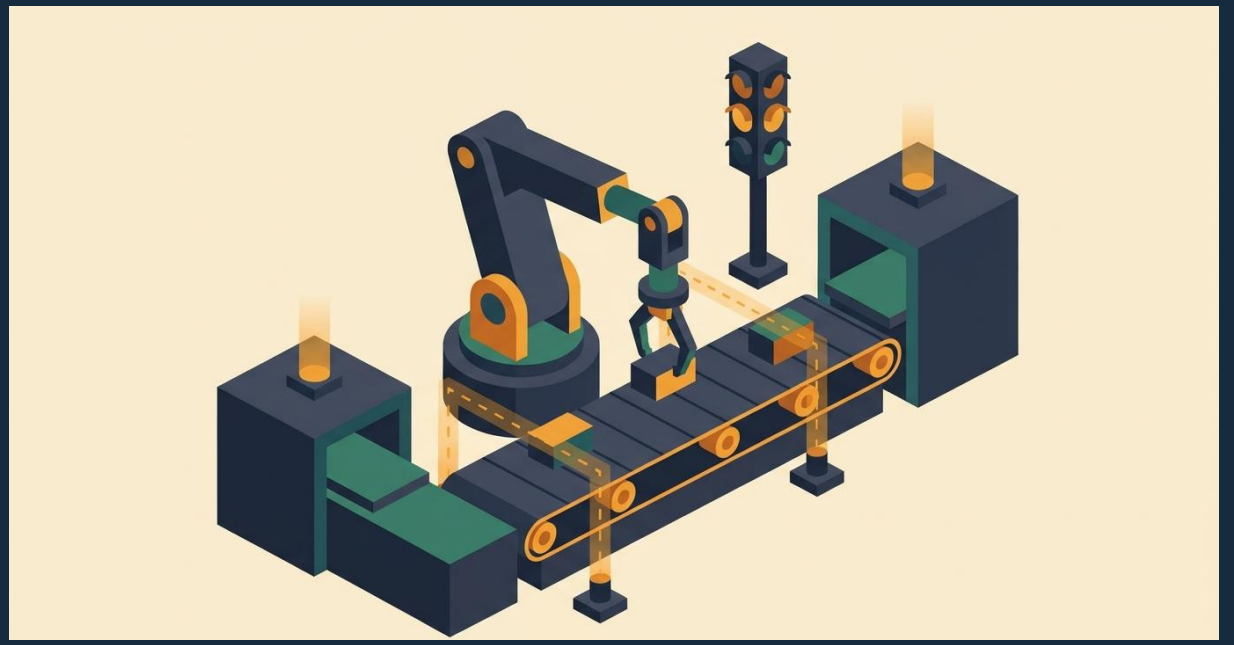
Third Year Electronic and Computer Engineering

Discrete Event Systems and Petri Nets



Modeling, Control, Observation, and Safety Verification in Ninety Minutes

Complete Course Summary: Four Modules on Automata and Supervisory Control plus Two Modules on Petri Nets

Six Connected Modules Guide Engineers From Models to Safe Automation

Each module answers a practical question about how modern automated systems work and stay safe.

Part 1: Languages

How do events form rules?

Part 2: Automata

How do machines connect?

Part 3: Control

How do supervisors stop errors?

Part 4: Observation

How do we see hidden faults?

Part 5: Petri Basics

How do tokens model parallelism?

Part 6: Petri Analysis

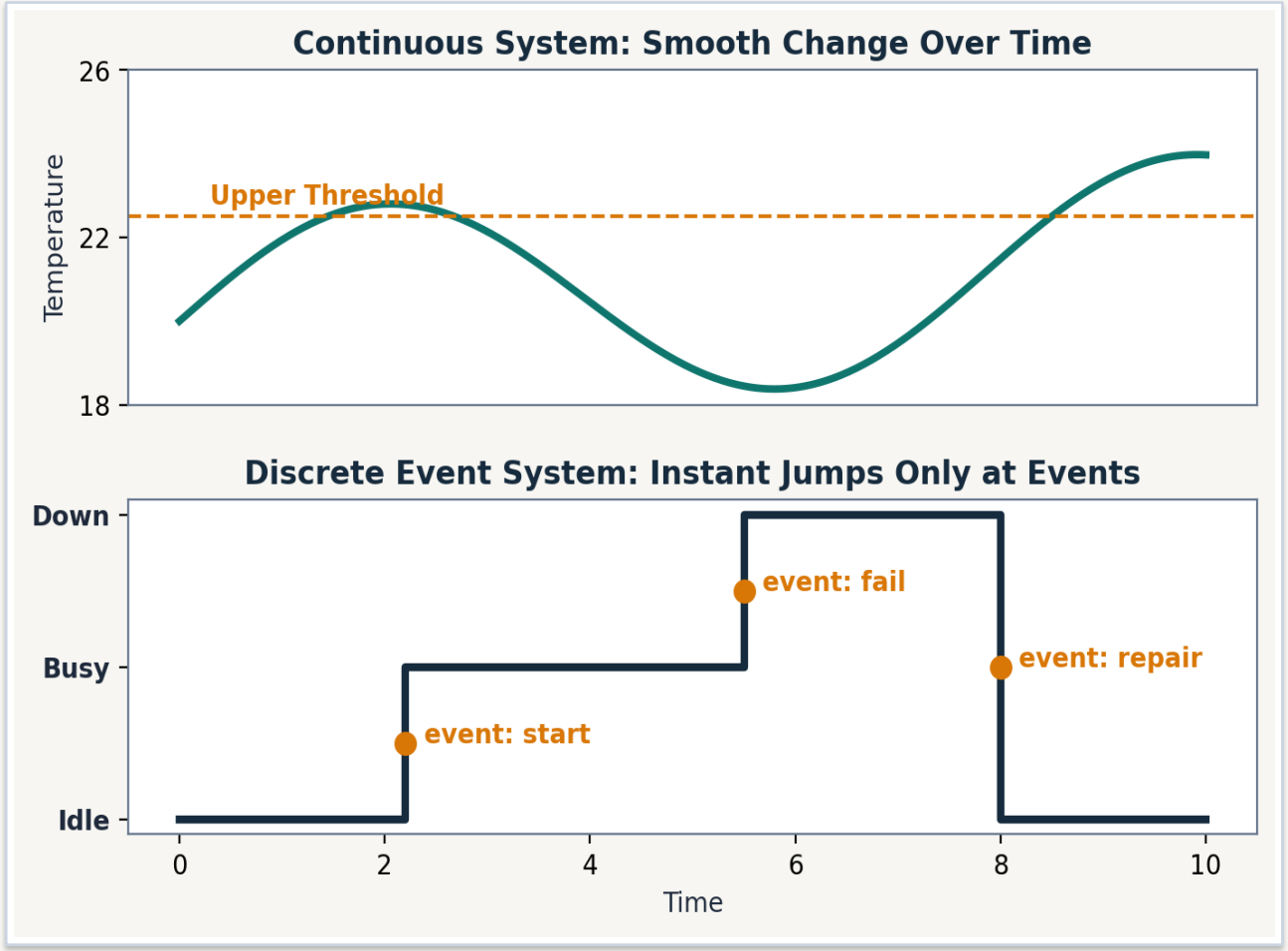
How do we prove zero deadlocks?

Continuous Systems Flow Smoothly While Discrete Systems Jump

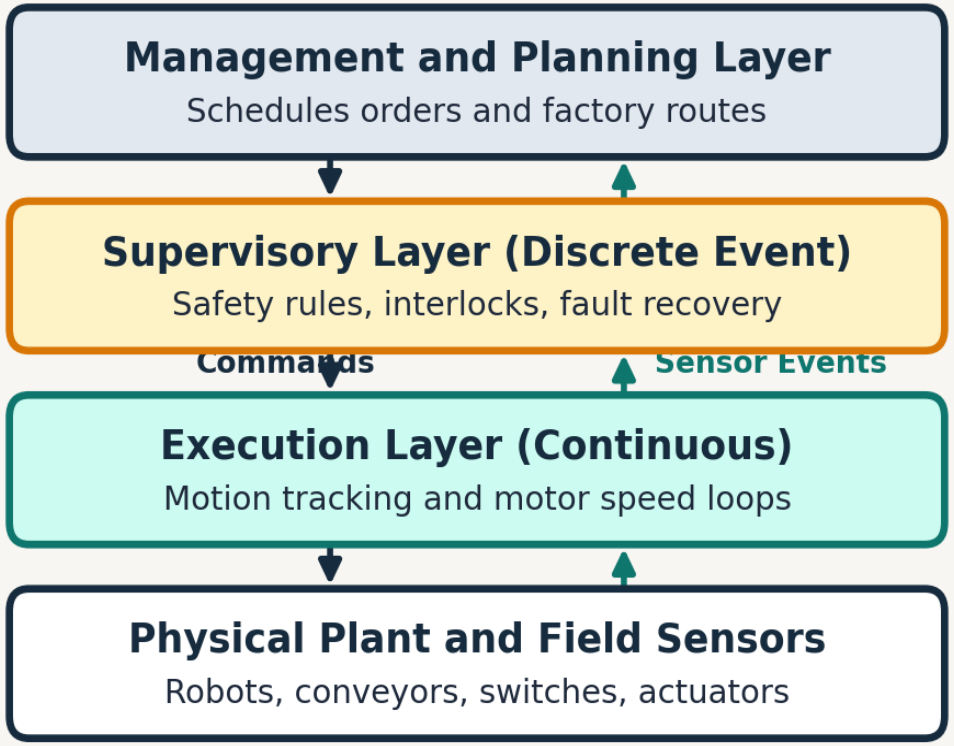
Engineers separate smooth physical signals from sudden event jumps.

- Continuous states change smoothly over time
- Discrete states wait until events happen
- Transitions jump instantly between discrete states
- Events arrive asynchronously without fixed clocks
- Examples include factories, networks, thermostats

Key idea: Discrete models care about event order, not motor speed.



Discrete Event Control Coordinates Continuous Machine Loops Above



Discrete supervisors do not replace motion controllers; they guide them.

- Management layer schedules factory production tasks
- Supervisory layer enforces safety and recovery
- Continuous layer tracks smooth robot motion
- Physical plant holds motors and sensors
- Supervisor switches continuous controllers on safely

Key idea: Continuous loops keep stability; discrete supervisors keep global order.

Three Modeling Layers Answer Different Engineering Verification Questions

Choose logical, timed, or stochastic models depending on your engineering goal.

Modeling Layer	Main Focus	Mathematical Tools	Example Question
Logical (Untimed)	Event order and rules	Languages, Automata, Petri Nets	Can deadlock happen?
Timed (Deterministic)	Exact times and delays	Timed Automata, Timed Nets	Is five-second deadline met?
Stochastic (Random)	Random arrival and service	Markov Chains, Queue Models	What is average queue length?

Takeaway: This course focuses mainly on logical models for safety and coordination.

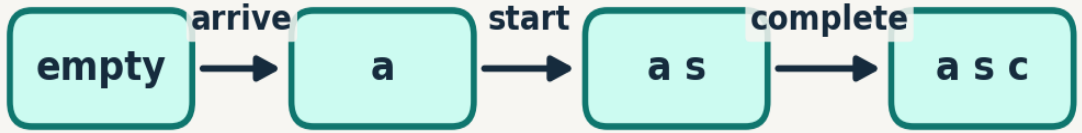
Event Sequences Form Valid Words Inside System Languages

We model system rules like a grammar over an event alphabet.

- Alphabet lists every possible system event
- Strings record observed event sequences clearly
- Empty string means zero events happened
- Language collects all allowed event traces
- Prefix closure represents step-by-step

Key idea: Real systems must pass every prefix before finishing a task.

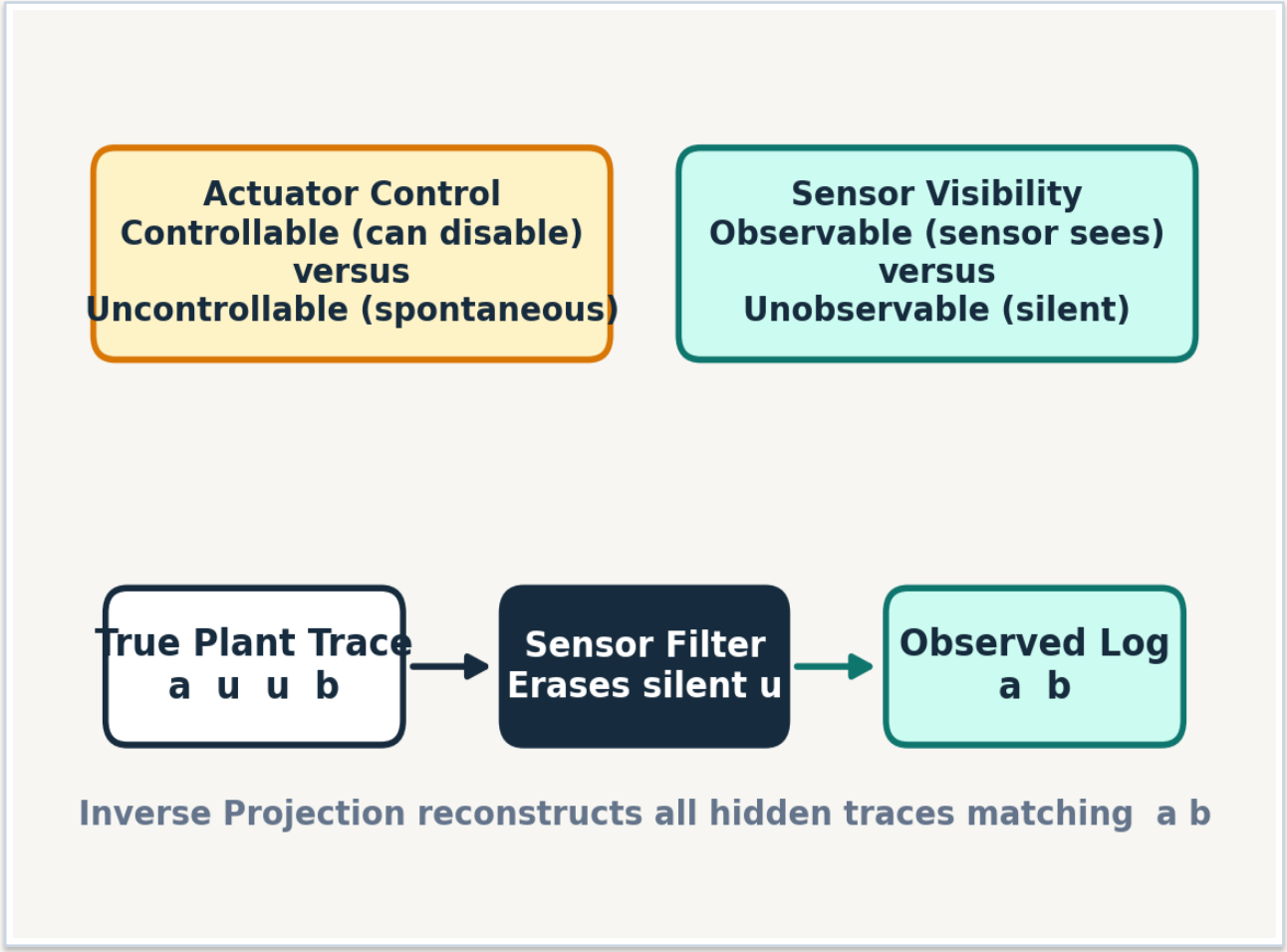
Physical Execution Builds Traces Step by Step



Every prefix of a valid trace is also valid (Prefix Closure)

Forbidden Trace: s c (cannot start empty buffer)

Sensors and Actuators Split Events Into Distinct Groups



Not every event can be stopped by a controller or seen by a sensor.

- Controllable events can be safely disabled
- Uncontrollable events happen spontaneously without warning
- Observable events trigger clear sensor signals
- Unobservable events happen silently inside machines
- Projection filter erases all silent events

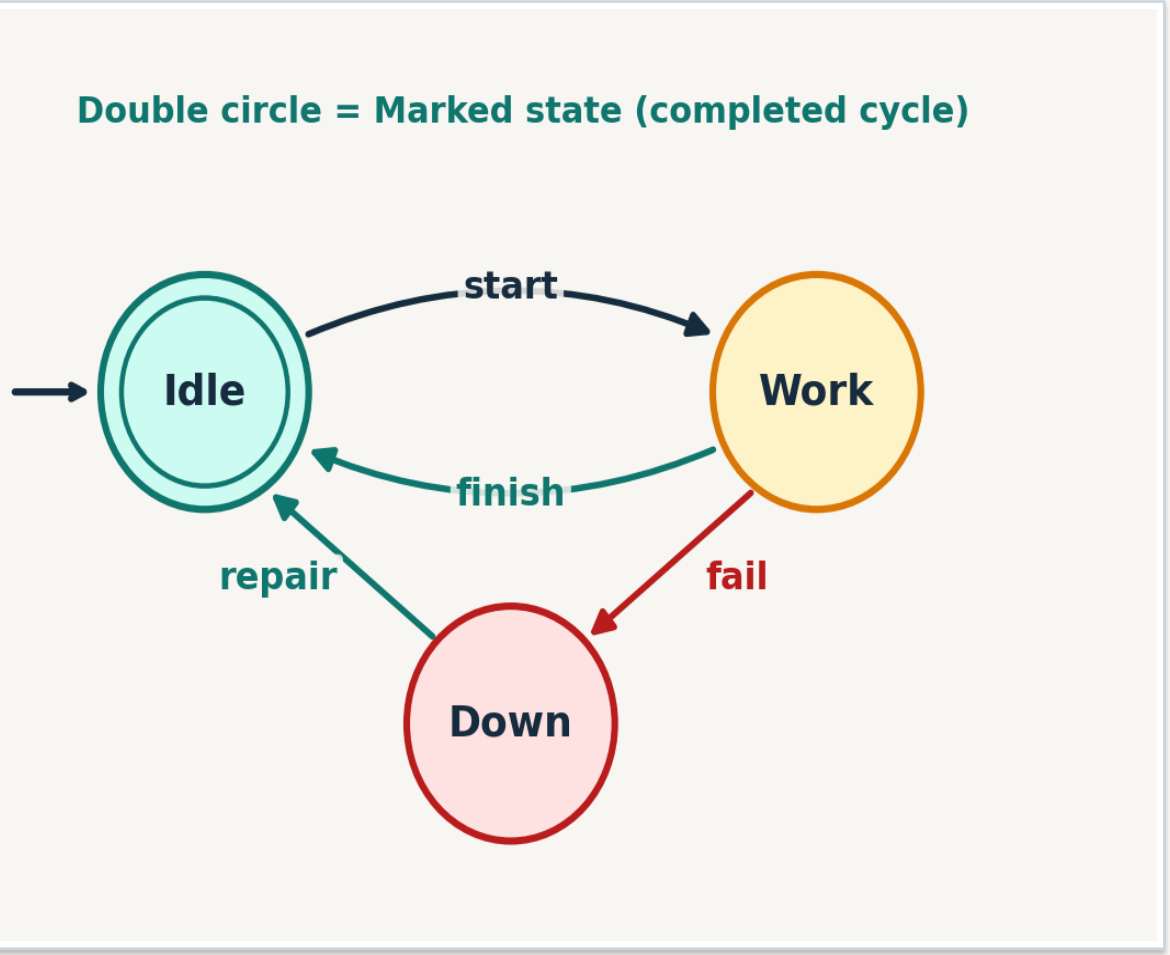
Key idea: Inverse projection reconstructs every hidden trace matching sensor logs.

Finite Automata Turn Infinite Languages Into Compact Graphs

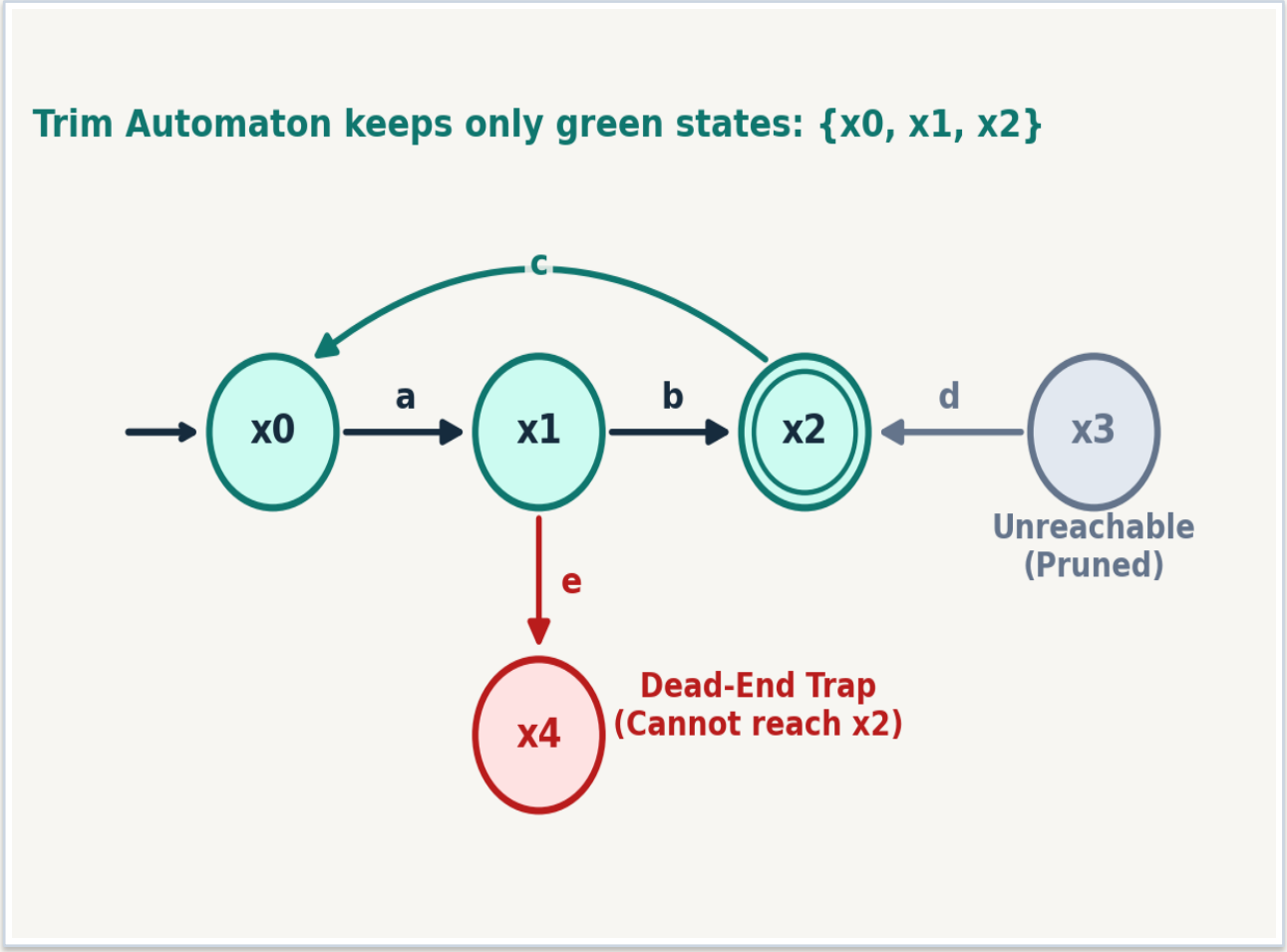
Pure lists cannot store infinite words, so we build state machines.

- Pure languages grow infinitely for cycles
- States summarize past history into circles
- Transition arrows show how events move
- Generated language captures all running traces
- Marked states represent completed work cycles

Key idea: Double circles mark safe resting states and finished tasks.



Pruning Removes Unreachable States and Dead-End Traps Together



Clean models keep only states that connect from start to goal.

- Forward search finds all accessible states
- Backward search finds all co-accessible states
- Unreachable states never occur from start
- Non-co-accessible states trap systems forever
- Trim operation keeps only useful states

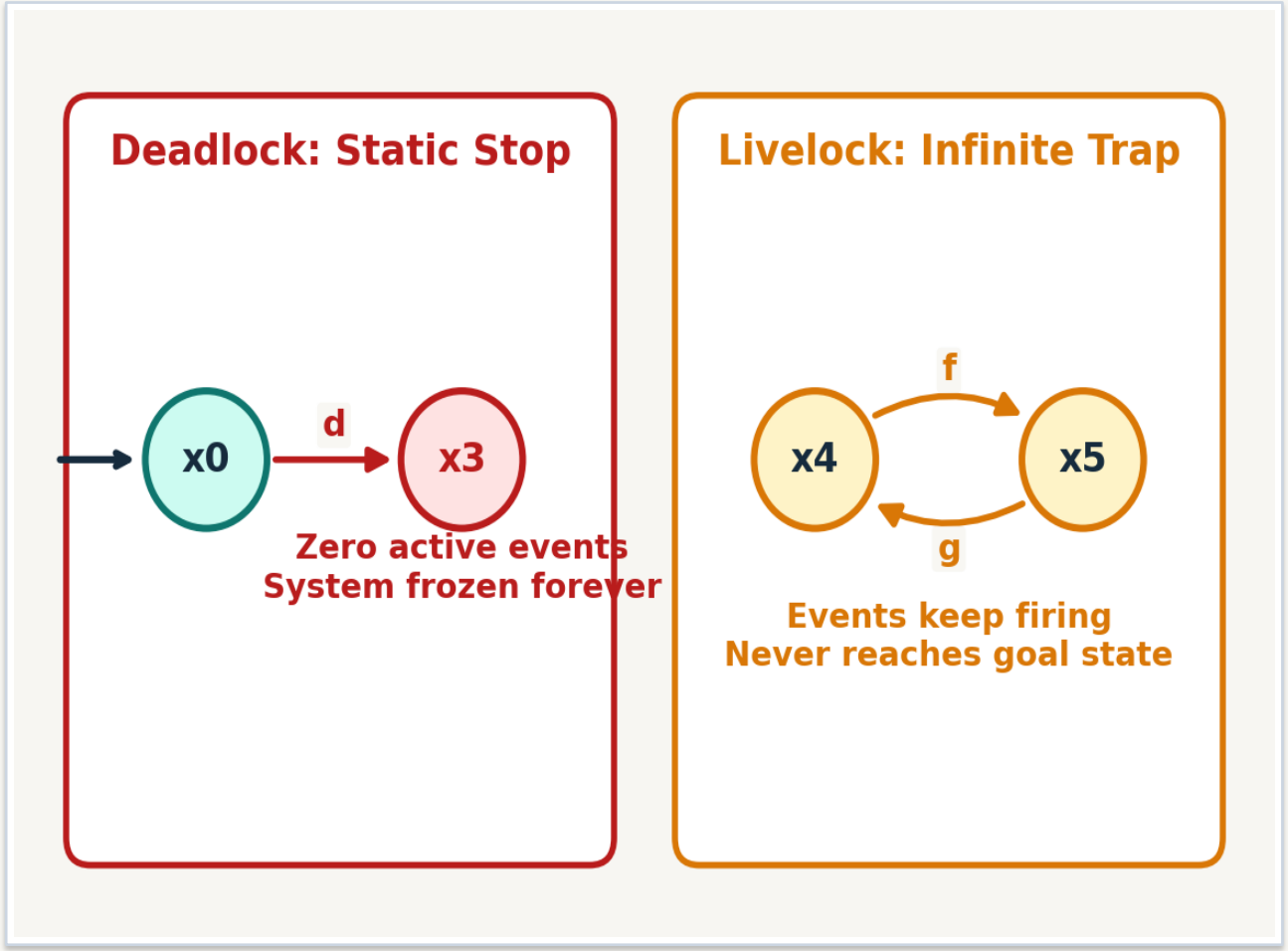
Key idea: A system is nonblocking only when every reachable state can finish.

Blocking Happens Through Static Deadlocks or Endless Livelock Loops

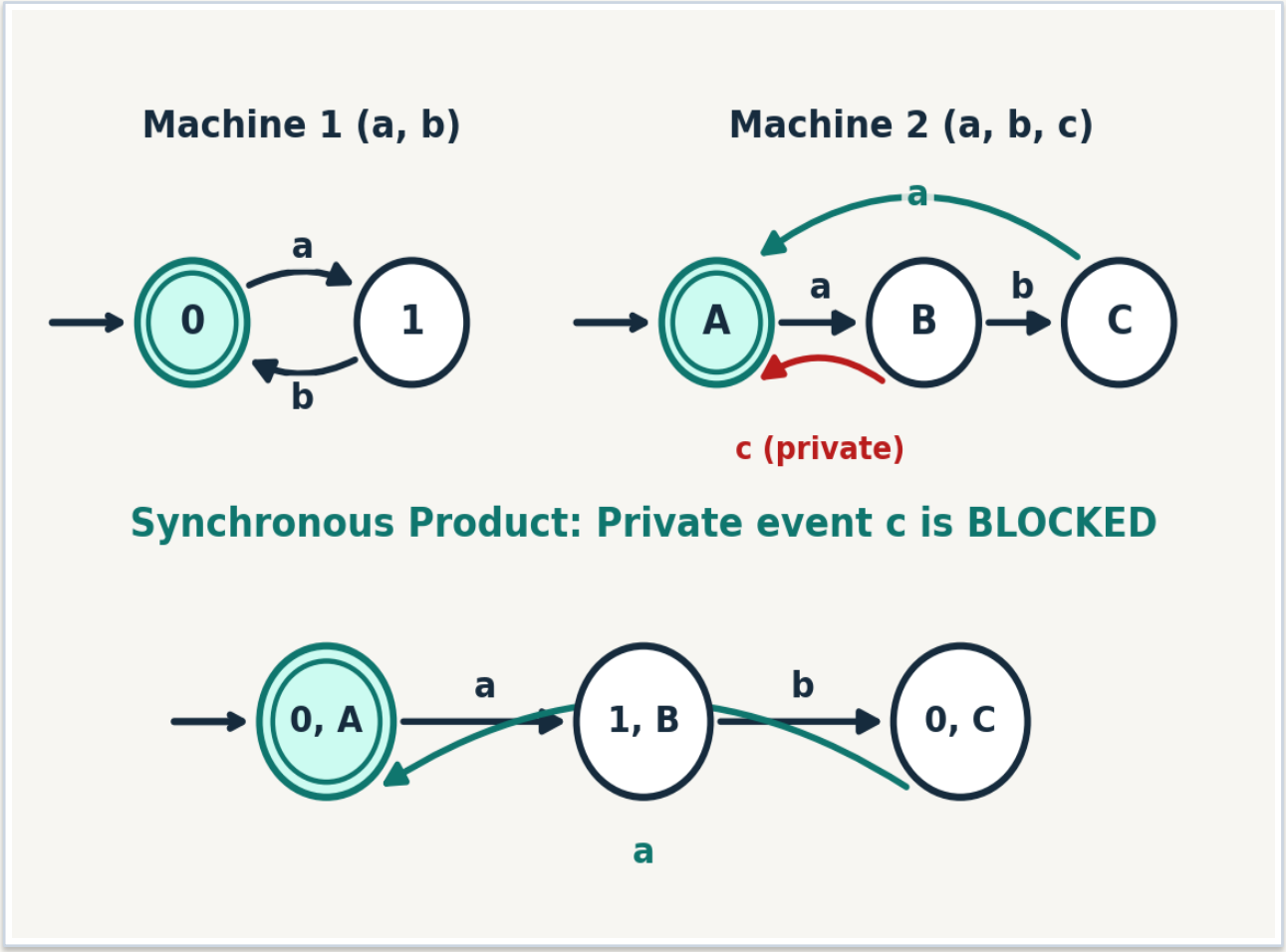
Engineers distinguish between machines that freeze and machines that spin uselessly.

- Nonblocking means every trace can finish
- Deadlock stops completely with zero events
- Deadlock nodes have zero outgoing arrows
- Livelock keeps running inside useless loops
- Both errors violate safe task completion

Key idea: Good supervisors guarantee zero deadlocks and zero livelock traps.



Synchronous Product Forces Two Machines Into Strict Lockstep



When we build factories from parts, we combine smaller machine models.

- Large factories use small subsystem models
- Product connects components on shared events
- Private events are completely blocked here
- Both machines must move together simultaneously
- Result equals intersection of both

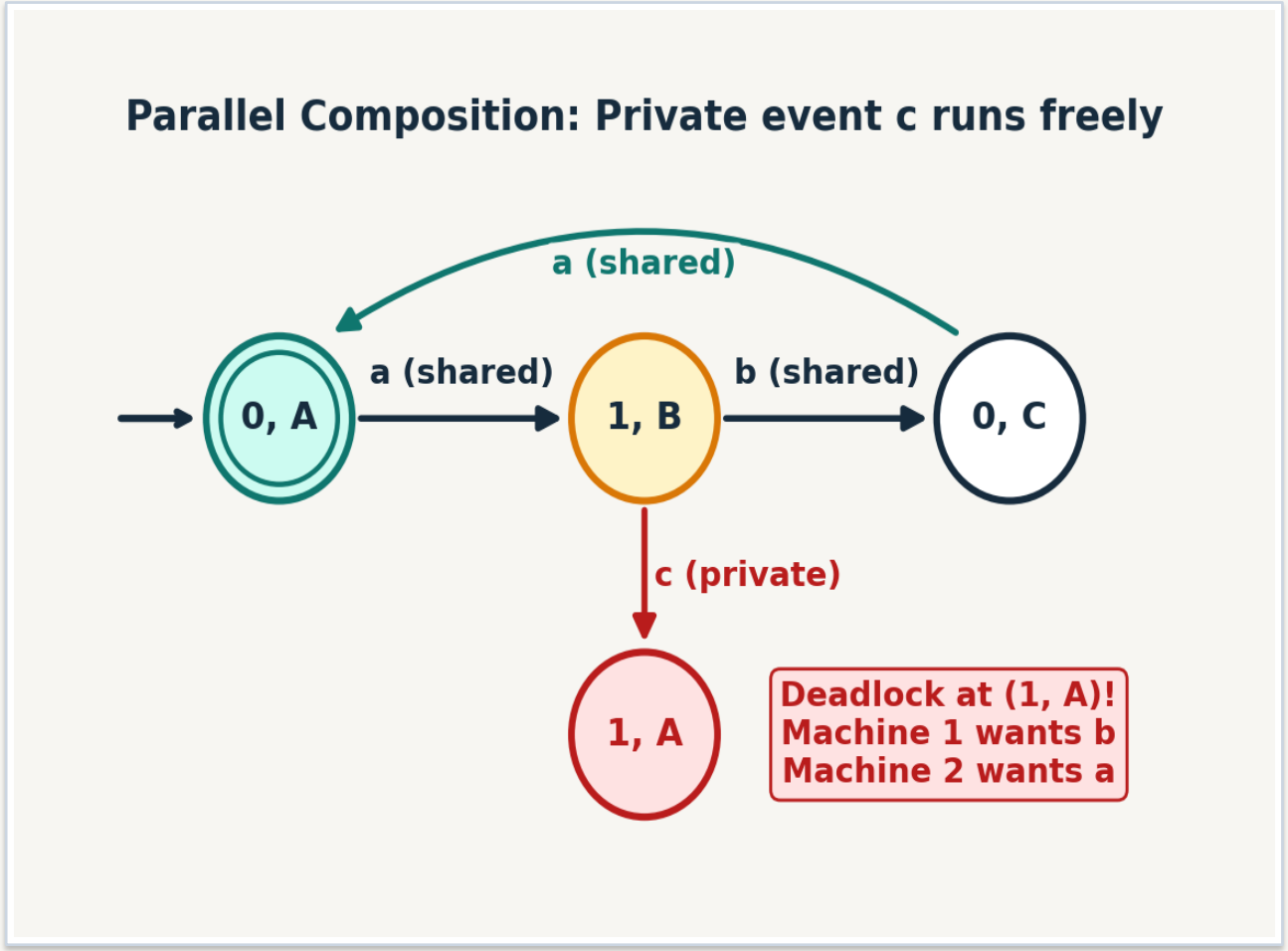
Key idea: Synchronous product models strict two-hand safety interlocks cleanly.

Parallel Composition Lets Private Events Run Independently Free

In real factories, machines work alone until they meet at a shared step.

- Shared events require simultaneous machine handshake
- Private events execute independently without waiting
- Explores larger state spaces than product
- Can introduce unexpected new blocking states
- Always check combined models for

Key idea: Two safe machines can deadlock when connected in parallel.



Synchronous Product Synchronizes Everything While Parallel Allows Private Moves

Both composition rules handle shared and private events in completely different ways.

Property	Synchronous Product	Parallel Composition
Shared Events	Both move together simultaneously	Both move together simultaneously
Private Events	Blocked and disabled forever	Run independently without waiting
Result Language	Exact intersection of languages	Lifted intersection over combined alphabet
State Space Size	Usually smaller reachable graph	Larger graph with concurrent paths
Main Engineering Use	Enforcing safety interlock specifications	Modeling real distributed factory plants

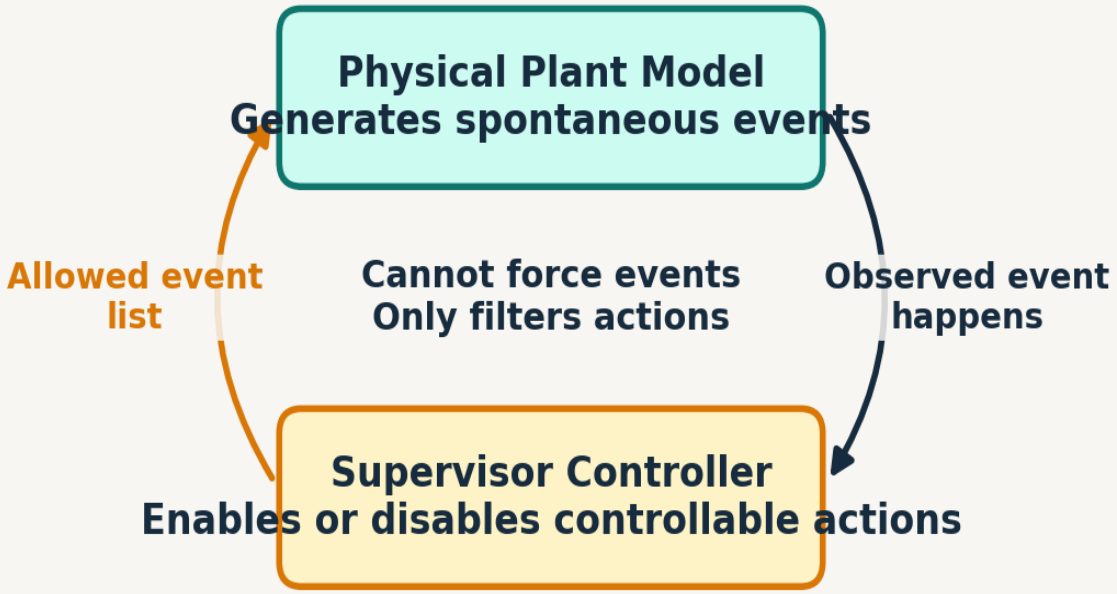
Takeaway: When both machines share the exact same alphabet, both operations match.

Supervisors Filter Unsafe Actions Without Forcing Machine Events

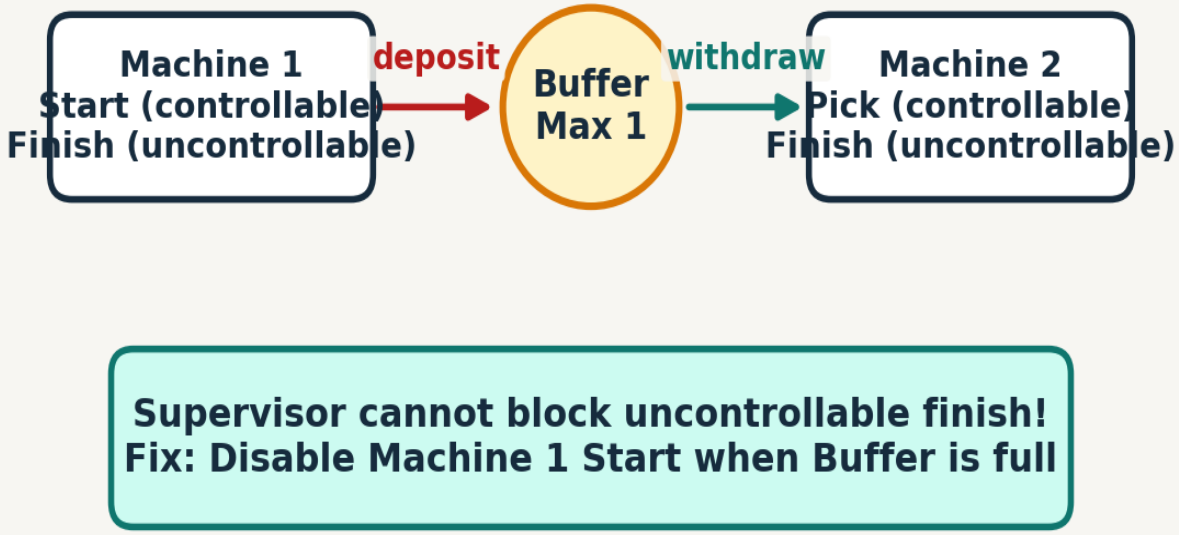
A supervisor watches the plant and disables dangerous controllable choices.

- Plant generates all spontaneous system events
- Supervisor acts as an enabling filter
- Supervisor can never force events forward
- Closed loop uses parallel composition rules
- Goal keeps system safe and nonblocking

Key idea: Think of the supervisor as smart traffic lights, not an engine.



Supervisors Must Prevent Danger Upstream Before Uncontrollable Events



Because actuators cannot stop spontaneous events, control must act earlier.

- Machine finish events are often uncontrollable
- Supervisor cannot stop spontaneous part drops
- Buffer overflow happens after second finish
- Supervisor must act one step earlier
- Disable controllable start when buffer fills

Key idea: Stop controllable starts early so uncontrollable finishes stay safe.

Exact Control Needs Both Controllability and Marking Alignment Together

A nonblocking supervisor exists if and only if two exact conditions hold.

Condition 1: Controllability

- Uncontrollable steps stay inside safe rules
- Plant cannot leak past boundaries
- Disable earlier controllable actions instead
- Prevents sudden forbidden state visits

Condition 2: Relative Closure

- Plant markings match specification markings
- Supervisor cannot erase physical completions
- Prevents artificial deadlocks at tasks
- Keeps closed-loop system strictly nonblocking

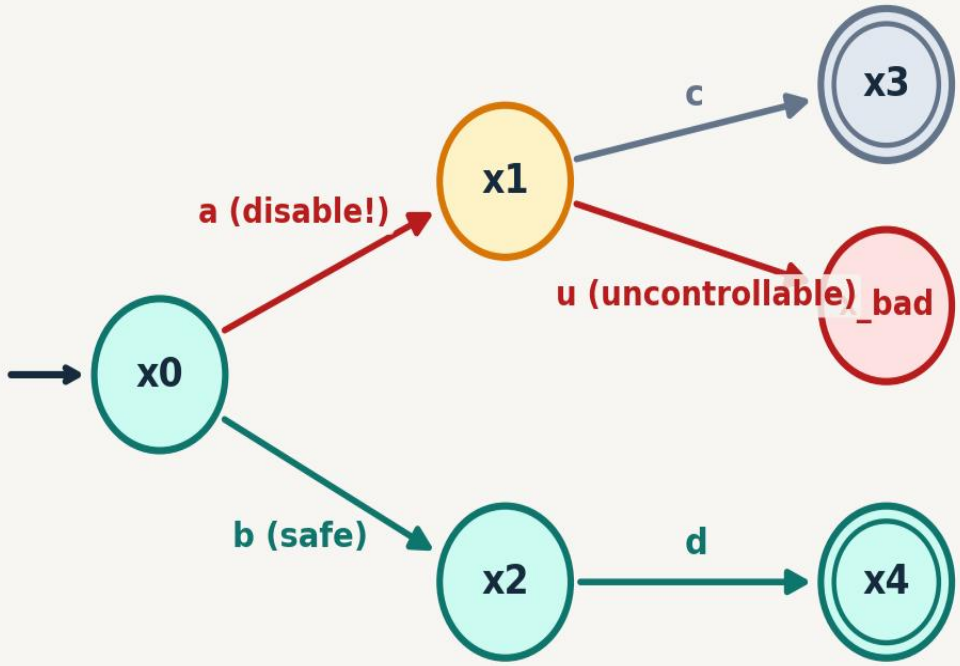
Fundamental Theorem: Both conditions together guarantee exact safe and nonblocking control.

Backward State Pruning Finds the Largest Safe Behavior

When a wish list is unsafe, we trim bad paths until safe.

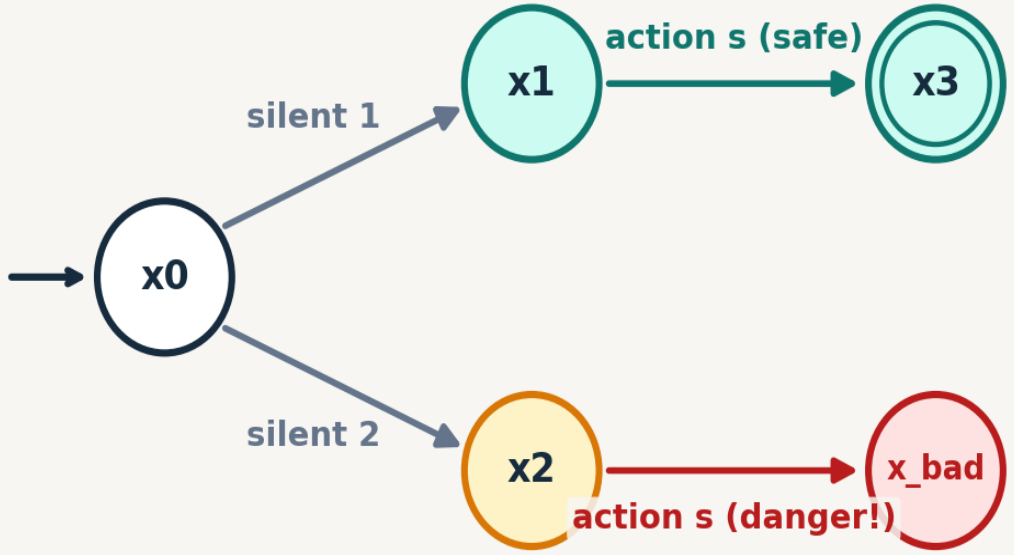
- Ideal rules often allow uncontrollable leaks
- We keep the largest controllable subset
- Step backward from forbidden states
- Prune states linked by uncontrollable arrows
- Disable controllable arrows entering pruned states

Key idea: Optimal synthesis stays maximally permissive while guaranteeing total safety.



Step back from x_bad: prune x1, disable controllable event a

Limited Sensors Cause Ambiguity Between Indistinguishable System Paths



Both silent paths look identical: supervisor cannot decide!

A supervisor cannot make different decisions if two paths look identical.

- Sensors miss silent internal machine events
- Different traces produce identical sensor logs
- Identical logs force identical control choices
- Safe path and danger path collide
- Normality restores a unique best solution

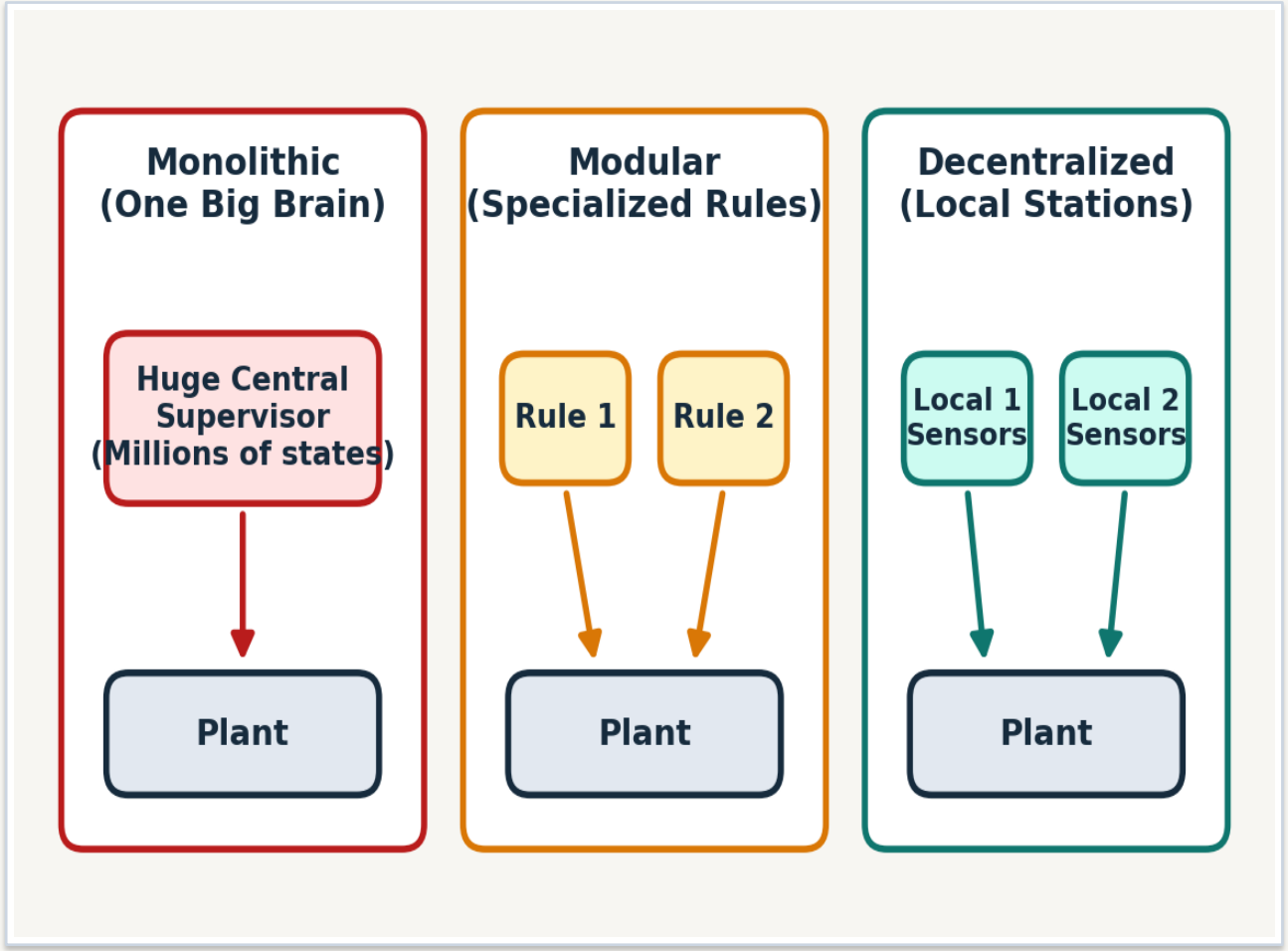
Key idea: Add sensors or use stronger normality rules to avoid confusion.

Modular and Decentralized Supervisors Avoid State Space Explosion

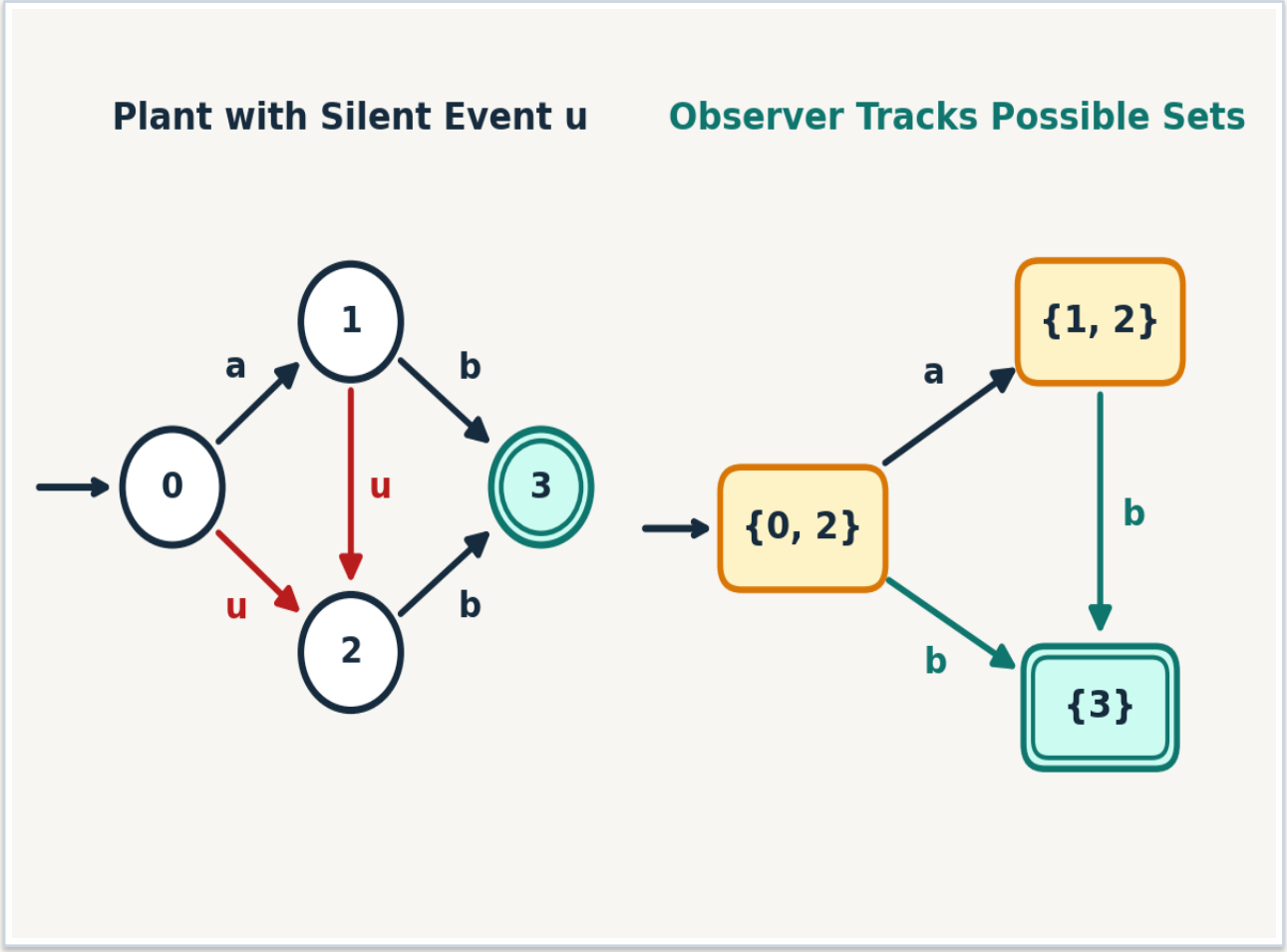
Big factories have millions of states, so one central controller fails.

- Ten robots create ten million states
- Monolithic control runs out of memory
- Modular control splits rules into parts
- Modular rules can still conflict together
- Decentralized agents control distant local stations

Key idea: Break big plants into small supervisors, then check for conflicts.



Observers Track Sets of Possible States Between Sensor Signals



When sensors stay quiet, an observer lists every state the plant could reach.

- Silent transitions happen between sensor triggers
- Silent reach finds all hidden moves
- Observer groups possible states into sets
- New sensor events shrink the set
- Observer stays completely deterministic over sensors

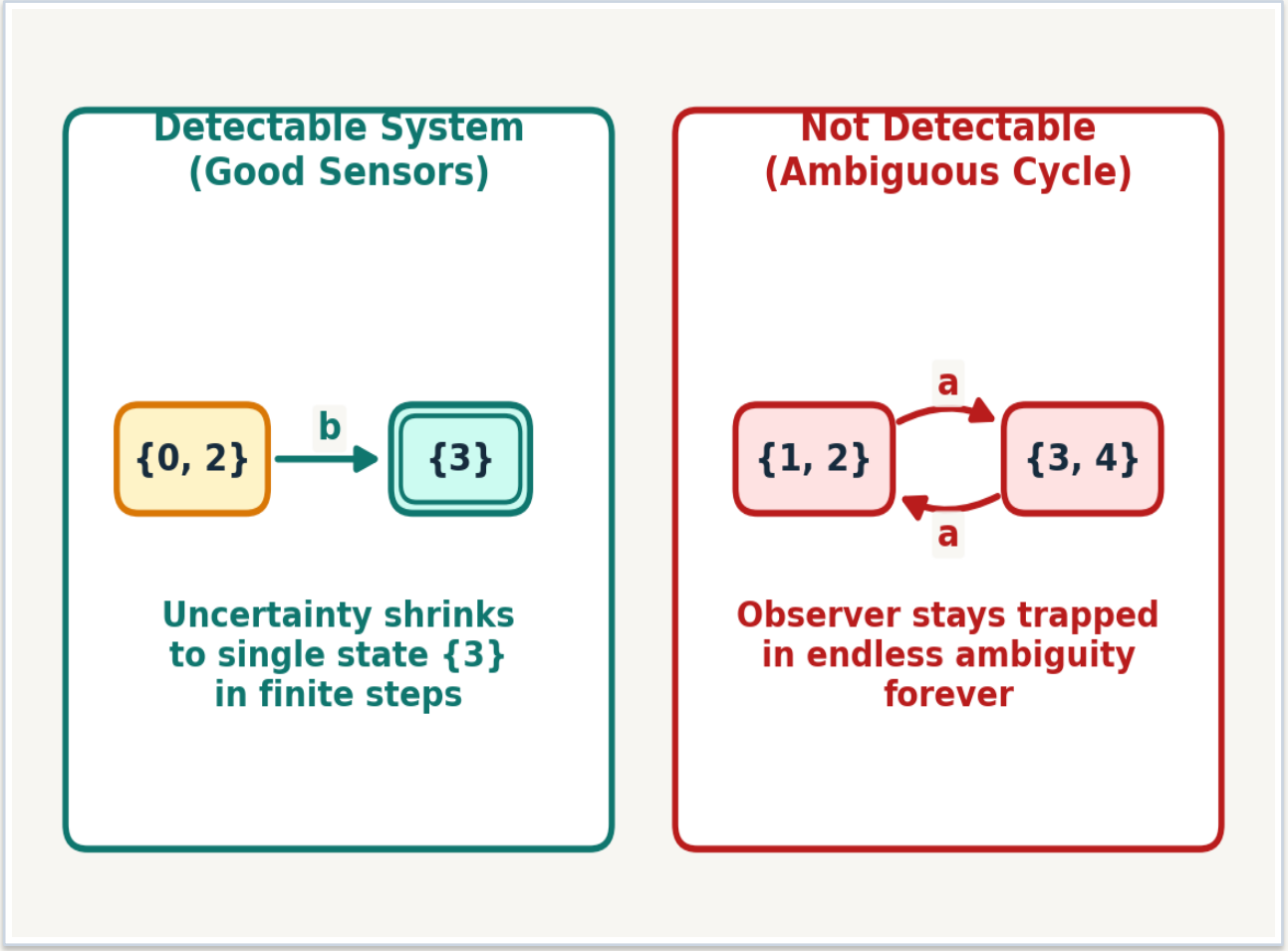
Key idea: Subset construction turns uncertain plants into deterministic state estimators.

Detectable Systems Shrink State Uncertainty After Finite Sensor Steps

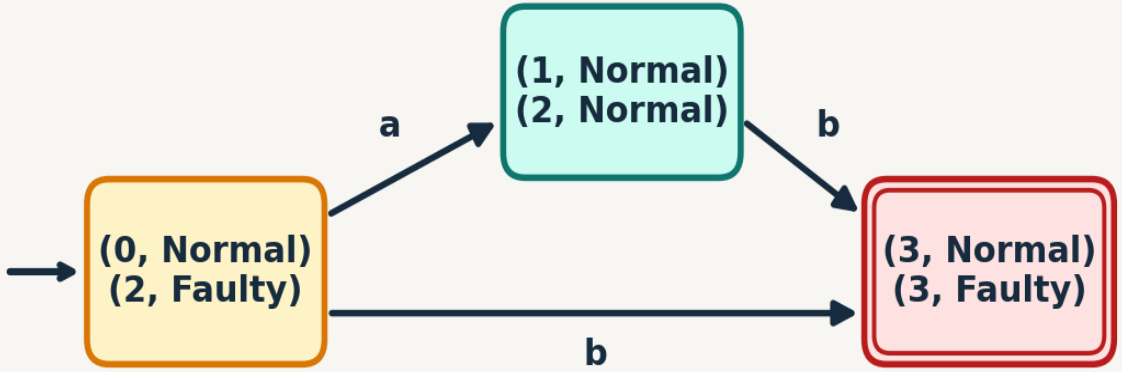
We rarely know every state instantly, so we check if ambiguity disappears.

- State observability demands single-state certainty always
- Real sensors allow brief initial uncertainty
- Detectability resolves true state eventually
- Check observer graph for ambiguous loops
- Ambiguous cycles trap observers in

Key idea: Every cycle in a detectable observer contains only single states.



Fault Diagnosers Remember Silent Failures Using Permanent Health Labels



Standard observers forget past faults, so diagnosers attach health tags to states.

- Critical failures often happen completely silently
- Observers forget faults once machines recover
- Diagnosers pair each state with labels
- Labels stay Normal until failure happens
- Faulty labels stay permanent along paths

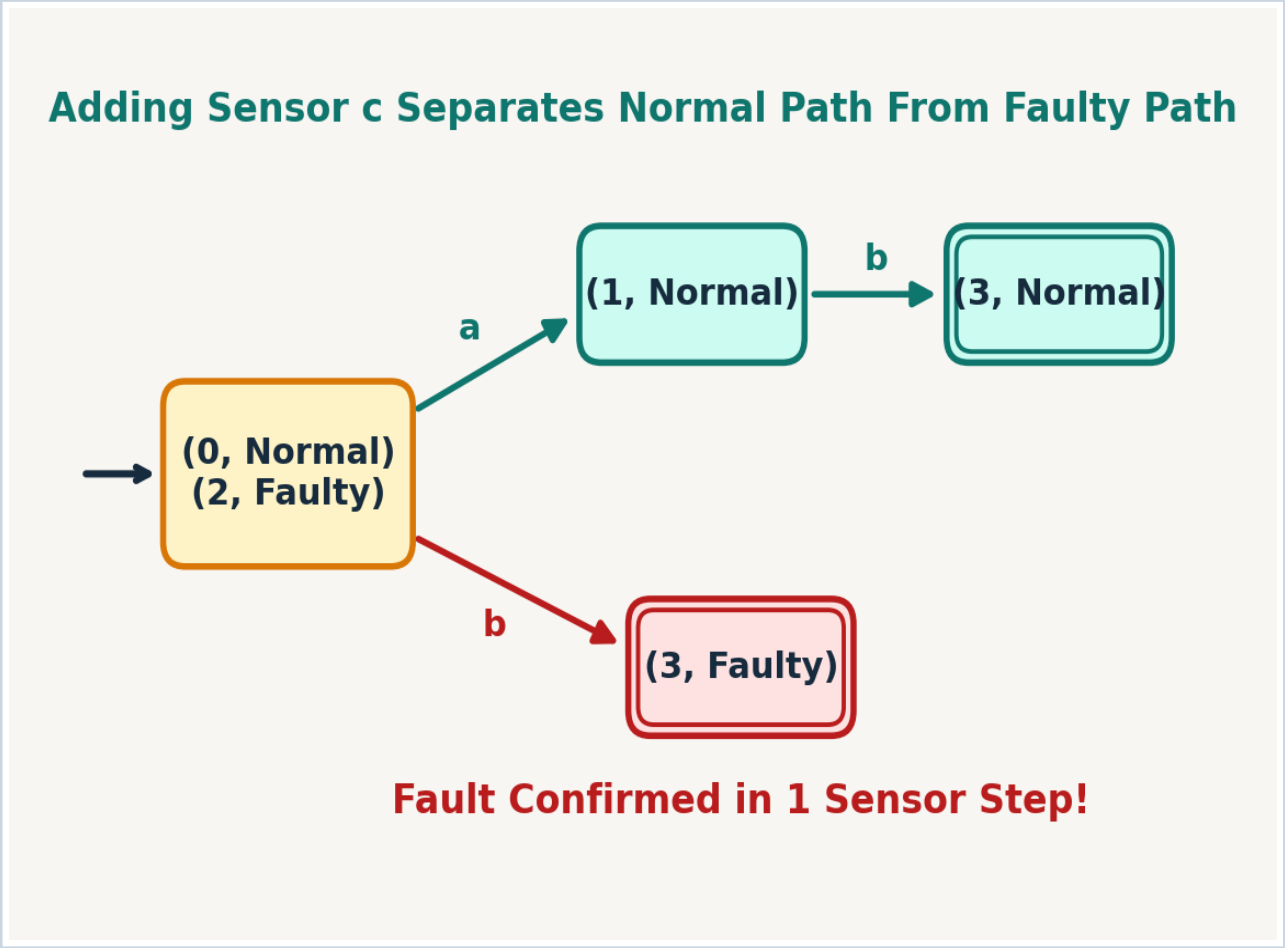
Key idea: When every pair shares a Faulty label, alarm triggers with certainty.

Adding One Sensor Breaks Endless Loops and Guarantees Fast Alarms

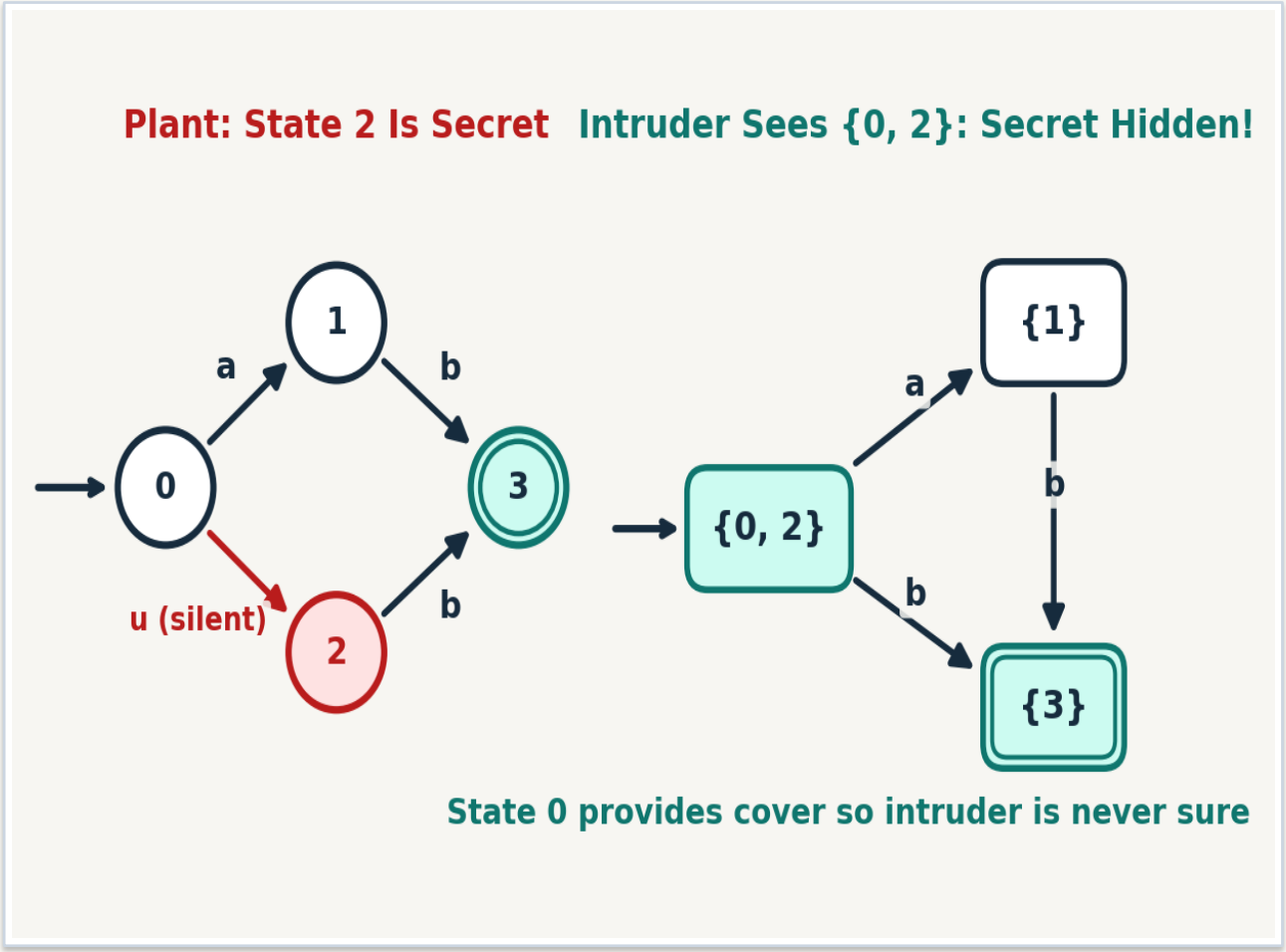
A system is diagnosable when silent faults cannot hide forever.

- Diagnosability guarantees alarms within bounded steps
- Uncertain loops hide silent faults forever
- No indeterminate cycles means provable diagnosability
- Adding observable sensors splits ambiguous paths
- Here sensor c confirms faults immediately

Key idea: Check diagnoser graphs for indeterminate cycles before deploying sensors.



Opacity Protects Secrets by Keeping Passive Intruders Always Uncertain



Security is the opposite of diagnosis: we intentionally preserve uncertainty.

- Passive intruders eavesdrop on sensor messages
- Current-state opacity hides secret system locations
- Every observer set includes non-secret states
- Language opacity hides confidential action sequences
- Defenses block leaks or hide signals

Key idea: Secret state 2 stays hidden inside {0, 2} with plausible cover.

Estimation, Diagnosis, and Opacity Solve Three Opposite Observation Problems

All three methods use subset tracking under silent events with opposite goals.

Dimension	State Estimation	Fault Diagnosis	System Opacity
Observer Role	Friendly remote monitor	Internal health monitor	Hostile external intruder
What Is Tracked	Current physical state location	Past silent failure events	Secret states or secret paths
State Structure	Subsets of plant states	Pairs of state and label	Subsets of plant states
Desired Goal	Shrink uncertainty to one	Raise certain fault alarm	Keep intruder always uncertain

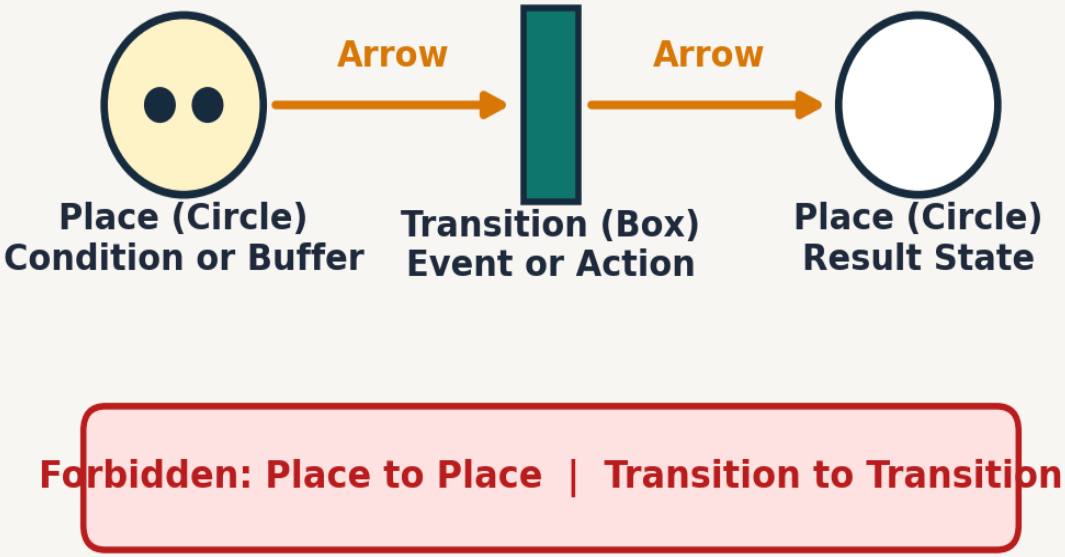
Takeaway: Diagnosis destroys uncertainty to catch failures; opacity preserves uncertainty to hide secrets.

Petri Nets Model Parallel Resources Without Drawing Huge Graphs

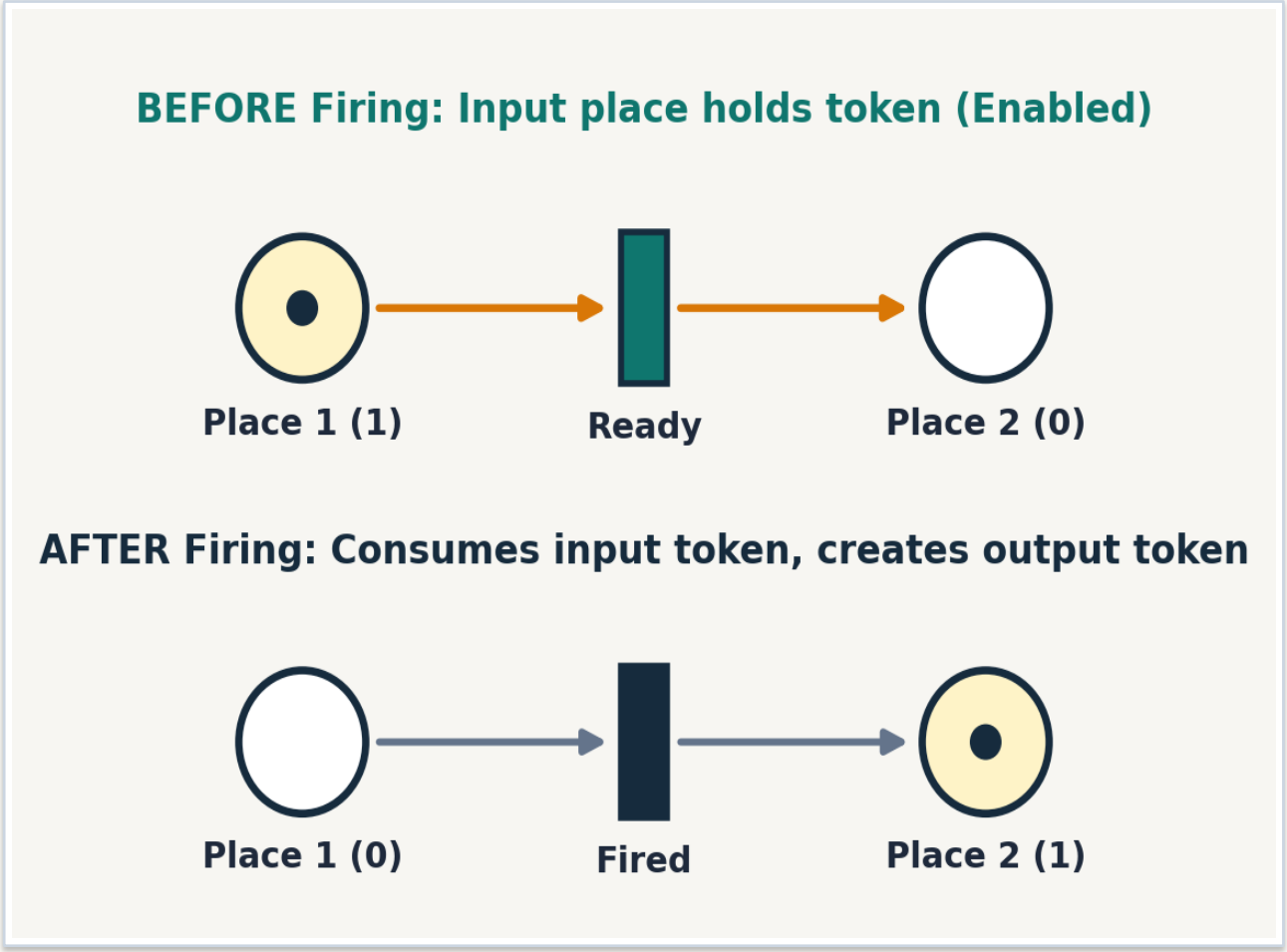
Instead of listing every global state, Petri Nets track distributed tokens.

- Places are circles holding resource tokens
- Transitions are boxes representing active events
- Tokens are black dots inside places
- Arrows alternate between places and transitions
- Marking vector counts tokens in places

Key idea: Circles never connect to circles; boxes never connect to boxes.



Transitions Fire Instantly When All Input Places Hold Tokens



Two simple rules drive every Petri Net simulation step by step.

- All input places need enough tokens
- Enabled transitions act like green lights
- Firing destroys tokens at input places
- Firing creates tokens at output places
- Logical firing happens in zero time

Key idea: Tokens are consumed at inputs and created at outputs, not slid along.

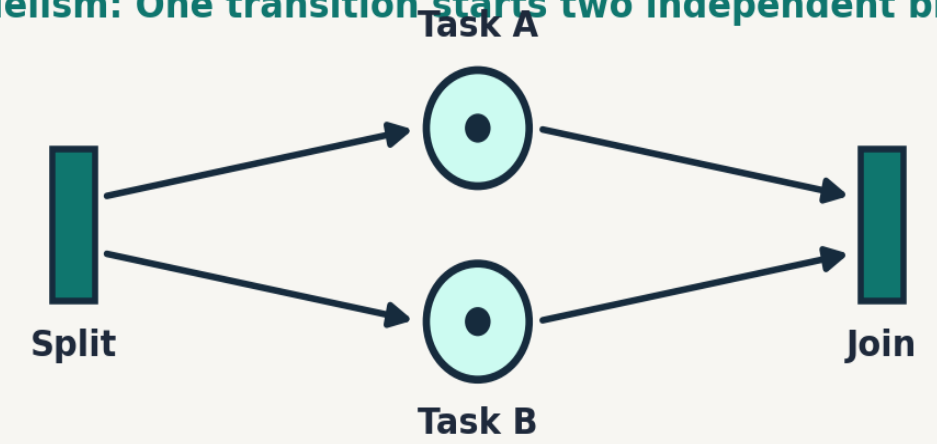
Petri Nets Separate Parallel Branches From Shared Resource Conflicts

The same graphical language handles simultaneous tasks and mutual exclusion.

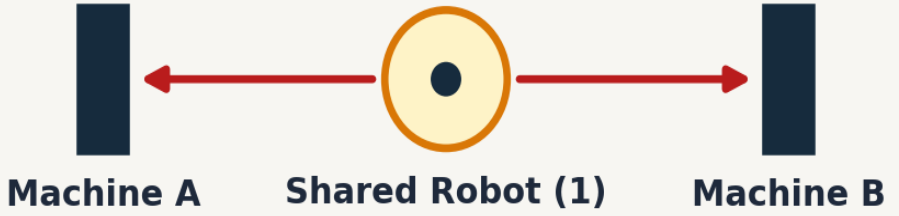
- Fork splits flow into parallel tasks
- Join waits until all branches finish
- Conflict happens when transitions share tokens
- Winning transition consumes token, disables rival
- Choice is resolved non-deterministically at runtime

Key idea: A single token in a shared place enforces mutual exclusion naturally.

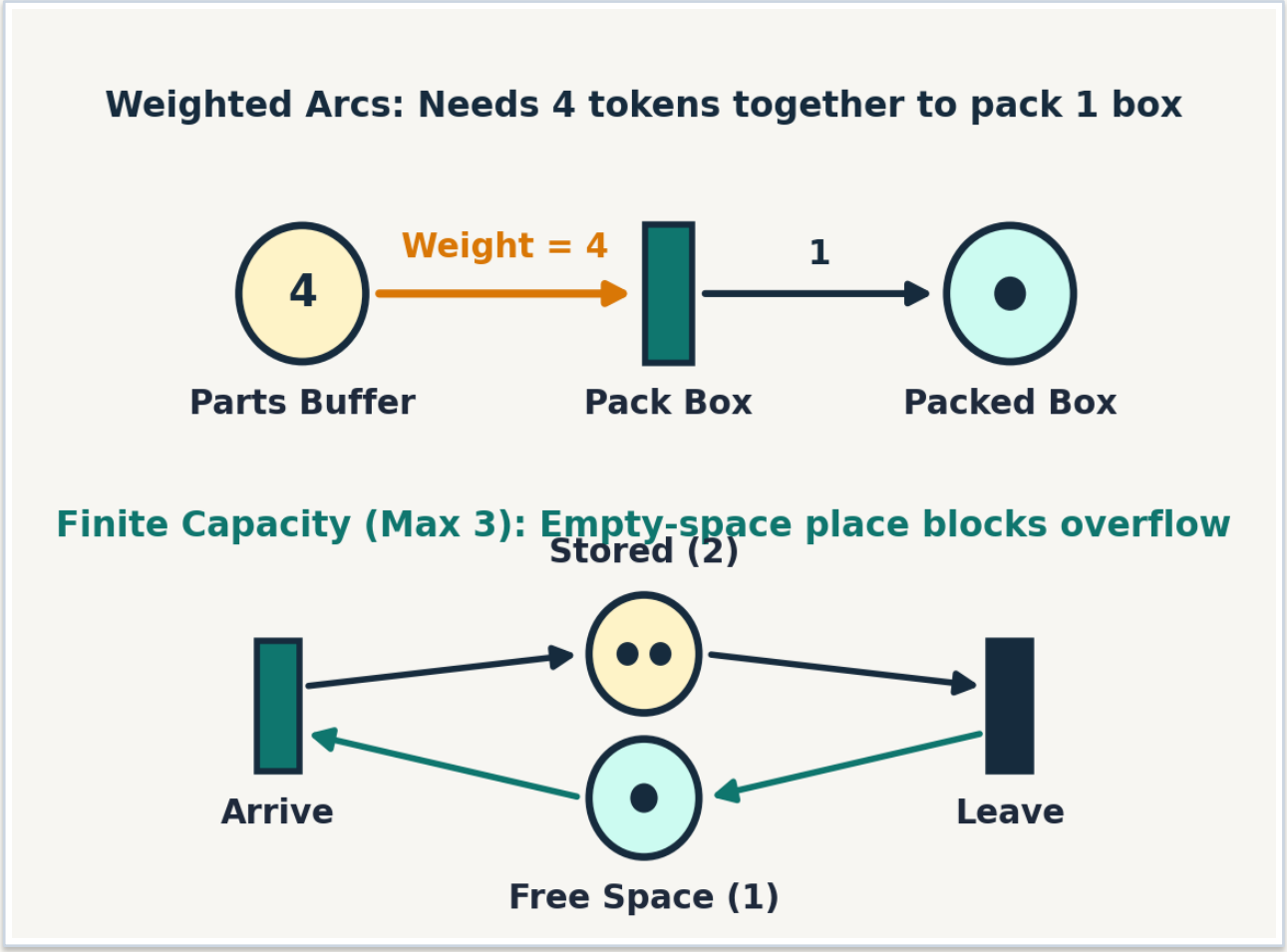
Parallelism: One transition starts two independent branches



Conflict: Two machines compete for one shared robot token



Weighted Arcs Group Batches and Complementary Places Limit Storage



Simple extensions let Petri Nets model batch machines and finite physical buffers.

- Arrow weights require multiple tokens together
- Batch packing needs four parts first
- Real buffers cannot store infinite items
- Complementary places count remaining empty spaces
- Empty-space tokens block incoming overflow naturally

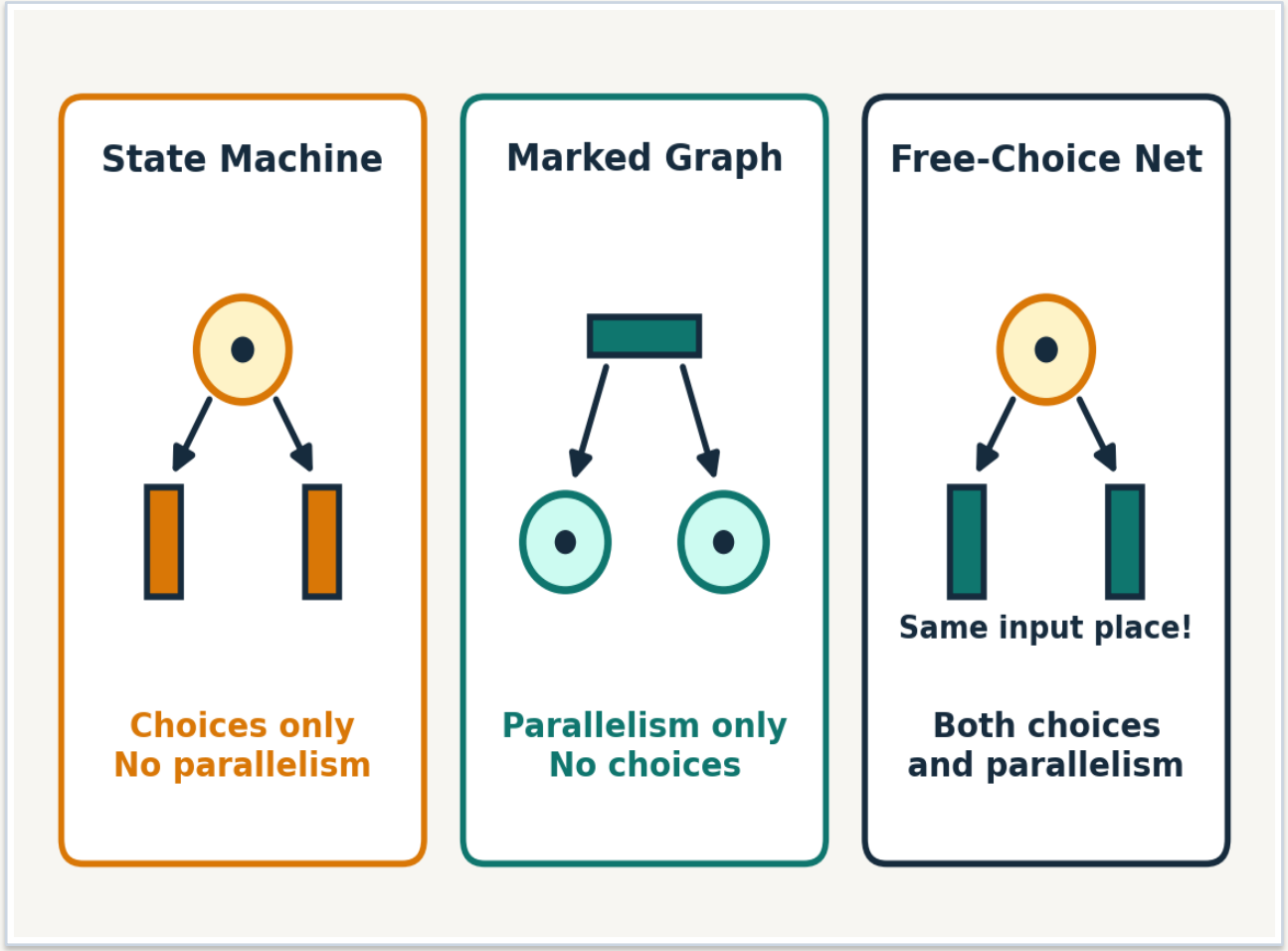
Key idea: Stored tokens plus empty-space tokens always equal maximum buffer size.

Three Structural Subclasses Trade Decision Power Against Parallelism

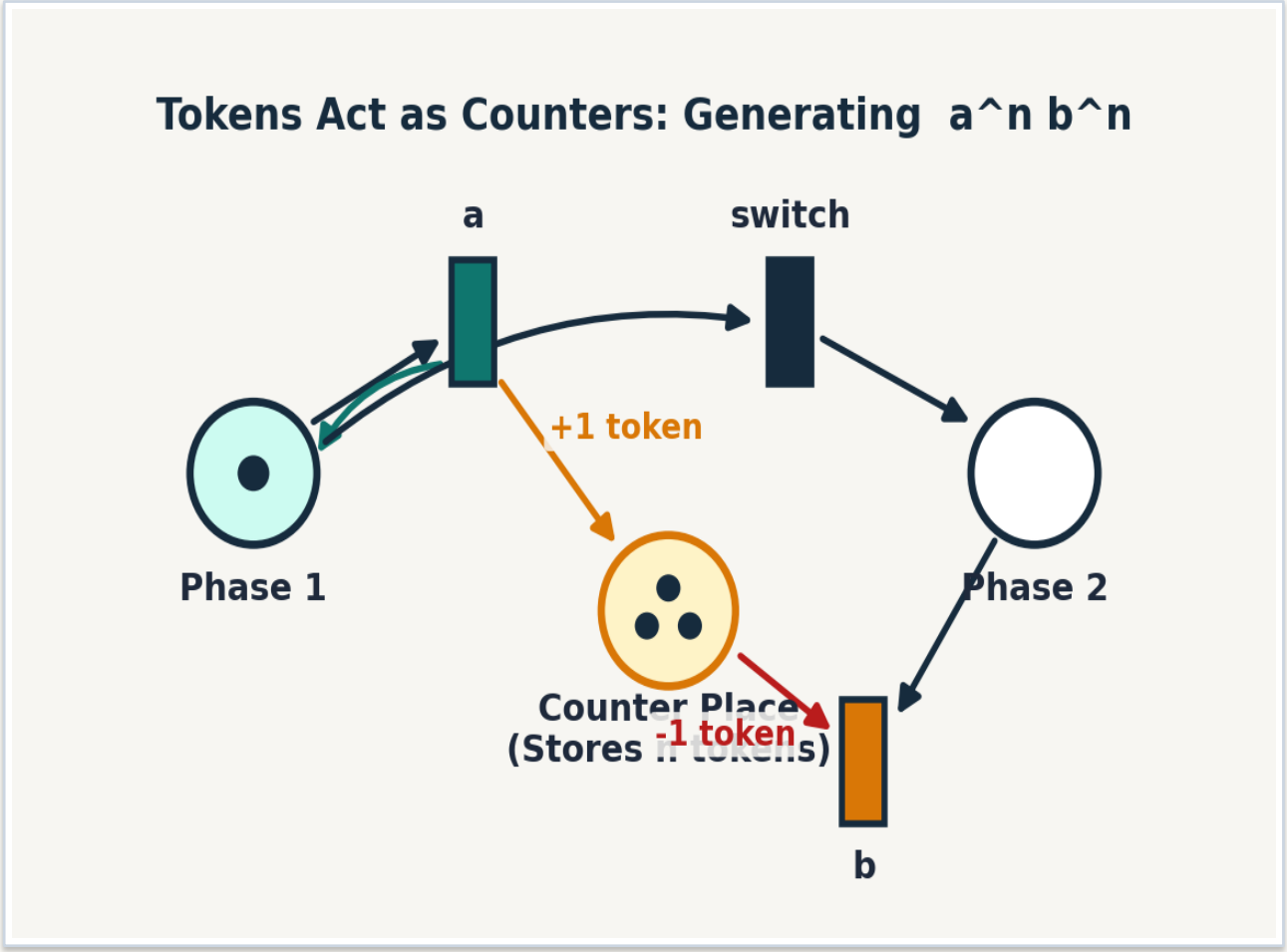
Restricting how arrows connect makes Petri Nets much easier to verify.

- State machines allow choices without parallelism
- Marked graphs allow parallelism without choices
- Free-choice nets combine both safely
- Shared input places share identical conditions
- Simpler structures speed up verification algorithms

Key idea: Free-choice nets prevent confusing conflicts between choice and parallel paths.



Accumulating Tokens Lets Petri Nets Count Beyond Finite Automata



Labeling transitions produces execution languages richer than finite state machines.

- Labeled transitions print symbols when firing
- Valid firing sequences build language words
- Finite automata have fixed finite memory
- Petri nets count using accumulated tokens
- Can generate non-regular balanced

Key idea: Unbounded places act as counters, giving Petri Nets higher language power.

Six Critical Questions Test Every Petri Net Before Building Hardware

Engineers check three safety questions and three long-term health questions.

Safety Question 1

Will storage buffers overflow? (Boundedness)

Safety Question 2

Can dangerous states occur? (Reachability)

Safety Question 3

Will machines freeze completely? (Deadlock)

Health Question 4

Can every action fire again? (Liveness)

Health Question 5

Are physical resources conserved? (Place Invariants)

Health Question 6

Can cycles repeat endlessly? (Transition Invariants)

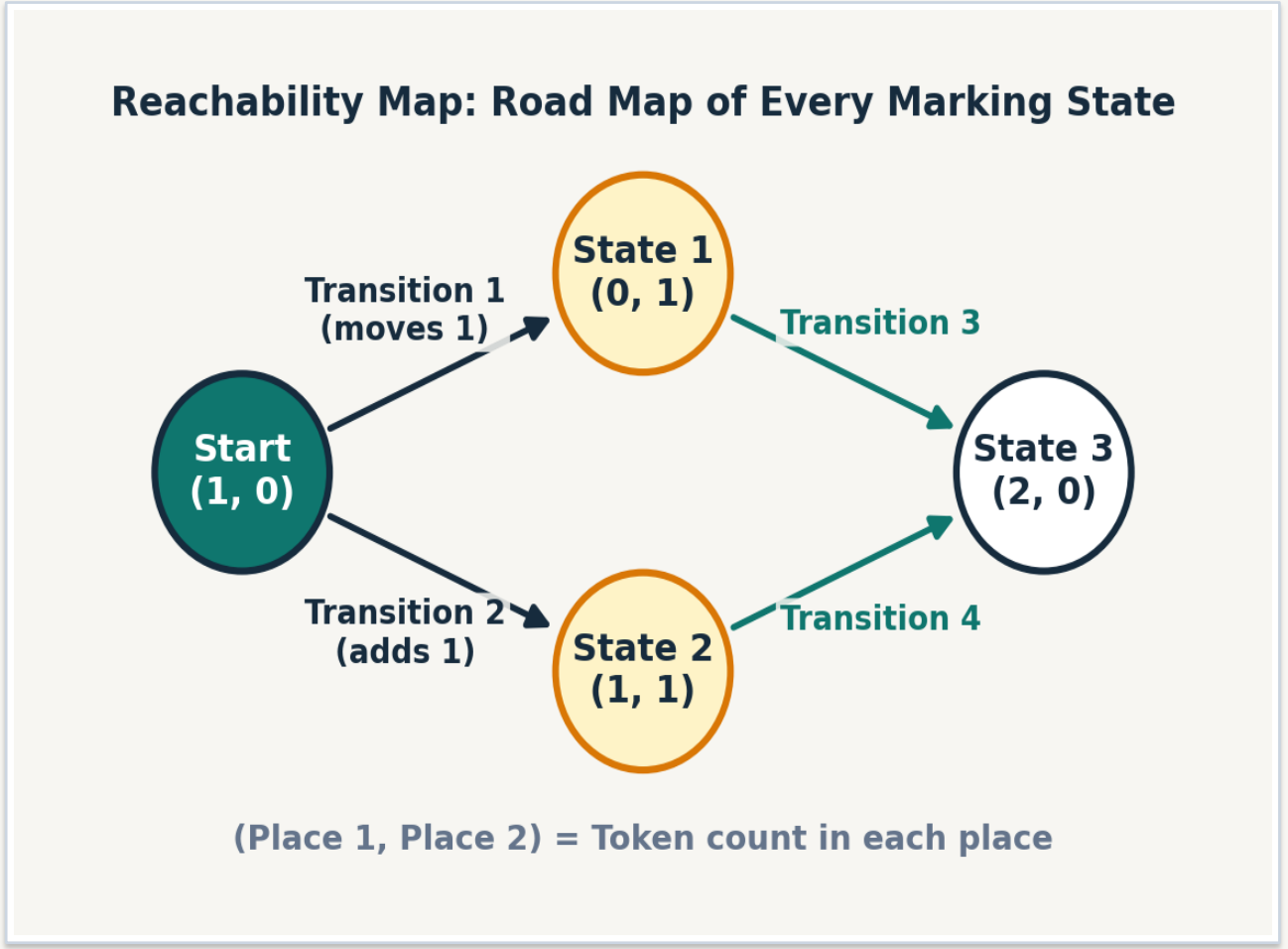
Goal: Answer all six questions mathematically before spending money on physical hardware.

Reachability Maps Trace Every Possible Marking Like Road Maps

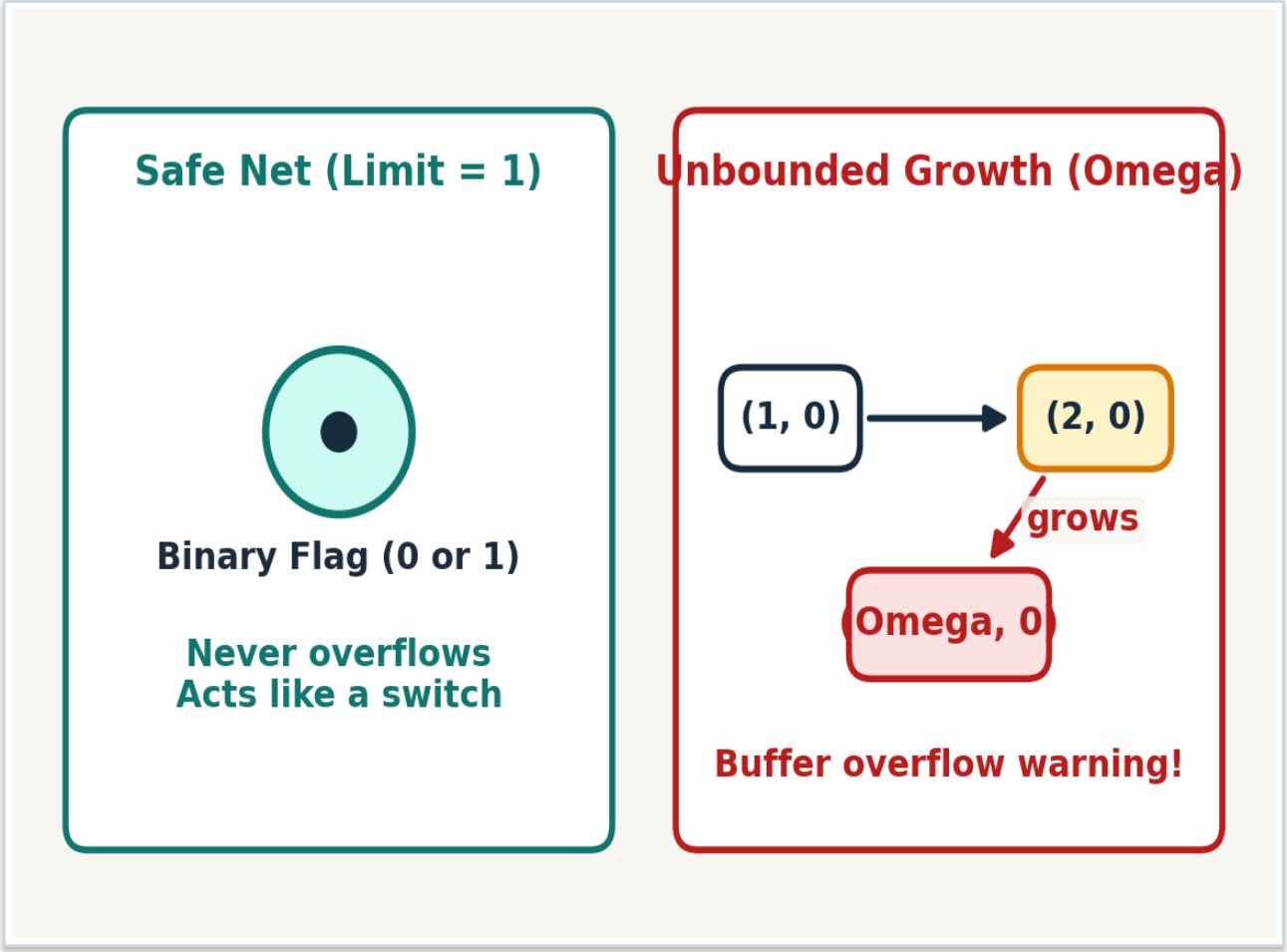
Each node shows token counts per place; each arrow shows one transition firing.

- Marking numbers count tokens per place
- Initial state sits at root node
- Arrows update token counts after firing
- Different paths can reach identical states
- Proves dangerous states stay unreachable forever

Key idea: Always verify unsafe markings sit outside the Reachability Set.



Omega Symbols Catch Buffer Overflows Inside Infinite State Trees



When tokens grow without limit, engineers replace infinite counts with Omega.

- Safe nets hold one token maximum
- Bounded nets guarantee zero buffer overflow
- Unbounded loops grow tokens toward infinity
- Omega symbol keeps state trees finite
- Seeing Omega flags dangerous overflow bugs

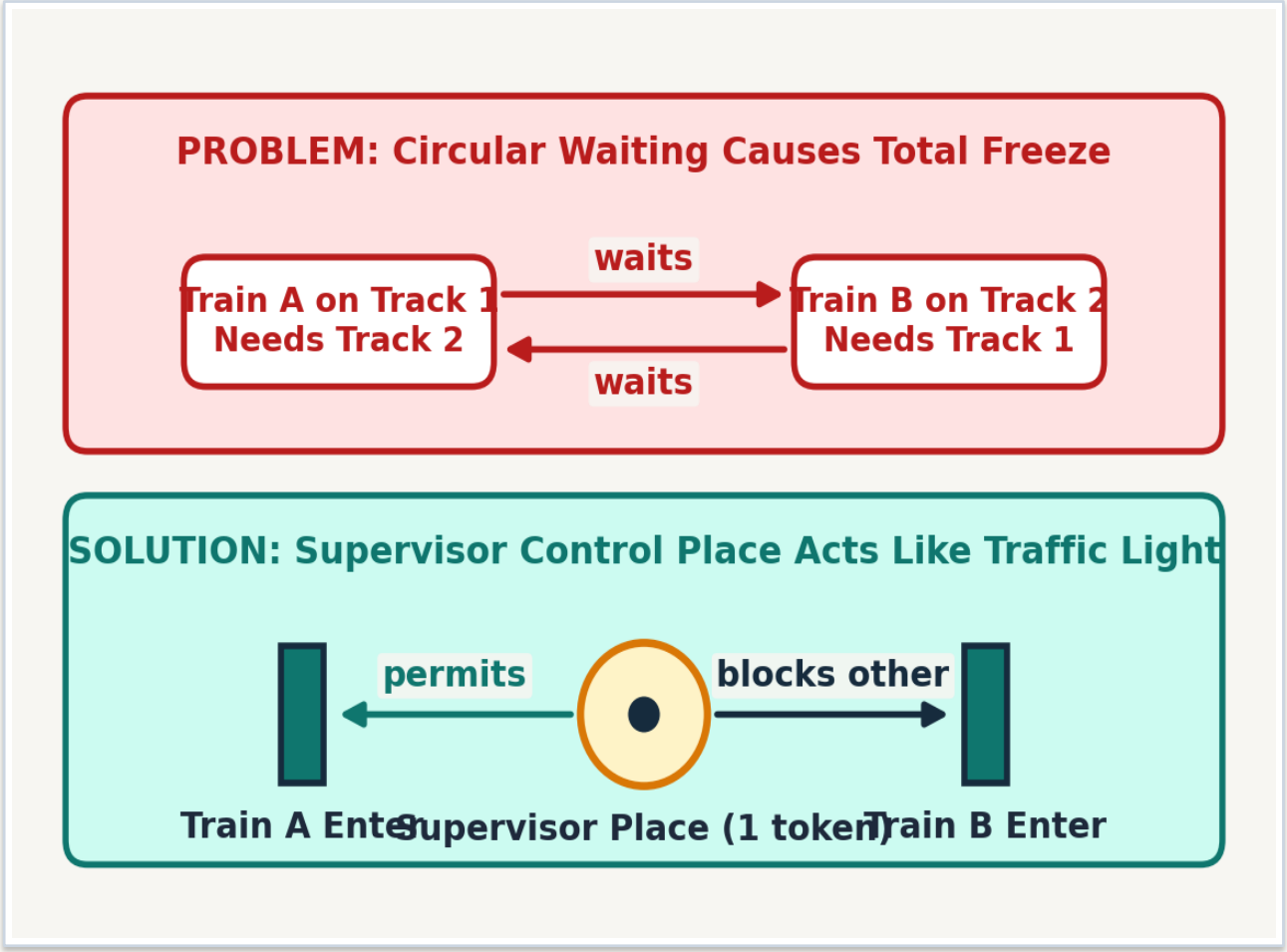
Key idea: Zero Omega symbols in the Coverability Tree proves finite memory.

Supervisor Control Places Break Circular Waiting to Prevent Deadlocks

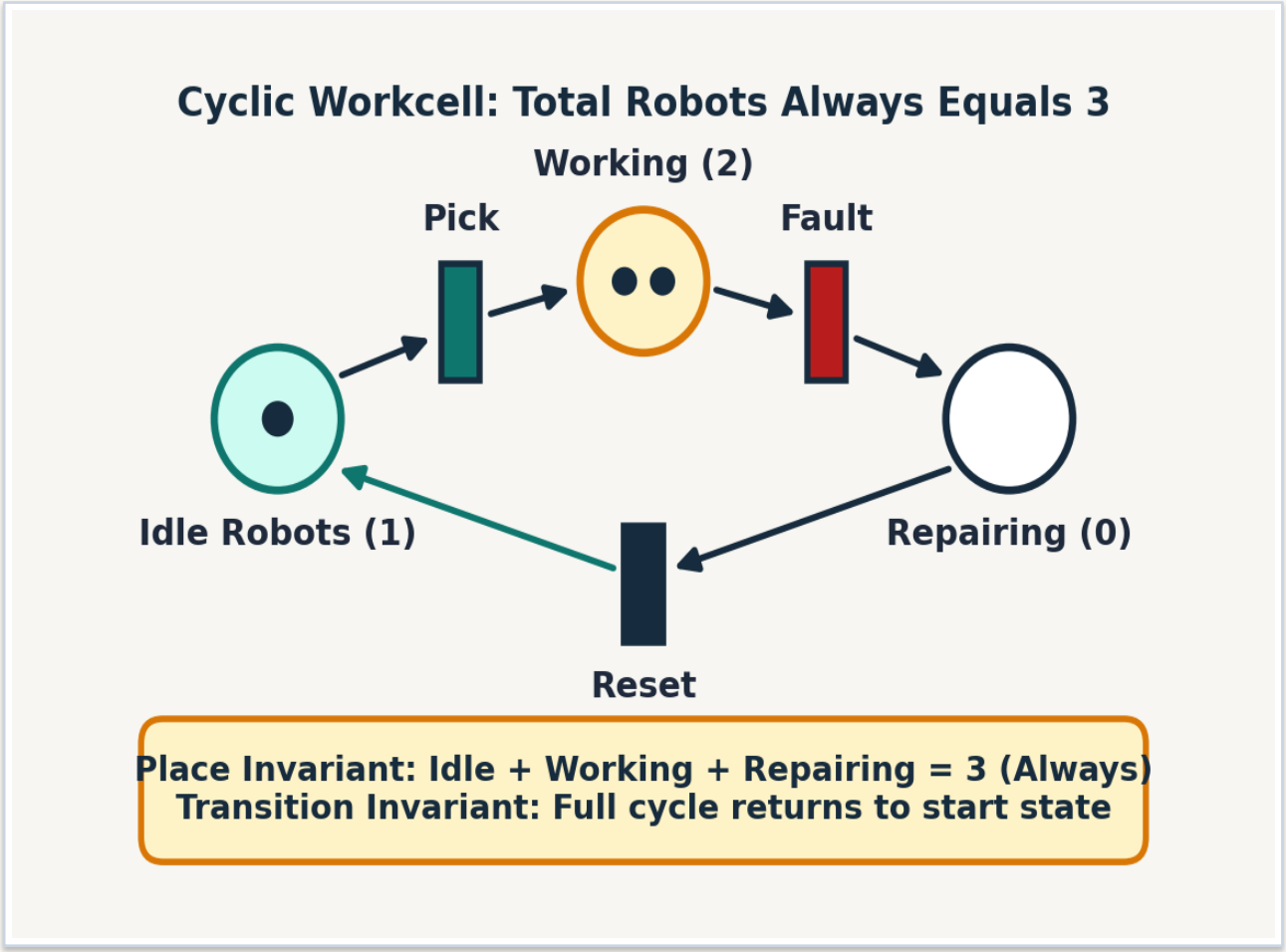
Petri Nets both reveal circular waits and fix them with traffic-light places.

- Deadlock means zero transitions are enabled
- Liveliness level four keeps actions active
- Two trains waiting freeze both tracks
- Control place acts like traffic light
- Restricts entry until track clears safely

Key idea: One added control place makes circular-wait systems provably deadlock-free.



Incidence Matrices Prove Resource Conservation and Cyclic Work Loops



Linear algebra turns graphical Petri Net checks into fast matrix equations.

- Incidence matrix records net token changes
- State equation screens target markings quickly
- Math solutions can still be impossible
- Place invariants prove constant resource totals
- Transition invariants verify repeatable cyclic loops

Key idea: Three robots stay three robots across idle, working, and repair states.

Automata and Petri Nets Form One Complete Engineering Toolkit

Choose the right mathematical tool for each real factory or software problem.

Finite Automata

Best for order and rules

Check deadlocks and combine machines

Supervisory Control

Best for safe restrictions

Disable controllable actions before faults

Observers and Diagnosers

Best for limited sensors

Find hidden failures or hide secrets

Petri Nets

Best for parallel tokens

Verify capacity, liveness, and cycles

Goal Achieved: Build safe, nonblocking, observable, and deadlock-free engineering systems from day one.