Principles of Cyber-Physical Systems

Safety Requirements

Instructor: Lanshun Nie
nls@hit.edu.cn

Requirements

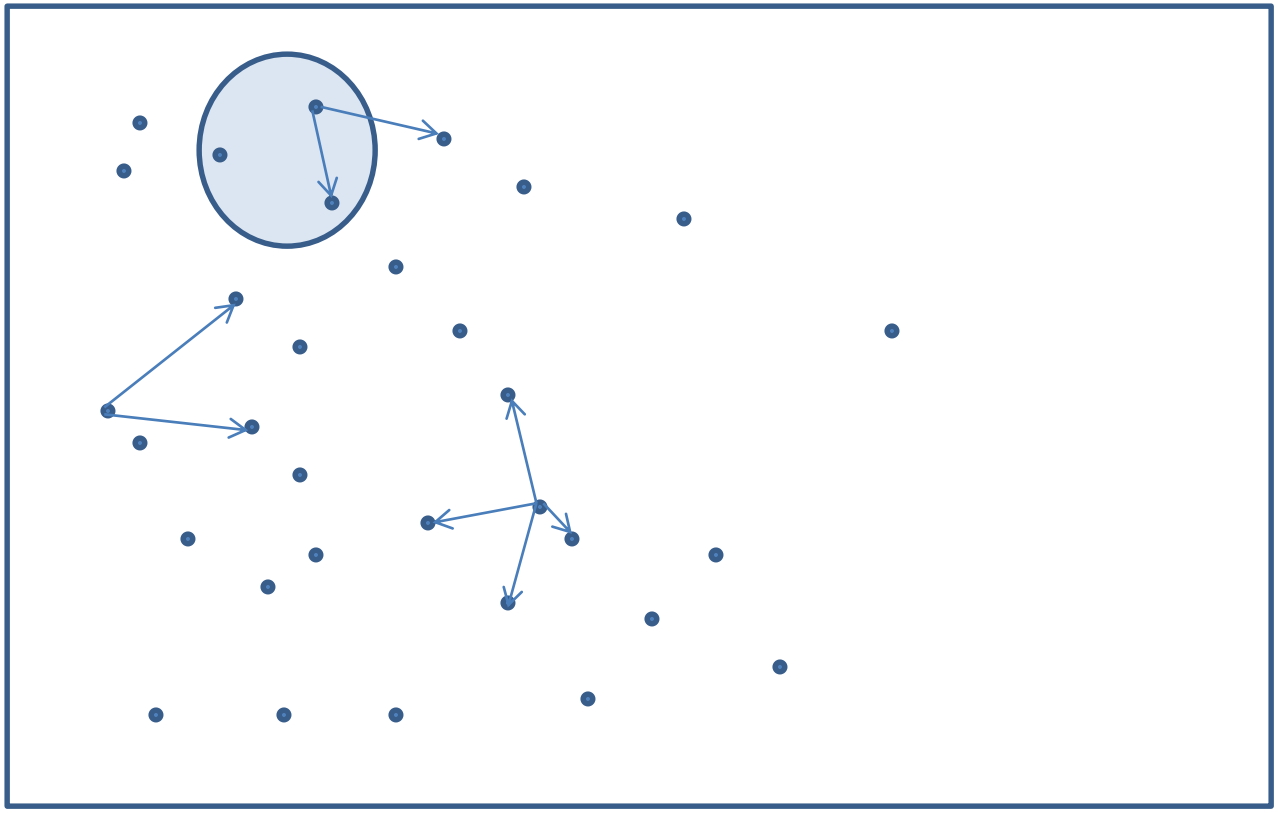
- Requirement: Desirable property of the executions of the system
  - Informal: Either implicit, or stated in English in documents
  - Formal: Stated explicitly in a mathematically precise manner
- High assurance / safety-critical systems: Formal requirements
- Model/design/system meets the requirements if every execution satisfies all the requirements
- Clear separation between requirements (what needs to be implemented) and system (how it is implemented)
- Verification problem: Given a requirement φ and a system/model C, prove/disprove that the system C satisfies the requirement φ

Safety Requirements

- A safety requirement states that a system always stays within "good' states (i.e. a nothing bad ever happens)
- Leader election: it is never the case that two nodes consider them to be leaders
- Collision avoidance: Distance between two cars is always greater than some minimum threshold
- Different class of requirements: Liveness
  - System eventually attains its goal
  - Leader election: Each node eventually makes a decision
  - Cruise controller: Actual speed eventually equals desired speed
- Formalization and analysis techniques for safety and liveness differ significantly, so let us first focus on safety

Transition Systems

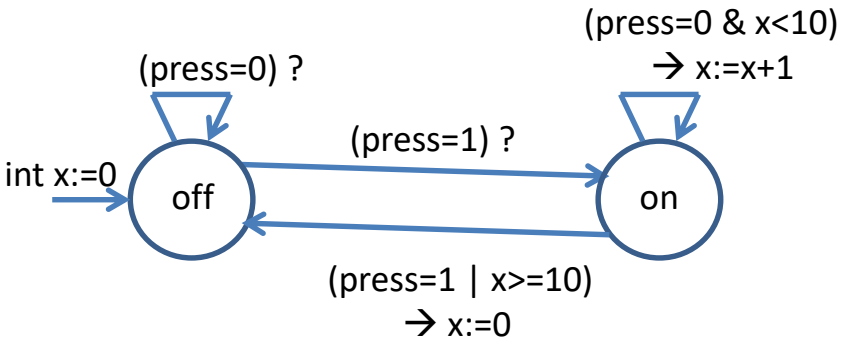
States + Initial states + Transitions between states

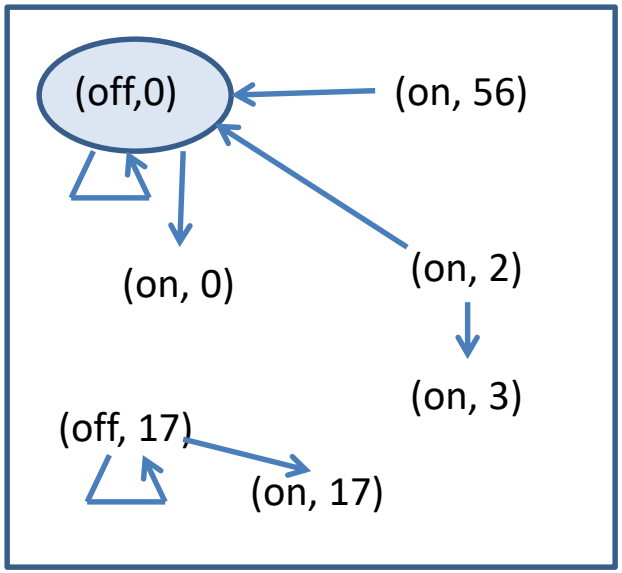


Definition of Transition System

- Syntax: a transition system T has
  1. a set S of (typed) state variables
  2. Initialization Init for state variables
  3. Transition description Trans given by code to update state vars
- Semantics:
  1. Set Q_S of states
  2. Set [Init] of initial states (this is a subset of Q_S)
  3. Set [Trans] of transitions, subset of $Q_S \times Q_S$
- Synchronous reactive components, programs, and more generally systems, all have an underlying transition system

Switch Transition System





State variables:

{off, on} mode, int x

Initialization:

mode = off; x = 0

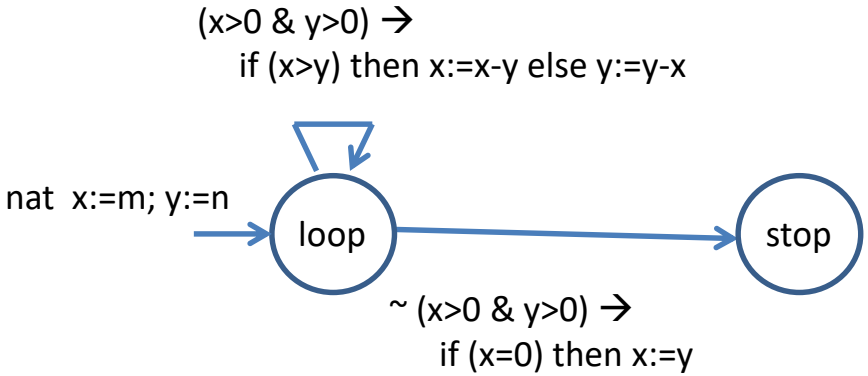
Transitions:

(off,n) -> (off,n);
(off,n) -> (on,n);
(on,n) -> (on,n+1) if n<10;
(on,n) -> (off,0)

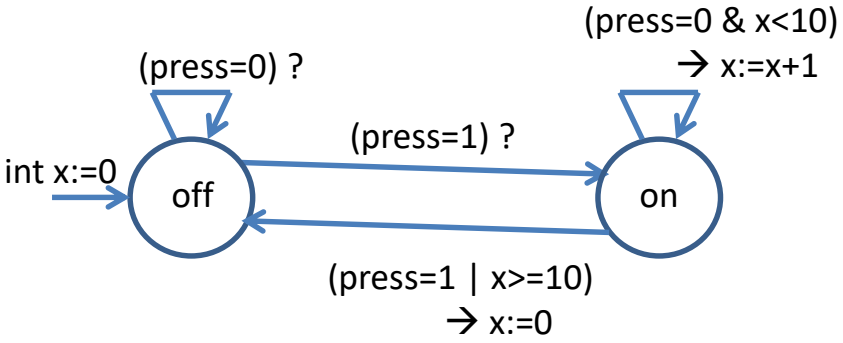
Input/output variables become local
Values for input vars chosen nondeterministically

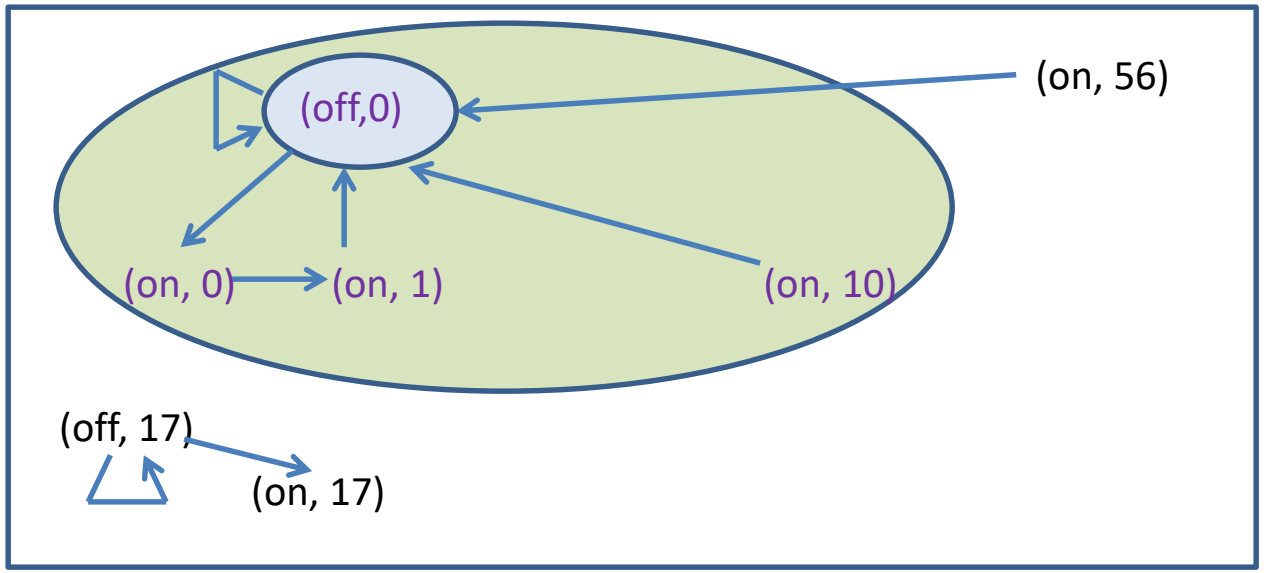
Euclid's GCD Algorithm

Classical program to compute greatest common divisor of (non-negative) input numbers m and n

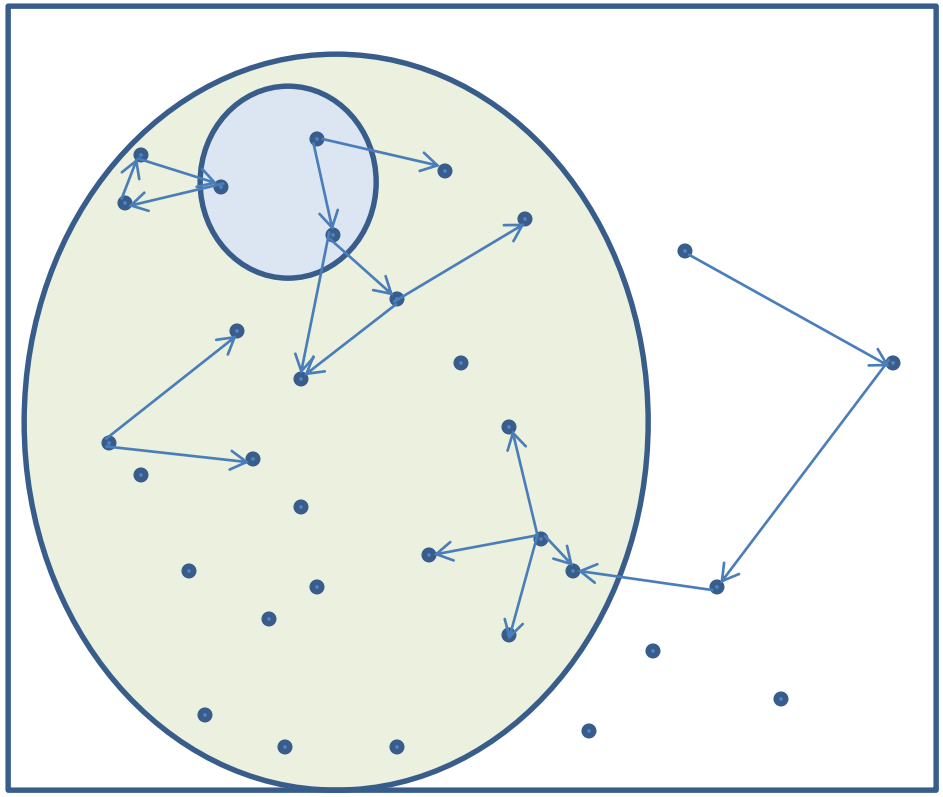


Reachable States



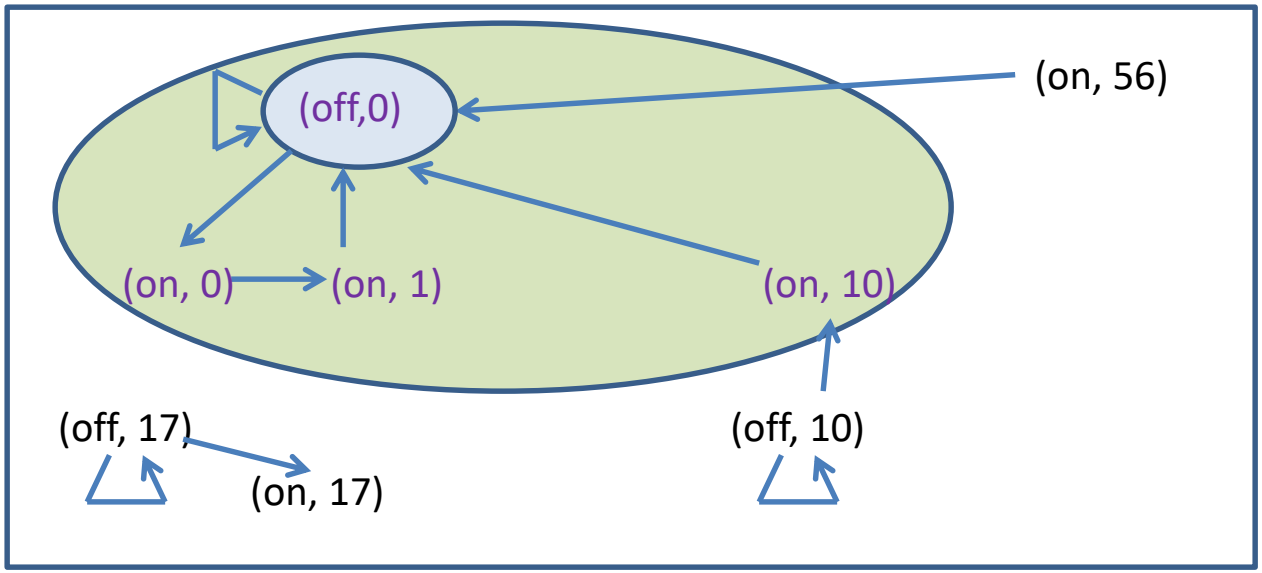


Reachable States of Transition Systems



A state s of a transition system is reachable if there is an execution starting in an initial state and ending in the state s

Invariants



- Property of a transition system: Boolean-valued expression φ over state variables
- Property φ is an invariant of T if every reachable state satisfies φ
- Examples of invariants: (x <= 10); (x <= 50); (mode= off -> x=0)
- Examples of properties that are not invariants: (x < 10); (mode=off)

Invariants

- Express desired safety requirement as property φ of state variables
  - If φ is an invariant then the system is safe
  - If φ is not an invariant, then some state satisfying $\sim\varphi$ is reachable (execution leading to such a state is counterexample)

- Leader election:

  $(r_n = N \rightarrow id_n = \max P)$, P : set of identifiers of all nodes

- Euclid's GCD Program:

  (mode=stop -> x = gcd(m,n))

Formal Verification

Grand challenge:

Automate verification as much as possible !

Analysis Techniques

- Dynamic Analysis (runtime)
  - Execute the system, possibly multiple times with different inputs
  - Check if every execution meets the desired requirement
- Static Analysis (design time)
  - Analyze the source code or the model for possible bugs
- Trade-offs
  - Dynamic analysis is incomplete, but accurate (checks real system, and bugs discovered are real bugs
  - Static analysis can catch design bugs early !
  - Many static analysis techniques are not scalable (solution: analyze approximate versions, can lead to false warnings)

Invariant Verification

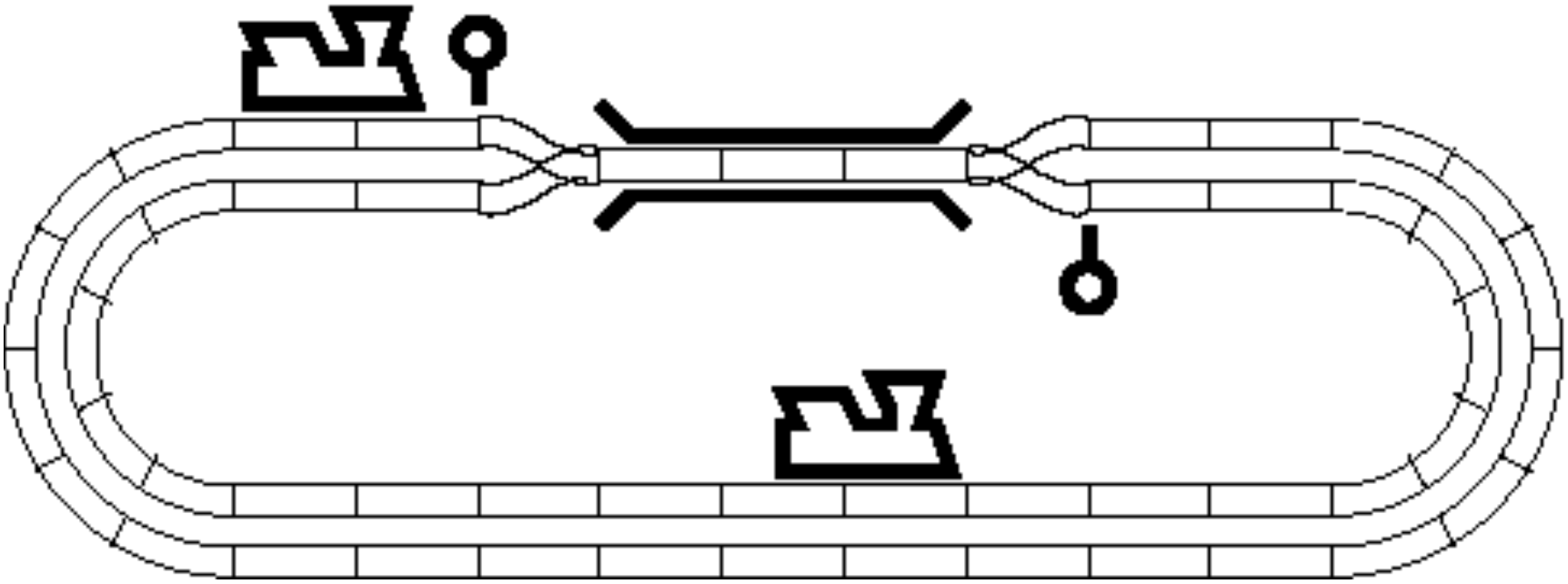
- Simulation
  - Simulate the model, possibly multiple times with different inputs
  - Easy to implement, scalable, but no correctness guarantees
- Proof based
  - Construct a proof that system satisfies the invariant
  - Requires manual effort (partial automation possible)
- State-space analysis (Model checking)
  - Algorithm explores "all" reachable states to check invariants
  - Not scalable, but current tools can analyze many real-world designs (relies on many interesting theoretical advances)

Requirements-based Design

- Systematic approach to design of systems
- Given:
  - Input/output interface of system C to be designed
  - Model E of the environment
  - Safety property φ of the composite system

- Design problem: Fill in details of C (state variables, initialization, and update) so that C || E satisfies the invariant φ

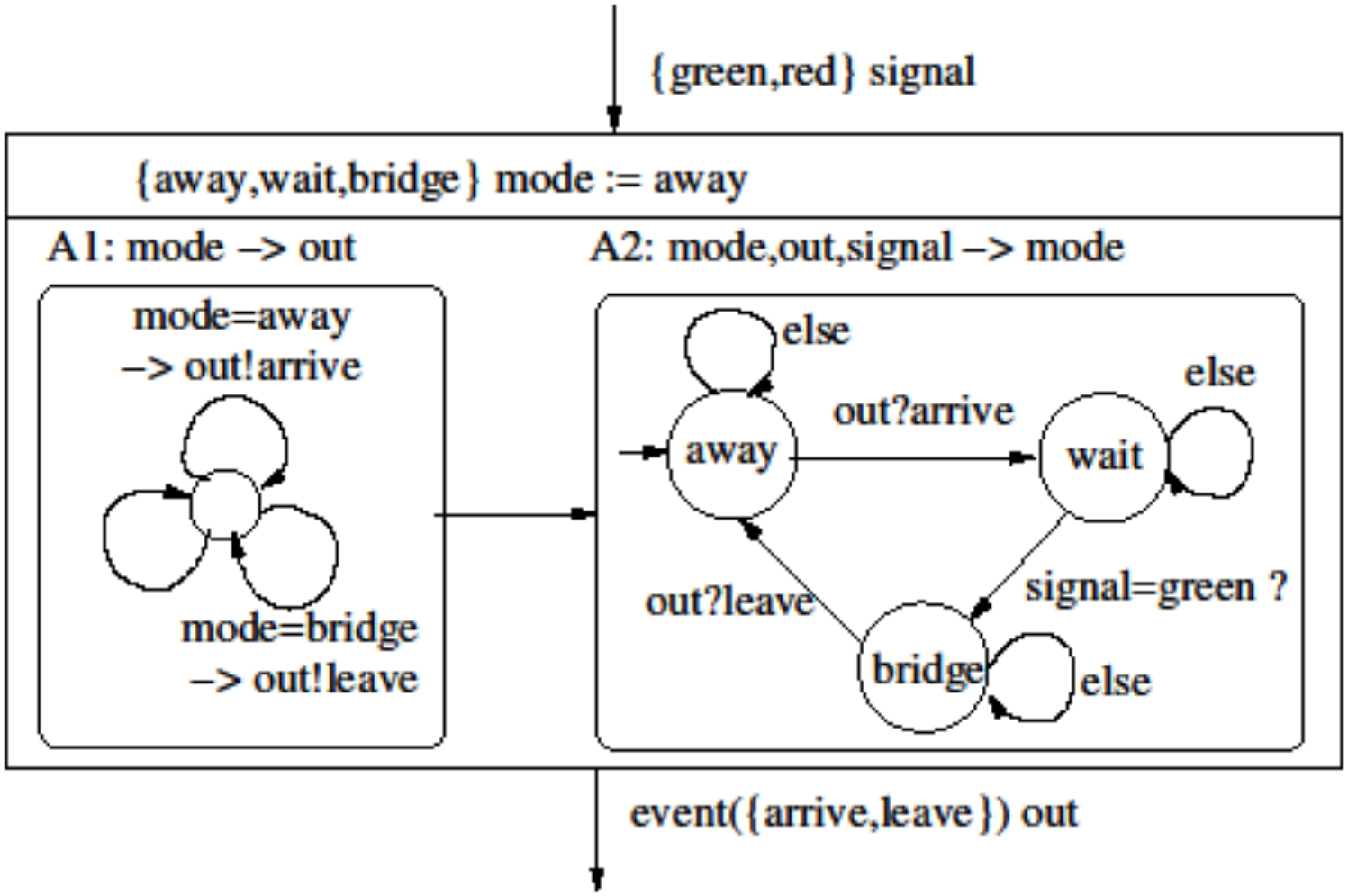
Railroad Controller Example



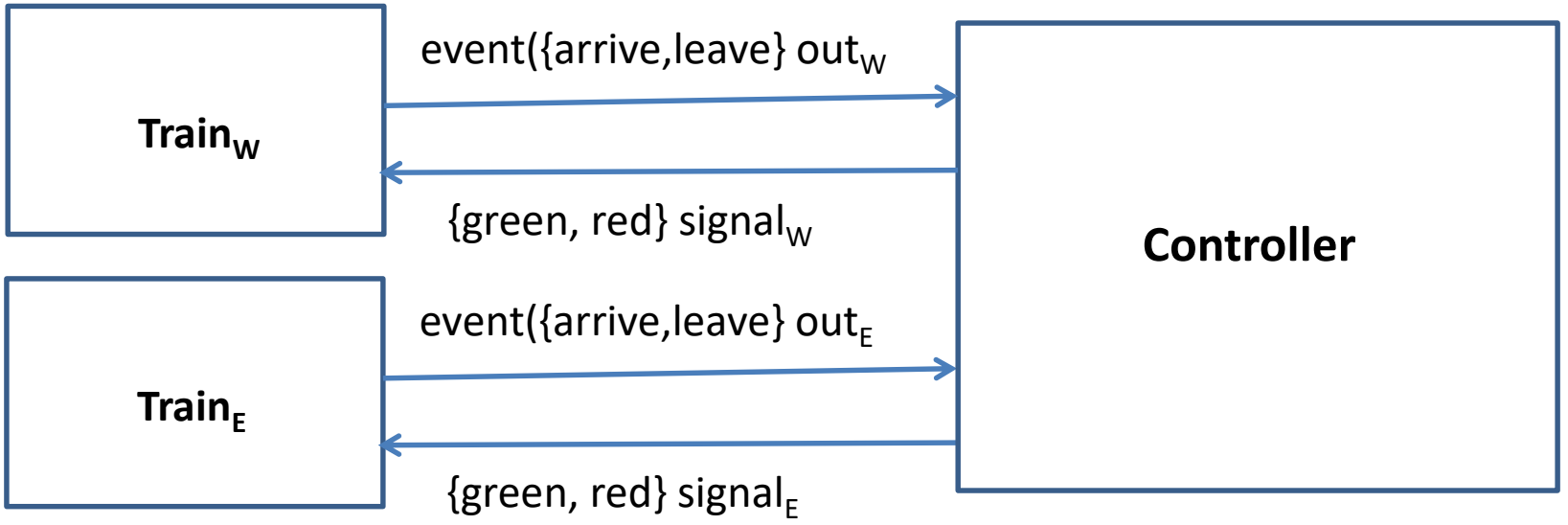
Train Model

- From the perspective of the controller, train is initially far away
- Train can be away for an arbitrarily long period
- When the train gets close, it communicates with the controller via an event, say, arrive, and now it is in a different state, say, wait
- When near, train is monitoring the signal:
  - If the signal is green, it enters the bridge
  - If the signal is red, it continues to wait
- A train can stay on bridge for a duration that is no exactly known (and not directly under the control of the traffic controller)
- When the train leaves the bridge, it communicates with the controller via an event, say, leave, and goes back to away state
- This behavior repeats: an away train may again request an entry
- Both trains have symmetric behavior

Synchronous Component Train



Controller Design Problem



Safety Requirement: Following should be an invariant:

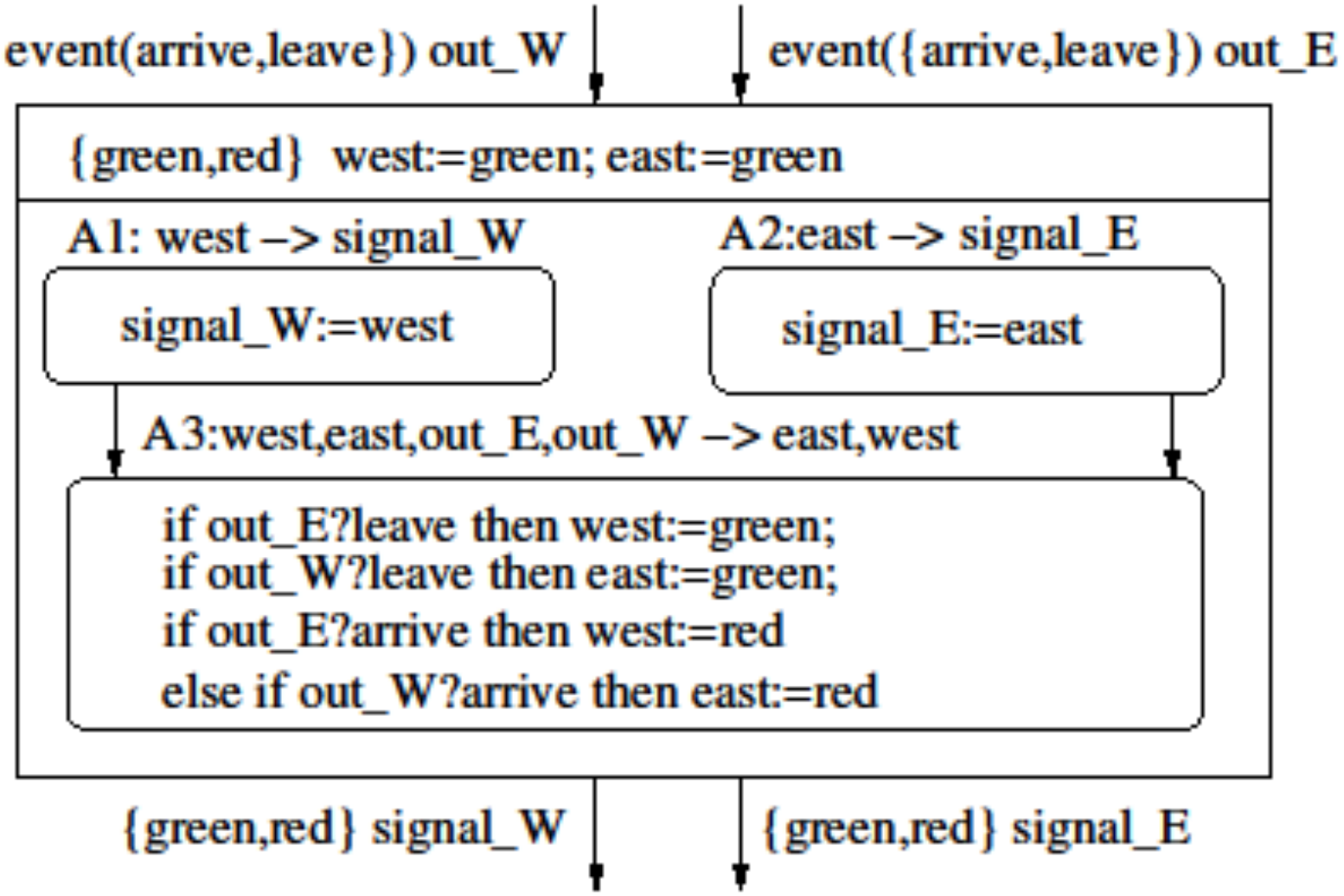
~ (mode$_W$ = bridge & mode$_E$ = bridge)

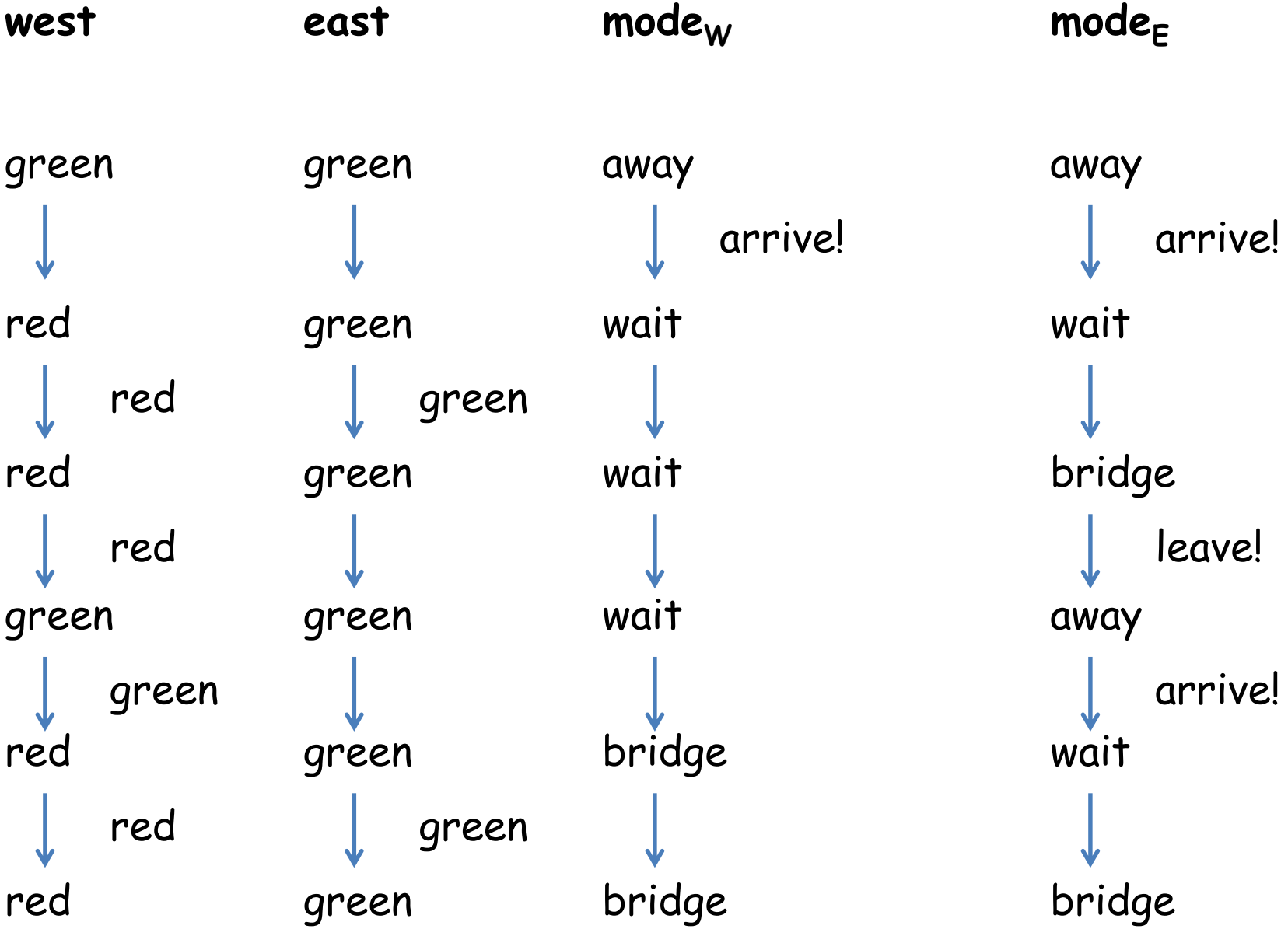
Trains should not be on bridge simultaneously

First Attempt at Controller Design

- Controller maintains state variables to track the state of each signal
- Both state variables are initially green
- Set the output signals based on the corresponding state vars
- If a train arrives, then update the opposite signal var to red to block the other train from entering
- If a train leaves, reset the opposite signal var to green
- What happens if both trains arrive simultaneously?
- Give priority to east train: set west signal var to red

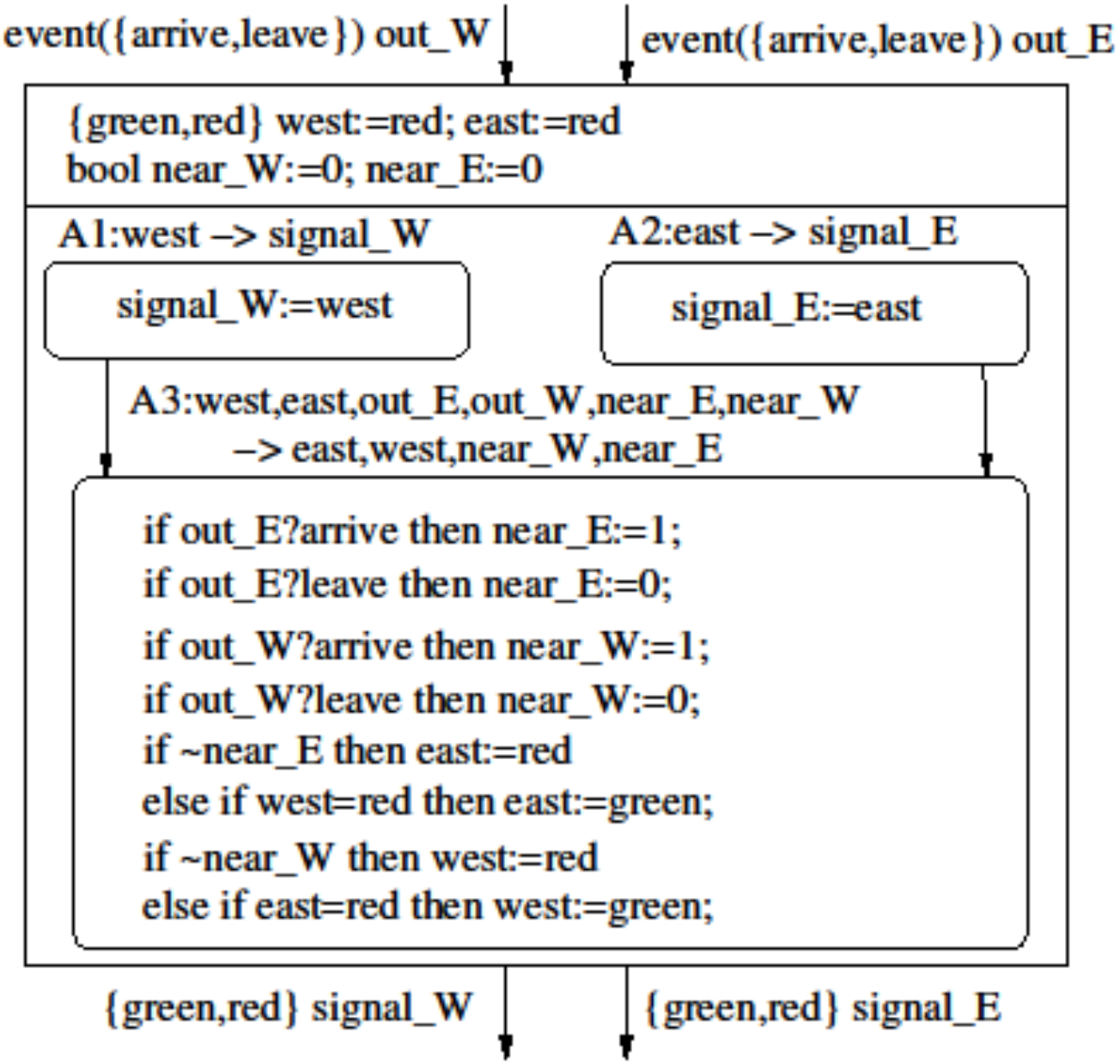
Synchronous Component Controller1




west
east
mode$_W$
mode$_E$
green
green
away
away
arrive!
arrive!
red
green
wait
wait
red
green
red
green
wait
bridge
red
green
wait
leave!
green
green
wait
away
green
green
arrive!
red
green
bridge
wait
red
green
red
green
bridge
bridge

Second Attempt at Controller Design

- What went wrong the first time? Controller did not remember whether a train was waiting at each entrance
- Boolean variable $near_W$ remembers whether the west train wants to use the bridge
  - Initially 0
  - When the west train issues arrive, changed to 1
  - When the west train issues leave, reset back to 0
- Invariant: $mode_W$ = away if and only if $near_W = 0$
- Variable $near_E$ is symmetric
- Let's also now keep both signals red by default
- A signal is changed to green if the corresponding train is near, the other signal is not red, and changed back to red when train is away
- Need still to resolve simultaneous arrivals by preferring one train


event({arrive,leave}) out_W
event({arrive,leave}) out_E
{green,red} west:=red; east:=red
bool near_W:=0; near_E:=0
A1:west -> signal_W
A2:east -> signal_E
signal_W:=west
signal_E:=east
A3:west,east,out_E,out_W,near_E,near_W
-> east,west,near_W,near_E
if out_E?arrive then near_E:=1;
if out_E?leave then near_E:=0;
if out_W?arrive then near_W:=1;
if out_W?leave then near_W:=0;
if ~near_E then east:=red
else if west=red then east:=green;
if ~near_W then west:=red
else if east=red then west:=green;
{green,red} signal_W
{green,red} signal_E

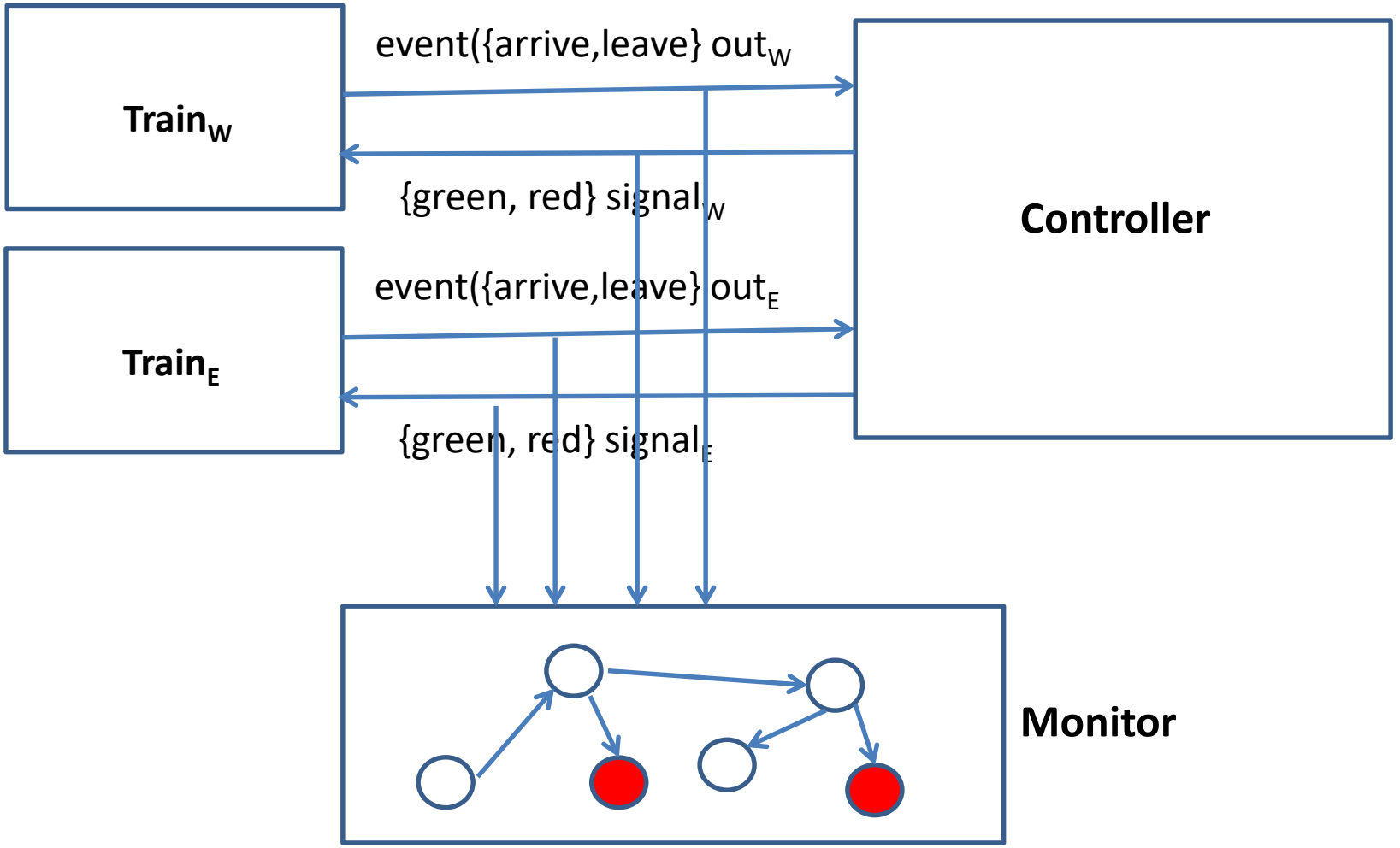
Properties of Controller2

- The system RailRoadSystem2 = Controller2 || TrainW || TrainE satisfies the safety invariant
- What about some additional properties?
  1. If the west train is waiting then west signal will eventually become green
  2. If the west train is waiting for its signal to turn green, other train should be allowed on bridge no more than once
- Requirement 1 is a "liveness" requirement and will be addressed in Chapter 4
- Requirement 2 is a safety property: its violation can demonstrated by a (finite) execution in which east train enters, leaves, and enters again while west train keeps waiting with its signal red
- But cannot be encoded as an invariant on system state variables

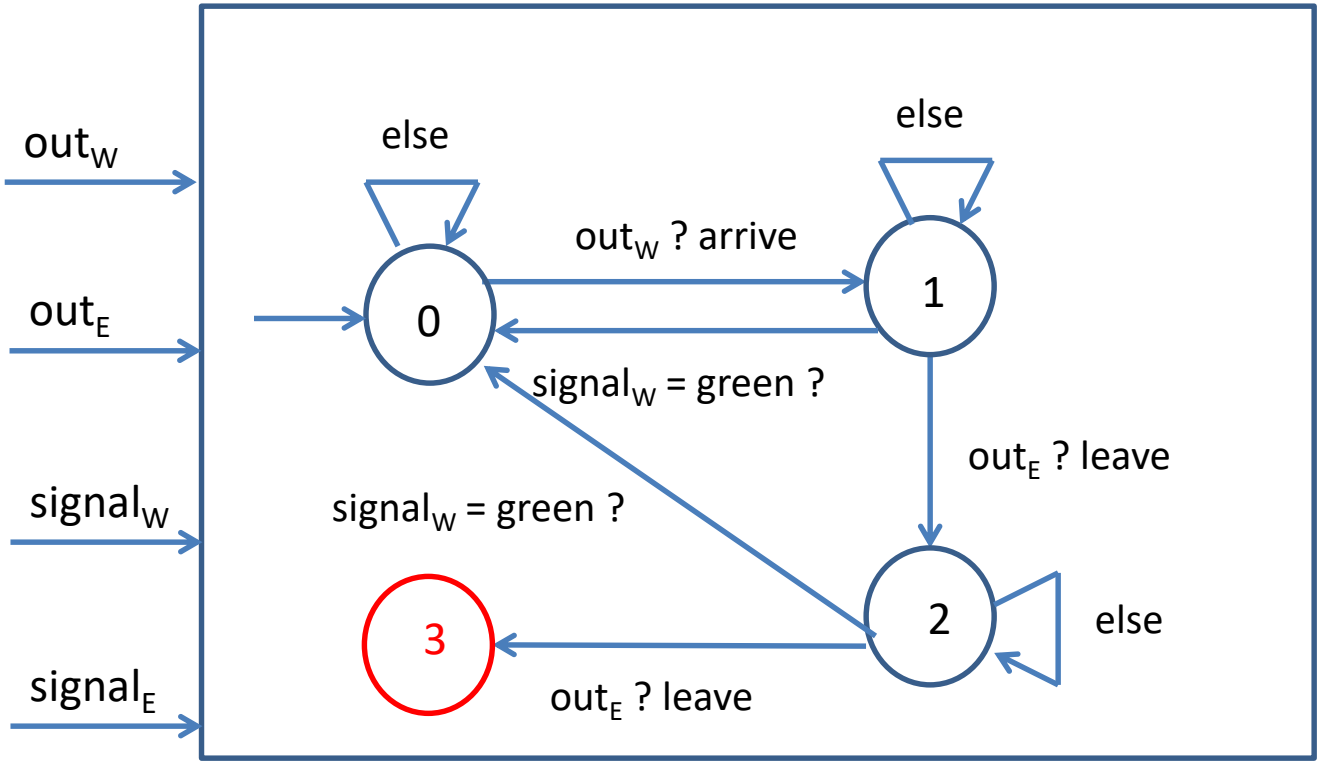
Safety Monitor

- Monitor M for a system observes its inputs/outputs, and enters an error state if undesirable behavior is detected
- Monitor M is specified as extended state machine
  1. The set of input variables of M = input/output variables of system being monitored
  2. An output of M cannot be an input to system (Monitor does not influence what the system does)
  3. A subset F of modes of state-machine declared as accepting
- Undesirable behavior: An execution that leads monitor state to F
- Safety verification: Check whether (monitor.mode not in F) is an invariant of System C || M

Safety Monitors



Monitor to check "fairness" for railroad



Error execution:

As west train waits, east train is allowed on bridge twice