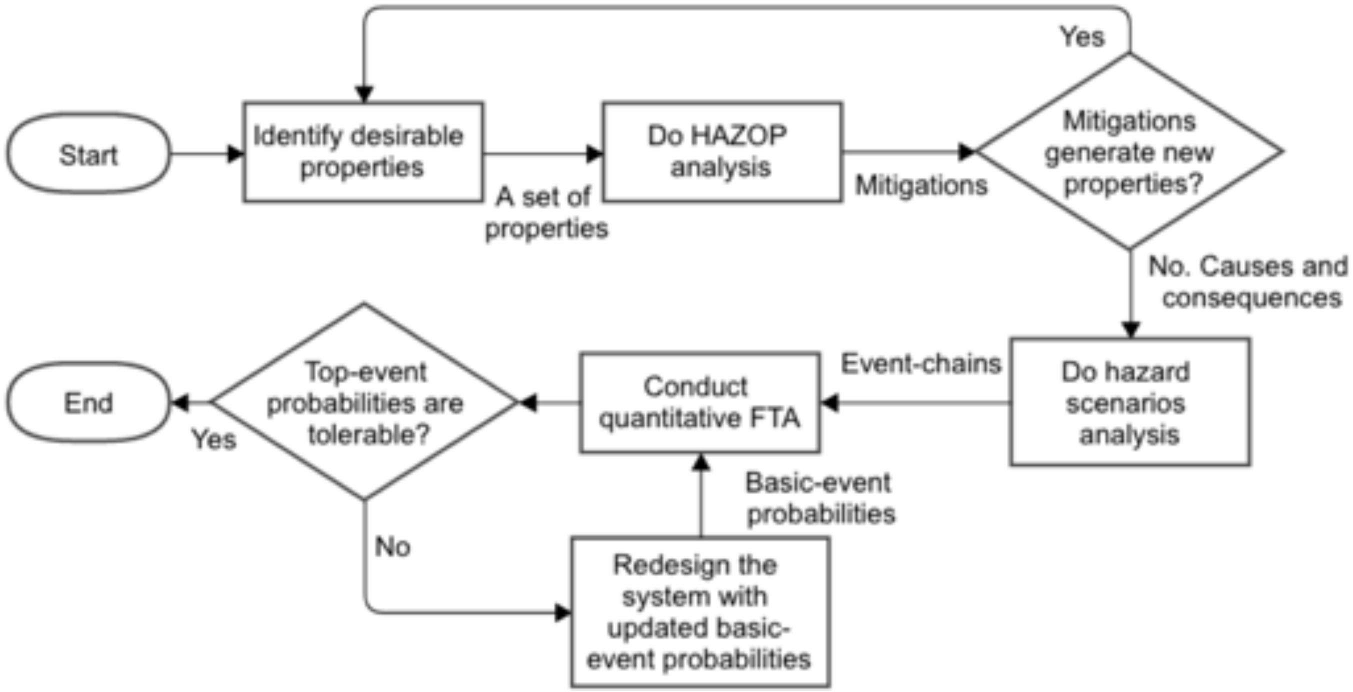
- Expected: EB must apply a maximum braking command to the engine.
- NO OR NOT: EB does not generate any braking command.
- LESS: EB applies less than max. braking.
- LATE: EB applies max. braking but after a delay of 2 seconds.
- REVERSE: EB generates an acceleration command instead of braking.

HAZOP Exercise: Lane Keeping Assist

Guide Word	Meaning
NO OR NOT	Complete negation of the design intent
MORE	Quantitative increase
LESS	Quantitative decrease
AS WELL AS	Qualitative modification/increase
PART OF	Qualitative modification/decrease
REVERSE	Logical opposite of the design intent
OTHER THAN / INSTEAD	Complete substitution
EARLY	Relative to the clock time
LATE	Relative to the clock time
BEFORE	Relating to order or sequence
AFTER	Relating to order or sequence

- **Component:** ML model for lane detection
  - **Expected behavior (SPEC):** Given a sensor image of the ground, the ML model detects the presence/absence of lane markings
- Apply HAZOP guidewords to identify different ways in which this component might deviate from expected behavior

HAZOP: Benefits & Limitations



- Encourages systematic reasoning about component faults
- Can be combined with FTA to generate faults (i.e., basic events in FTA)
- Potentially labor-intensive; relies on engineer's judgement
- Does not guarantee to find all failures (but this is true for every method!)

Design Patterns for Robustness

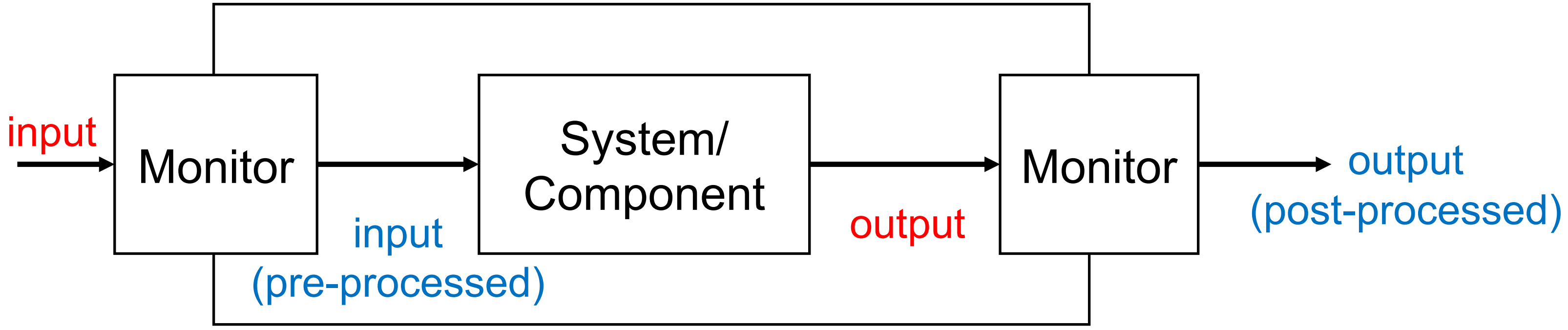
Design Patterns For Robustness

- *Having identified possible failure scenarios, how do we re-design the system to improve its robustness?*
- Many design patterns for robustness! We will cover:
  - Guardrails
  - Redundancy
  - Separation
  - Graceful degradation
  - Human in the loop
  - Undoable actions

Design Patterns For Robustness

- **Guardrails**
- Redundancy
- Separation
- Graceful degradation
- Human in the loop
- Undoable actions

Guardrails



- **Goal:** Protect a system/component from unexpected inputs or faulty outputs
- **Input monitor**: Check for an unexpected or potentially risky input
  - If unwanted input is detected, discard or pre-process it to a safe value
  - **Goal**: Improve robustness against **external** faults
- **Output monitor**: Check for a potentially faulty output
  - If fault is detected, discard or post-process it to a safe value
  - **Goal**: Improve robustness against **internal** faults

Type of Guardrail: Precondition Checking

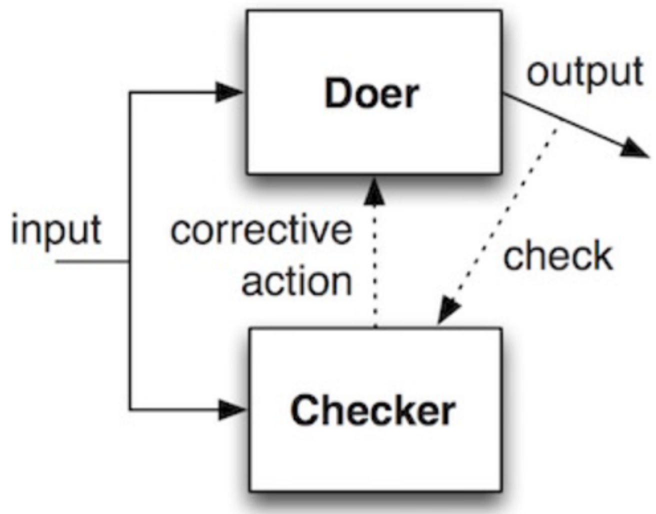
- **Precondition**: A condition that must be true of an input for a component to function correctly
- Identify and clearly document all **preconditions** over input parameters
- Check whether input satisfies the preconditions; if not, perform safe error handling
  - e.g., throw an error to the client and/or return a safe default response

```python
@app.route('/process_data', methods=['POST'])
def process_data():
    # Assuming JSON data is being sent in the request
    data = request.get_json()

    # Check if 'input_data' key exists in the JSON payload
    if 'input_data' not in data:
        return jsonify({'error': 'Input data missing'}), 400

    input_data = data['input_data']

    # Check for unexpected inputs
    if not isinstance(input_data, str):
        return jsonify({'error': 'Unexpected input format. Expected string.'}), 400

    # Additional checks can be added based on the requirements of your application

    # Process the input data
    processed_data = process_input(input_data)

    return jsonify({'result': processed_data})
```

Type of Guardrail: Interlock

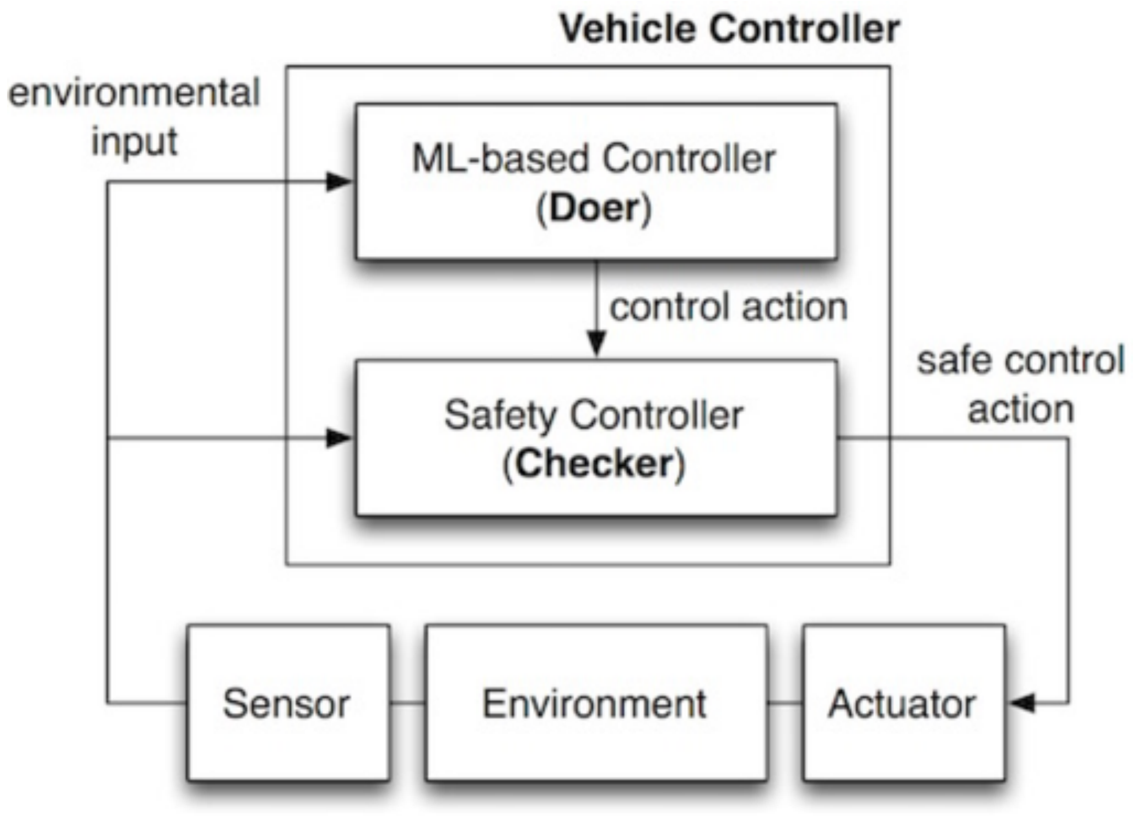
- Disable actions from being performed by a client/user under a certain context
- **Examples**
  - Disable the nurse from entering a radiation dose higher than a safe threshold
  - Disable an untrusted, third-party app from invoking critical OS functions
  - Disable an admin user of scheduling app from reading patient info in the central DB

Type of Guardrail: Doer-Checker Pattern



- **Doer**: Component carrying out a task
- **Checker**: Check the output by Doer and override it if it is considered faulty or unsafe
  - Checker should be well-tested and verified for reliability
  - Usually, this means Checker is simpler than Doer

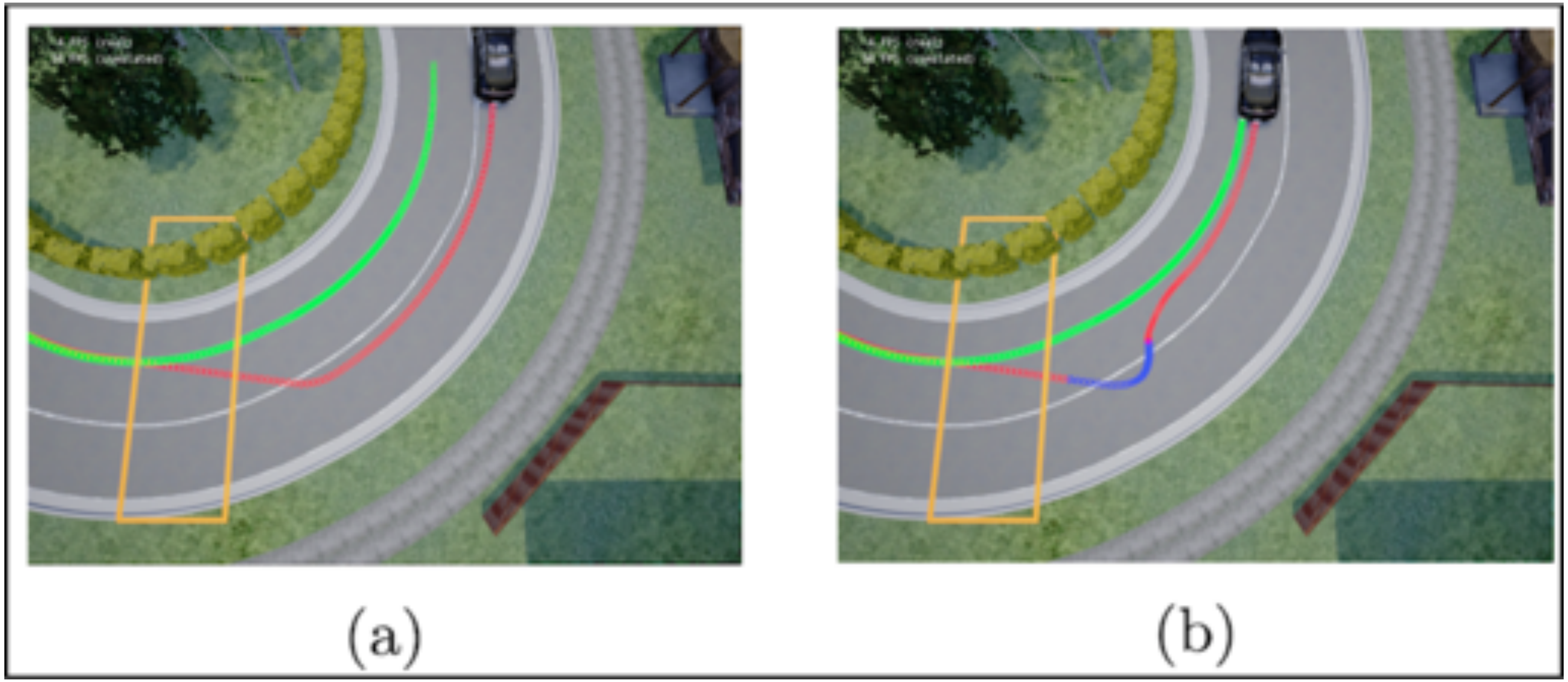
Doer-Checker Pattern: Example



- ML-based controller (**Doer**): Generate commands to steer the vehicle
  - Complex DNN; highly efficient
  - But poor performance over unexpected scenarios/inputs
- Safety controller (**Checker**): Check action from ML controller
  - Overrides with a safe default action if ML action is risky
  - Simpler, based on verifiable, transparent logic; performs conservative steering control

Runtime-Safety-Guided Policy Repair. Zhou et al. (2020)

Doer-Checker Pattern: Example

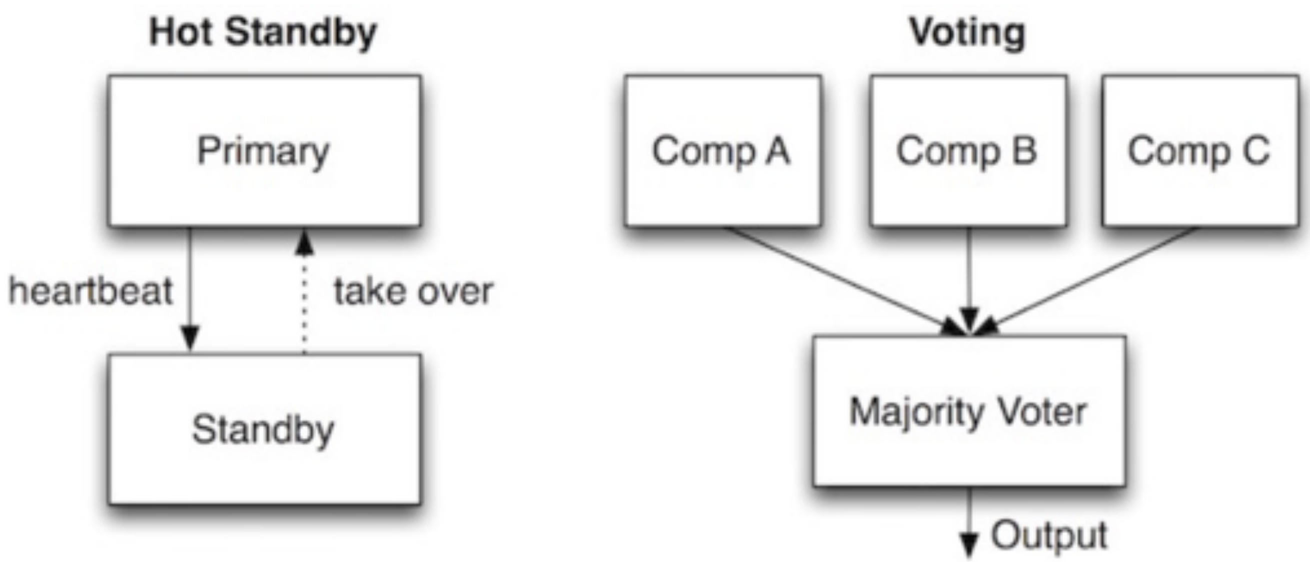


- **(a)** Yellow region: Slippery road, ignored by ML -> Causes loss of traction
- **(b)** Checker: Monitor detects lane departure; overrides ML with a safe steering command

Design Patterns For Robustness

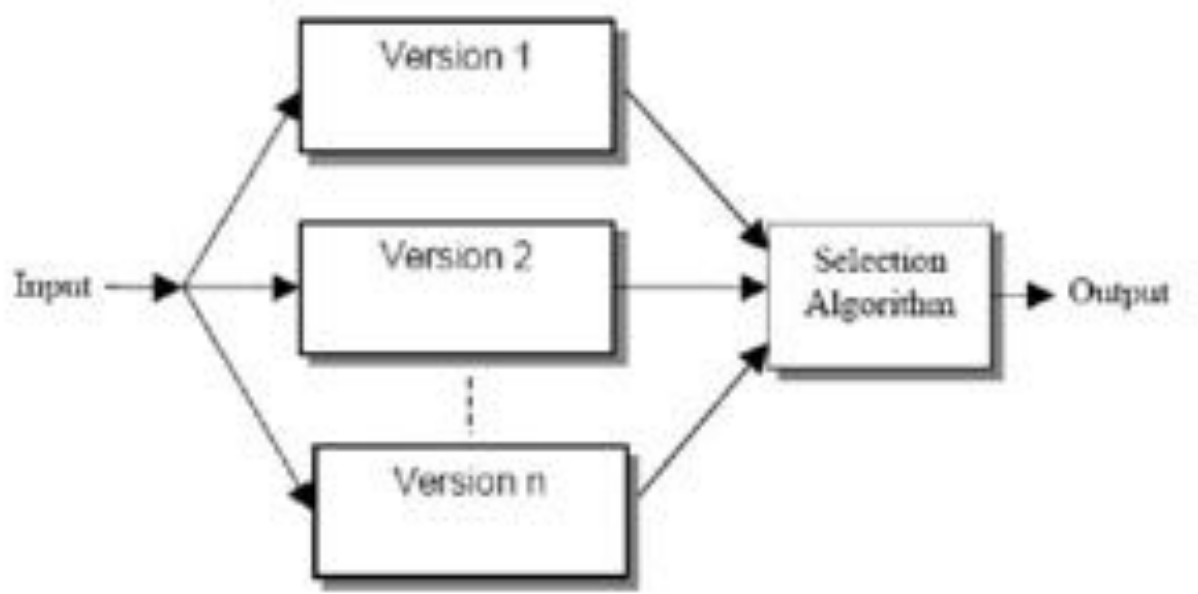
- Guardrails
- **Redundancy**
- Separation
- Graceful degradation
- Human in the loop
- Undoable actions

Redundancy



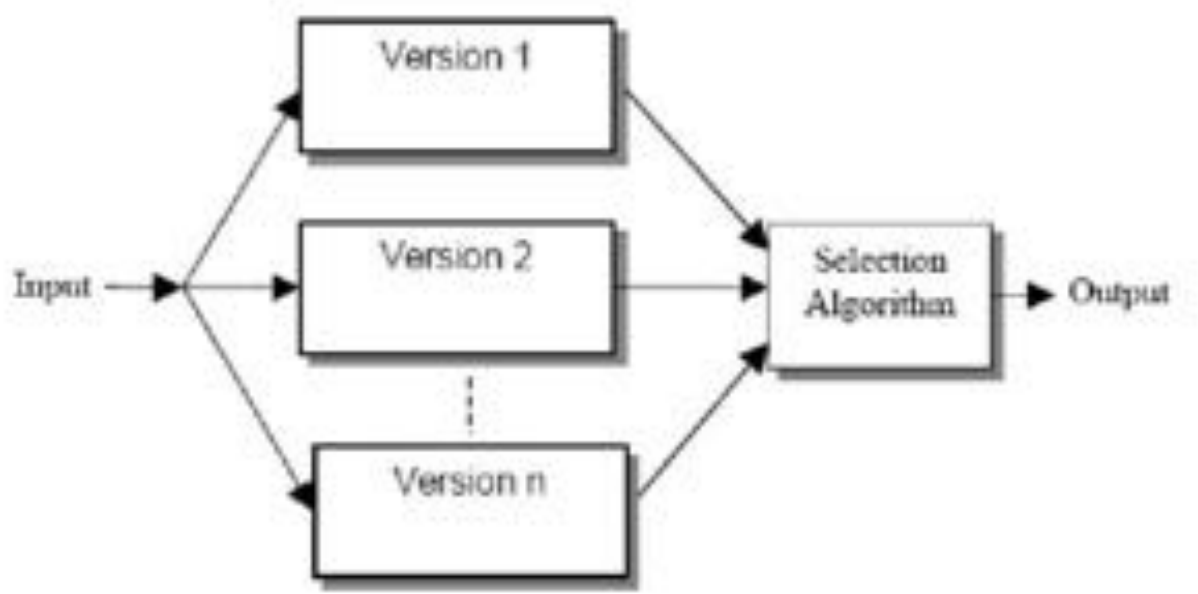
- **Goal**: If a component fails, continue to provide the same service
- Use redundant components to detect and/or respond to a fault
- Effective only if redundant components fail **independently**
- Common types of redundancy
  - **Hot Standby**: Standby watches & takes over when primary fails
  - **Voting**: Select the majority decision from multiple components

SW Redundancy: N-Version Programming



- Create different versions of a program from a shared specification
- Deploy them in parallel and take their majority or merge as final output
- **Approach**: Achieve independence through diversity in implementations
  - Developed by different teams, using different languages, libraries, and algorithms
- **Q. How well does this work in practice? What are its potential downsides?**

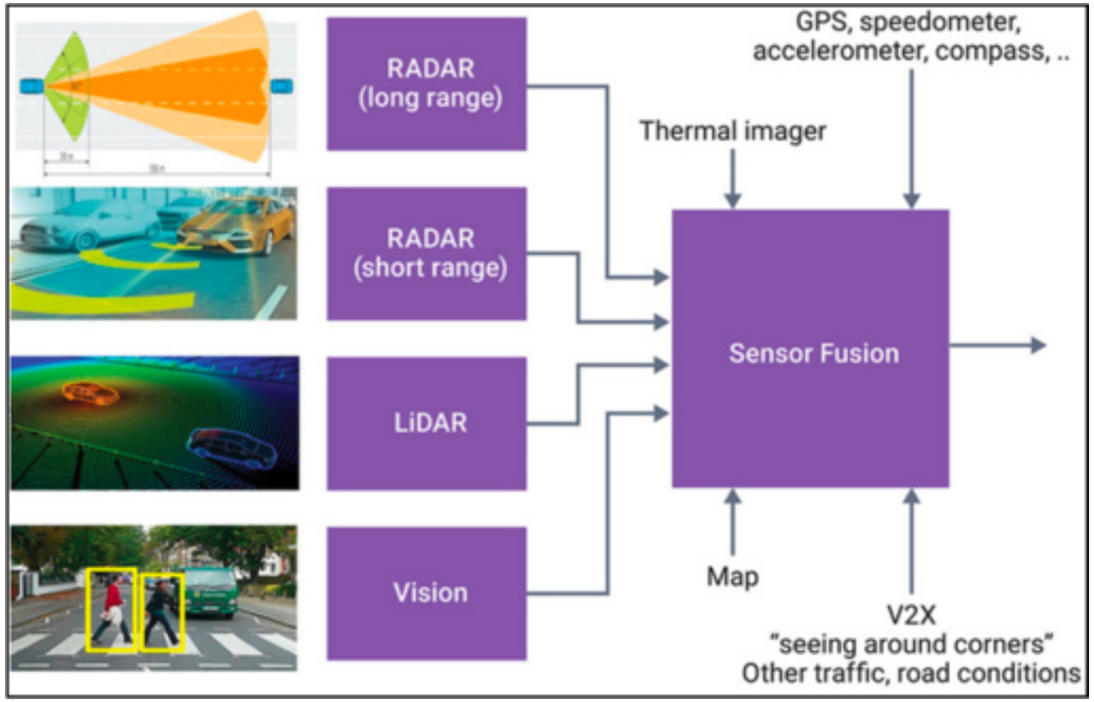
N-Version Programming: Limitations



- But in practice, independence of failures is rarely achieved
- Different teams make similar types of mistakes when working with the same specification!
- Overall, little improvement in reliability for high cost of developing & maintaining multiple versions

An experimental evaluation of the assumption of independence in multi-version programming. Knight & Leveson (1986)

Redundancy Example: Sensor Fusion

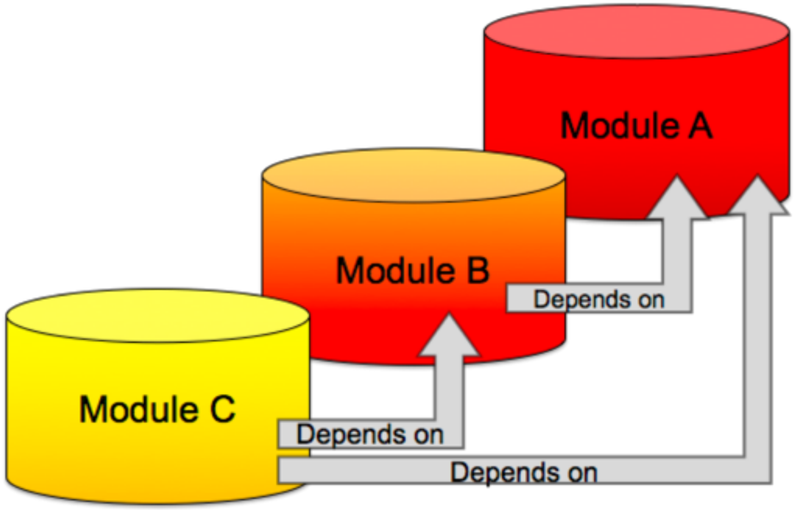


- Combine data from a wide range of sensors
- Provides partial information even when some sensor is faulty
- A critical part of modern autonomous systems
- **Q. Why does this approach work?**

Design Patterns For Robustness

- Guardrails
- Redundancy
- **Separation**
- Graceful degradation
- Human in the loop
- Undoable actions

Recall: Coupling



- **Coupling**: Component A is coupled to B (or “A depends on B”) if a change or a fault in B affects the correct functioning of A
- In general, loose coupling is desirable: If A does not depend on B, then B can be changed without affecting A
- Conversely, tight, unnecessary coupling is usually bad: If A depends on B, and B changes or fails, then A could also fail!

Failures due to Bad Coupling: Examples

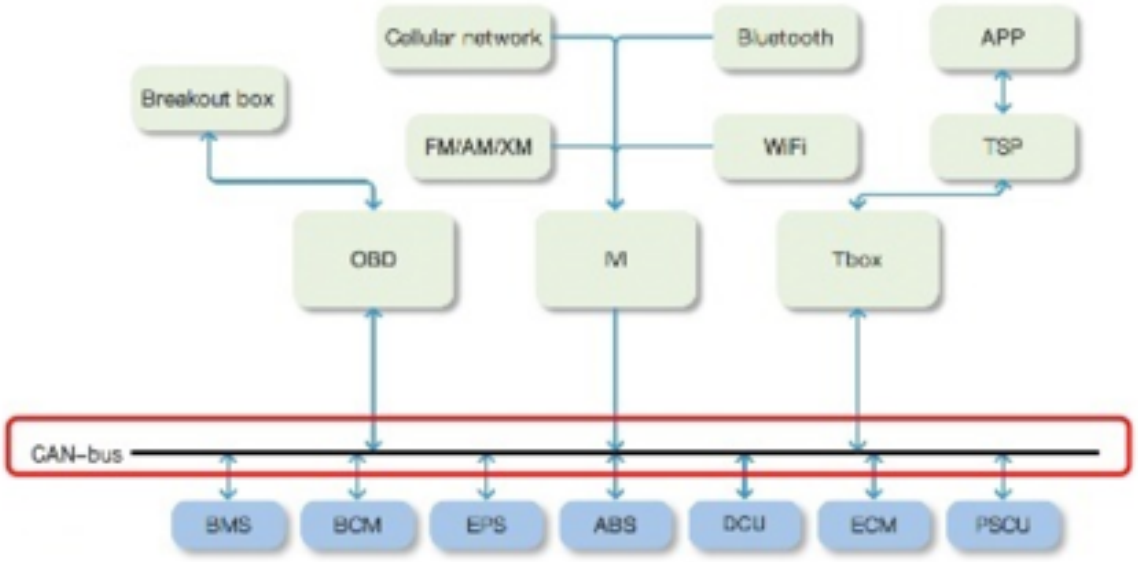
- **USS Yorktown, 1997**
  - Bad data entered into spreadsheet
  - Divide-by-zero crashes entire network
  - Ship dead in water for 3 hours

Failures due to Bad Coupling: Examples

- USS Yorktown, 1997
  - Bad data entered into spreadsheet
  - Divide-by-zero crashes entire network
  - Ship dead in water for 3 hours
- **Swissair Flight 111, 1998**
  - In-flight entertainment (IFE) shared wiring with main systems
  - Overheats & causes a widespread fire
  - 229 passengers killed

Failures due to Bad Coupling: Examples

- USS Yorktown, 1997
  - Bad data entered into spreadsheet
  - Divide-by-zero crashes entire network
  - Ship dead in water for 3 hours
- Swissair Flight 111, 1998
  - In-flight entertainment (IFE) shared wiring with main systems
  - Overheats & causes a widespread fire
  - 229 passengers killed
- **Automotive Systems**
  - Main components connected through a common CAN bus; no access control
  - Can control brake/engine by playing a CD with malicious music files



Comprehensive Experimental Analyses of Automotive Attack Surfaces. Checkoway et al. (2011)

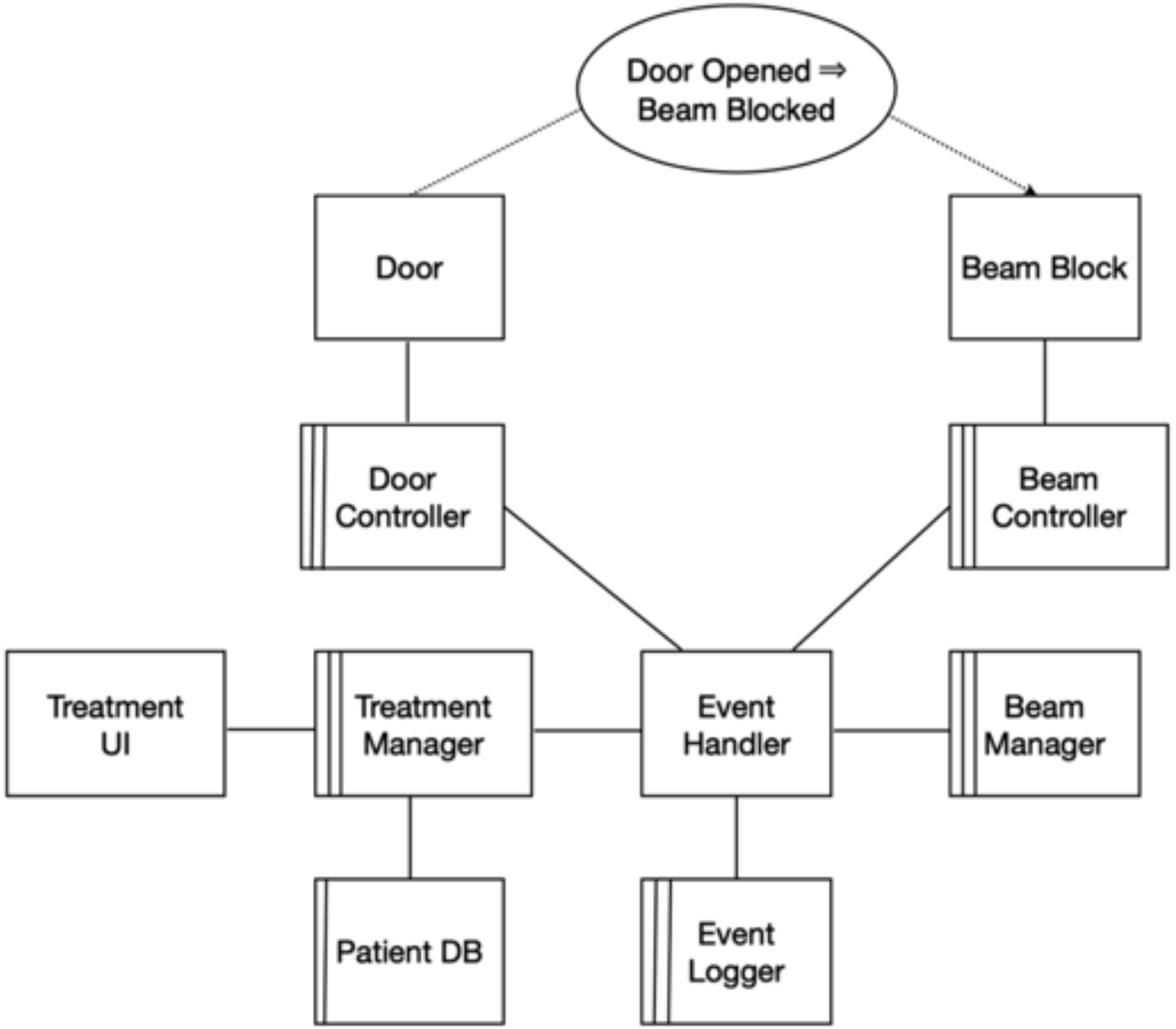
Separation

- **Principle**: A component that performs a high-critical (HC) function should not depend on an unreliable component (UC)
- What makes a component unreliable?
  - Complex or black-box component: Difficult to test or analyze
  - Responsible for multiple functions: More possible faults (recall: single-responsibility principle)
  - Receives inputs from unknown, external sources
- **Goal**: Remove or reduce dependency between HCs and UCs
- Construct a component diagram and identify the set of components that are responsible for achieving a high-critical requirement
- If these include UCs, re-design the system to remove them from the set

Separation Example: Radiation Therapy

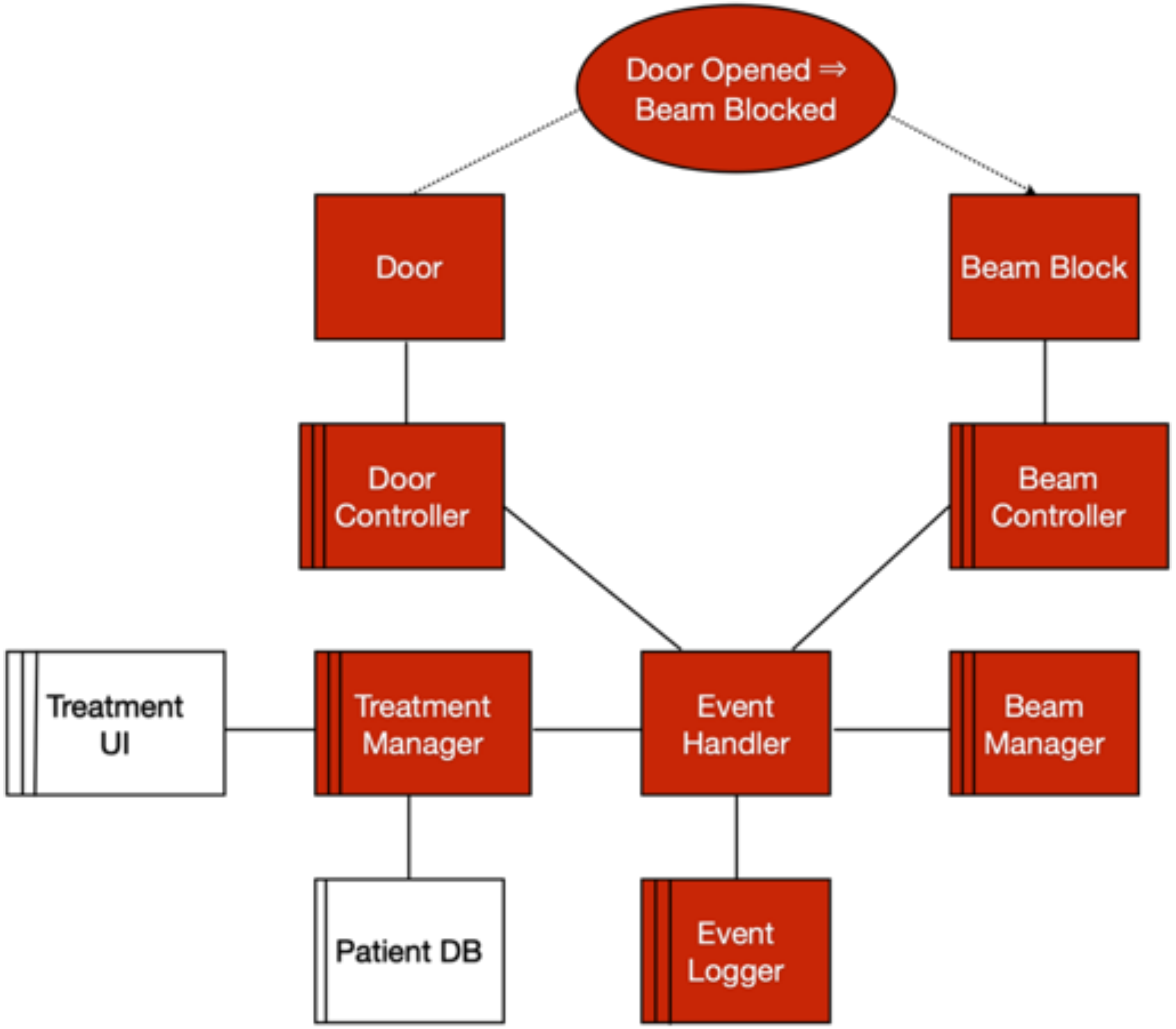
Radiation Therapy: Safety Requirement

"If door is opened during treatment, immediately stop the radiation by inserting the beam block"



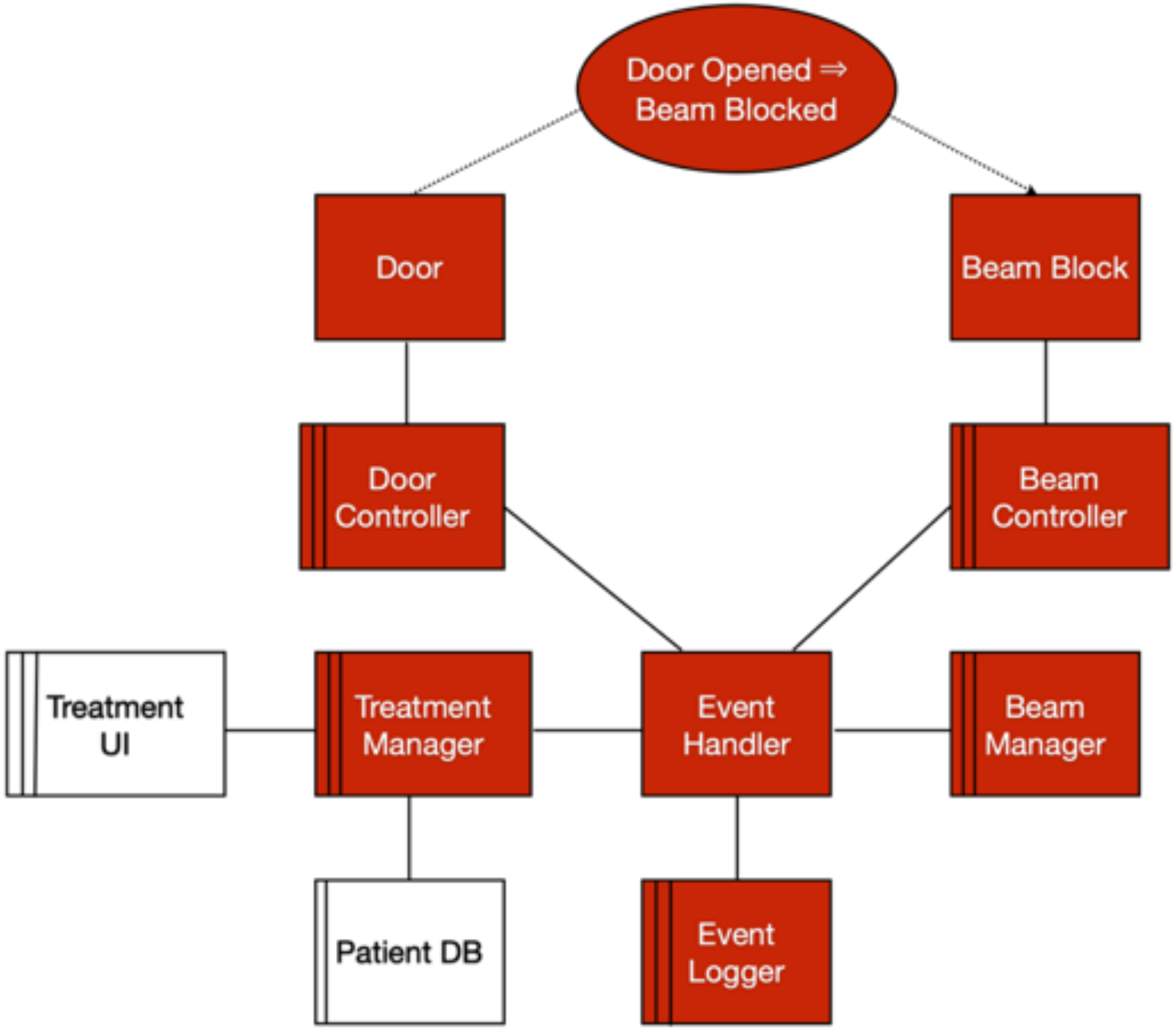
Component Responsibilities

- **Event Handler**: Generic pub-sub framework, handles all messages within the system
- **Event Logger**: Logs every message sent & received over the pub-sub network
- **Treatment Manager**: Receives sensor input from Door Controller and send instruction to Beam Manager
- **Beam Manager**: Send command to Beam Controller



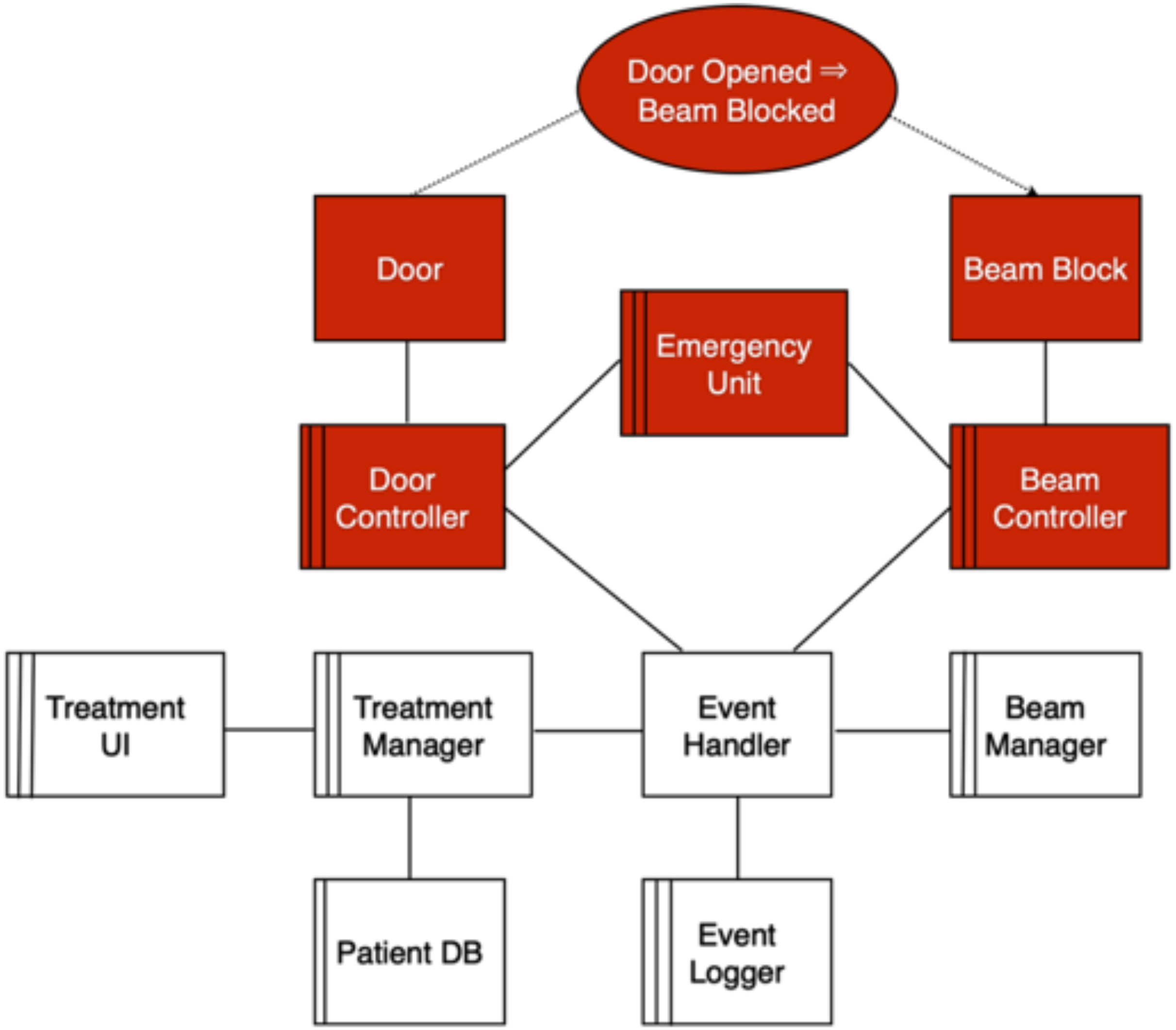
Reliable Components?

- **Event Handler**: Little control over timing; possible delay under heavy traffic
- **Event Logger**: May throw an error if the disk is full
- **Treatment Manager**: Also handles requests from the UI, put into the same queue as other requests
- To ensure the requirement, we need to rely on these components not failing!



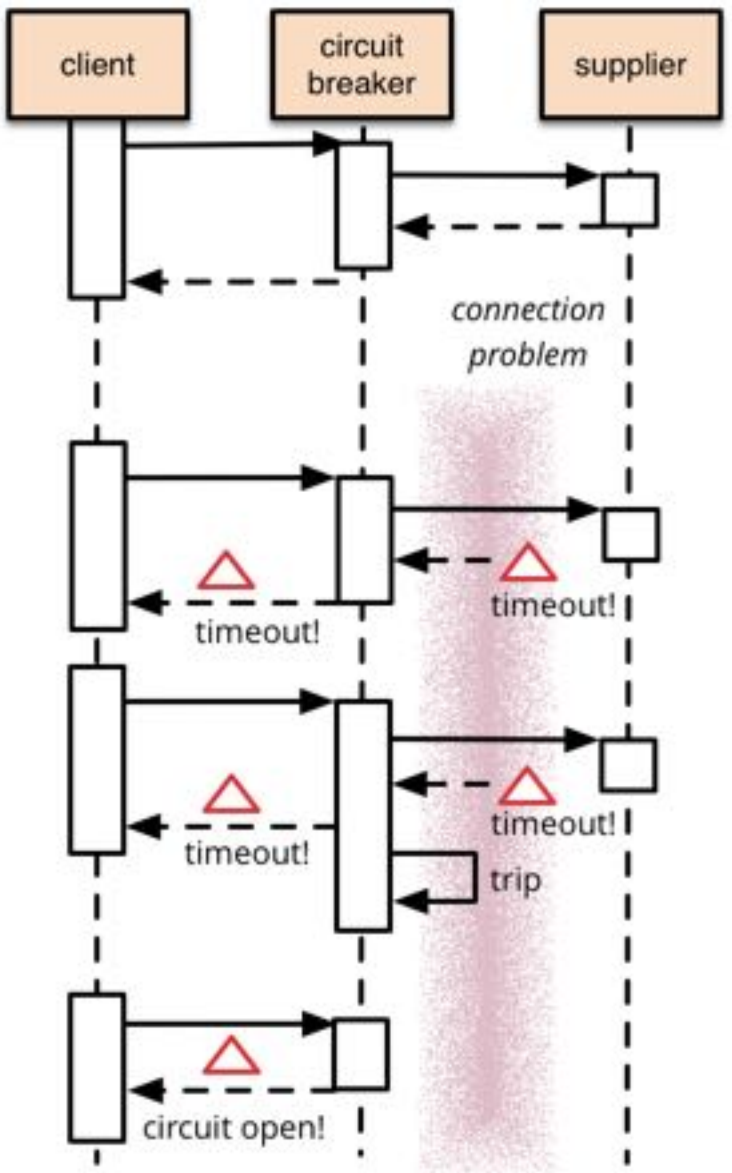
Separation: Trusted Computing Base (TCB)

- **TCB**: A set of components dedicated to ensuring critical requirements
  - Should ideally be small, testable, and isolated
- Emergency Unit serves a single purpose and is much simpler; can be made reliable
- Can't eliminate all risk of failure, but significantly reduce it
- However, also makes the overall system more complex and costly



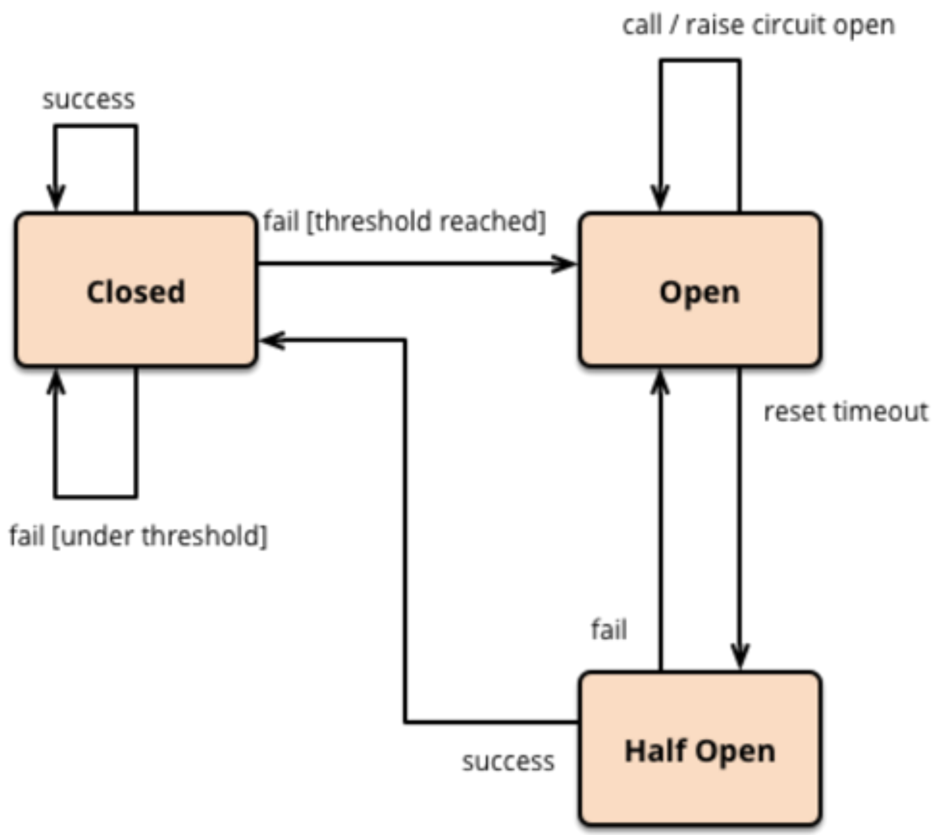
Separation: Circuit Breaker

- **Goal**: Prevent cascading failures by removing a connection from a failed component
- **Circuit breaker**: A wrapper between a client & a component that might fail ("supplier")
- If the failure persists, **"trip"** the circuit breaker by preventing further connections



Separation: Circuit Breaker

- If the failure persists, “**trip**” the circuit breaker by preventing further connections
  - Threshold for # retries before tripping
- After a reset timeout, try to reach the supplier again
  - If successful, “**close**” the breaker and allow the client to connect again
- Client must implement its own logic for dealing with situations when the breaker is open



https://martinfowler.com/bliki/CircuitBreaker.html

Design Patterns For Robustness

- Guardrails
- Redundancy
- Separation
- **Graceful degradation**
- Human in the loop
- Undoable actions

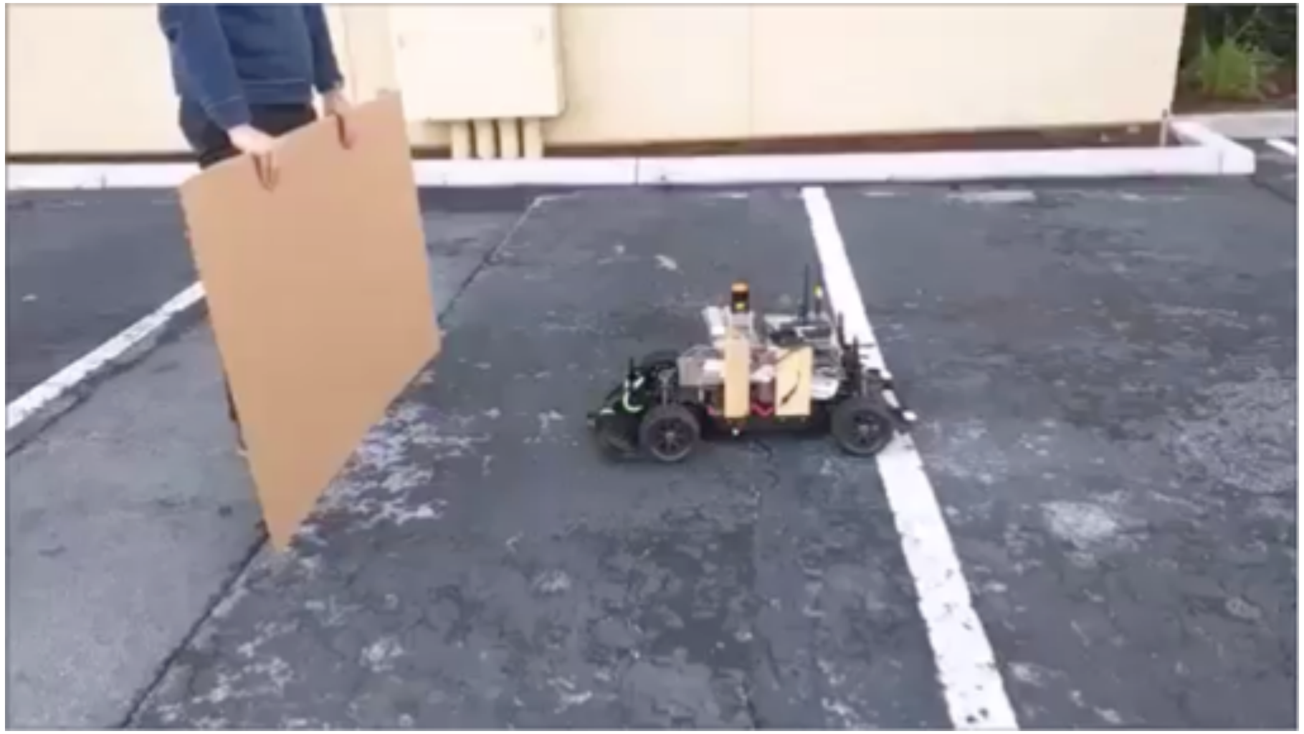
Graceful Degradation (Fail-soft)

- **Goal**: When one or more component fails, temporarily reduce system functionality or performance of the system
  - instead of shutting down the entire system (**fail-safe**)
- **Approaches**: When a component fails,
  - Return a pre-determined, degraded response to client
  - Disable the service but continue to offer other services

Graceful Degradation: Examples

- **Content streaming**: In a network failure or congestion, stream a low-resolution version of a media file
- **Web page rendering**: If certain Javascript libraries are missing on the client's machine, load a basic, HTML-only version
- **Denial-of-service (DoS) attack**: If a server becomes overwhelmed due to an attack, re-route the traffic to other available servers using a load balancer slower performance)
- **Buffering in a chat/e-mail client**: If a network connection is lost, buffer the messages and send them once it becomes available again (delayed delivery)
- **Q. Other examples?**

Graceful Degradation: Another Example

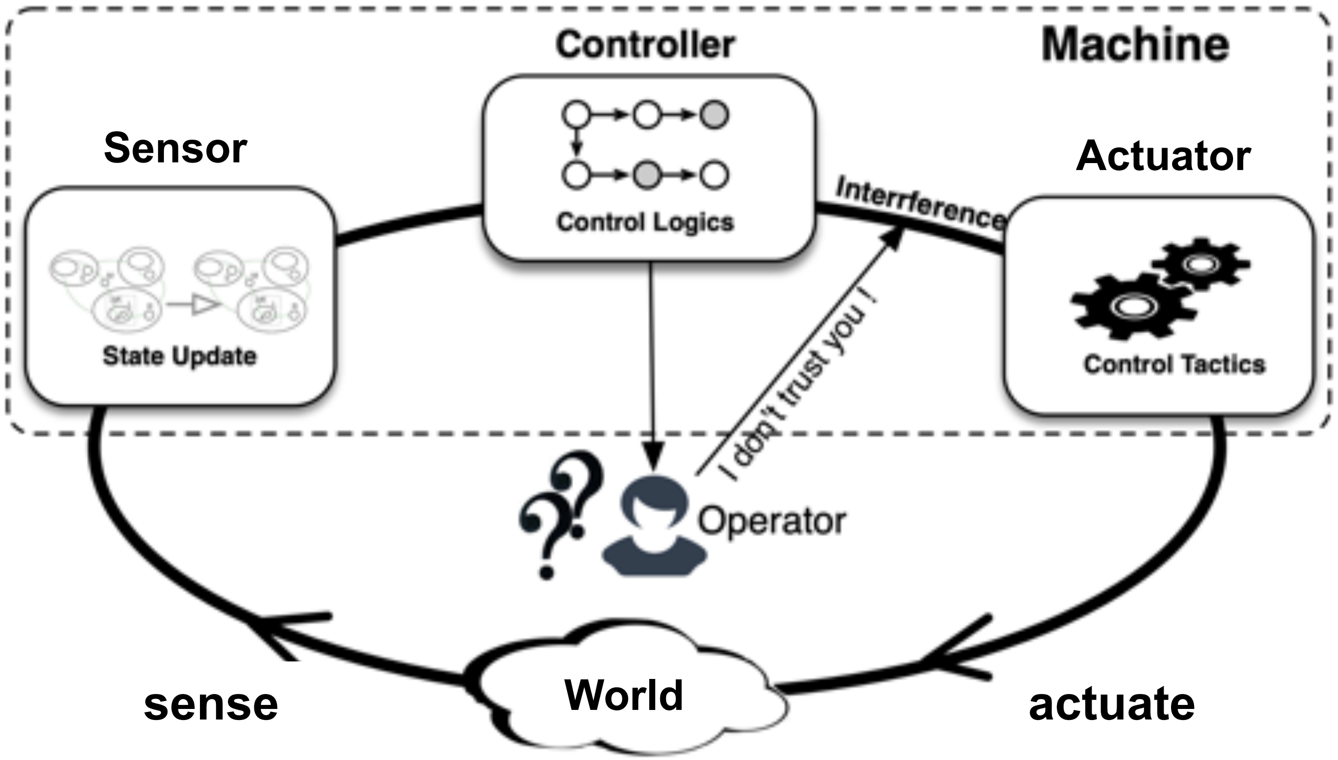


- Self-driving vehicle with multiple sensors (Lidar & camera)
- When a sensor fails, degrade performance but preserve safety by increasing distance to the leading object
- There is a **limit** on how far system can be degraded! When enough faults occur, fail safely by shutting down

Design Patterns For Robustness

- Guardrails
- Redundancy
- Separation
- Graceful degradation
- **Human in the loop**
- Undoable actions

Human in the Loop



- **Goal**: Prevent or recovery from system/component failures through human intervention
- An operator monitors the output of a component ("controller") and intervene if the output action is potentially faulty

Human in the Loop: Examples

- Remote operator for self-driving vehicles
  - Overtake in scenarios where the system (e.g., ML-based controller) is unable to make confident decisions

Human in the Loop: Examples

- Event monitoring & alerting
  - Monitor for certain events (e.g., workload spikes) and send alerts to an engineer for intervention
  - Several modern frameworks available (e.g., Prometheus, Grafana, Thanos)

Human in the Loop: Challenges

- Notification fatigue, complacency
  - After frequent alarms, human may ignore/take them less seriously
- Deciding when to allow or disallow intervention by human
  - Consider (slow) human reaction time: Does it make sense to rely on the human for a resolution?
  - **Recall**: Humans also make mistakes! Can we rely on them to carry out the task correctly?
- Mental model mismatch
  - Does the human have an accurate understanding of the system state when intervening?
  - (More on this in "Design for usability" lecture)

Design Patterns For Robustness

- Guardrails
- Redundancy
- Separation
- Graceful degradation
- Human in the loop
- **Undoable actions**

Undoable Actions

- **Goal**: Provide a way for the system to reverse the effect of an erroneous action
- Design the system to make certain (critical) actions undoable
- If the system reaches an undesirable state or at the request of a client/user, revert back to the previous desirable state
- **Challenges**
  - Not every action can be undone; some effects are irreversible
  - Undoing action adds complexity: Must keep track of a history of past actions and system states
  - Delayed undo: It may be too late before determining when an action should be reversed

Undoable Actions: Examples

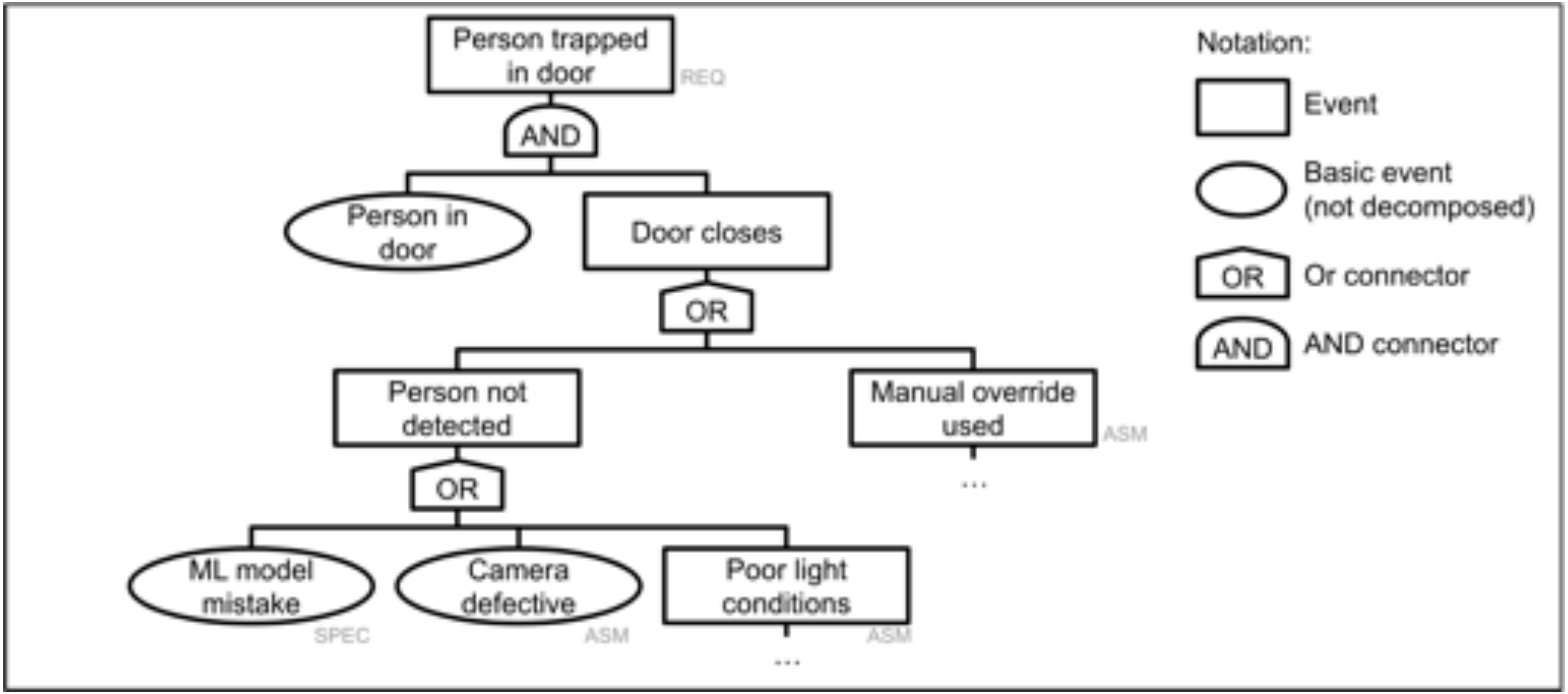
- **Version control systems**: Undo changes to codebase & revert back to a previous snapshot of a repository
- **Database transactions:** Rollback to a previous database state if a transaction fails; ensures integrity of the data
- **Graphics/text editors**: Undo previous editing actions (e.g., ”delete”)
- **E-mail client**: “Undo” send feature in Gmail (what is its limitation?)
- **Factory resets**: Mobile devices or computers, to remove effect of malware or data corruption
- **Q. Examples of systems where undoing an action is difficult/impossible?**

Exercise: Autonomous Train

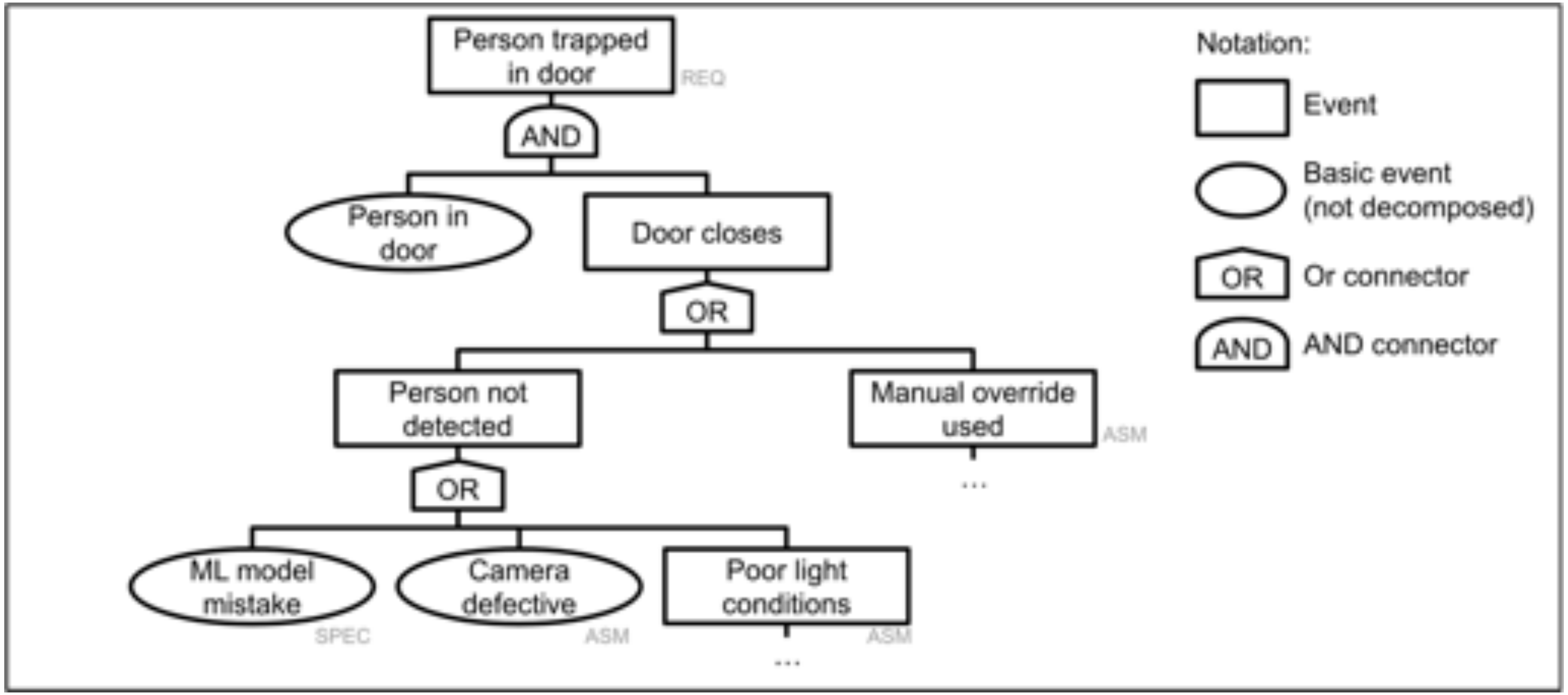
Patterns for Robustness
Guardrails
Redundancy
Separation
Graceful degradation
Human in the loop
Undoable actions

- **Requirements**: The train shall not depart all doors are closed. The train shall not trap people between the doors.
- ML-based system to detect people & control door closings
- Consider the failure scenarios identified earlier using FTA
- Design ways to improve its robustness using the patterns

Recall: FTA for Autonomous Train

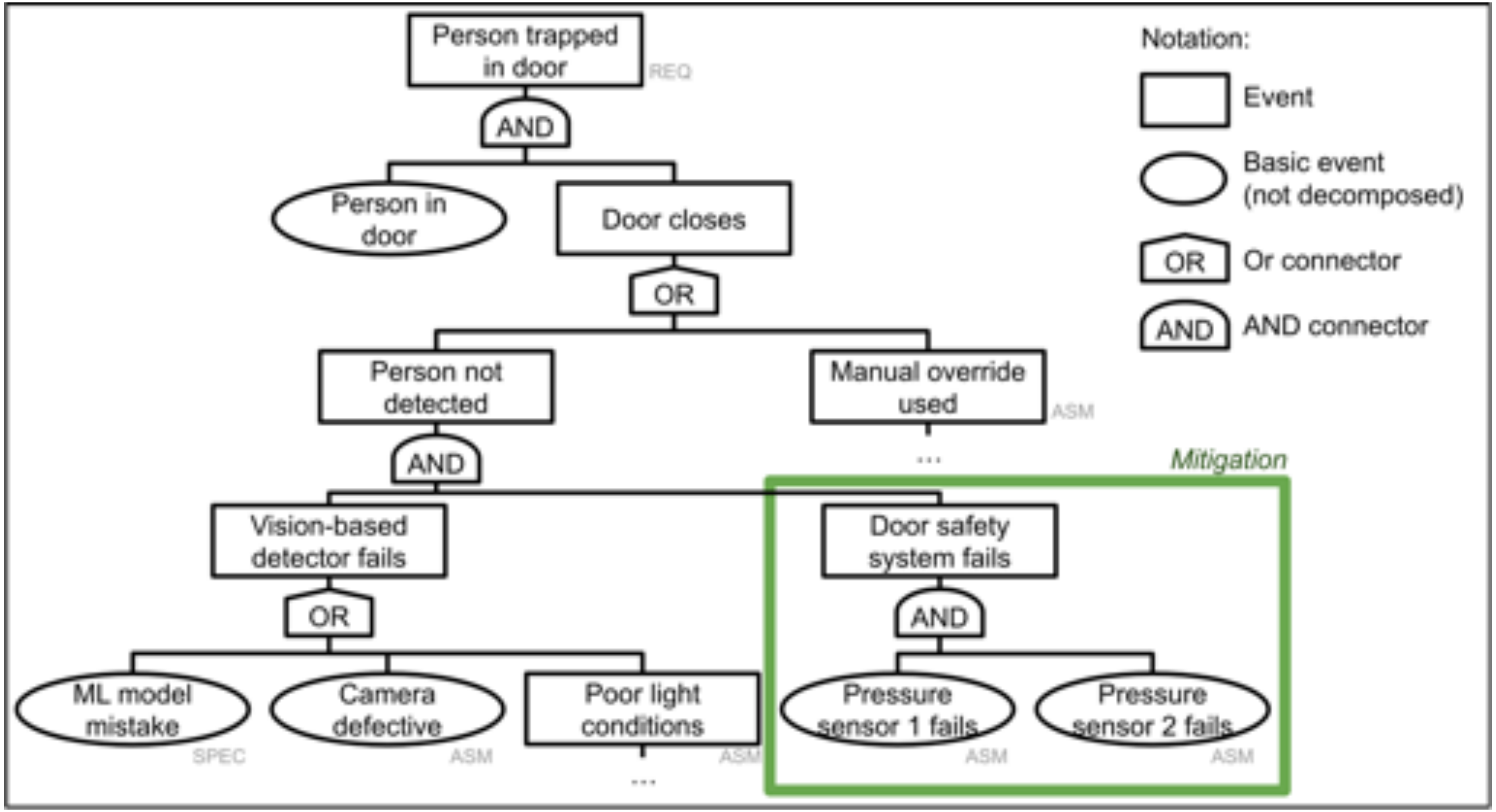


Robustness Improvement as Modifications to FTA



- Remove or reduce the likelihood of basic events
- Increase the size of cut sets by requiring additional basic events to occur

Adding Mitigations



Summary: Design Patterns for Robustness

- We talked about different patterns/strategies for improving robustness: Guardrails, redundancy, separation, graceful degradation, human in the loop, and undoable actions
- There's no silver bullet! Different strategies are suitable for different contexts and applications
- Each pattern will also increase the overall system complexity and add to the development cost
- In practice, it is impossible to predict and prevent every possible failure
  - Failure analysis methods like FTA and HAZOP help, but also require domain knowledge
- But systematically thinking about possible failures & mitigations during the design is a critical step!
  - If you don't design for robustness, your system is unlikely to be robust by "accident"

Summary

- Exit ticket!