

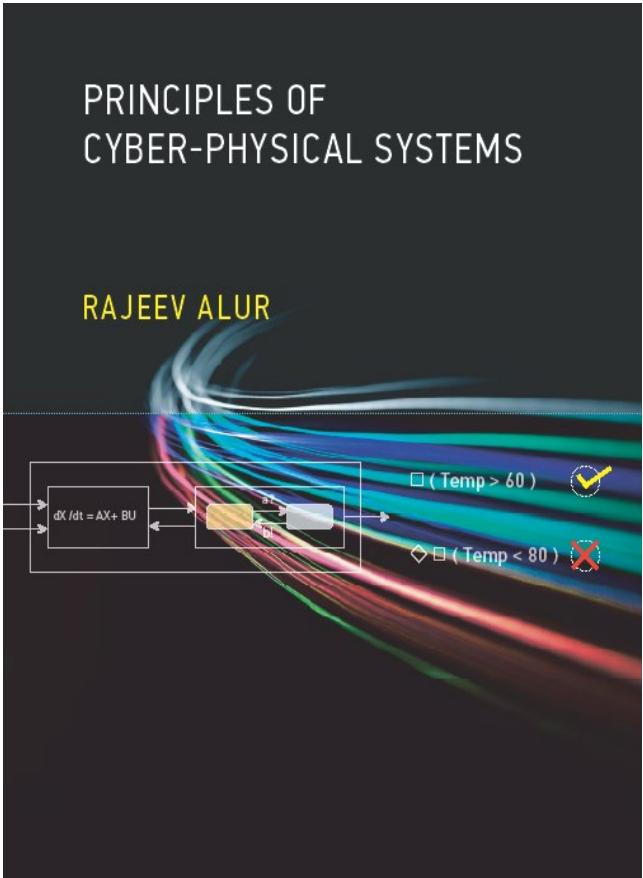
Models and Tools for Cyber-Physical Systems

Chapter 1: Introduction

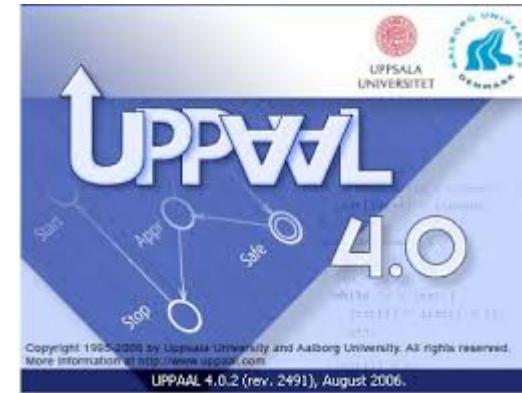
Instructors: Martijn Goorden, Kim G. Larsen,
Christian Schilling, Max Tschaikowski
{mgoorden,kgl,christianms,tschaikowski}@cs.aau.dk

Slides courtesy of Rajeev Alur
alur@cis.upenn.edu

Models and Tools for Cyber-Physical Systems



Reference book



MATLAB®
&SIMULINK®

Tools

Course Organization and Contents

Session	Activity	Contents	Instructors
1	Lecture Exercises 1	Introduction	
2	Lecture Exercises 2	Synchronous Model chapters 1, 2	
3	Lecture Exercises 3	Safety Requirements	Christian
4	Lecture Exercises 4	Asynchronous Model	
5	Lecture Exercises 5	chapters 3, 4	
6	Lecture Exercises 6	Timed Model	
7	Lecture Exercises 7	Real Time Scheduling	Kim
8	Lecture Exercises 8	chapters 7, 8	

Session	Activity	Contents	Instructors
9	Mini project	Uppaal	Christian, Martijn
10	Lecture Exercises 9	Dynamical Systems	
11	Lecture Exercises 10	Hybrid Systems	Max
12	Lecture Exercises 11	chapters 6, 9	
13	Mini project	Matlab	Martijn, Max

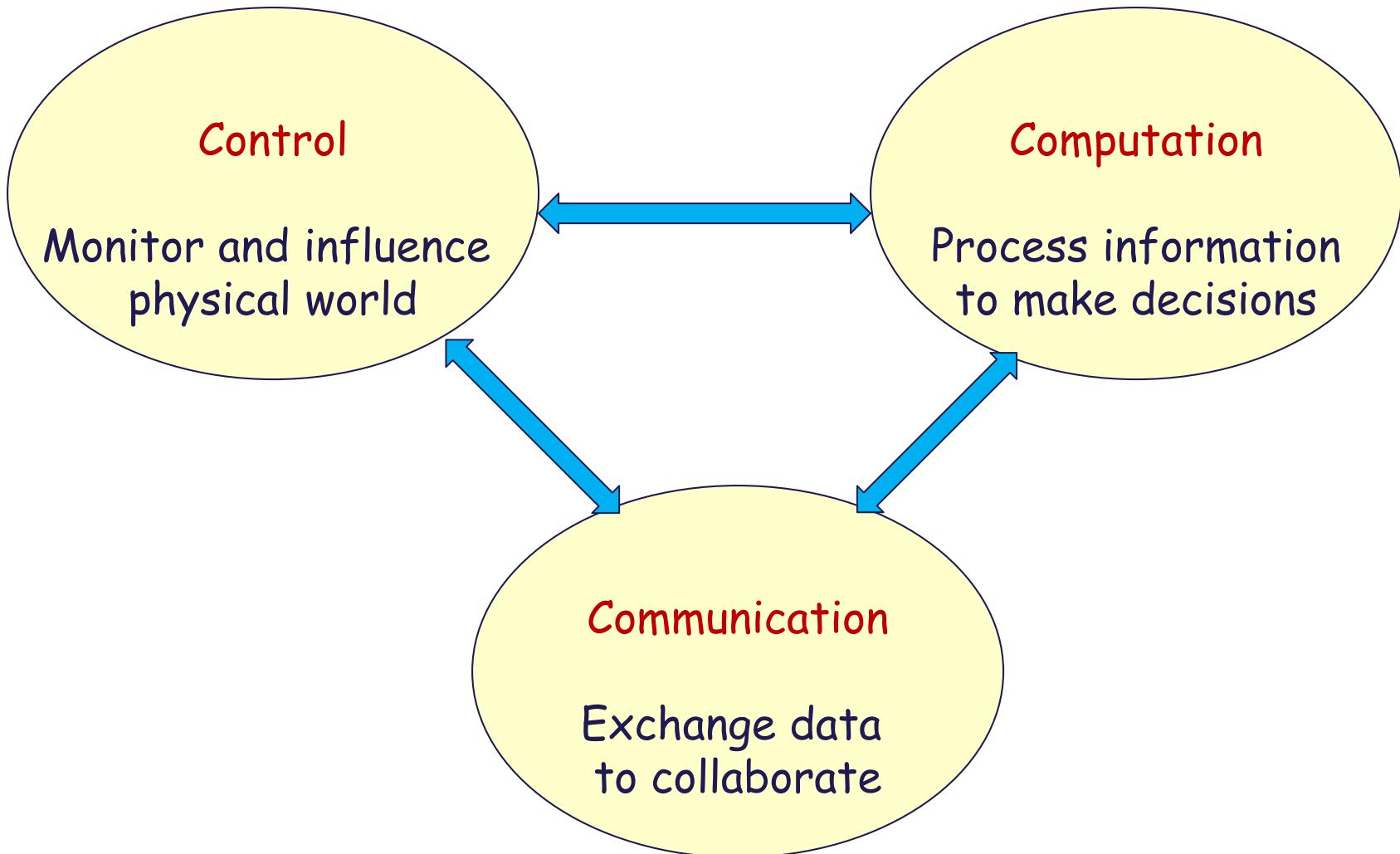
From Desktops to Cyber-Physical Systems

- Traditional computers: Stand-alone devices running software applications (e.g., data processing)
- Traditional controllers: Devices interacting with physical world via sensors and actuators (e.g., thermostat)
- Embedded systems
 - Special-purpose systems with integrated microcontroller/software
 - Cameras, watches, washing machines...

Embedded Systems Everywhere!



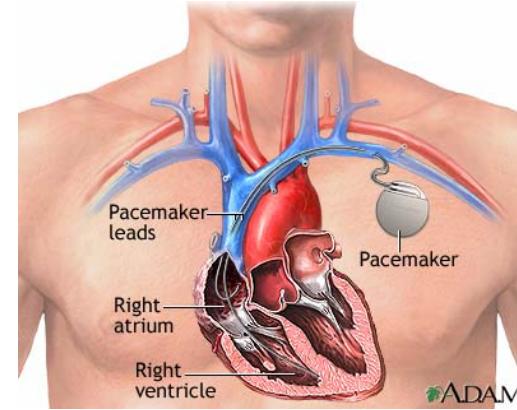
Cyber-Physical Systems



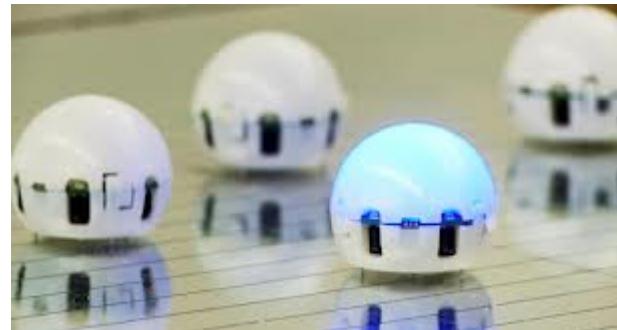
Cyber-Physical Systems



Driverless cars



Medical devices



Coordinating robots

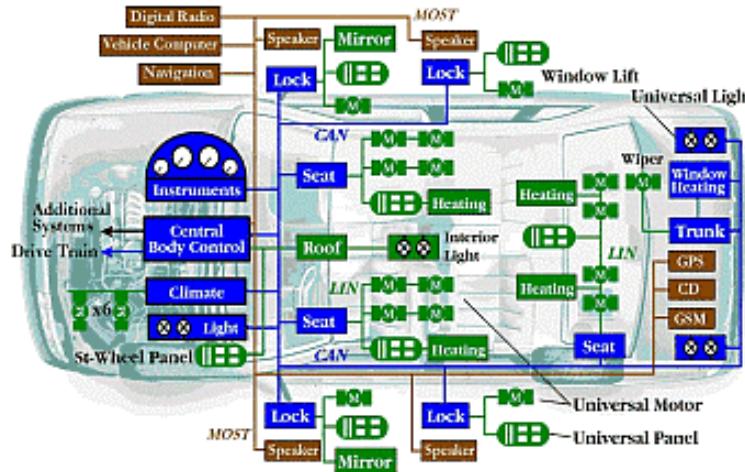
Ariane 5 Explosion



"It took the European Space Agency 10 years and \$7 billion to produce Ariane 5. All it took to explode that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, was a small computer program trying to stuff a 64-bit number into a 16-bit space"

A bug and a crash, J. Gleick, New York Times, Dec 1996

Prius Brake Problems Blamed on Software Glitches



"Toyota officials described the problem as a "disconnect" in the vehicle's complex anti-lock brake system (ABS) that causes less than a one-second lag. With the delay, a vehicle going 60 mph will have traveled nearly another 90 feet before the brakes begin to take hold"

CNN Feb 4, 2010

Software: The Achilles' Heel

Software everywhere means bugs everywhere

2002 study by NIST: Software bugs cost US economy \$60 billion annually (0.6% of GDP)

2017 Tricentis study: \$1.7 trillion in 314 companies

Lack of trust in software as technology barrier

Would you use an autonomous software-controlled round-the-clock monitoring and drug-delivery device?

Grand challenge for computer science

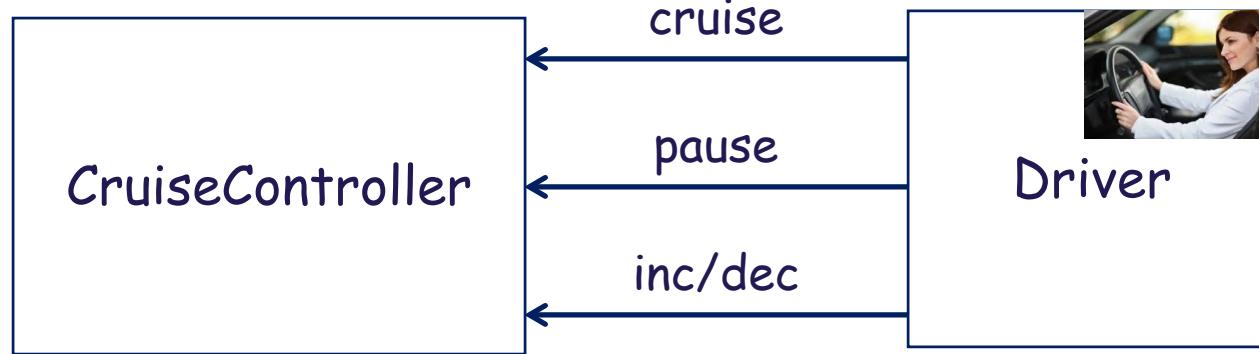
Technology for designing **reliable** cyber-physical systems

Case study: Design a Cruise Controller



The goal of a cruise controller is to automatically adjust the speed of a car so that it matches the speed desired by the driver

Interfaces for Components: Inputs and Outputs



Driver interacts with the system using 4 buttons:

Cruise button to turn cruise control on or off

Pause button to suspend/restart its operation

Inc and Dec buttons to increment or decrement desired speed

Interfaces for Components: Inputs and Outputs

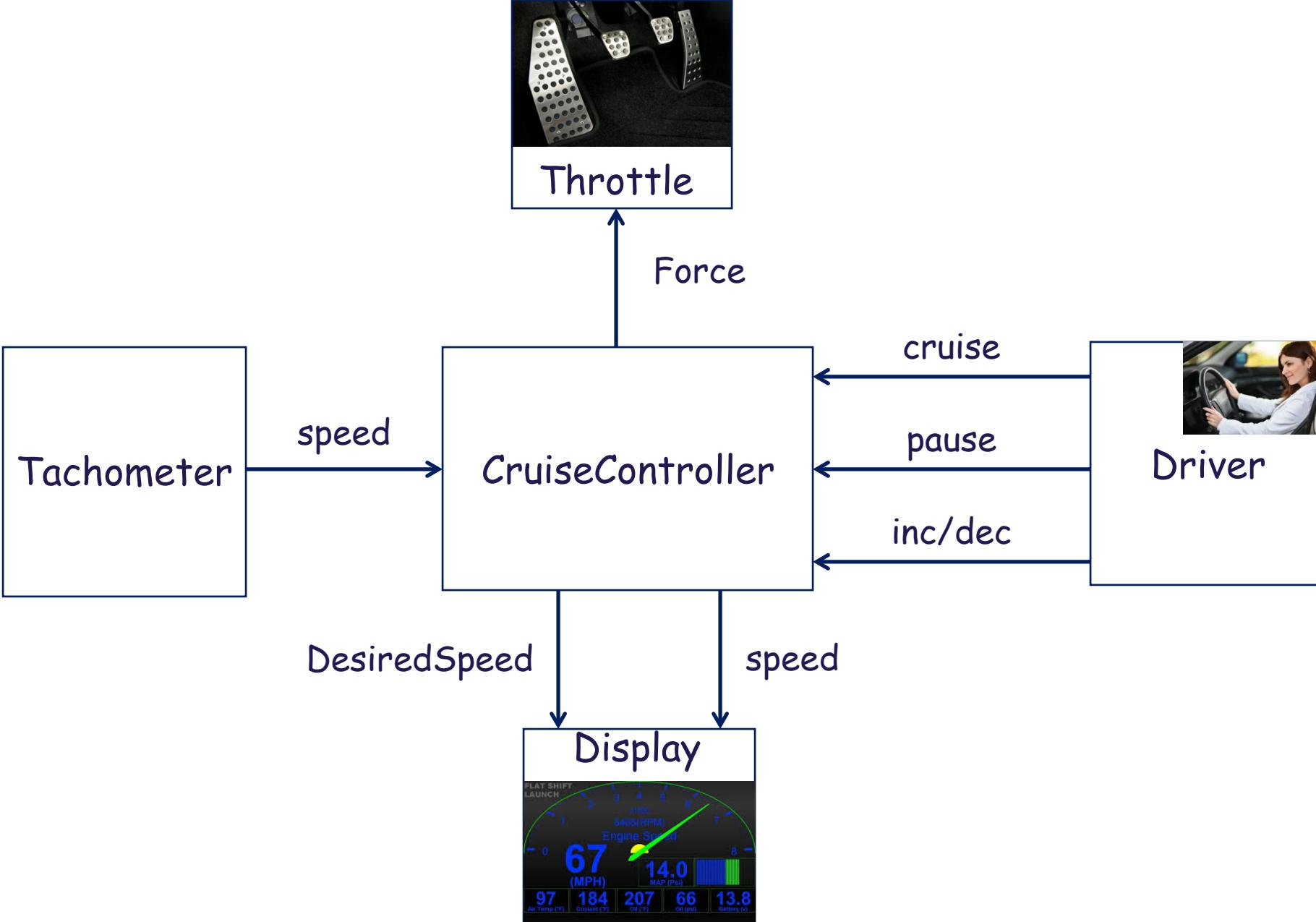


What other information does the cruise controller need ?
And who supplies it?

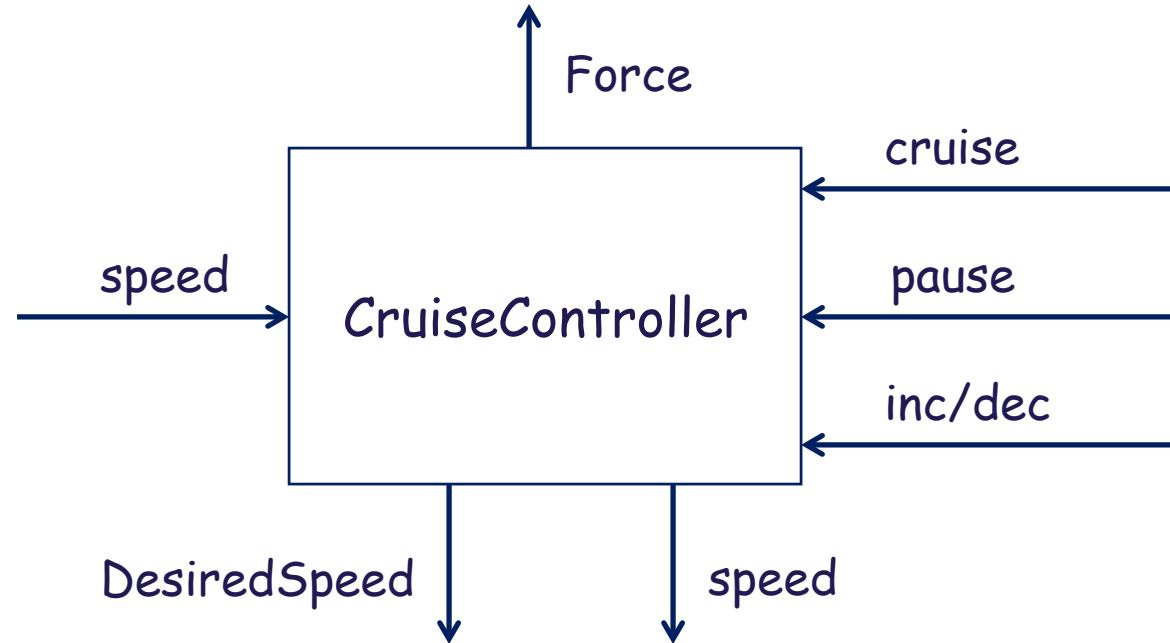
Interfaces for Components: Inputs and Outputs



What should be the outputs of the cruise controller?
And who needs these outputs?

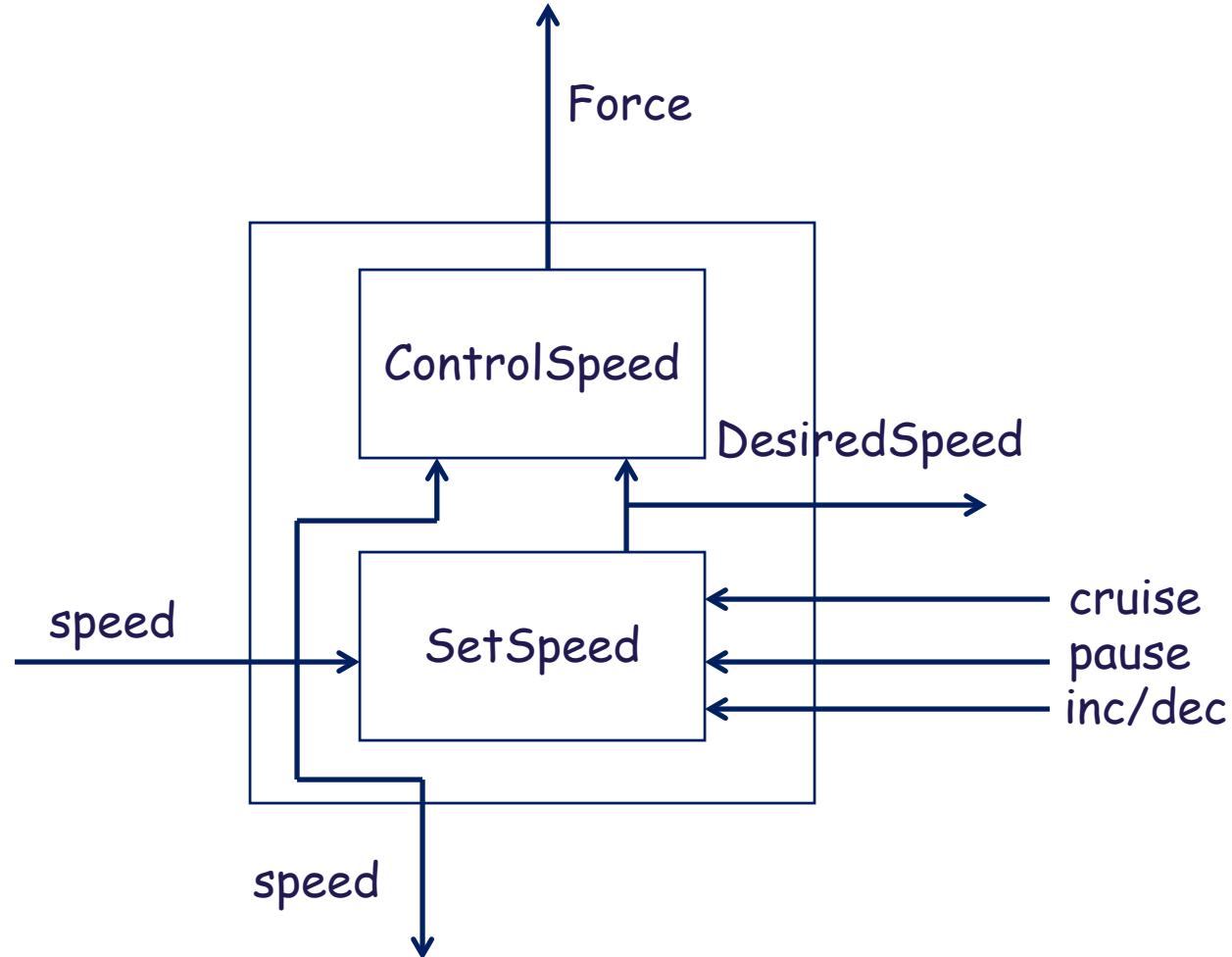


Compositional Design

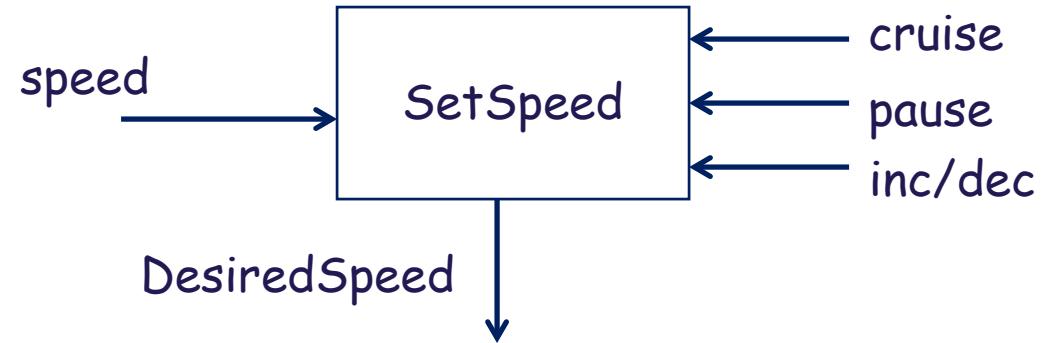


How to break up the computation of the cruise controller into subtasks?

Decomposing the Cruise Controller

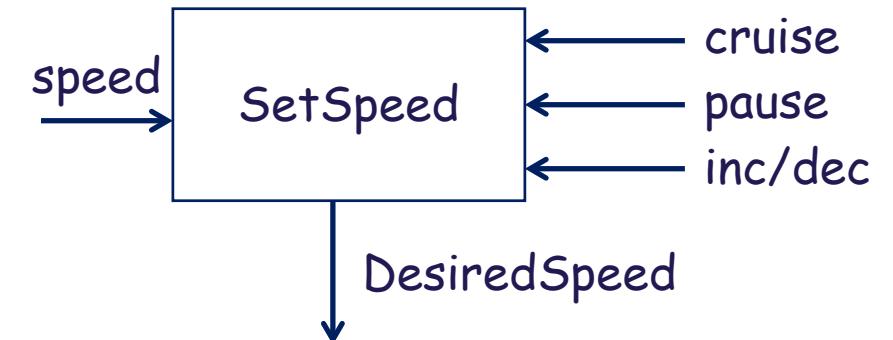
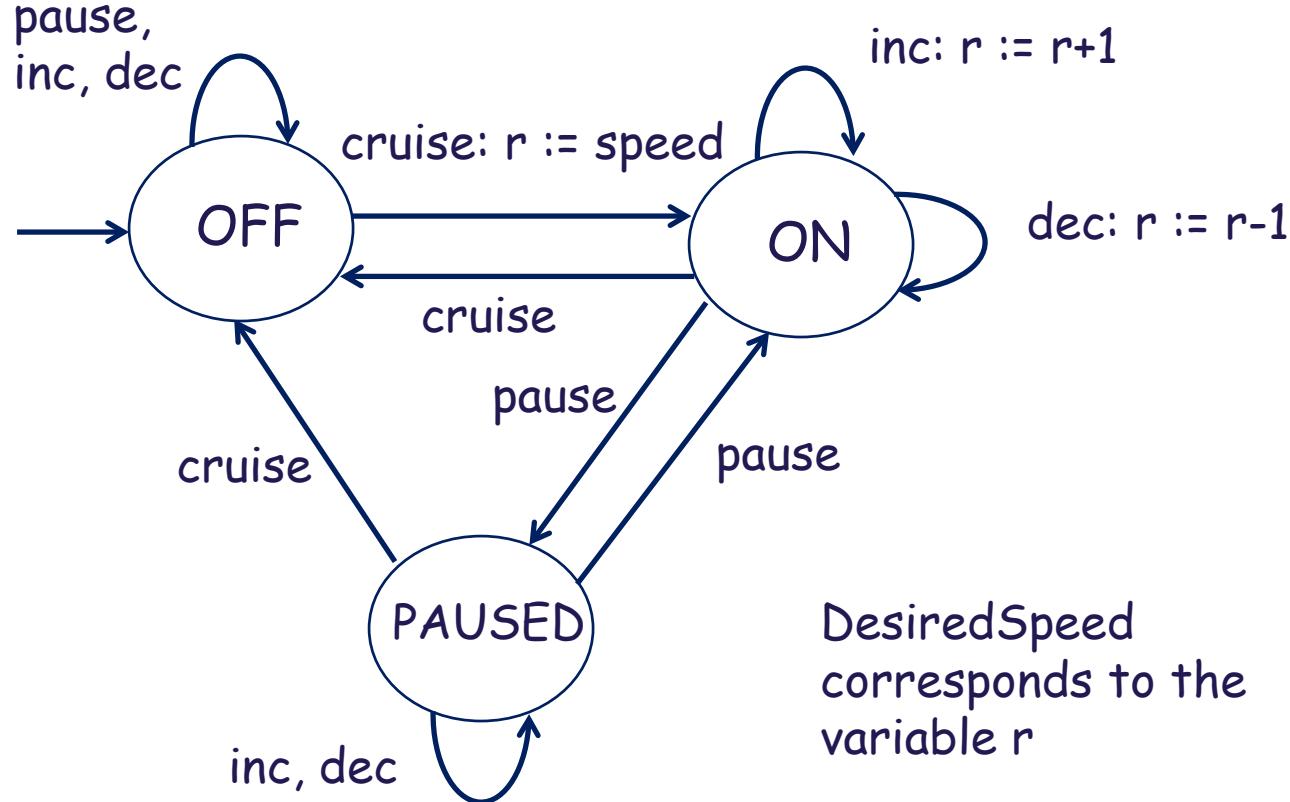


Designing SetSpeed Component

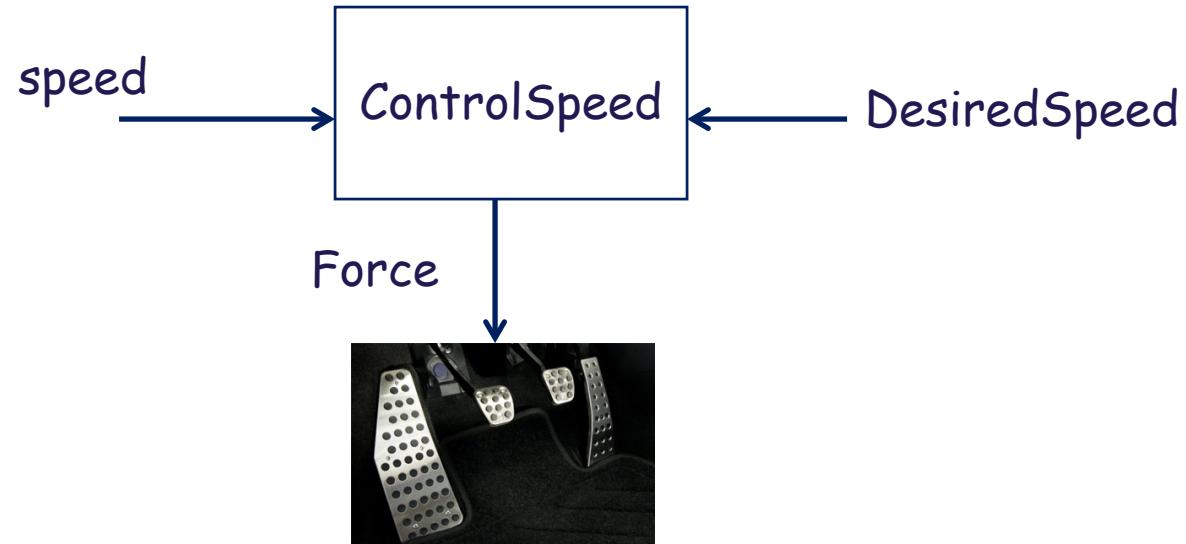


Goal: Compute the desired cruising speed in response
to the commands from the driver

Designing SetSpeed: State Machine

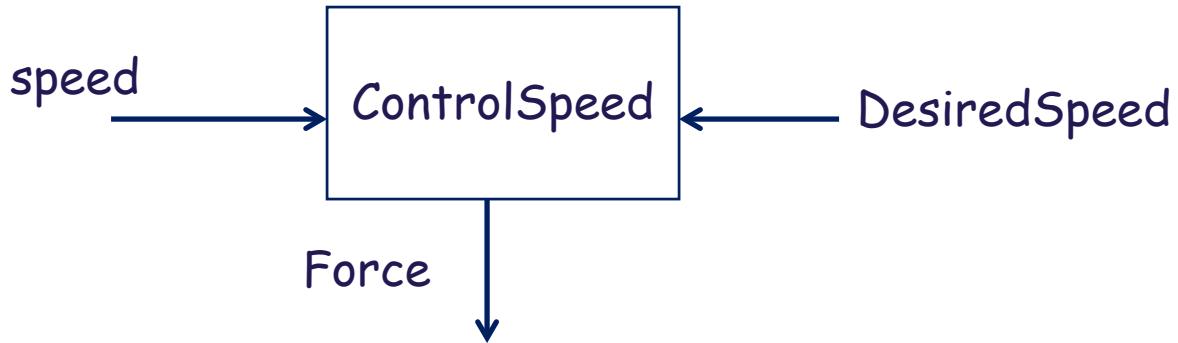


Designing ControlSpeed Component



Goal: Determine the force to be applied to throttle
so that speed becomes equal to DesiredSpeed

Capturing Requirements



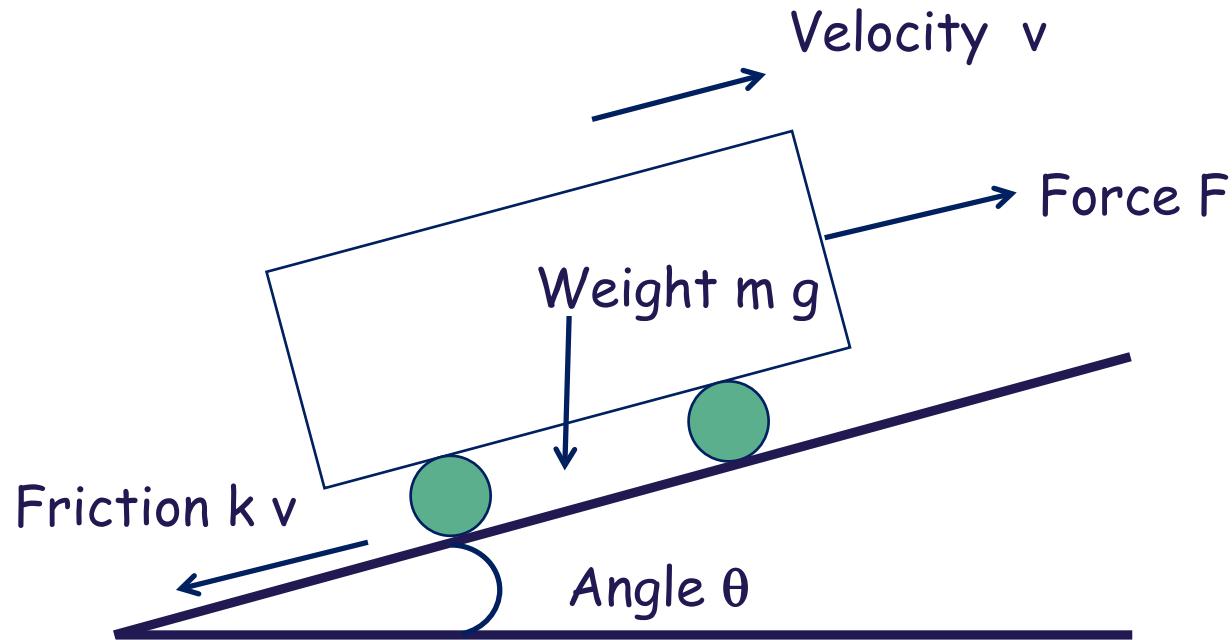
Requirements: Mathematically precise description of what a system is supposed to do.

Writing requirements is key to ensuring reliability of systems

Requirement 1: Actual speed eventually converges to desired speed

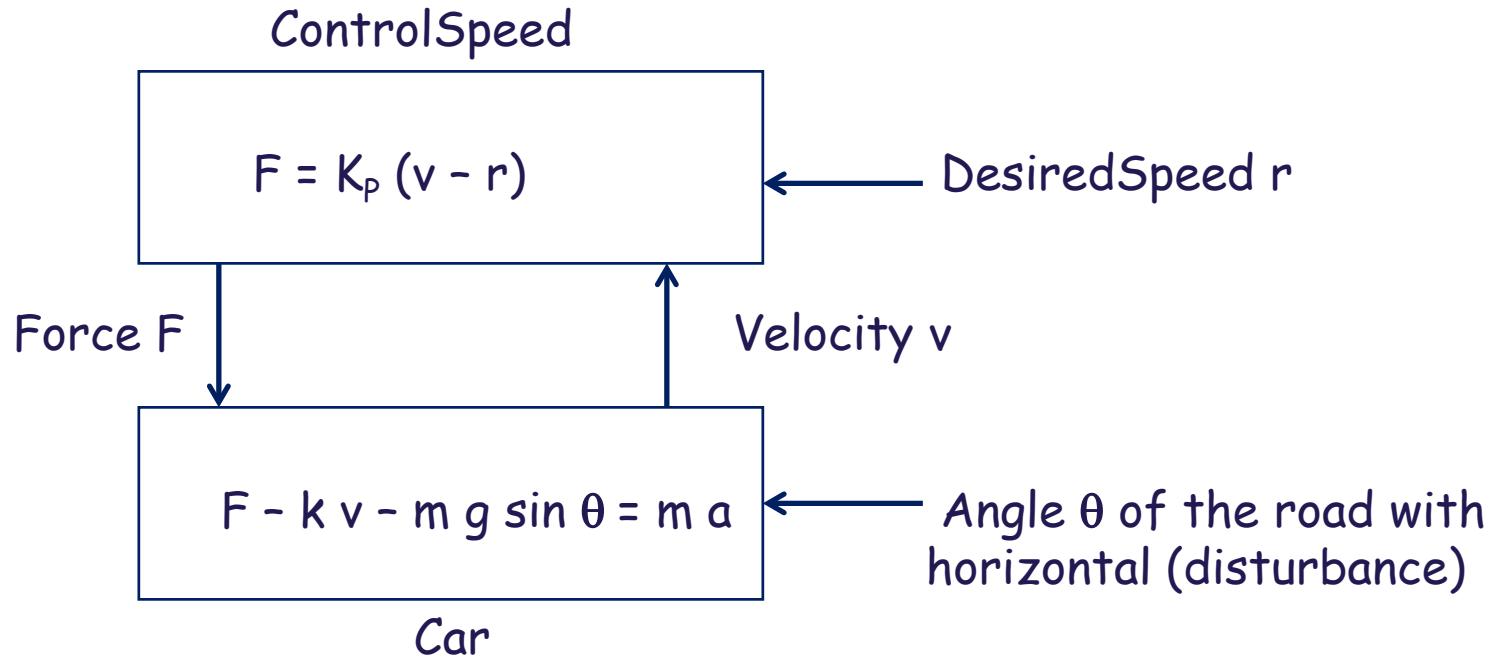
Requirement 2: Speed of the car stays "stable"

A Bit of Physics: Modeling a Car



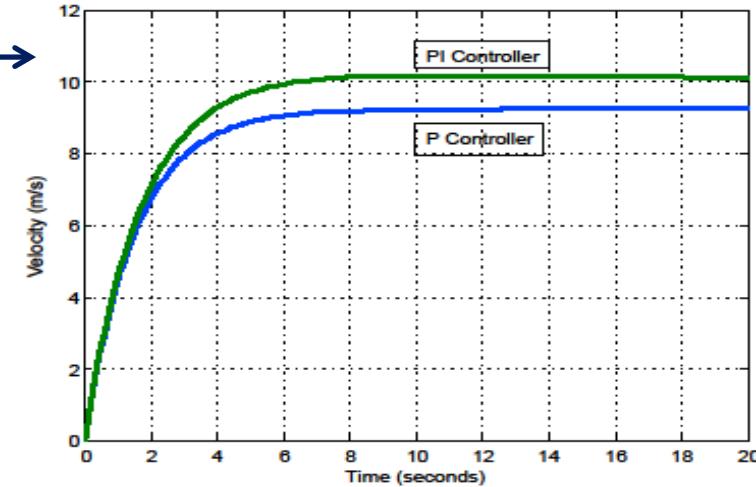
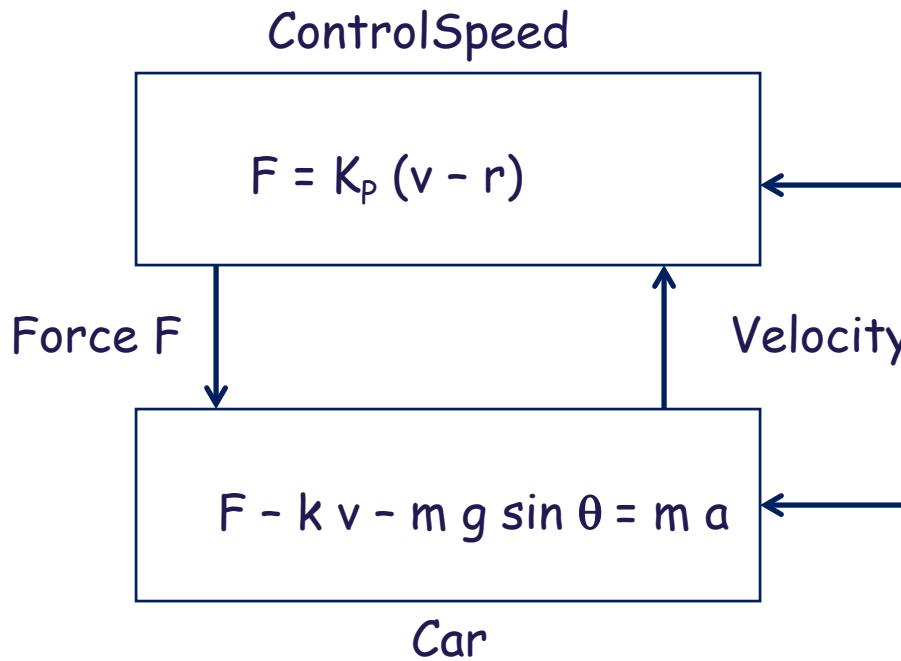
Newton's law of motion gives
 $F - k v - m g \sin \theta = m a$

ControlSpeed Component



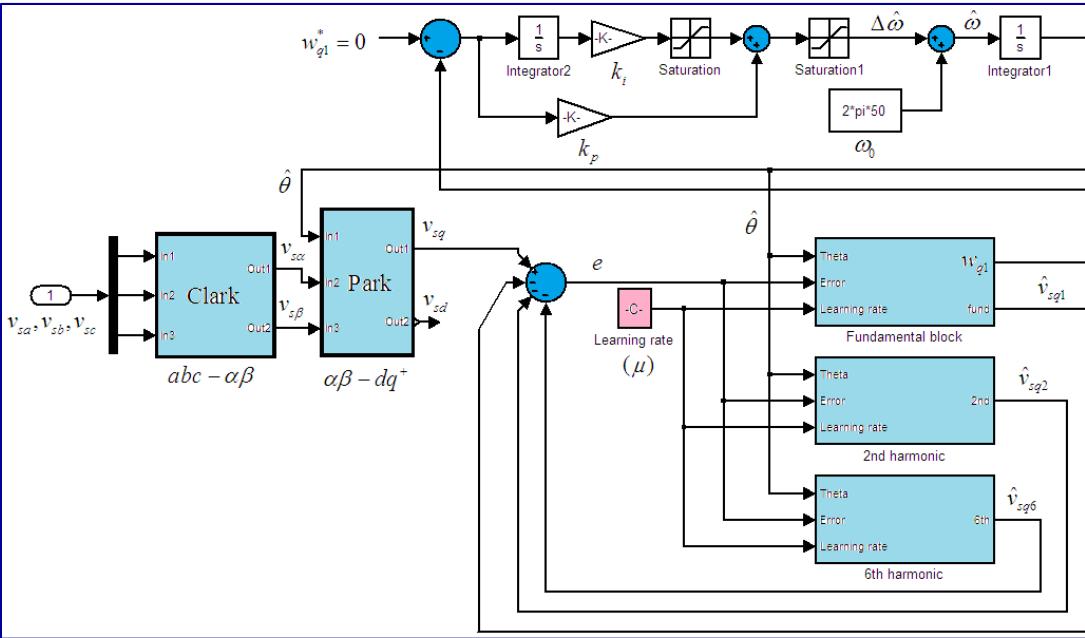
Control Theory: Mathematical techniques to compute force (F) as a function of velocity (v) and desired speed (r)

Does our controller work?



Verification tools: Allow you to check if system model indeed works as expected, that is, satisfies the requirements

Model-based design != Coding



```

1  #make sure that we are allowed to recurse many times
2  import sys
3  sys.setrecursionlimit(20000)
4
5
6  def probOfStreak(numCoins, minHeads, headProb, saved=None):
7      #Computes the probability (i.e. S[n,k]) of getting a run of minHeads
8      #(i.e. K) or more heads in a row out of numCoins
9      #independent coin tosses where the probability of getting a
10     #heads each time is headProb (i.e. p) and the
11     #probability of a tails is 1-headProb (i.e. q).
12
13     #We will be using the recursion:
14     #S[N,K] = p^K + sum_{j=1,K} p^(j-1) (1-p) S[N-j,K]
15     #As well as the base cases S[0,k] = 0 and S[N,K] = 0 for K>N
16
17     #if it's our first call, allocate a hash table to store saved values
18     if saved == None: saved = {}
19
20     #get a unique identifier for the value that they want to compute
21     ID = (numCoins, minHeads, headProb)
22
23     #if it has been computed before, just return the precomputed value.
24     #there is no point in wasting time computing the same thing again.
25     if ID in saved: return saved[ID]
26     else: #if it's never been computed before
27         #handle the base case where we have no coins or where we have
28         #more heads we are looking for than we have coins
29         if minHeads > numCoins or numCoins <= 0:
30             result = 0
31         else:
32             #use our recursive relationship to compute S[n,k] by
33             #breaking it into a sum of terms involving S[n-j,k] for 1<=j<=k
34             result = headProb**minHeads  #S[n,k] = p^K + ...
35             #S[n,k] = ... + sum_{j=1,K} p^(j-1) (1-p) S[n-j,k]
36             for firstTail in xrange(1, minHeads+1):
37                 pr = probOfStreak(numCoins-firstTail, minHeads, headProb, saved)
38                 result += (headProb**firstTail)*(1-headProb)*pr
39
40             #save the resulting value so that we can use it later, if need be
41             saved[ID] = result
42
43     #return the computed value
44     return result
45
46

```

Design using high-level block diagrams and state machines gets automatically compiled into low-level code !
Not only a model of the designed system, but also of its environment

Verification != Simulation/Testing



Program testing can be used to show the presence
of bugs, but never their absence!

Edsger W. Dijkstra

Formal Verification



- Goal: Establish that model satisfies requirements under all possible scenarios
- First challenge: Need formal definitions of "model" and "requirement" to make the problem mathematically precise
- Second challenge: Need verification techniques and tools

Course Topics

- Goal: Introduction to principles of design, specification, analysis and implementation of CPS
- Disciplines
 - Model-based design
 - Concurrency theory
 - Distributed algorithms
 - Formal specification
 - Verification techniques and tools
 - Control theory
 - Real-time systems
 - Hybrid systems
- Emphasis on mathematical concepts

Theme 1: Formal Models

- Mathematical abstractions to describe system designs
- Modeling formalisms
 - Synchronous models (Chapter 2)
 - Asynchronous models (Chapter 4)
 - Timed models (Chapter 7)
 - Continuous-time dynamical systems (Chapter 5)
 - Hybrid systems (Chapter 9)
- Modeling concepts
 - Syntax vs semantics
 - Composition
 - Input/output interfaces
 - Nondeterminism, fairness, ...

Theme 2: Specification and Analysis

- Formal techniques to ensure correctness at design time
- Requirements
 - Safety (invariants, monitors)
 - Liveness (temporal logic, automata over infinite sequences)
 - Stability
 - Schedulability
- Analysis techniques
 - Deductive: Inductive invariants and ranking functions
 - Enumerative and symbolic search for state-space exploration
 - Model checking
 - Linear-algebra-based analysis of dynamical systems
 - Verification of timed and hybrid systems

Theme 3: Model-based Design

- Design and analysis of illustrative computing problems
- Design methodology
 - Structured modeling (bottom-up, top-down)
 - Requirements-based design and design-space exploration
- Case studies
 - Distributed coordination: mutual exclusion, consensus, leader election
 - Communication: Reliable transmission, synchronization
 - Control design: PID, cruise controller
 - CPS: Pacemaker, obstacle avoidance for robots, multi-hop control network

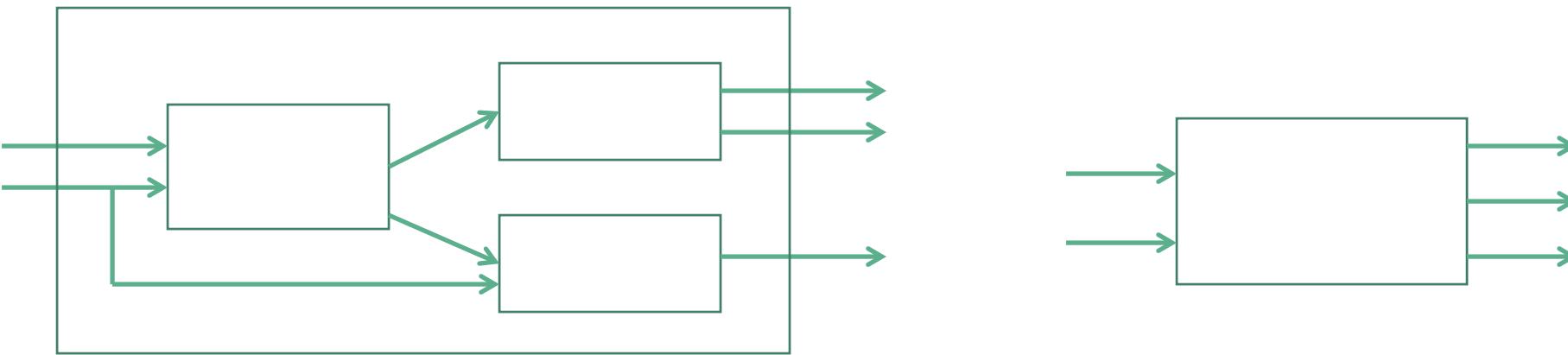
Models and Tools for Cyber-Physical Systems

Chapter 2: Synchronous Model

Instructors: Martijn Goorden, Kim G. Larsen,
Christian Schilling, Max Tschaikowski
`{mgoorden,kgl,christianms,tschaikowski}@cs.aau.dk`

Slides courtesy of Rajeev Alur
`alur@cis.upenn.edu`

Model-Based Design



❑ Block Diagrams

- Widely used in industrial design
- Tools: Simulink, Modelica, LabView, RationalRose...

❑ Key questions:

- What is the execution semantics?
- What is a base component?
- How do we compose components to form complex components?

Functional vs Reactive Computation

□ Classical model of computation: Functional

- Given inputs, a program produces outputs
- Desired functionality described by a mathematical function
- Examples: Sorting of names; shortest path in a weighted graph
- Theory of computation provides foundation
- Canonical models: Turing machine, lambda calculus, ...



□ Reactive

- System interacts with its environment via inputs and outputs in an ongoing manner
- Desired behaviors: which sequences of observed input/output interactions are acceptable?
- Example: Cruise controller in a car, coffee machine

Sequential vs Concurrent Computation

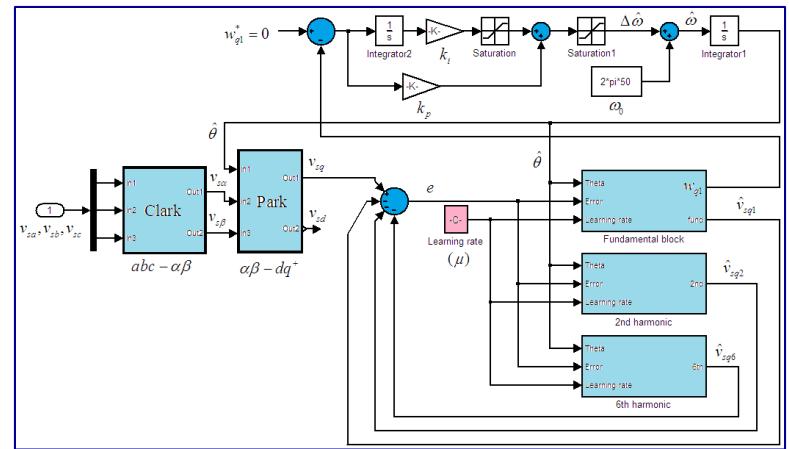
□ Classical model of computation: Sequential

- A computation is a sequence of instructions executed one at a time
- Well understood and canonical model: Turing machines

□ Concurrent Computation

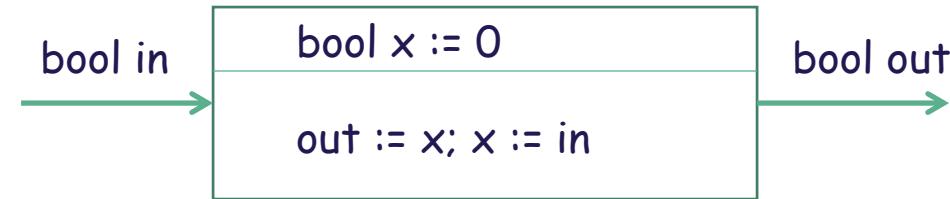
- Multiple components/processes exchanging information and evolving concurrently
- Logical vs physical concurrency
- Broad range of formal models for concurrent computation
- Key distinction: Synchronous vs Asynchronous

Synchronous Models



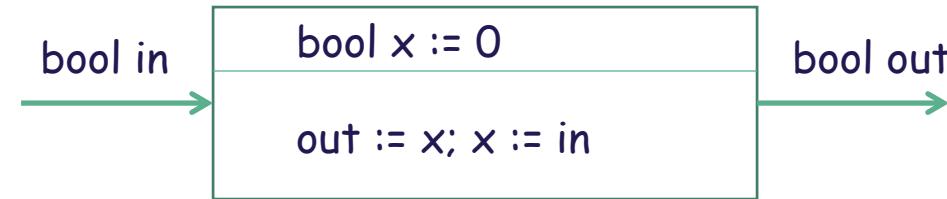
- All components execute in a sequence of (logical) rounds in lock-step
- Example: Component blocks in a digital hardware circuit
 - Clock drives all components in a synchronized manner
- Key idea: Design system using a synchronous round-based computation
 - Benefit: Design is simpler
 - Challenge: Ensure synchronous execution even if implementation platform is not single-chip hardware

First Example: Delay



- Input variable: `in` of type Boolean
- Output variable: `out` of type Boolean
- State variable: `x` of type Boolean
- Initialization of state variables: assignment `x := 0`
- In each round, in response to an input, produce output and update state by executing the update code: `out := x; x := in`

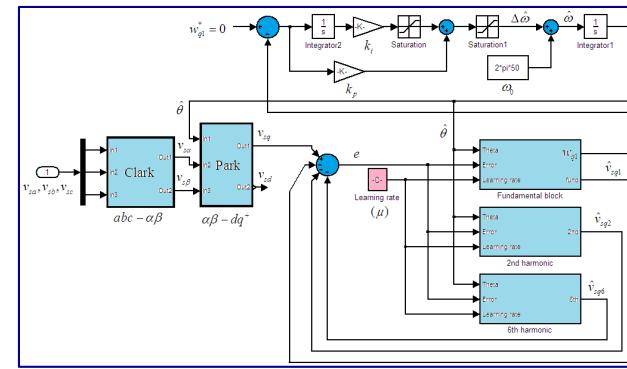
Delay: Round-based Execution



- Initialize state to 0
- Repeatedly execute rounds. In each round:
 - Choose a value for the input variable **in**
 - Execute the update code to produce output **out** and change state
- Sample execution:

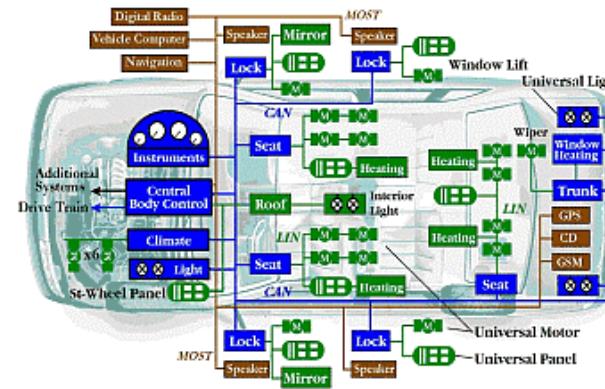


Synchrony Hypothesis



- Assumption: Time needed to execute the update code is negligible compared to delay between successive input arrivals
- Logical abstraction:
 - Execution of update code takes no time
 - Production of outputs and reception of inputs occurs at same time
- When multiple components are composed, all execute synchronously and simultaneously
- Implementation must ensure that this design-time assumption is valid

Components in an Automobile

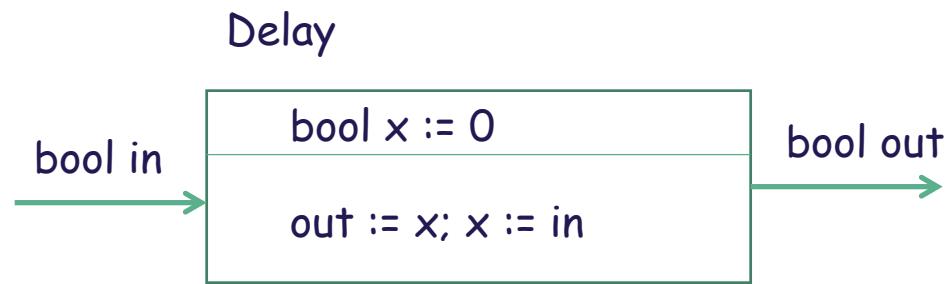
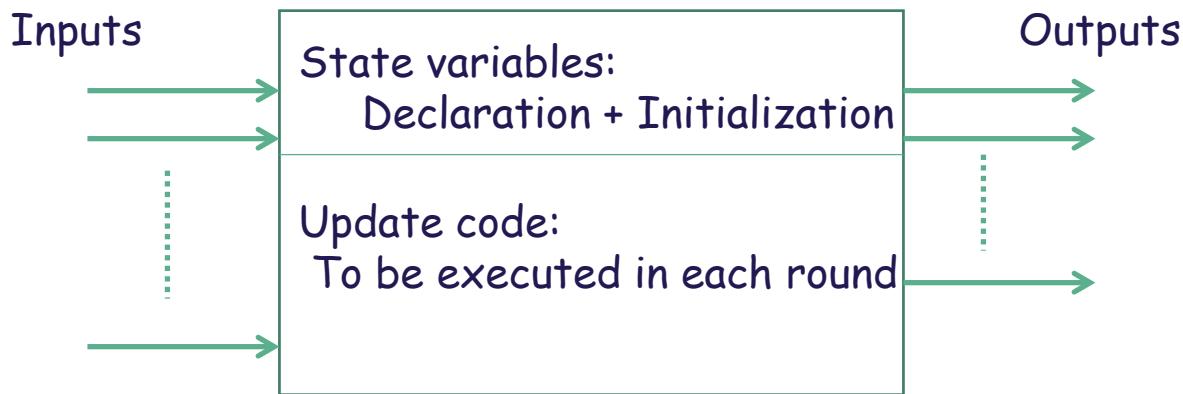


- ❑ Components need to communicate and coordinate over a shared bus
- ❑ Design abstraction: Synchronous **time-triggered communication**
 - Time is divided into slots
 - In each slot, exactly one component sends a message over the bus
- ❑ CAN protocol implements time-triggered communication

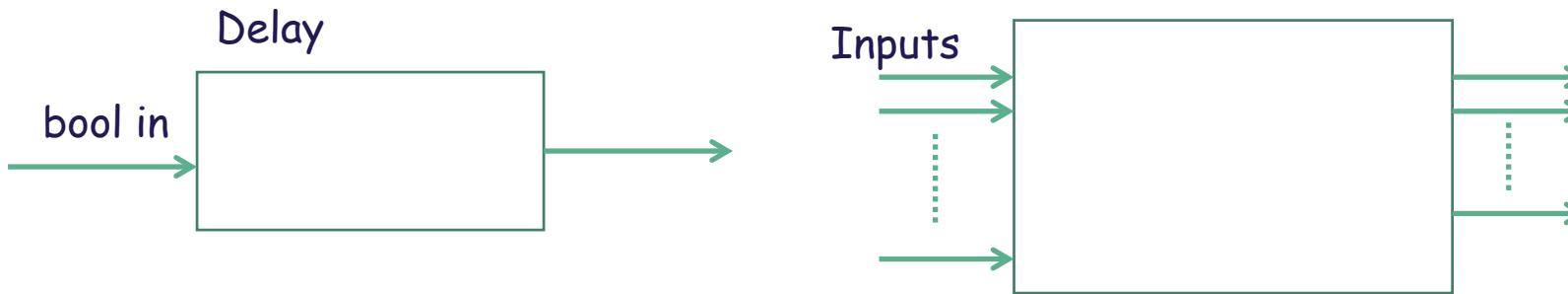
Model Definition

- Syntax: How to describe a component?
 - Variable declarations, types, code describing update ...
- Semantics: What does the description mean?
 - Defined using mathematical concepts such as sets, functions ...
- Formal: Semantics is defined precisely
 - Necessary for tools for analysis, compilation, verification ...
 - Defining formal semantics for a "real" language is challenging
 - But concepts can be illustrated on a "toy" modeling language
- Our modeling language: Synchronous Reactive Components
 - Representative of many "academic" proposals
 - Industrial-strength synchronous languages
Esterel, Lustre, VHDL, Verilog, Stateflow...

Synchronous Reactive Component (SRC)

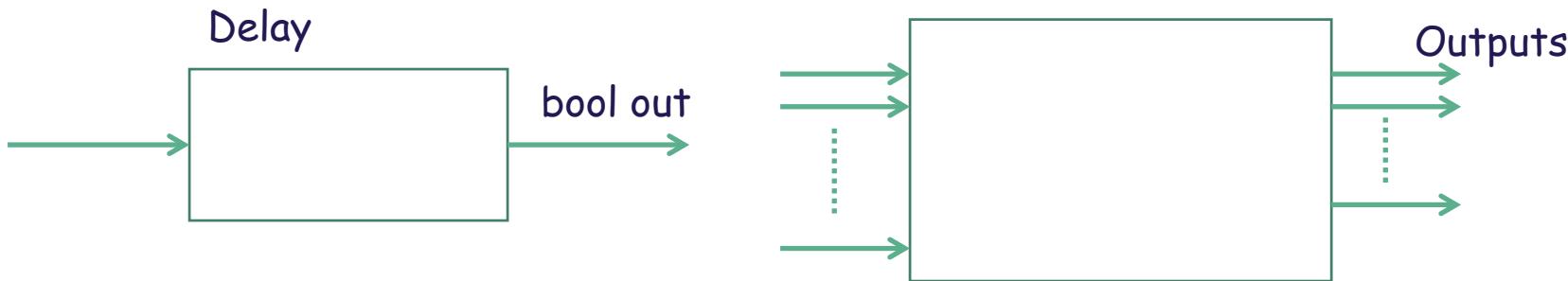


SRC Definition (1): Inputs



- Each component has a set I of input variables
 - Variables have types, e.g., bool, int, nat, real, {on, off} ...
- Input: Valuation of all the input variables
 - The set of inputs is denoted Q_I
- For Delay
 - I contains a single variable i of type bool
 - The set of inputs is $\{0, 1\}$
- Example: I contains two variables: int x , bool y
 - Each input is a pair: (integer value for x and 0/1 value for y)

SRC Definition (2): Outputs



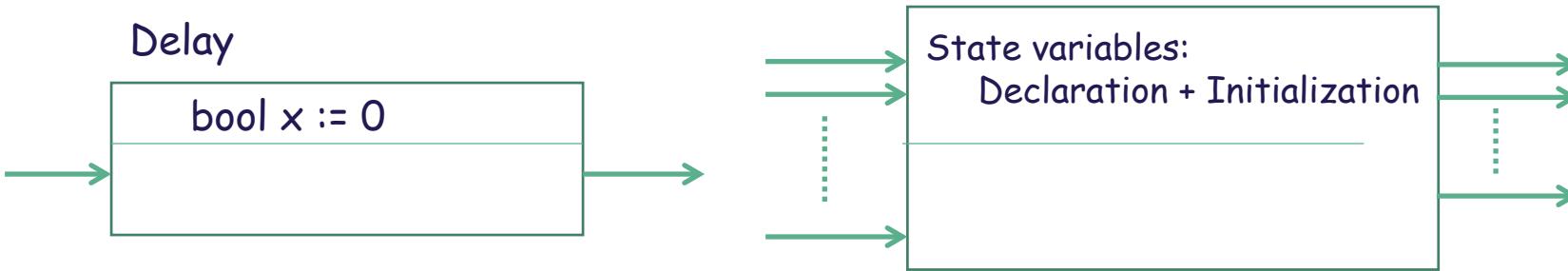
- Each component has a set O of typed output variables
- Output: Valuation of all the output variables
 - The set of outputs is denoted Q_O
- For Delay
 - O contains a single variable out of type bool
 - The set of outputs is $\{0, 1\}$

SRC Definition (3): States



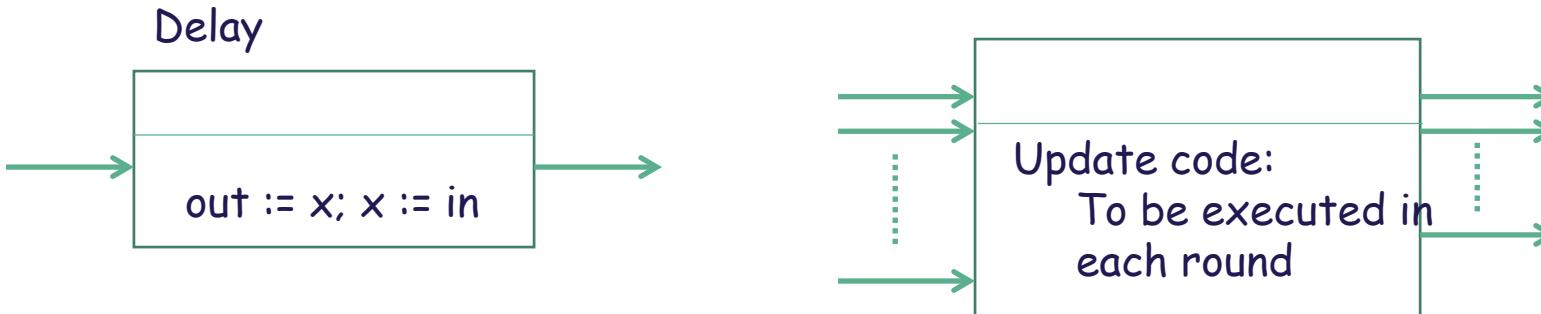
- ❑ Each component has a set S of typed state variables
- ❑ State: Valuation of all the state variables
 - The set of states is denoted Q_S
- ❑ For Delay
 - S contains a single variable x of type bool
 - The set of states is $\{0, 1\}$
- ❑ State is internal and maintained across rounds

SRC Definition (4): Initialization



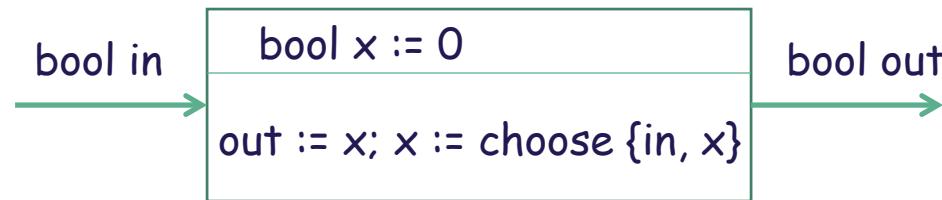
- Initialization of state variables specified by Init
 - Sequence of assignments to state variables
- Semantics of initialization:
 - The set [Init] of initial states, which is a subset of Q_s
- For Delay
 - Init is given by the code fragment $x:=0$
 - The set [Init] of initial states is $\{0\}$
- Component can have multiple initial states
 - Example: $\text{bool } x := \text{choose }\{0, 1\}$

SRC Definition (5): Reactions



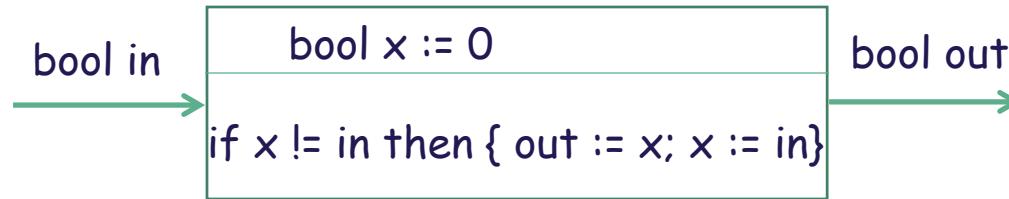
- Execution in each round given by code fragment React
 - Sequence of assignments and conditionals that assign output variables and update state variables
- Semantics of update:
 - The set [React] of reactions, where each reaction is of the form
(old) state - input / output → (new) state
 - [React] is a subset of $Q_S \times Q_I \times Q_O \times Q_S$
- For Delay:
 - React is given by the code fragment $\text{out}:=\text{x}; \text{x}:=\text{in}$
 - There are 4 reactions: $0 - 0/0 \rightarrow 0; 0 - 1/0 \rightarrow 1; 1 - 0/1 \rightarrow 0; 1 - 1/1 \rightarrow 1$

Multiple Reactions



- During update, either x is updated to input in , or left unchanged
 - Motivation: models that an input may be “lost”
- Nondeterministic reactions
 - Given (old) state and input, output/new state need not be unique
 - The set [React] of reactions now contains
$$0 -0/0-> 0$$
$$0 -1/0-> 1; 0 -1/0-> 0$$
$$1 -0/1-> 0; 1 -0/1-> 1$$
$$1 -1/1-> 1$$

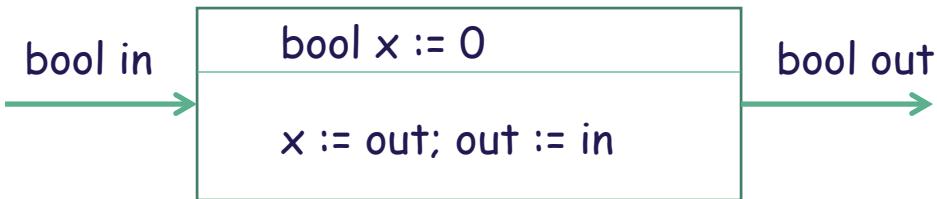
Multiple Reactions



- A component may not accept all inputs in all states
 - Motivation: “blocking” communication

- Possible set of reactions in certain state/input combinations may be empty
 - The set [React] of reactions now contains
$$0 -1/0\rightarrow 1$$
$$1 -0/1\rightarrow 1$$

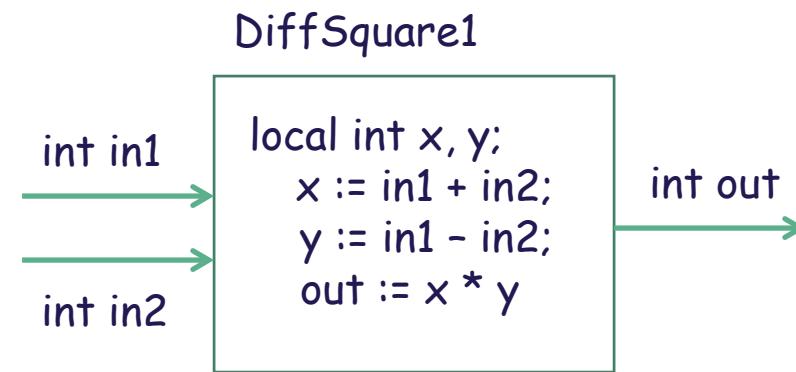
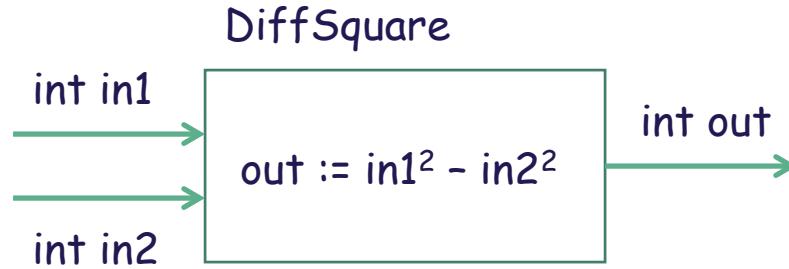
Syntax Errors



- If update code cannot be executed, then no reaction possible
 - In above: set [React] of reactions is the empty set

- Update code expected to satisfy a number of requirements
 - Types of variables and expressions should match
 - Output variables must first be written before being read
 - Output variable must be explicitly assigned a value

Semantic Equivalence



- Both have identical sets of reactions
- Syntactically different but semantically equivalent
- Compiler can optimize code as long as semantics is preserved!

Synchronous Reactive Component Definition

- Set I of typed input variables: gives set Q_I of inputs
- Set O of typed output variables: gives set Q_O of outputs
- Set S of typed state variables: gives set Q_S of states
- Initialization code Init : defines set $[\text{Init}]$ of initial states
- Reaction description React : defines set $[\text{React}]$ of reactions of the form $s -i/o\rightarrow t$, where s, t are states, i is an input, and o is an output

Synchronous languages in practice:

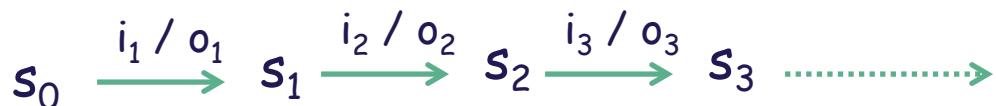
Richer syntactic features to describe React

Key to understanding: what happens in a single reaction?

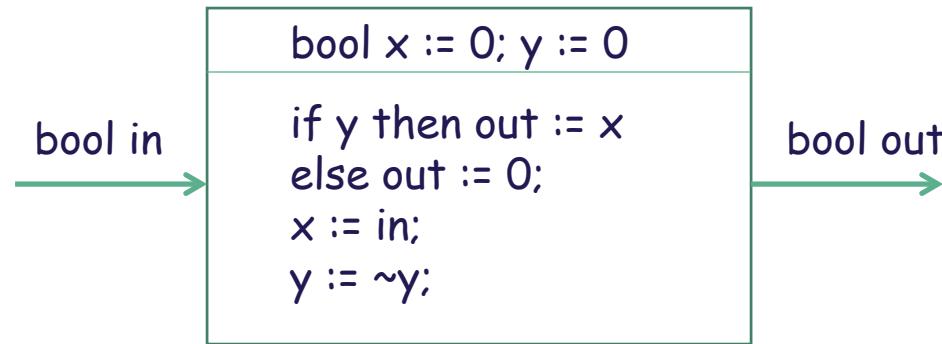
Formal semantics necessary for development of tools

Definition of Executions

- Given component $C = (I, O, S, \text{Init}, \text{React})$, what are its executions?
- Initialize state to some state s_0 in [Init]
- Repeatedly execute rounds. In each round $n=1,2,3,\dots$
 - Choose an input value i_n in Q_I
 - Execute React to produce output o_n and change state to s_n
that is, $s_{n-1} - i_n / o_n \rightarrow s_n$ must be in [React]
- Sample execution:

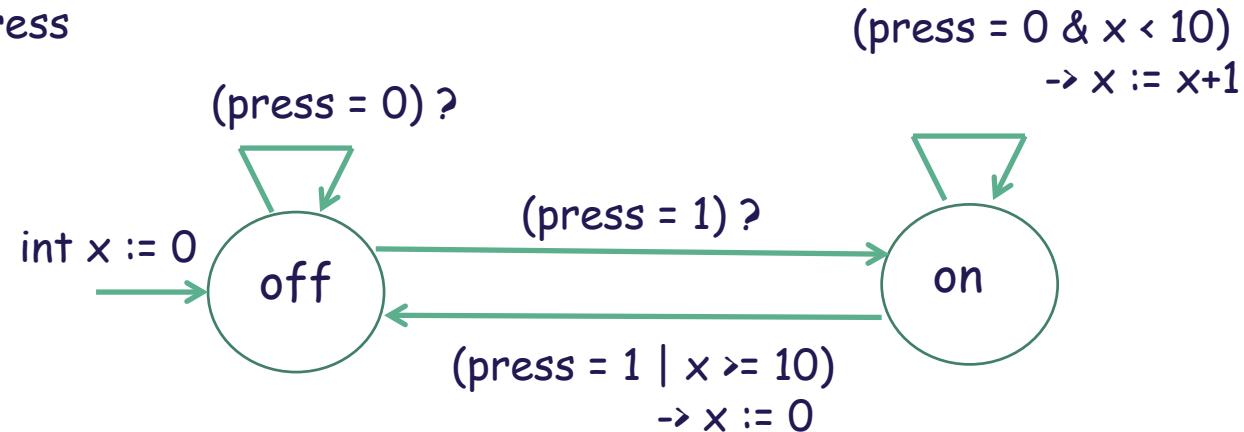


What Does This Component do?



Extended State Machines

Input: bool press



mode is a state variable ranging over {on, off}

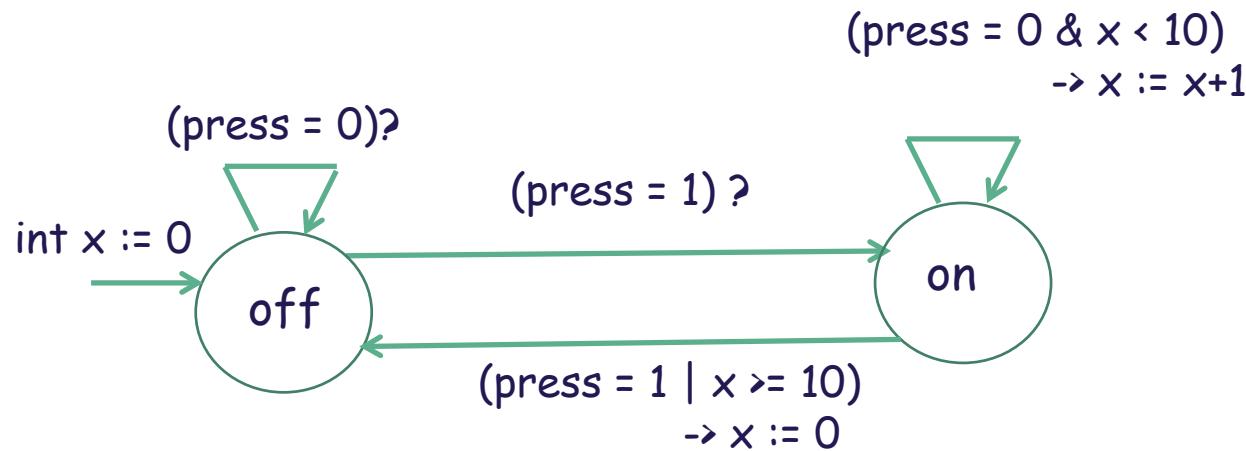
Reaction corresponds to executing a **mode-switch**

Example mode-switch: from on to off with

guard **(press = 1 | x >= 10)** and update code **x := 0**

Executing ESMs: Switch

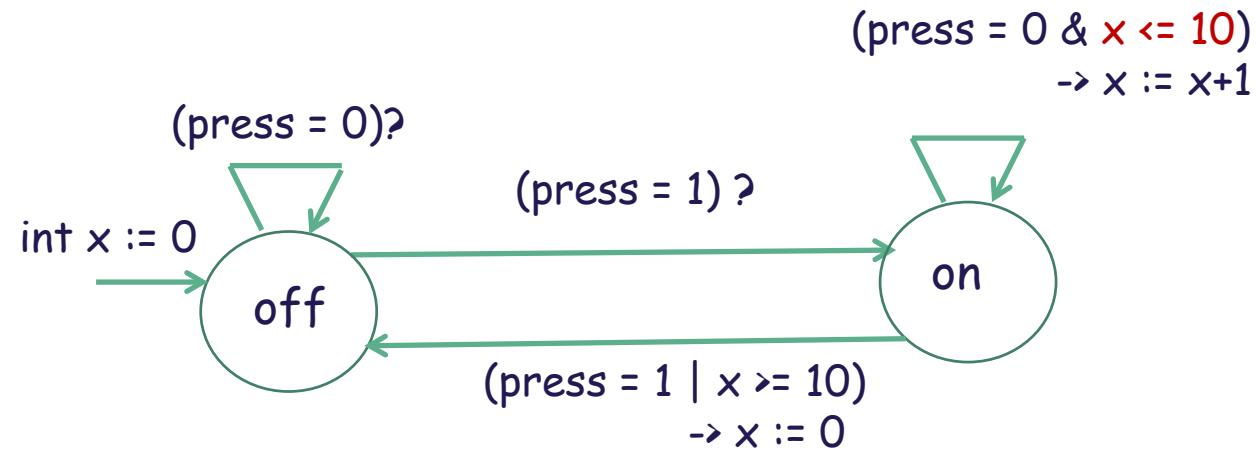
Input: bool press



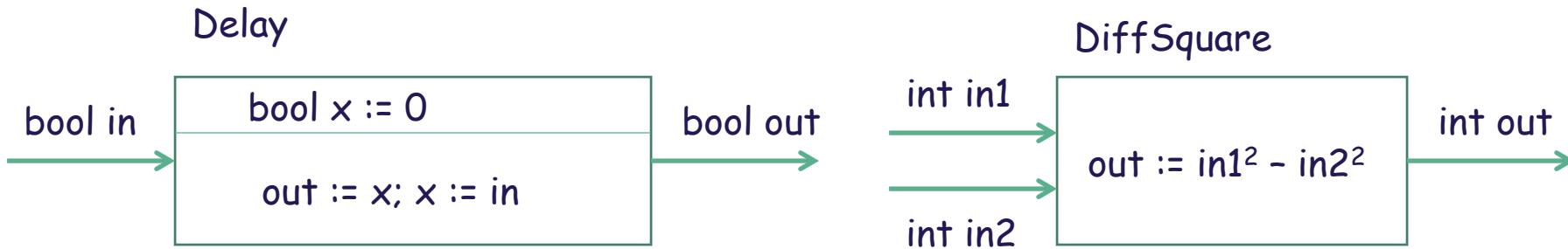
- State of the component Switch assigns values to mode and x
- Initial state: (off, 0)
- Sample Execution:
(off,0) -0-> (off,0) -1-> (on,0) -0-> (on,1) -0-> (on,2) ... -0->(on,10) -0-> (off,0)

Modified Switch: Which Executions are Possible?

Input: bool press

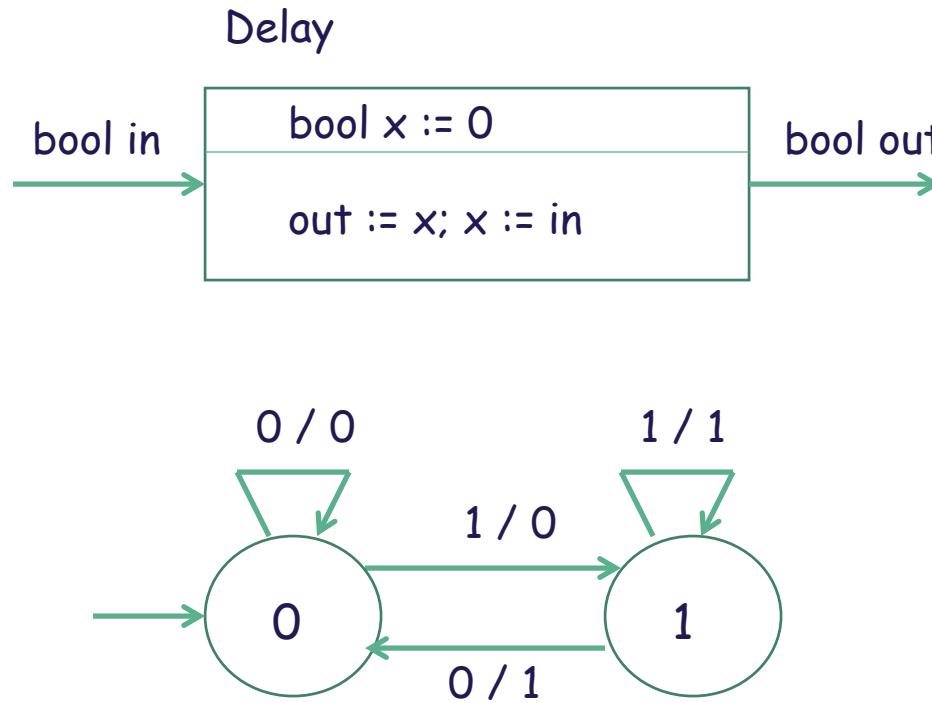


Finite-State Components



- A component is finite-state if all its variables range over finite types
 - Finite types: bool, enumerated types (e.g. {on, off}), int[-5, 5]
 - Delay is finite-state, but DiffSquare is not

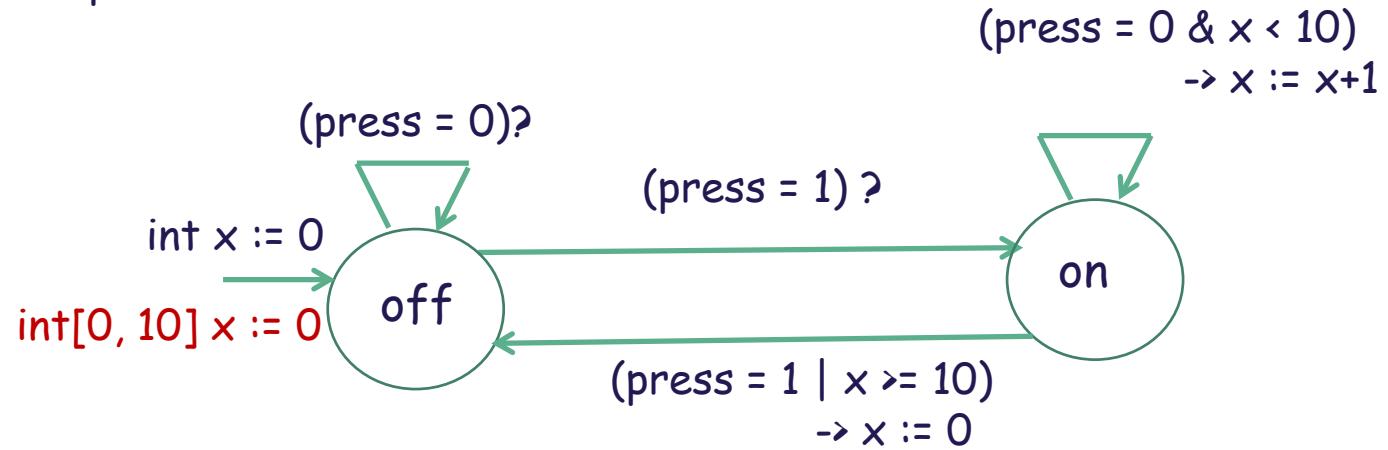
Mealy Machines (for Finite-State Components)



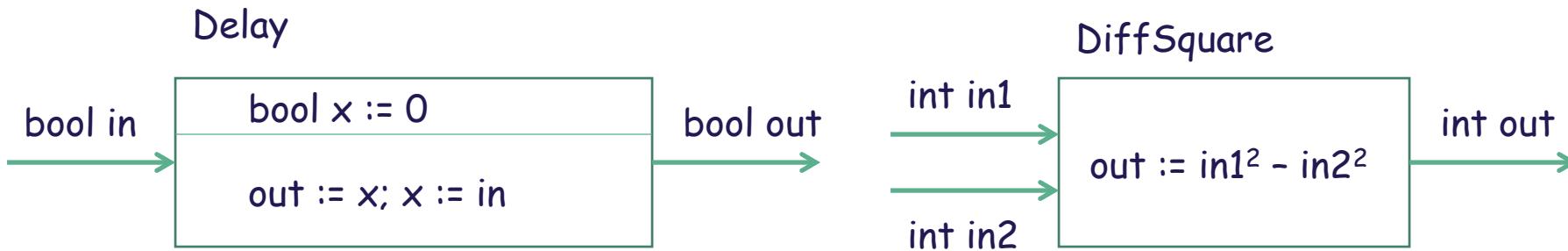
- Finite-state components are amenable to exact, algorithmic analysis

Switch: Is it Finite-State?

Input: bool press



Combinational Components



- A component is combinational if it has no state variables
 - DiffSquare is combinational, but Delay is not
 - Hardware gates are combinational components

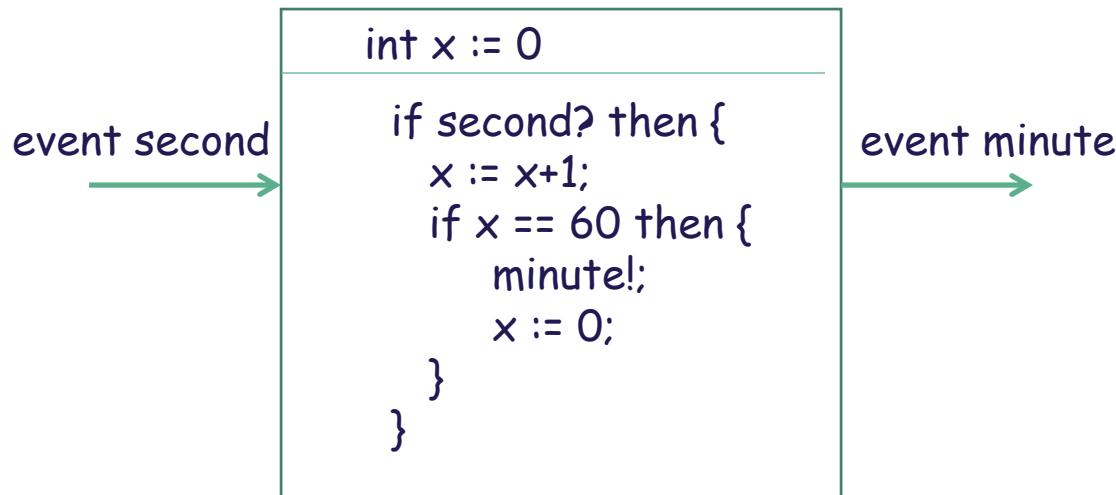
Events

- Input/output variable can be of type **event**
- An event can be absent, or present, in which case it has a value
 - **event x** means x ranges over {absent, present}
 - **event(bool) x** means x ranges over {absent, 0, 1}
 - **event(nat) x** means x ranges over {absent, 0, 1, 2, ...}
- Syntax: **$x?$** means the test for presence
- Syntax: **$x!v$** means the assignment $x := v$ (and **$x!$** means $x := \text{present}$)
- Event-based communication:
 - If no value is assigned to an output event, then it is absent (by default)
 - Event-triggered components execute only in rounds where input events are present (actual definition slightly more general, see textbook)
 - Motivation: notion of "clock" can be different for different components

Second-to-Minute

Desired behavior (spec):

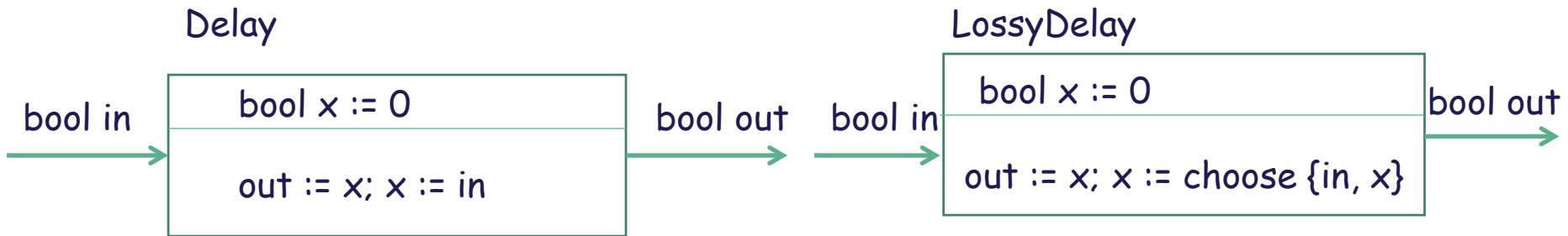
Issue the output event every 60th time the input event is present



□ Event-Triggered Components

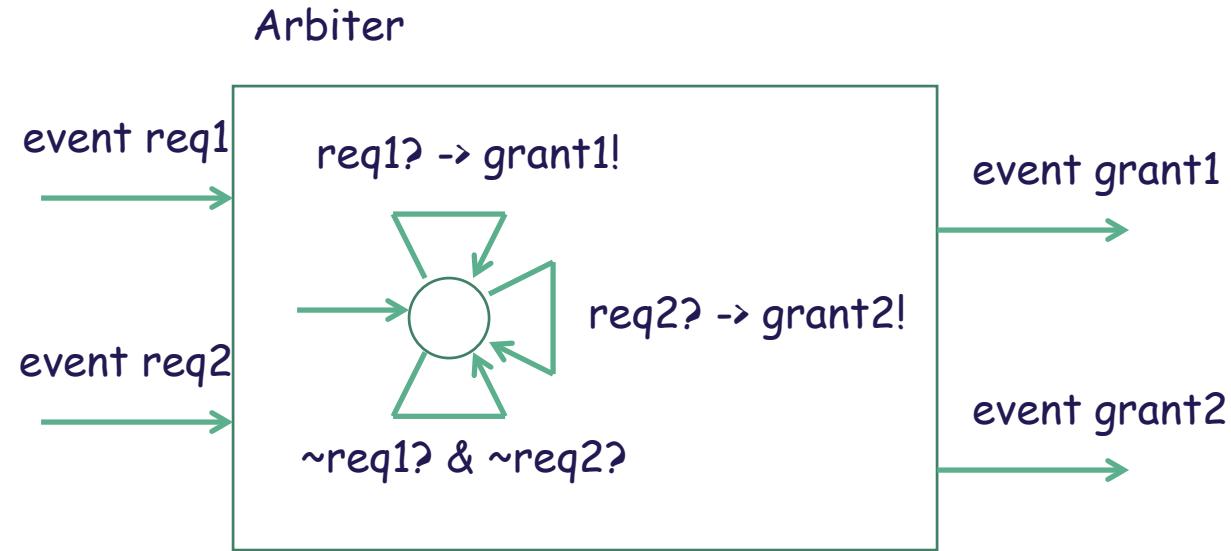
- No need to execute in a round where triggering input events absent
- See textbook for formal definition

Deterministic Components

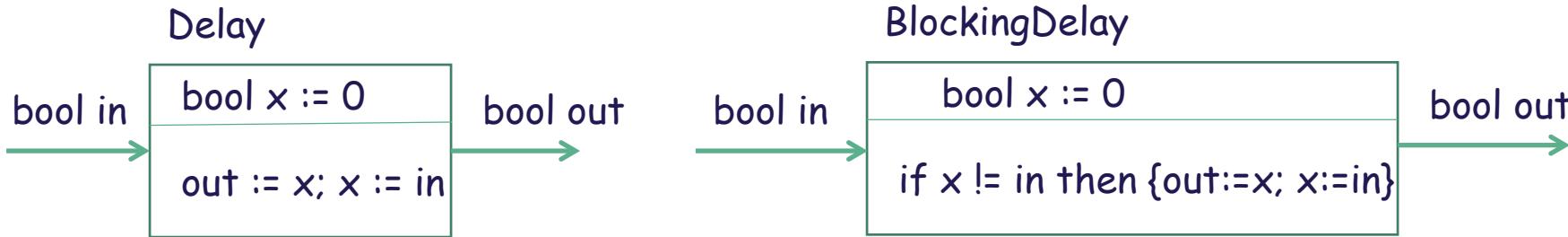


- A component is deterministic if (1) it has a single initial state, and (2) for every state s and input i , there is a unique state t and output o such that $s - i/o \rightarrow t$ is a reaction
 - Delay is deterministic, but LossyDelay is not
- Deterministic: If same sequence of inputs supplied, same outputs observed (predictable, repeatable behavior)
- Nondeterminism is useful in modeling uncertainty /unknown
- Nondeterminism is not same as probabilistic (or random) choice

Nondeterminism

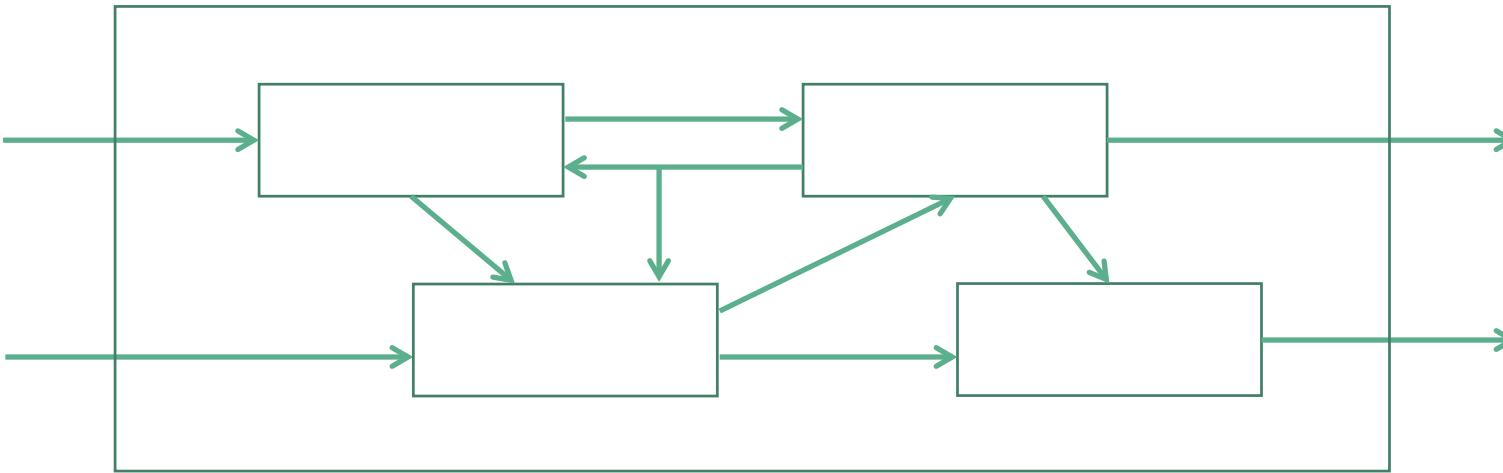


Input-Enabled Components



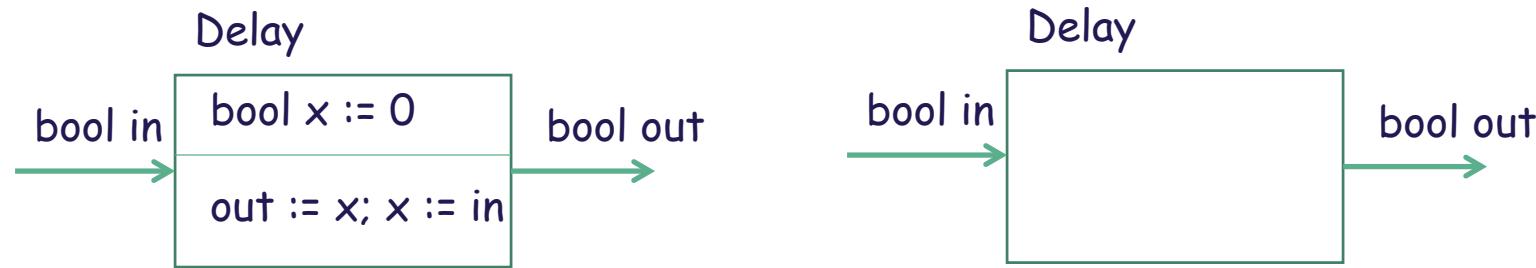
- A component is input-enabled if for every state s and input i , there exists a state t and an output o such that $s - i/o \rightarrow t$ is a reaction
 - Delay is input-enabled, but BlockingDelay is not
- Not input-enabled means component is making assumptions about the context in which it is going to be used
 - When rest of system is designed, must check that it indeed satisfies these assumptions

Block Diagrams



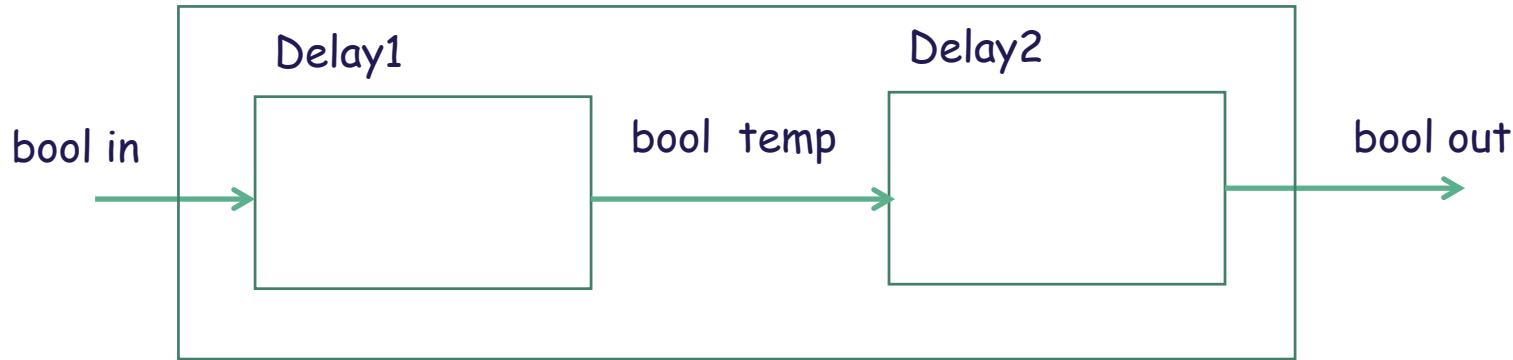
- Structured modeling
 - How do we build complex models from simpler ones
 - What are basic operations on components?

DoubleDelay



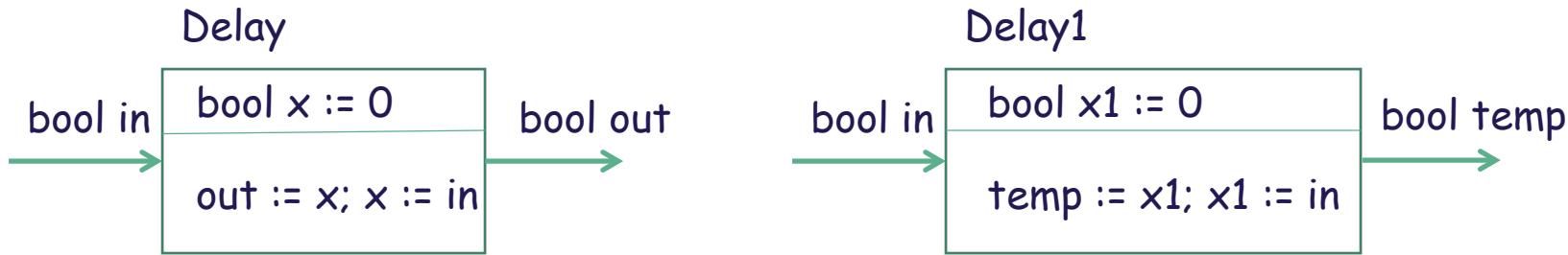
- Design a component with
 - Input: bool in
 - Output: bool out
 - Output in round n should equal input in round n-2

DoubleDelay



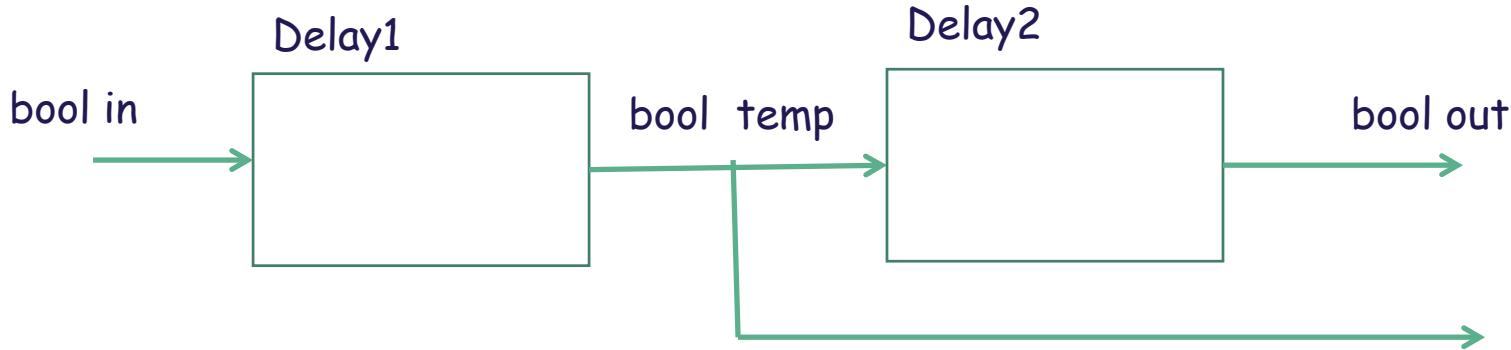
- Instantiation: Create two instances of Delay
 - Output of Delay1 = Input of Delay2 = Variable temp
- Parallel composition: Concurrent execution of Delay1 and Delay2
- Hide variable temp: Encapsulation

Instantiation / Renaming



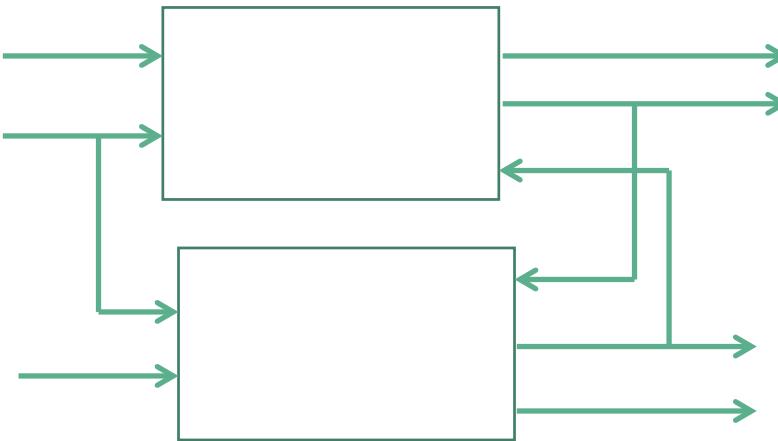
- $\text{Delay1} = \text{Delay}[\text{out} \rightarrow \text{temp}]$
 - Explicit renaming of input/output variables
 - Implicit renaming of state variables
 - Components (I,O,S,Init,React) of Delay1 derived from Delay
- $\text{Delay2} = \text{Delay}[\text{in} \rightarrow \text{temp}]$
- Renaming always chooses a fresh variable name

Parallel Composition



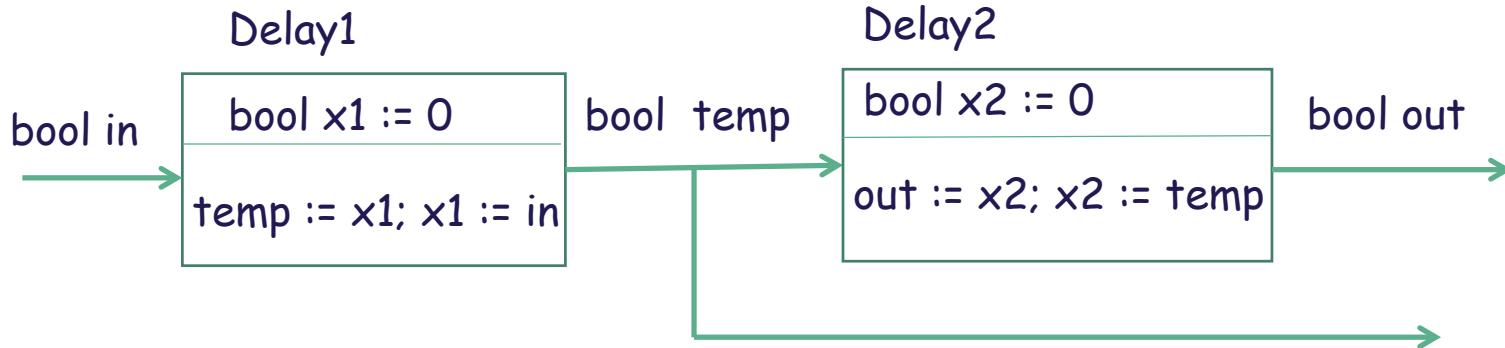
- DDelay = Delay1 || Delay2
 - Execute both concurrently
- When can two components be composed?
- How to define parallel composition precisely?
 - Input/output/state variables, initialization, and reaction description of composite defined in terms of components

Compatibility of Components C1 and C2



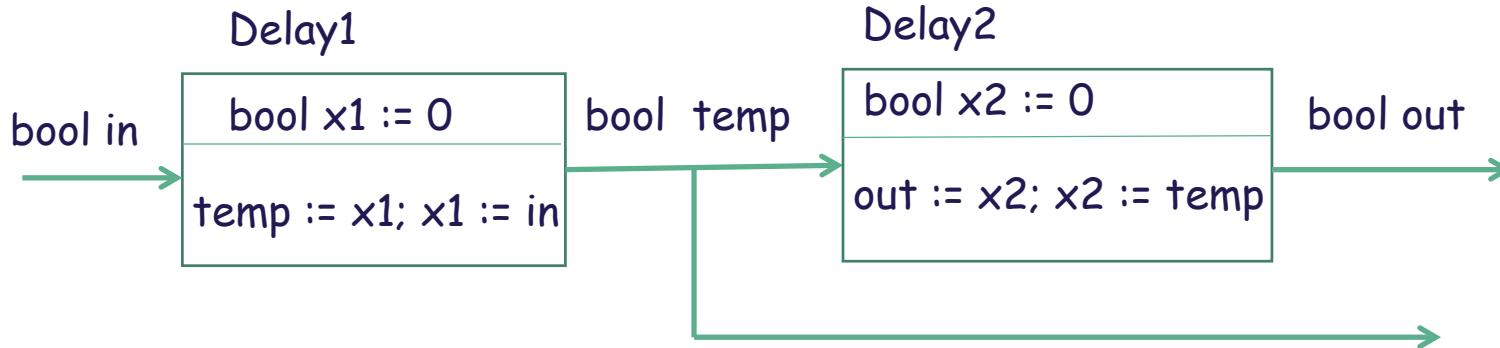
- Can have common input variables
- Cannot have common output variables
 - A unique component responsible for values of any given variable
- Cannot have common state variables
 - State variables can be implicitly renamed to avoid conflicts
- Input variable of one can be output of another, and vice versa

Outputs of Product



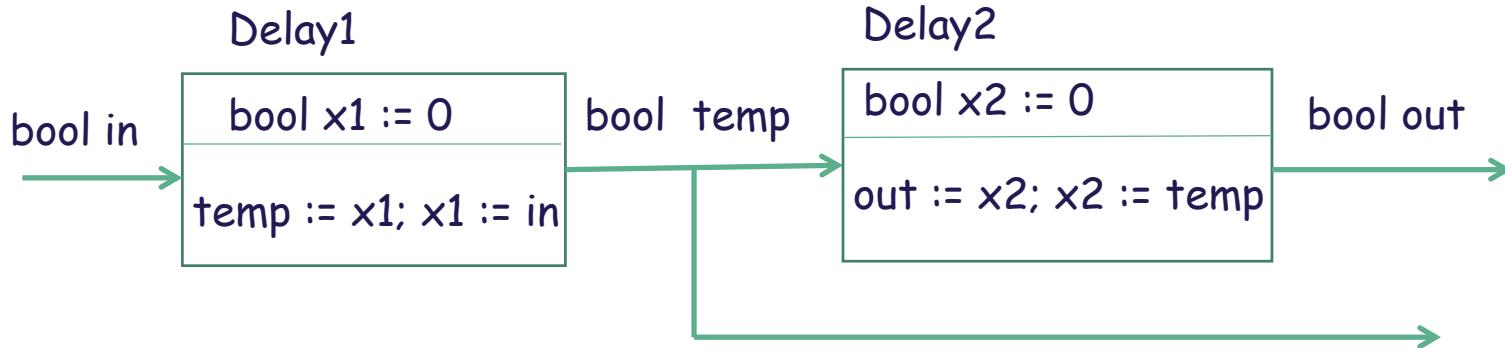
- Output variables of $Delay1 \parallel Delay2$ is $\{temp, out\}$
 - Note: By default, every output is available to outside world
- If C_1 has output vars O_1 and C_2 has output vars O_2 then the product $C_1 \parallel C_2$ has output vars $O_1 \cup O_2$

Inputs of Product



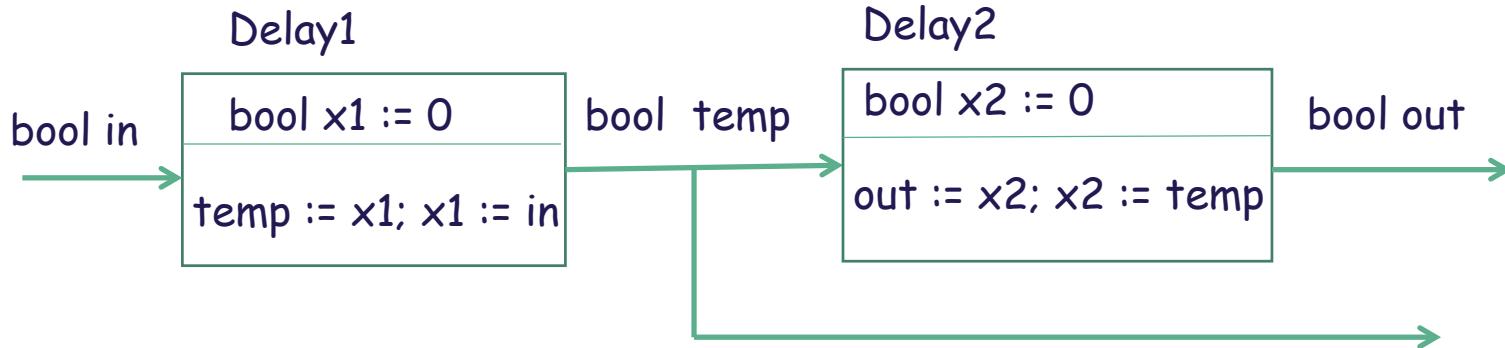
- Input variables of Delay1 || Delay2 is {in}
 - Even though temp is input of Delay2, it is not an input of product
- If C_1 has input vars I_1 and C_2 has input vars I_2 then the product $C_1 || C_2$ has input vars $(I_1 \cup I_2) \setminus (O_1 \cup O_2)$
 - A variable is an input of the product if it is an input of one of the components, and not an output of the other

States of Product



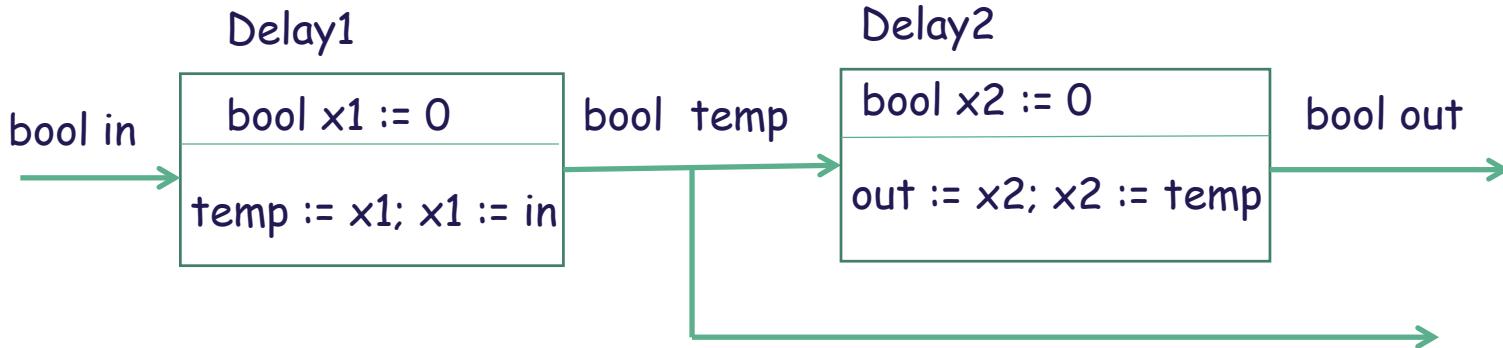
- State variables of $\text{Delay1} \parallel \text{Delay2}$: $\{x_1, x_2\}$
- If C_1 has state vars S_1 and C_2 has state vars S_2 then the product has state vars $(S_1 \cup S_2)$
 - A state of the product is a pair (s_1, s_2) , where s_1 is a state of C_1 and s_2 is a state of C_2
 - If C_1 has n_1 states and C_2 has n_2 states then the product has $(n_1 \times n_2)$ states

Initial States of Product



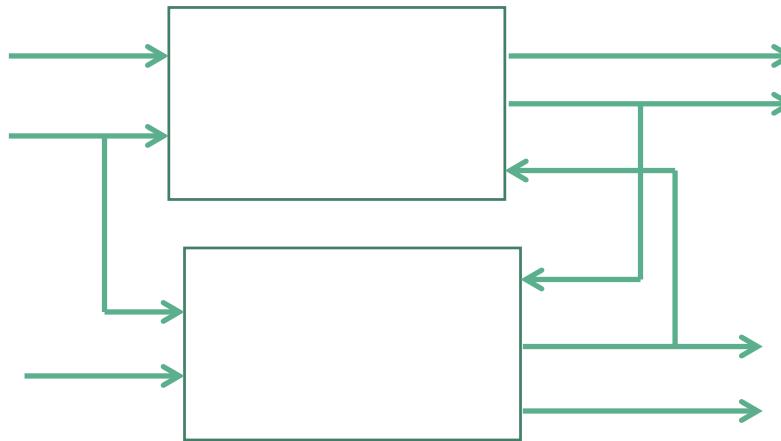
- The initialization code Init for $\text{Delay1} \parallel \text{Delay2}$ is " $x1 := 0; x2 := 0$ "
 - Initial state is $(0,0)$
- If $C1$ has initialization Init1 and $C2$ has initialization Init2 then the product $C1 \parallel C2$ has initialization $\text{Init1}; \text{Init2}$
 - Order does not matter
 - $[\text{Init}]$ is the product of sets $[\text{Init1}] \times [\text{Init2}]$

Reactions of Product



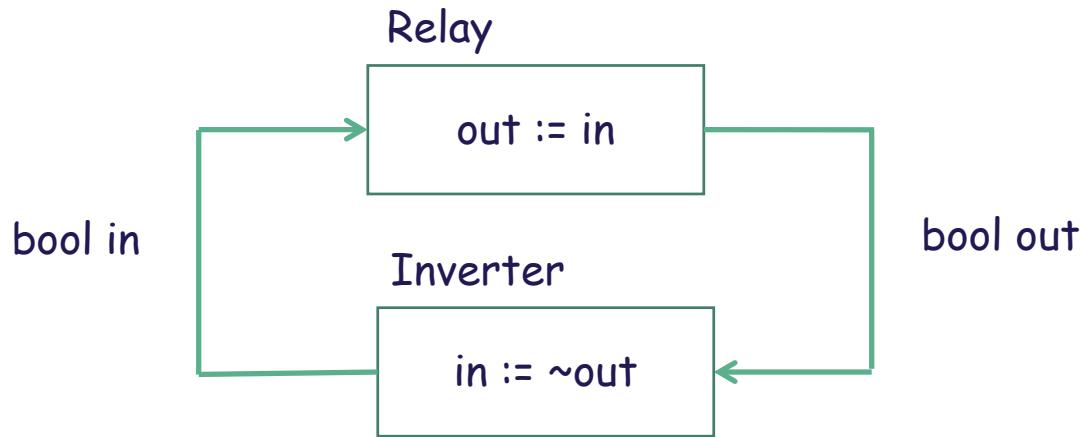
- Execution of $\text{Delay1} \parallel \text{Delay2}$ within a round
 - Environment provides input value for variable `in`
 - Execute code "`temp := x1; x1 := in`" of `Delay1`
 - Execute code "`out := x2; x2 := temp`" of `Delay2`

Feedback Composition



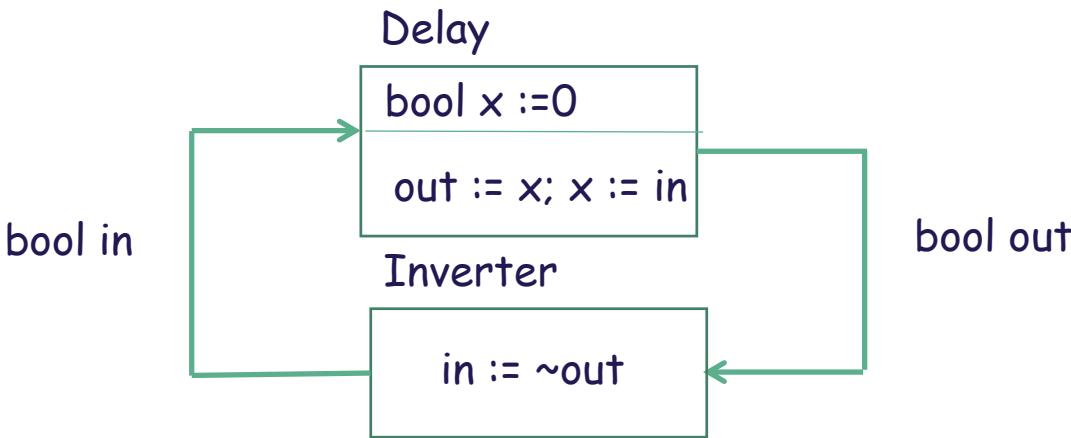
- When some output of C1 is an input of C2, and some output of C2 is an input of C1, how do we order the executions of reaction descriptions React1 and React2?
- Should such composition be allowed at all?

Feedback Composition



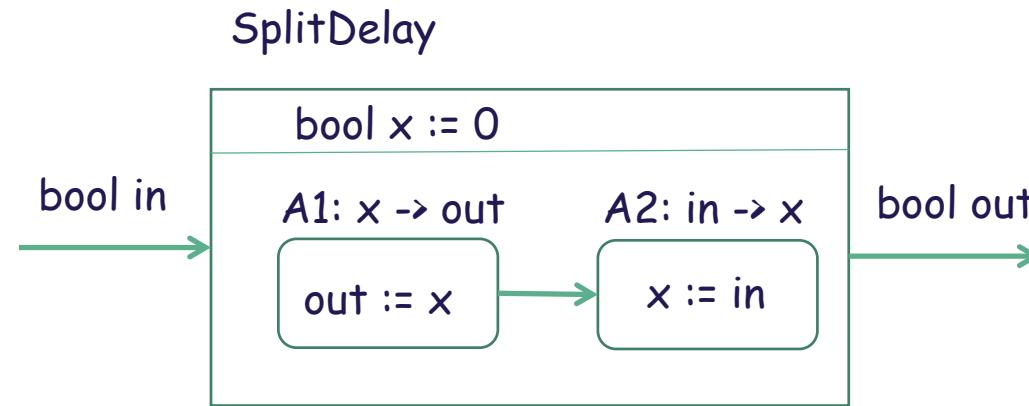
- For Relay, its output **out** “awaits” its input **in**
- For Inverter, its output **in** “awaits” its input **out**
- In product, we cannot order the execution
- With such cyclic dependencies, composition is disallowed
 - Intuition: Combinational cycles should be avoided

Feedback Composition



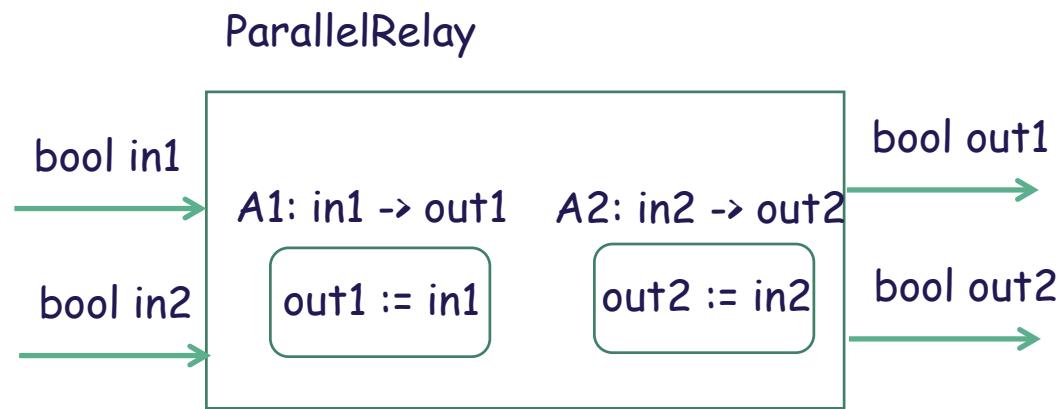
- For Delay, possible to produce output without waiting for its input by executing the assignment "out := x"
- Reaction code for product can be "out := x; in := ~out; x := in"
- Goal: Refine specification of reaction description so that "await" dependencies among output-input variables are easy to detect
 - Ordering of code-blocks during composition should be easy

Splitting Reaction Code Into Tasks



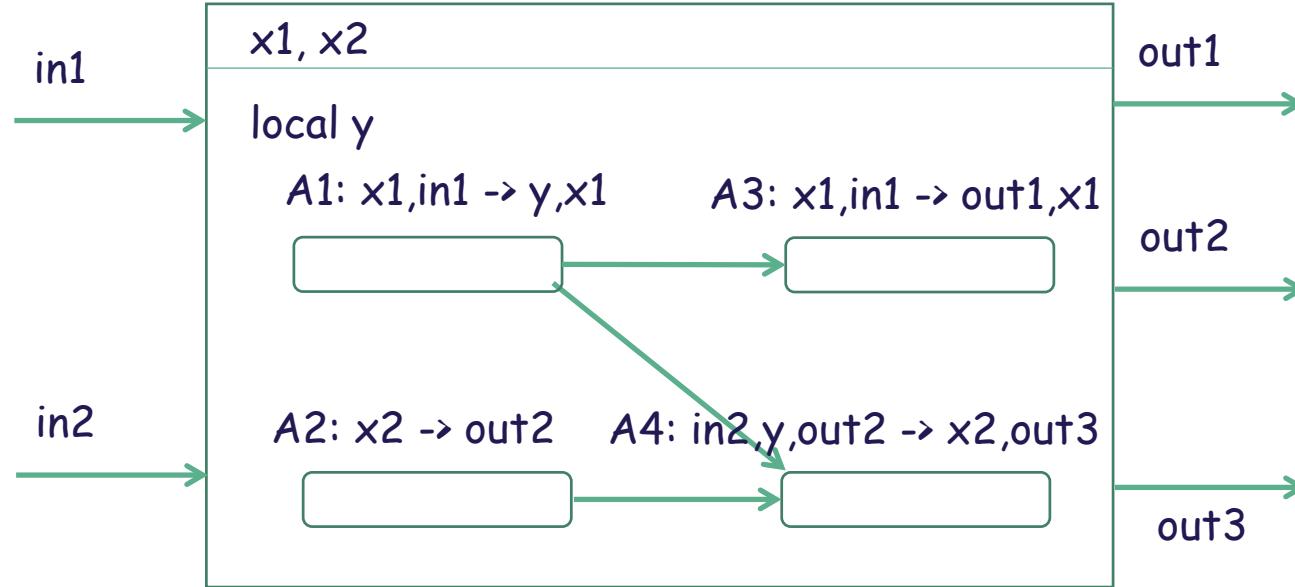
- `A1` and `A2` are tasks (atomic blocks of code)
 - Each task specifies variables it reads and writes
 - `A1` reads `x` and writes `out`
- Task Graph: Vertices are tasks and edges denote precedence
 - $A1 < A2$ means that `A1` should be executed before `A2`
 - Graph should be acyclic

Example Task Graph



- Tasks A1 and A2 are unordered
 - Possible “schedules” (linear ordering of tasks): A1, A2 and A2, A1
 - All consistent schedules give the same result
- I/O await dependencies: out1 awaits in1, out2 awaits in2

Example Task Graph



- What are possible schedules consistent with precedence constraints?
- What are I/O await dependencies?

Task Graphs: Definition

- For a synchronous reactive component C with input vars I , output vars O , state vars S , and local vars L , a reaction description is given by a set of tasks and precedence edges \prec over these tasks
- Each task A is specified by:
 1. Read-set R
 - must be a subset of $I \cup S \cup O \cup L$
 2. Write-set W
 - must be a subset of $O \cup S \cup L$
 3. Update: code to write variables in W based on values of vars in R
 - [Update] is a subset of $Q_R \times Q_W$

Requirements on Task Graph (1)

The precedence relation $<$ must be acyclic

- Notation: $A' <^+ A$ means that there is a path from task A' to task A in the task graph using precedence edges ($<^+$ denotes the "transitive closure" of the relation $<$)
- Task schedule: Total ordering A_1, A_2, \dots, A_n of all the tasks consistent with the precedence edges
 - If $A' < A$, then A' must appear before A in the ordering
 - Multiple schedules possible
- If $A' <^+ A$ then A' must appear before A in every schedule
- Acyclicity means that there is at least one task schedule

Requirements on Task Graph (2)

Each output variable is in the write-set of exactly one task

- If output y is in write-set of task A , then as soon as A executes the output y is available to the rest of the system
- If task A writes output y , then y awaits an input variable x , denoted $y > x$, if
 - either the task A reads x
 - or another task A' reads x such that $A' \leftarrow A$
- y awaits x means that y cannot be produced before x is supplied

Requirements on Task Graph (3)

Output/local variables are written before being read:

- If an output or a local variable y is in the read-set of a task A , then y must be in the write-set of some task A' such that $A' \leftarrow^+ A$

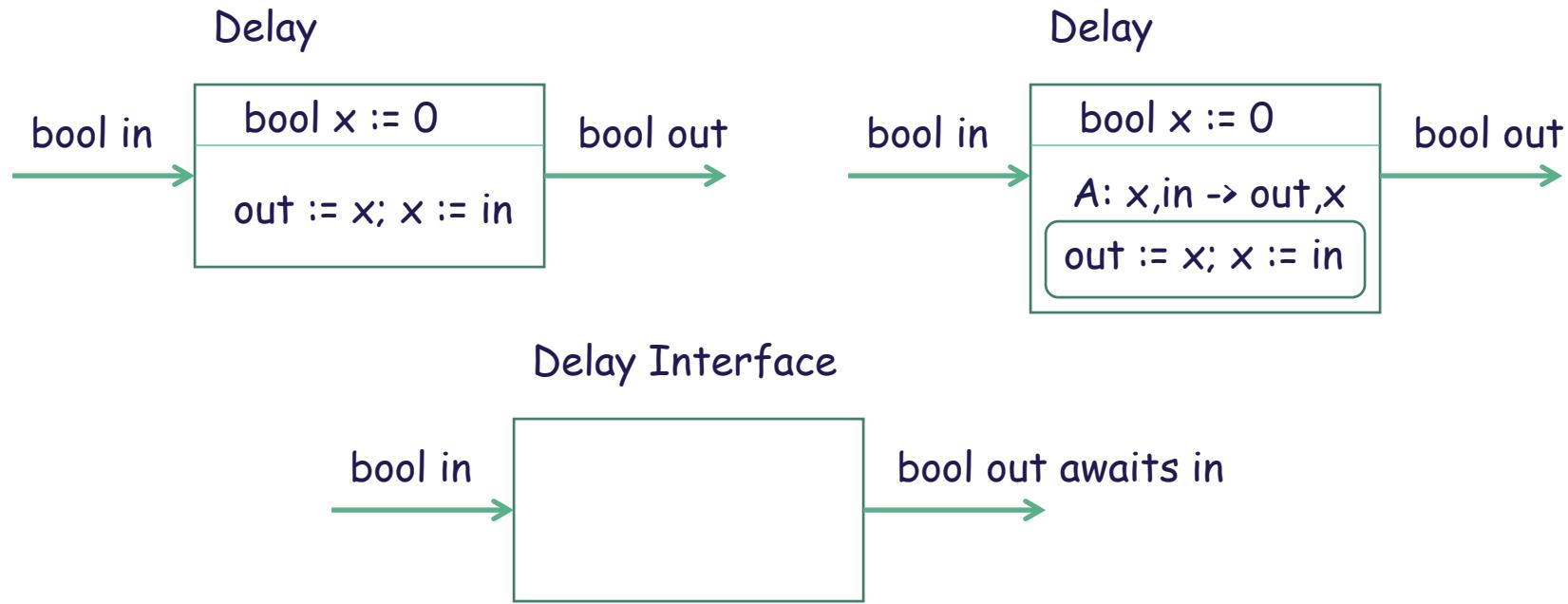
Requirements on Task Graph (4)

- Write-conflict between tasks A and A' :
 - There exists a variable that A writes and is either read or written by A'
- If A and A' have write-conflict, then the result depends on whether A executes before A' or vice versa.
 - Example: Update of A is $x := x+1$; Update of A' is $out := x$
- Requirement: Tasks with a write conflict must be ordered:
 - If tasks A and A' have write-conflict then either $A <^+ A'$ or $A' <^+ A$
- The set of reactions resulting from executing all the tasks do not depend on the task schedule

Properties of Tasks

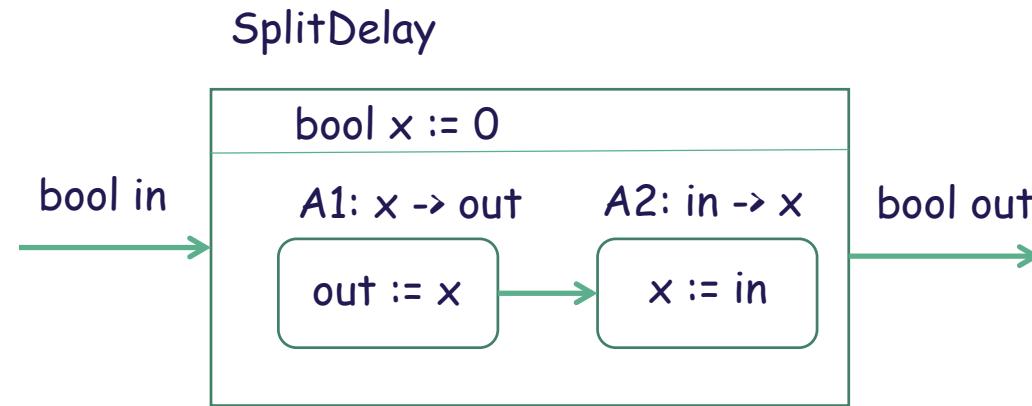
- Task $A = (R, W, \text{Update})$ is deterministic if for every value u in Q_R there is a unique value v in Q_W such that (u,v) is in $[\text{Update}]$
- If all tasks of a component are deterministic, what can we conclude about the component itself?
- Task $A = (R, W, \text{Update})$ is input-enabled if for every value u in Q_R there exists at least one value v in Q_W such that (u,v) is in $[\text{Update}]$
- If all tasks of a component are input-enabled, what can we conclude about the component itself?

Interfaces

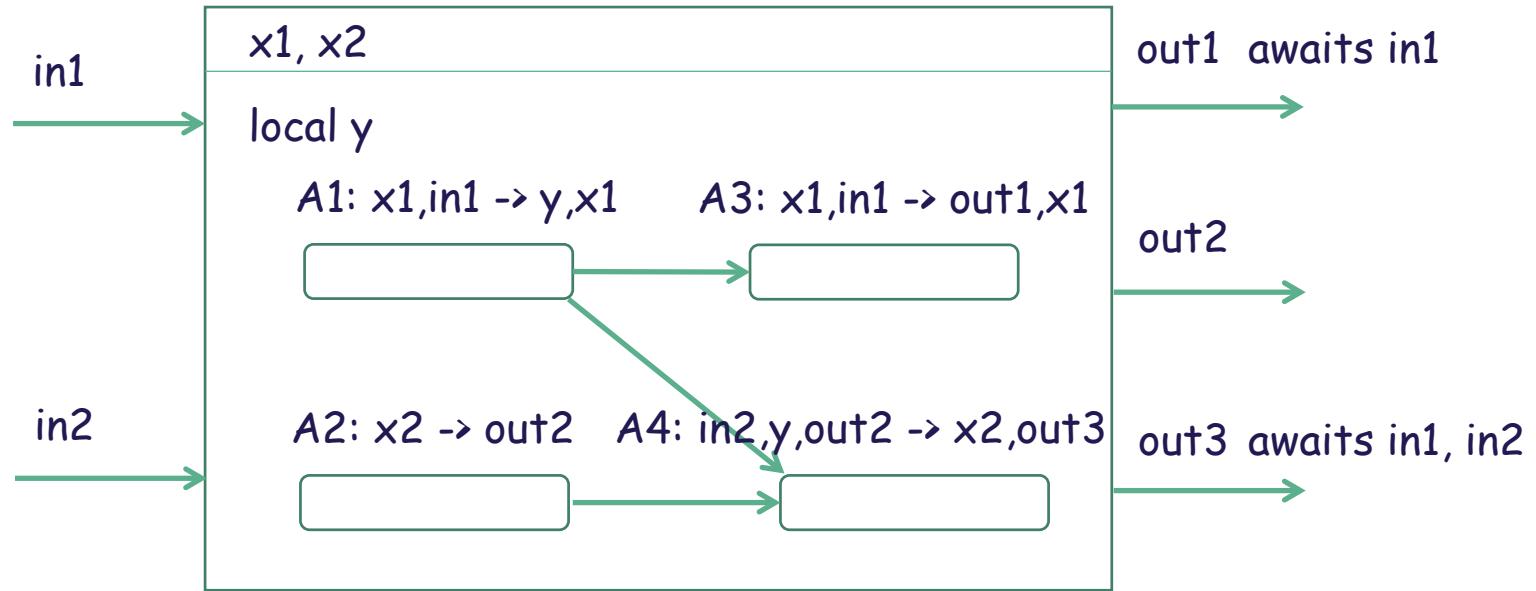


- Interface = Input variables, Output variables, Await dependencies

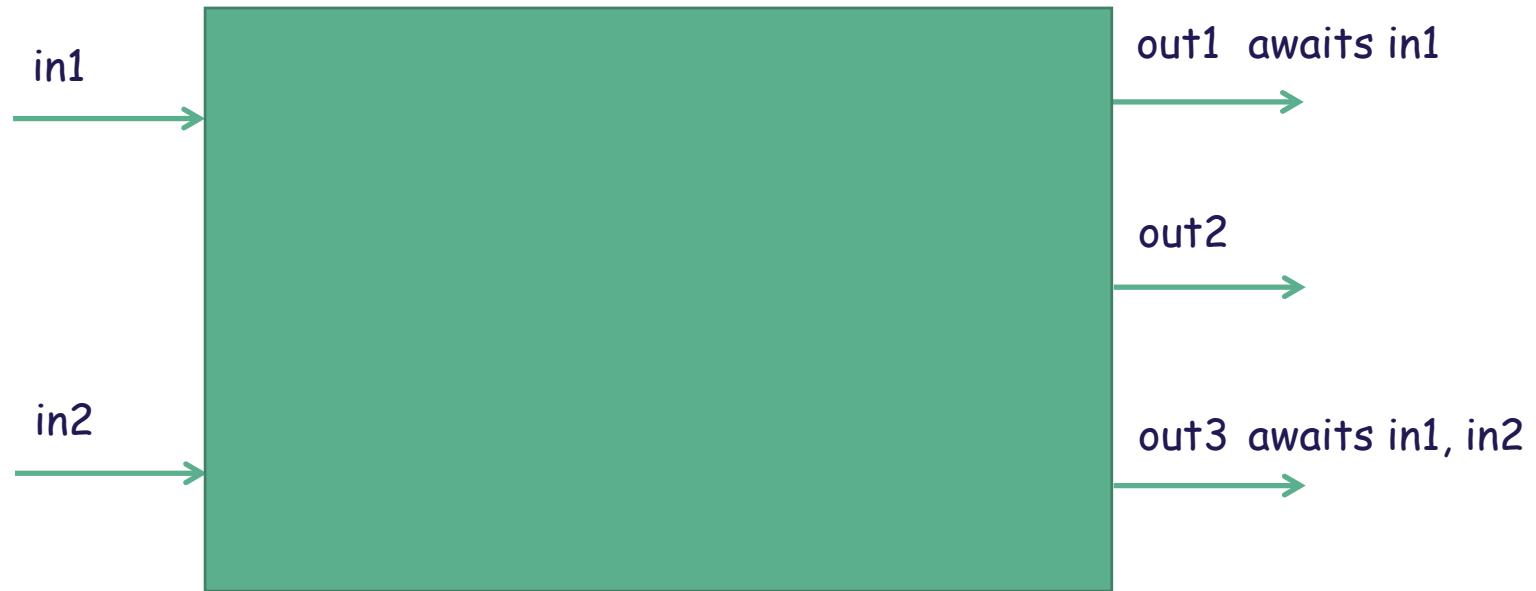
Interface: SplitDelay



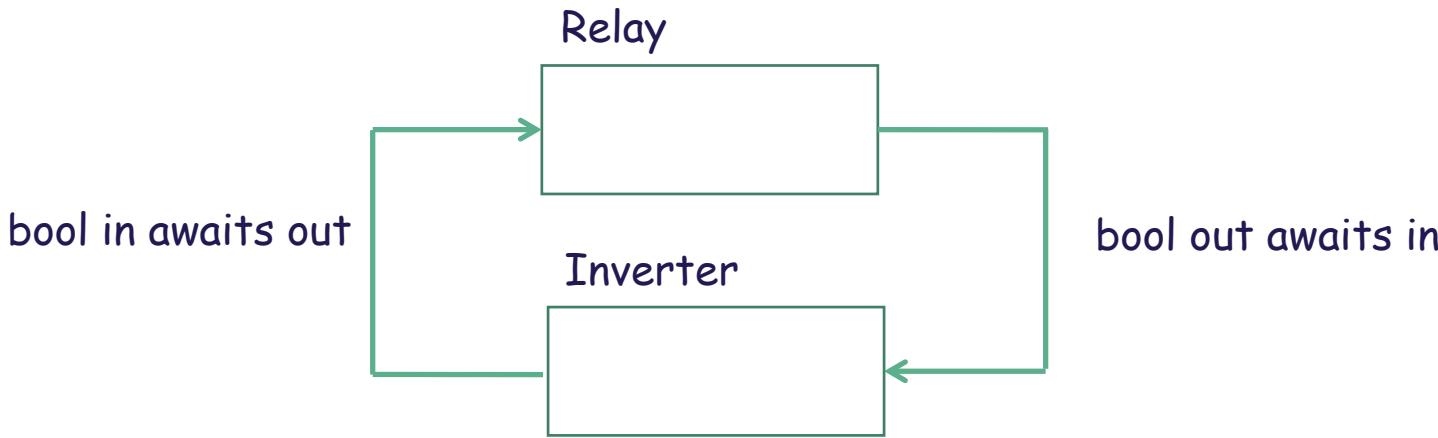
Example Interface



Example Interface

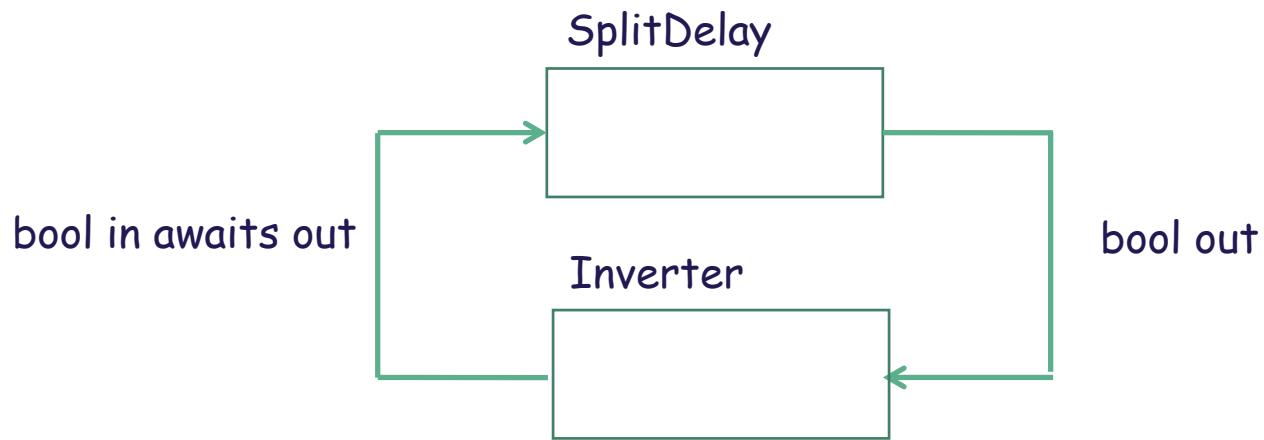


Back to Parallel Composition



- Relay and Inverter are not compatible since there is a cycle in their combined await dependencies

Composing SplitDelay and Inverter

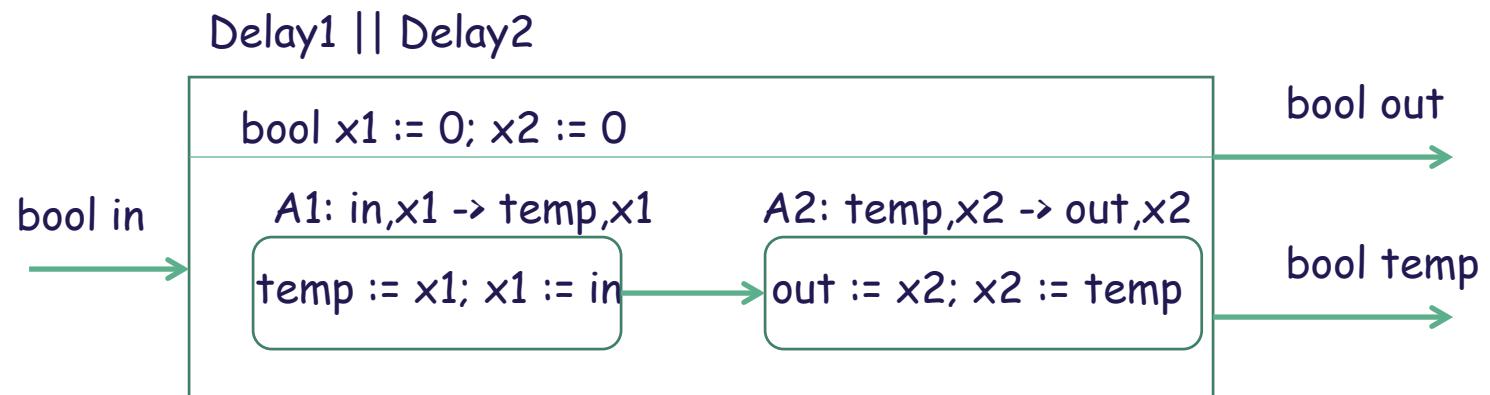
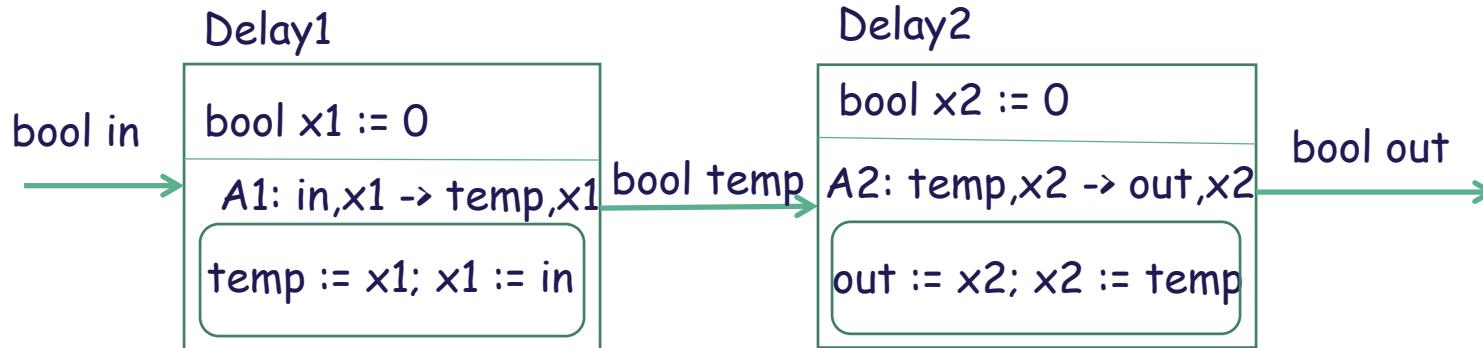


- SplitDelay and Inverter are compatible since there is no cycle in their combined await dependencies
- Note: Delay and Inverter are **not** compatible

Component Compatibility Definition

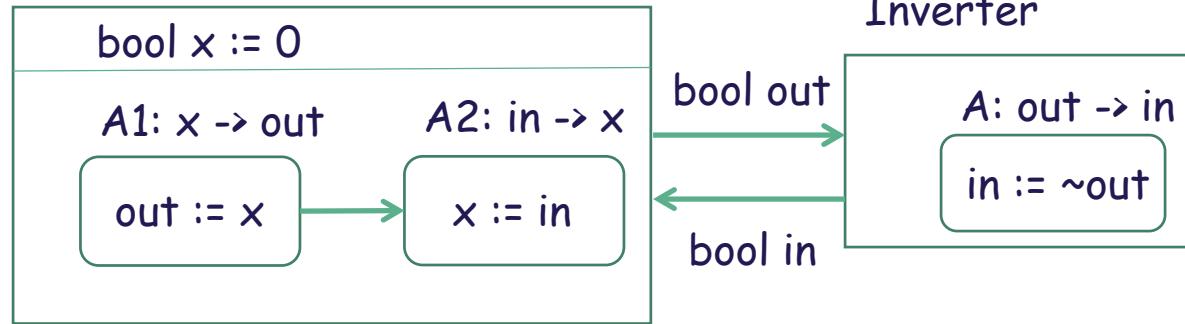
- Given:
 - Component C_1 with input vars I_1 , output vars O_1 , and awaits-dependency relation $>_1$
 - Component C_2 with input vars I_2 , output vars O_2 , and awaits-dependency relation $>_2$
- The components C_1 and C_2 are compatible if
 - No common outputs: sets O_1 and O_2 are disjoint
 - The relation $(>_1 \cup >_2)$ of combined await-dependencies is acyclic
- Parallel Composition is allowed only for compatible components

Defining the Product

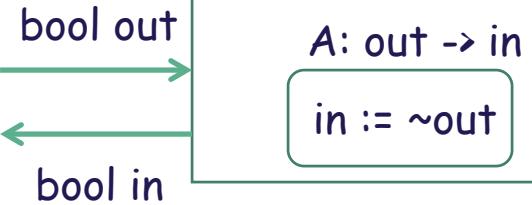


Composing SplitDelay and Inverter

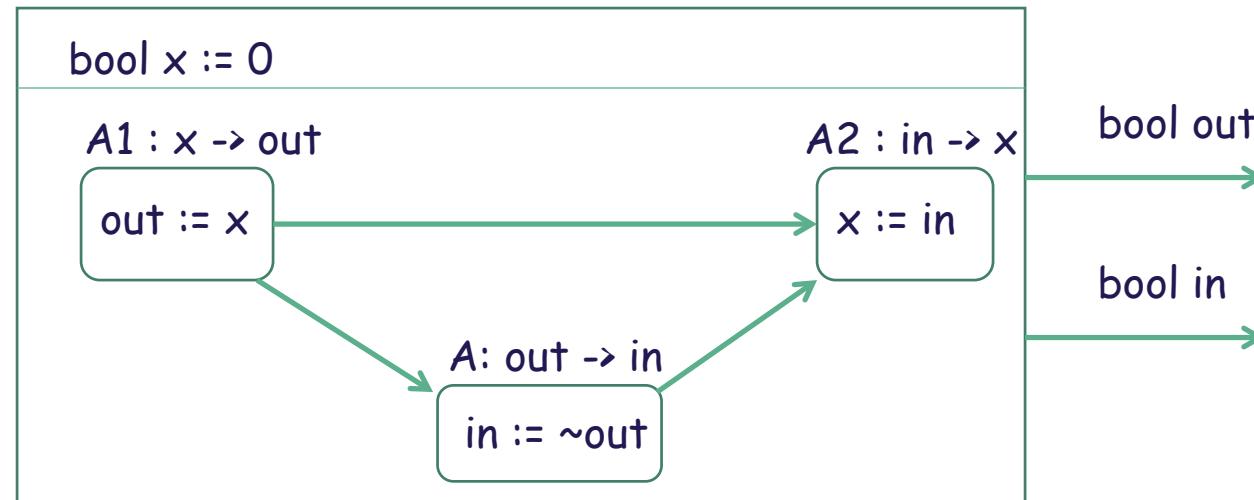
SplitDelay



Inverter



SplitDelay || Inverter



Parallel Composition Definition

- Given compatible components $C1 = (I1, O1, S1, \text{Init}1, \text{React}1)$ and $C2 = (I2, O2, S2, \text{Init}2, \text{React}2)$, what's the definition of product $C = C1 \parallel C2$?
- We already defined I, O, S, and Init for C
- Suppose React1 specified using local variables L1, set of tasks Π_1 , and precedence $<_1$, and React2 given using local vars L2, set of tasks Π_2 , and precedence $<_2$
- Reaction description for product C has
 - Local variables $L1 \cup L2$
 - Set of tasks $\Pi_1 \cup \Pi_2$
 - Precedence edges: Edges in $<_1$ + Edges in $<_2$ + Edge between tasks A1 and A2 of different components if A2 reads a variable written by A1

Parallel Composition Definition

- Why is the parallel composition operation well-defined?
 - Can the new edges make task graph of the product cyclic?
- Recall: Await-dependencies among I/O variables of compatible components must be acyclic
- Proposition 2.1: Awaits compatibility implies acyclicity of product task graph
- Bottomline: Interfaces capture enough information to define parallel composition in a consistent manner
- Aside: possible to define more flexible (but complex) notions of awaits dependencies

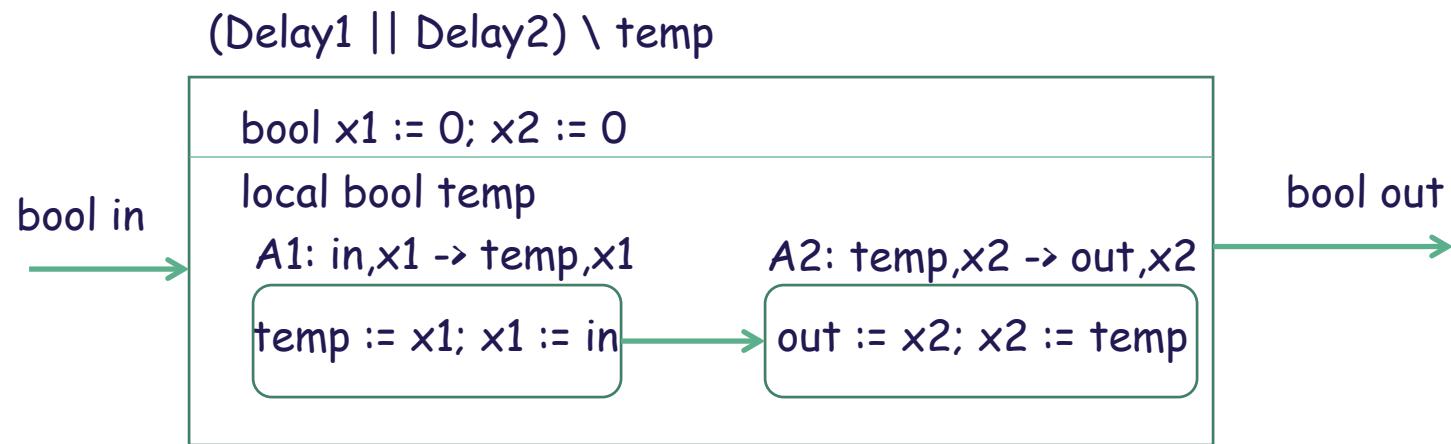
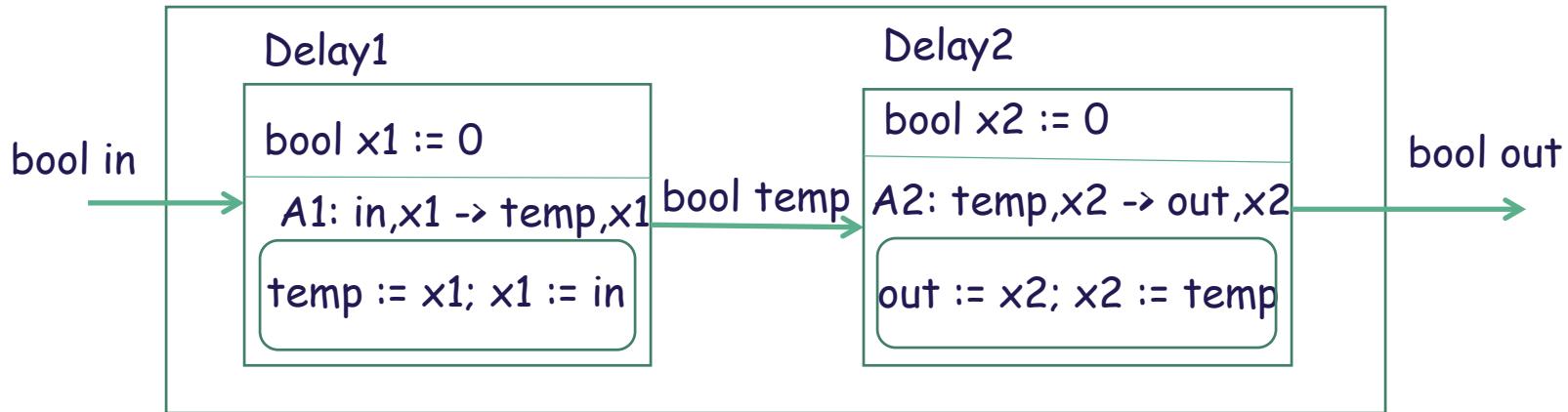
Properties of Parallel Composition

- Commutative: $C_1 \parallel C_2$ is same as $C_2 \parallel C_1$
- Associative: Given C_1, C_2, C_3 , all of $(C_1 \parallel C_2) \parallel C_3, C_1 \parallel (C_2 \parallel C_3), (C_1 \parallel C_3) \parallel C_2, \dots$ give the same result
 - If compatibility check fails in one case, will also fail in others
 - Bottomline: Order in which components are composed does not matter
- If both C_1 and C_2 are finite-state, then so is product $C_1 \parallel C_2$
 - If C_1 has n_1 states and C_2 has n_2 states then the product has $(n_1 \times n_2)$ states
- If both C_1 and C_2 are deterministic, then so is product $C_1 \parallel C_2$

Output Hiding

- Given a component C , and an output variable y , the result of hiding y in C , written as $C \setminus y$, is basically the same component as C , but y is no longer an output variable, and becomes a local variable
 - Not available to the outside world
 - Useful for limiting the scope (encapsulation)

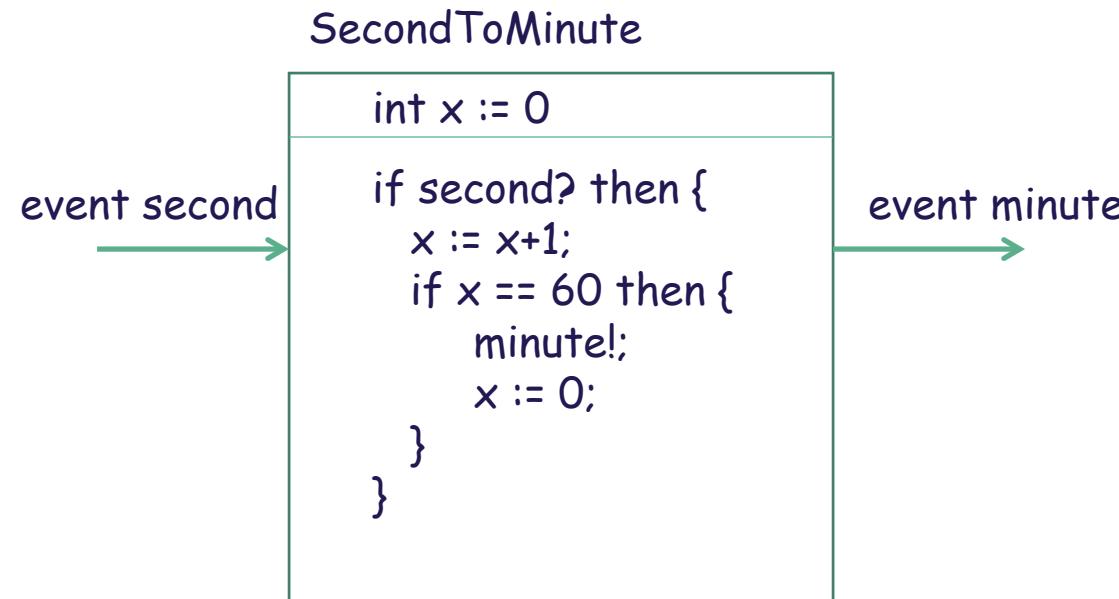
DoubleDelay



Second-To-Minute

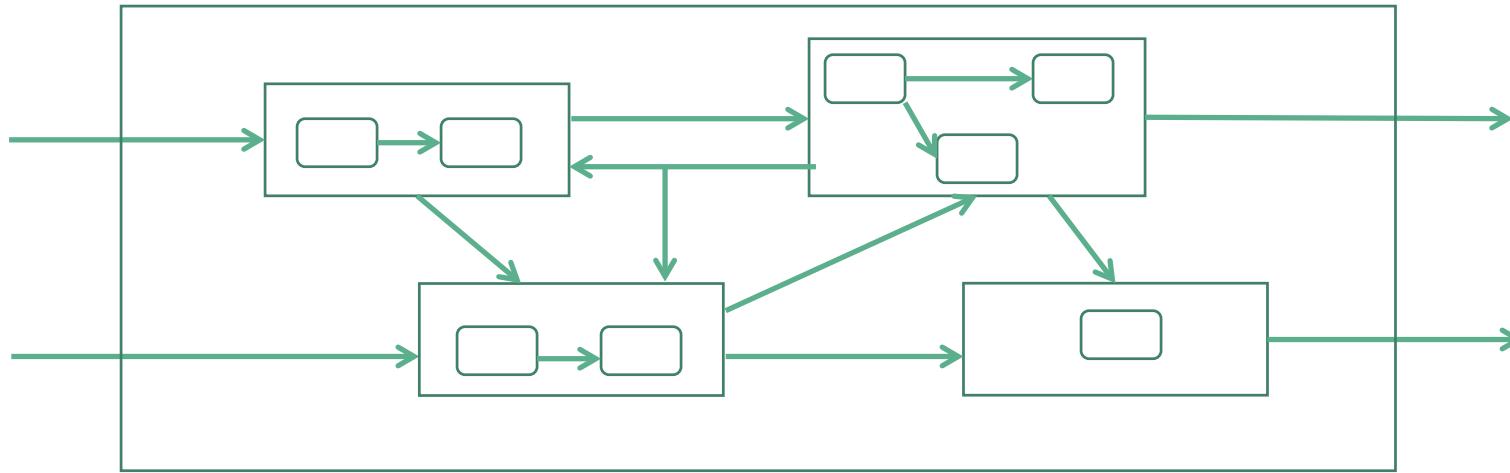
Desired behavior (specification):

Issue the output event every 60th time the input event is present



- Design the component Second-To-Hour such that it issues its output every 3600th time its input event is present

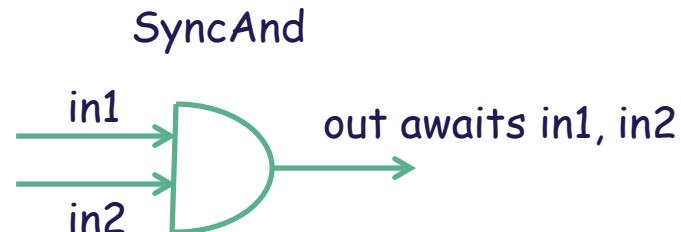
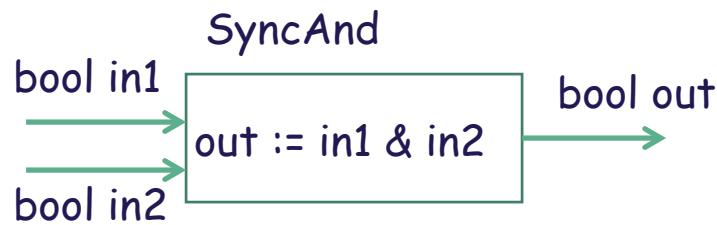
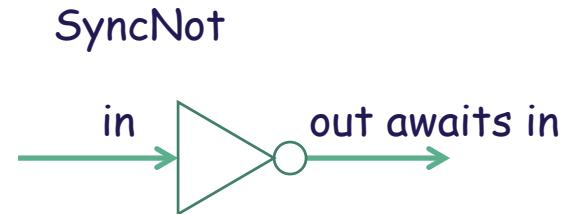
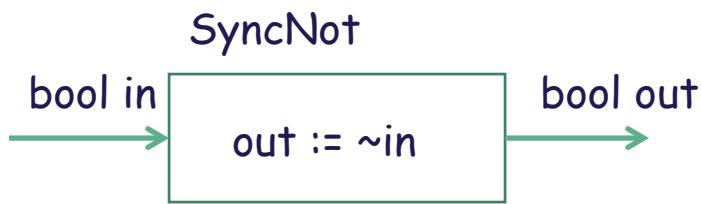
Synchronous Block Diagrams



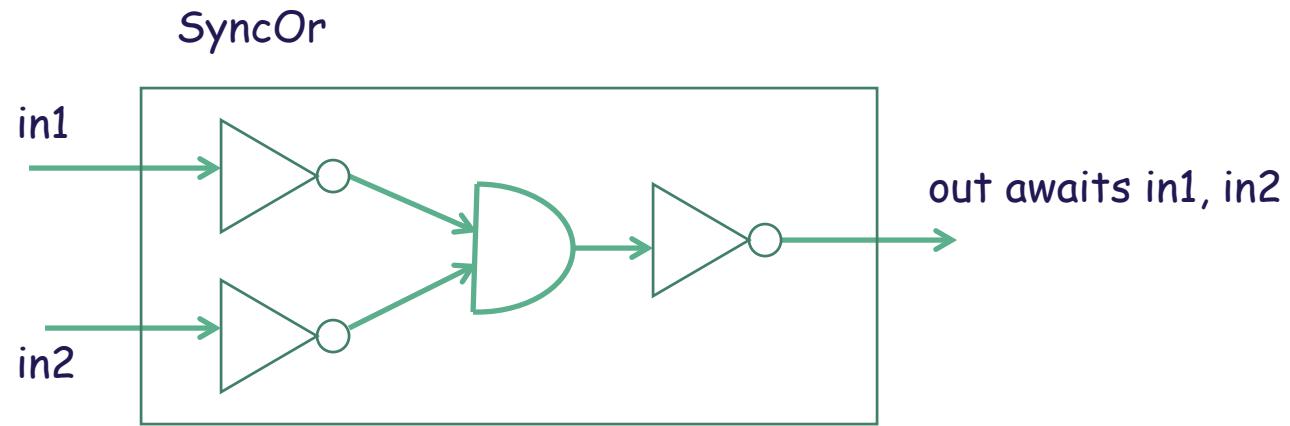
Bottom-up Design

- Design basic components
- Compose existing components in block-diagrams to build new components
- Maintain a library of components, and try to reuse at every step
- Canonical example: Synchronous circuits

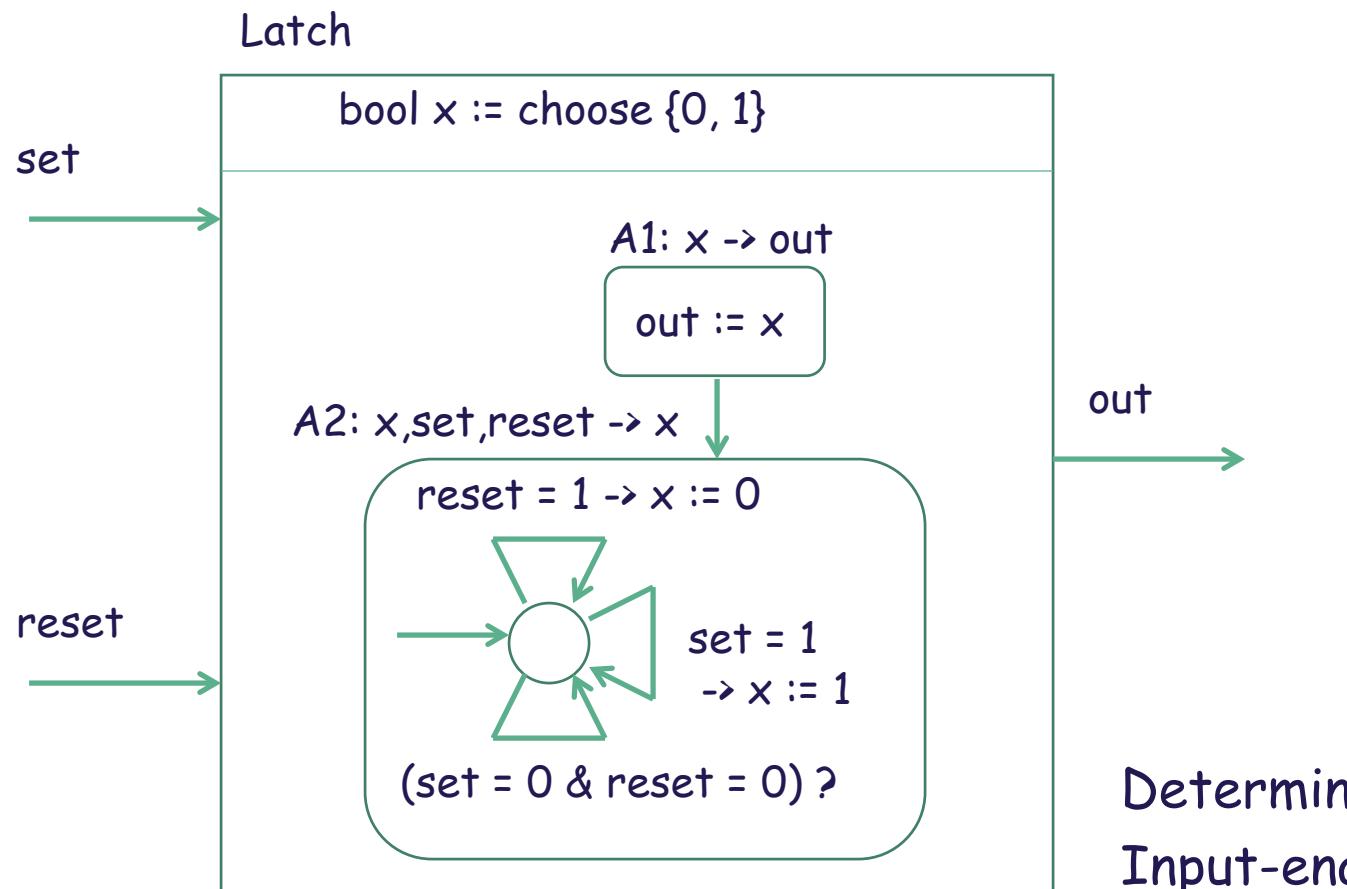
Combinational Circuits



Designing OR Gate

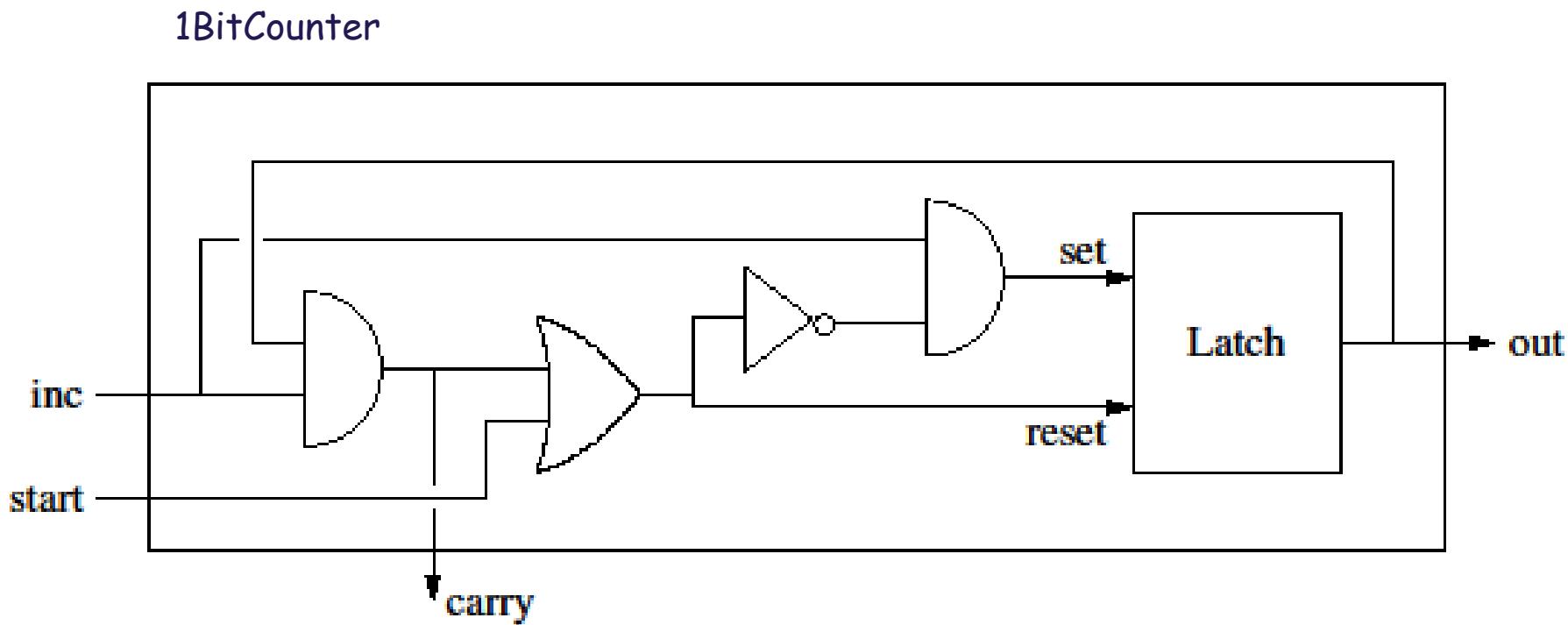


Synchronous Latch



Deterministic?
Input-enabled?

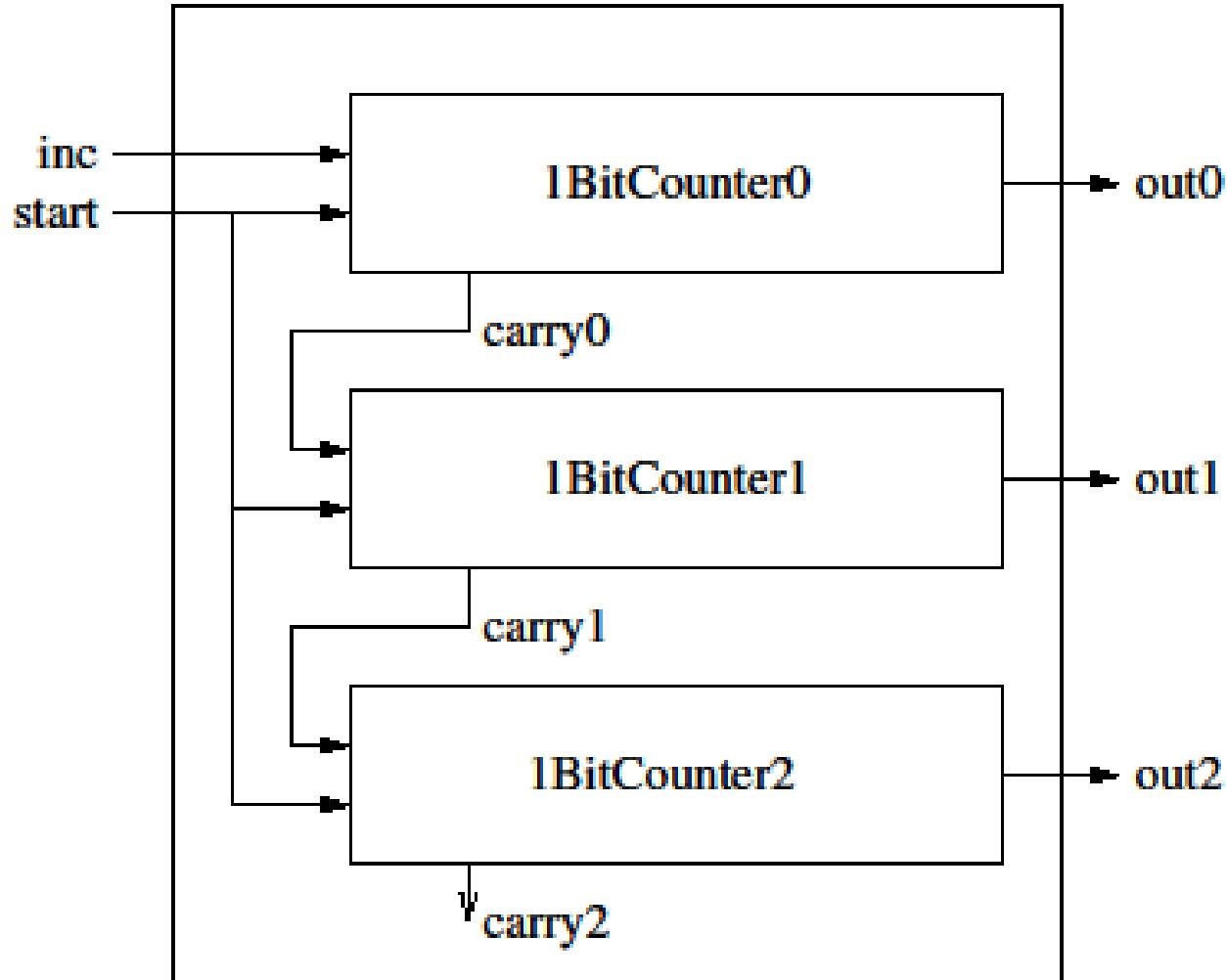
Designing Counter Circuit (1)



- Are await-dependencies acyclic?

Designing Counter Circuit (2)

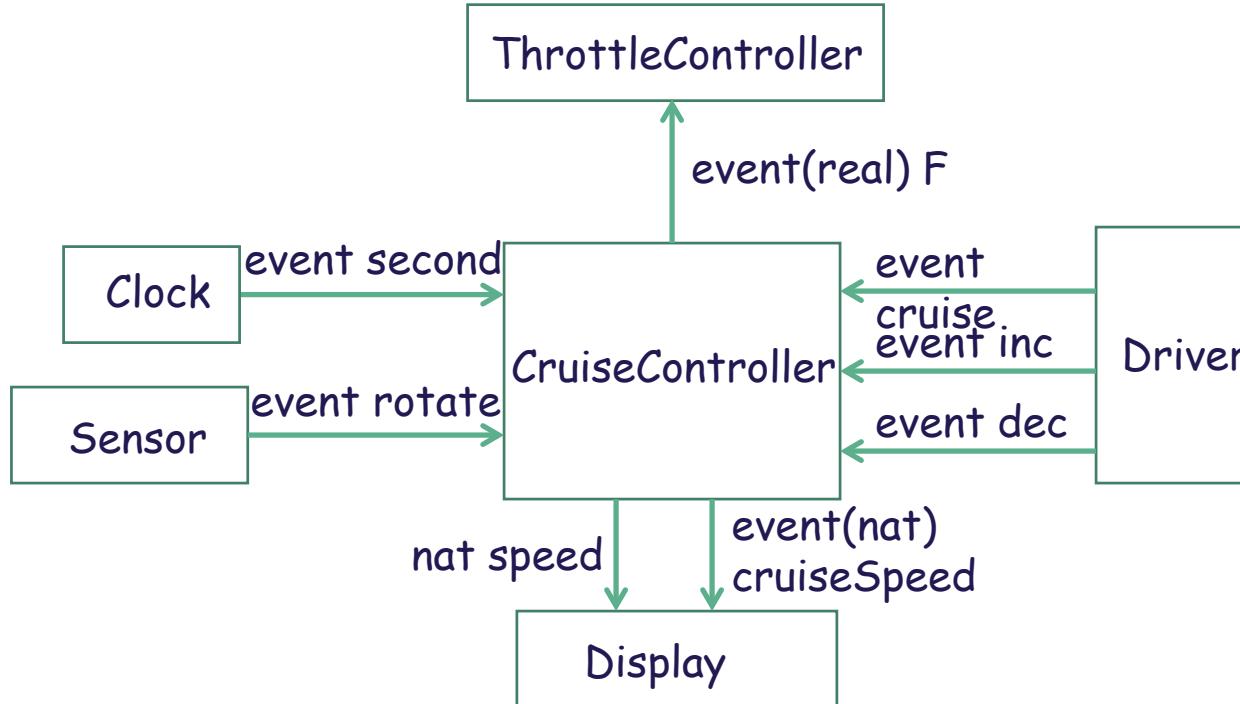
Construct a 3BitCounter



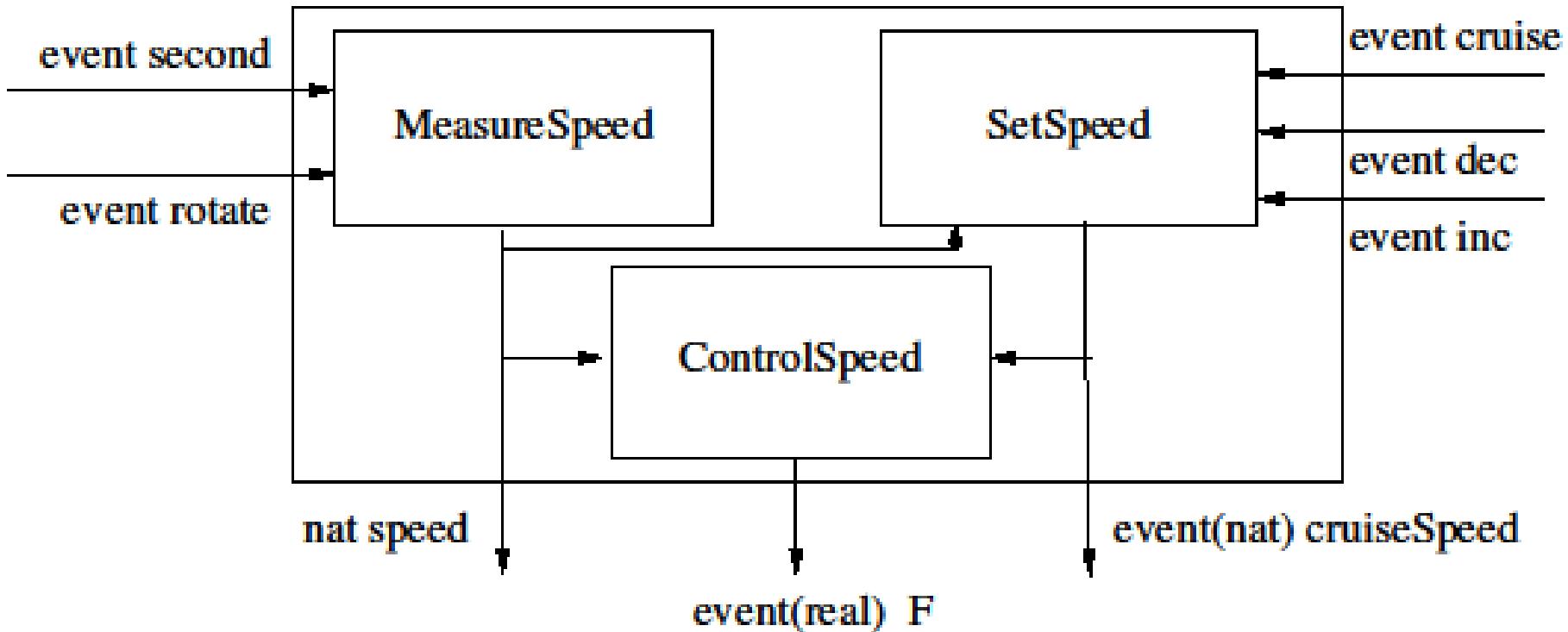
Top-Down Design

- Starting point: Inputs and outputs of desired design C
- Models/assumptions about the environment in which C operates
- Informal/formal description of desired behavior of C
- Example: Cruise Controller

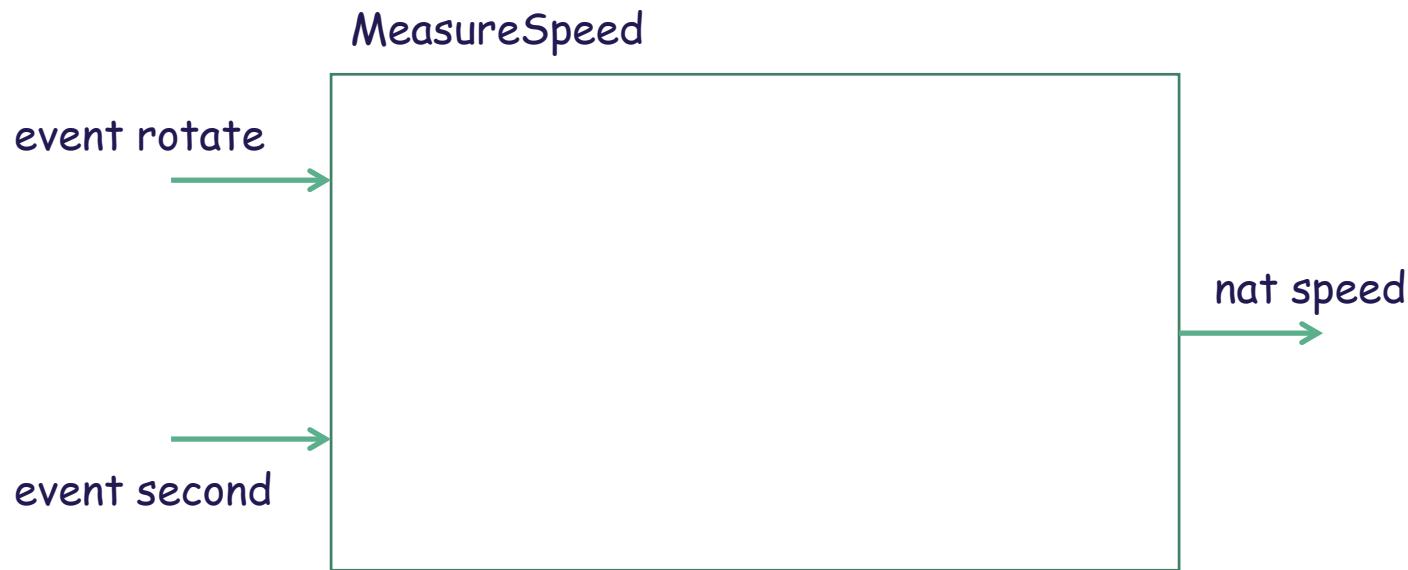
Top-Down Design of a Cruise Controller



Decomposing CruiseController

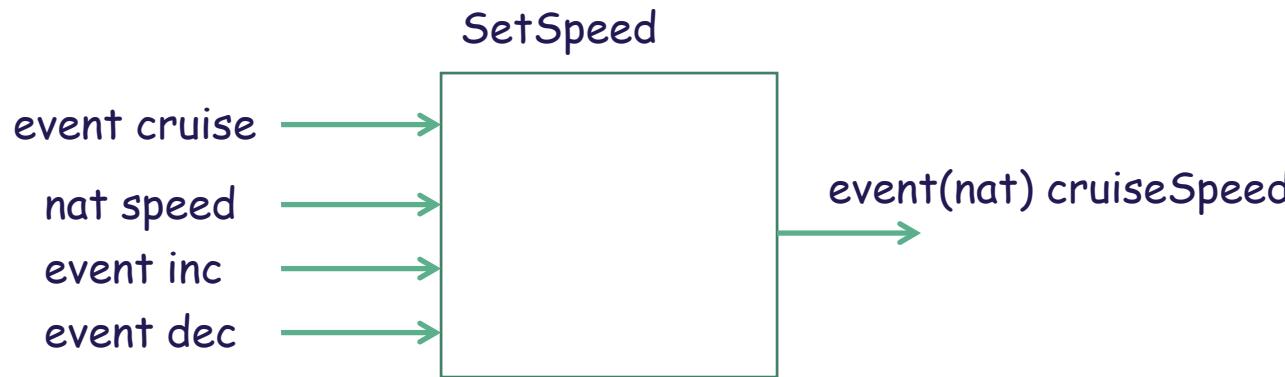


Tracking Speed



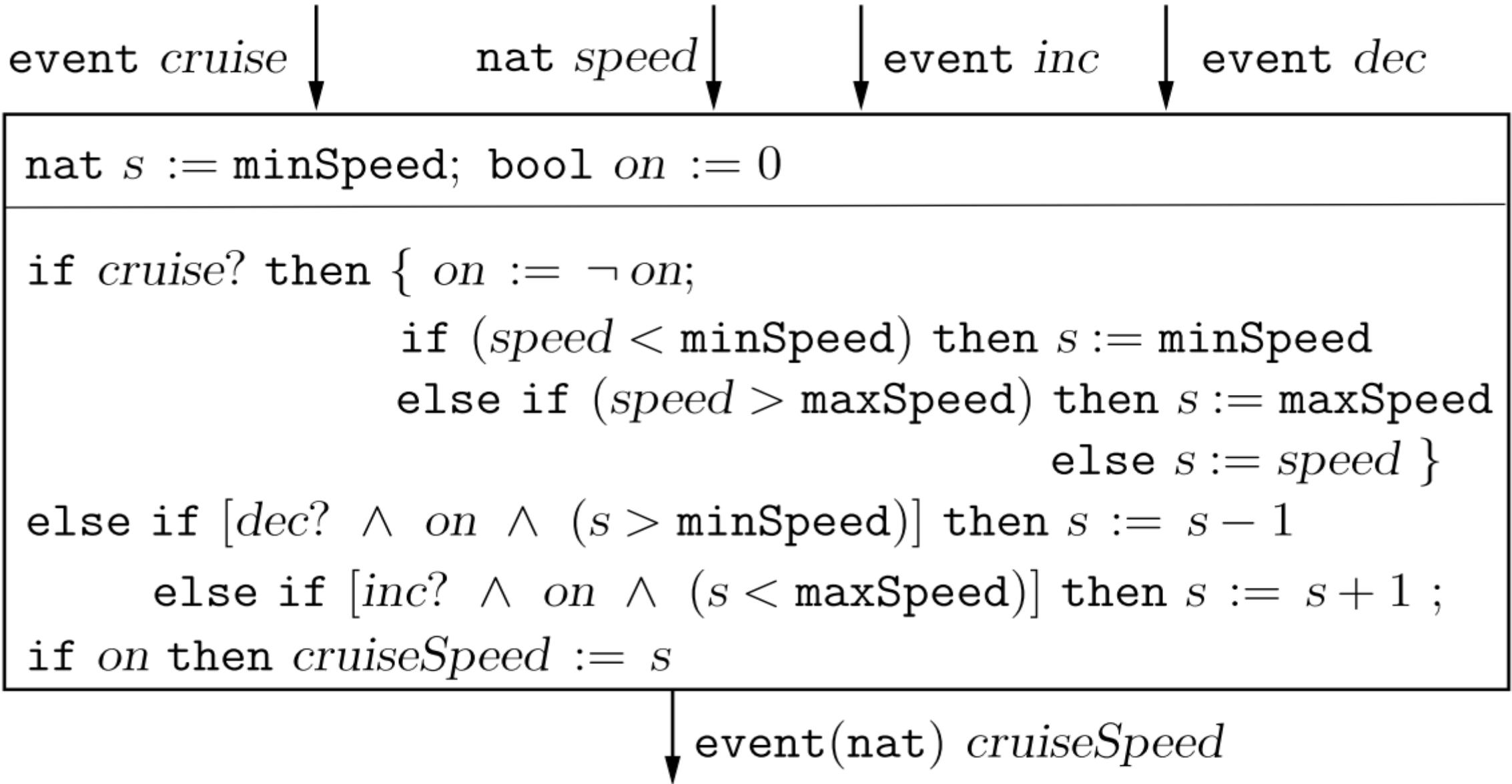
- Inputs: Events rotate and second
- Output: current speed
- Computes the number of rotate events per second

Tracking Cruise Settings

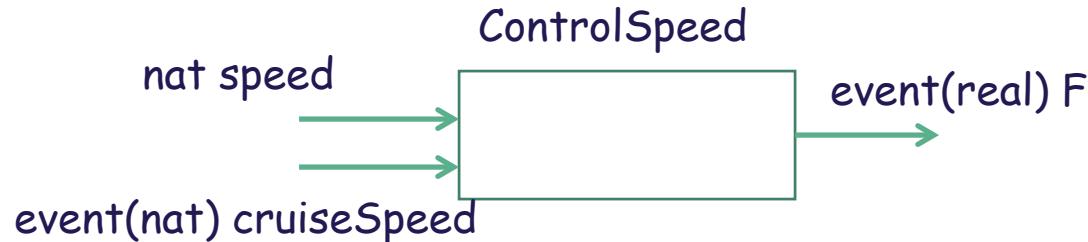


- Inputs from the driver: Commands to turn the cruise-control on/off and to increment/decrement desired cruising speed from driver
- Input: Current speed
- Output: Desired cruising speed
- What assumptions can we make about simultaneity of events?
- Should we include safety checks to keep desired speed within bounds?

Tracking Cruise Settings

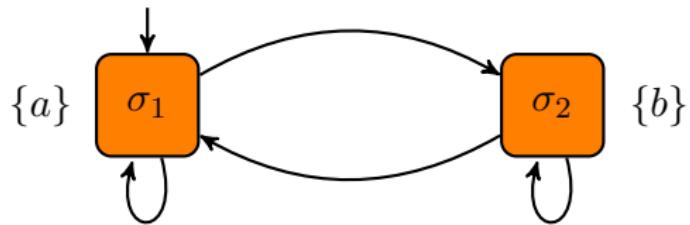


Controlling Speed

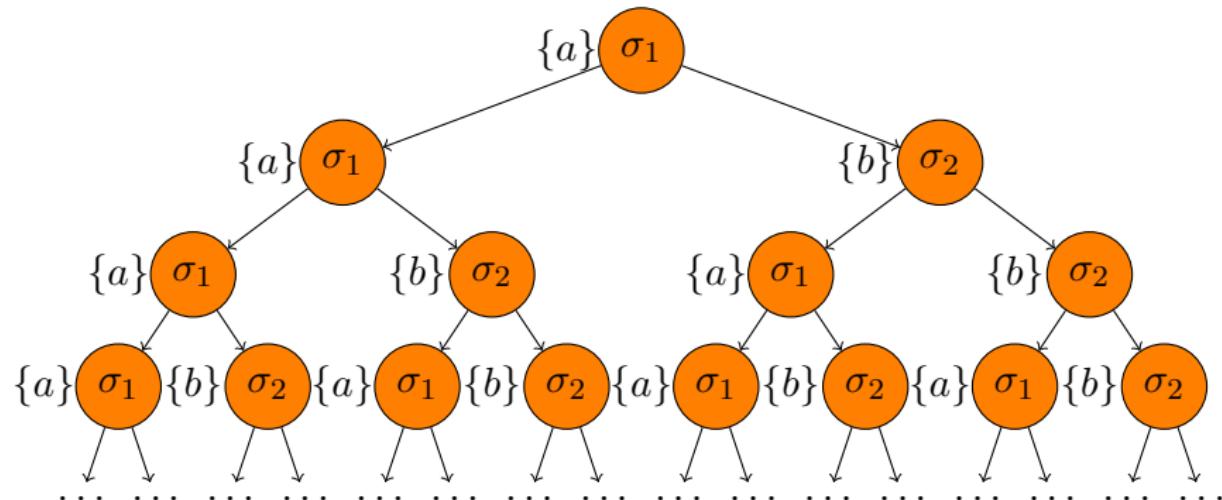
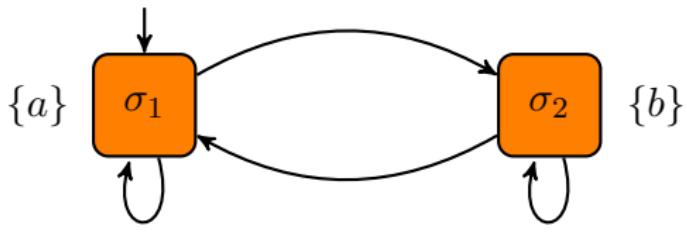


- Inputs: Actual speed and desired speed
- Output: Pressure on the throttle
- Goal: Make actual speed equal to the desired speed (while maintaining key physical properties such as stability)
- Design relies on theory of dynamical systems

Computation tree of a (labeled state) transition system \mathcal{T}



Computation tree of a (labeled state) transition system \mathcal{T}



CTL syntax

CTL state formulas:

$$\psi ::= a \mid (\psi \wedge \psi) \mid (\neg\psi) \mid (E\varphi) \mid (A\varphi)$$

where a is an atomic proposition and φ is a CTL path formula

CTL path formulas:

$$\varphi ::= \Box\psi \mid \Diamond\psi \mid \underbrace{\Box\psi \mid \psi U \psi}_{\text{not needed in this course}}$$

where ψ are CTL state formulas

CTL formulas are **CTL state formulas**

We omit parentheses when causing no confusion

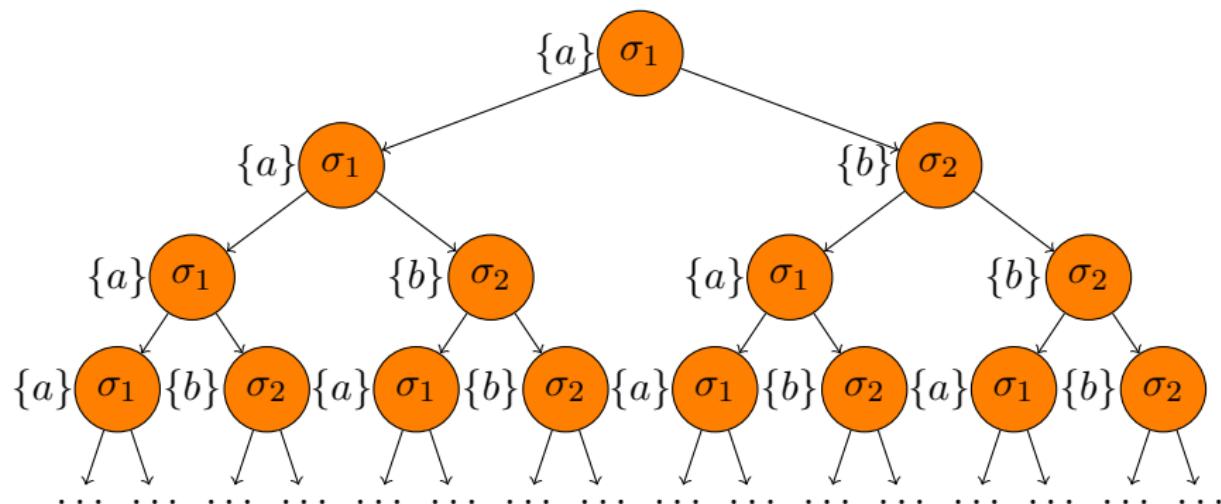
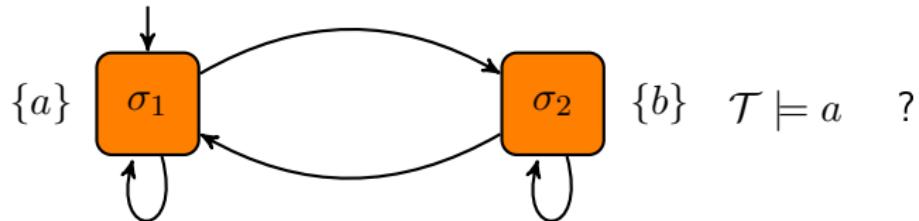
CTL semantics

Convention: For a path $\pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots$, let $\pi(i)$ denote σ_i

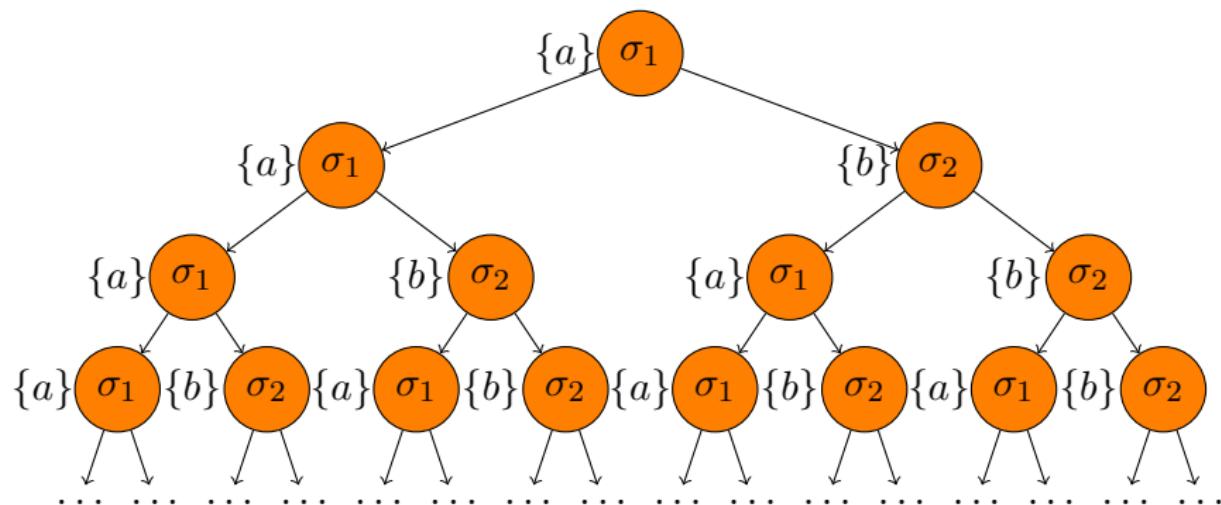
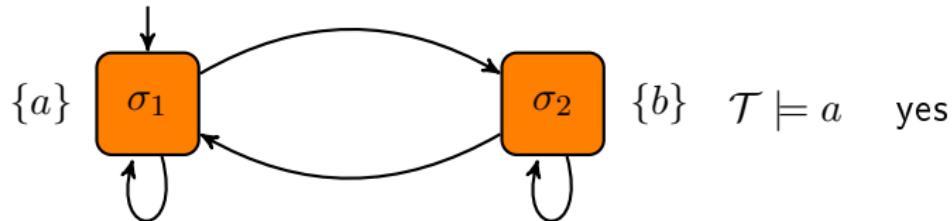
$\sigma \models a$	iff a is a label of σ
$\sigma \models \psi_1 \wedge \psi_2$	iff $\sigma \models \psi_1$ and $\sigma \models \psi_2$
$\sigma \models \neg\psi$	iff $\sigma \not\models \psi$
$\sigma \models E\varphi$	iff $\pi \models \varphi$ for <u>some</u> $\pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots$ with $\sigma_0 = \sigma$
$\sigma \models A\varphi$	iff $\pi \models \varphi$ for <u>all</u> $\pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots$ with $\sigma_0 = \sigma$
$\pi \models \Box\psi$	iff $\pi(i) \models \psi$ for <u>all</u> $i \geq 0$
$\pi \models \Diamond\psi$	iff $\pi(i) \models \psi$ for <u>some</u> $i \geq 0$
$\pi \models \bigcirc\psi$	iff $\pi(1) \models \psi$
$\pi \models \psi_1 \cup \psi_2$	iff $\exists j \geq 0 \ \pi(j) \models \psi_2 \wedge \forall 0 \leq i < j \ \pi(i) \models \psi_1$

$\mathcal{T} \models \psi$ iff $\sigma_0 \models \psi$ for all initial states σ_0 of transition system \mathcal{T}

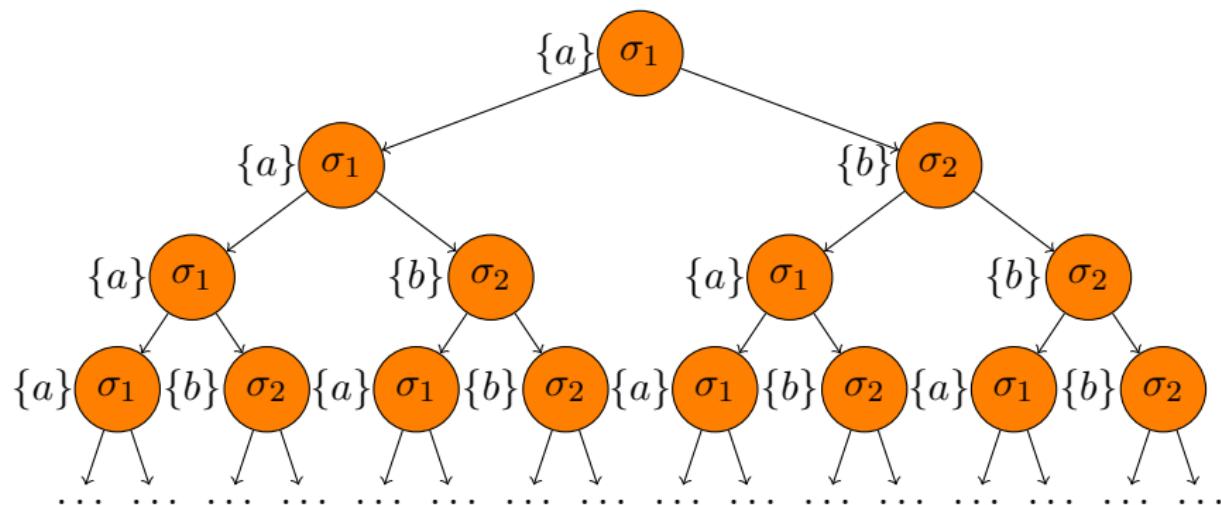
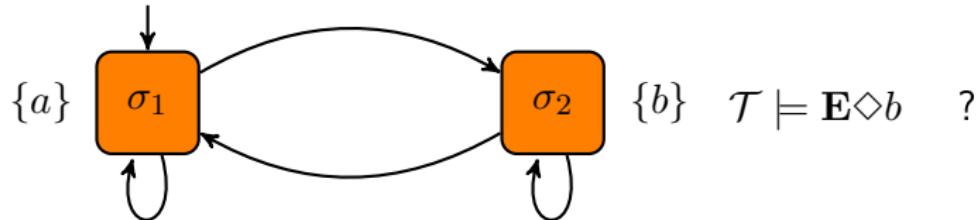
Computation tree



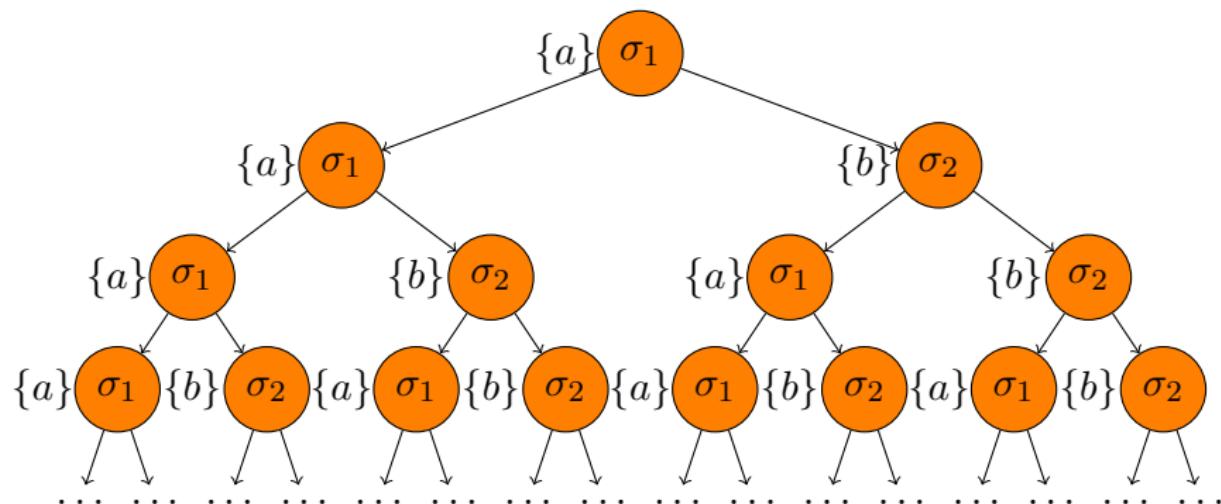
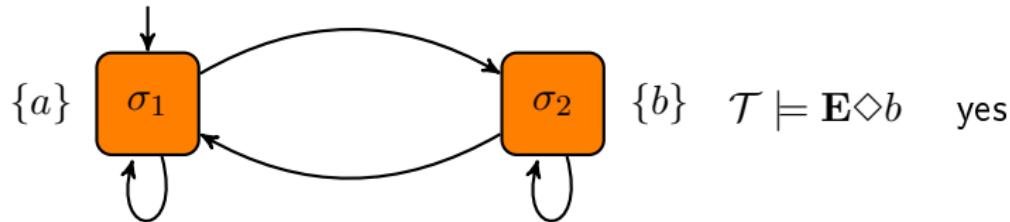
Computation tree



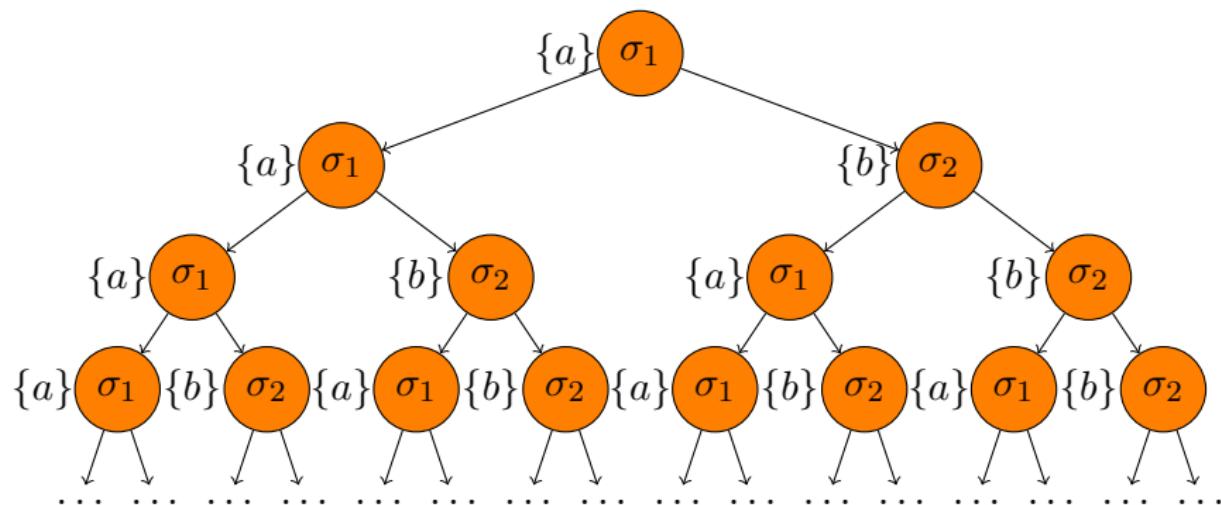
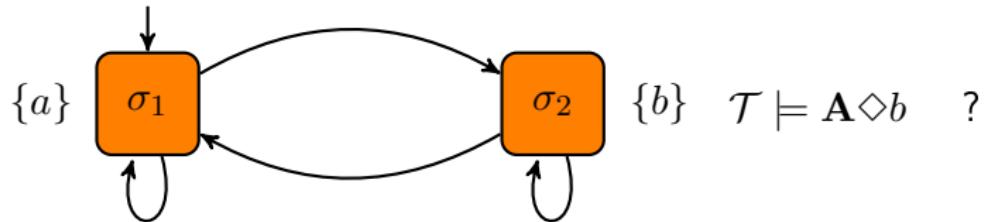
Computation tree



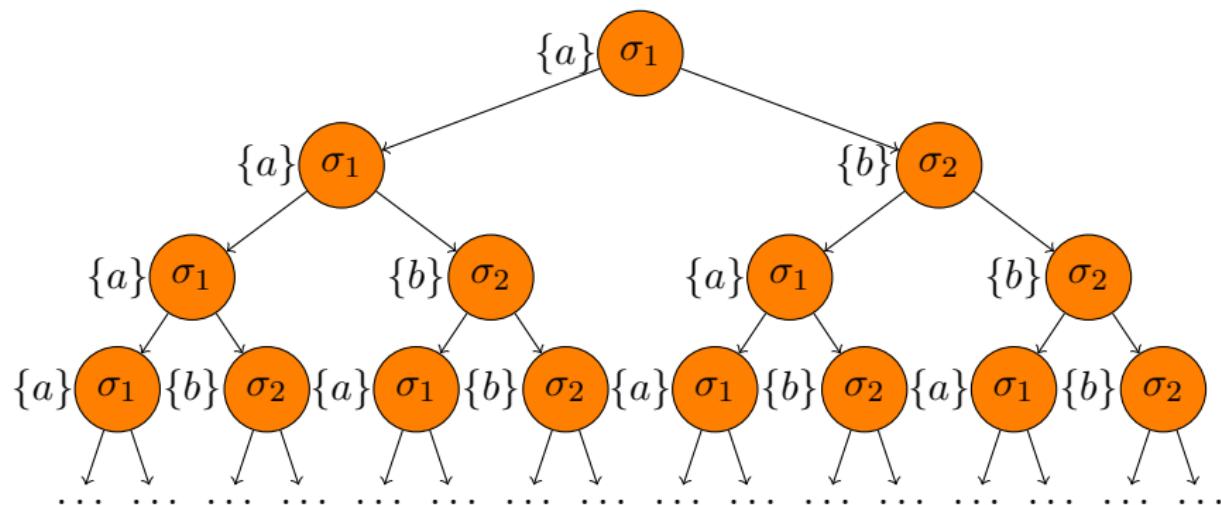
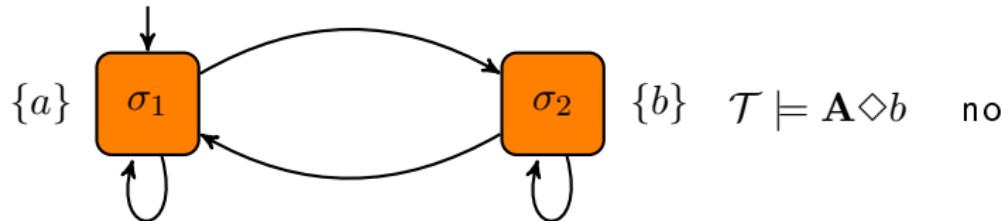
Computation tree



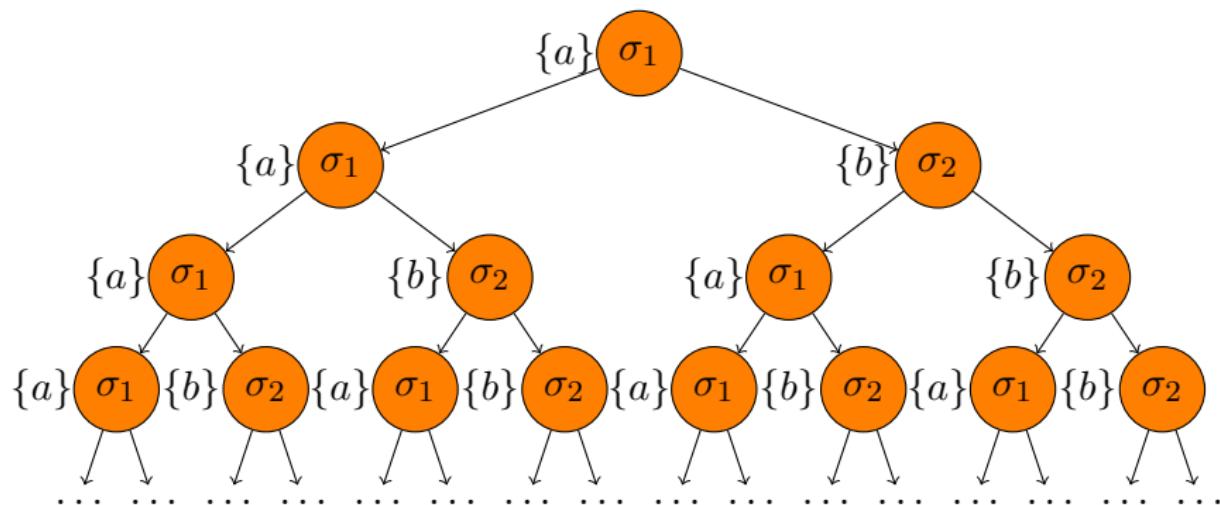
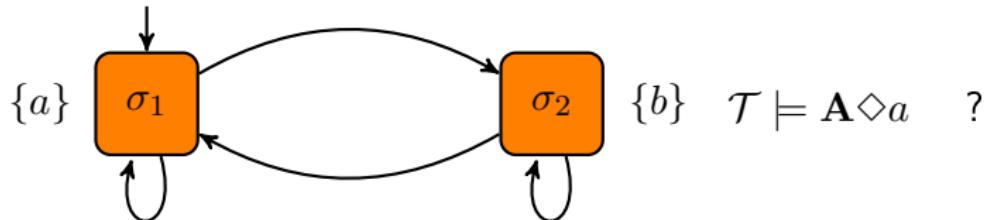
Computation tree



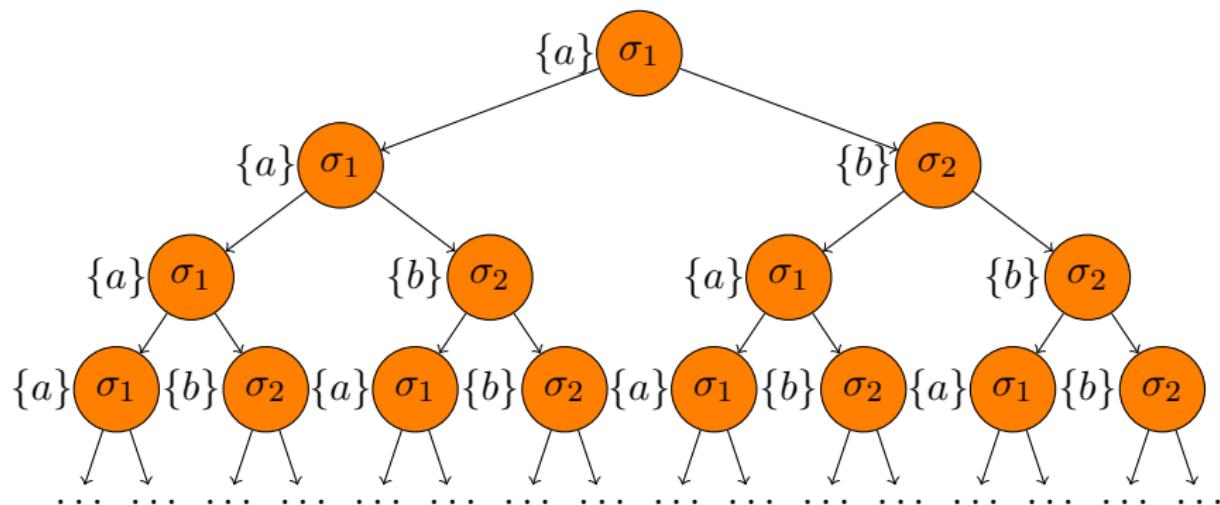
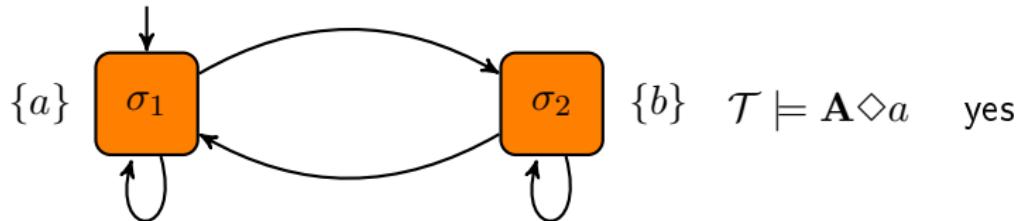
Computation tree



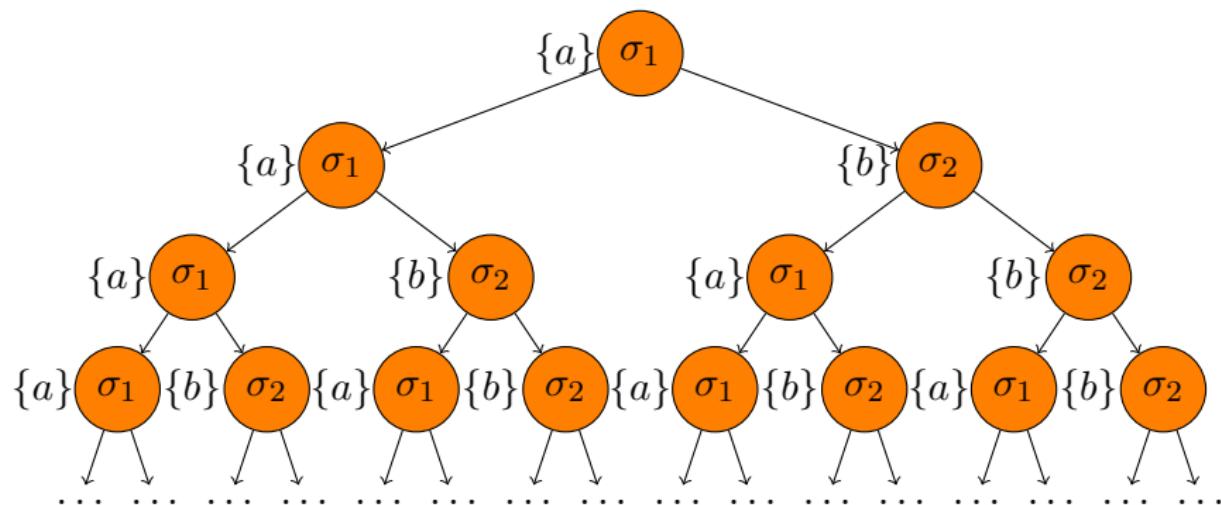
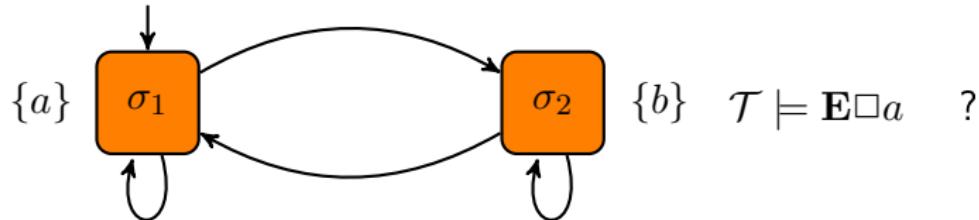
Computation tree



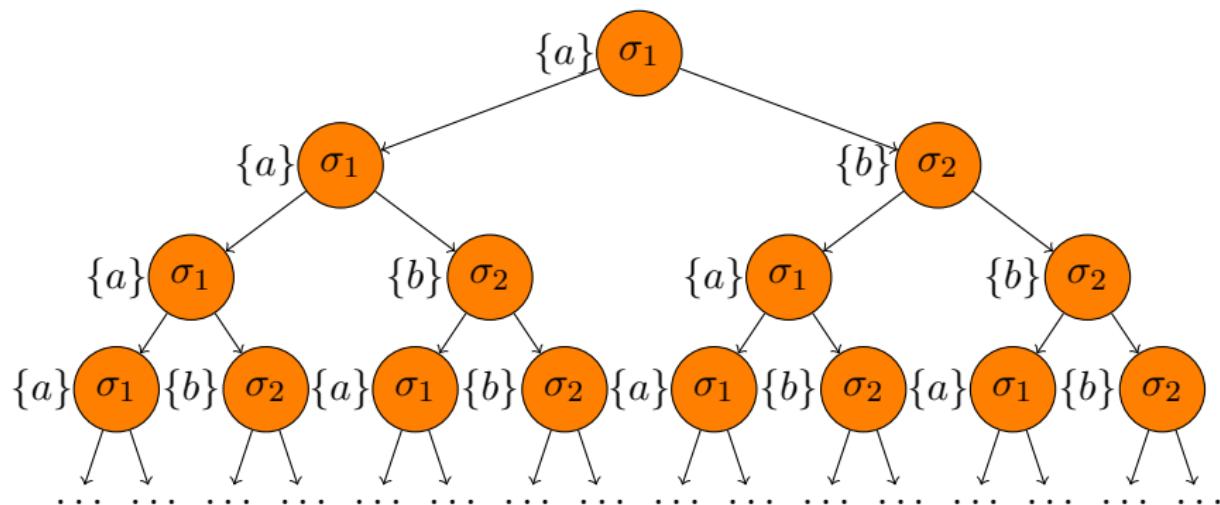
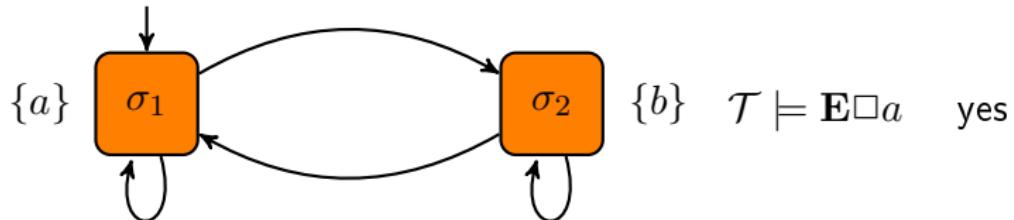
Computation tree



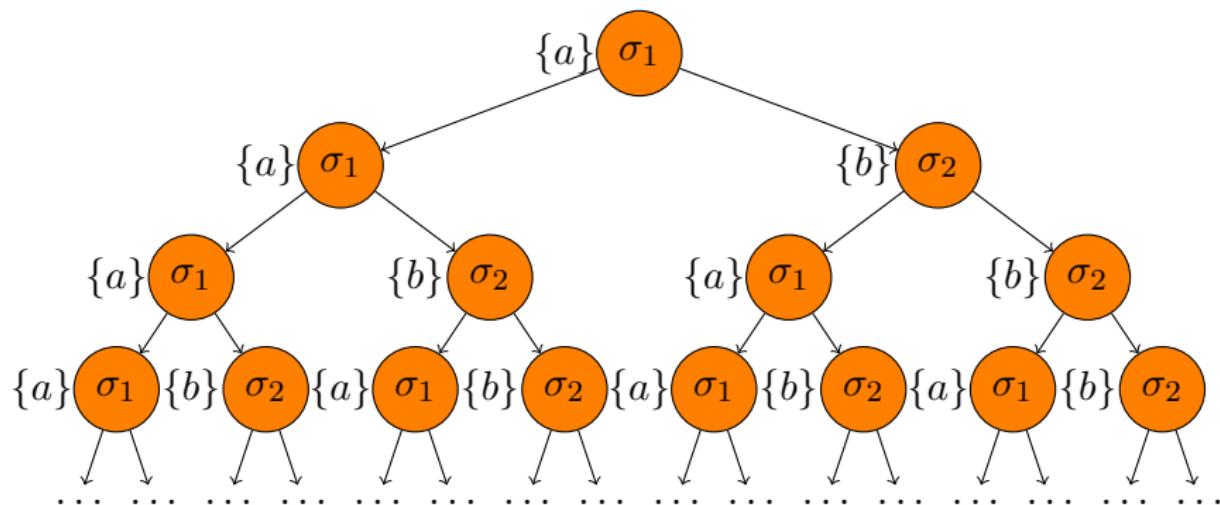
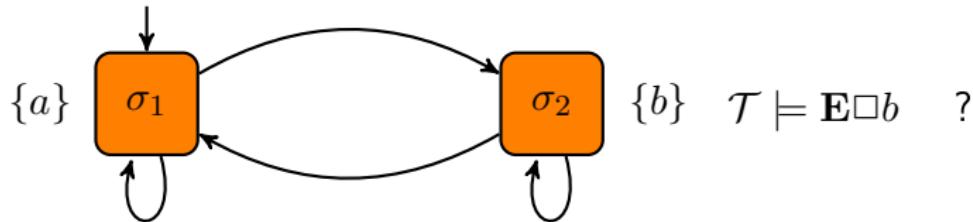
Computation tree



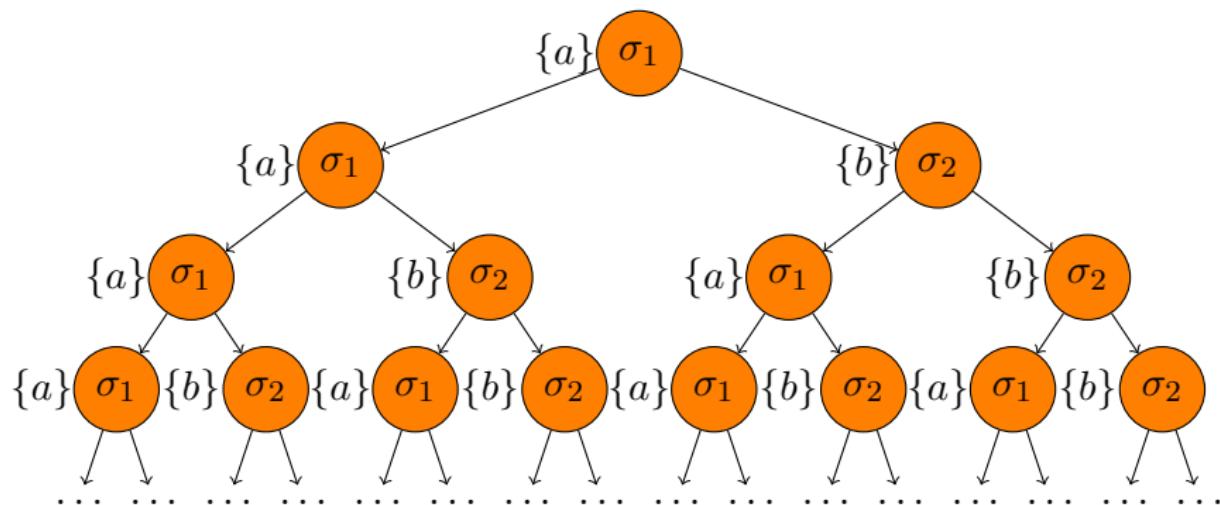
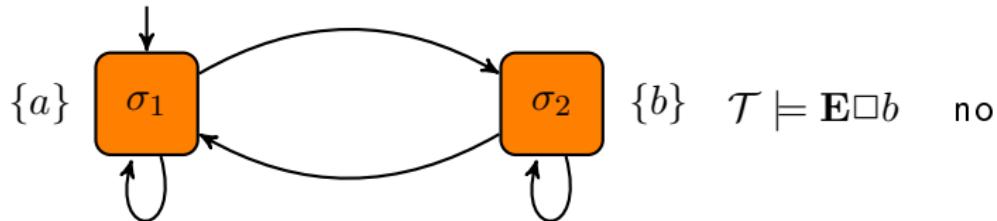
Computation tree



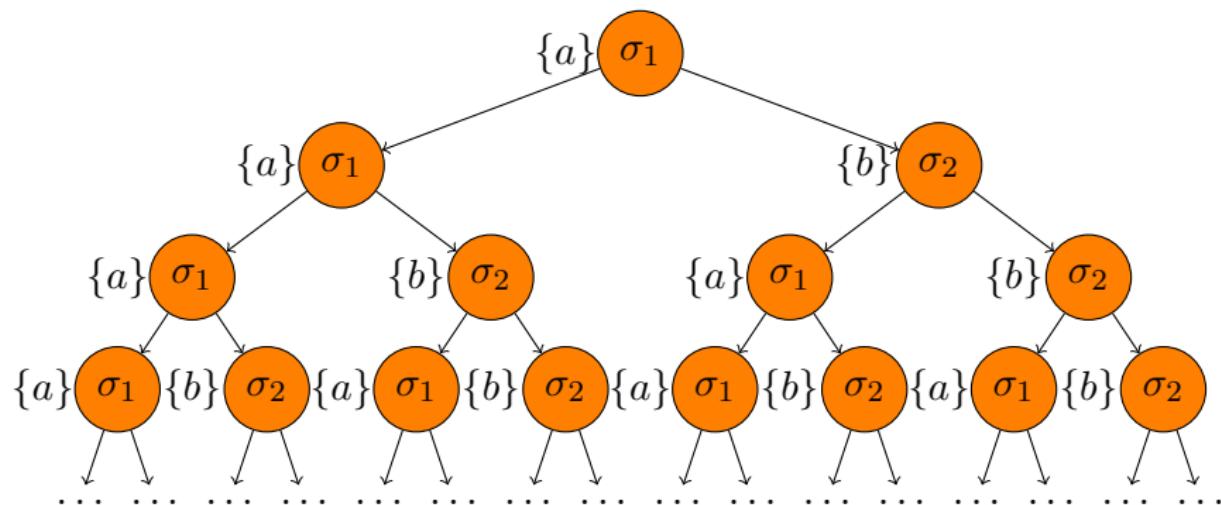
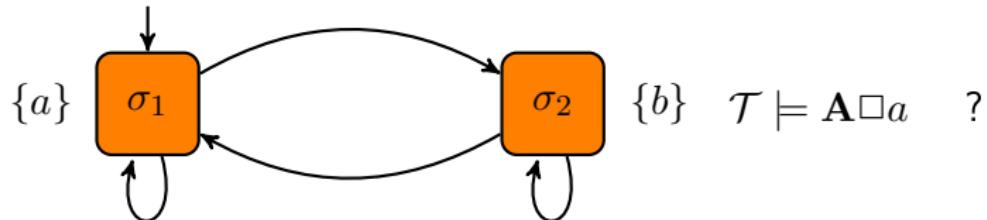
Computation tree



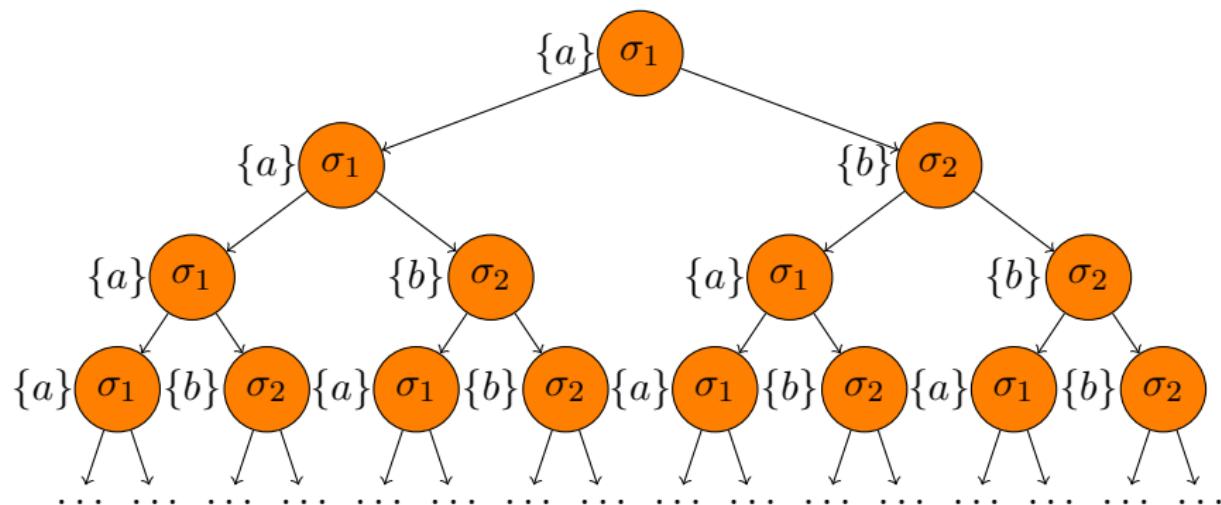
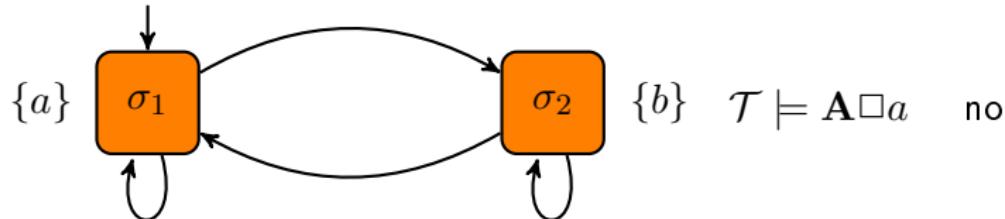
Computation tree



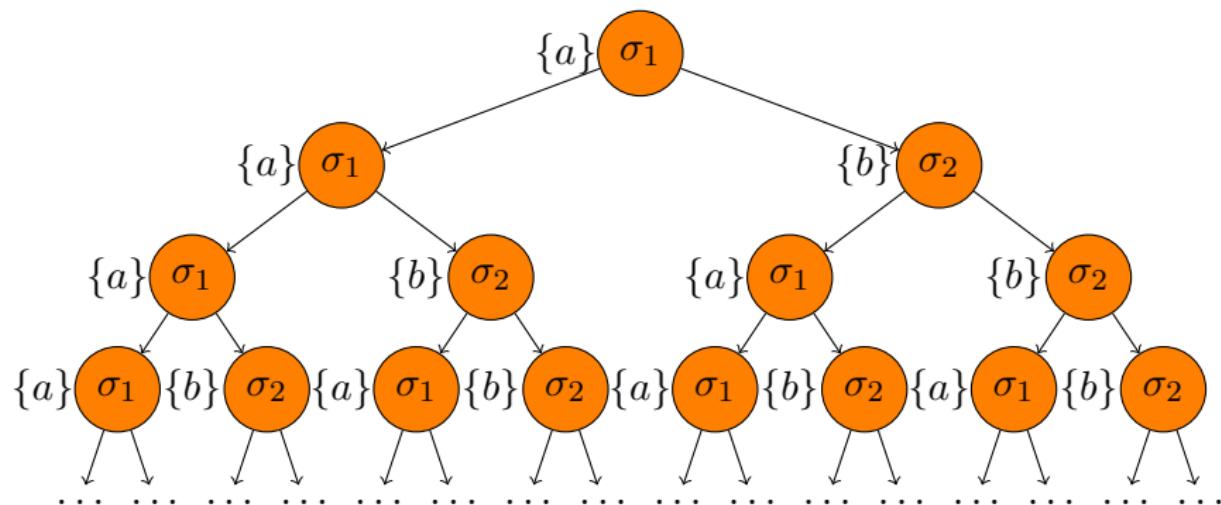
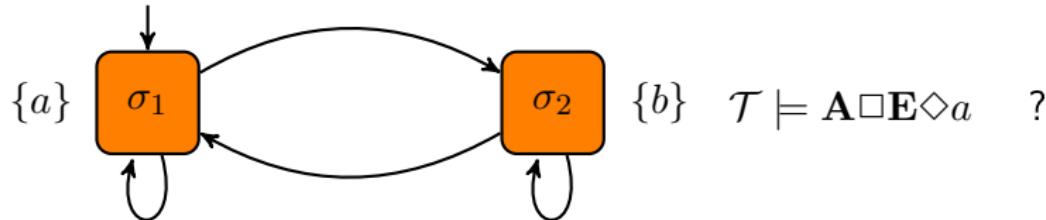
Computation tree



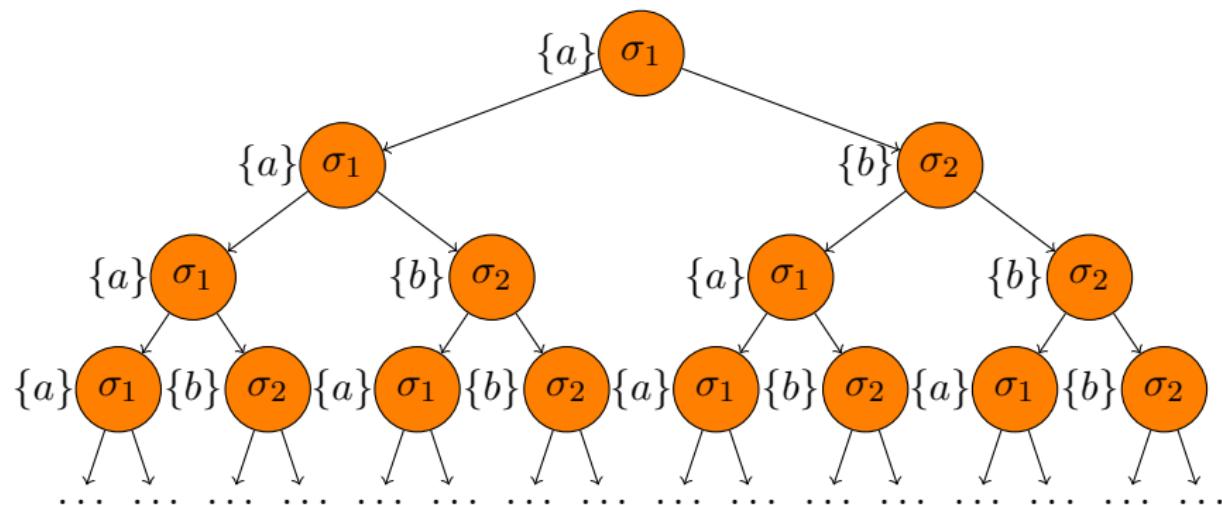
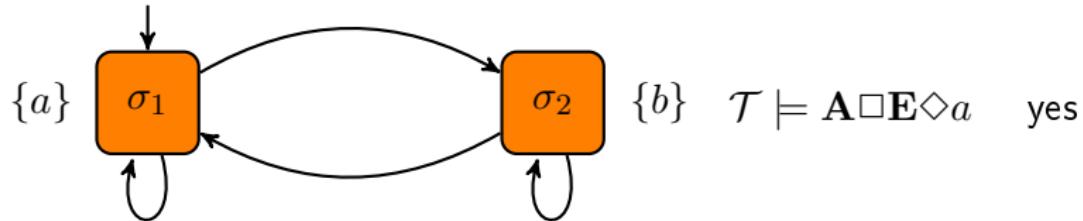
Computation tree



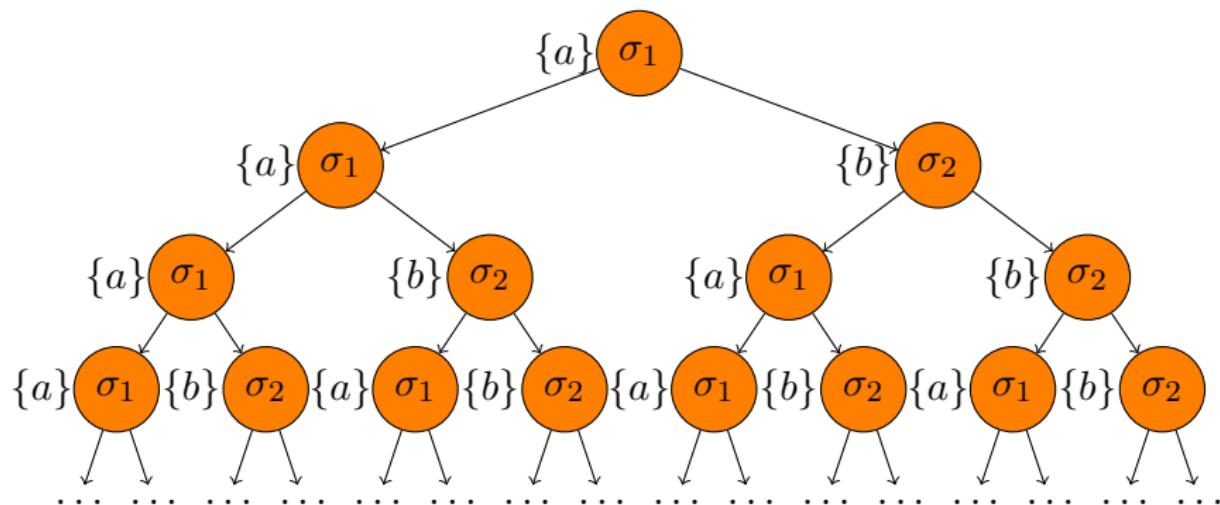
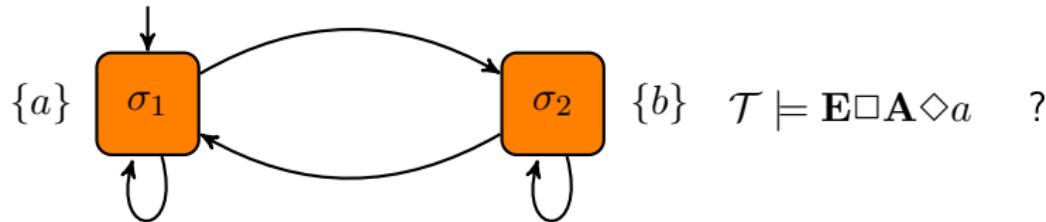
Computation tree



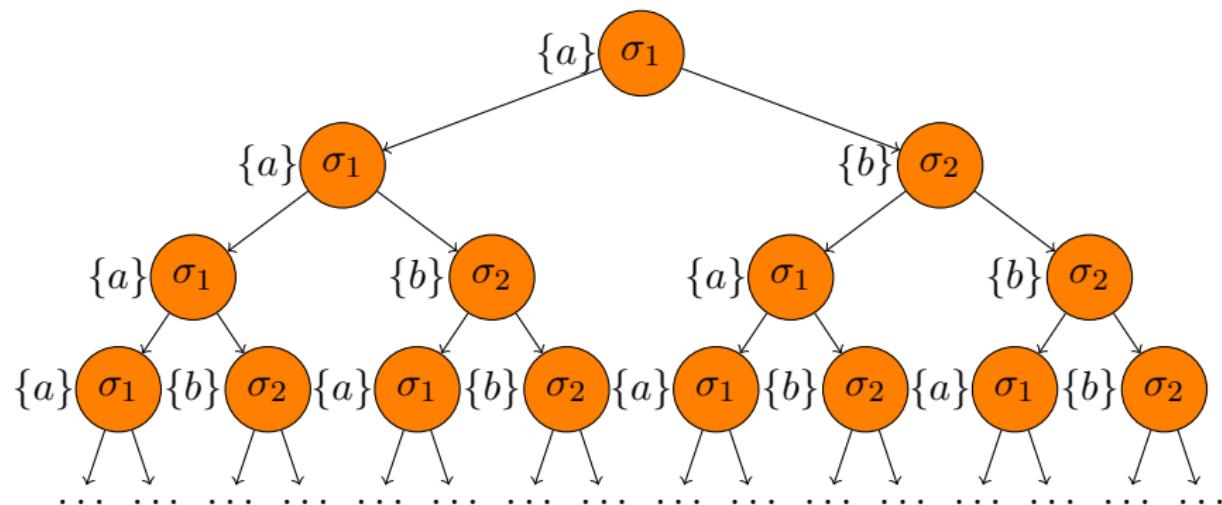
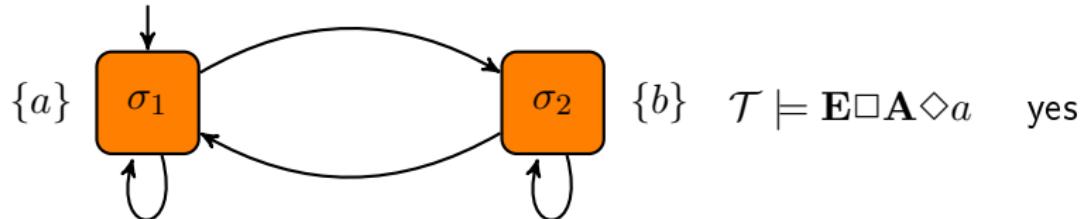
Computation tree



Computation tree



Computation tree



Models and Tools for Cyber-Physical Systems

Chapter 3: Safety Requirements

Instructors: Martijn Goorden, Kim G. Larsen,
Christian Schilling, Max Tschaikowski
{mgoorden,kgl,christianms,tschaikowski}@cs.aau.dk

Slides courtesy of Rajeev Alur
alur@cis.upenn.edu

Requirements

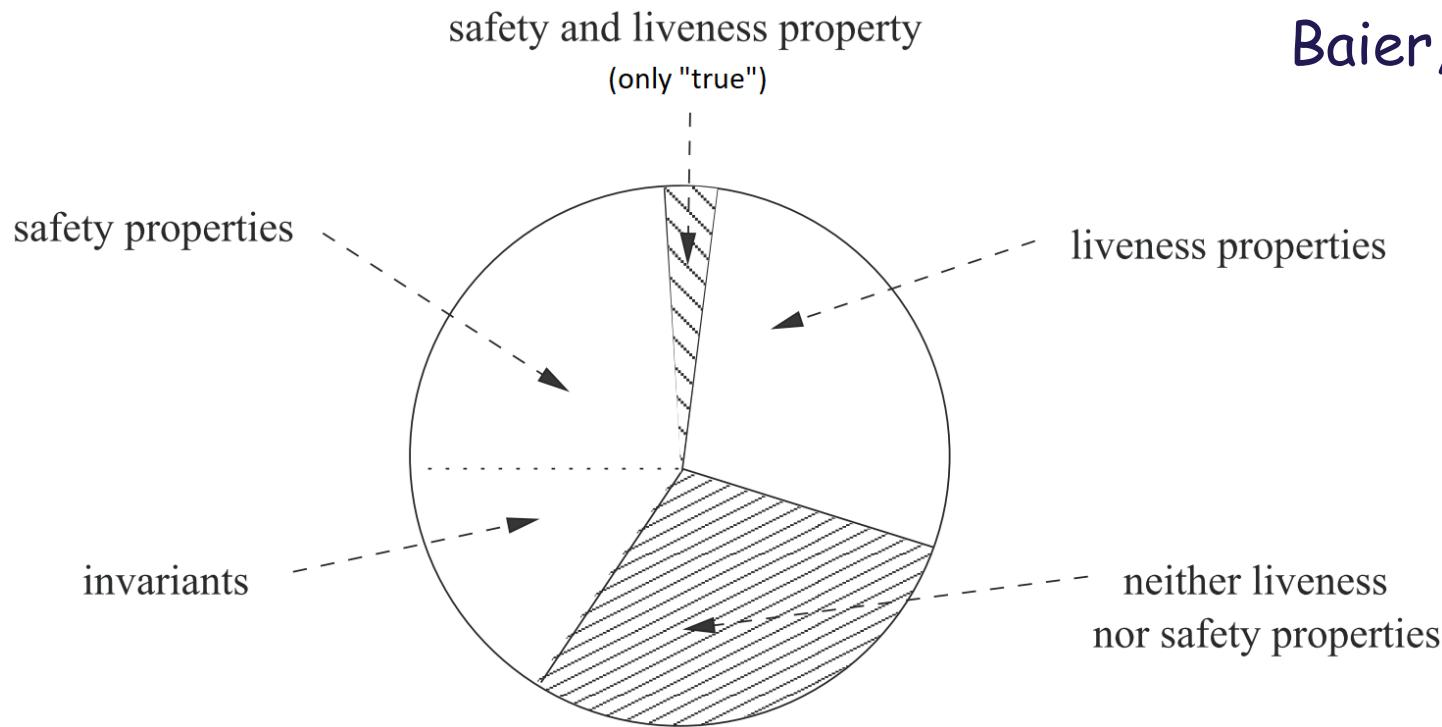
- Requirement: Desirable property of the executions of the system
 - Informal: Either implicit or stated in English
 - Formal: Stated explicitly in a mathematically precise manner
 - Wanted for high assurance / safety-critical systems
- Model/design/system meets the requirements if every execution satisfies 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/model C satisfies the requirement φ

Safety and Liveness Requirements

- A safety requirement states that a system always stays within "good" states (i.e., "nothing bad ever happens")
- Cruise controller: Desired speed stays between bounds
- Collision avoidance: Distance between two cars is always greater than some minimum threshold

- A liveness requirement states that a system eventually gets to a goal state (i.e., "something good eventually happens")
 - Cruise controller: Actual speed eventually equals desired speed
 - Multithreading: Each thread eventually terminates
- Formalization and analysis techniques for safety and liveness differ significantly; we focus on safety

Safety vs. Liveness

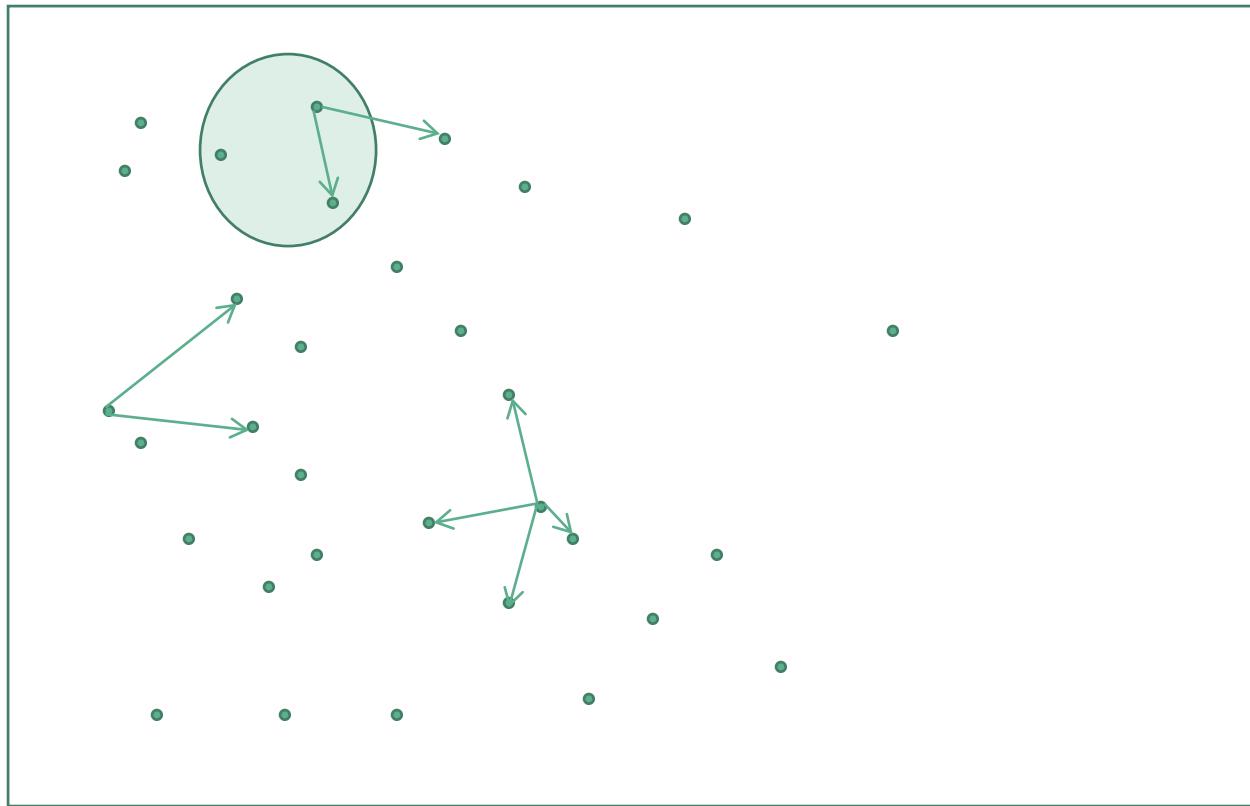


Baier, Katoen 2008

- “ $x = 1$ holds at most once” (safety)
- “ $x = 1$ holds at least once” (liveness)
- “ $x = 1$ holds at exactly once” (neither)
- Every property is the intersection of a safety and a liveness property

Transition Systems

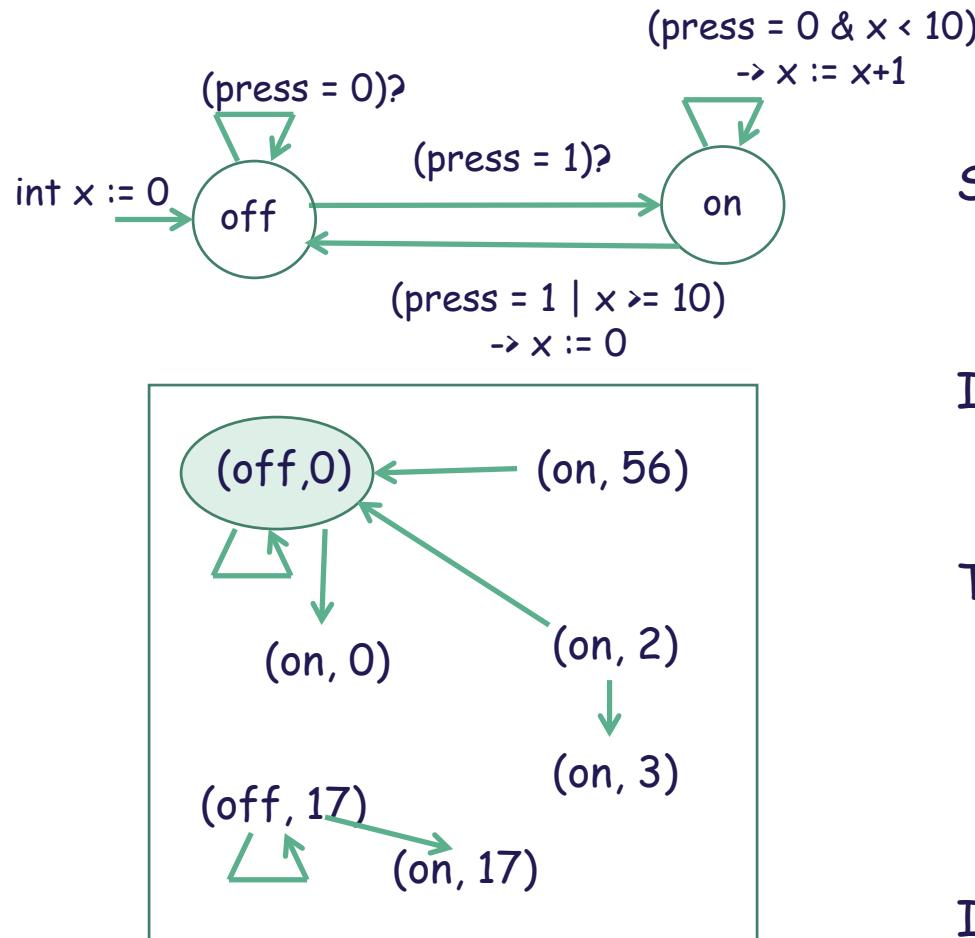
States + Initial states + Transitions between states



Definition of Transition System

- Syntax: a transition system $T = (S, \text{Init}, \text{Trans})$ consists of
 1. a set S of (typed) state variables
 2. Initialization Init for state variables
 3. Transition description Trans to update state variables
- Semantics:
 1. Set Q_S of states
 2. Set $[\text{Init}]$ of initial states (a subset of Q_S)
 3. Set $[\text{Trans}]$ of transitions (a subset of $Q_S \times Q_S$)
- Most systems (e.g., synchronous reactive components, programs, ...) have an underlying transition system

Switch Transition System



State variables:
 $\{\text{off}, \text{on}\}$ mode, int x

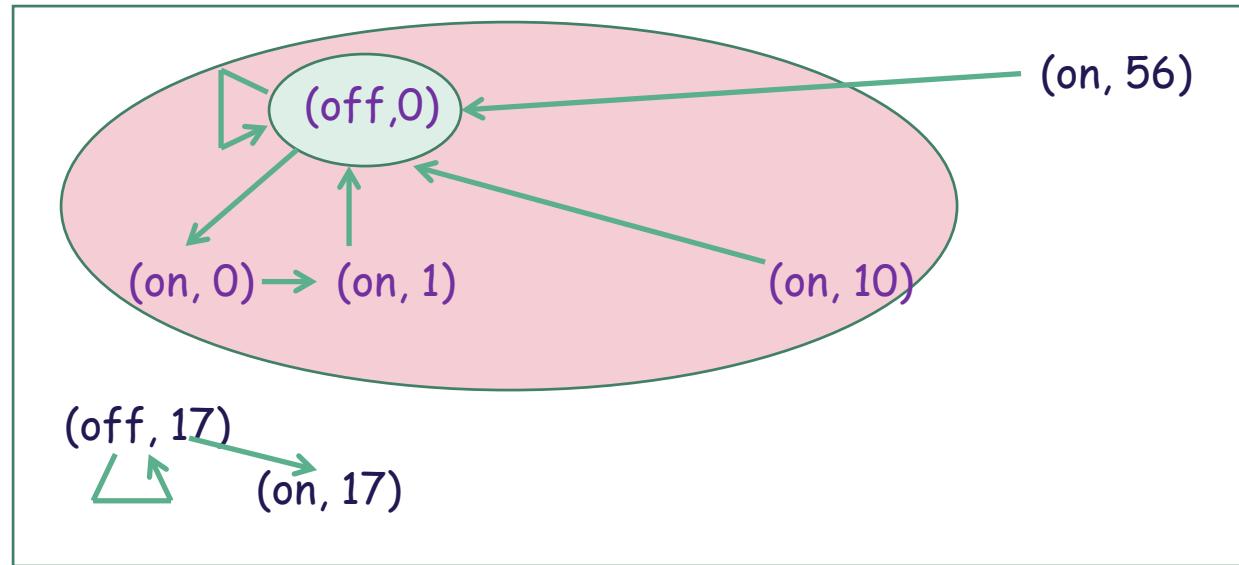
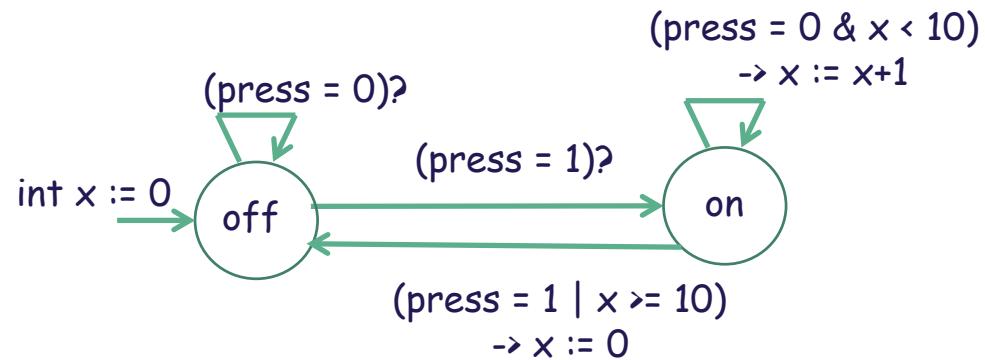
Initialization:
 $\text{mode} = \text{off}; x = 0$

Transitions:
 $(\text{off}, n) \rightarrow (\text{off}, n);$
 $(\text{off}, n) \rightarrow (\text{on}, n);$
 $(\text{on}, n) \rightarrow (\text{on}, n+1) \text{ if } n < 10;$
 $(\text{on}, n) \rightarrow (\text{off}, 0)$

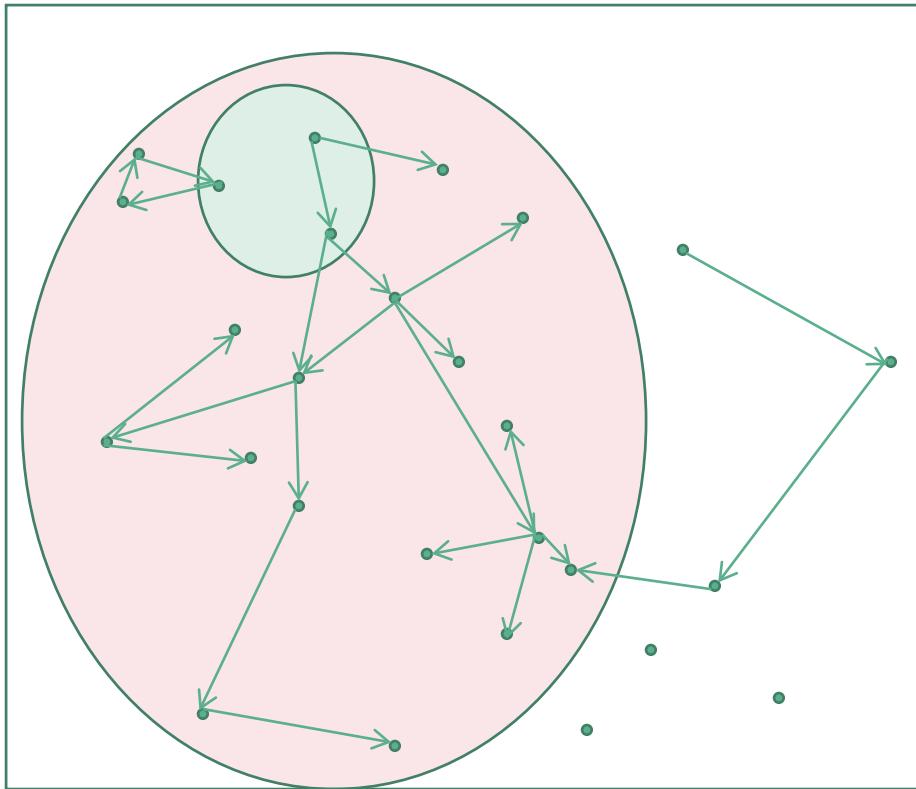
Input/output variables become local variables

Values for input variables chosen nondeterministically

Reachable States

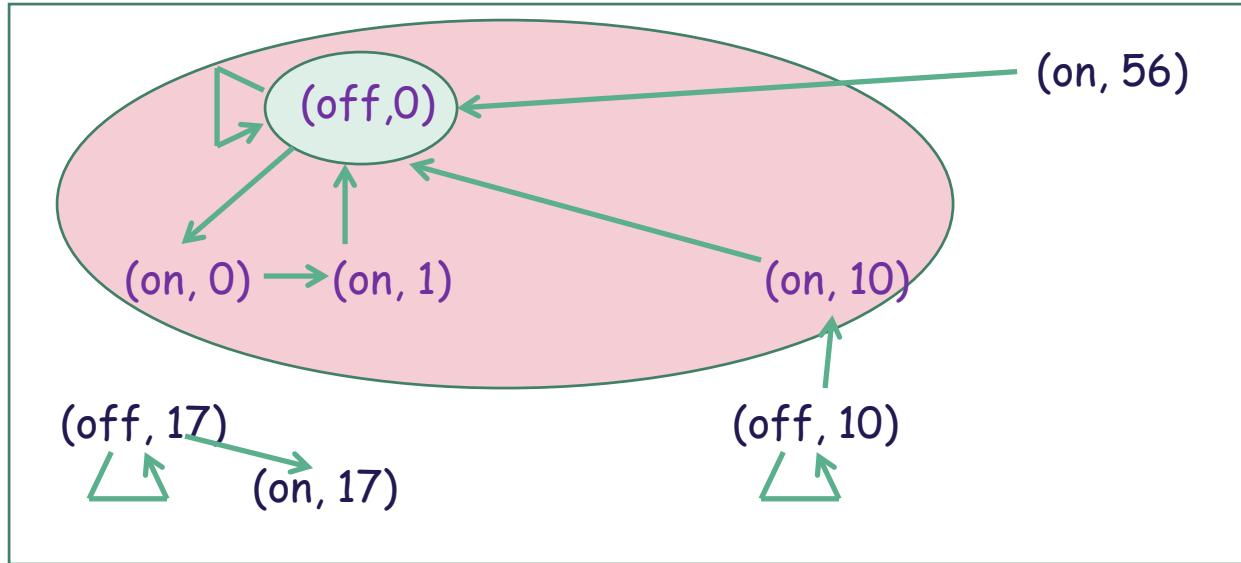


Reachable States of Transition Systems



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: $x \leq 10$; $x \leq 50$; $\text{mode} = \text{off} \rightarrow x = 0$
- Examples of properties that are not invariants: $x < 10$; $\text{mode} = \text{off}$

Invariants for Safety Requirements

- Invariants are safety properties
- 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 (and an execution leading to such a state is a **counterexample**)
- Invariants can express all state-based safety requirements
 - Example for non-state-based requirement: "The button should be pressed at most three times" (unless the number of "button pressed" can be read from the state)
 - We will see later how to deal with general safety requirements

Dynamic vs. Static Analysis

- Dynamic analysis (runtime)
 - Execute the system 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, bugs discovered are real bugs)
 - Static analysis can catch design bugs early
 - Static analysis is often not scalable (solution: analyze approximate versions, which can lead to false warnings)

Invariant Verification

- Simulation
 - Simulate the system multiple times with different inputs
 - Easy to implement, scalable, but no correctness guarantees
- Proof based (Theorem proving)
 - Construct a proof that system satisfies the invariant
 - May require manual effort (partial automation possible)
- State-space analysis (Model checking)
 - Algorithm explores “all” reachable states to check invariants
 - Not scalable in theory, but tools can analyze many real-world designs

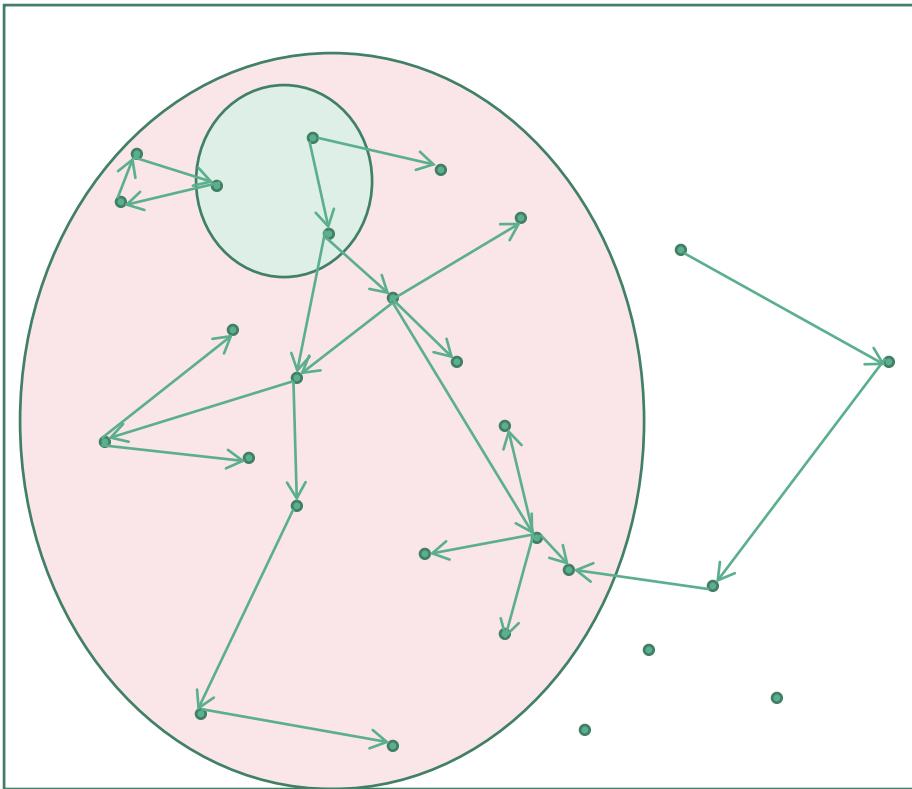
Recap: Inductive Proof Over Natural Numbers

- Show that a statement P holds for all natural numbers n
 - Base case: Prove that P holds for $n = 0$
 - Assume that P holds for some natural number $n = k$
 - Prove (using the assumption) that the statement holds for $n = k+1$
- Benefit: can prove properties over an infinite domain

Structural Induction

- Example structure: binary tree
- Show that a statement P holds at all nodes of the tree
 - Base case: Prove that P holds at the root node
 - Assume that P holds at the nodes up to some level k
 - Prove (using the assumption) that P holds at all nodes of level $k+1$
- Generalization of induction over natural numbers
 - Natural numbers: infinite unary tree
 - Root node: 0
 - Child node of node k : $k+1$
- Example proof: number of leaves is number of inner nodes + 1

Reachable States are Inductive



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**

Proving Invariants

- Inductive definition of reachable states
 - All initial states are reachable using 0 transitions
 - If a state s is reachable in k transitions and (s, t) is a transition, then the state t is reachable in $k+1$ transitions
 - Reachable = Reachable in n transitions, for some n
- Given a transition system T , and a property φ , prove that φ is an invariant, i.e., that all reachable states of T satisfy φ
- Prove: for all n , states reachable in n transitions satisfy φ
 - Base case: Show that all initial states satisfy φ
 - Inductive step: Show that if a (reachable?) state s satisfies φ and (s, t) is a transition, then t must satisfy φ

Inductive Invariant

- A property φ is an **inductive invariant** of transition system T if
 1. Every initial state of T satisfies φ
 2. If a state s satisfies φ and (s, t) is a transition of T , then t must satisfy φ
- Note that we dropped the restriction that s is reachable
- **Theorem:** If φ is an inductive invariant, then all reachable states of T must satisfy φ , and thus, φ is an invariant

Proving Inductive Invariant Example (1)

- Consider transition system T given by
 - State variable int x , initialized to 0
 - Transitions given by "if $(x < m)$ then $x := x+1$ "
- Is the property $\varphi : 0 \leq x \leq m$ an inductive invariant of T ?
- Base case: Consider initial state ($x = 0$). Check that it satisfies φ
- Inductive case:
 - Consider an arbitrary state s , suppose $s(x) = a$
 - Assume that s satisfies φ , i.e., assume $0 \leq a \leq m$
 - Consider the state t obtained by executing the transition from s
 - If $a < m$, then $t(x) = a+1$, otherwise $t(x) = a$
 - In either case, $0 \leq t(x) \leq m$
 - So t satisfies the property φ , and the proof is complete
- Thus φ is an inductive invariant

Proving Inductive Invariant Example (2)

- Consider transition system T given by
 - State variables int x, y ; x is initially 0, y is initially m
 - Transitions given by "if $(x < m)$ then $\{x := x+1; y := y-1\}$ "
- Is the property $\varphi : 0 \leq y \leq m$ an inductive invariant of T ?
- Base case: Consider initial state $(x = 0, y = m)$. Check that it satisfies φ
- Inductive case:
 - Consider an arbitrary state s with $x = a$ and $y = b$
 - Assume that s satisfies φ , i.e., assume $0 \leq b \leq m$
 - Consider the state t obtained by executing the transition from s
 - If $a < m$, then $t(y) = b-1$, otherwise $t(y) = b$
 - Can we conclude that $0 \leq t(y) \leq m$?
 - No! The proof fails! In fact, φ is not an inductive invariant of T !

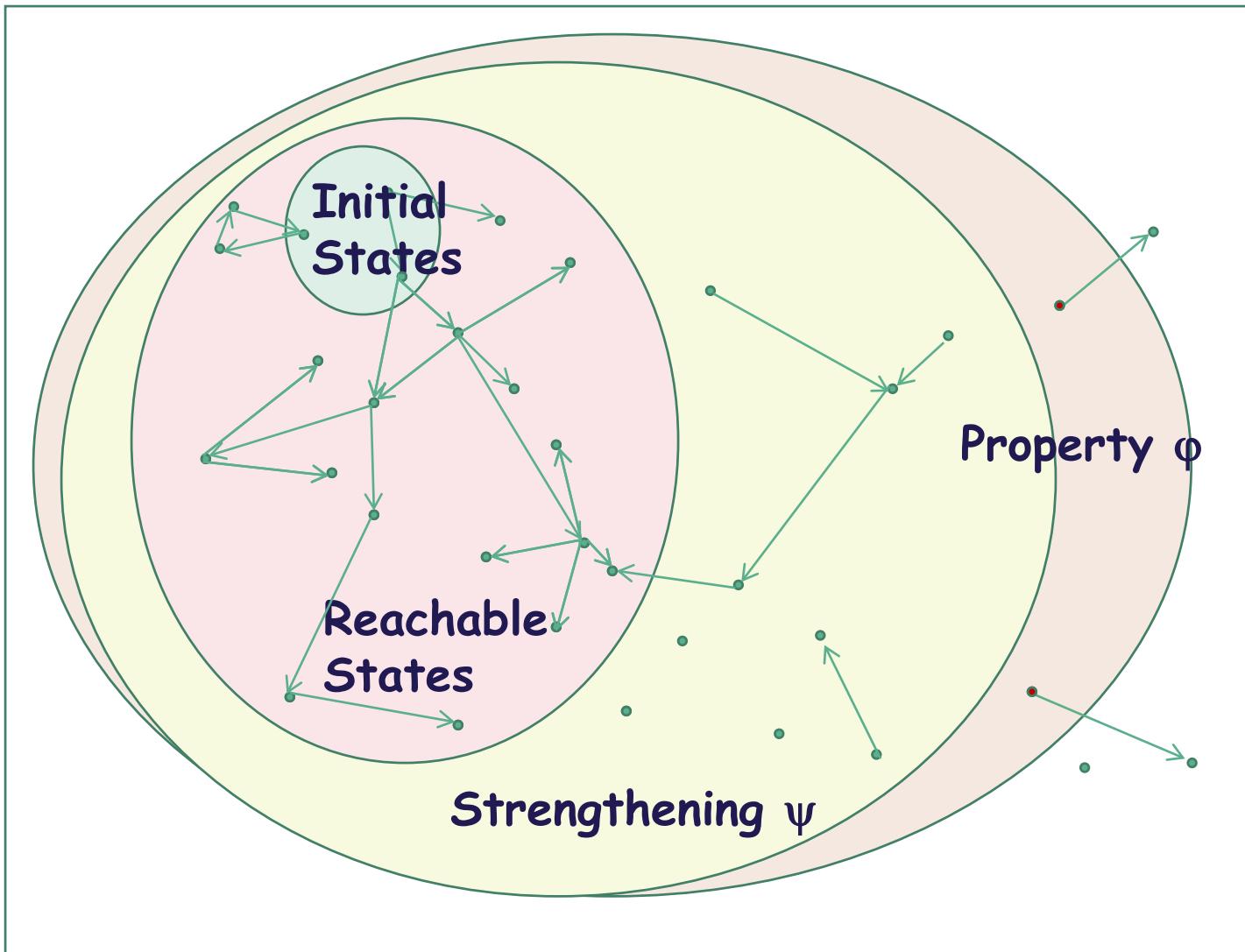
Why did the proof fail?

- Consider the state $s = (x = 0, y = 0)$
 - State s satisfies $\varphi: 0 \leq y \leq m$
 - Executing transition from s leads to state $t = (x = 1, y = -1)$
 - State t does not satisfy φ !
- However, the state s in the above argument is not reachable!
- Cause of failure: Property φ did not capture correlation between variables x and y
- Solution: **Inductive strengthening**
 - Consider property $\psi: (0 \leq y \leq m) \wedge (x+y = m)$
 - Property ψ implies property φ
 - While φ is not an inductive invariant, ψ is (see next slide)
 - It follows that all reachable states must also satisfy φ

Proving Inductive Invariant Example (3)

- Consider transition system T given by
 - State variables int x, y ; x is initially 0, y is initially m
 - Transitions given by "if $(x < m)$ then $\{x := x+1; y := y-1\}$ "
- Property $\psi : (0 \leq y \leq m) \wedge (x+y = m)$
- Base case: Consider initial state $(x = 0, y = m)$. Check that it satisfies ψ
- Inductive case:
 - Consider an arbitrary state s with $x = a$ and $y = b$
 - Assume that s satisfies ψ , i.e., assume $0 \leq b \leq m$ and $a+b = m$
 - Consider the state t obtained by executing a transition from s
 - If $a < m$ then $t(x) = a+1$ and $t(y) = b-1$, otherwise $t(x) = a$ and $t(y) = b$
 - But if $a < m$, since $a+b = m$ holds, $b > 0$, and thus $b-1 \geq 0$
 - In either case, the condition $0 \leq t(y) \leq m \wedge t(x)+t(y) = m$ holds!
- Conclusion: Property ψ is an inductive invariant!

Inductive Invariants



Proof Rule for Proving Invariants (1)

- Recall:
Theorem: If φ is an inductive invariant, then all reachable states of T must satisfy φ , and thus, φ is an invariant
- Say we want to prove that φ is an invariant
If we can prove that φ is an inductive invariant, the theorem implies that φ is an invariant
- However, the converse of the theorem does generally not hold
 - Not every invariant is inductive, so φ may not be inductive
- Good news: If φ is indeed an invariant, there is always an inductive strengthening ψ of φ that is an inductive invariant - why?
 - Pick ψ as "set of reachable states" (which satisfies φ by definition)
- Bad news: The set of reachable states (ψ) cannot generally be computed

Proof Rule for Proving Invariants (2)

- To establish that a property φ is an invariant of transition system T
- Find another property ψ such that
 1. ψ implies φ (that is, a state satisfying ψ must satisfy φ)
 - If you view φ and ψ as sets, then ψ is a subset of φ
 2. ψ is an inductive invariant
 - Show that every initial state satisfies ψ
 - Assume that a state s satisfies ψ . Consider any state t such that (s, t) is a transition. Show that t satisfies ψ

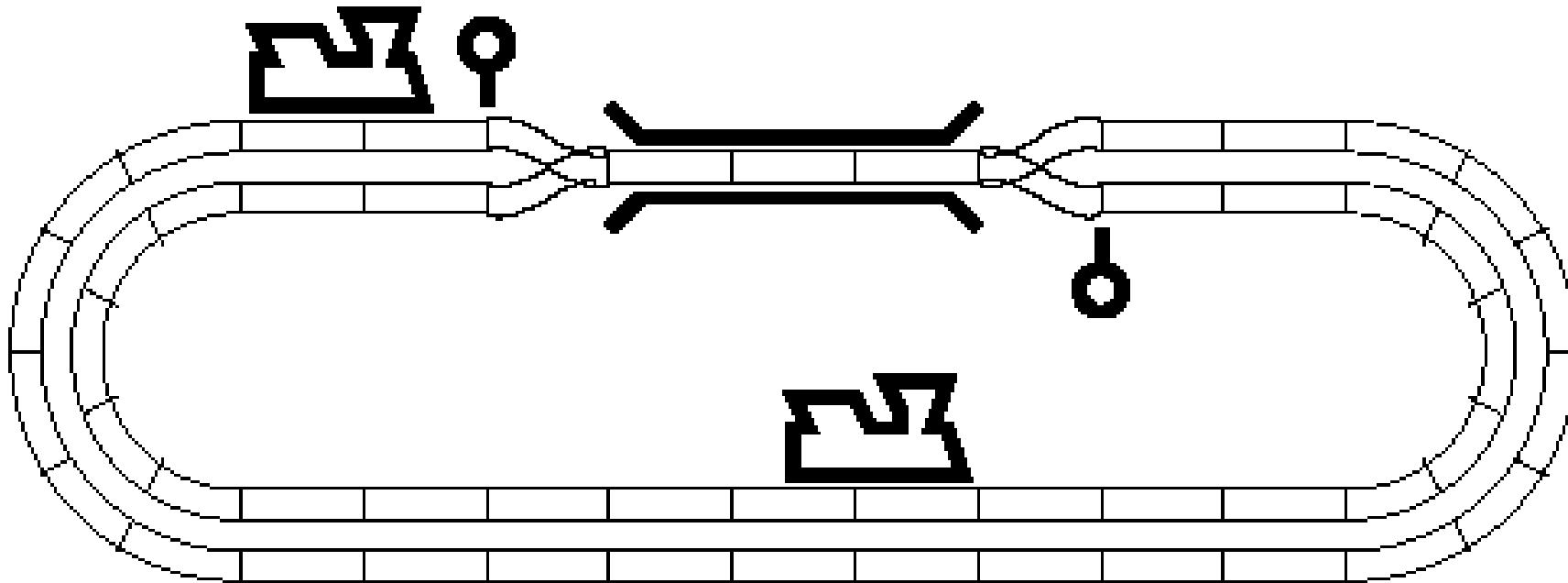
Summary of Invariants

- General-purpose proof technique for proving invariants of programs/models/systems
- Inductive invariant:
 - Must hold in initial states
 - Preserved by every transition/reaction
- To be inductive, a property typically needs to capture relevant relationship among all state variables
- Benefit of finding inductive invariants:
 - Correctness reasoning becomes local (only need to think about what happens in one step)
 - Tools available to check whether a property is an inductive invariant
- Area of active research: can a tool discover them automatically?

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 $C \parallel E$ satisfies the invariant φ

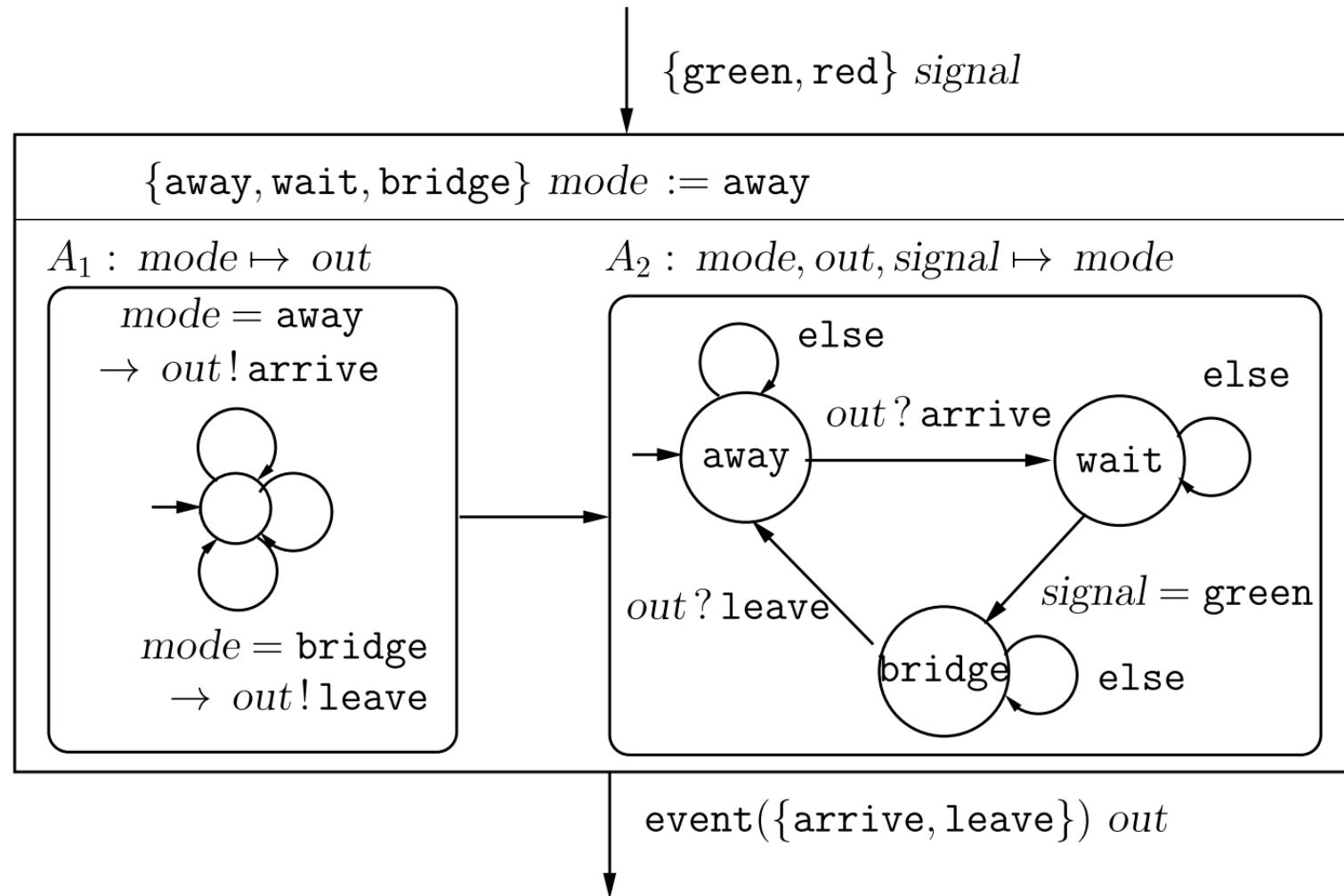
Railroad Controller Example



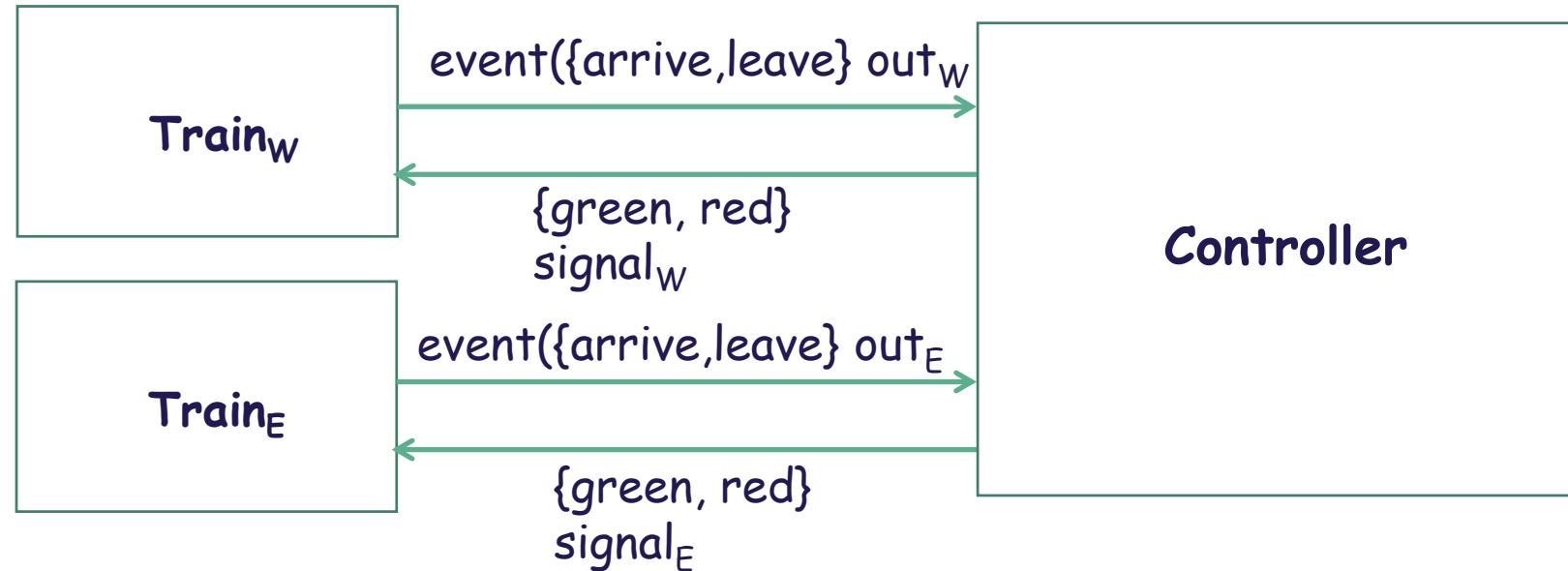
Train Model

- From the perspective of the controller, the train is initially (far) **away**
- The 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 in **wait** state, the train monitors the **signal**:
 - If the **signal** is **green**, the train enters the bridge (state **bridge**)
 - If the **signal** is **red**, the train continues to wait
- A train can stay on the bridge for a duration that is not exactly known (and not influenced by the 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 entry
- Both trains have symmetric behavior

Synchronous Component Train



Controller Design Problem



Safety requirement: Trains are not on the bridge simultaneously

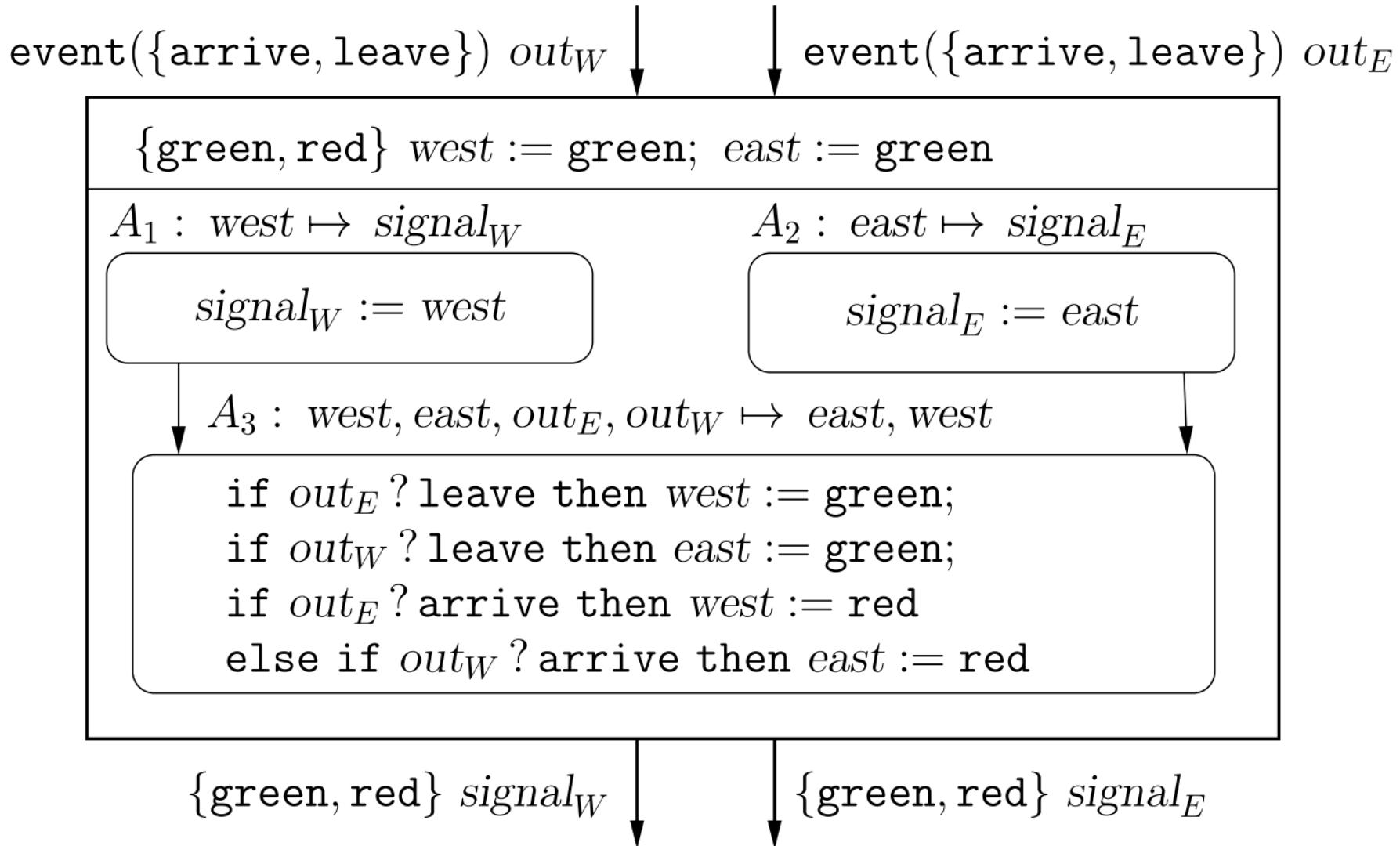
Invariant: $\sim(\text{mode}_W = \text{bridge} \ \& \ \text{mode}_E = \text{bridge})$

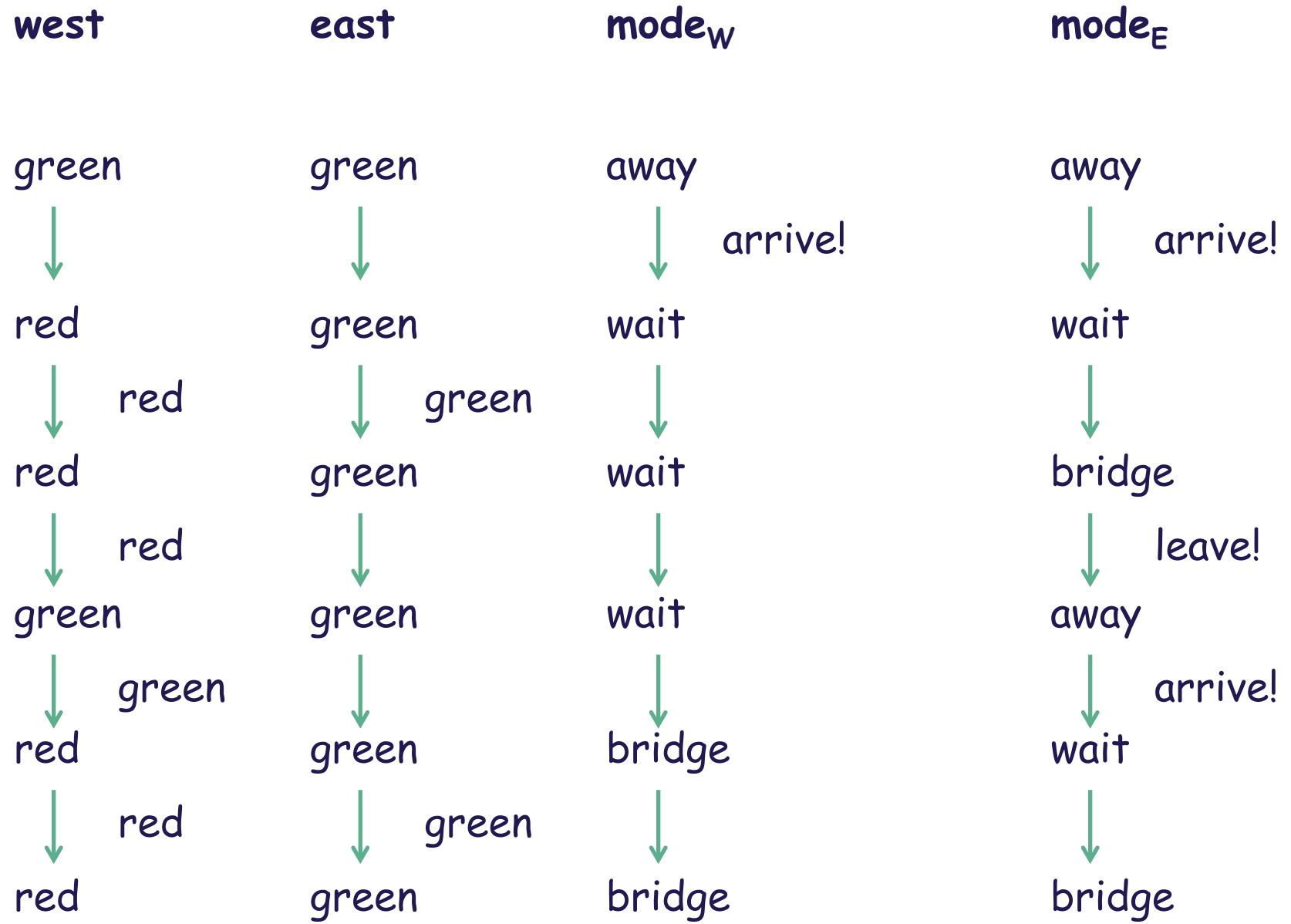
First Attempt at Controller Design

- Controller maintains two state variables to track the state of each signal
- Both state variables are initially green
- Set the output variables based on the corresponding state variables
- If a train arrives, then update the opposite signal variable to red to block the other train from entering
- If a train leaves, reset the opposite signal variable to green

- What happens if both trains arrive simultaneously?
- Tie breaker: Give priority to east train

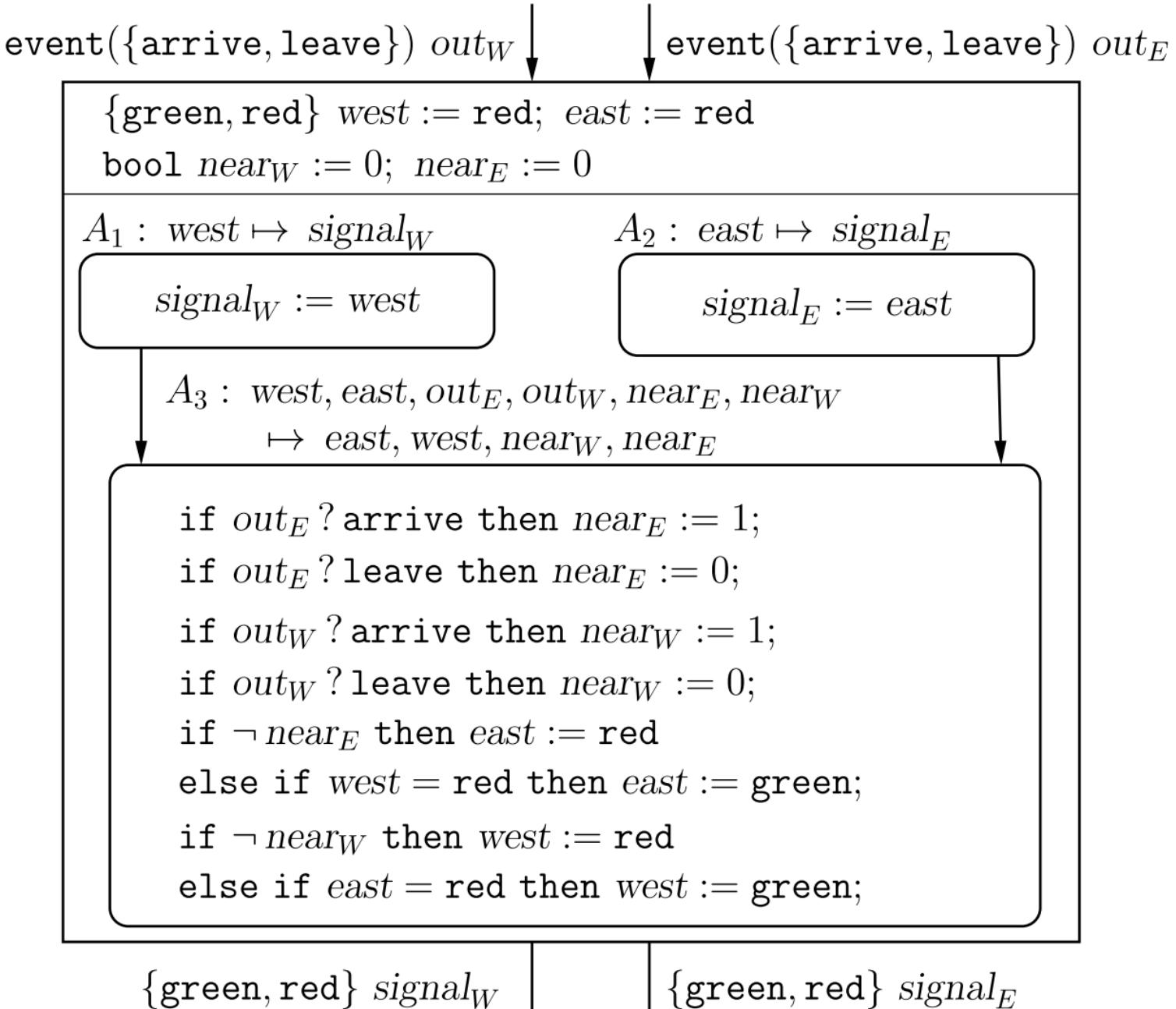
Synchronous Component Controller1





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: $\text{mode}_W = \text{away}$ if and only if $\text{near}_W = 0$
- Variable near_E is symmetric
- We also now keep both signals red by default
- A signal is changed to green if the corresponding train is near and the other signal is red, and changed back to red when the train is away
- Still resolve simultaneous arrivals by preferring one train



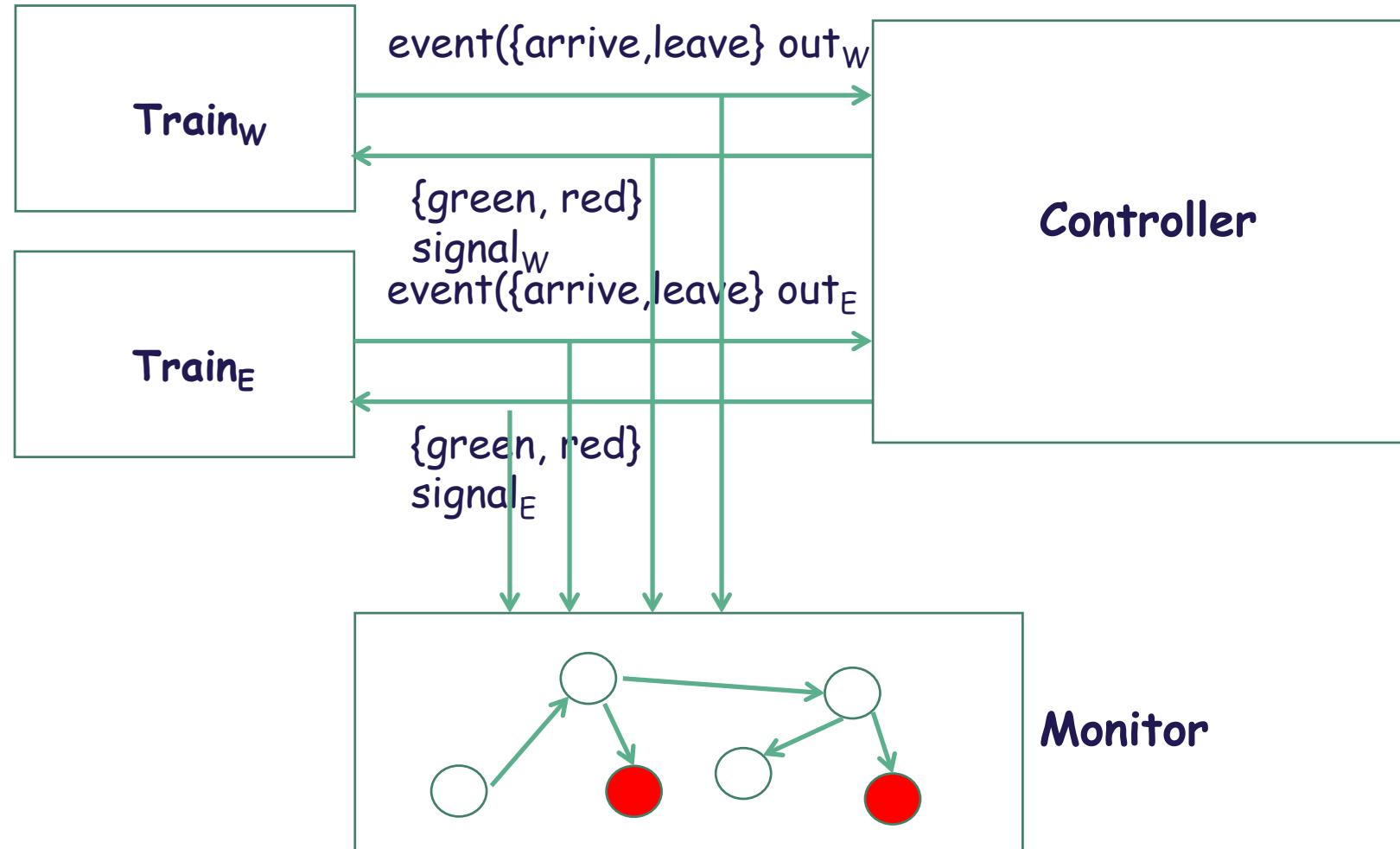
Properties of Controller2

- The system $\text{RailRoadSystem2} = \text{Controller2} \parallel \text{TrainW} \parallel \text{TrainE}$ satisfies the safety requirement (invariant)
- Additional (alternative) requirements we could be interested in
 1. If the west train is waiting, the west signal will eventually become green
 2. If the west train is waiting for its signal to turn green, the other train should not be allowed on the bridge more than once
- Requirement 1 is a “liveness” requirement (see Chapter 4)
- Requirement 2 is a safety property: its violation can be demonstrated by a (finite) execution in which the east train enters, leaves, and enters again while the west train keeps waiting with its signal red
 - Not an invariant (not expressible over state variables only)

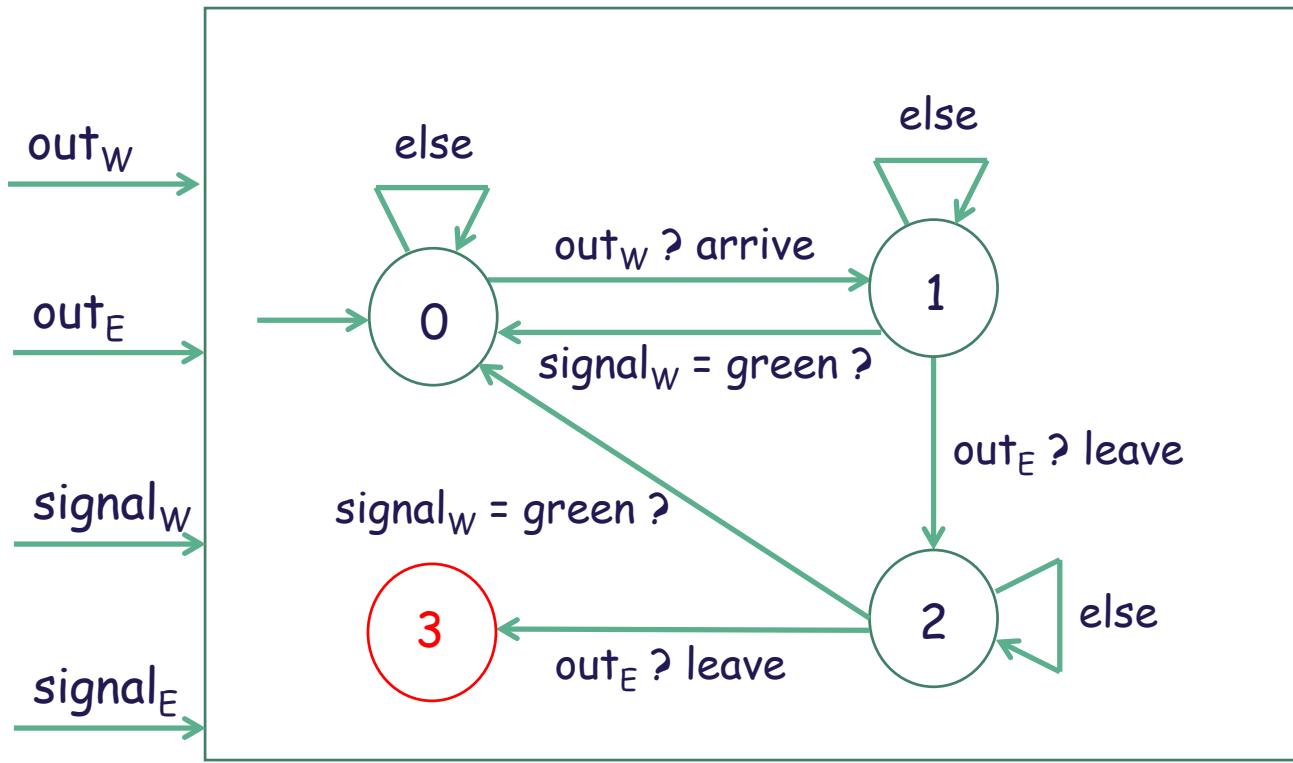
Safety Monitor

- A **monitor** for observes the inputs/outputs of a system and enters an error state if undesirable behavior is detected
- A monitor M is typically specified as extended state machine
 1. The input variables of M are the input/output variables of the system being monitored
 2. An output of M cannot be an input to the system (monitor is not supposed to influence what the system does)
 3. A subset F of the modes of the state machine is accepting ("fail")
- Undesirable behavior: An execution that leads monitor state to F
- Safety verification: Check whether $(\text{monitor.mode} \text{ not in } F)$ is an invariant of $\text{System } C \parallel M$

Safety Monitor



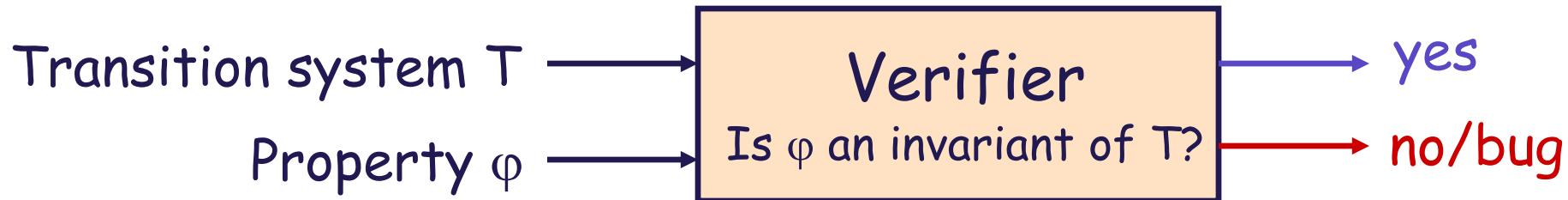
Monitor to Check Train Requirement 2



"If the west train is waiting for its signal to turn green, the other train should not be allowed on the bridge more than once"

Error execution: As west train waits, east train is allowed on bridge twice

Invariant Verification Problem



Can such a verifier exist?

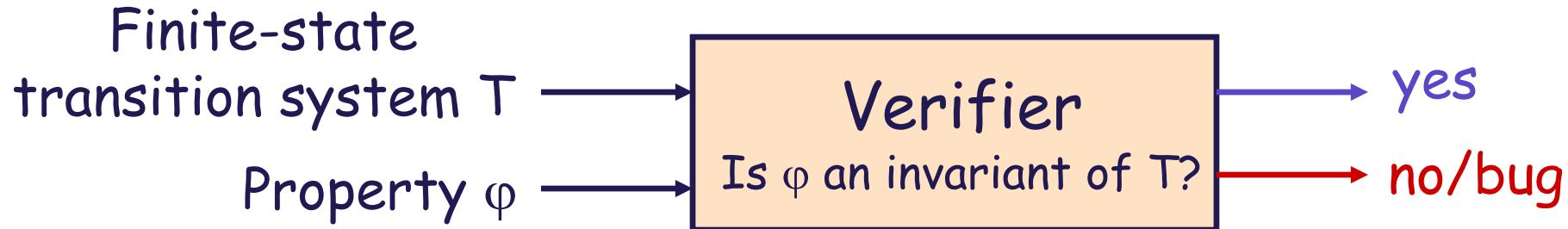
If so, what is the computational complexity of the verification problem?

Theorem: Invariant verification problem is undecidable

If some state variables in T are of type int, then T can correspond to an arbitrary program (or Turing machine)

Intuition: there is no a priori bound on number of reachable states in such case, so examining all reachable states is not possible

Finite-State Invariant Verification Problem



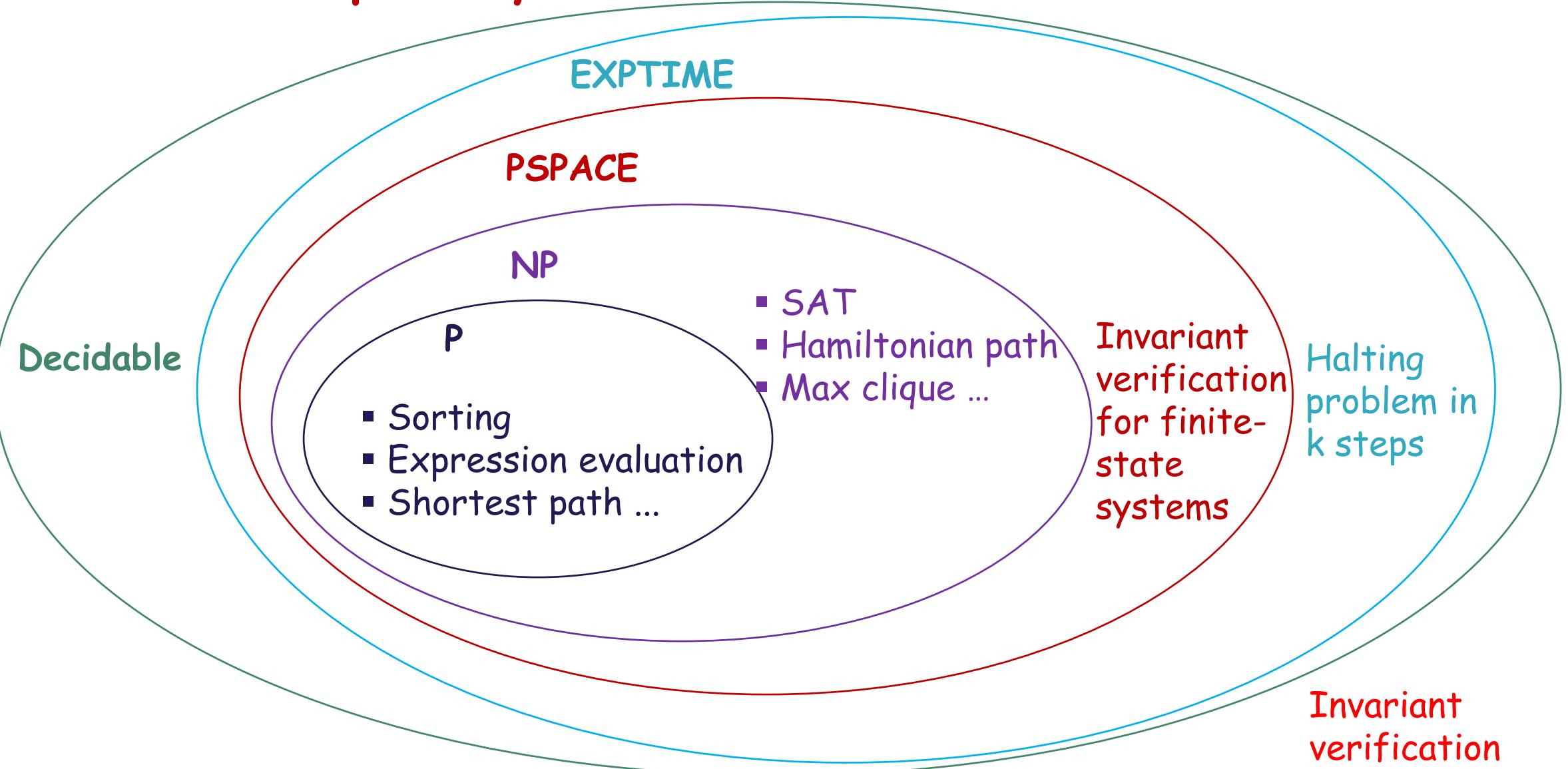
Theorem: Invariant verification problem for finite-state systems is decidable

If T has k Boolean variables, then total number of states is 2^k

Verifier can systematically search through all possible states

Complexity is exponential, more precisely, PSPACE (a class of problems harder than NP problems)

Complexity Classes

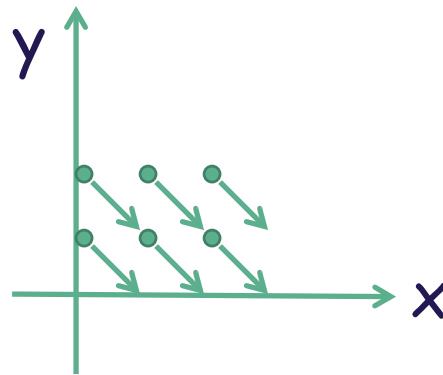


Solving Invariant Verification

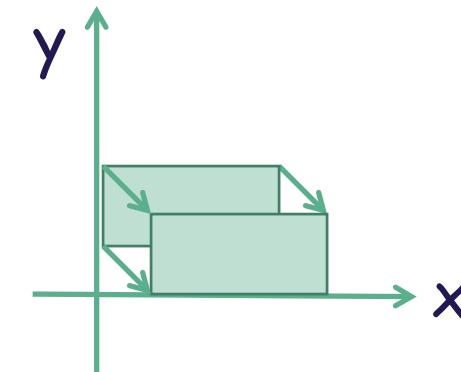
- Establishing that the system is safe is important, but there is no efficient algorithm to solve the verification problem
- Solution 1: Use simulation-based analysis (testing and monitoring)
 - Simulate the model multiple times and check that each state encountered on each execution satisfies the desired invariant
 - Practical in most real-world applications, but no guarantee and known to miss hard-to-find bugs
- Solution 2: Write a formal proof using inductive invariants
 - Only partial tool support, so requires considerable effort
 - Some successes: verified operating systems (e.g., seL4 microkernel) and compilers (e.g., CompCert C compiler)
- Solution 3: Exhaustive search through state-space
 - Fully automated, but has scalability limitations (may not work!)
 - Complementary to simulation, increasingly used in industry
 - On-the-fly enumerative search, **Symbolic search**

Computing Reachable States

- Search algorithm can start with initial states, and explore transitions out of initial states, in a systematic manner
- Example: integer state variables x, y ; initially $0 \leq x \leq 2$ and $1 \leq y \leq 2$, and each transition increments x and decrements y



Enumerative:
Consider individual states



Symbolic:
Consider set of states

Toward Symbolic Search Algorithm

- We need a way to represent and manipulate sets of states
- Basic data structure: **region**
 - For now, a region is given by a formula (= Boolean-valued expression)
 - Example 1: $(x \& y) \mid z$, where x, y, z are Boolean variables
 - Example 2: $x \leq y+1 \& 0 \leq x \leq 3$, where x, y are int/real variables
- Which operations are needed on regions?
 - Will become clear as we develop the algorithm
- "Better" implementation of regions other than formulas?
 - Binary decision diagrams (BDDs) for Boolean variables
 - Intervals or polyhedra (matrices) for real-valued variables
 - More depending on the system dynamics

Symbolic Representation of Transition System

- Consider a transition system T with variables S
- How to represent the set of initial states?
 - Formula φ_I over variables S
 - Easy to obtain from the initialization description Init
 - For our example program from invariant proof, φ_I is $x = 0 \ \& \ y = m$
- How to represent the set of transitions?
 - Convention: Primed variable denotes updated value (i.e., describes the target state of a transition)
 - Transitions are given by a formula φ_T over variables $S \cup S'$
 - Example: In one transition, increment x and decrement y :
 - $x' = x+1 \ \& \ y' = y-1$
 - A pair of states (s, t) over x, y satisfies this formula exactly when $t(x) = s(x)+1$ and $t(y) = s(y)-1$
- If T has k variables, then a set of states is given by a region over k variables, and transitions are given by a region over $2k$ variables

Transition Formula

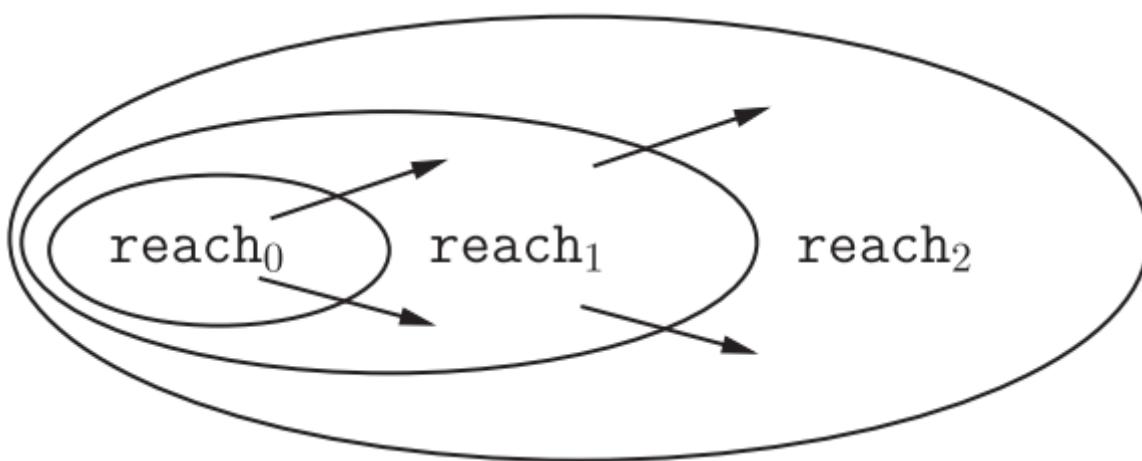
- Suppose a single step is given by:
$$\text{if } (x > 0 \ \& \ y > 0) \text{ then } \{ \text{if } (x > y) \text{ then } x := x - y \text{ else } y := y - x \}$$
- Find a formula φ_T over variables x, y, x', y' to capture this
- Assignment $x := x - y$ corresponds to constraint $x' = x - y$
- But this is not enough. If y is not assigned explicitly, it stays unchanged. In formulas, we must say $y' = y$ to enforce this
- Assignment $x := x - y$ corresponds to formula $x' = x - y \ \& \ y' = y$
- Assignment $y := y - x$ corresponds to formula $x' = x \ \& \ y' = y - x$
- Conditional statement $\text{if } (x > y) \text{ then } x := x - y \text{ else } y := y - x$ becomes
$$\phi: [x > y \ \& \ x' = x - y \ \& \ y' = y] \mid [\sim(x > y) \ \& \ x' = x \ \& \ y' = y - x]$$
- Desired transition formula φ_T for the entire statement is
$$[x > 0 \ \& \ y > 0 \ \& \ \phi] \mid [\sim(x > 0 \ \& \ y > 0) \ \& \ x' = x \ \& \ y' = y]$$

Symbolic Transition System

- Region over variables X is a data structure that represents a set of states assigning values to X
- Transition system $T = (S, \text{Init}, \text{Trans})$ can be represented by
 - Region φ_I over S for initial states
 - Region φ_T over $S \cup S'$ for transitions
- Symbolic representation can be compiled automatically from code
 - To get φ_T from reaction description of a Synchronous Reactive Component, local/input/output variables must be existentially quantified (see textbook for examples)

Image Computation

- Given a region A , define $\text{Post}(A, \varphi_T)$ to be set of successors of states in A
$$\text{Post}(A, \varphi_T) = \{ t \mid \text{there exists a state } s \text{ in } A \text{ and a transition } (s, t) \}$$
- Core problem in symbolic search: Given A and the transition formula, how to compute $\text{Post}(A, \varphi_T)$?
- If we can implement this operation, the set of all reachable states can be enumerated iteratively by breadth-first search



$\text{reach}_0 = \text{initial states}$

each reach_{i+1} obtained from
 reach_i by applying Post

Image Computation: Example 1

- Suppose T has a single variable x of type real
- Consider transition formula $\varphi_T: x' = 2x+1$
- How to compute $\text{Post}(A, \varphi_T)$ systematically, where A given by $0 \leq x \leq 10$
- Step 1: Conjoin (i.e., intersect) A with φ_T
 - Result: $0 \leq x \leq 10 \ \& \ x' = 2x+1$
 - Intuition: this represents all transitions (x, x') starting in region A
- Step 2: Existentially quantify x
 - Result: $\exists x. 0 \leq x \leq 10 \ \& \ x' = 2x+1$
 - Intuition: formula involving only x' , for which there exists some x such that x is in A and (x, x') is a transition
 - Result simplifies to $0 \leq (x'-1)/2 \leq 10$, which is equivalent to $1 \leq x' \leq 21$
- Step 3: We want $\text{Post}(A)$ to be a formula involving x , so rename x' to x
 - Result: $1 \leq x \leq 21$
 - Check that $[1, 21]$ is exactly the image of $[0, 10]$ if x goes to $2x+1$

Image Computation: Example 2

- Suppose T has variables x and y of type real
- Consider transition formula $\varphi_T: x' = x+1 \ \& \ y' = x$
- Suppose A is given by $0 \leq x \leq 4 \ \& \ y \leq 7$

- Step 1: Conjoin (i.e., intersect) A with φ_T
 - Result: $0 \leq x \leq 4 \ \& \ y \leq 7 \ \& \ x' = x+1 \ \& \ y' = x$

- Step 2: Existentially quantify x and y (= projection) and simplify
 - Let us first project out x : $0 \leq y' \leq 4 \ \& \ y \leq 7 \ \& \ x' = y'+1$
 - Now project out y : $0 \leq y' \leq 4 \ \& \ x' = y'+1$

- Step 3: Rename x' to x and y' to y
 - Result: $0 \leq y \leq 4 \ \& \ x = y+1$

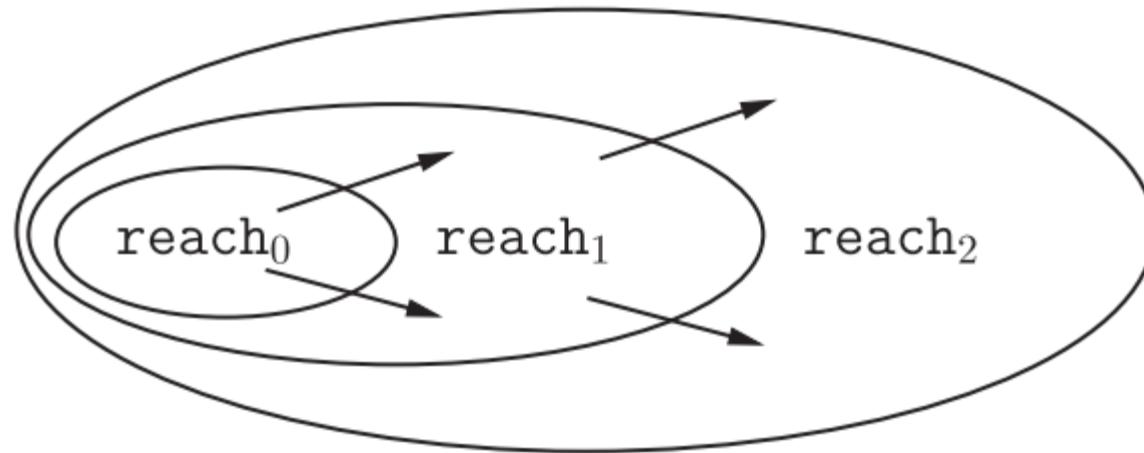
Operations on Regions

- In general, we want to represent sets of states by a data type **reg**, which should support the following operations
- $\text{Disj}(A,B)$: Returns region that contains states in A or in B
 - For formulas, this is just " $A \mid B$ "
- $\text{Conj}(A,B)$: Returns region containing states that are in both A and B
 - For formulas, this is just " $A \& B$ "
- $\text{Diff}(A,B)$: Returns region containing states in A but not in B
 - For formulas, this is " $A \& \sim B$ "
- $\text{IsEmpty}(A)$: Returns 0 if region A contains some state, and 1 otherwise
 - For formulas, this requires testing "satisfiability": can the variables be assigned values to make the formula true
- $\text{Exists}(A,X)$: Returns projection of A by quantifying variables in X
 - For formulas, this requires "quantifier elimination"
- $\text{Rename}(A,X,Y)$: Rename variables in X to corresponding variables in Y
 - For formulas, this is textual substitution

Symbolic Image Computation

- Given:
 - Region A of type reg over state variables S
 - Region φ_T of type reg over $S \cup S'$
- $\text{Post}(A, \varphi_T) = \text{Rename}(\text{Exists}(\text{Conj}(A, \varphi_T), S), S', S)$
 1. Take conjunction of A and φ_T
 2. Project out the variables in S using existential quantification
 - Rename primed variables to get a region over S

Symbolic Breadth-First-Search Algorithm



reach_0 = initial states

each reach_{i+1} obtained from
 reach_i by applying Post

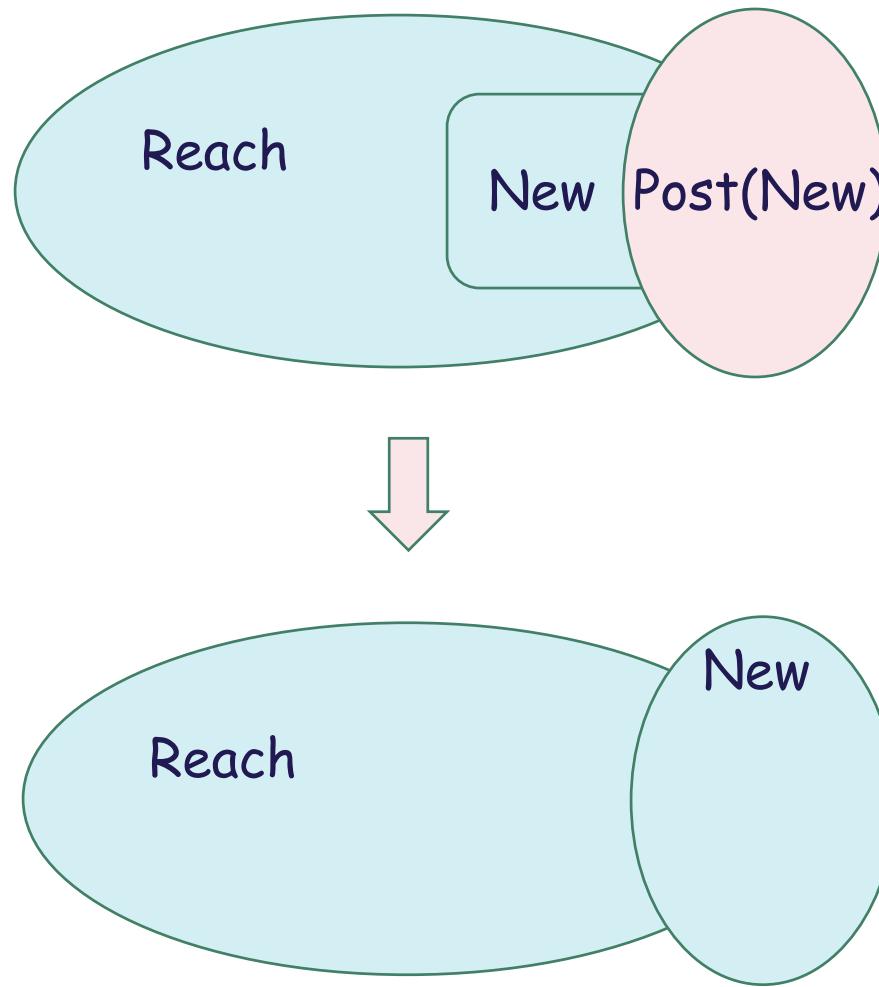
- Algorithm for checking whether a property φ is an invariant of T
- Same as checking if the “error” states $\sim\varphi$ are reachable
- We need to check at every step if error states are reached; if so, stop
- If no new states are encountered, then also stop (invariant satisfied)

Symbolic BFS Algorithm

Given region φ_I over S , region φ_T over $S \cup S'$, and region φ over S ,
if φ is reachable in T then return 1, else return 0

```
reg Reach :=  $\varphi_I$ ; /* states found to be reachable */  
reg New :=  $\varphi_I$ ; /* states not yet explored for outgoing transitions */  
while IsEmpty(New) = 0 { /* while there are states to be explored */  
    if IsEmpty(Conj(New,  $\varphi$ )) = 0 /* property  $\varphi$  found reachable */  
        then return 1;  
    New := Diff(Post(New,  $\varphi_T$ ), Reach);  
    /* These are states in the post-image of New, but not  
       previously found reachable, so they have to be explored */  
    Reach := Disj(Reach, New); /* update Reach by newly found states */  
};  
return 0; /* all states explored without encountering  $\varphi$  */
```

Frontier Computation in Symbolic BFS



Symbolic Search

- Correctness: When the algorithm stops, its answer (whether the property φ is reachable or not) is correct
- Termination: Number of iterations depends on
 - length of shortest execution leading to a state satisfying φ
 - Diameter: smallest d such that all states reachable within d steps (this may not be bounded if the system is not finite-state)
 - In practice, terminates if one of these numbers is small
- Used in practice for hardware verification and protocol verification
 - Industrial-strength symbolic model checkers (e.g., Cadence)
 - Open-source widely used academic tool: NuSMV

Implementation of Regions

- Key to efficient implementation: How to represent regions?
 - Operations: Disj, Conj, Diff, IsEmpty, Exists, Rename
- Suppose all variables are Booleans
- We can represent regions with formulas (with $\&$, $|$, \sim)
 - Disj, Conj, Diff, Rename easy
 - $\text{Exists}(\varphi, x)$ same as $\varphi [x \rightarrow 0] | \varphi [x \rightarrow 1]$
 - $\text{IsEmpty}(\varphi)$ requires test for satisfiability (SAT)
- SAT is computationally demanding (NP-complete), but more importantly, size of formula representing Reach keeps growing as we apply operations
 - Key to performance: Simplify formulas as much as possible
 - Solution: (reduced ordered) binary decision diagrams ((RO)BDDs)

Models and Tools for Cyber-Physical Systems

Chapter 4: Asynchronous Model

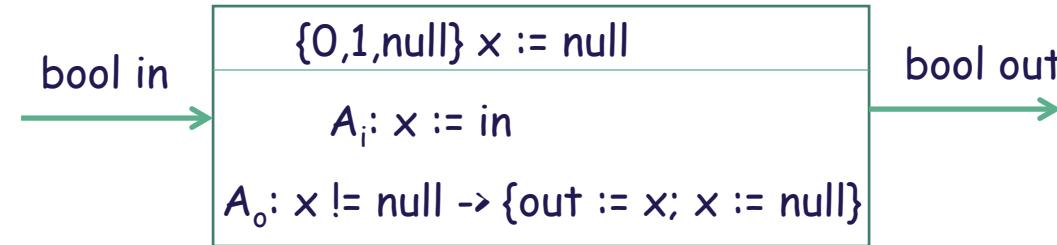
Instructors: Martijn Goorden, Kim G. Larsen,
Christian Schilling, Max Tschaikowski
`{mgoorden,kgl,christianms,tschaikowski}@cs.aau.dk`

Slides courtesy of Rajeev Alur
`alur@cis.upenn.edu`

Asynchronous Models

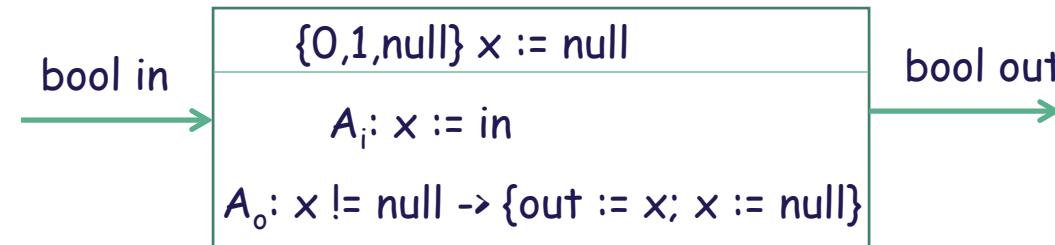
- Recap: In a synchronous model, all components execute in a sequence of (logical) rounds in lock-step
- Asynchronous model: Speed at which different components execute are independent (or unknown)
 - Processes in a distributed system
 - Threads in a typical operating system
- Key design challenge: how to achieve coordination?

Example: Asynchronous Buffer



- Input channel: in of type Boolean
- Output channel: out of type Boolean
- State variable: x , can be null or hold 0/1 value
- Initialization of state variables: assignment $x := \text{null}$
- Input task A_i for processing of inputs: code: $x := \text{in}$
- Output task A_o for producing outputs:
guard: $x \neq \text{null}$; code: $\text{out} := x; x := \text{null}$

Example: Asynchronous Buffer

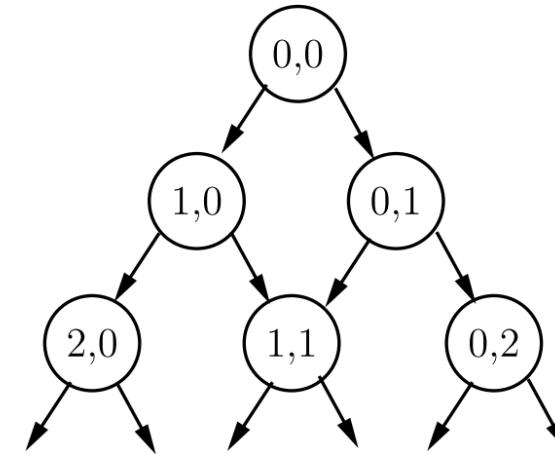


- Execution model: In one step, only a single task is executed
 - Processing of inputs (by input tasks) is decoupled from production of outputs (by output tasks)
- A task can be executed if it is enabled, i.e., its guard condition holds
 - If multiple tasks are enabled, only one of them is executed
- Sample execution:



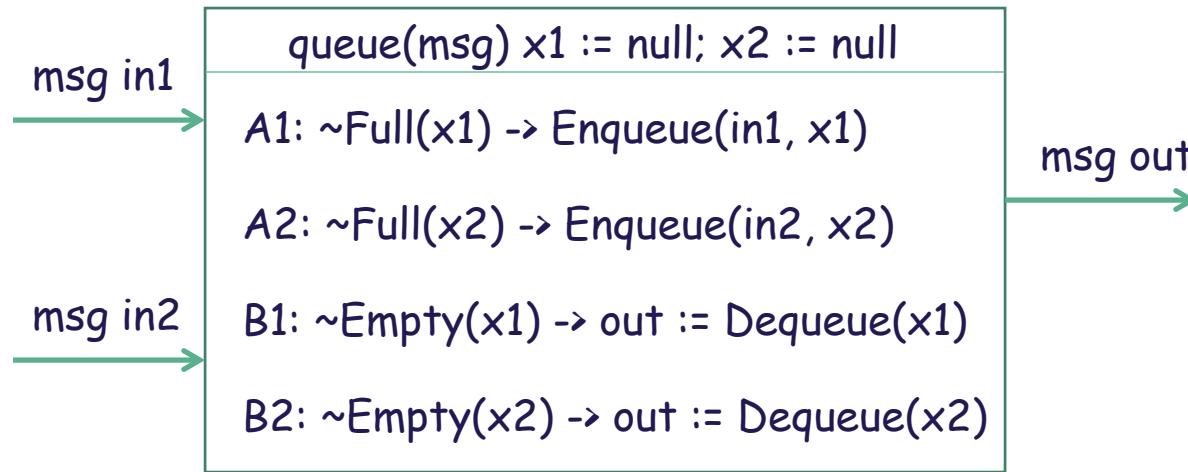
Example: Asynchronous Increments

nat $x := 0; y := 0$
$A_x: x := x + 1$
$A_y: y := y + 1$



- Internal task: Does not involve input or output channels
 - Can have guard condition
 - Execution of internal task: Internal action
- In each step, execute either task A_x or task A_y
- Sample execution:
 $(0,0) - \varepsilon \rightarrow (1,0) - \varepsilon \rightarrow (1,1) - \varepsilon \rightarrow (1,2) - \varepsilon \rightarrow (1,3) - \varepsilon \rightarrow \dots$
 $- \varepsilon \rightarrow (1,105) - \varepsilon \rightarrow (2,105) \dots$
- For all natural numbers m, n , the state (m, n) is reachable
- Interleaving semantics

Asynchronous Merge



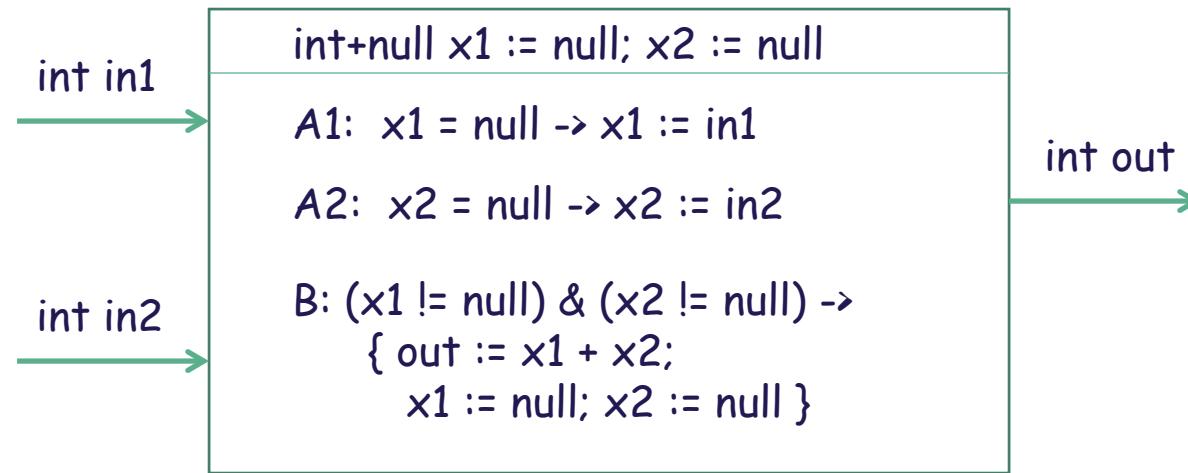
Sequence of messages on output channel is an arbitrary merge
of sequences of values on the two input channels

At every step, exactly one of the four tasks executes
(provided its guard condition holds)

Sample execution:

$([],[]) - \text{in1?}5 \rightarrow ([5],[]) - \text{in2?}0 \rightarrow ([5],[0]) - \text{out!}0 \rightarrow ([5],[])$
 $- \text{in1?}6 \rightarrow ([5,6],[]) - \text{in2?}3 \rightarrow ([5,6],[3]) - \text{out!}5 \rightarrow ([6],[3]) \dots$

What does this process do?



Definition: Asynchronous Process (1)

- Set I of (typed) input channels
 - Defines the set of inputs of the form $x?v$, where x is an input channel and v is a value
- Set O of (typed) output channels
 - Defines the set of outputs of the form $y!v$, where y is an output channel and v is a value
- Set S of (typed) state variables
 - Defines the set of states Q_s
- Initialization Init
 - Defines the set [Init] of initial states

Definition: Asynchronous Process (2)

- Set of input tasks; each such task is associated with an input channel x
 - Guard condition over state variables S
 - Update code from read-set $S \cup \{x\}$ to write-set S
 - Defines a set of input actions of the form $s - x?v \rightarrow t$
- Set of output tasks; each task is associated with an output channel y
 - Guard condition over state variables S
 - Update code from read-set S to write-set $S \cup \{y\}$
 - Defines a set of output actions of the form $s - y!v \rightarrow t$
- Set of internal tasks
 - Guard condition over state variables S
 - Update code from read-set S to write-set S
 - Defines a set of internal actions of the form $s - \varepsilon \rightarrow t$

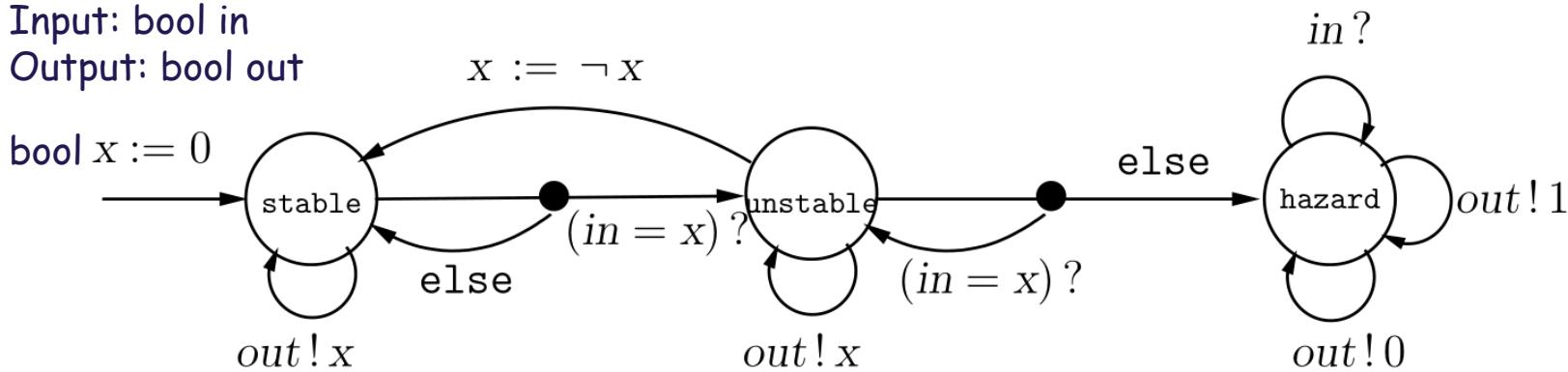
Asynchronous Gates



- Why design asynchronous circuits?
 - Input can be changed even before the effect propagates through the entire circuit
 - Can be faster than synchronous circuits, but design is more complex
- Asynchronous NOT gate
 - When input changes, gate enters *unstable state* until it gets a chance to update the output value
 - If input changes again in unstable state, this causes a hazard where behavior is unpredictable

Asynchronous NOT Gate as Extended State Machine

Input: bool in
Output: bool out



Each mode-switch/transition corresponds to a task

Example input task: (mode = stable) \rightarrow if ($in = x$) then mode := unstable

Example output task: (mode = stable) \rightarrow out := x

Example internal task: (mode = unstable) \rightarrow { $x := \sim x$; mode := stable}

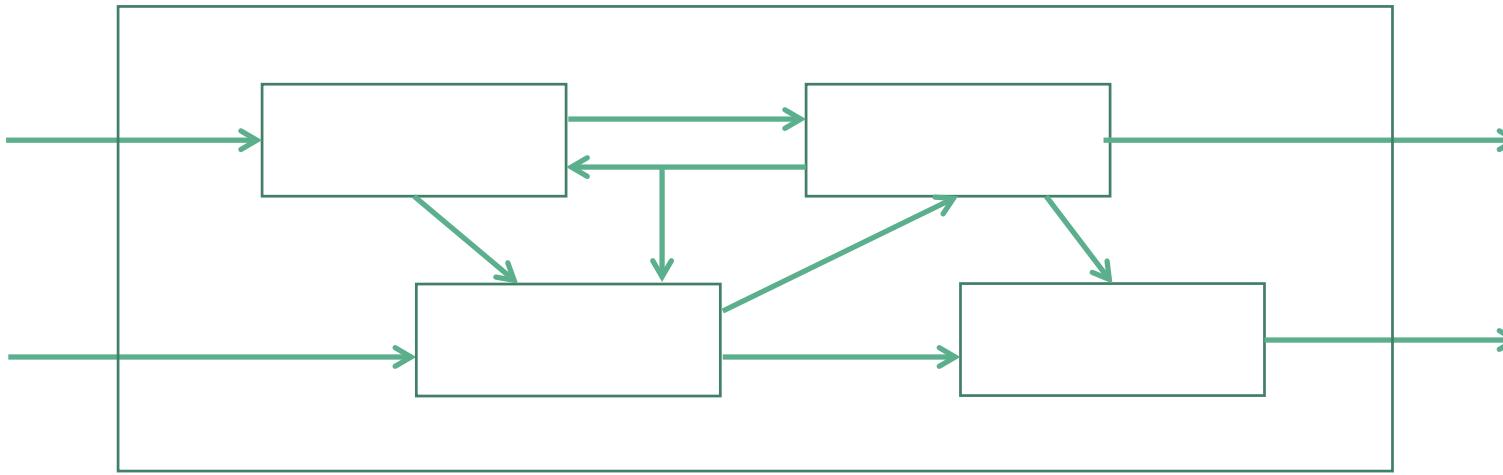
Sample execution:

$(stable,0) - in?1 \rightarrow (stable,0) - out!0 \rightarrow (stable,0) - in?0 \rightarrow (unstable,0)$
 $- \varepsilon \rightarrow (stable,1) - out!1 \rightarrow (stable,1) - in?1 \rightarrow (unstable,1) - out!1 \rightarrow (unstable,1)$
 $- in?0 \rightarrow (hazard,1) - out!0 \rightarrow (hazard,1) - out!1 \rightarrow (hazard,1) \dots$

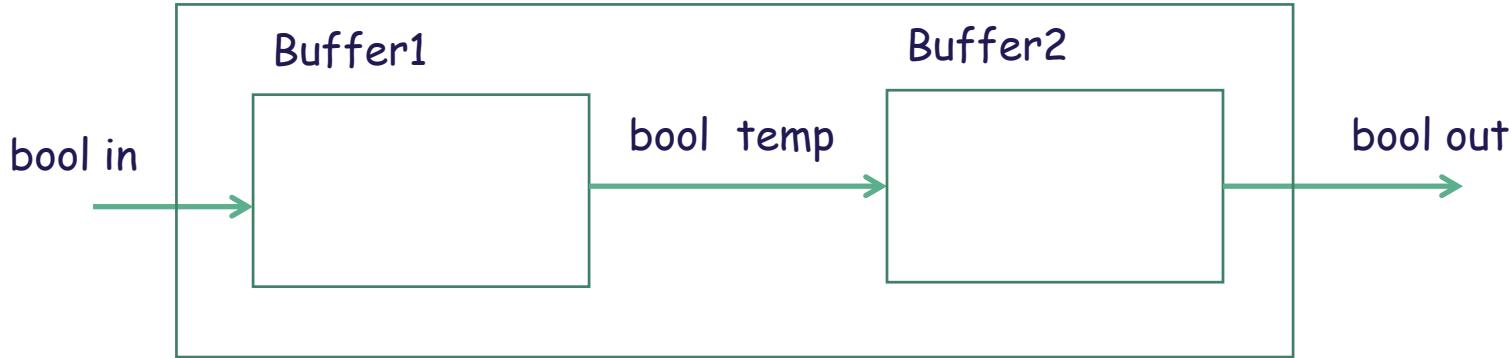
How can an environment ensure that the gate does not enter the hazard mode?

When the input toggles, wait to observe a change in the value of the output

Block Diagrams



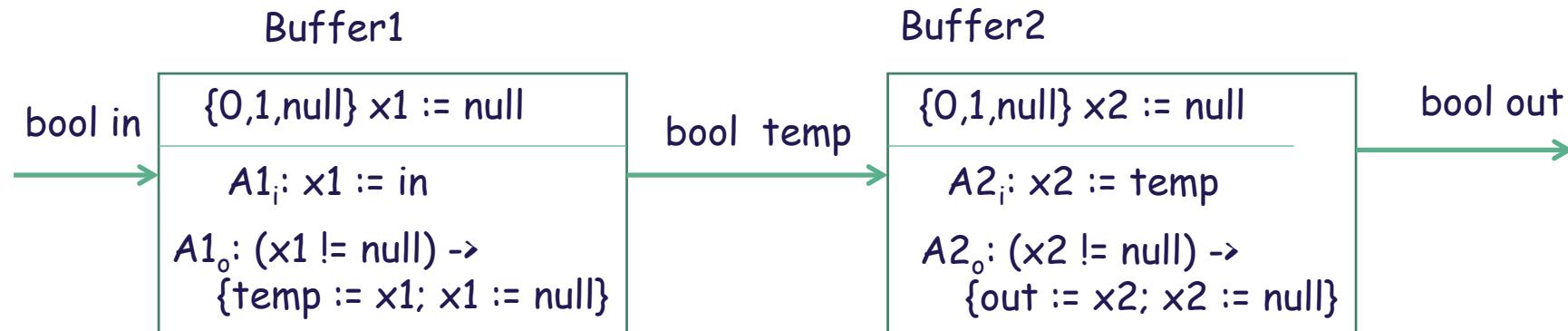
DoubleBuffer



(Buffer[out -> temp] | Buffer[in -> temp]) \ temp

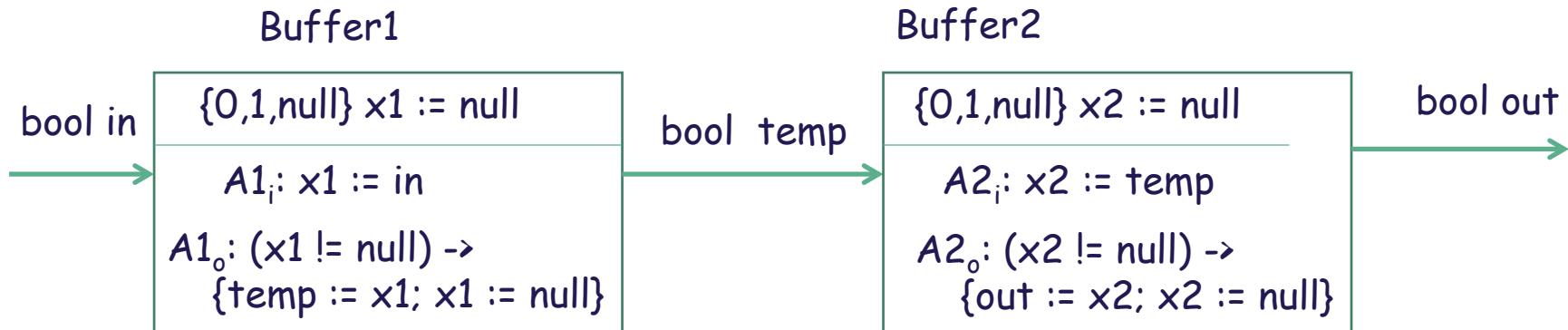
- Instantiation: Create two instances of Buffer
 - Output of Buffer1 = Input of Buffer2 = variable temp
- Parallel composition: Asynchronous concurrent execution of Buffer1 and Buffer2 (note: new syntax A | B)
- Hide variable temp: Encapsulation (temp becomes local)

Composing Buffer1 and Buffer2

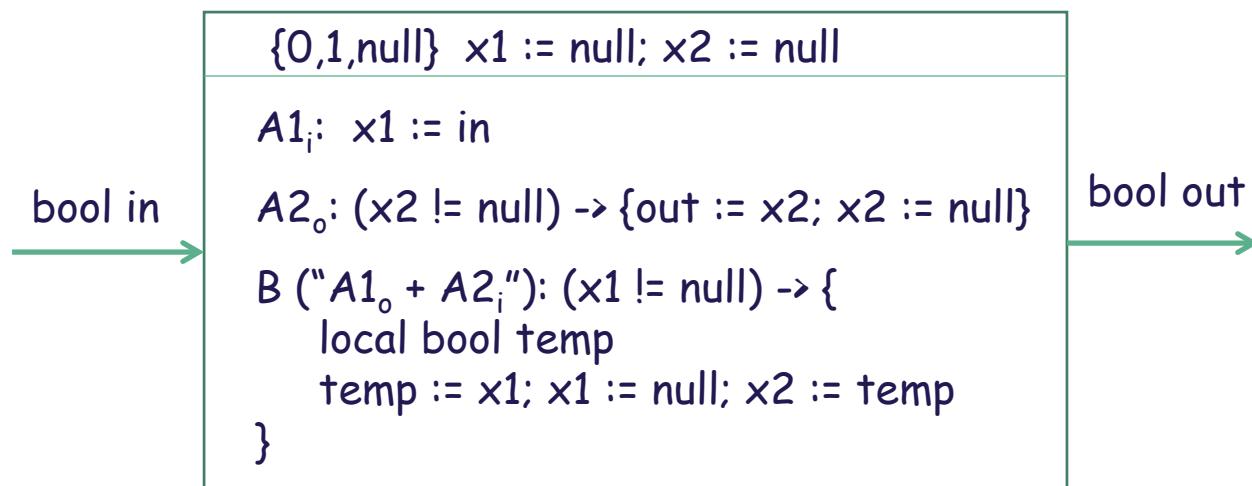


- Inputs, outputs, states, and initialization for composition obtained in the same manner as in synchronous case
- What are the tasks of the composition?
 - Production of output on channel temp by Buffer1 synchronized with consumption of input on channel temp by Buffer2

Compiled DoubleBuffer



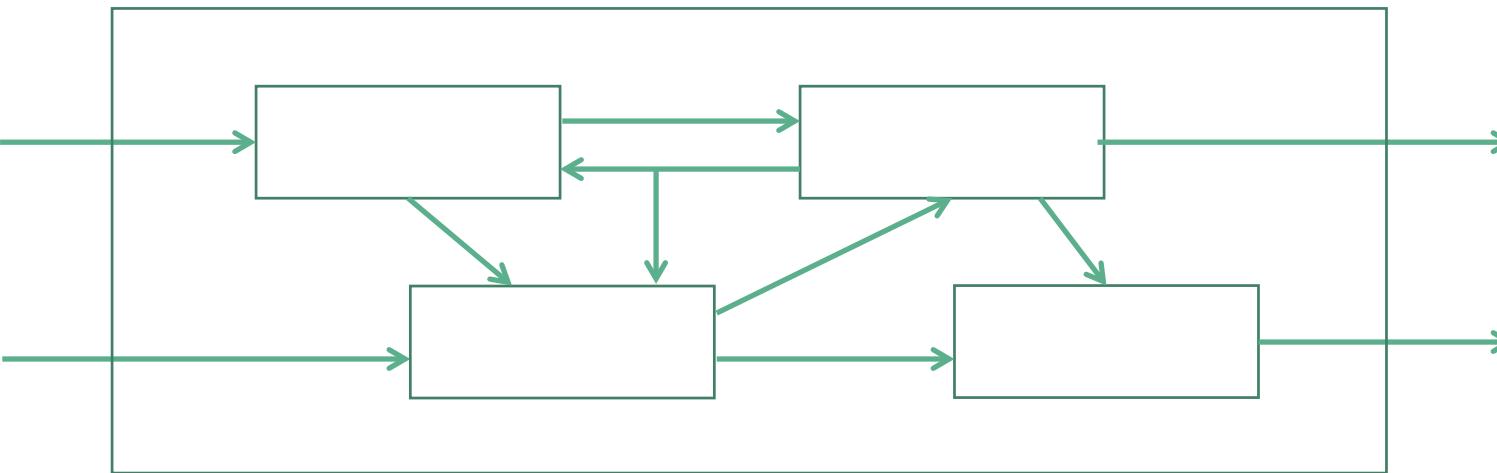
(Buffer[out -> temp] | Buffer[in -> temp]) \ temp



Definition of Asynchronous Composition

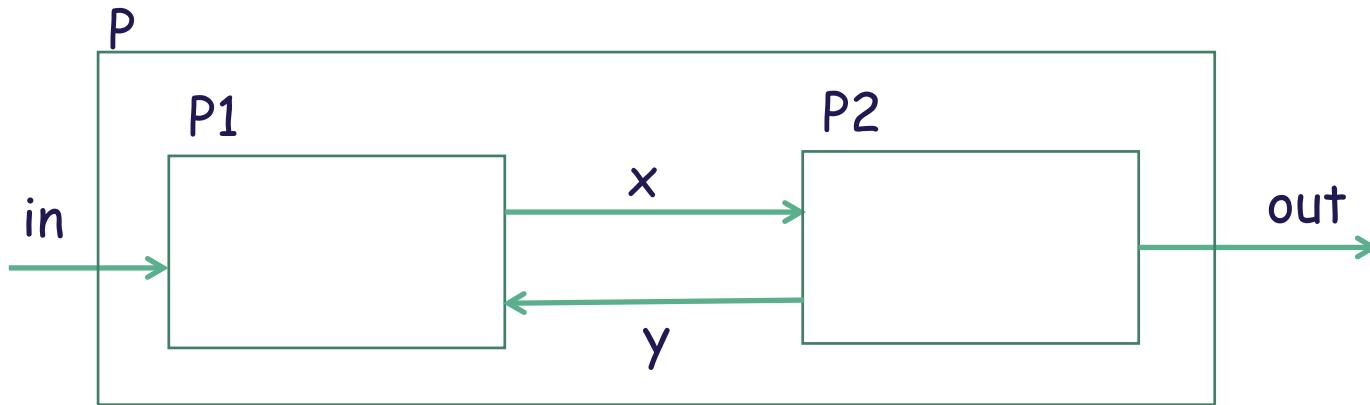
- Given asynchronous processes P1 and P2, how to define $P1 \mid P2$?
- Note: In each step of execution, only one task is executed
 - Concepts such as await-dependencies are not relevant
- Sample case (see textbook for complete definition):
 - If y is an output channel of P1 and input channel of P2, and
 - $A1$ is an output task of P1 for y with code: $\text{Guard1} \rightarrow \text{Update1}$, and
 - $A2$ is an input task of P2 for y with code: $\text{Guard2} \rightarrow \text{Update2}$, then
 - Composition has an output task for y with code:
 $(\text{Guard1} \& \text{Guard2}) \rightarrow \text{Update1}; \text{Update2}$

Execution Model: Another View



- A single step of execution
 - Execute an internal task of one of the processes
 - Process input on an external channel x : Execute an input task for x of every process to which x is an input
 - Execute an output task for an output y of some process, followed by an input task for y for every process to which y is an input
- If multiple enabled choices, choose one non-deterministically
 - No constraint on relative execution speeds

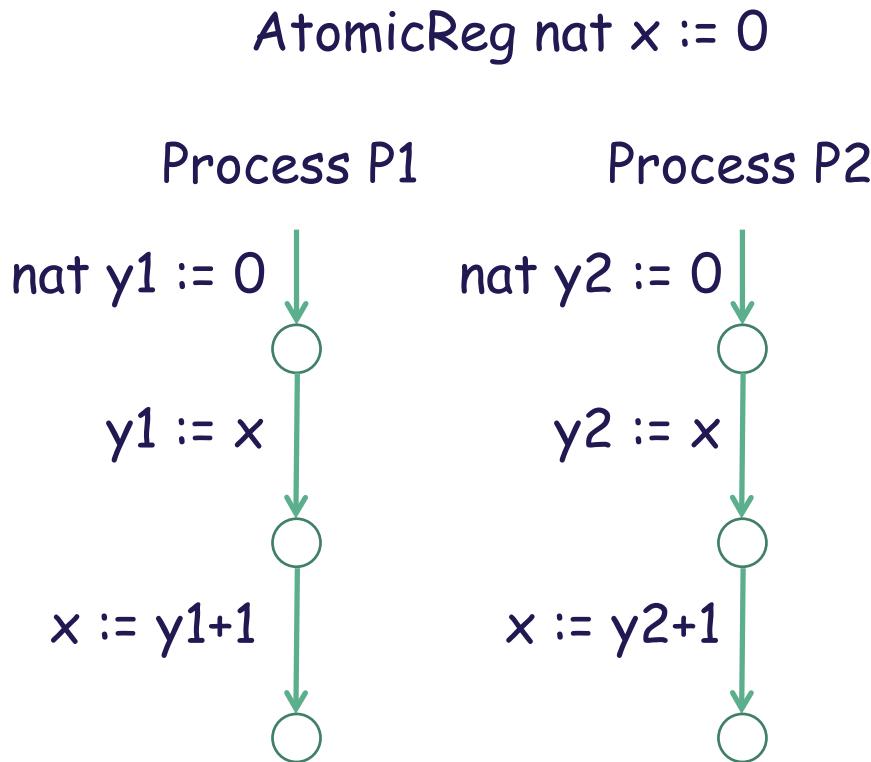
Asynchronous Execution



What can happen in a single step of this asynchronous model P ?

- $P1$ synchronizes with the environment to accept input on in
- $P2$ synchronizes with the environment to send output on out
- $P1$ performs some internal computation (one of its internal tasks)
- $P2$ performs some internal computation (one of its internal tasks)
- $P1$ produces output on channel x , followed by its consumption by $P2$
- $P2$ produces output on channel y , followed by its consumption by $P1$

Shared Memory Programs



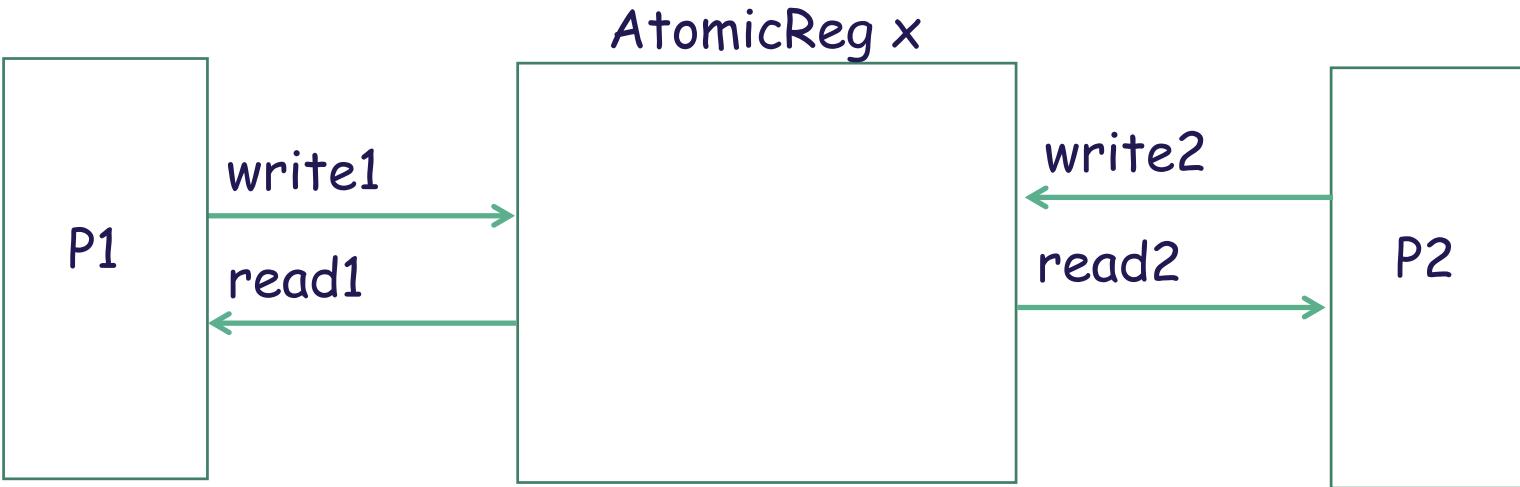
Declaration of shared variables
+ code for each process

Key restriction: Each statement of a process either
changes local variables,
reads a single shared variable, or
writes a single shared variable

Execution model: execute one step of
one of the processes

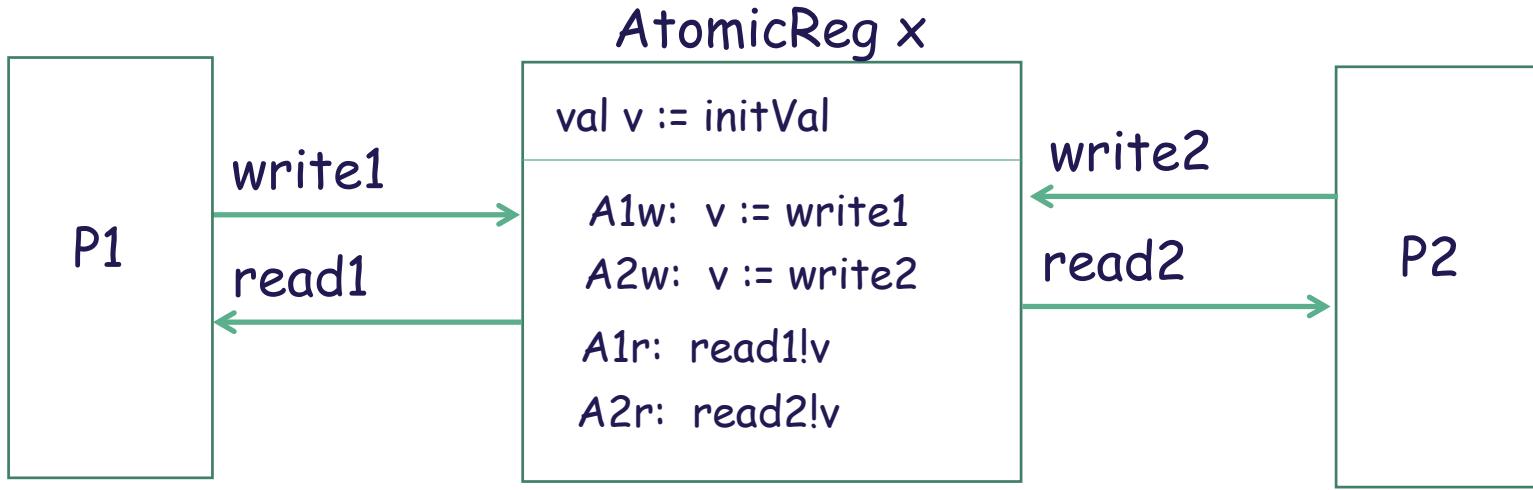
Can be formalized as asynchronous
processes

Shared Memory Processes



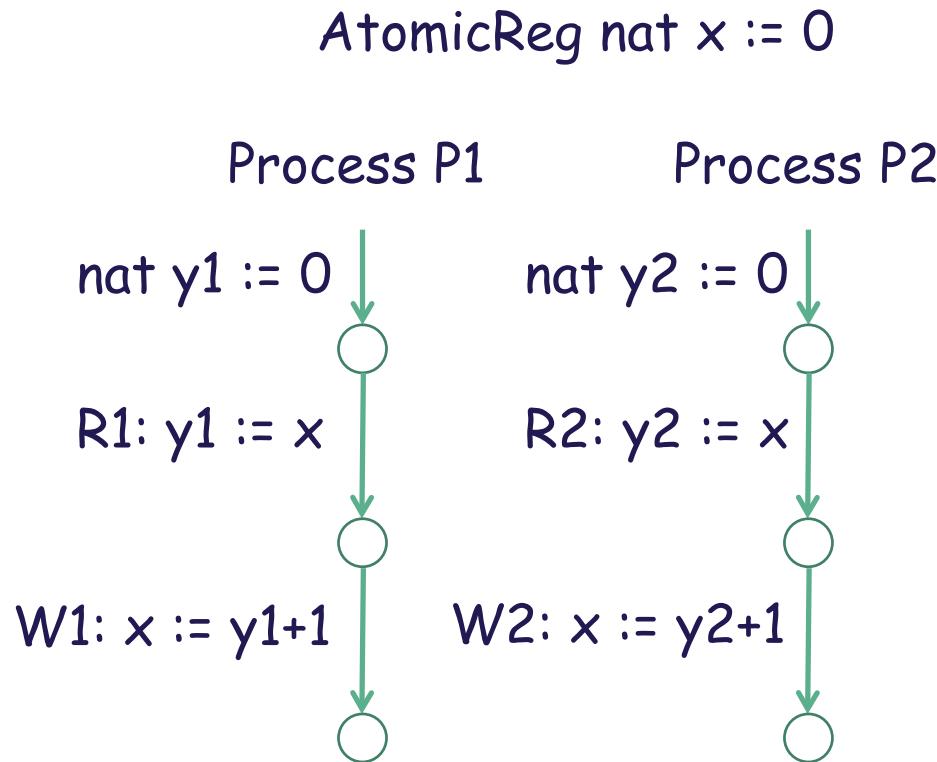
- Processes P1 and P2 communicate by reading/writing shared variables
- Each shared variable can be modeled as an asynchronous process
 - State of each such process is the value of the corresponding variable
 - In implementation, shared memory can be a separate subsystem
- Read and write channel between each process and each shared variable
 - To write x, P1 synchronizes with x on "write1" channel
 - To read x, P2 synchronizes with x on "read2" channel

Atomic Registers



- By definition of our asynchronous model, each step above is either internal to P1 or P2, or involves exactly one synchronization: either read or write one shared variable by one of the processes
- Atomic register: Basic primitives are read and write
 - To “increment” such a register, a process first needs to read and then write back the incremented value
 - But these two are separate steps, and the register value can be changed in between by another process

Data Races



What are the possible values of x after all steps are executed?

x can be 1 or 2

Possible executions:

R1, R2, W1, W2

R1, W1, R2, W2

R1, R2, W2, W1

R2, R1, W1, W2,

R2, W2, R1, W1

R2, R1, W2, W1

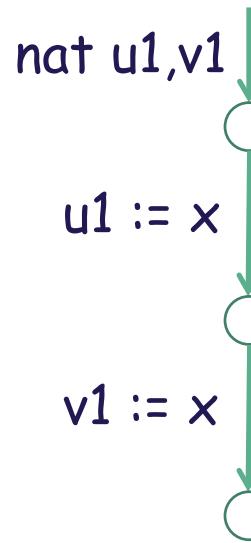
Data race: Concurrent accesses to a shared object where the result depends on the order of execution

This should be avoided!

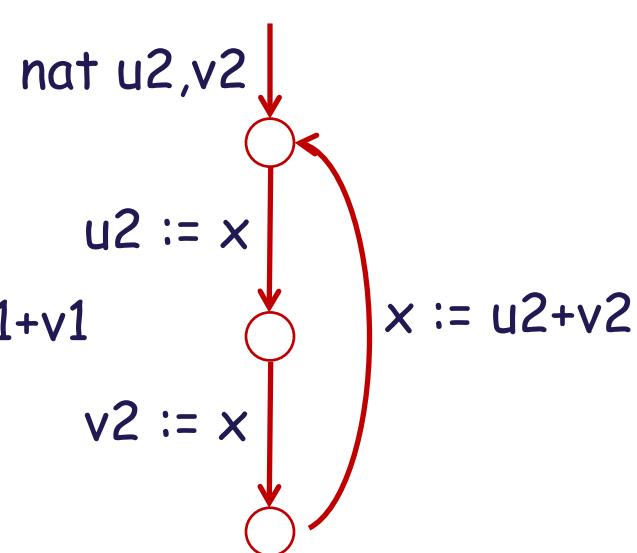
Puzzle

AtomicReg nat $x := 1$

Process P1

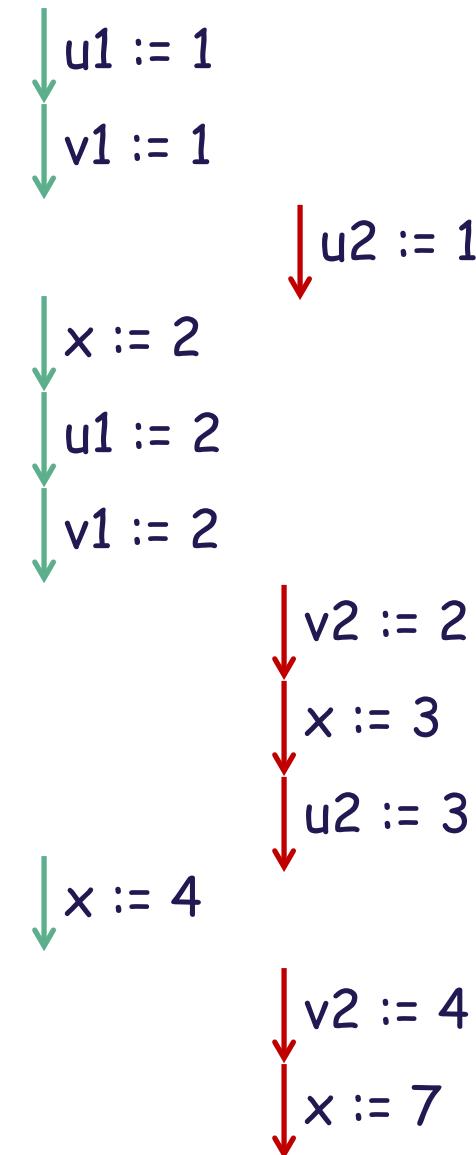


Process P2



Which values can x have?

Every number $> 0!$



Mutual Exclusion Problem

Process P1

Entry code

Critical section

} To be designed

Process P2

Entry code

Critical section

- Critical section: Part of code that an asynchronous process should execute without interference from others
 - Critical section can include code to update shared objects/database
- Mutual exclusion problem: Design code to be executed before entering critical section by each process
 - Coordination using shared atomic registers
 - No assumption about how long a process stays in critical section
 - A process may want to enter critical section repeatedly

Mutual Exclusion Problem

Process P1

Entry code

Critical section

To be designed

Process P2

Entry code

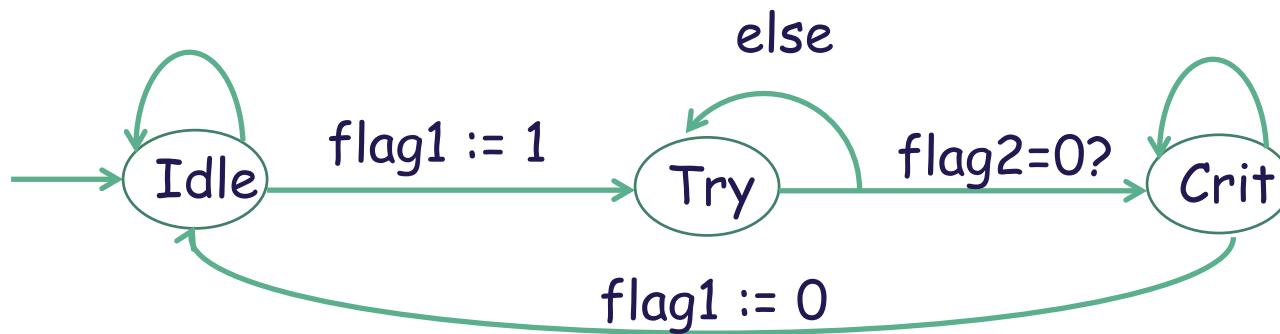
Critical section

- Safety requirement: Both processes should not be in critical section simultaneously (can be formalized as an invariant)
- Liveness requirement: Starvation freedom: If a process is trying to enter, then eventually it should be able to enter

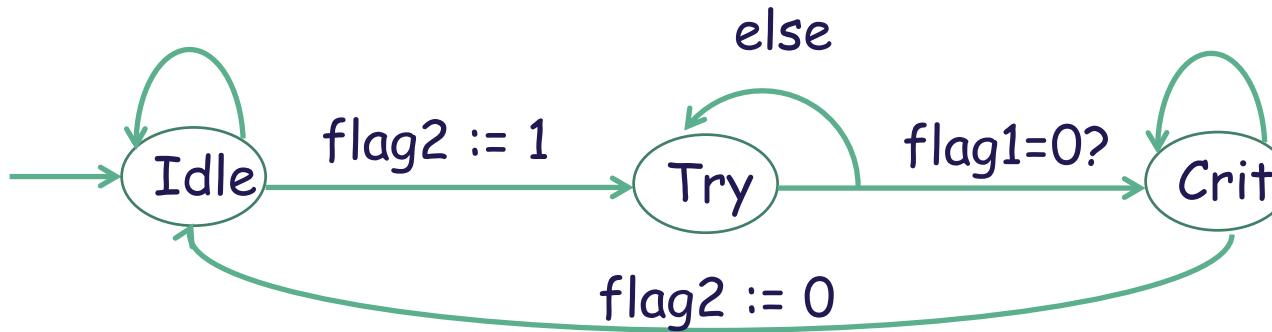
Mutual Exclusion: First Attempt

AtomicReg bool flag1 := 0; flag2 := 0

Process P1



Process P2

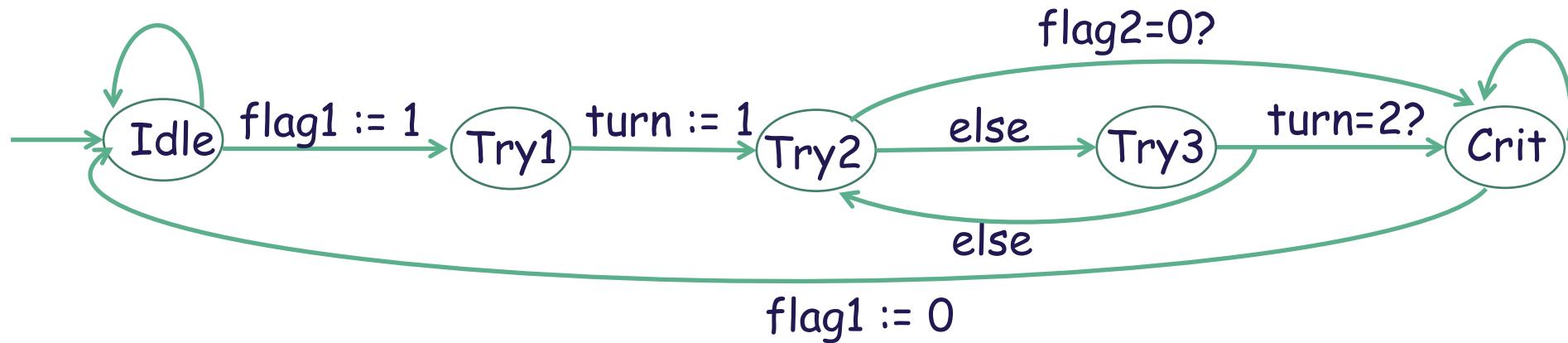


Not correct!
(see Uppaal model)

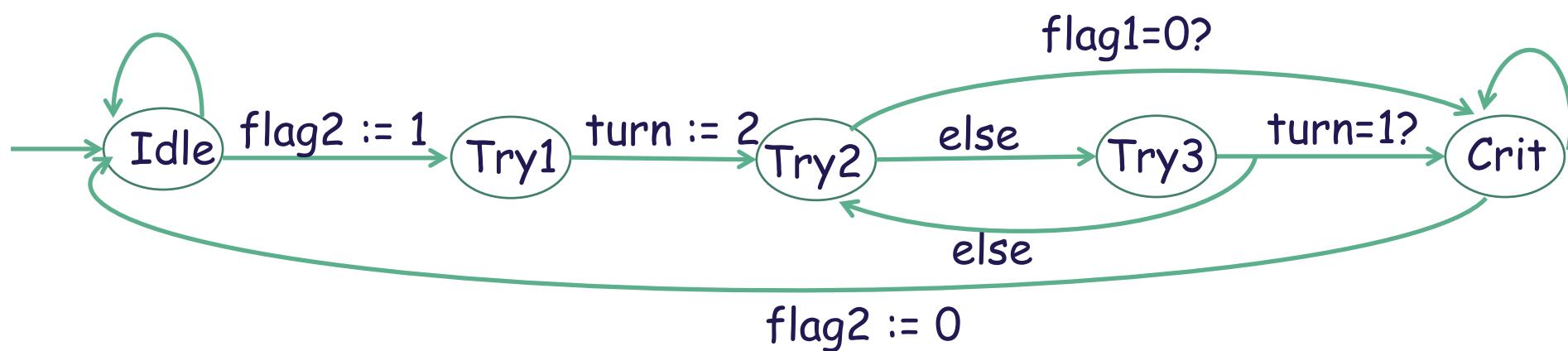
Peterson's Mutual Exclusion Protocol

AtomicReg bool flag1 := 0; flag2 := 0; {1,2} turn

Process P1

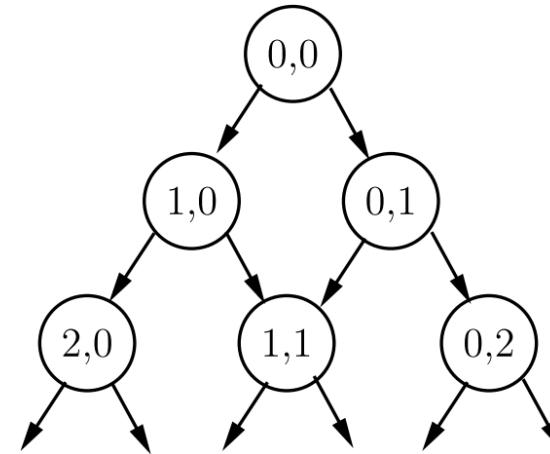


Process P2



Another Look at Asynchronous Execution Model

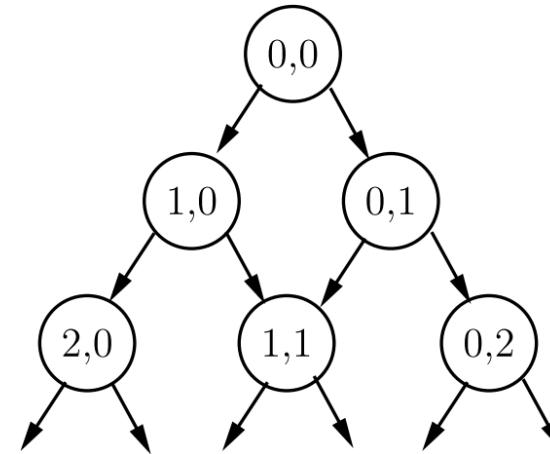
nat $x := 0; y := 0$
$A_x: x := x + 1$
$A_y: y := y + 1$



- Tasks A_x and A_y execute in an arbitrary order
 - Motivation: If we establish that all possible executions of this asynchronous design satisfy some requirement, then this requirement will hold in every implementation
- Are the following realistic executions?
 - $(0,0) - A_x \rightarrow (1,0) - A_x \rightarrow (2,0) - A_x \rightarrow (3,0) \dots - A_x \rightarrow (105,0) - A_x \rightarrow \dots$
 - $(0,0) - A_x \rightarrow (1,0) - A_x \rightarrow (2,0) - A_y \rightarrow (2,1) - A_y \rightarrow (2,2) \dots - A_y \rightarrow (2,105) - A_y \rightarrow \dots$
- Does the system satisfy the following requirement?
 - "In every execution, values of both x and y eventually exceed 10"

Fairness Assumption

nat $x := 0; y := 0$
$A_x: x := x + 1$
$A_y: y := y + 1$



- Fairness assumption for a task
 - Assumption about the underlying platform/scheduler
 - Informally, an infinite execution is unfair for a task if the task does not get a chance to execute
- Unfair to A_y : $(0,0) - A_x \rightarrow (1,0) - A_x \rightarrow (2,0) - A_x \rightarrow (3,0) \dots - A_x \rightarrow (105,0) \dots$
- Unfair to A_x : $(0,0) - A_x \rightarrow (1,0) - A_x \rightarrow (2,0) - A_y \rightarrow (2,1) - A_y \rightarrow (2,2) \dots (2,105) \dots$
- Fairness assumptions restrict the set of “possible” executions to realistic ones without putting concrete bounds on relative speeds

Formalizing Fairness

nat $x := 0; y := 0$
$A_x: x := x + 1$
$A_y: x = 0 \rightarrow y := y + 1$

- **Definition 1 (Unconditional fairness):** An infinite execution is unconditionally fair to a task A if task A is executed repeatedly (i.e., infinitely often) during this execution
 - Is the following execution fair to task A_y ?
 $(0,0) -A_x \rightarrow (1,0) -A_x \rightarrow (2,0) -A_x \rightarrow (3,0) -A_x \rightarrow \dots (105,0) -A_x \rightarrow \dots$
 - After the first step, task A_y is not enabled and, thus, cannot be executed
This is a possible execution and should be considered fair
 - **Definition 2 (Weak fairness):** An infinite execution is weakly fair to a task A if, repeatedly, task A is either executed or disabled
 - In other words: if A is almost always* enabled, then A is infinitely often taken
- *only finitely many exceptions

Weak vs Strong Fairness

nat $x := 0; y := 0$
$A_x: x := x+1$
$A_y: \text{even}(x) \rightarrow y := y+1$

- Is the following execution fair to task A_y ?
 $(0,0) -A_x \rightarrow (1,0) -A_x \rightarrow (2,0) -A_x \rightarrow (3,0) \dots -A_x \rightarrow (105,0) -A_x \rightarrow \dots$
- Task A_y is not continuously enabled and, thus, this execution is weakly fair according to Definition 2 (but it may seem unfair)
- Definition 3 (Strong fairness):** An infinite execution is strongly fair to a task A if A is either repeatedly executed or almost always disabled
 - In other words: if A is infinitely often enabled, then A is infinitely often taken
- The above execution is weakly fair to task A_y , but not strongly fair

Fairness Assumption

- Fairness assumption for an asynchronous process P: for each output task and internal task we have either
 - No assumption
 - Weak-fairness assumption
 - Strong-fairness assumption
- Restricts the set of possible infinite executions
- Affects whether the process meets a requirement:
 - Maybe not all executions satisfy the given requirement, but all fair executions satisfy the requirement

Requirements under Fairness Assumptions

Process P1

nat $x := 0; y := 0$

$A_x: x := x+1$

$A_y: y := y+1$

Process P3

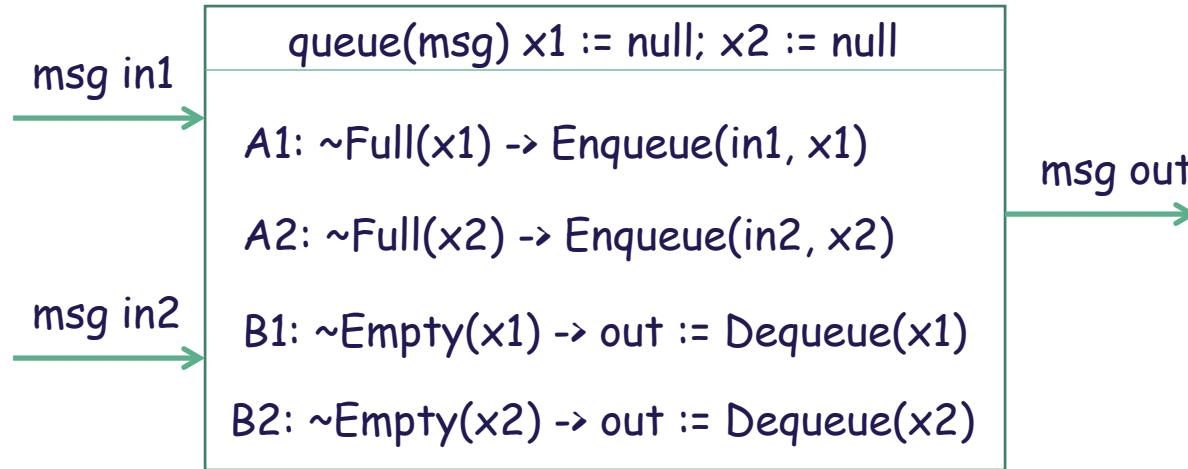
nat $x := 0; y := 0$

$B_x: x := x+1$

$B_y: \text{even}(x) \rightarrow y := y+1$

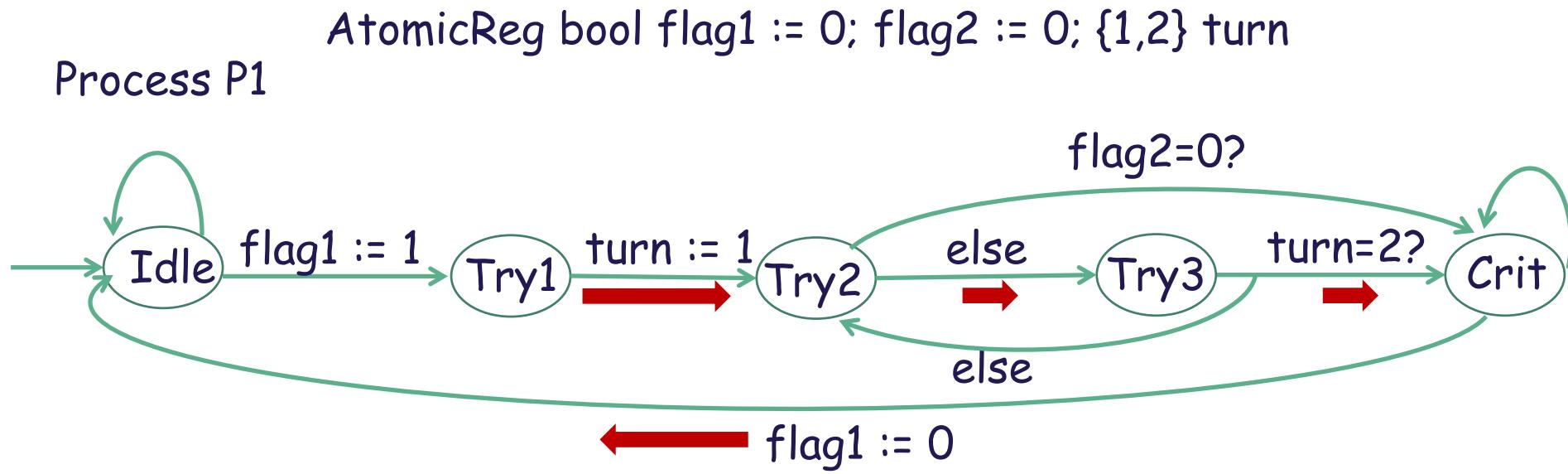
- Under which fairness assumptions do P1 and P3 satisfy the following?
- Requirement: Eventually $x+y > 10$
 - Both satisfy this without any fairness assumption
- Requirement: Eventually $x > 10$
 - P1 with weak fairness for A_x , P3 with weak fairness for B_x
- Requirement: Eventually $y > 10$
 - P1 with weak fairness for A_y , P3 with strong fairness for B_y
- Requirement: Eventually $x > y$
 - Not satisfied, no matter which fairness assumption we make

Asynchronous Merge



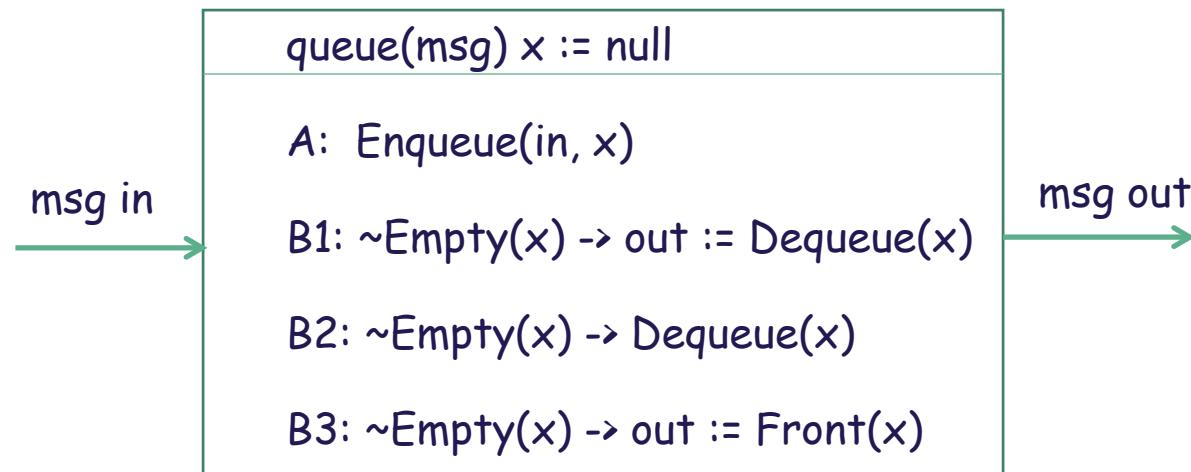
- ❑ Consider the following requirement:
“Whenever an input message is received, it is eventually output”
 - Does not hold without fairness assumptions
- ❑ Which fairness assumptions should we make?
 - Weak fairness for output tasks B1 and B2 suffices

Fairness Assumptions for Mutual Exclusion Protocol



- Consider the following requirement:
“Whenever a process wants to enter the critical section, it eventually will”
- Which fairness assumptions should we make?
 - Weak fairness for highlighted steps/tasks suffices

Unreliable FIFO

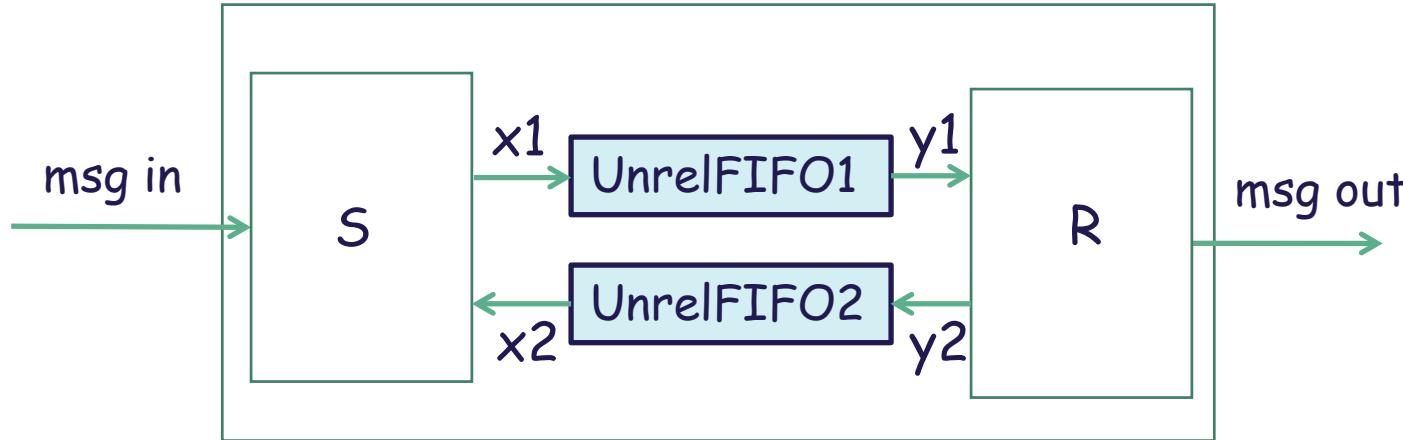


- Models a link that may lose and/or duplicate messages
 - Tasks A and B1 model normal input/output behavior
 - Task B2: Loss of message
 - Task B3: Duplication of message
- What are “natural” fairness assumptions for these tasks?
 - Strong fairness for B1, no assumptions for B2 and B3

Fairness Summary

- Fairness is an assumption about the underlying platform or scheduler
 - The fewer assumptions we make, the better
- For each output and internal task, we can assume weak or strong fairness, as needed
 - Strong fairness needed if the task can switch between enabled and disabled due to execution of other tasks
- Restricts the set of possible infinite executions, and allows satisfaction of more requirements
 - Does not affect the set of reachable states and safety properties
 - Does not change underlying coordination
- Distinction in terminology:
 - Fairness assumption for tasks (ensure tasks get executed as expected)
 - "Fairness" requirement for protocols (high-level goal)

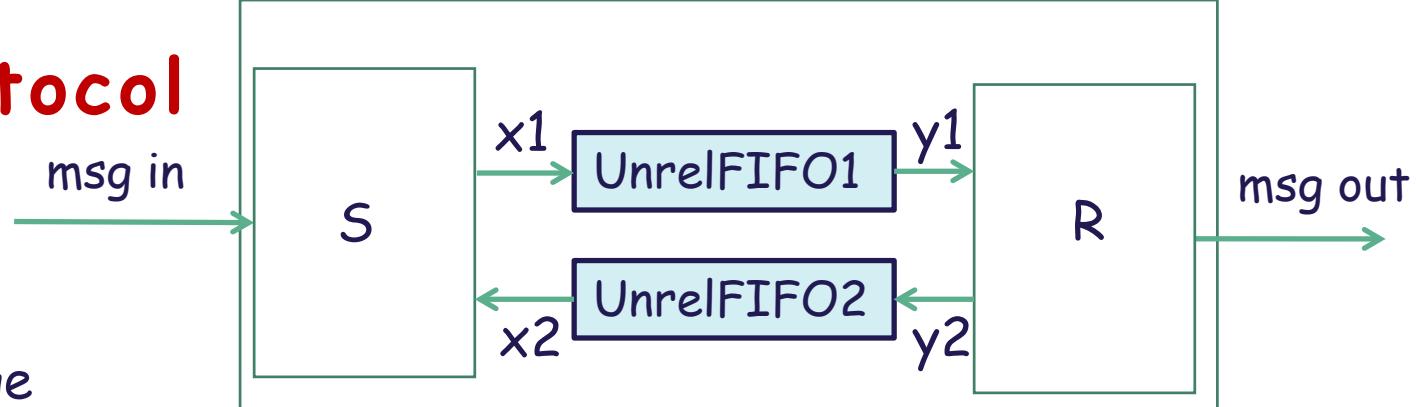
Reliable Transmission Problem



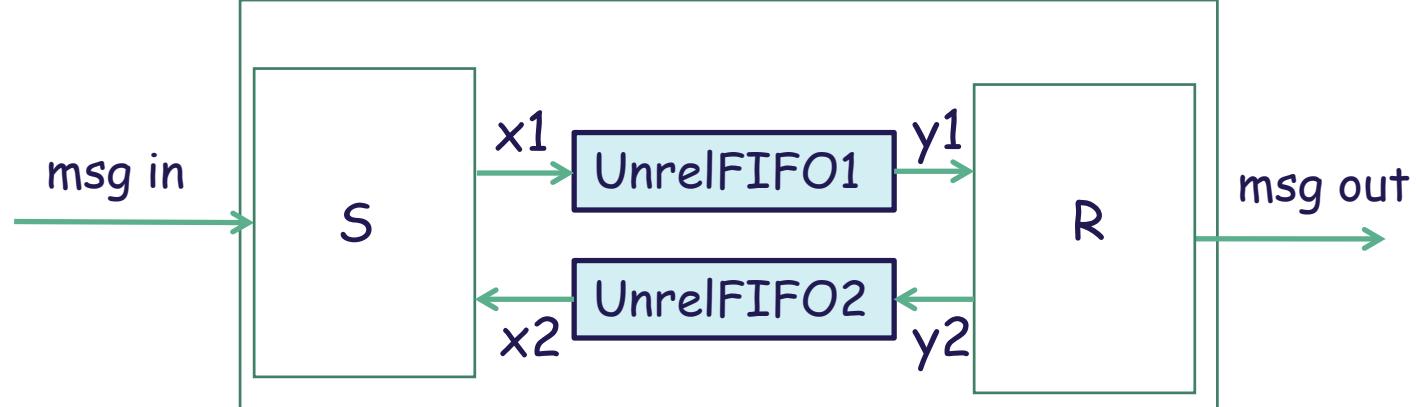
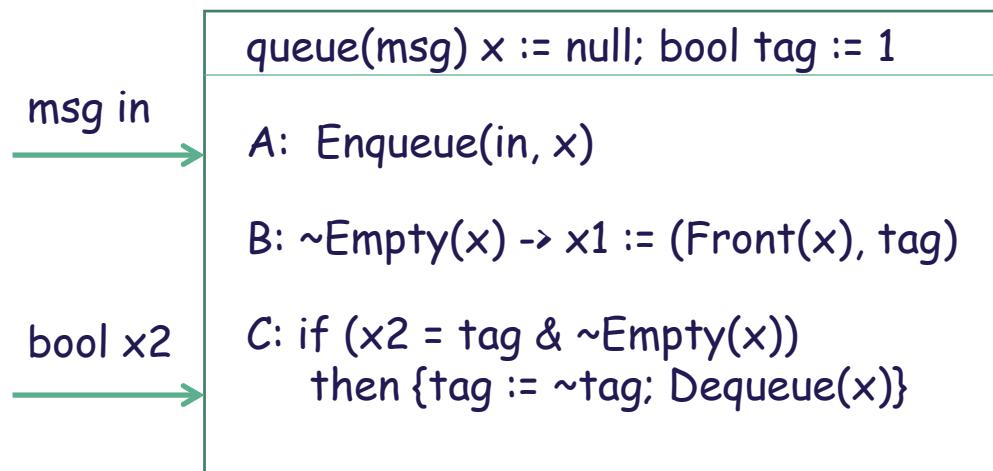
- How to implement a reliable FIFO link using unreliable ones?
- Design asynchronous processes S and R so that the sequence of messages received on the channel in coincides with the sequence of messages delivered on the channel out

Alternating Bit Protocol

- How can sender S be sure that receiver R got a copy of the message in presence of message losses?
 - S must repeatedly send a message
 - R must send back an acknowledgement, and do so repeatedly
- How can R distinguish between a duplicated/repeated copy and a fresh message?
 - Each message must be tagged with "extra" bits
- Alternating bit protocol:
 - Tagging each message as well as acknowledgement with a single bit suffices
 - Both S and R keep a local tag bit
 - If the tag of the incoming message matches with the local tag, then the message is considered "fresh" and the local tag is toggled



ABP Sender

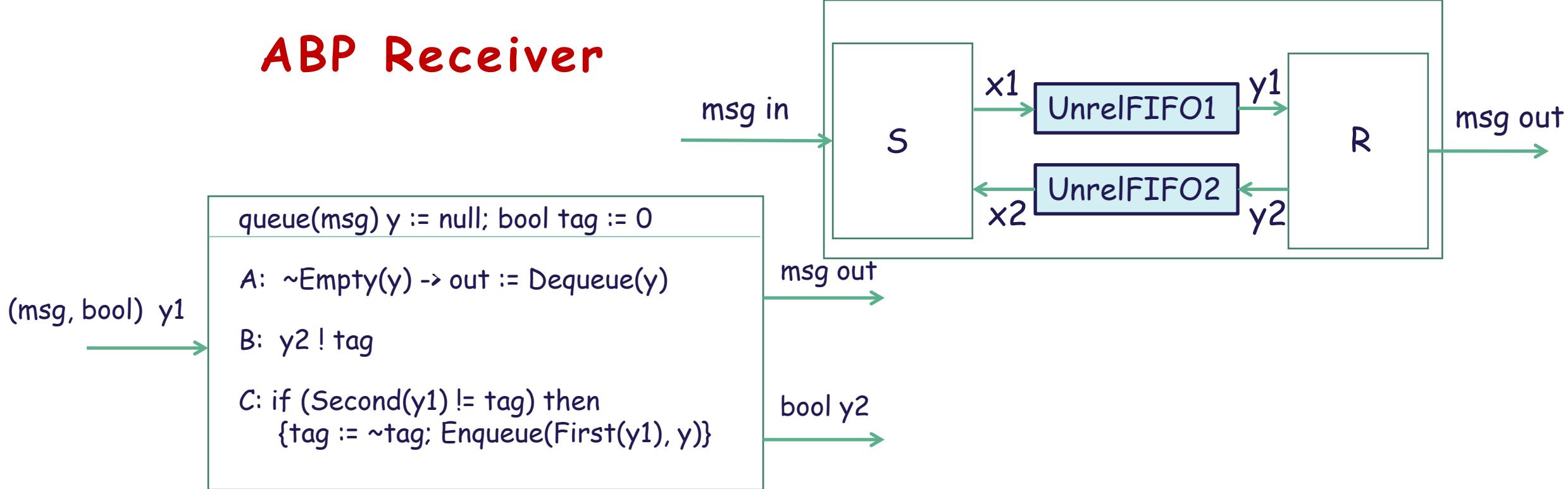


Task A: Store incoming messages in queue x

Task B: Transmit message at front of queue x tagged with local tag
Do not remove the message: this ensures it is transmitted repeatedly

Task C: If acknowledgment matches tag, the message was successfully delivered; so remove it from x and flip tag

ABP Receiver



Task A: Transmit outgoing messages from queue **y** to output channel

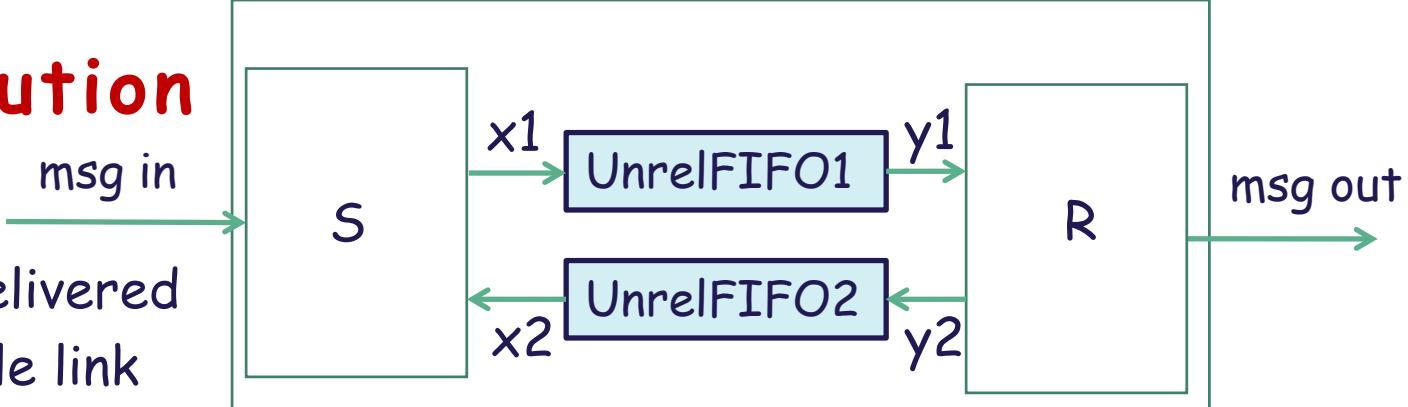
Task B: Transmit local tag as acknowledgement on channel **y2**

Note: Same acknowledgement is potentially transmitted repeatedly

Task C: If tag of incoming message matches local tag, then the message is new; so add it to **y** and flip tag

ABP Sample Execution

- Initially $S.tag = 1$ and $R.tag = 0$
- Suppose S receives message m to be delivered
- S repeatedly sends $(m, 1)$ over unreliable link
- Eventually, R gets at least one copy of $(m, 1)$
- Before receiving $(m, 1)$, R repeatedly sends acknowledgement 0, which is simply ignored by S
- When R receives $(m, 1)$ the first time, it stores m in queue y (and this message will eventually be transmitted on out), and sets tag to 1
- Duplicate versions of $(m, 1)$ are ignored by R
- R repeatedly sends acknowledgment 1
- Eventually, S receives at least one acknowledgment 1, and then removes m from input queue and sets its tag to 0
- Duplicate versions of acknowledgment 1 are ignored by S
- Messages to be sent are queued up in x , and S will now repeat the whole cycle by sending the next message m' along with tag 0, i.e., $(m', 0)$



Principles of Cyber-Physical Systems

Chapter 7: Timed Model

Instructors: Martijn Goorden, Kim G. Larsen,
Christian Schilling, Max Tschaikowski
`{mgoorden,kgl,christianms,tschaikowski}@cs.aau.dk`

Slides courtesy: Rajeev Alur
`alur@cis.upenn.edu`

Models of Reactive Computation

□ Synchronous model

- Components execute in a sequence of discrete rounds in lock-step
- Computation within a round: Execute all tasks in an order consistent with precedence constraints

□ Asynchronous model

- Speeds at which different components execute are independent
- Computation within a step: Execute a single task that is enabled

□ Timed model

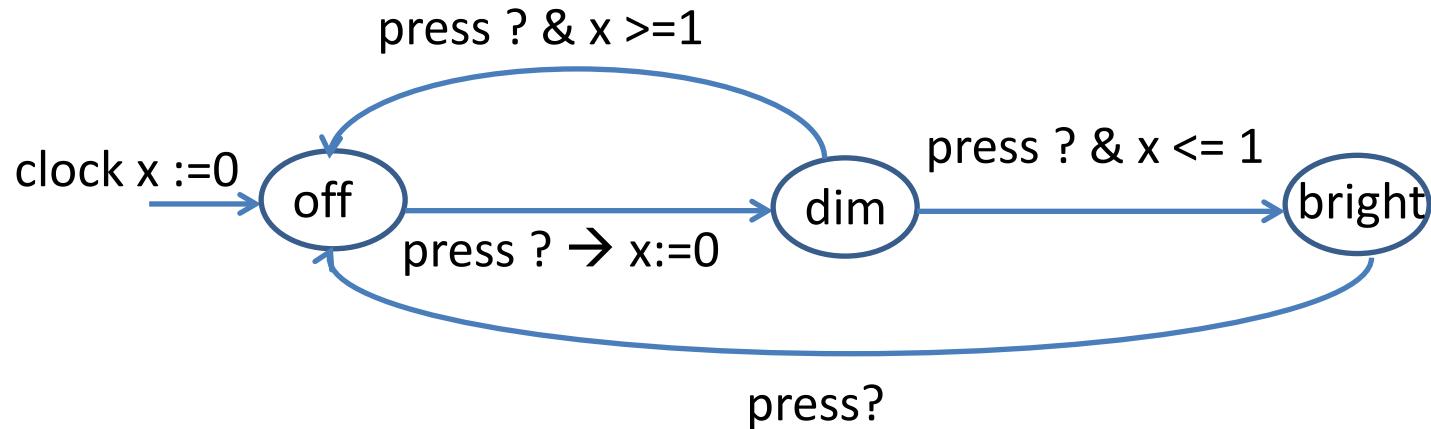
- Like asynchronous for communication of information
- Can rely on global time for coordination



Quantitative aspects
(timing, energy, bandwidth, memory,...)
crucial for several embedded systems



Example Timed Model



Initial state = (mode = off, $x = 0$)

Timed transition: (off, 0) – 0.5 → (off, 0.5)

Input transition: (off, 0.5) – press? → (dim, 0)

Timed transition: (dim, 0) – 0.8 → (dim, 0.8)

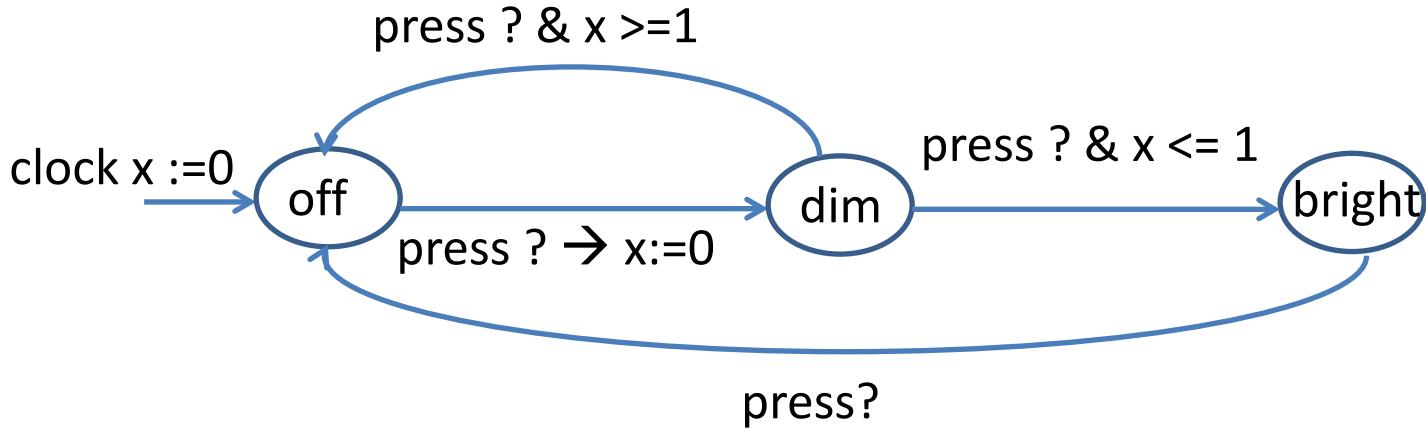
Input transition: (dim, 0.8) – press? → (bright, 0.8)

Timed transition: (dim, 0.8) – 1 → (dim, 1.8)

Input transition: (dim, 1.8) – press? → (off, 1.8)

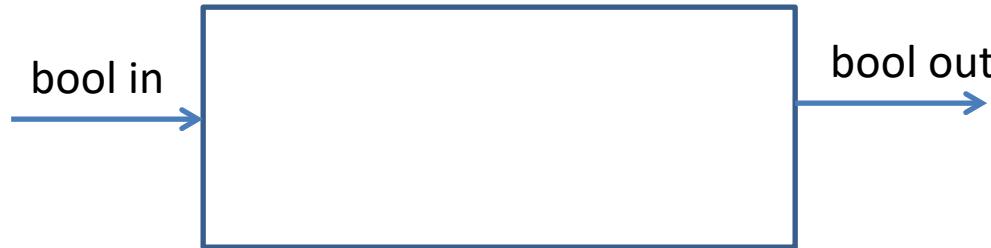


Example Timed Model



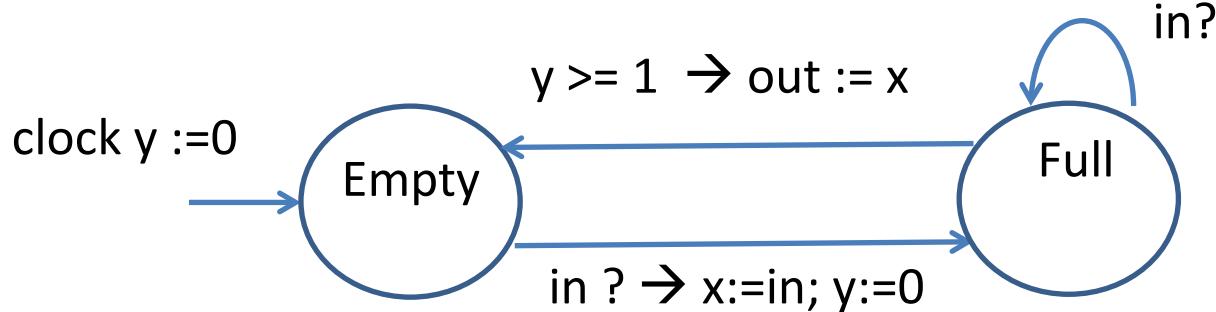
- Clock variables
 - Tests and updates in mode-switches like other variables
 - New feature: During a timed transition of duration δ , the value of a clock variable increases by an amount equal to δ
- Timing constraint: Setting x to 0 for " $off \rightarrow dim$ " and guard $x \leq 1$ for " $dim \rightarrow bright$ " specifies that timing of these two transitions is ≤ 1 apart

Example: Timed Buffer



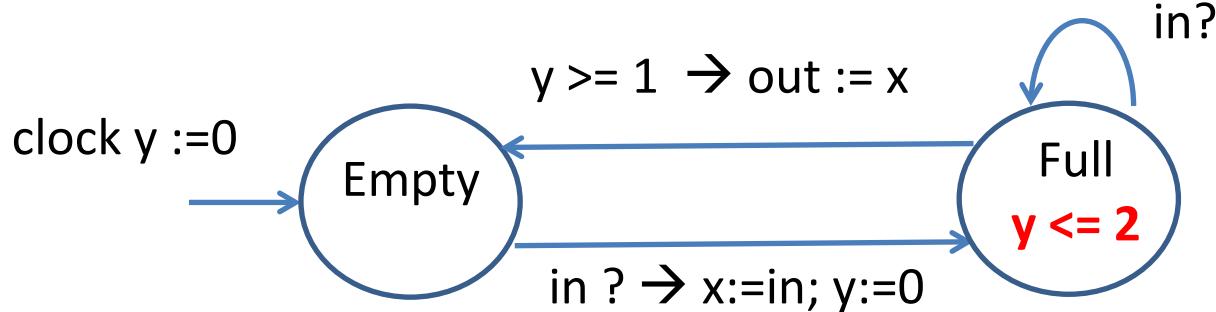
- Buffer with a bounded delay
- Behavior to be modeled: Input received on the channel in is transmitted on the output channel after a delay of δ such that
$$LB \leq \delta \leq UB$$
(i.e. we know lower/upper bounds on this delay)

Modeling Timed Buffer



- Mode says whether the buffer is full or not
- State variable x remembers the last input value when buffer is full
- Clock variable y tracks the time elapsed since buffer got full
- When full, input events are ignored
- Guard $y \geq 1$ ensures that at least 1 time unit elapses in mode Full
- How to ensure that mode-switch from Full to Empty is executed before the clock x exceeds the upper bound 2?

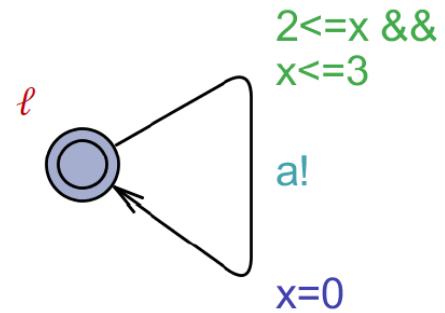
Clock Invariants



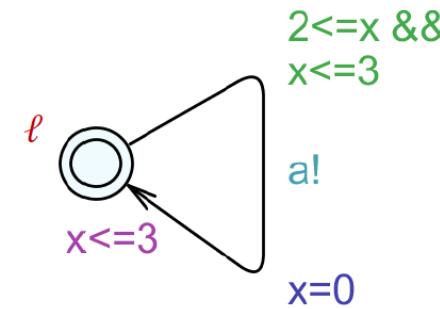
- The constraint " $y \leq 2$ " associated with the mode Full is called "clock invariant"
- A timed transition of duration δ is allowed only when the clock invariant remains true during the entire duration
 - Timed transition (Full, $x, y=0.8$) - 0.7 → (Full, $x, y=1.5$) allowed
 - Timed transition (Full, $x, y=0.8$) - 1.4 → (Full, $x, y=2.2$) disallowed
- Clock invariants useful to limit how long a process stays in a mode

Invariants or Not?

Not

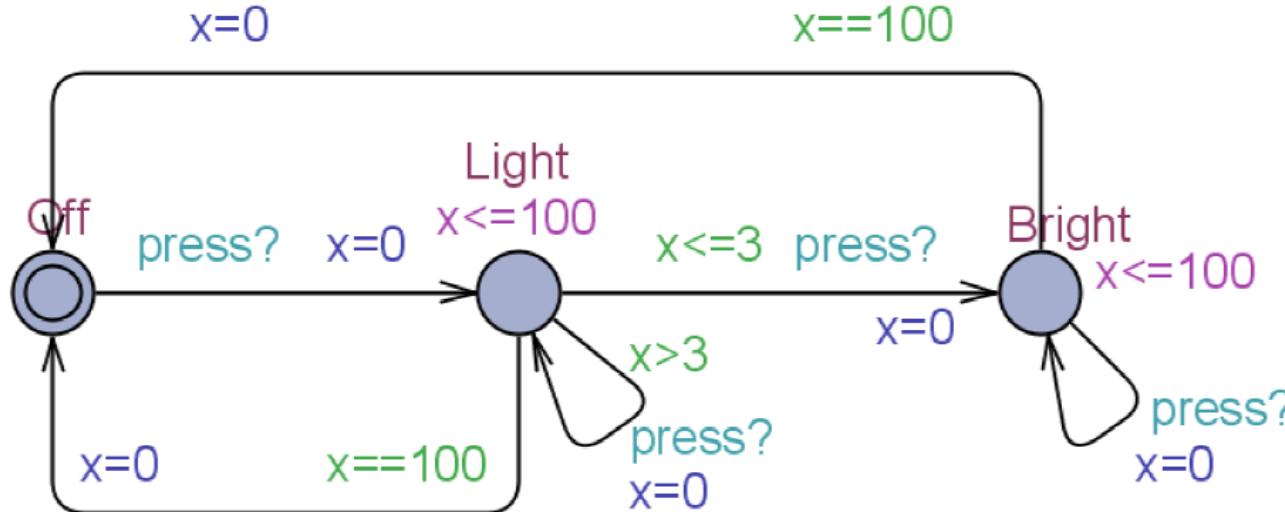


Invariant



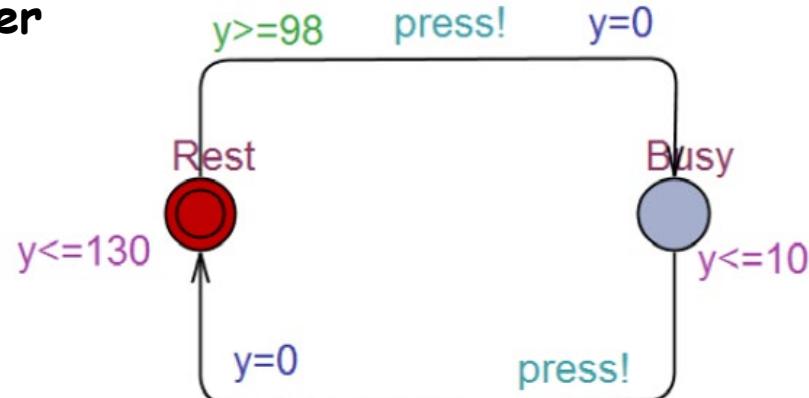
Intelligent Light Control in UPPAAL

Contr



What does **Contr** do?
In any mode if a press? has not occurred for 100tu then time-out, i.e. go to Off.

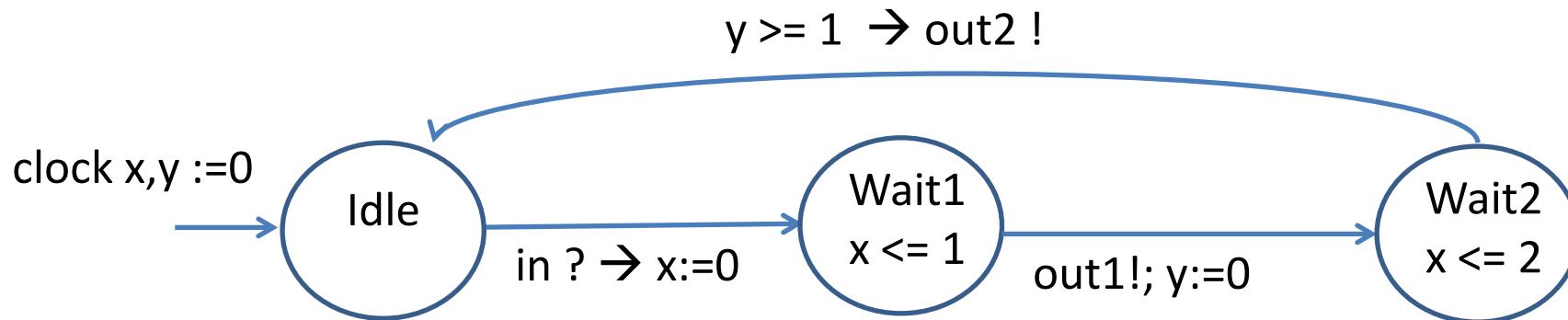
User



What does **User** do?

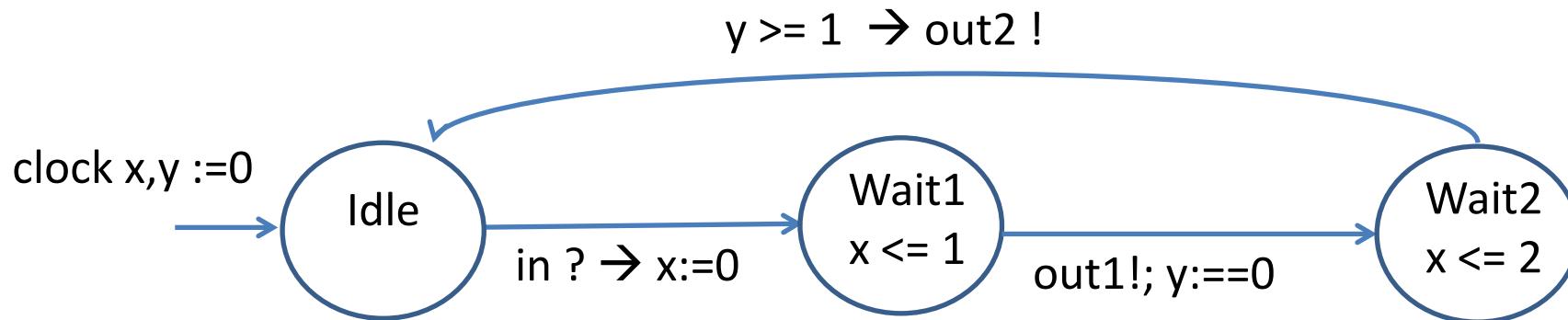
What does **Contr | User** do?
Let us consult UPPAAL!

Example with Two Clocks



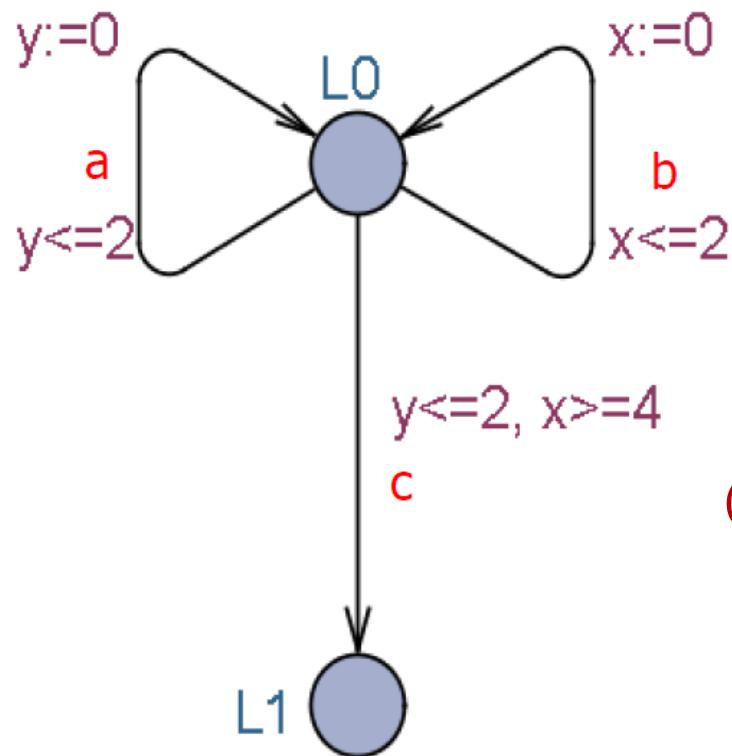
- Input event: in; Output events out1, out2
- Two clock variables x and y
- In mode Wait2, as time elapses both clocks increase (at same rate)
- Sample timed transitions:
 $(\text{Wait2}, x=0.8, y=0) -0.3 \rightarrow (\text{Wait2}, 1.1, 0.3) -0.72 \rightarrow (\text{Wait2}, 1.82, 1.02)$

Two Clock Example



- Clock x tracks time elapsed since the occurrence of input event
- Clock y tracks time elapsed since the occurrence of output event out1
- What is the behavior of this model?
- If input event occurs at time t, the process issues an output on channel out1 at time t' within the interval [t, t+1], and then produces output on channel out2 at time t'' within the interval [t'+1, t+2]

Another Two Clock Example



Is L_1 reachable ?

$$\begin{aligned}(L_0, x = 0, y = 0) - \epsilon(2) &\rightarrow \\&(L_0, x = 2, y = 2) \\-\alpha &\rightarrow \\&(L_0, x = 2, y = 0) \\-\epsilon(2) &\rightarrow \\&(L_0, x = 4, y = 2) \\-\gamma &\rightarrow\end{aligned}$$

Example Specification

Consider a timed process with

Input event i , Output events u and v

Desired behavior

Whenever it receives input, it produces both its output events

Time delay between $i?$ and $u!$ is in the interval $[2, 4]$

Time delay between $i?$ and $v!$ is in the interval $[3, 5]$

Input events received during this are ignored

Let us draw a timed state machine that captures this behavior

Consider a timed process with

Input event i , Output events u and v

Desired behavior

Whenever it receives input, it produces both its output events

Time delay between $i?$ and $u!$ is in the interval $[2, 4]$

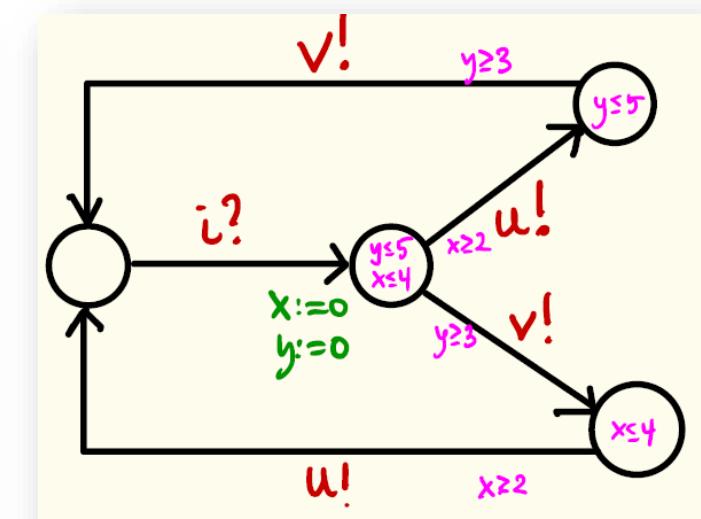
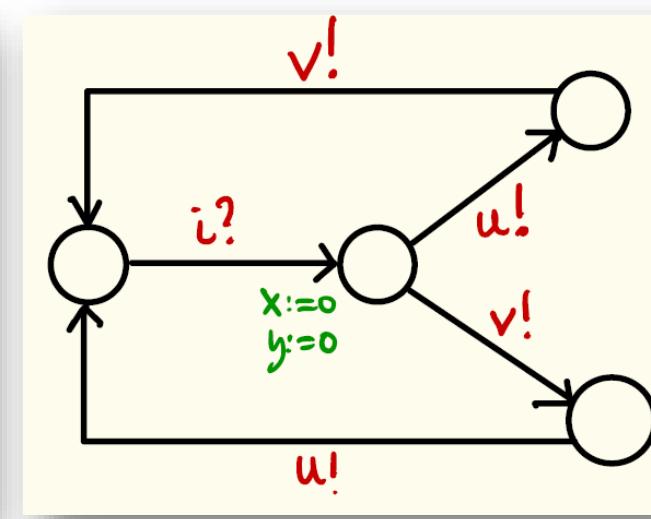
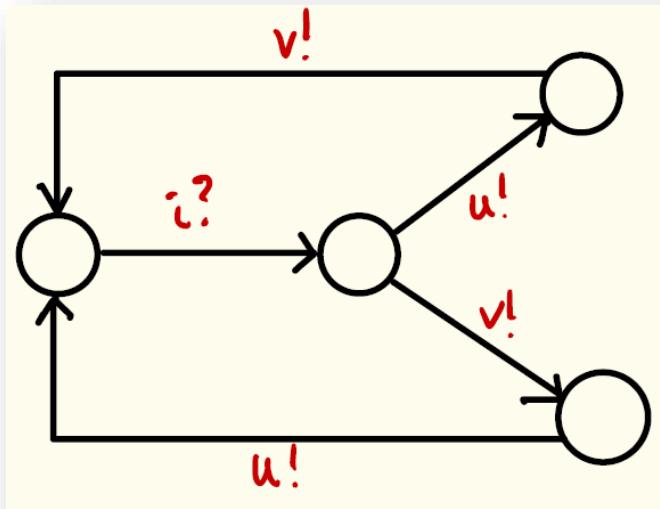
Time delay between $i?$ and $v!$ is in the interval $[3, 5]$

Input events received during this are ignored

Let us draw a timed state machine that captures this behavior

Example Specification

Clock x measure time delay between $i?$ and $u!$
Clock y measure time delay between $i?$ and $v!$



Example Specification

Consider a timed process with

Input event i , Output events u and v

Desired behavior

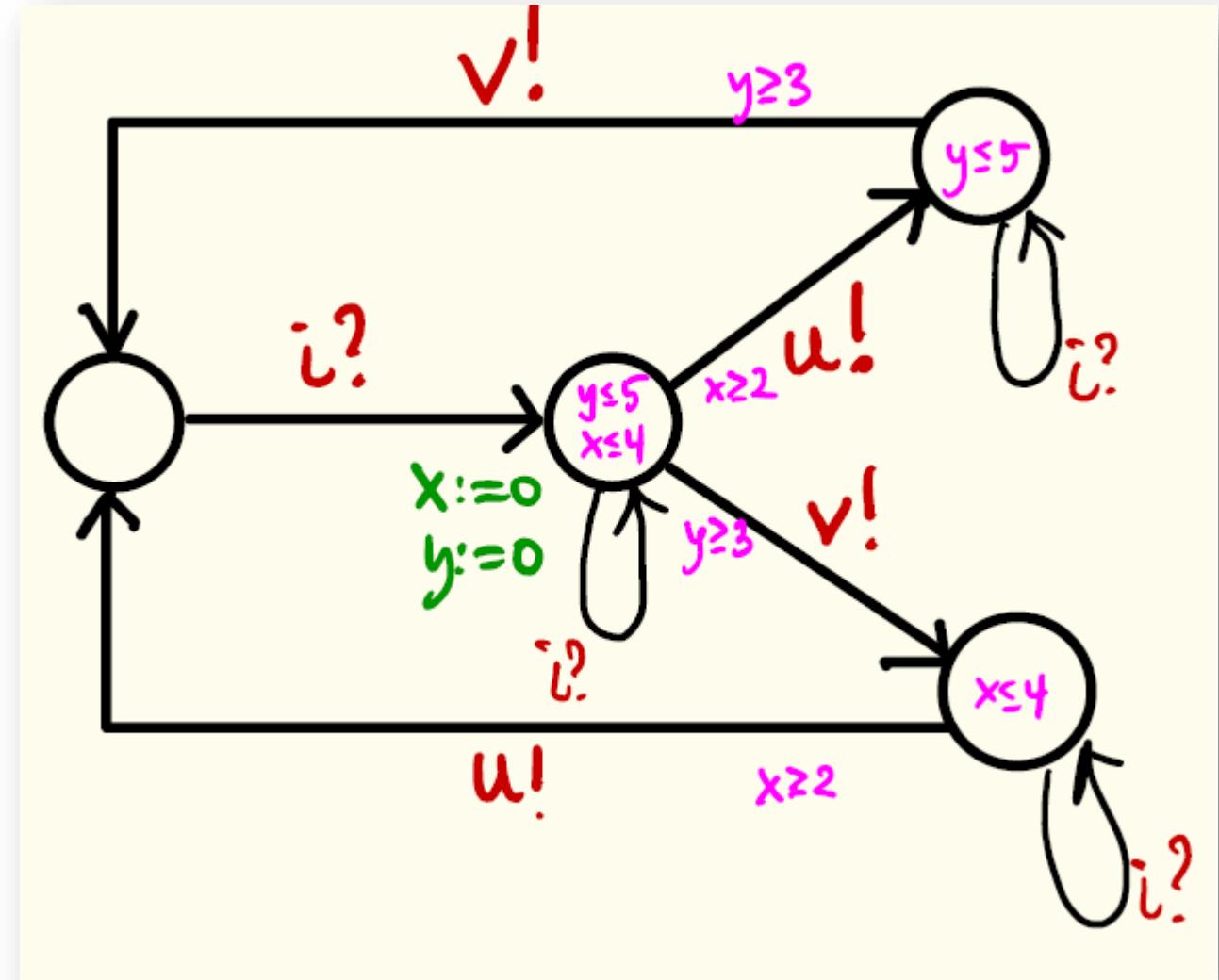
Whenever it receives input, it produces both its output events

Time delay between $i?$ and $u!$ is in the interval $[2, 4]$

Time delay between $i?$ and $v!$ is in the interval $[3, 5]$

Input events received during this are ignored

Let us draw a timed state machine that captures this behavior

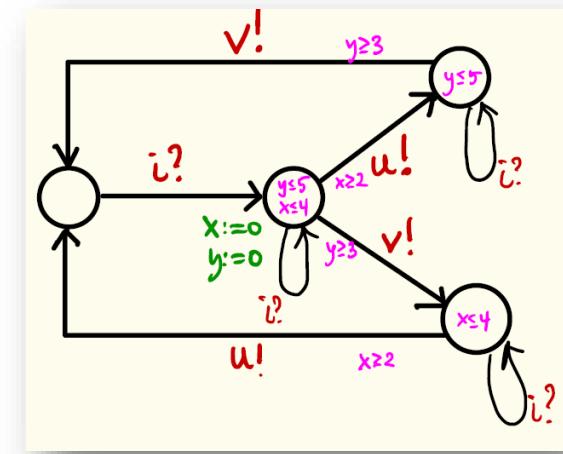


BREAK

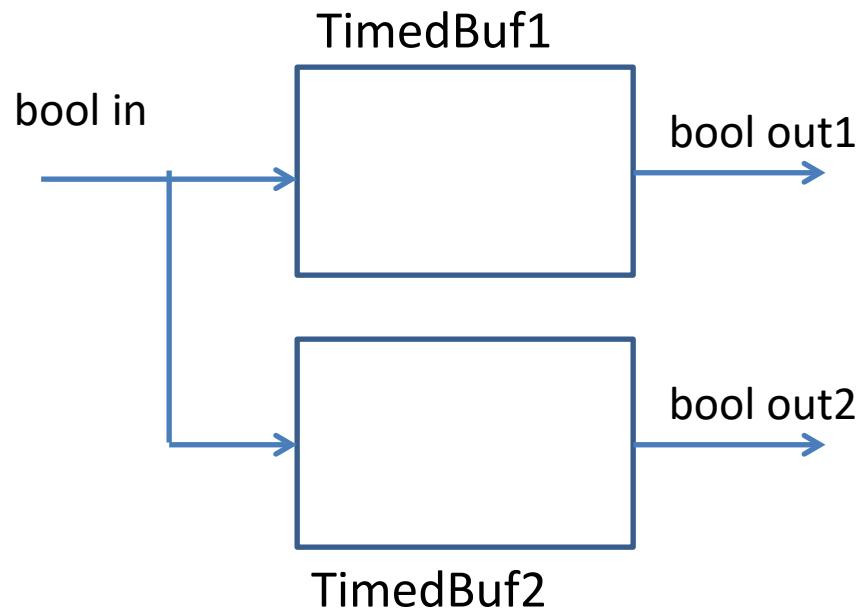


Definition of Timed Process

- A timed process TP consists of
 1. An asynchronous process P, where some of the state variables can be of type clock (ranging over non-negative real numbers)
 2. A **clock invariant CI** which is a Boolean expression over the clock variables of P
- Define inputs, outputs, states, initial states, internal actions, input actions, output actions exactly the same as the asynchronous model
- **Timed actions:** Given a state s and real-valued time $\delta > 0$, $s - \delta \rightarrow s + \delta$ is a transition of duration δ if the state $s + t$ satisfies the expression CI for all values of t in the interval $[0, \delta]$
 - For a state s and time δ , $s + \delta$ is another state which assigns the value $s(x) + \delta$ to every clock variable x , and $s(y)$ to every variable y of a type other than clock
 - If clock-invariant is a convex constraint then it is sufficient to check that the end-states s and $s + \delta$ satisfy CI



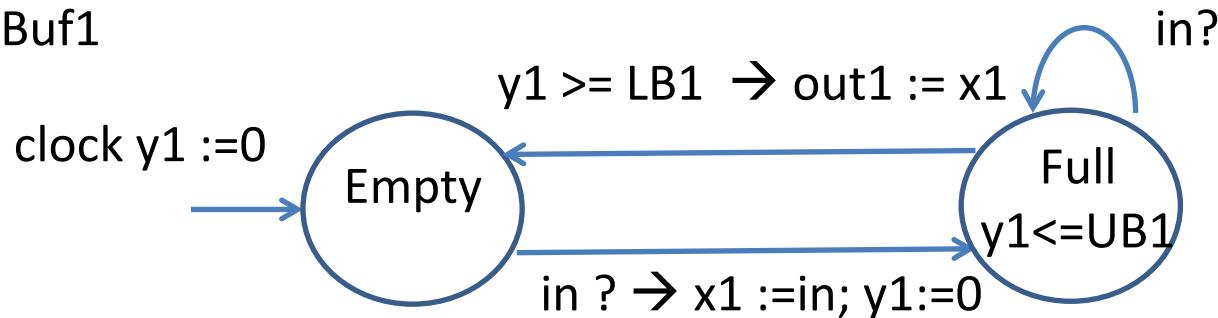
Composition of Processes



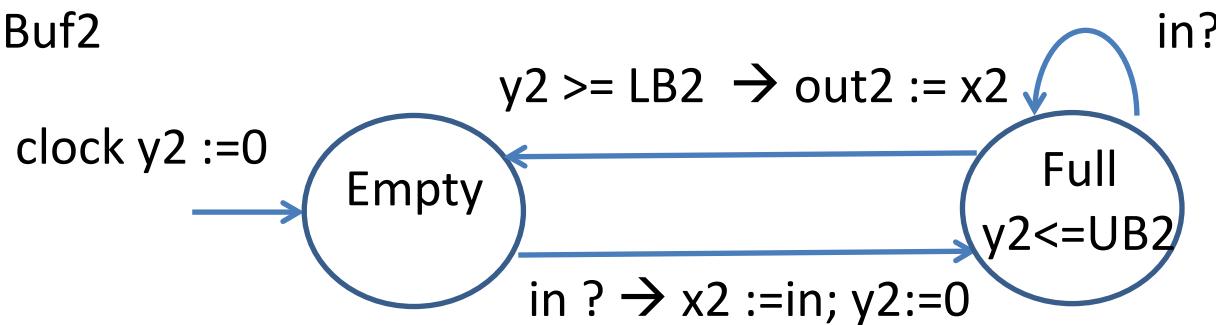
- How to construct timed process corresponding to the composition of the two processes?
- What are the possible behaviors of the composite process?

Composition of Timed Processes

TimedBuf1

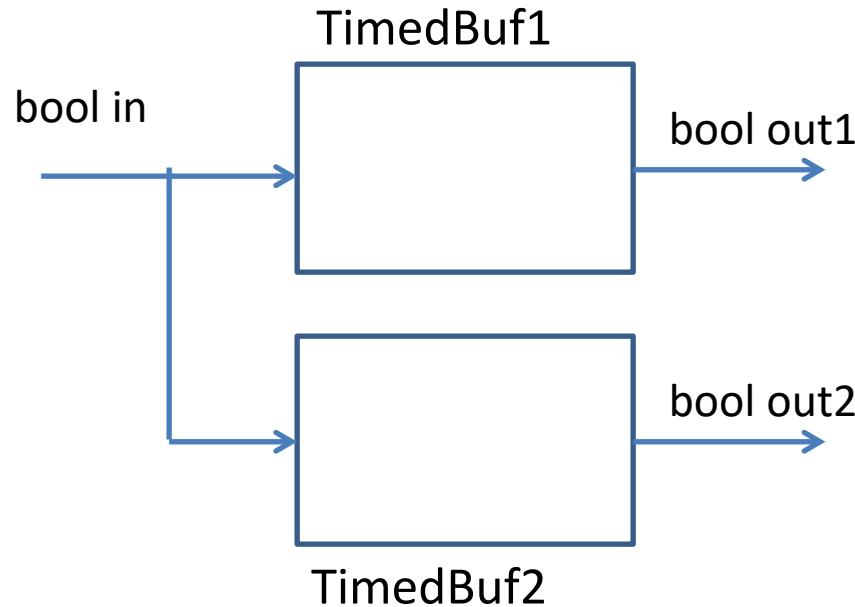


TimedBuf2



Construct the composite process with four modes

Composition of Processes

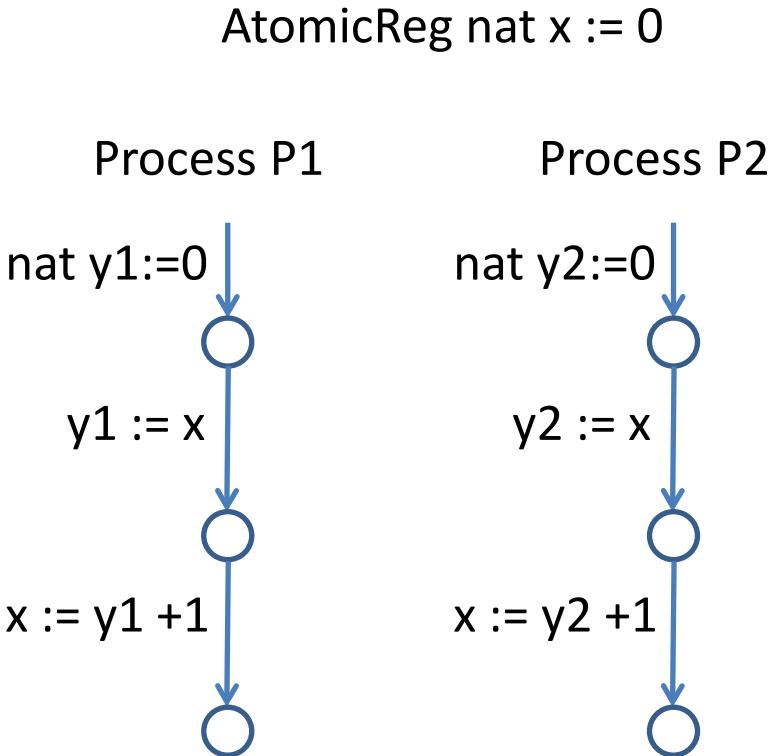


- If UB1 < LB2 then out1 guaranteed to occur before out2
 - Implicit coordination based on bounds on delays
- Is it possible to observe 2 out1 events without intervening out2 event?
 - Depends on relative magnitudes of bounds (need timing analysis!)

Definition of Parallel Composition

- Consider timed processes $TP1 = (P1, CI1)$ and $TP2 = (P2, CI2)$
- When is the parallel composition $TP1 \mid TP2$ defined?
 - Exactly when the asynchronous parallel composition $P1 \mid P2$ is defined (that is, when the outputs of the two are disjoint)
- $TP1 \mid TP2 = (P1 \mid P2, CI1 \& CI2)$
 - Asynchronous composition of $P1$ and $P2$ defines the internal, input and output actions of the composite
 - Conjunction of the clock-invariants defines the clock-invariant of the composite
- Consequence: The composite process can allow a timed action of duration δ exactly when both $TP1$ and $TP2$ are willing to wait for time δ
- Timed model is sometimes called the "semi-synchronous" model (mix of asynchronous and synchronous)

Shared Memory Programs with Atomic Registers



Declaration of shared variables
+ Code for each process

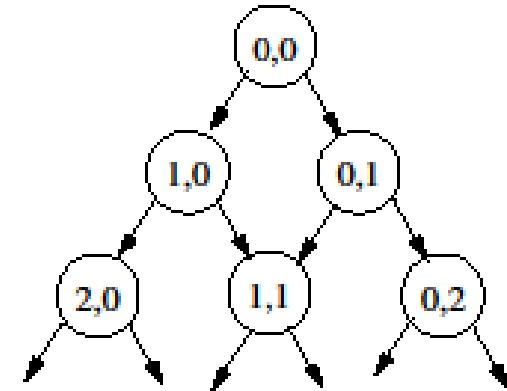
Key restriction: Each statement of a process either
changes local variables,
reads a single shared var, or
writes a single shared var

Execution model: execute one step of one of the processes

What if we knew lower and upper bounds on how long a read or a write takes? Can we solve coordination problems better?

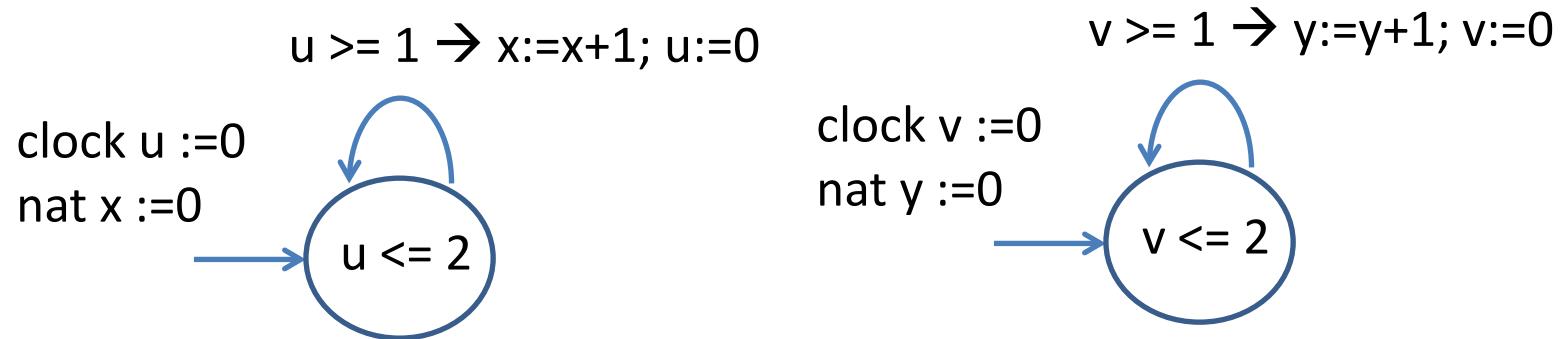
Asynchronous Execution Model

```
nat x:=0; y:=0
Ax: x := x+1
Ay: y :=y+1
```



- Tasks A_x and A_y execute in an arbitrary order
- For every possible choice of numbers m, n , the state $(x=m, y=n)$ is reachable
- Recall: Fairness assumptions can be used to rule out infinite executions where one of the tasks is ignored forever (but this does not affect the set of reachable states)
- What if we know how long do each of these increments take?

Timed Increments



- Task A_x increments x , and this takes between 1 to 2 time units
- Task A_y increments y , and this also takes between 1 to 2 time units
- Two tasks execute in parallel, asynchronously, but timing introduces loose coordination
- Which states are reachable? What is the co-relationship between m and n so that the state $(x=m, y=n)$ is reachable?

Mutual Exclusion Problem

Process P1

Entry Code

Critical Section

To be designed

Process P2

Entry Code

Critical Section

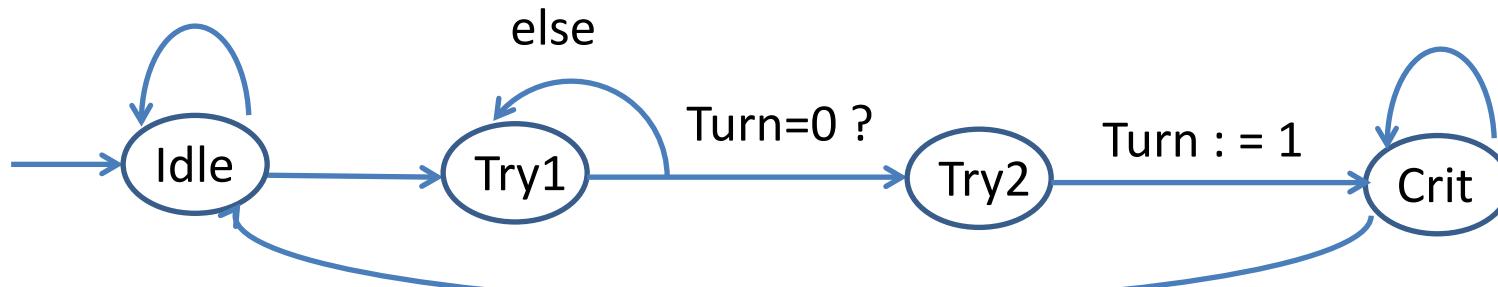
- Safety Requirement: Both processes should not be in critical section simultaneously (can be formalized using invariants)
- Absence of deadlocks: If a process is trying to enter, then some process should be able to enter



Mutual Exclusion: Attempted Solution

AtomicReg {0, 1, 2} Turn := 0

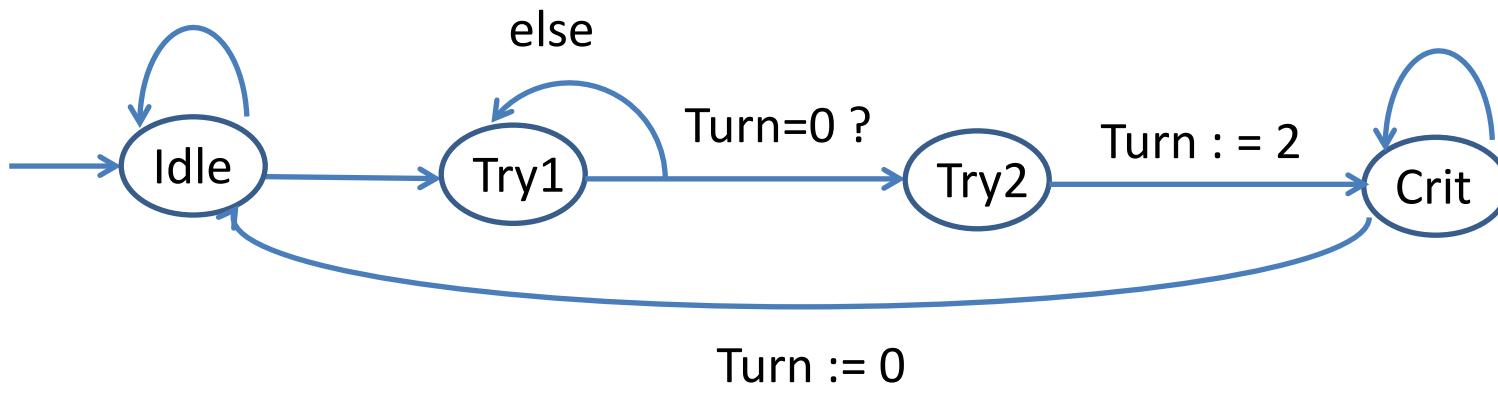
Process P1



Process P2

Turn := 0

What's the bug?

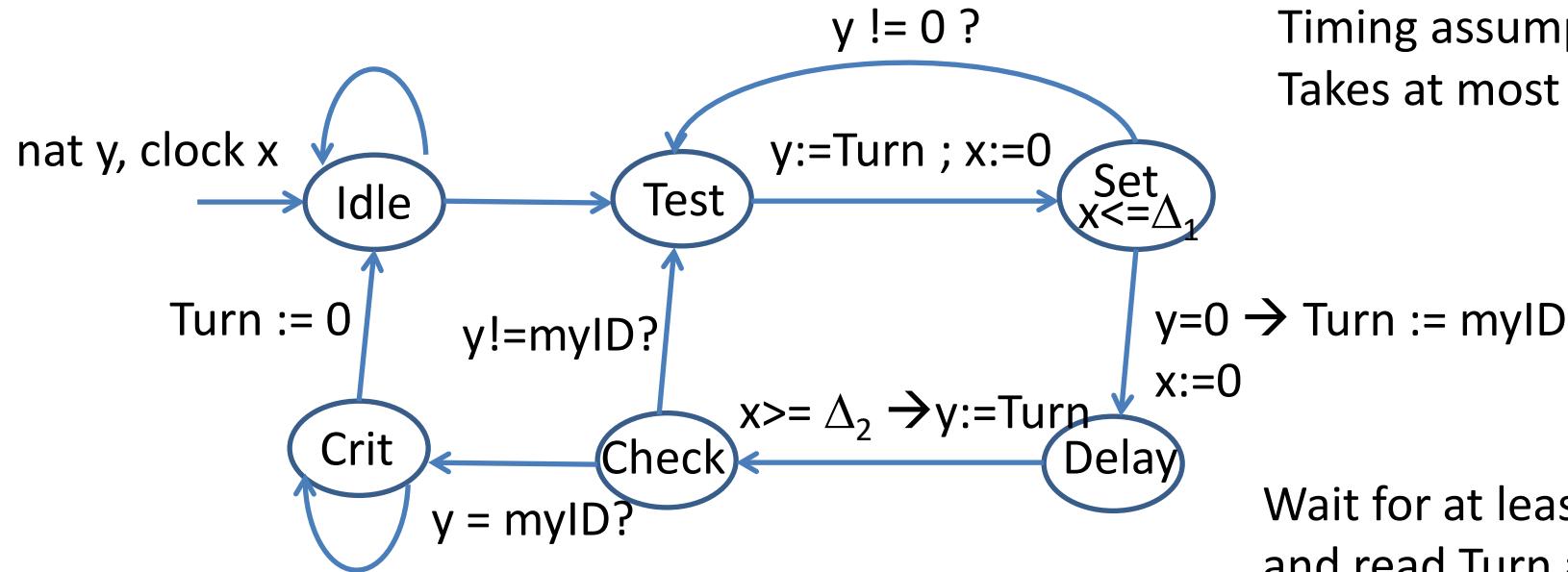


Timing-based Mutual Exclusion

1. When a process wants to enter critical section, it reads the shared variable Turn
2. If Turn != 0 then try again (goto step 1)
3. If Turn = 0 then set Turn to your ID
4. Proceeding directly to critical section is a problem (since the other process may also have concurrently read Turn to be 0, and updating Turn to its own ID)
5. Solution: Delay and wait till you are sure that concurrent writes are finished
6. Read Turn again: if Turn equals your own ID then proceed to critical section, if not goto step 1 and try again
7. When done with critical section, set Turn back to 0

Fisher's Protocol for Mutual Exclusion

AtomicReg Turn := 0



Timing assumption:
Takes at most Δ_1 to write

$y=0 \rightarrow Turn := myID$

Wait for at least Δ_2 time,
and read Turn again

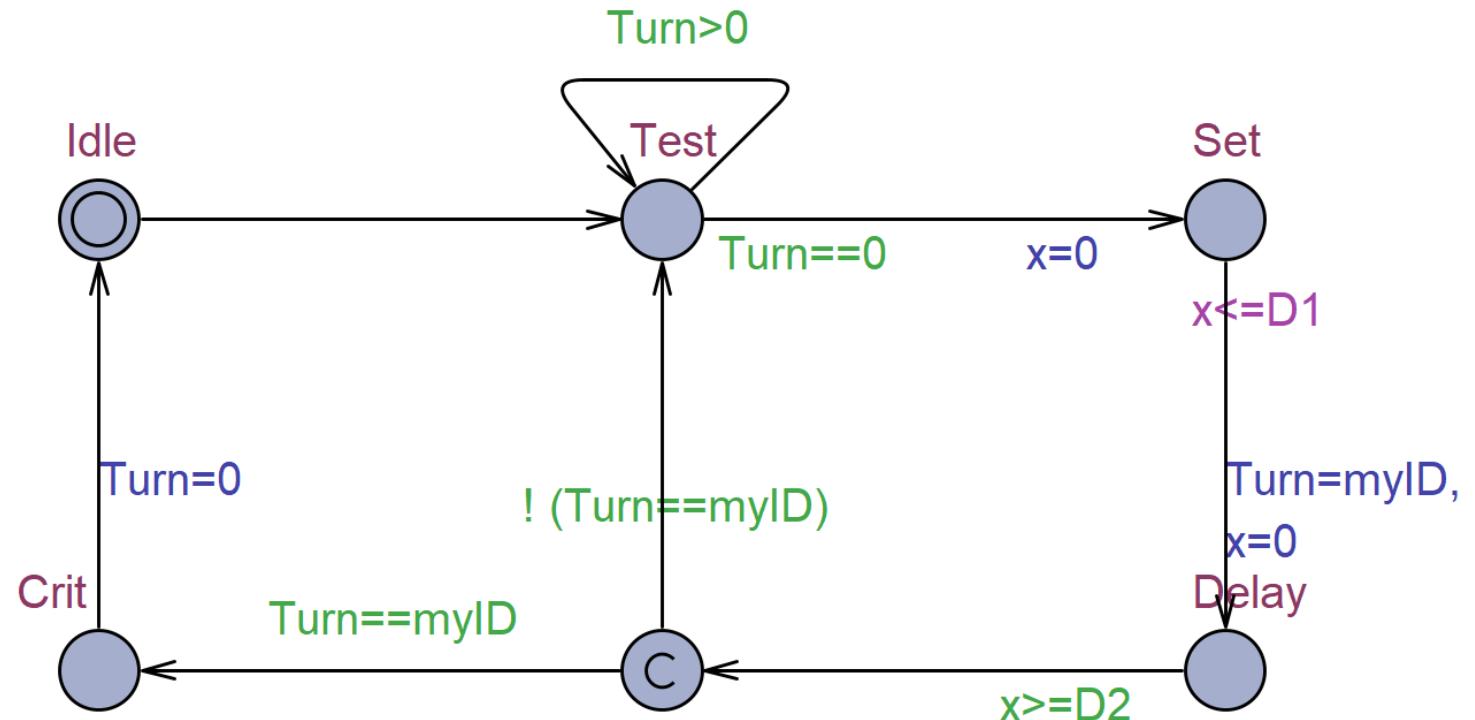
Why does it work ?



Properties of Fisher's Protocol

- Assuming $\Delta_2 > \Delta_1$, the algorithm satisfies:
 - Mutual exclusion: Two processes cannot be in critical section simultaneously
 - Deadlock freedom: If a process wants to enter critical section then some process will enter critical section
- Protocol works for arbitrarily many processes (not just 2)
 - Note: In the asynchronous model, mutual exclusion protocol for N processes is lot more complex than Peterson's algorithm
- Does the protocol satisfy the stronger property of starvation freedom: If a process P_i wants to enter critical section then it eventually will ?
- If $\Delta_2 \leq \Delta_1$ then does mutual exclusion hold? Does deadlock freedom hold?

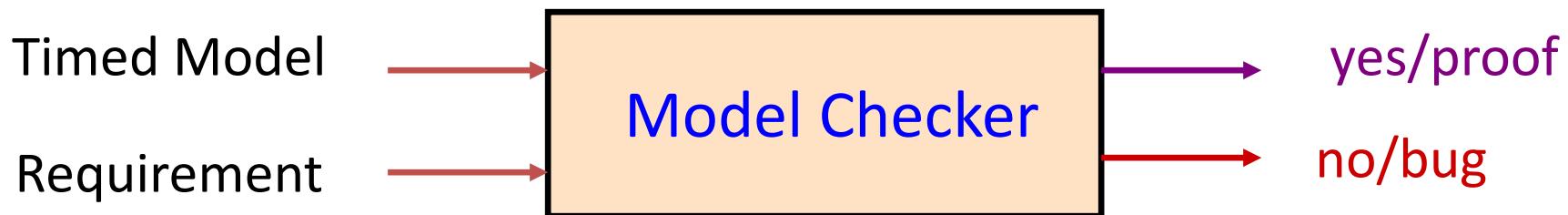
Fischer's Protocol in UPPAAL



BREAK

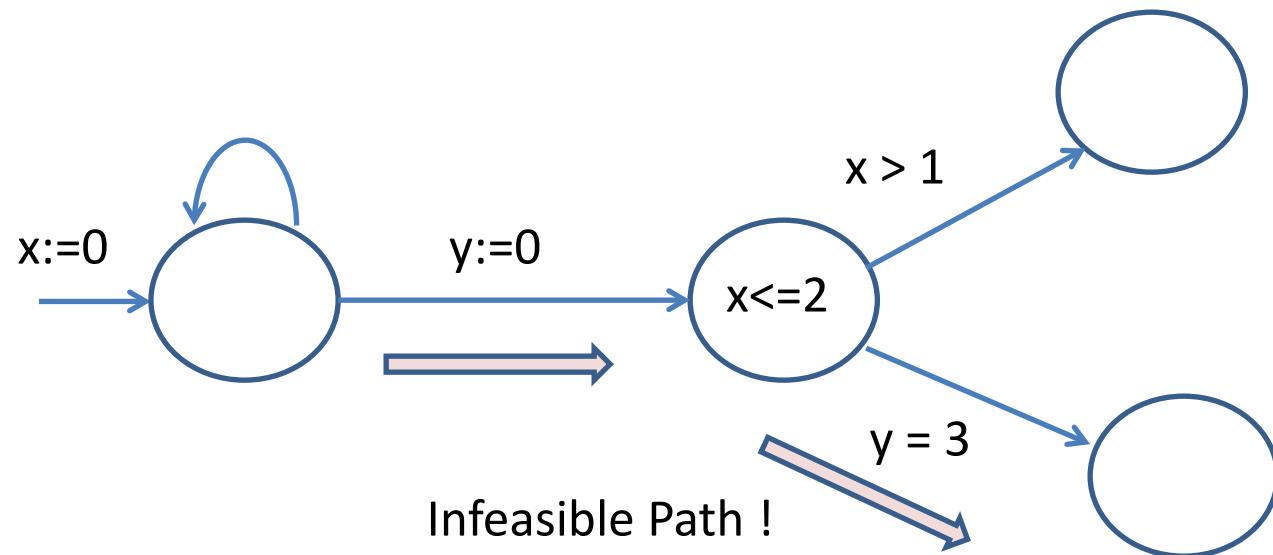


Timing Analysis



- How to adapt algorithms for searching through the state-space of a model in presence of clock variables and timing constraints?
- Application: Formal analysis of timing-based coordination and communication protocols, and many other systems ...
- Must handle the space of clock valuations symbolically!
- Popular model checker: Uppaal (according to Rajeev ☺)

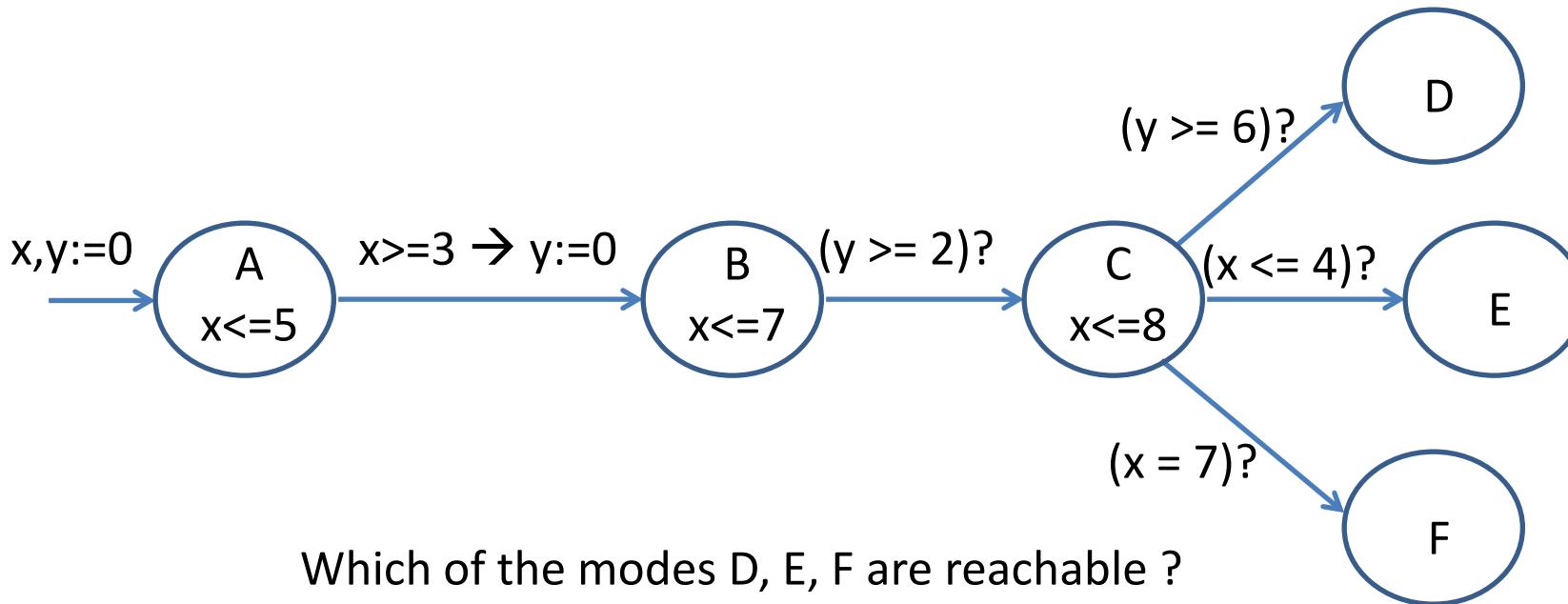
Timing Analysis Example



Timed Automata

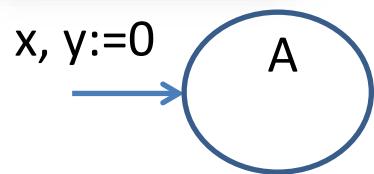
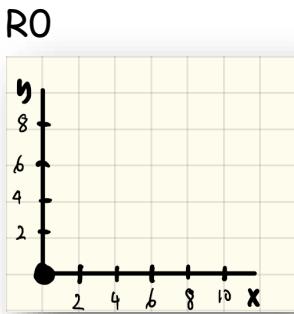
- Motivation: When is exact analysis of timing constraints possible?
- A timed process TP is a timed automaton if for every clock variable x
 - Assignments to x in the description of TP are of the form $x:=0$
 - An atomic expression involving x in the description of TP (in clock-invariants or in guards) must of the form " $x \sim k$ ", where k is a constant and \sim is a comparison operation ($=, \leq, >, <, \geq$)
- In such a model, one can express constant lower and upper bounds on timing delays
 - Closed under parallel composition: If TP1 and TP2 are timed automata then $TP1 \parallel TP2$ is also a timed automaton
- Finite-state timed automaton: A timed automaton where all variables other than clock variables have finite types (e.g. Boolean, enumerated)
 - State-space is still infinite due to clock variables, but verification is solvable exactly

Timing Analysis Example



Requires propagation of the reachable combinations of x and y symbolically

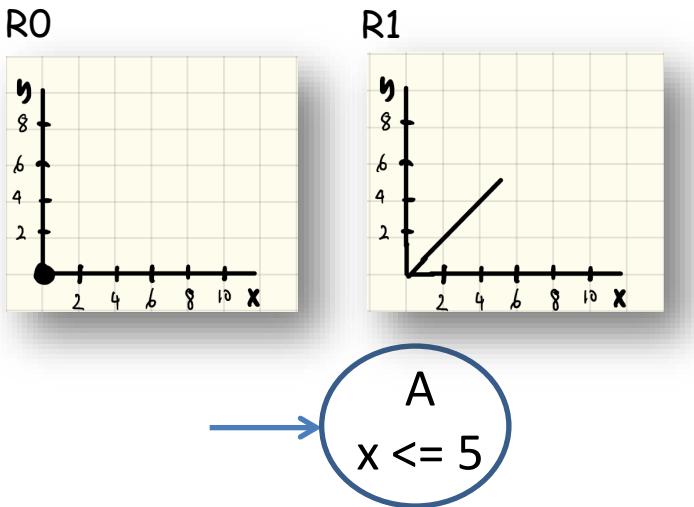
Timing Analysis Example



Initial set of clock-valuations: $x = 0$ & $y= 0$

Clock-zone: Uniform representation of constraints that arise during analysis

Timing Analysis Example

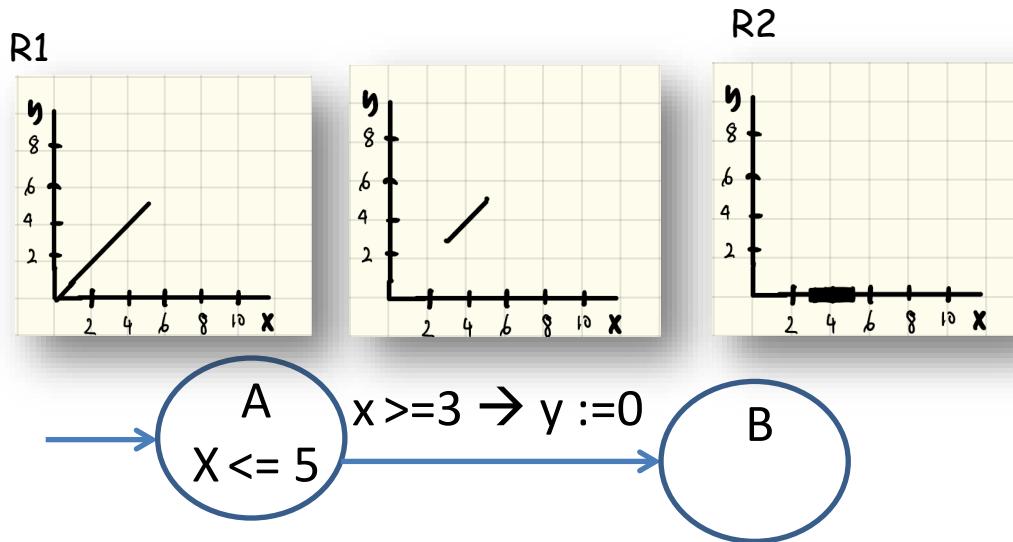


Starting from a state in R_0 , as time elapses, which clock-valuations are reachable ?

During a timed transition, values of all clocks increase.

Compute effect of timed transitions observing the clock-invariant
note that different clocks increase with the same speed (diagonals)

Timing Analysis Example



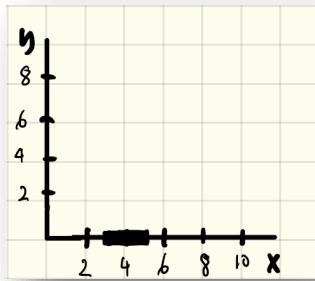
Desired clock-zone R_2 : What are set of clock-valuations upon entry to B ?

Intersect guard $3 \leq x$ with the clock-zone R_1

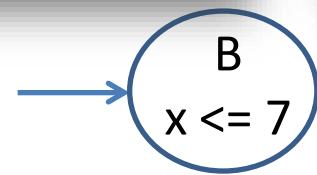
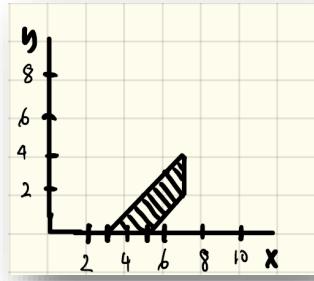
Capture the effect of assignment $y := 0$ (project on x-axis)

Timing Analysis Example

R2



R3

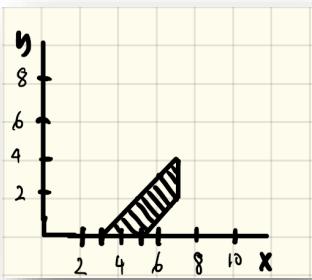


Starting from a state in R_2 , as time elapses, which clock-valuations are reachable ?

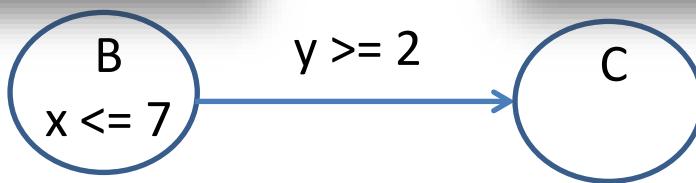
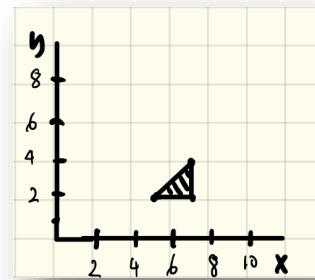


Timing Analysis Example

R3



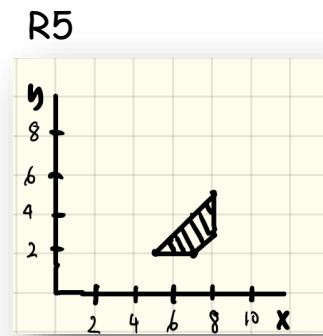
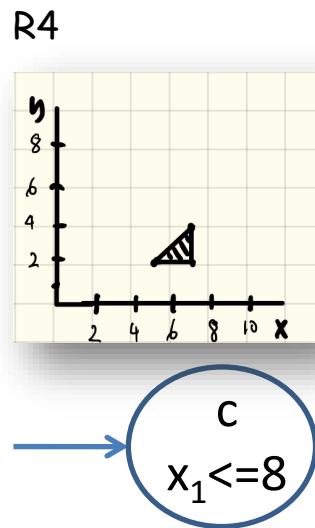
R4



Compute set of clock-valuations upon entry to C

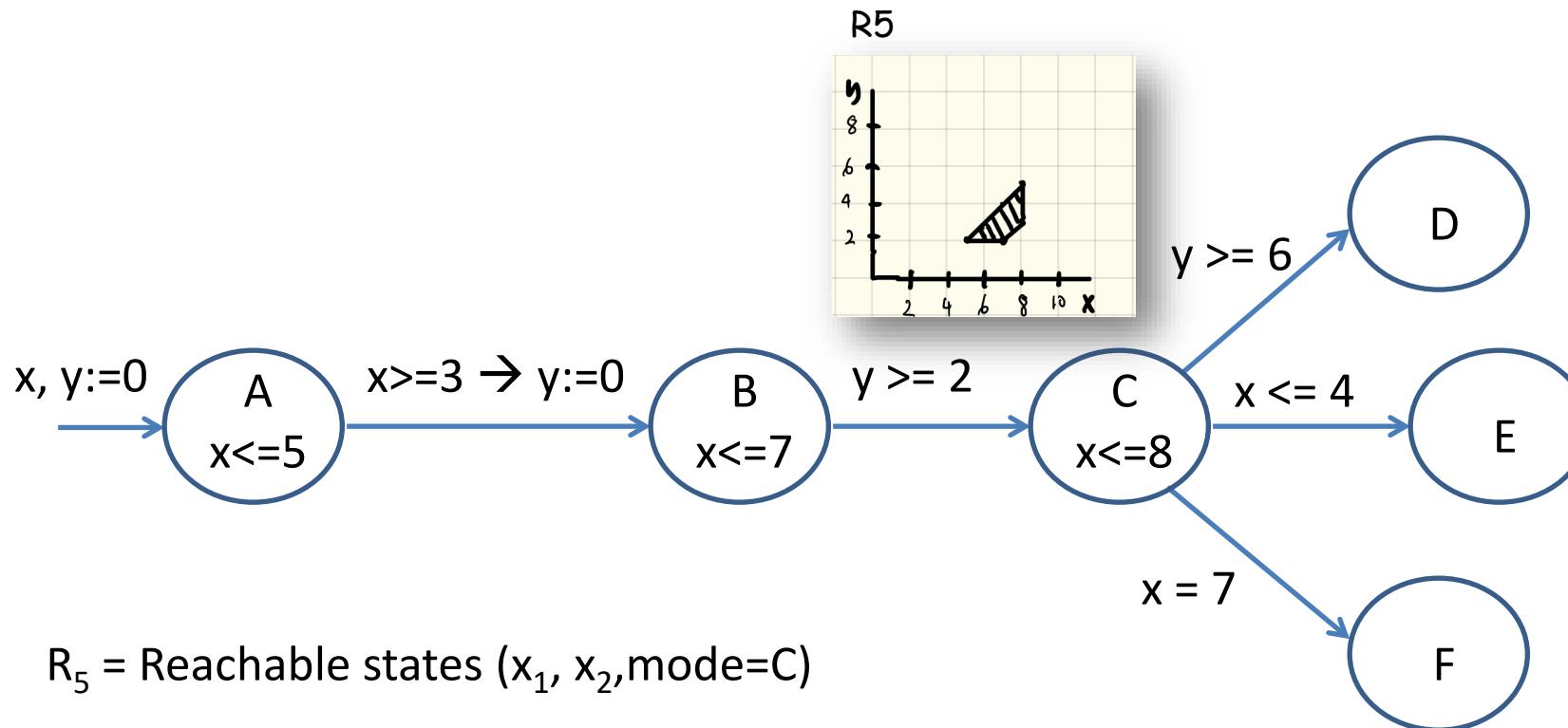
E.g. Conjoin $y >= 2$ to R3.

Timing Analysis Example



Compute set of clock-valuations reachable due to timed transitions in mode C

Timing Analysis Example



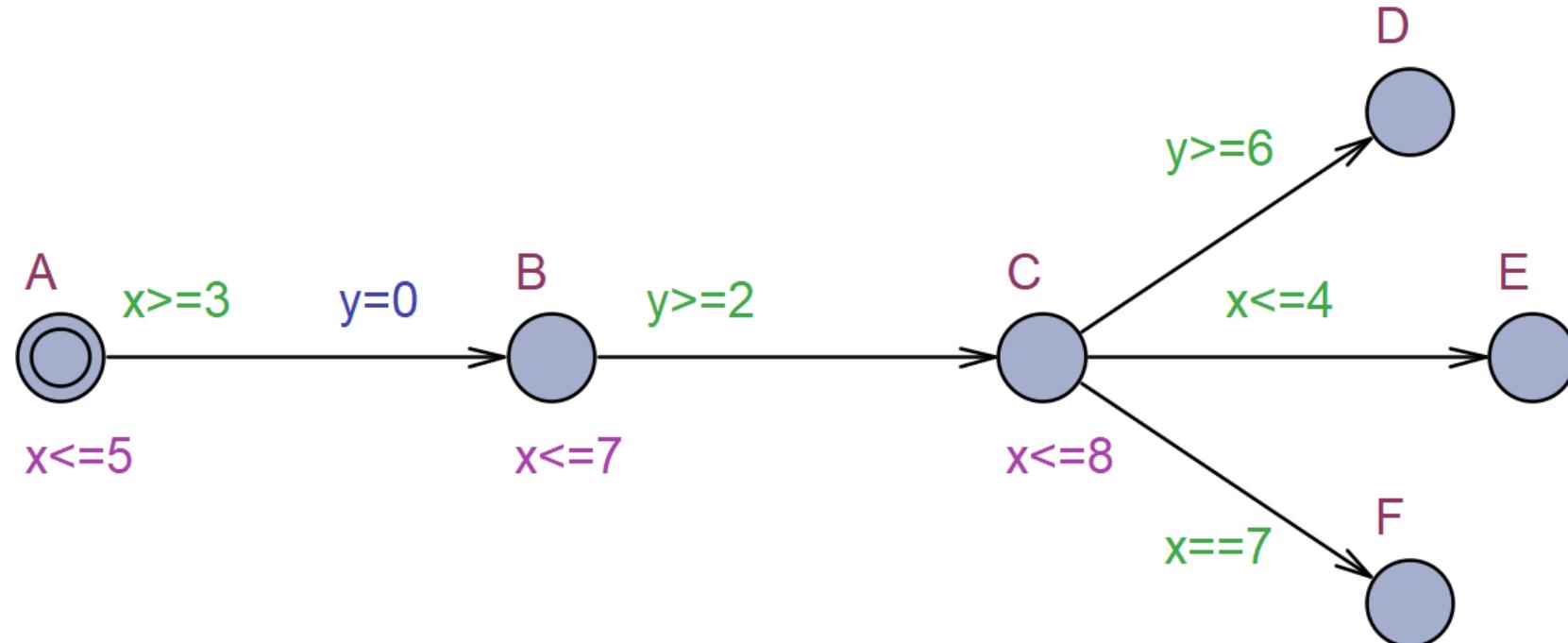
$R_5 = \text{Reachable states } (x_1, x_2, \text{mode}=\text{C})$

Intersection of R_5 and $y \geq 6$ unsatisfiable; means mode D not reachable

Intersection of R_5 and $x \leq 4$ unsatisfiable; means mode E not reachable

Intersection of R_5 and $x=7$ satisfiable; means mode F is reachable

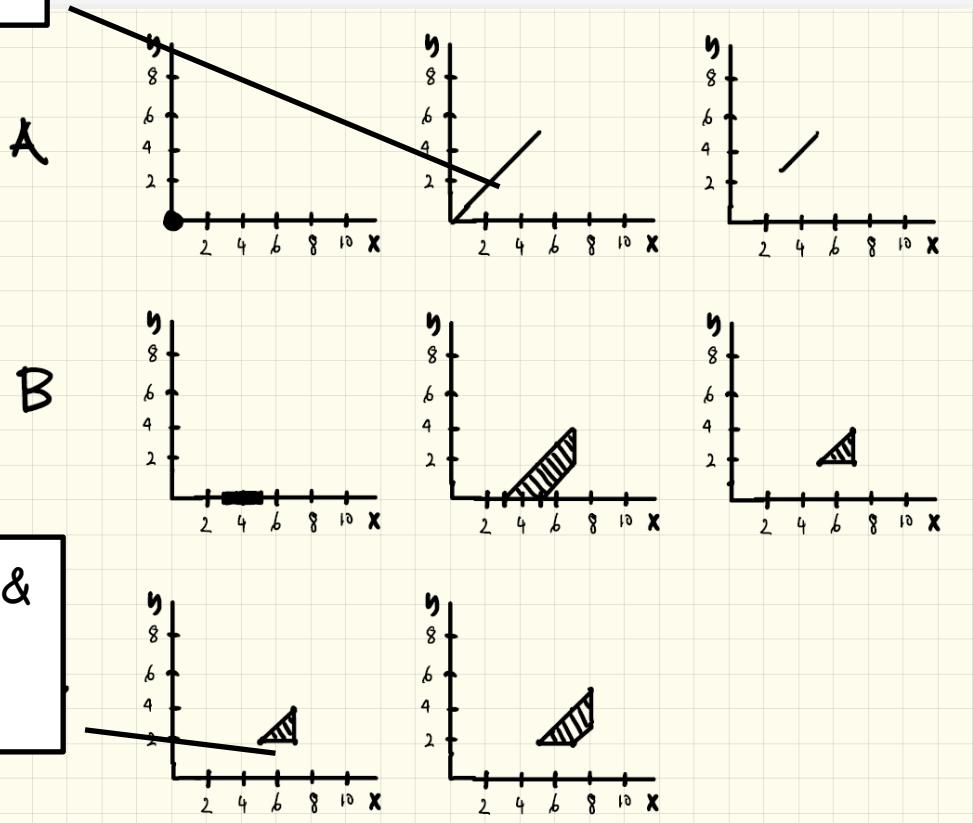
Timing Analysis Example in UPPAAL



$$0 \leq x-y \leq 0 \text{ &} \\ x \leq 5$$

Symbolic Data Structures for Zones

$$3 \leq x-y \leq 5 \text{ &} \\ x \leq 7 \text{ &} \\ y \geq 2$$



Zones

- = conjunction of constraints
 $x \sim n$ and $x-y \sim n$
- = Difference Constraints (Belmann)

Principles of Cyber-Physical Systems: Timed Model

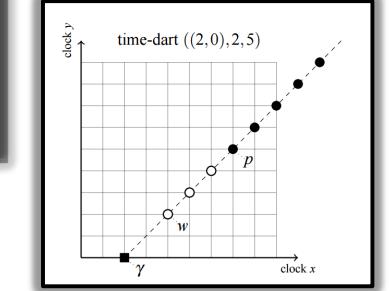
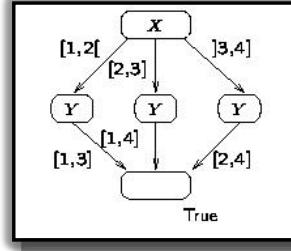
DBMs [d889, mkl949, lm97CS, bengtsson02] are efficient data structures to represent time. They are used in UPPAAL [lp97, lp02, d02] as the core data structure to represent time. The delay, or horizon, allows (part), general update, different ext-reduces functions, etc.

References

- [d889] David L. Dill, Timing Assumptions and Verification of Finite-State Concurrent Systems, Ph.D. thesis, U. California Berkeley, 1988.
- [mkl949] Tomas Gerhard Raskin, Representing and Modeling Digital Circuits, Ph.D. thesis, U. California Berkeley, 1994.
- [lm97CS] Kim G. Larsen, Paul Petersen, and Wang Yi, Model-Checking for Real-Time Systems, LNCS 945 pages 62-88.
- [bengtsson02] John Bengtsson, Clocks, DBMs, and States in Timed systems, Ph.D. thesis, Uppsala University, 2002.

Datastructures for Zones

- Difference Bounded Matrices (DBMs)
- Minimal Constraint Form [RTSS97]
- Clock Difference Diagrams [CAV99]
- Timed Darts [NASA FM 2014]



TU Graz, May 2017

Kim Larsen [22]



Efficient Representations and Operations
NEXT LECTURE

Principles of Cyber-Physical Systems

UPPAAL

Modelling, Specification and Verification

Instructors: Kim G. Larsen, Marco Muniz, Max Tschaikowski
{kgl,Muniz,Tschaikowski}@cs.aau.dk

Slides courtesy: Rajeev Alur
alur@cis.upenn.edu



Agenda

- UPPAAL
 - History and Demos
- Train Gate Example
 - Modelling Formalism
(Networks of Extended Timed Automata)
 - Specification Formalism
- UPPAAL Engine
- UPPAAL Options



Quantitative aspects
(timing, energy, bandwidth, memory,...)
crucial for several embedded systems

Main Readings (+Prin. of CPS chapter 7)



Chapter 1 A First Introduction to Uppaal

Frits Vaandrager

Abstract This chapter provides a first introduction to the use of the model checking tool Uppaal. Uppaal is an integrated tool environment that allows users to model the behavior of systems in terms of states and transitions between states, and to simulate and analyze the resulting models. Uppaal can also handle real-time issues, that is, the timing of transitions. Using an example of a jobshop, we explain in a step by step manner how one can make a simple Uppaal model, simulate its behavior and analyze its properties. This introduction is targeted at a broad audience, ranging from high school students to software engineers and researchers. We only require elementary knowledge of programming and mathematics. Although a rich theory of model checking has been developed over the last decades, which includes both clever algorithms and deep mathematics, this introduction focuses entirely on the



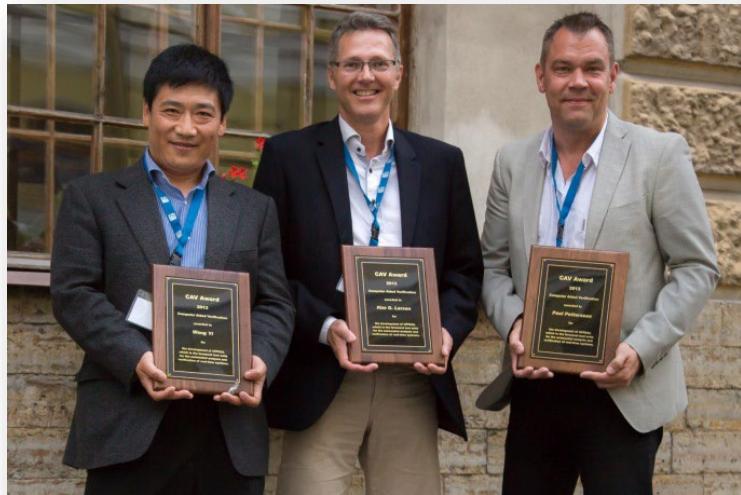
Chapter 1 More Features in UPPAAL

Alexandre David, Kim G. Larsen

Abstract Following the introduction to the model checking tool UPPAAL of the previous chapter, this chapter presents a number of additional modeling and verification features offered by the tool. These features include in particular a C-like imperative language with user-defined types and functions, allowing for readable and compact models with reusable updates of discrete variables. Using an example of a Train Gate, we demonstrate the use(fulness) of these features. Also, the chapter presents the full query language of UPPAAL covering both safety, liveness and time-bounded liveness properties, again illustrated using the Train Gate example. Finally, directions are given on modelling choices and use of verification options that may improve time- and/or space-performance of the UPPAAL verifier



Origin of UPPAAL



TAU

CCS & Modal Transition Systems
Refinements
Modal Mu-Calculus
Explicit State Representation
Prolog

1995

UPPAAL

Timed Automata
TCTL
Zones
C++ & Java

EPSILON

TCCS
Timed Refinements
Timed Mu-Calculus
Regions
Prolog<

2007

UP4ALL

2013

2016

2018

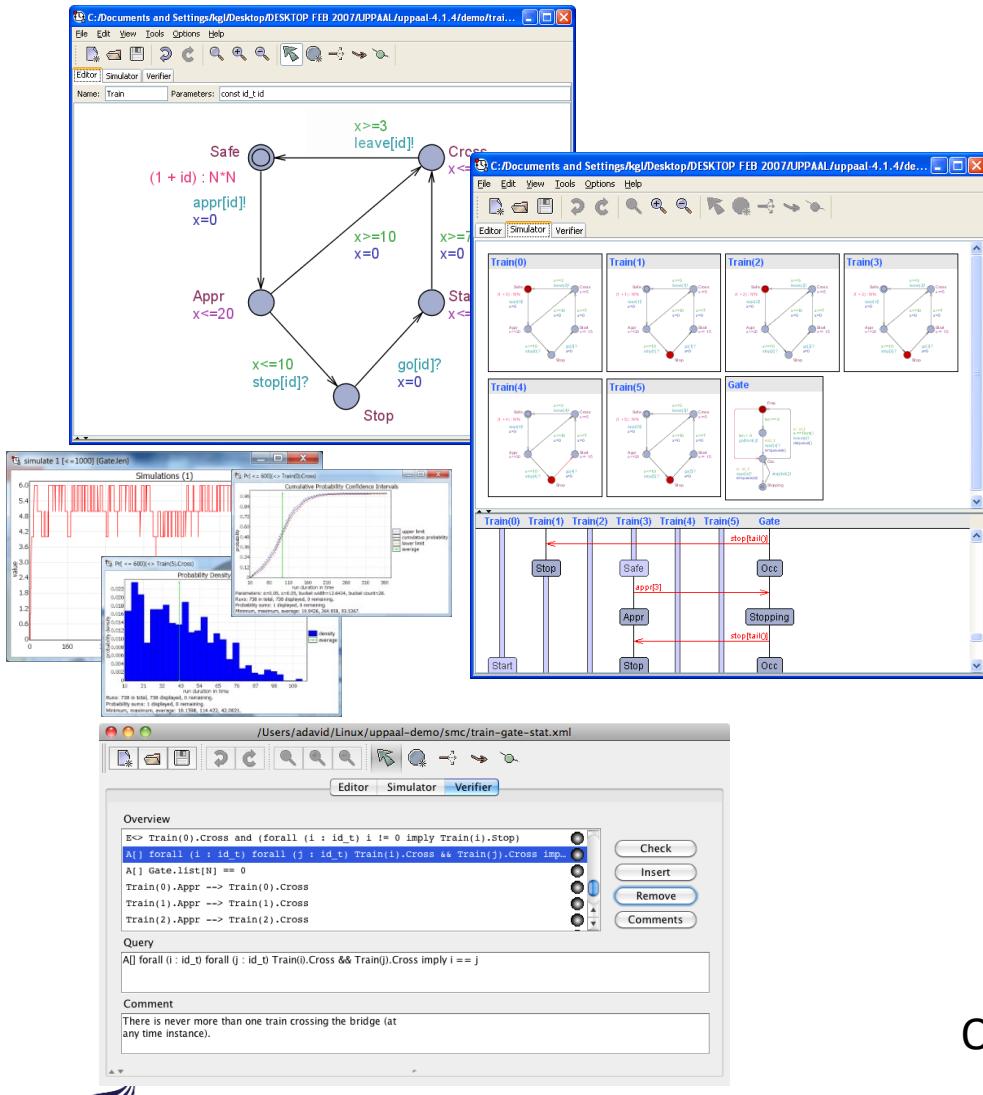
CAV Award

Grundfos Prize

ATS VeriAll



UPPAAL Tool Suit



Contributors

@UPPsala

- Wang Yi
- Paul Pettersson
- John Håkansson
- Anders Hessel
- Pavel Krcal
- Leonid Mokrushin
- Shi Xiaochun



@AALborg

- Marius Mikucionis
- Peter Gjøl Jensen
- Kenneth Y Jørgensen
- Danny Poulsen
- Marco Muniz
- Ulrik Nyman
- Arne Skou
- Brian Nielsen
- Alexandre David
- Gerd Behrman
- Jacob Illum Rasmussen
- Kim G Larsen

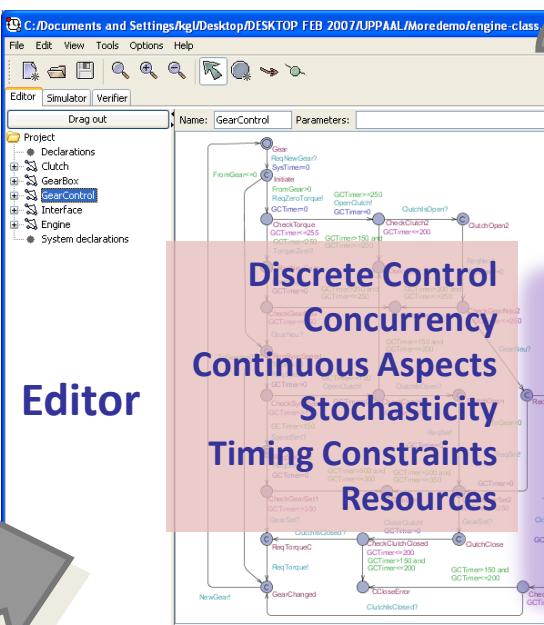


@Elsewhere

- Emmanuel Fleury, Didier Lime, Johan Bengtsson, Fredrik Larsson, Kåre J Kristoffersen, Tobias Amnell, Thomas Hune, Oliver Möller, Elena Fersman, Carsten Weise, David Griffioen, Ansgar Fehnker, Frits Vandraager, Theo Ruys, Pedro D'Argenio, J-P Katoen, Jan Tretmans, Judi Romijn, Ed Brinksma, Martijn Hendriks, Klaus Havelund, Franck Cassez, Magnus Lindahl, Francois Laroussinie, Patricia Bouyer, Augusto Burgueno, H. Bowmann, D. Latella, M. Massink, G. Faconti, Kristina Lundqvist, Lars Asplund, Justin Pearson...

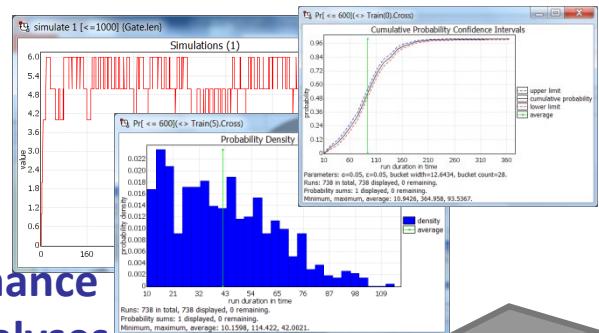


UPPAAL Model Checker

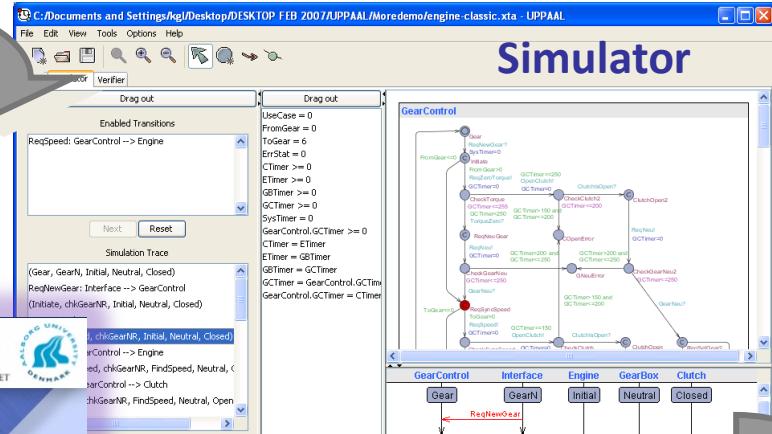


Editor

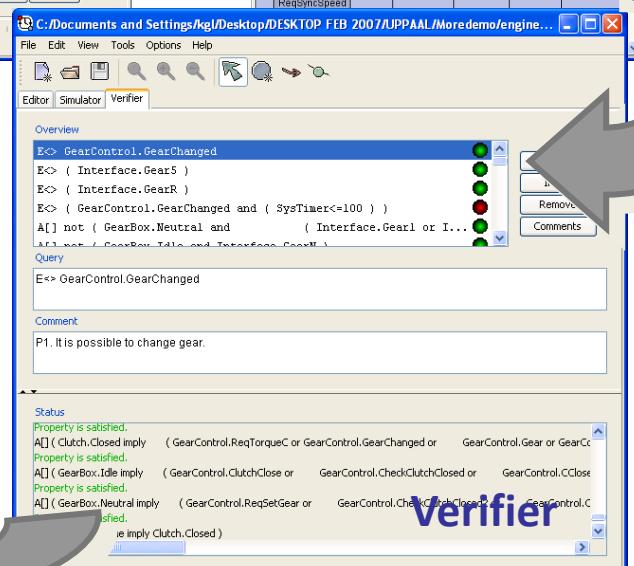
Discrete Control
Concurrency
Continuous Aspects
Stochasticity
Timing Constraints
Resources



Performance
Analyses



Simulator



Verifier

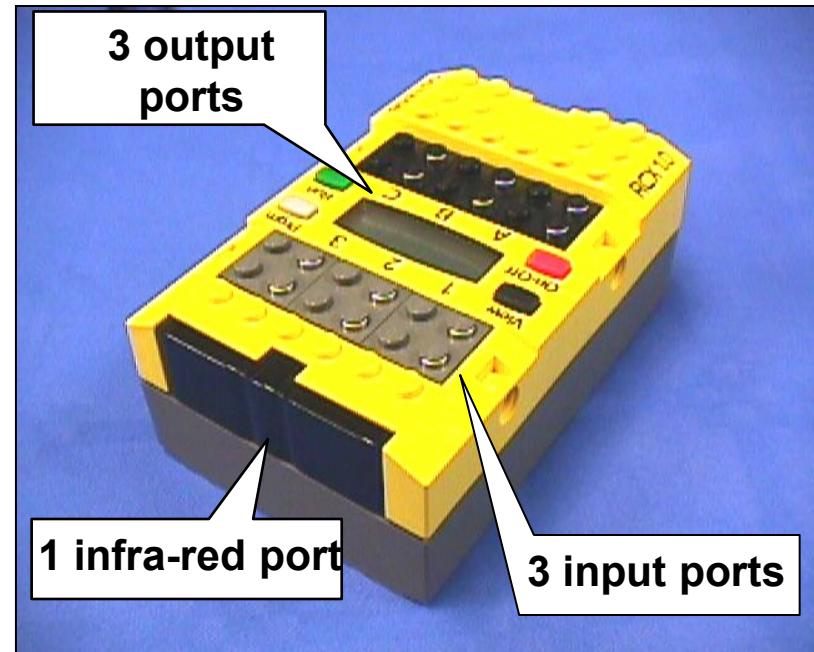


Brick Sorting



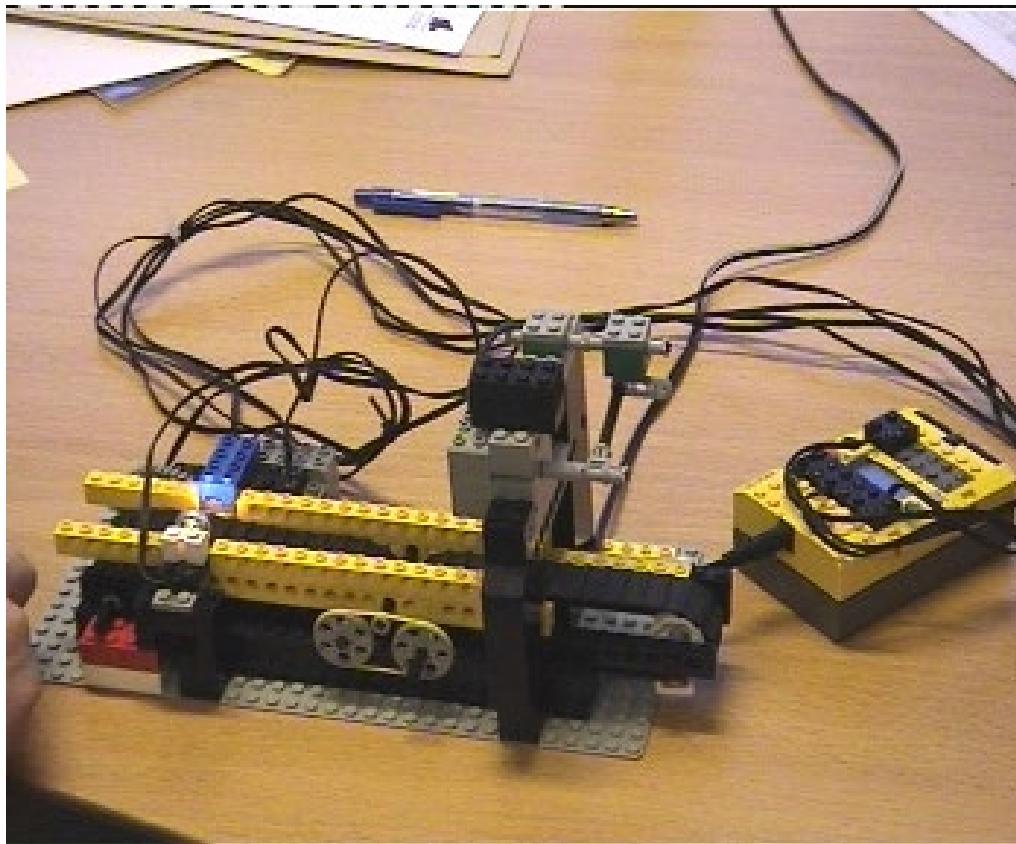
LEGO Mindstorms/RCX

- Sensors: temperature, light, rotation, pressure.
- Actuators: motors, lamps,
- Virtual machine:
 - 10 tasks, 4 timers, 16 integers.
- Several Programming Languages:
 - NotQuiteC, Mindstorm, Robotics, legOS, etc.



A Real Real Timed System

The Plant
Conveyor Belt
&
Bricks



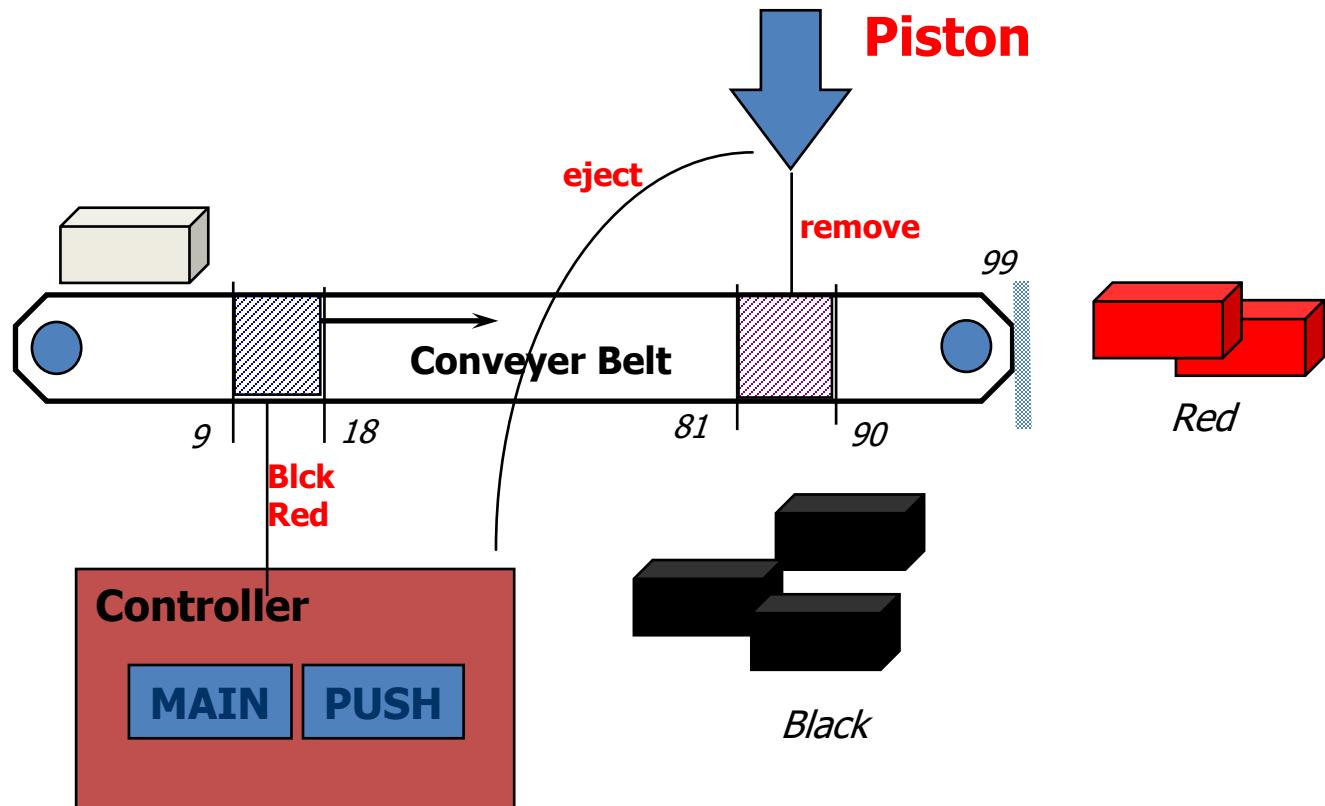
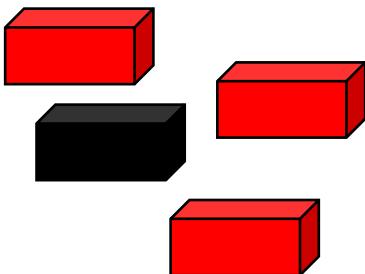
**Controller
Program**
LEGO MINDSTORM

First UPPAAL model

Sorting of Lego Boxes

Ken Tindell

Boxes



Exercise: Design **Controller** so that **black** boxes are being pushed out

NQC programs

```
task MAIN{
    DELAY=75;
    LIGHT_LEVEL=35;
    active=0;
    Sensor(IN_1, IN_LIGHT);
    Fwd(OUT_A,1);
    Display(1);

    start PUSH;

    while(true) {

        wait(IN_1<=LIGHT_LEVEL);
        ClearTimer(1);
        active=1;
        PlaySound(1);

        wait(IN_1>LIGHT_LEVEL);
    }
}
```

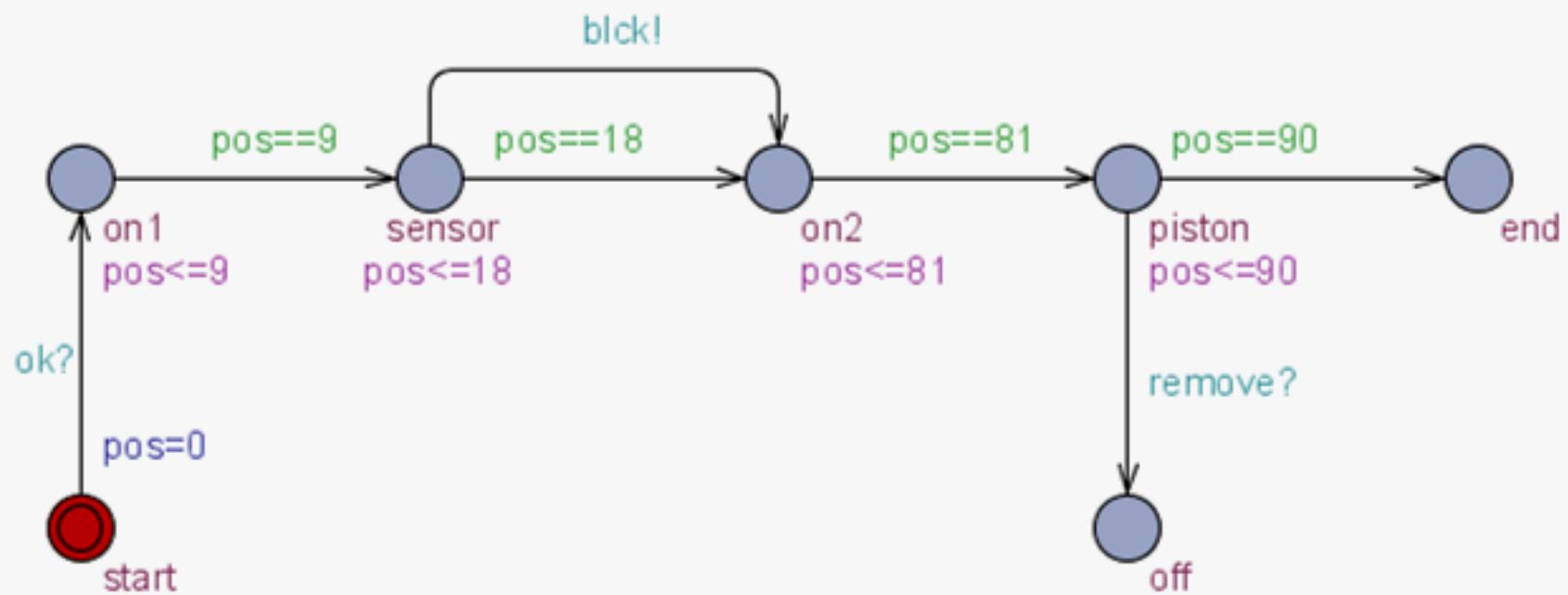
```
int active;
int DELAY;
int LIGHT_LEVEL;
```

```
task PUSH{
    while(true) {
        wait(Timer(1)>DELAY && active==1);
        active=0;
        Rev(OUT_C,1);
        Sleep(8);
        Fwd(OUT_C,1);
        Sleep(12);
        Off(OUT_C);
    }
}
```

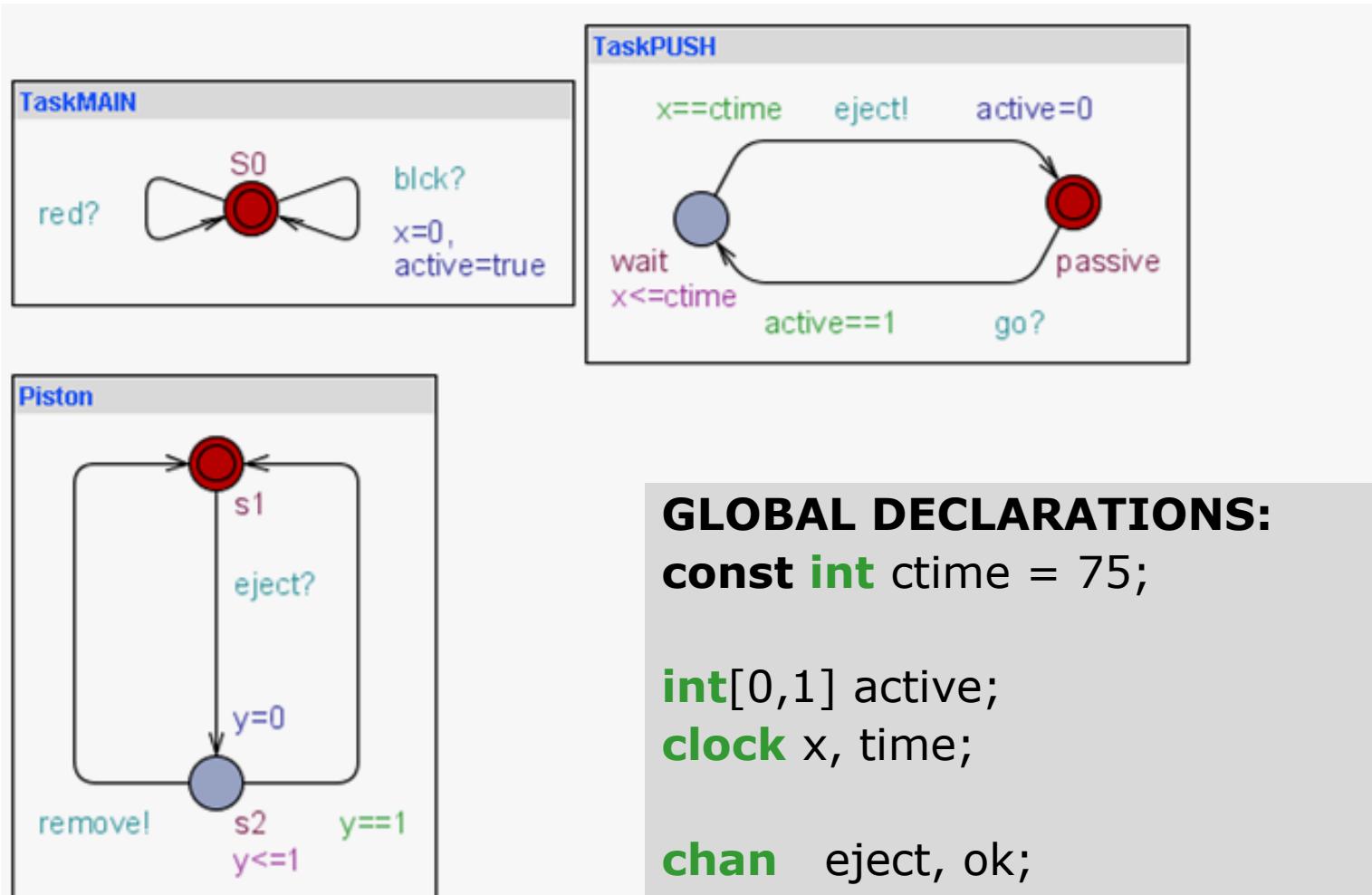


A Black Brick

B1



Control Tasks & Piston



GLOBAL DECLARATIONS:

const int ctime = 75;

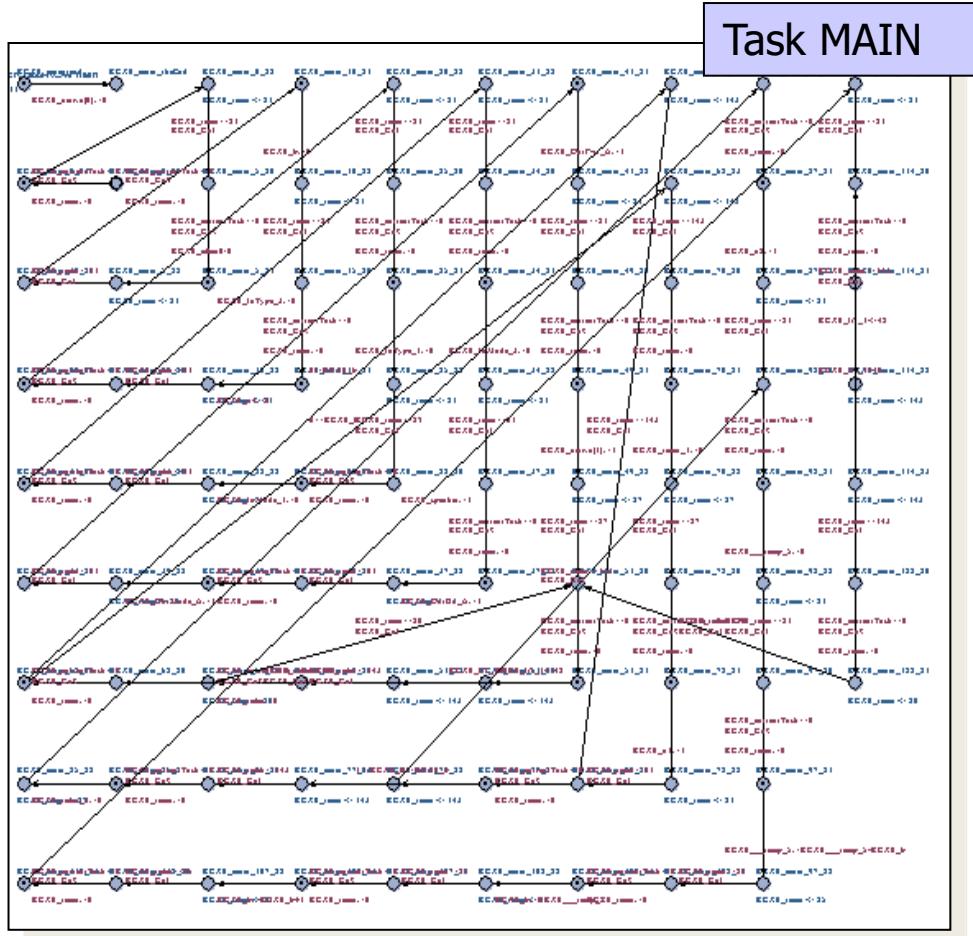
int[0,1] active;
clock x, time;

chan eject, ok;
urgent chan blk, red, remove, go;



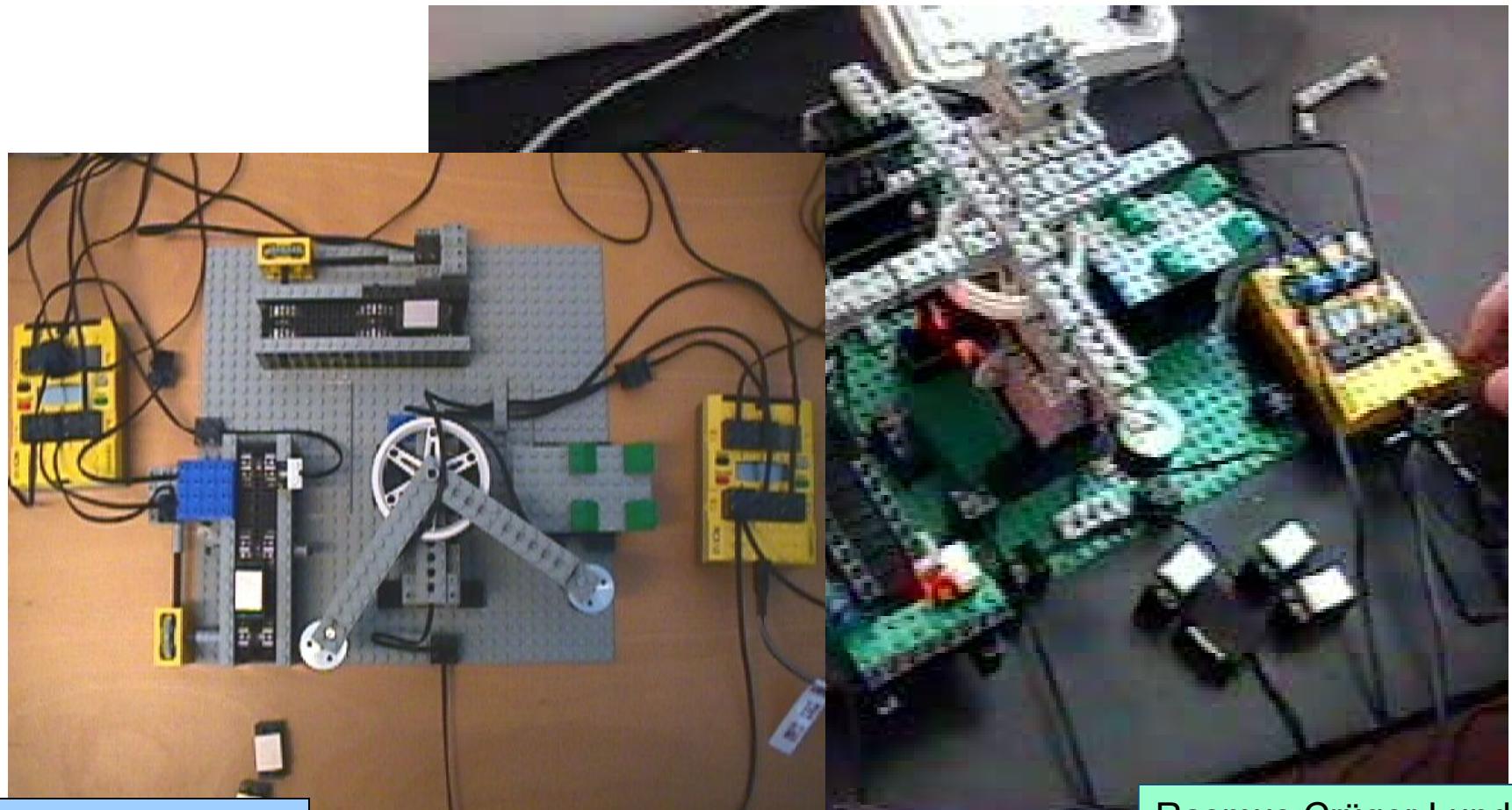
From RCX to UPPAAL - and back

- Model includes Round-Robin Scheduler.
 - Compilation of RCX tasks into TA models.
 - Presented at ECRTS 2000 in Stockholm.
 - From UPPAAL to RCX:
Martijn Hendriks.



The Production Cell in LEGO

Course at DTU, Copenhagen



Production Cell

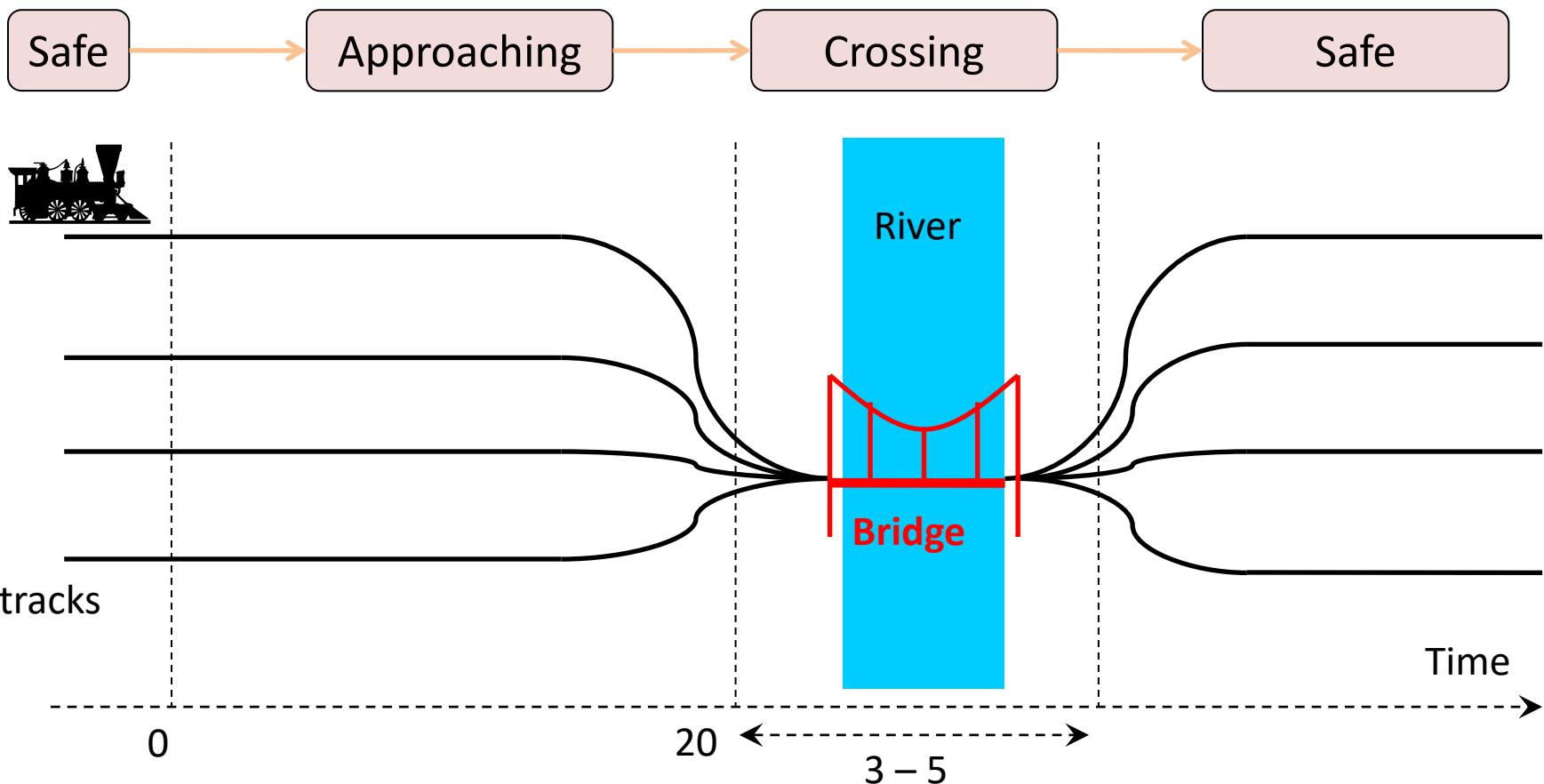
Rasmus Crüger Lund
Simon Tune Riemanni



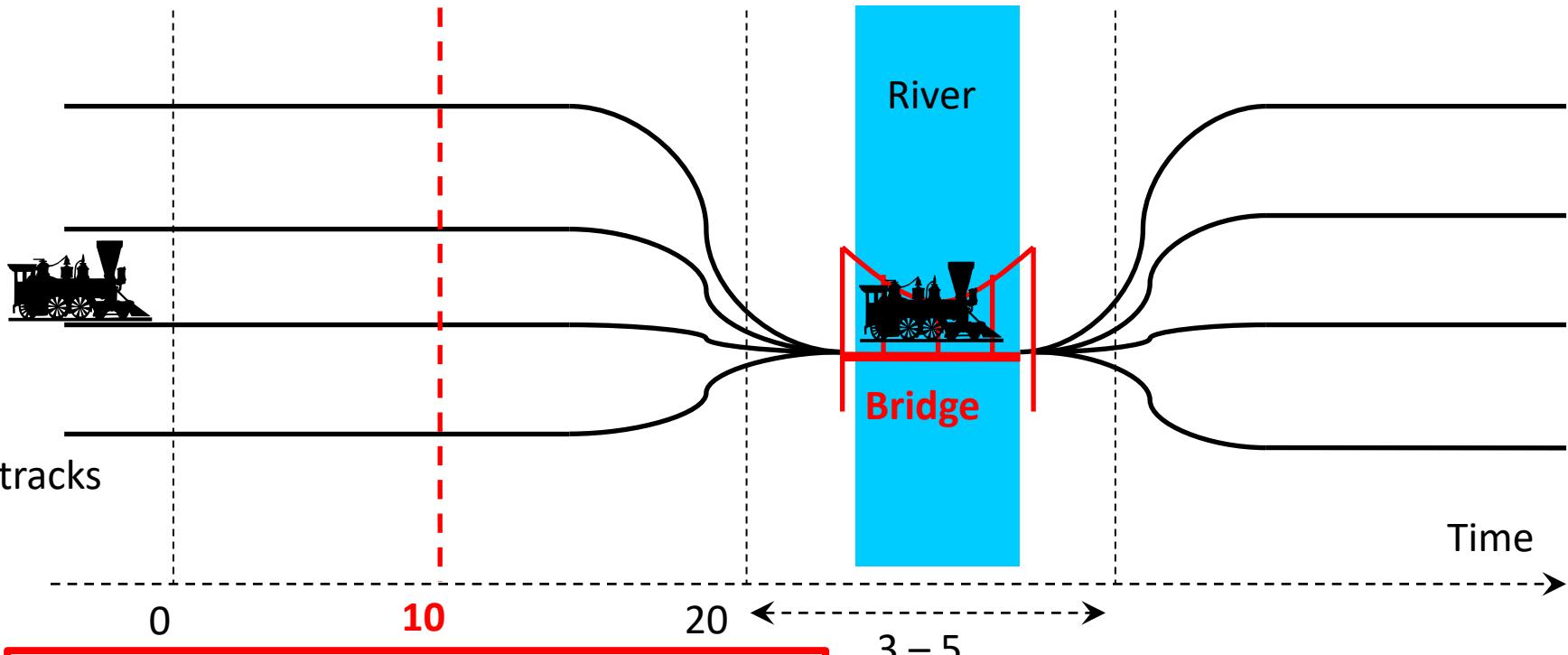
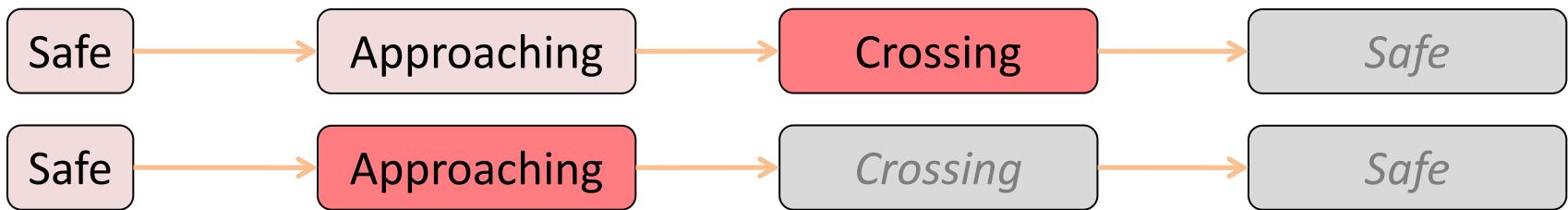
Train Crossing



Train Crossing

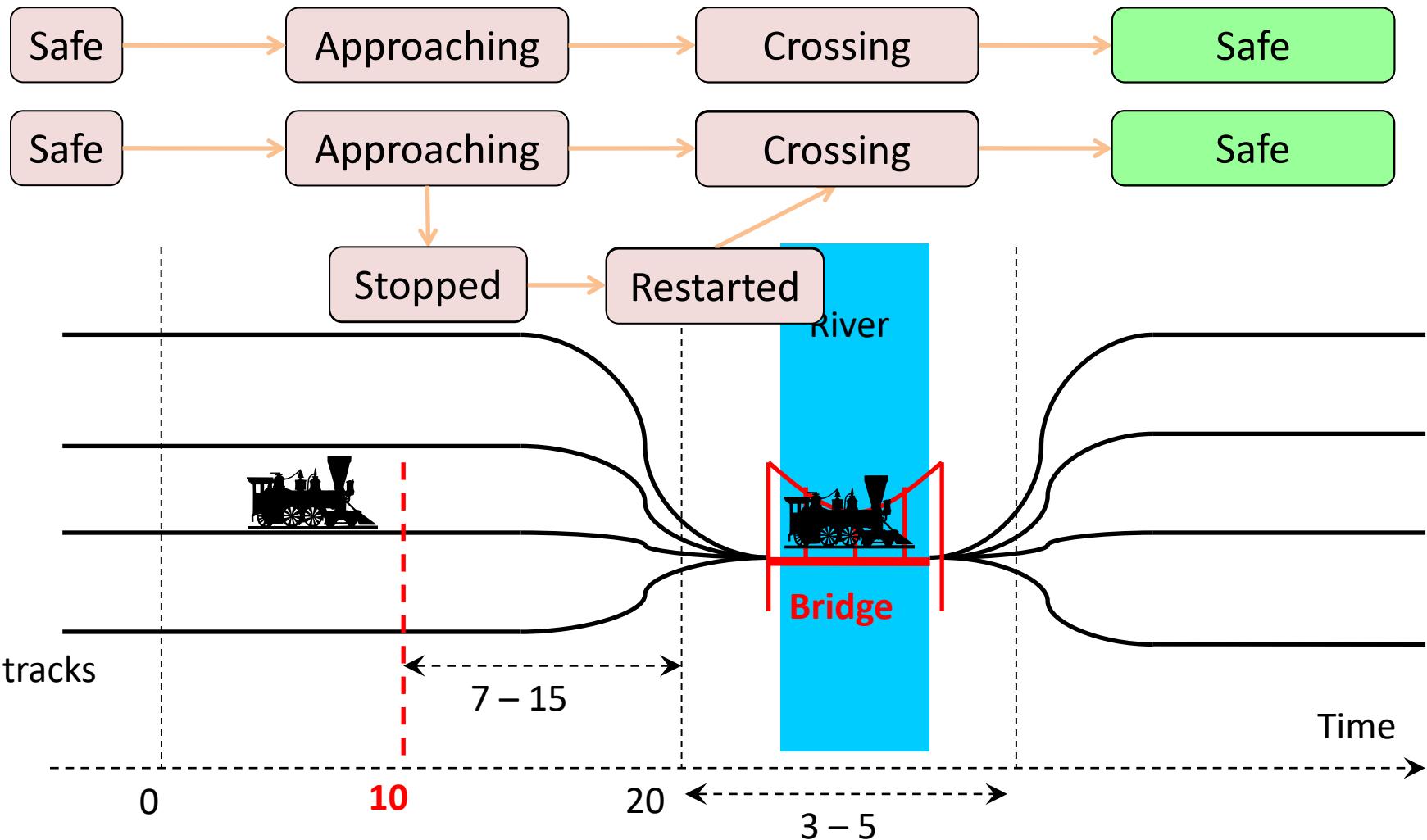


Train Crossing

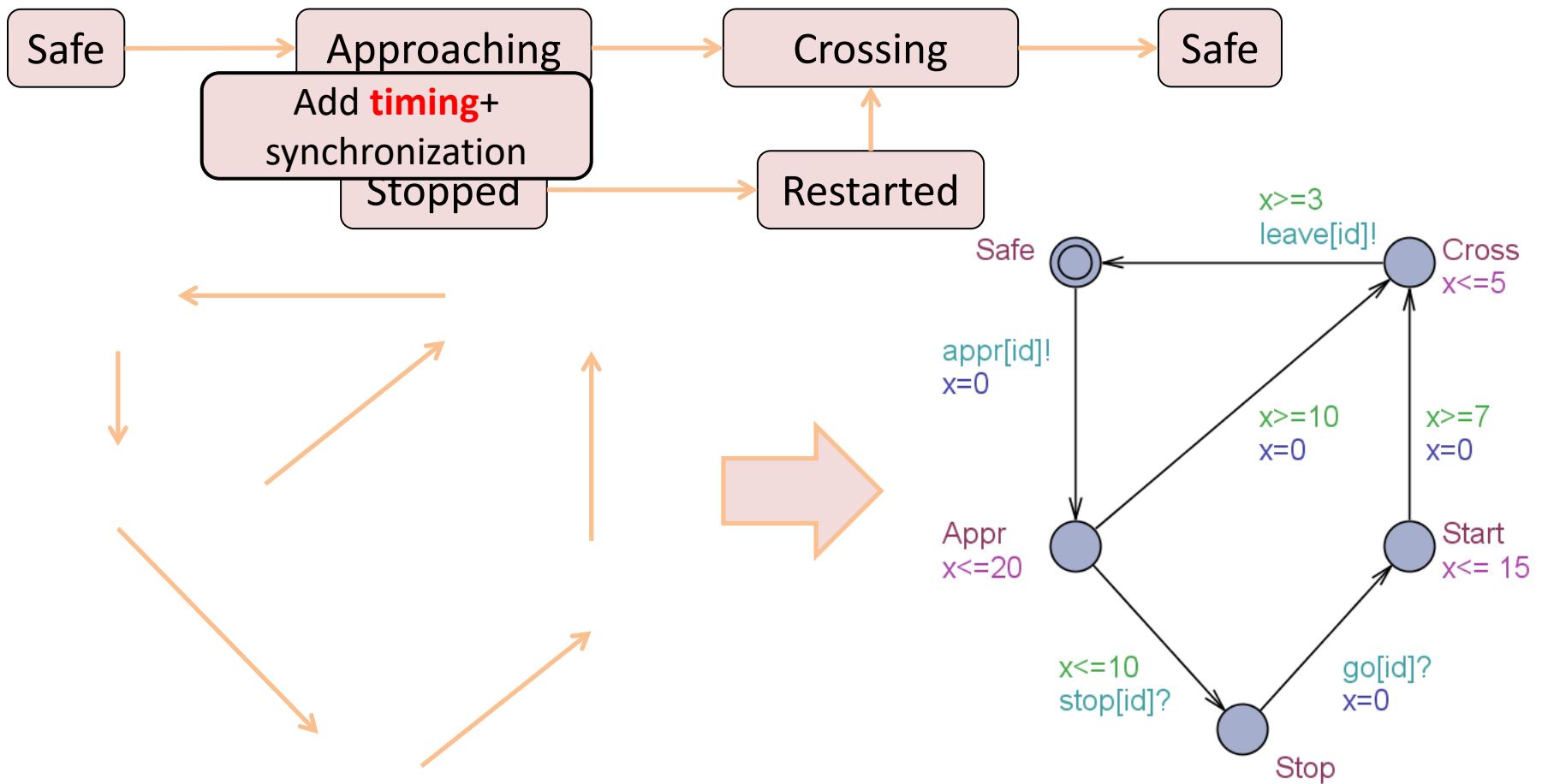


Stop the train while it still stoppable!

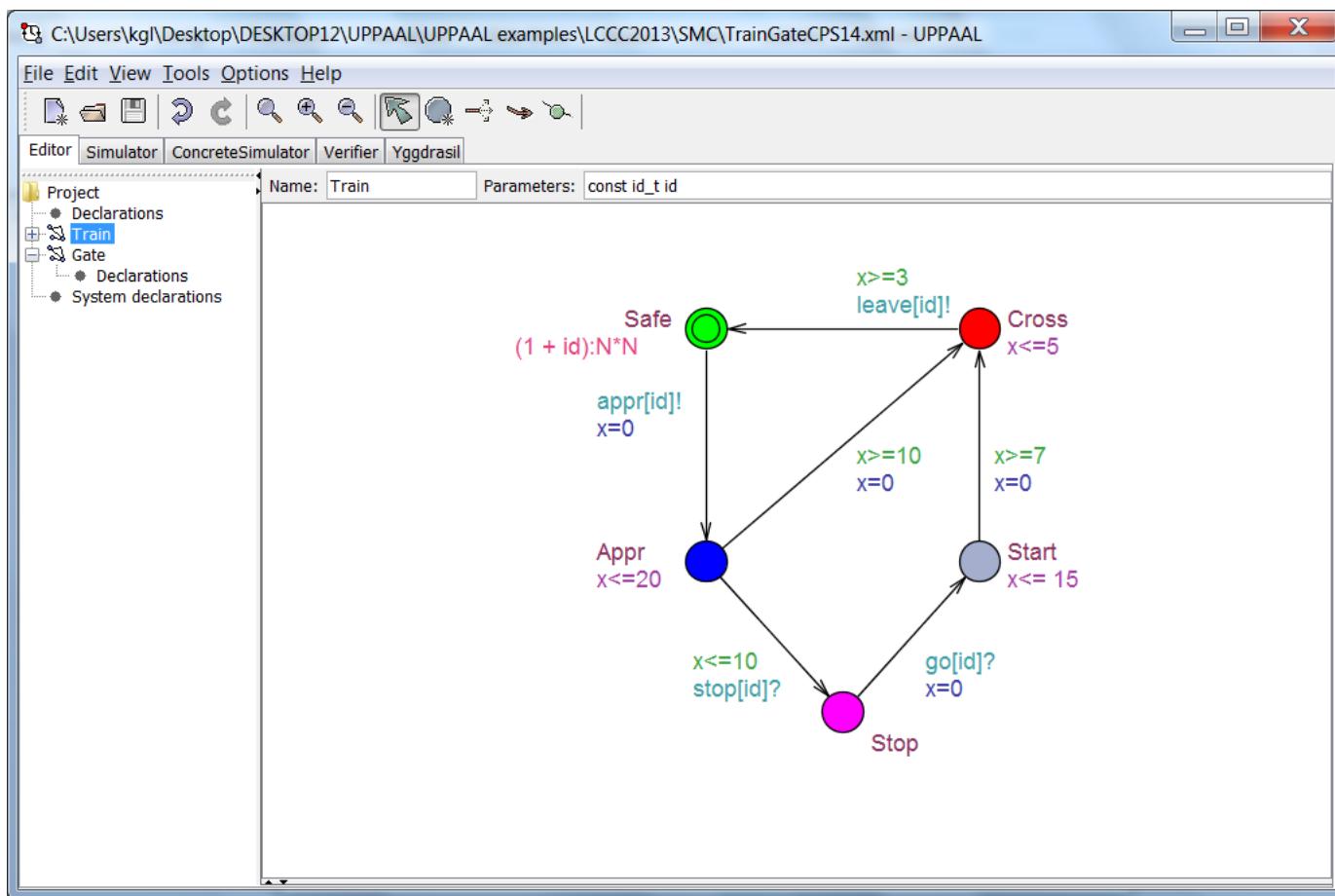
Train Crossing



Train Crossing



Demo



Logical Specifications

- Validation Properties

- Possibly:

$$E \leftrightarrow P$$

- Safety Properties

- Invariant: $A[] P$

- Pos. Inv.: $E[] P$

- Liveness Properties

- Eventually: $A \leftrightarrow P$

- Leadsto: $P \rightarrow Q$

- Bounded Liveness

- Leads to within: $P \rightarrow_{\leq t} Q$

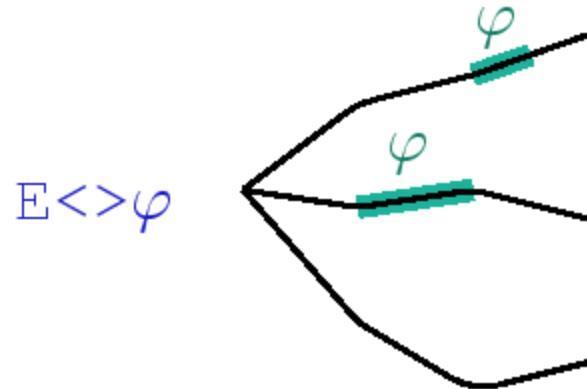
The expressions P and Q must be type safe, side effect free, and evaluate to a boolean.

Only references to integer variables, constants, clocks, and locations are allowed (and arrays of these).



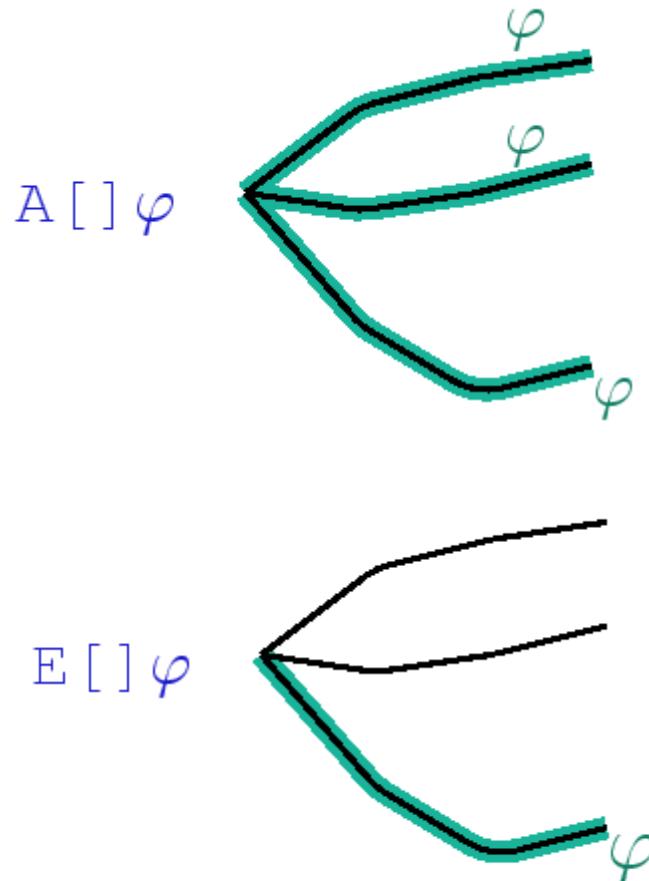
Logical Specifications

- Validation Properties
 - Possibly: $E \leftrightarrow P$
- Safety Properties
 - Invariant: $A[] P$
 - Pos. Inv.: $E[] P$
- Liveness Properties
 - Eventually: $A \diamond P$
 - Leadsto: $P \rightarrow Q$
- Bounded Liveness
 - Leads to within: $P \rightarrow_{\leq t} Q$



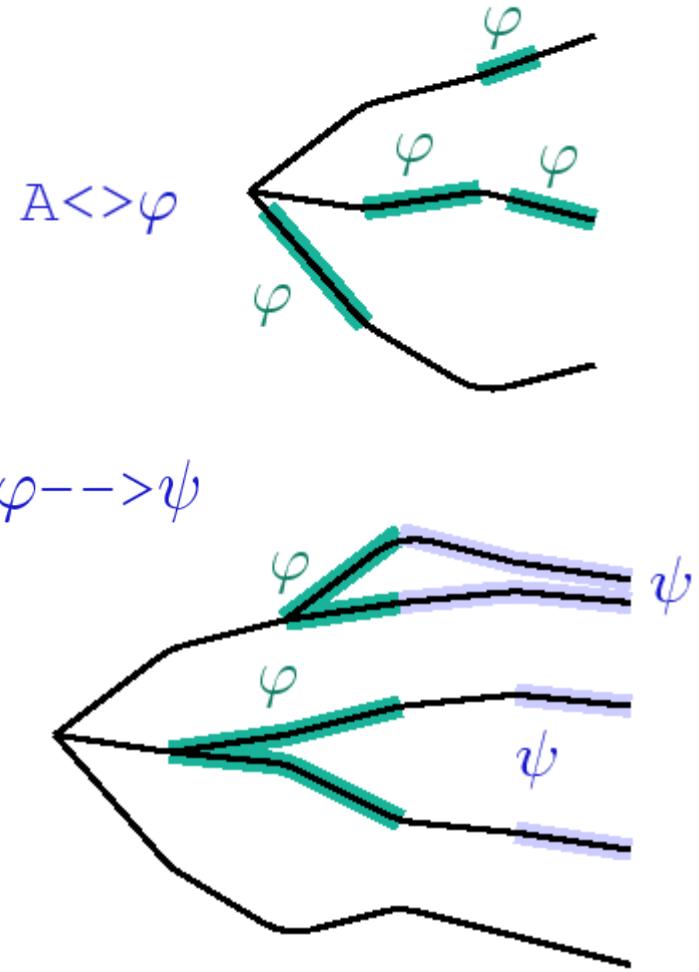
Logical Specifications

- Validation Properties
 - Possibly: $E \leftrightarrow P$
- Safety Properties
 - Invariant: $A[] P$
 - Pos. Inv.: $E[] P$
- Liveness Properties
 - Eventually: $A \leftrightarrow P$
 - Leadsto: $P \rightarrow Q$
- Bounded Liveness
 - Leads to within: $P \rightarrow_{\leq t} Q$



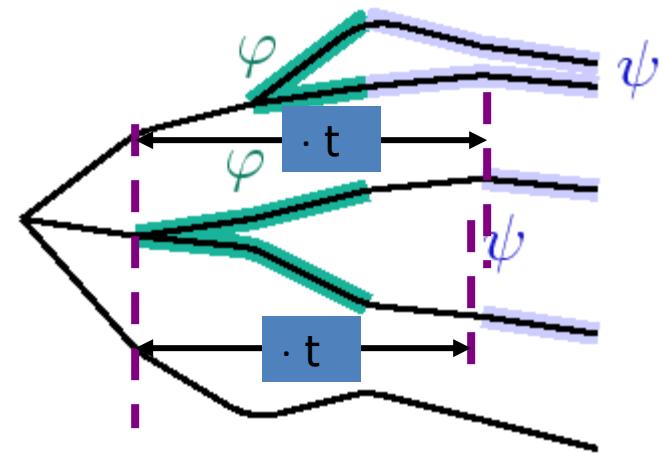
Logical Specifications

- Validation Properties
 - Possibly: $E \leftrightarrow P$
- Safety Properties
 - Invariant: $A[] P$
 - Pos. Inv.: $E[] P$
- Liveness Properties
 - Eventually: $A \leftrightarrow P$
 - Leadsto: $P \rightarrow Q$
- Bounded Liveness
 - Leads to within: $P \rightarrow_{\leq t} Q$

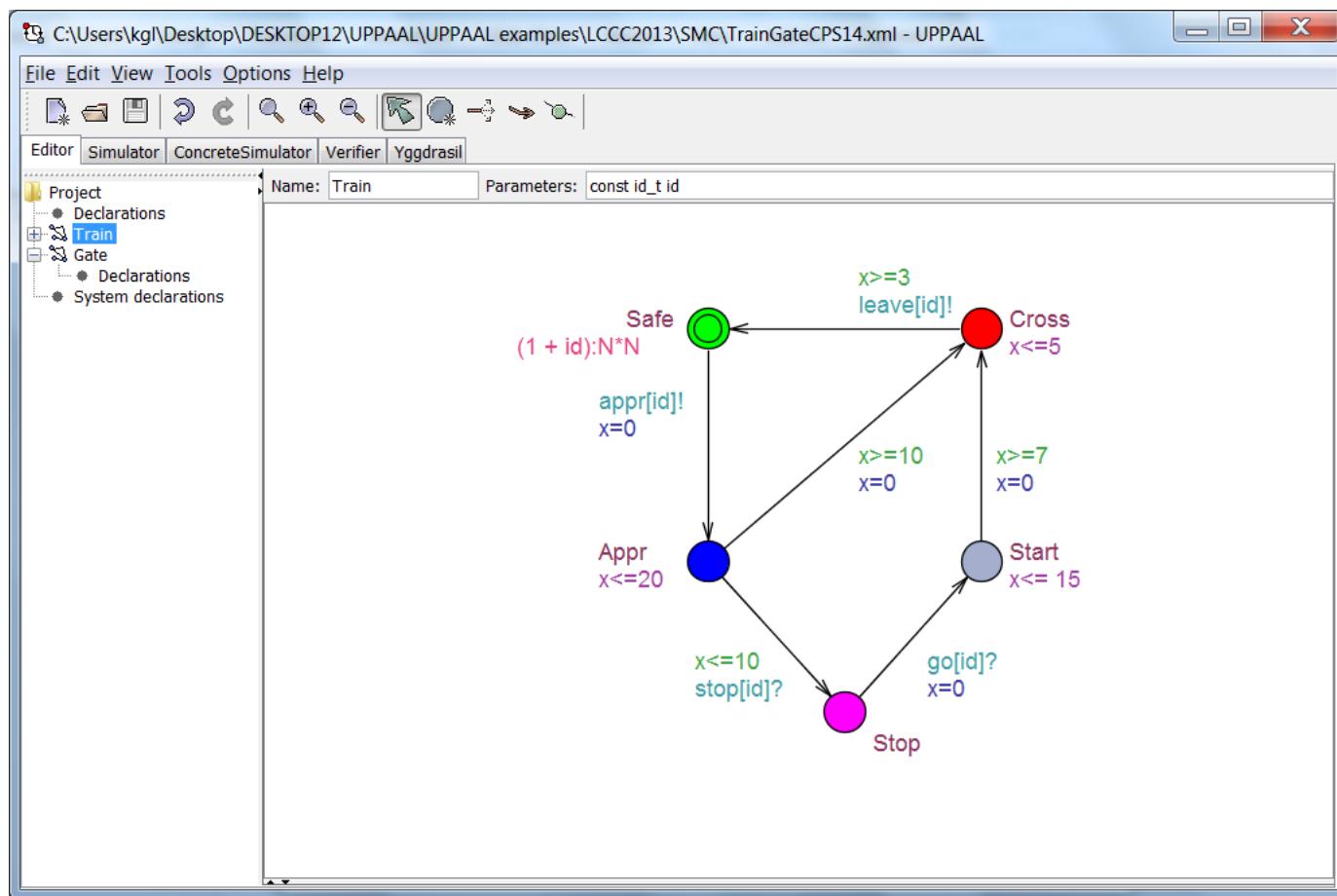


Logical Specifications

- Validation Properties
 - Possibly: $E \leftrightarrow P$
- Safety Properties
 - Invariant: $A[] P$
 - Pos. Inv.: $E[] P$
- Liveness Properties
 - Eventually: $A \leftrightarrow P$
 - Leadsto: $P \rightarrow Q$
- Bounded Liveness
 - Leads to within: $P \rightarrow_{\leq t} Q$



Demo



Bang & Olufsen IR-Link

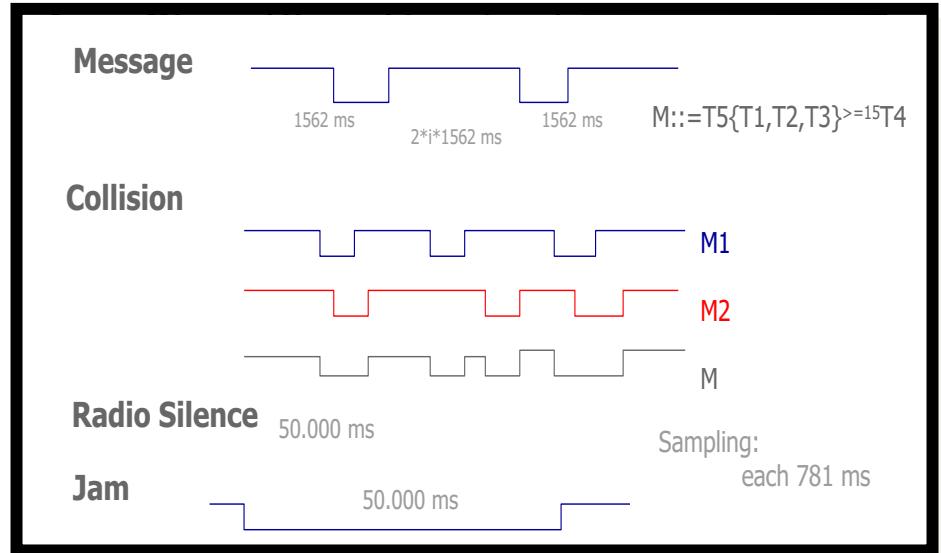
- Bug known to exist for 10 years
- Ill-described:
2.800 lines of assembler code + 3 flowchart + 1 B&O eng.
- 3 months for modeling.
- UPPAAL detects error with 1.998 transition steps (shortest)
- Error trace was confirmed in B&O laboratory.
- Error corrected and verified in UPPAAL.

Arne Skou, Klaus Havelund



Bang & Olufsen IR-Link

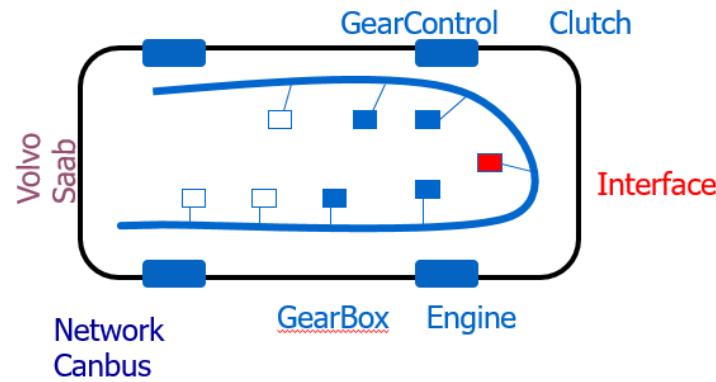
- Bug known to exist for 10 years
- Ill-described:
2.800 lines of assembler code + 3 flowchart + 1 B&O eng.
- 3 months for modeling.
- UPPAAL detects error with 1.998 transition steps (shortest)
- Error trace was confirmed in B&O laboratory.
- Error corrected and verified in UPPAAL.



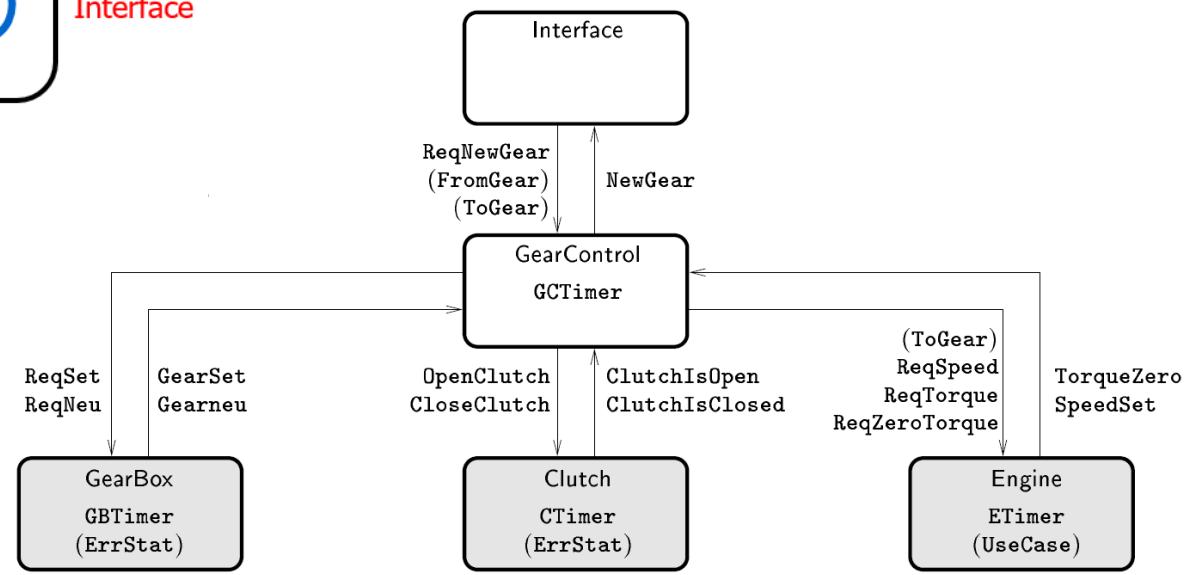
1st RTSS'97 talk, Klaus Havelund

Gear Controller

with MECEL AB



Flowgraph

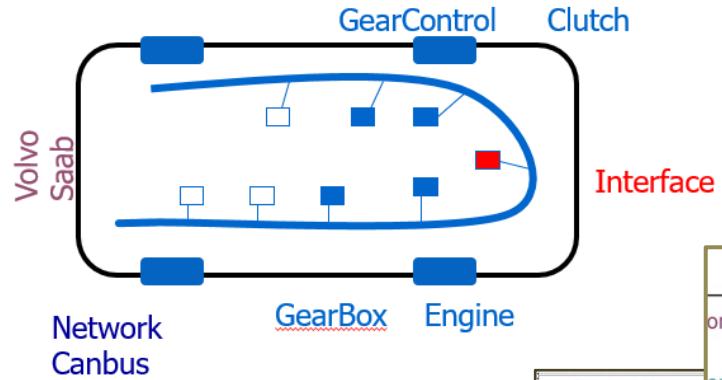


Magnus Lindahl
Paul Pettersson
Wang Yi
2001



Gear Controller

with MECEL AB

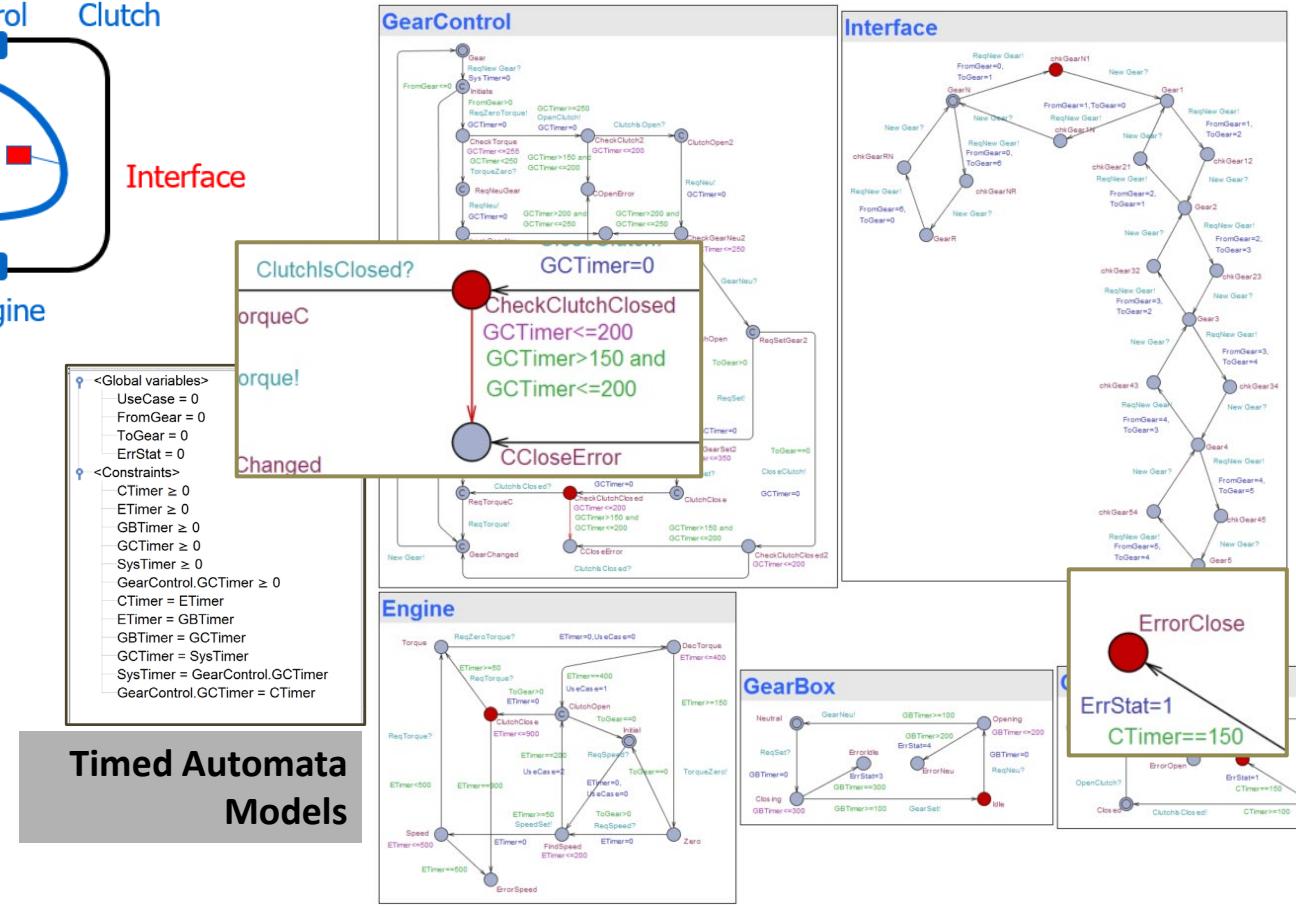


```

<Global variables>
  UseCase = 0
  FromGear = 0
  ToGear = 0
  ErrStat = 0
  <Constraints>
    CTimer ≥ 0
    ETimer ≥ 0
    GBTimer ≥ 0
    GCTimer ≥ 0
    SysTimer ≥ 0
    GearControl.GCTimer ≥ 0
    CTimer = ETimer
    ETimer = GBTimer
    GBTimer = GCTimer
    GCTimer = SysTimer
    SysTimer = GearControl.GCTimer
    GearControl.GCTimer = CTimer
  
```

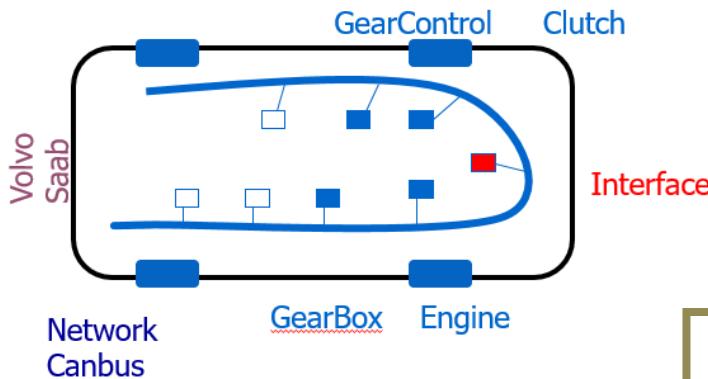
Timed Automata Models

Magnus Lindahl
Paul Pettersson
Wang Yi
2001



Gear Controller

with MECEL AB



Requirements

$\text{GearControl}@\text{Initiate} \rightsquigarrow_{\leq 1500} (\text{ErrStat} = 0) \Rightarrow \text{GearControl}@\text{GearChanged}$

$\text{GearControl}@\text{Initiate} \rightsquigarrow_{\leq 1000} (\text{ErrStat} = 0 \wedge \text{UseCase} = 0) \Rightarrow \text{GearControl}@\text{GearChanged}$

$\text{Clutch}@ErrorClose \rightsquigarrow_{\leq 200} \text{GearControl}@CCloseError$

$\text{Clutch}@ErrorOpen \rightsquigarrow_{\leq 200} \text{GearControl}@COpenError$

$\text{GearBox}@ErrorIdle \rightsquigarrow_{\leq 350} \text{GearControl}@GSetError$

$\text{GearBox}@ErrorNeu \rightsquigarrow_{\leq 200} \text{GearControl}@GNeuError$

$Inv (\text{GearControl}@CCloseError} \Rightarrow \text{Clutch}@ErrorClose)$

$Inv (\text{GearControl}@COpenError} \Rightarrow \text{Clutch}@ErrorOpen)$

$Inv (\text{GearControl}@GSetError} \Rightarrow \text{GearBox}@ErrorIdle)$

$Inv (\text{GearControl}@GNeuError} \Rightarrow \text{GearBox}@ErrorNeu)$

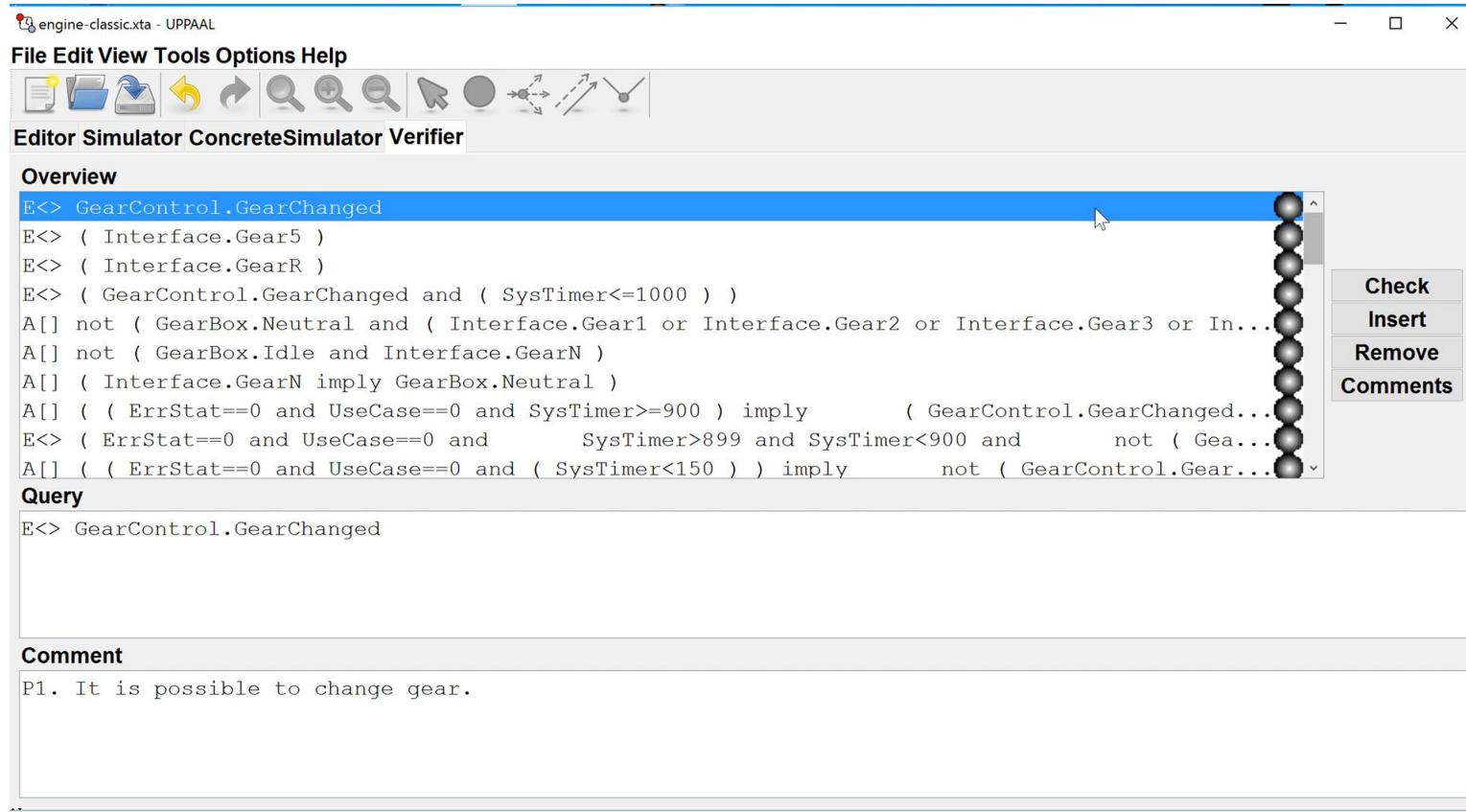
$Inv (\text{Engine}@ErrorSpeed} \Rightarrow \text{ErrStat} \neq 0)$

$Inv (\text{Engine}@Torque} \Rightarrow \text{Clutch}@Closed)$

Magnus Lindahl
Paul Pettersson
Wang Yi
2001



UPPAAL Model Checking - Demo



UPPAAL Model Checking - Demo

The screenshot shows the UPPAAL tool interface with the following details:

- Title Bar:** engine-classic.xta - UPPAAL
- Menu Bar:** File Edit View Tools Options Help
- Toolbar:** Includes icons for Open, Save, Undo, Redo, Search, Find, Copy, Paste, and others.
- Tab Bar:** Editor (selected), Simulator, ConcreteSimulator, Verifier
- Overview Panel:** Displays a list of UPPAAL assertions. One assertion is highlighted with a blue background:

```
A[] ( ( ErrStat==0 and UseCase==2 and SysTimer>=1205 ) imply      ( GearControl.GearChanged... )
E<> ( ErrStat==0 and UseCase==2 and SysTimer>1204 and      SysTimer<1205 and      not ( Ge...
A[] ( ( UseCase==2 and ( SysTimer<450 ) ) imply not ( GearControl.GearChanged or GearContro...
E<> ( UseCase==2 and GearControl.GearChanged and ( SysTimer==450 ) )
A[] ( ( ErrStat==0 and UseCase==2 and FromGear>0 and ToGear>0 and SysTimer<750 ) imply not ...
E<> ( ErrStat==0 and UseCase==2 and FromGear>0 and ToGear>0 and GearControl.GearChanged and...
A[] ( ( Clutch.ErrorClose and ( GearControl.GCTimer>200 ) ) imply GearControl.CCloseError )
A[] ( GearControl.CCloseError imply Clutch.ErrorClose )
A[] ( ( Clutch.ErrorOpen and ( GearControl.GCTimer>200 ) ) imply GearControl.COpenError )
A[] ( ( GearControl.COpenError ) imply Clutch.ErrorOpen )
```
- Check/Insert/Remove/Comments Buttons:** Located on the right side of the Overview panel.
- Query Panel:** Displays a query assertion:

```
A[] ( ( Clutch.ErrorClose and ( GearControl.GCTimer>200 ) ) imply
      GearControl.CCloseError )
```
- Comment Panel:** Displays a comment section:

P9. Clutch Errors.
a) If the clutch is not closed properly (i.e. a timeout occurs) the gearbox controller will enter the location CCloseError within 200 ms.



(Wireless) Protocols in UPPAAL

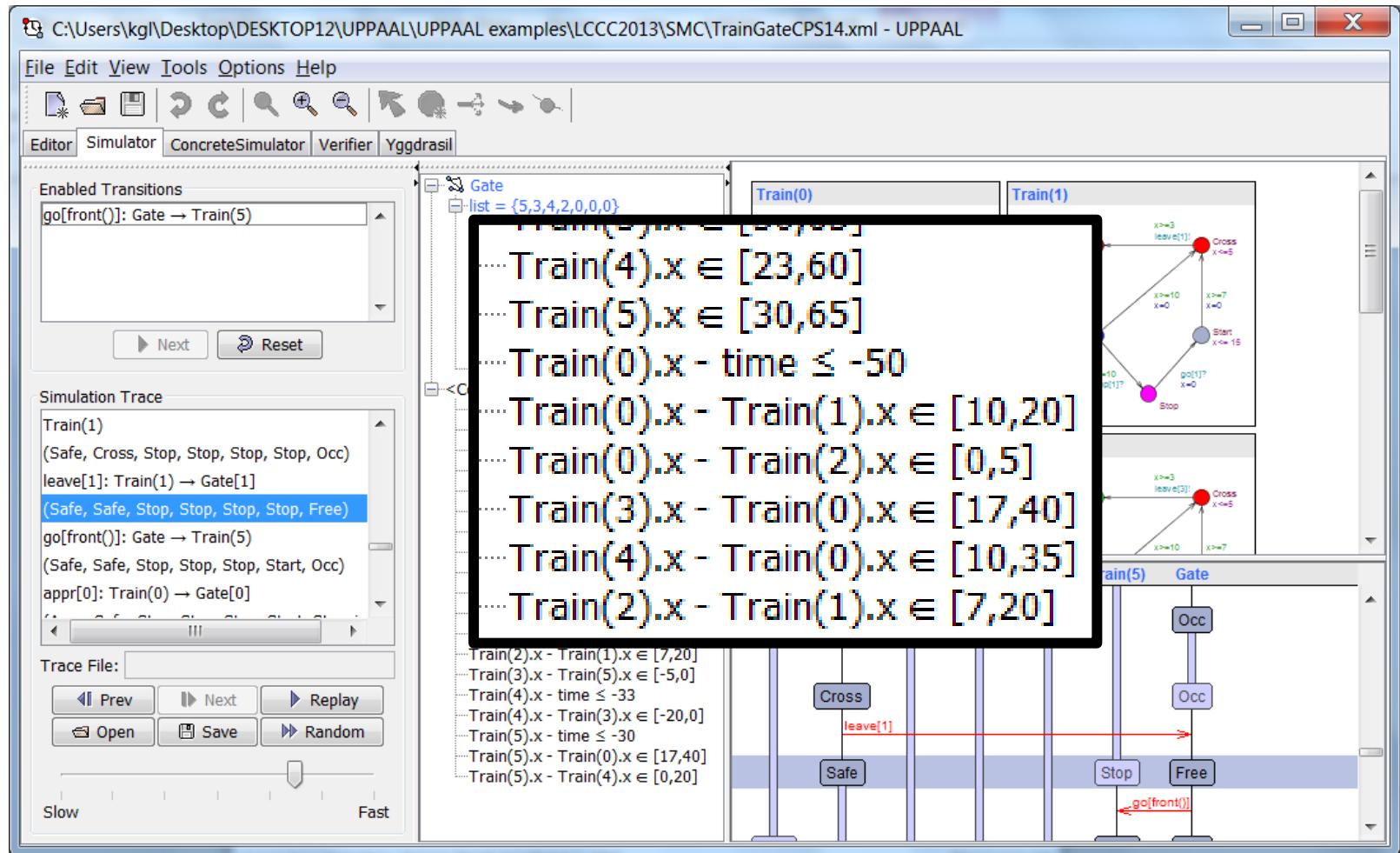
- Bang & Olufsen IR Link
- Philips Audio Protocol
- Collision-Avoidance Protocol
- Bounded Retransmission Protocol
- TDMA Protocol
- Multimedia Streams
- ATM ABR Protocol
- Lamport's Leader Election Protocol
- ABB Fieldbus Protocol
- IEEE 1394 Firewire Root Contention
- Bluetooth Protocol
- Distributed Agreement Protocol
- FlexRay
- CHESS MAC Protocol
- Proprietary WSN, Other Big Danish Company
- MESH Protocol (MAC & Routing), NEOCORTEC



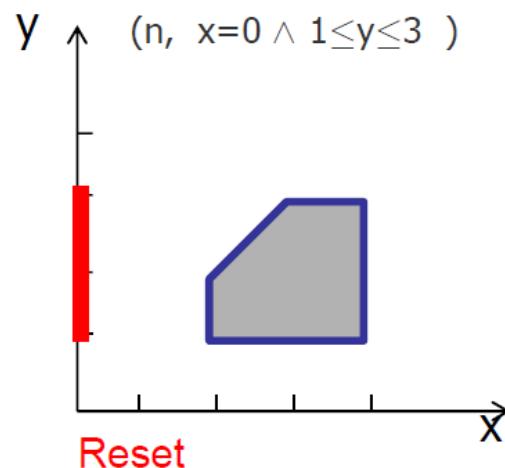
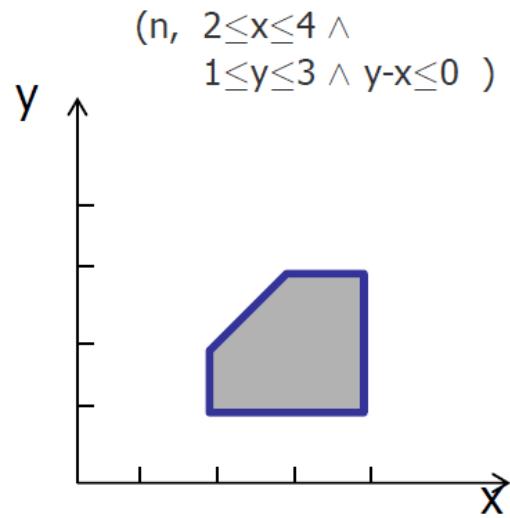
Verification Engine & Options



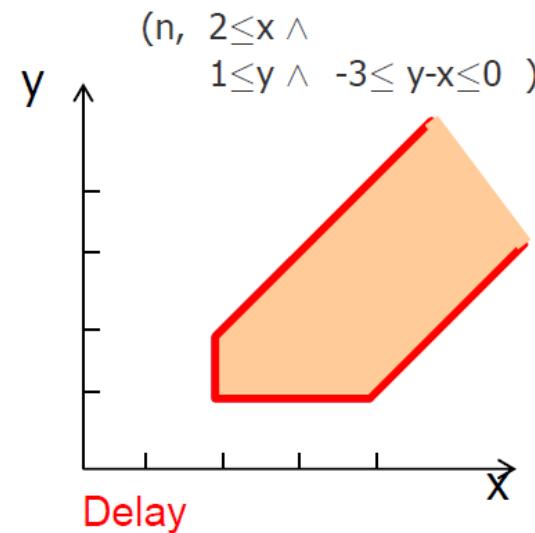
The "secret" of UPPAAL



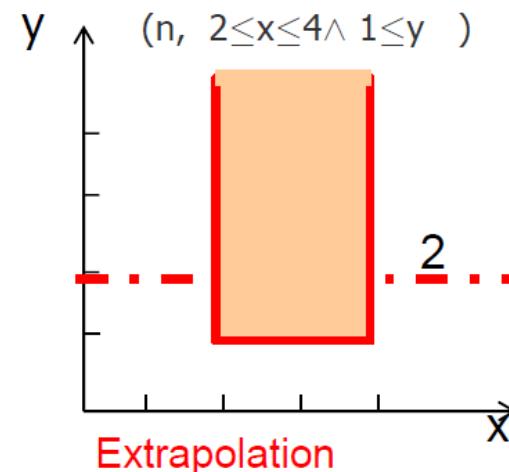
Zones - Operations



Reset



Delay

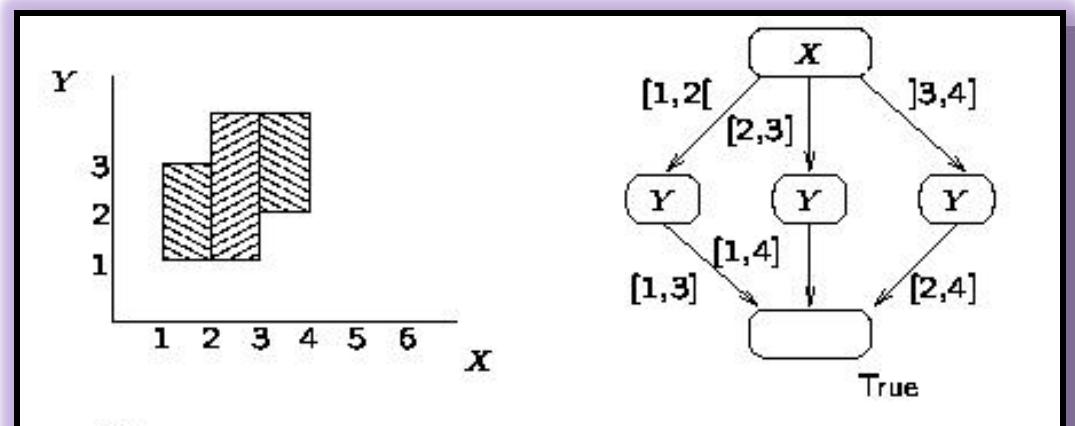
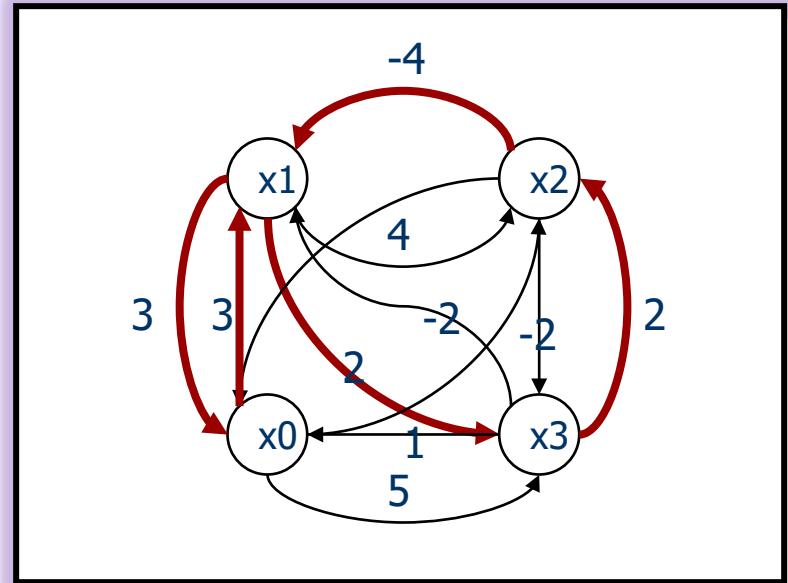


Extrapolation

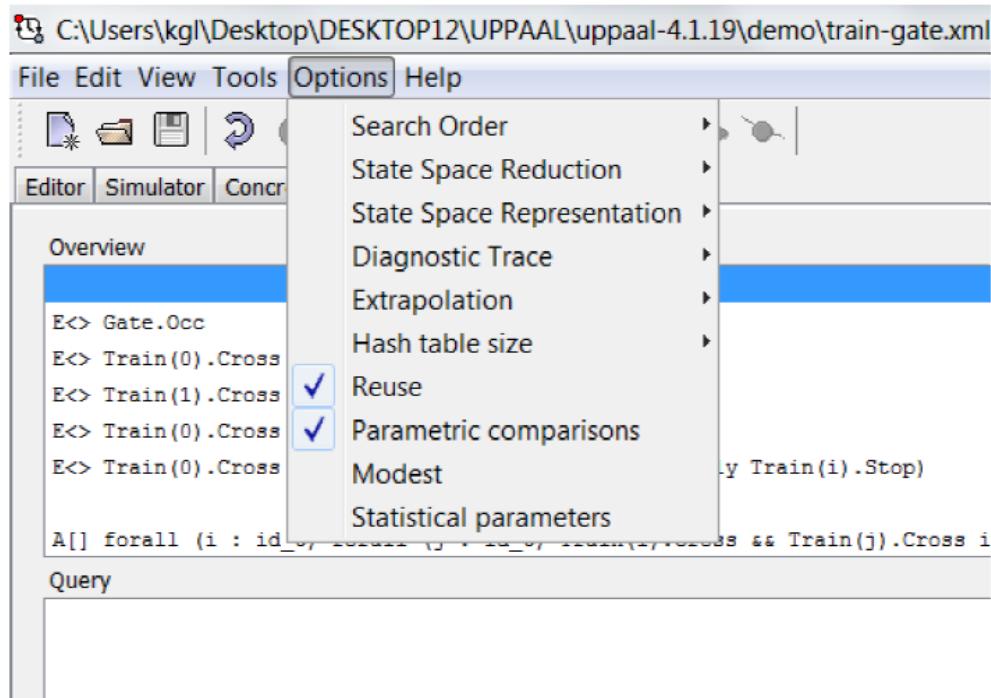


Datastructures for Zones

- Difference Bounded Matrices (DBMs)
- Minimal Constraint Form [RTSS97]
- Clock Difference Diagrams [CAV99]



Verification Options



Search Order

- Depth First
- Breadth First
- Random Depth First

State Space Reduction

- None
- Conservative
- Aggressive
- Extreme

State Space Representation

- DBM
- Compact Form
- Under Approximation
- Over Approximation

Diagnostic Trace

- Some
- Shortest
- Fastest

Extrapolation

Hash Table size

Reuse

Principles of Cyber-Physical Systems

Difference Bounded Matrices Real-Time Scheduling

Instructors:

Kim G. Larsen, Christian Schilling, Max Tschaikowski

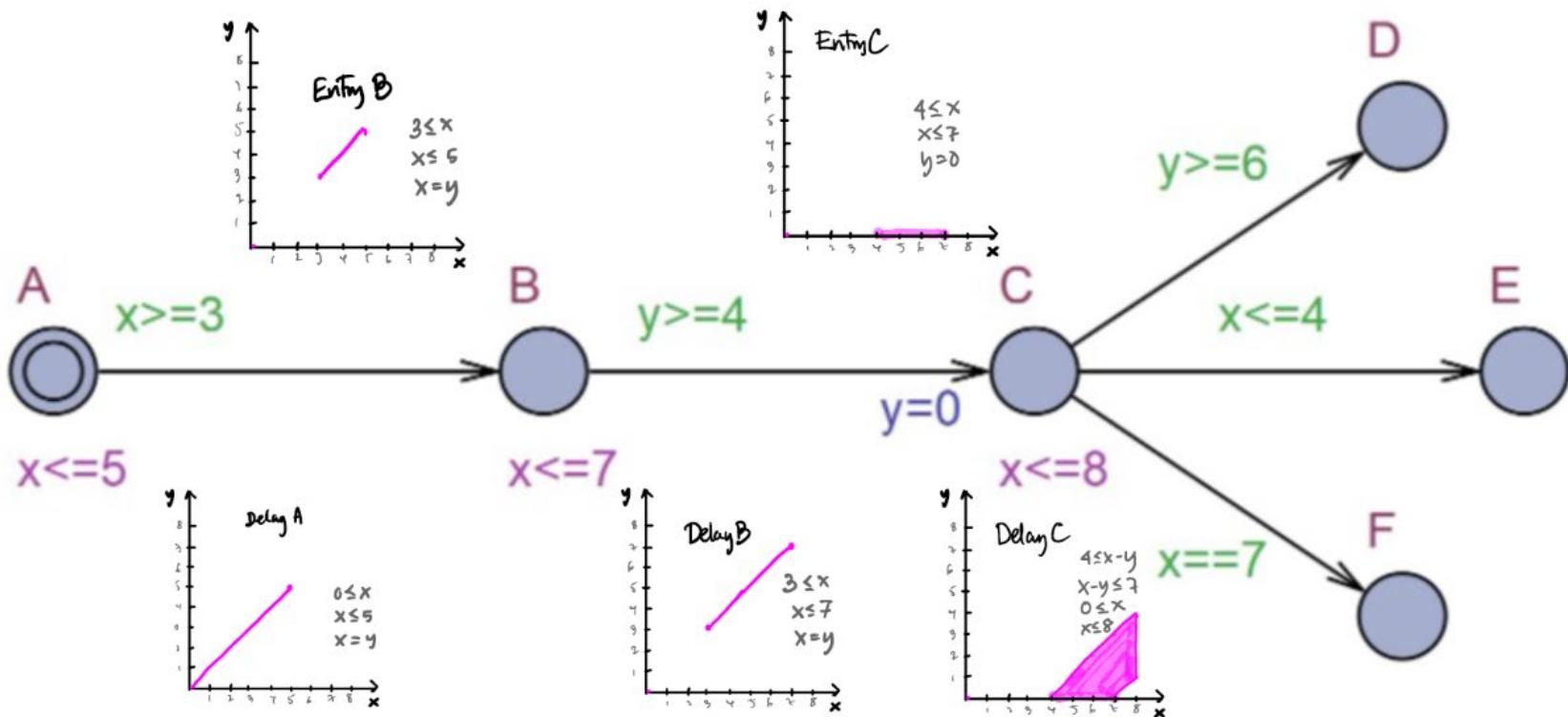


Zones

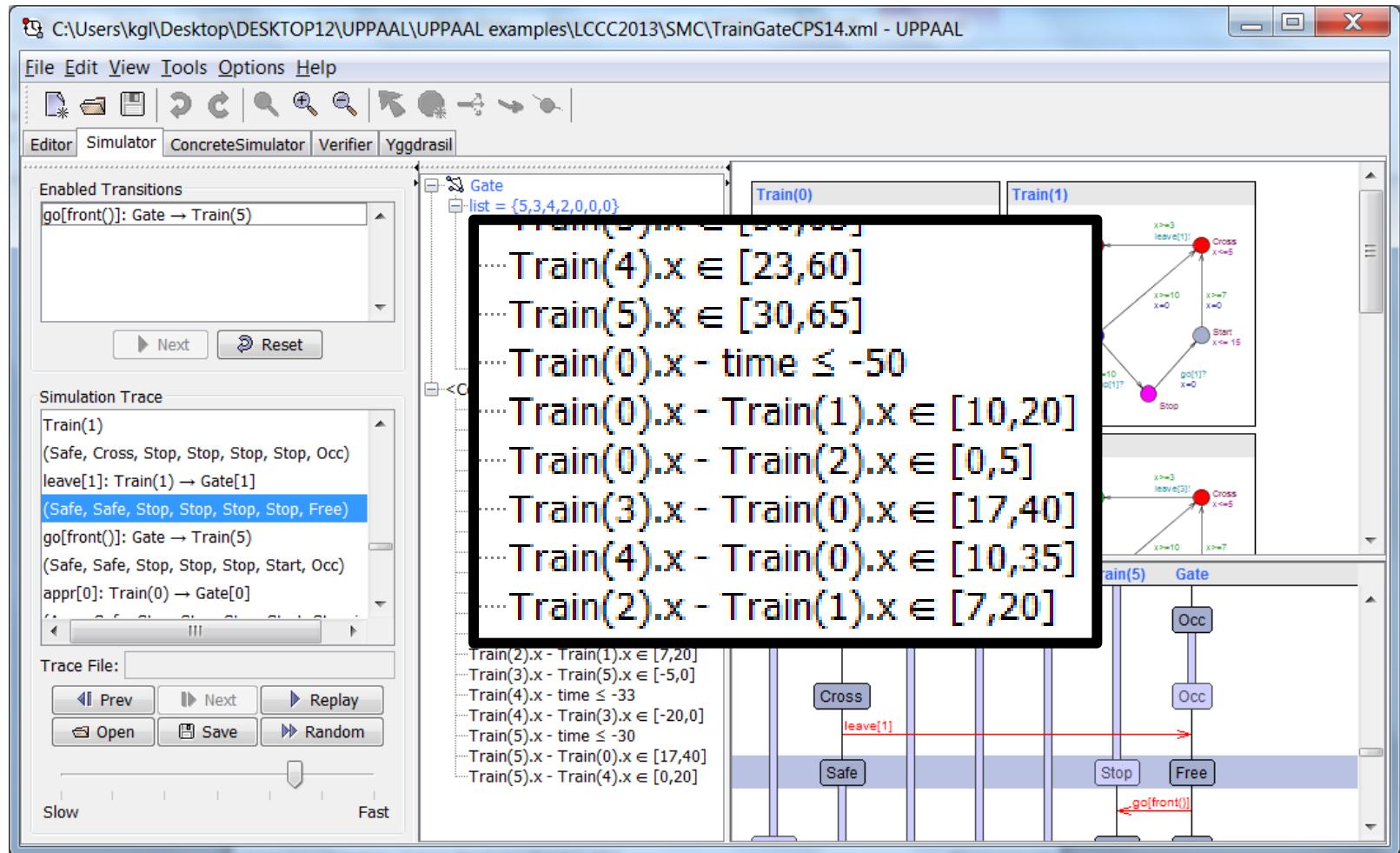
Difference Bounded Matrices



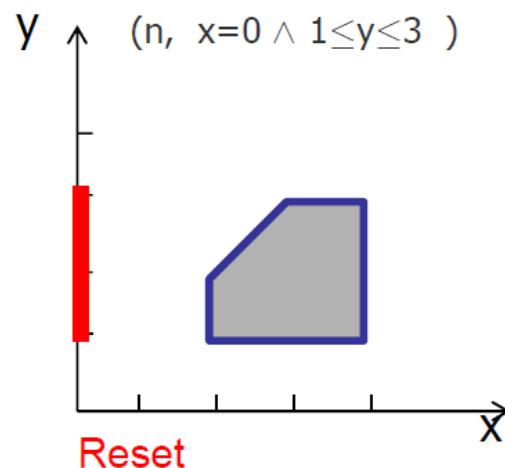
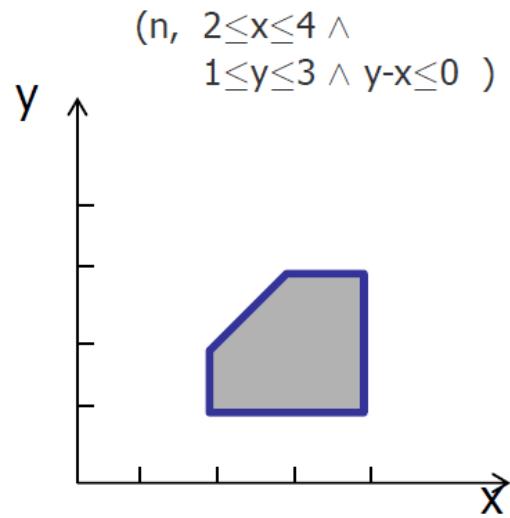
Zones



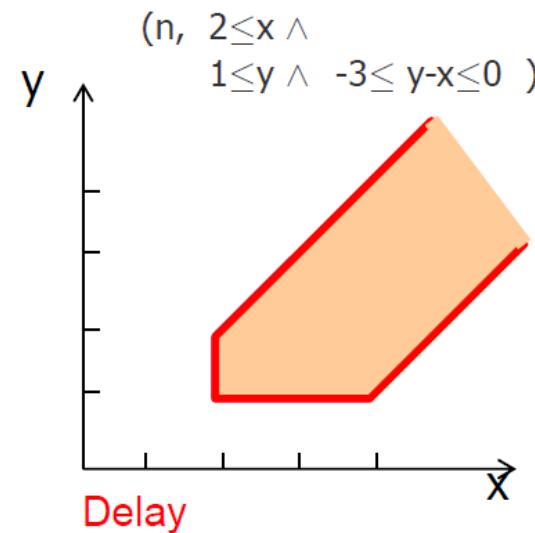
The "secret" of UPPAAL



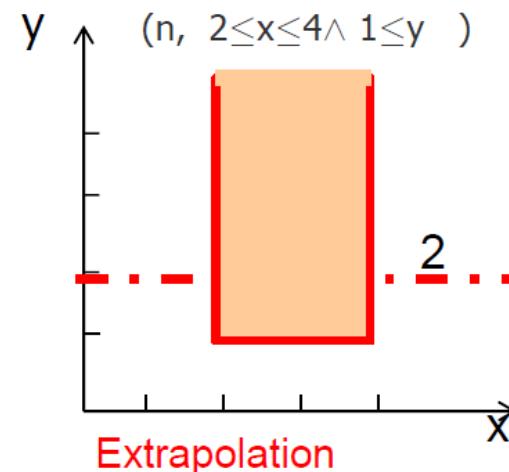
Zones - Operations



Reset



Delay

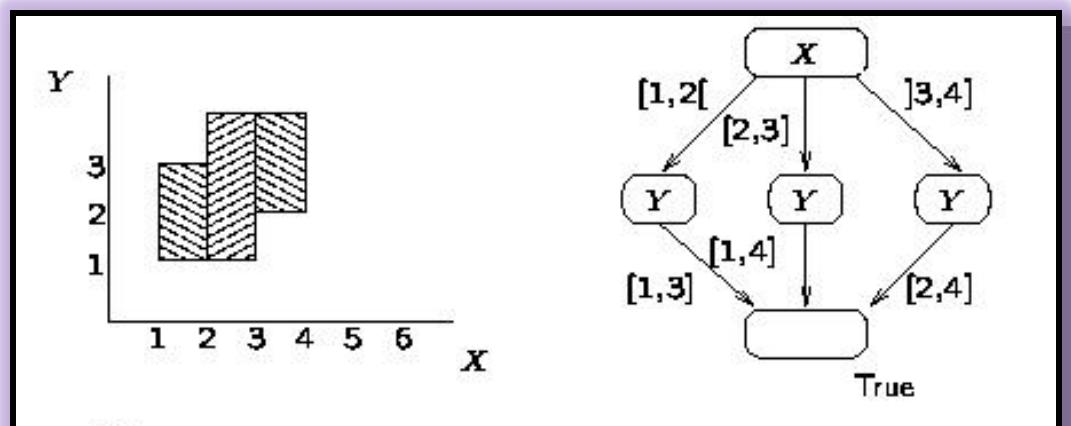
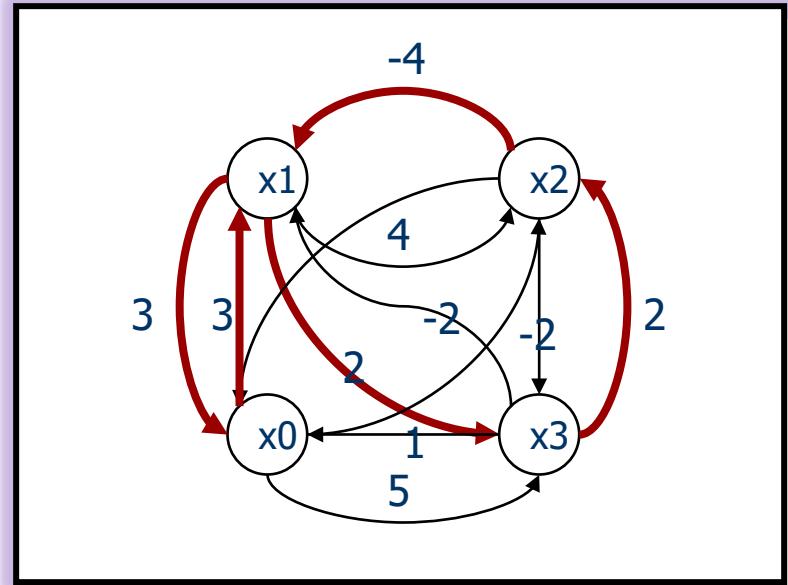


Extrapolation

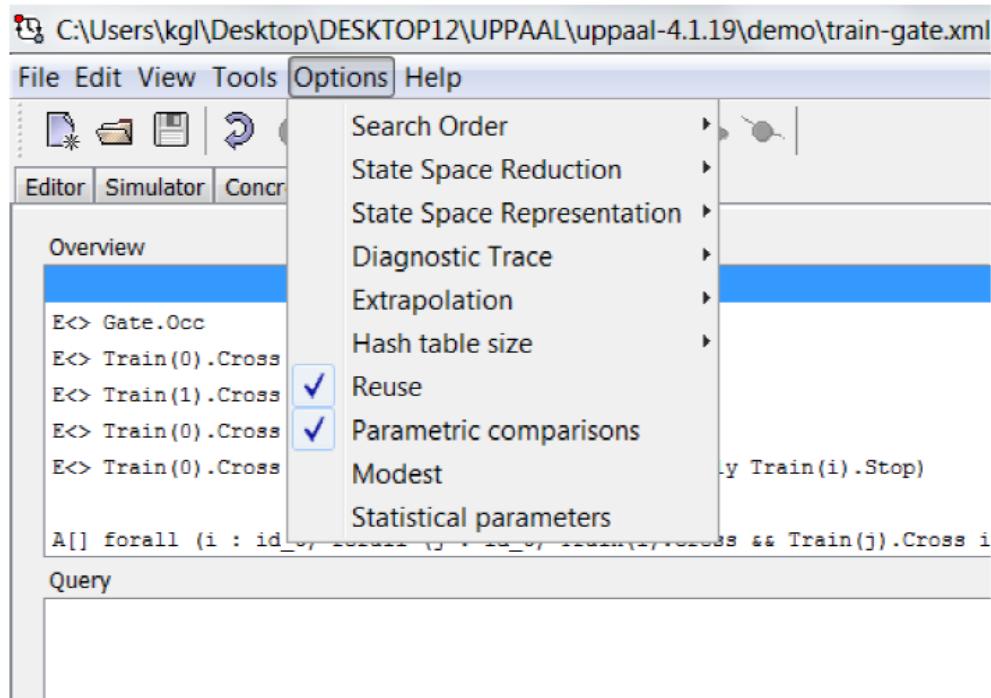


Datastructures for Zones

- Difference Bounded Matrices (DBMs)
- Minimal Constraint Form [RTSS97]
- Clock Difference Diagrams [CAV99]



Verification Options



Search Order

Depth First
Breadth First
Random Depth First

State Space Reduction

None
Conservative
Aggressive
Extreme

State Space Representation

DBM
Compact Form
Under Approximation
Over Approximation

Diagnostic Trace

Some
Shortest
Fastest

Extrapolation

Hash Table size

Reuse

OG NU TIL NOGET HELT
ANDET

(NÆSTEN)



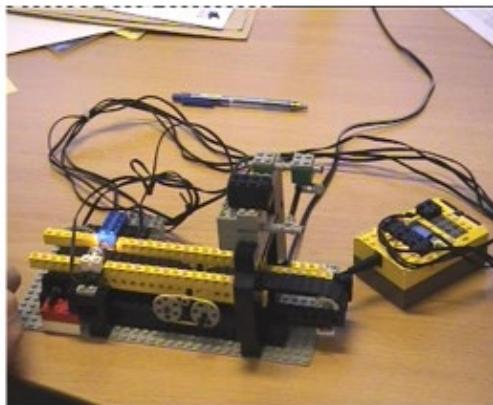
Real-Time Scheduling



Why Scheduling?

A Real Real Timed System

The Plant
Conveyor Belt & Bricks



Controller Program
LEGO MINDSTORM

Problem: only one CPU.

- only one task can execute at a time

Scheduling:

- which task should execute a given point in time.

Questions:

- how frequent should a task execute in order not to miss important events?
- What is the execution time of a task.

NQC programs

```
task MAIN{
    Sensor(IN_1, IN_LIGHT);
    Fwd(OUT_A,1);
    Display(1);

    start PUSH;

    while(true){

        wait(IN_1<=LIGHT_LEVEL);
        ClearTimer(1);
        active=1;
        PlaySound(1);

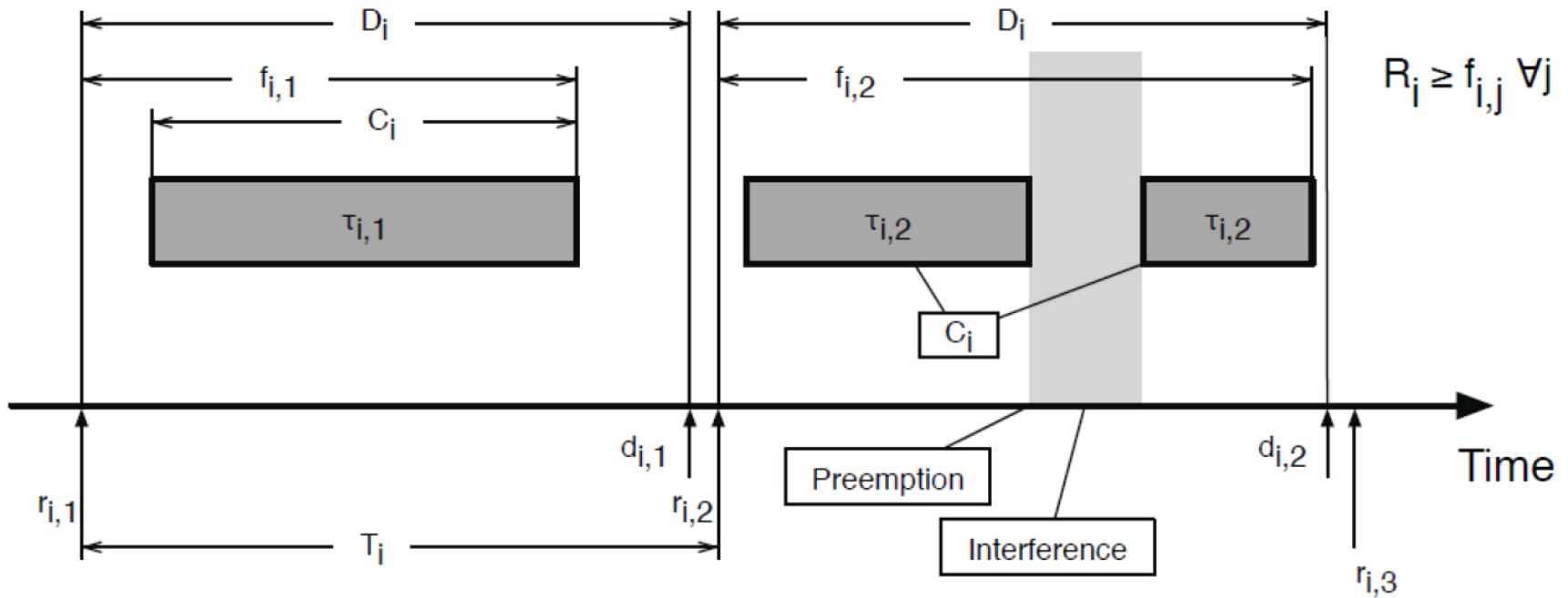
        wait(IN_1>LIGHT_LEVEL);
    }
}
```

```
int active;
int DELAY;
int LIGHT_LEVEL;
```

```
task PUSH{
    while(true){
        wait(Timer(1)>DELAY && active==1);
        active=0;
        Rev(OUT_C,1);
        Sleep(8);
        Fwd(OUT_C,1);
        Sleep(12);
        Off(OUT_C);
    }
}
```

Task Scheduling Terminology

Symbol	Meaning
τ_i	i -th task
$\tau_{i,j}$	j -th instance of τ_i
T_i	period of τ_i
D_i	relative deadline of τ_i
C_i	cost (WCET) of τ_i
R_i	worst-case response time of τ_i
$r_{i,j}$	release time of $\tau_{i,j}$
$f_{i,j}$	response time of $\tau_{i,j}$
$d_{i,j}$	absolute deadline of $\tau_{i,j}$



Feasible Schedule: $\forall i. R_i \leq D_i$



Feasibility?

Example

Process	Period	Computation Time
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Does there exist a feasible Schedule for these tasks?

Needs a systematic way of determine which process (task / job) to execute when!



Cyclic Executives



The Cyclic Executive Approach

Properties

- Offline generation of static schedule
- Tasks (or parts thereof) are implemented as procedures
- Procedures are mapped onto a sequence of minor cycles
- Minor cycles constitute the complete schedule: the major cycle
- Minor cycle determines the minimum period
- Major cycle determines the maximum cycle time
- Uses real time clock/interrupt for synchronisation
- Minor cycles are synchronisation points
- Concurrent design, but sequential code (collection of procedures)

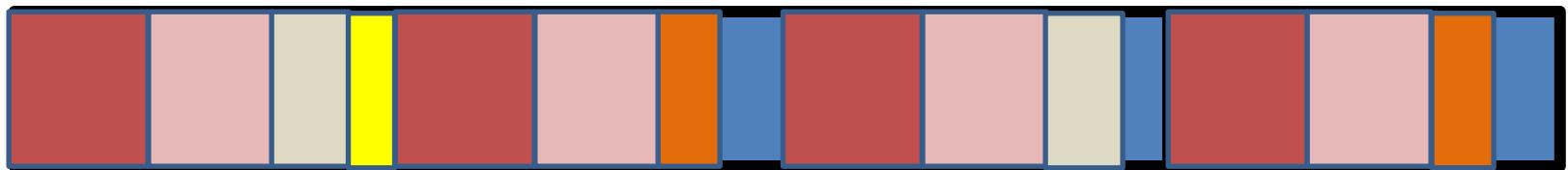


Feasibility?

Example

Process	Period	Computation Time
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Does there exist a feasible Schedule for these tasks?



0

100



Implementing the Scheduler

```
timer_setup(25);          /* Tm = 25 */

while(1) {
    wait_for_interrupt();
    task_a(); task_b(); task_c();

    wait_for_interrupt();
    task_a(); task_b(); task_d(); task_e();

    wait_for_interrupt();
    task_a(); task_b(); task_c();

    wait_for_interrupt();
    task_a(); task_b(); task_d();
}
```



How to determine minor and major cycles?

Minor Cycle

- ① Task periods must be **integer multiples** of minor cycle period
- ② Minor cycle period should be as large as possible

$$T_m = \text{gcd}(T_1, \dots, T_n)$$

Major Cycle

- ① **Integer multiple** of all task periods
- ② Major cycle period should be as small as possible

$$T_M = \text{lcm}(T_1, \dots, T_n)$$

Example

$$T_m = \text{gcd}(25, 25, 50, 50, 100) = 25$$

$$T_M = \text{lcm}(25, 25, 50, 50, 100) = 100$$



Problems!!

Mutually prime periods

- Minor cycle period of 1
- Impossible for “realistic” task cost
- Clock granularity too coarse

Large periods

- Major cycle determines maximum period
- Possible solution: *secondary scheduling*

Large execution times

- Impossible to schedule for cost greater than minor cycle
- Possible solution: *split tasks*
 - Hard to split; cross cutting concern
 - May cut across useful and well-established boundaries
 - Potentially very bad for software engineering (error prone)



More Problems!!

Problems

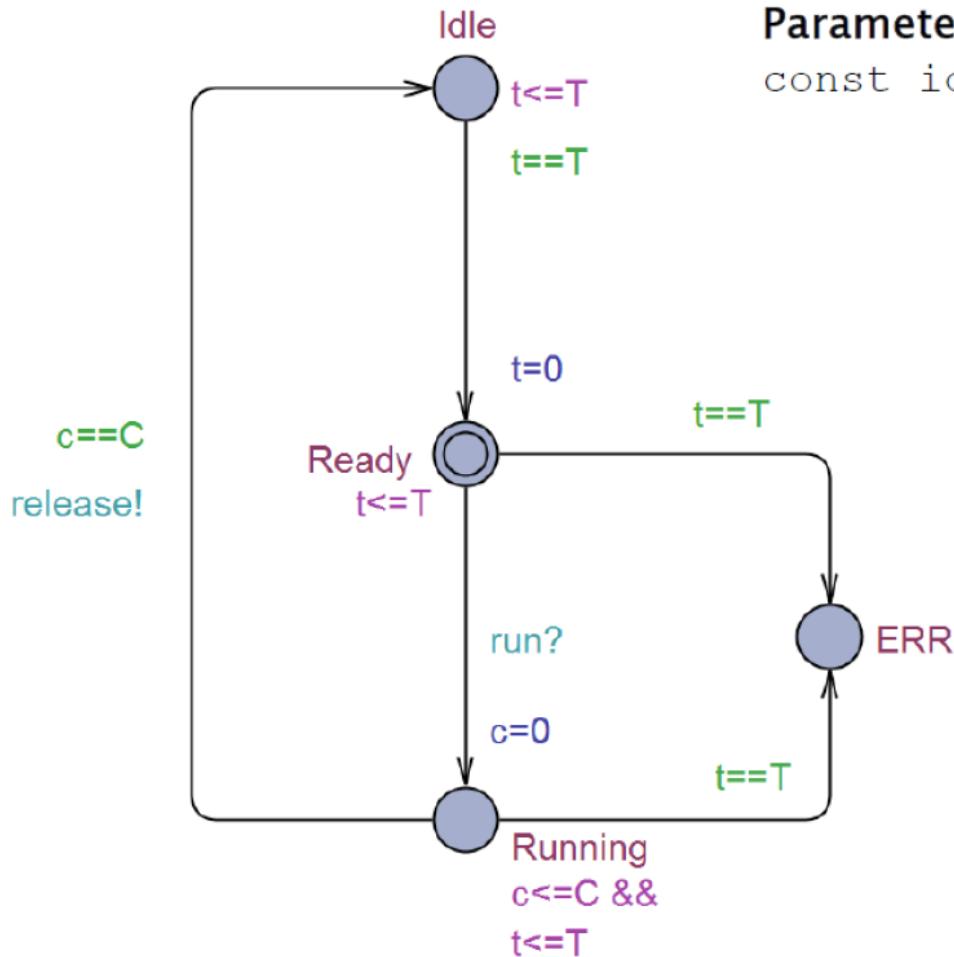
- Difficult to construct and maintain (NP-hard)
- Sporadic processes are difficult to incorporate
- More flexible scheduling methods are difficult to support
- Determinism is not required but **predictability** is



Cyclic Executives in UPPAAL



A Task



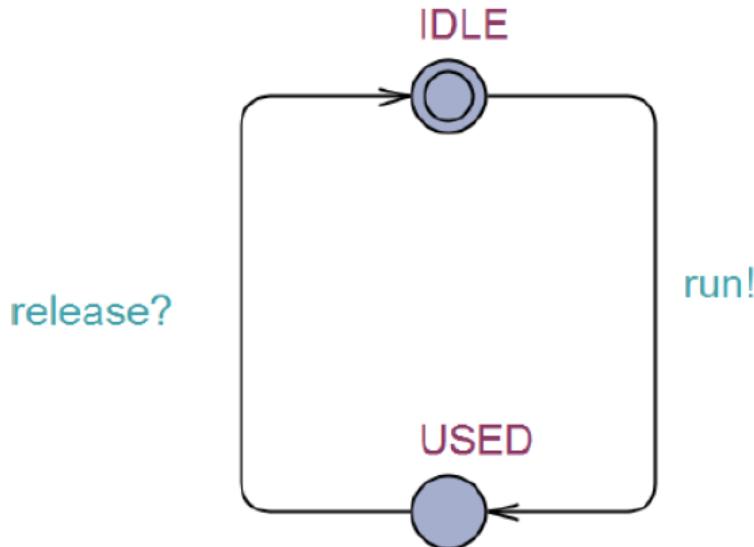
Parameters:

```
const id_t id, const int T, const int C
```

Local Declarations:

```
clock t, c;
```

The CPU



Global Declarations

```
const int N=6;  
typedef int [1,N] id_t;
```

```
chan ready, release;  
urgent chan run;
```

```
clock time;
```



System Declaration

```
// Place template instantiations here.  
PrA = TASK(1,25,10);  
PrB = TASK(2,25,8);  
PrC = TASK(3,50,5);  
PrD = TASK(4,50,4);  
PrE = TASK(5,100,2);  
  
// List one or more processes to be composed into a system.  
system PrA, PrB, PrC, PrD, PrE, CPU;
```



Cyclic Executives using UPPAAL

Reachability Query

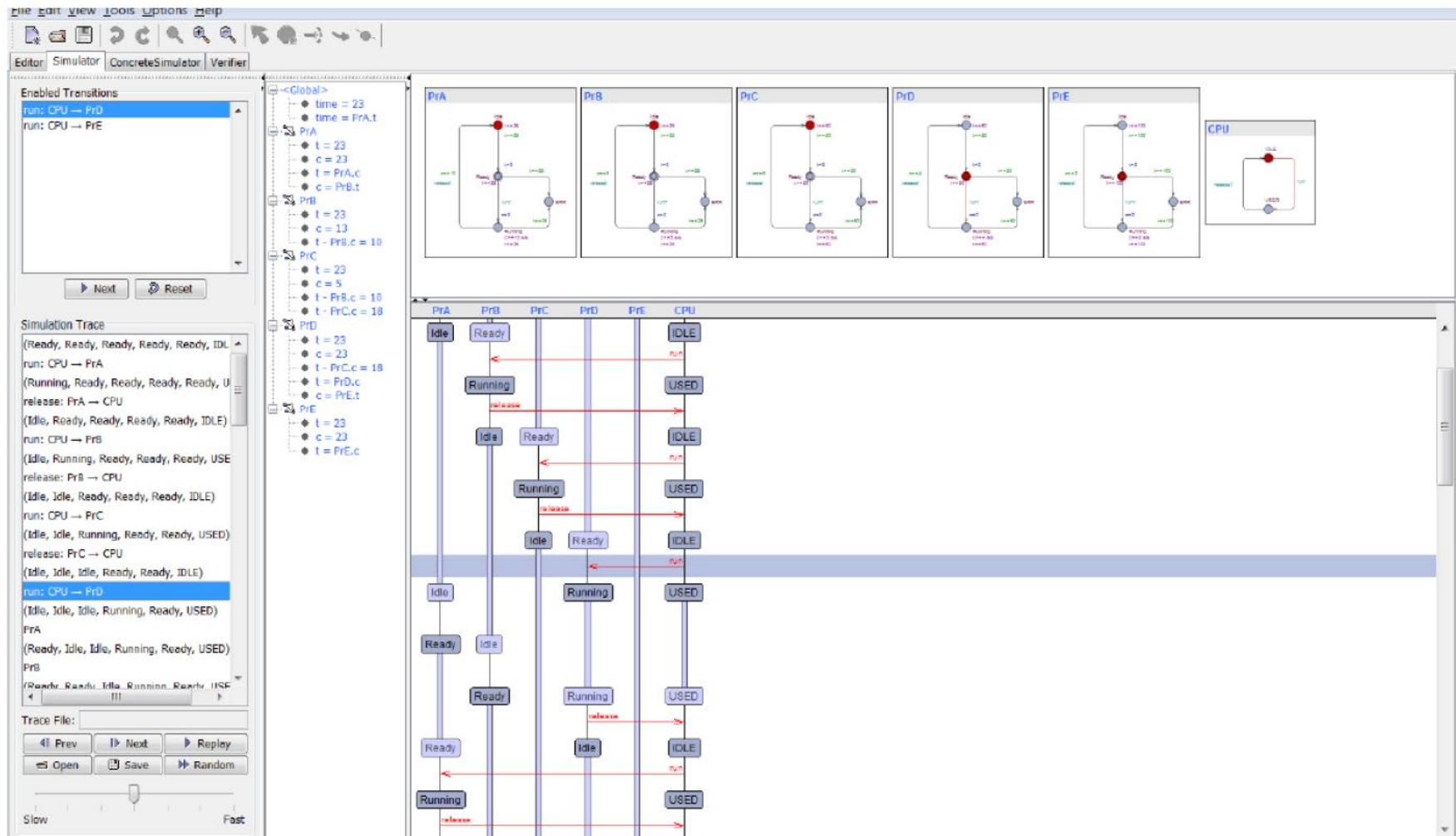
```
E<> PrA.Ready and PrB.Ready and PrC.Ready and  
PrD.Ready and PrE.Ready and  
PrA.t==0 and PrB.t==0 and PrC.t==0 and  
PrD.t==0 and PrE.t==0 and time>0
```

Safety Query

```
E[] not (PrA.ERR or PrB.ERR or PrC.ERR or  
PrD.ERR or PrE.ERR)
```



Cyclic Executives using UPPAAL



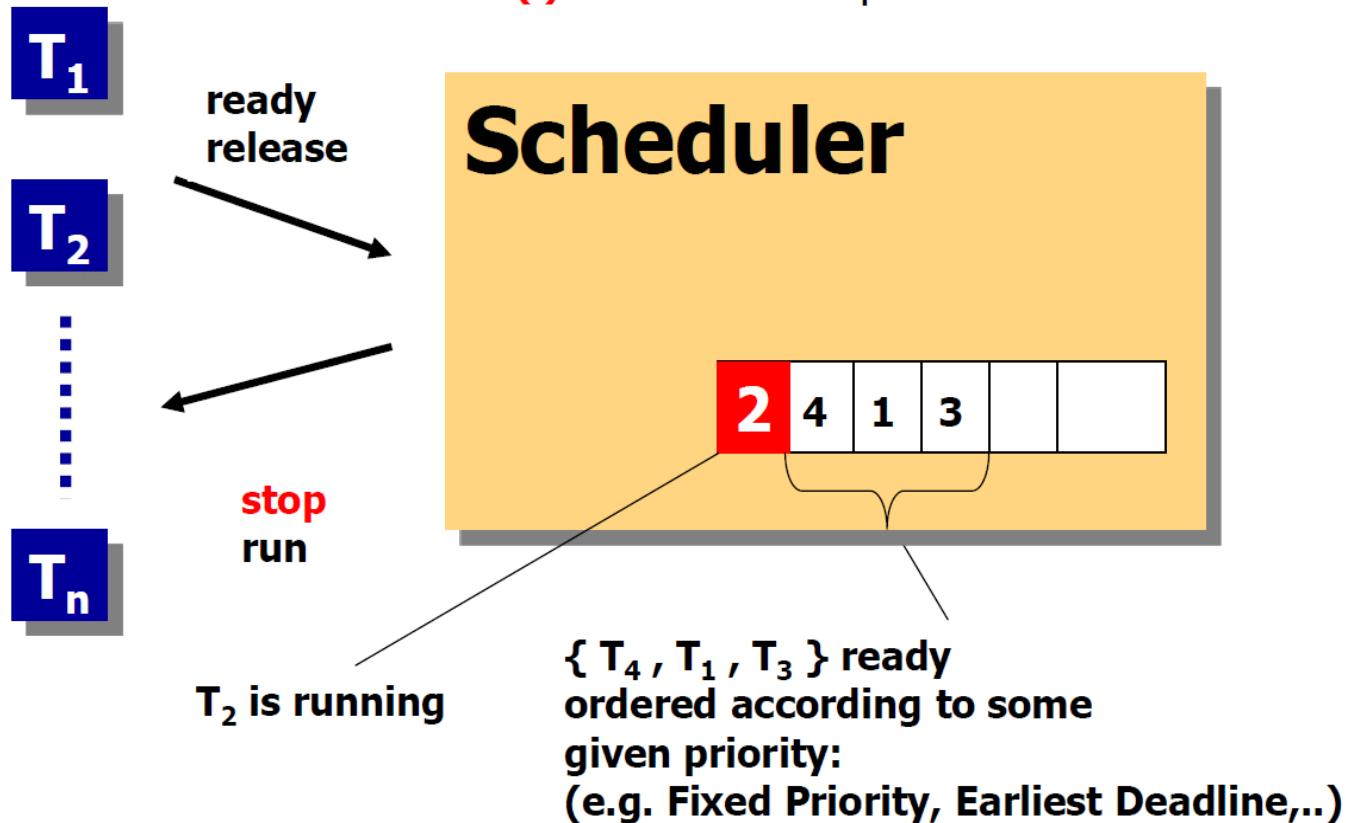
Task Based Scheduling Response Time Analysis

BREAK



Task Scheduling Architecture

$T(i)$: period of task
 $C(i)$: execution time for T_i ,
 $D(i)$: deadline for T_i



Fixed Priority Scheduling (FPS)

Definition (FPS)

- Each process has a fixed, **static**, priority assigned before run-time
- Priority determines execution order

Why FPS?

- Most widely used approach
 - Conceptually simple
 - Well-understood
 - Well-supported
- Main focus of the course

Priority \neq Importance

In RTSSs the “priority” of a process is derived from its **temporal requirements**, not its importance to the correct functioning of the system or its integrity



Earliest Deadline First (EDF)

Definition (EDF)

- Execution order is determined by the **absolute** deadlines
- The next process to run is the one with the **shortest** (nearest) deadline
- Absolute deadlines must be computed at run-time (**dynamic** scheduling)

Why (not) EDF?

- Optimal (anything schedulable in FPS can be scheduled in EDF)
- Less supported by language/OS
- Harder to implement / greater overhead



Preemption vs Non-Preemption

Preemption vs. non-preemption

- Priority-based scheduling: a high-priority process may be released during the execution of a lower priority one
- Preemptive scheduling: immediately switch to the higher-priority process
- Non-preemptive scheduling: lower-priority process will complete before the high-priority executes
- Preemption enables better reactivity for high-priority processes
- EDF and VBS can also take on a preemptive or non-preemptive form

Alternatives

- Alternative strategy: allow lower priority process to continue execution for a bounded time
- These schemes are known as deferred preemption or cooperative dispatching



How to assign priorities (for FPS)

- Rate Monotonic Priority Assignment
- Each process is assigned a (unique) priority based on its period:
 $T_i < T_j \implies P_i > P_j$
- Note: priority 1 (one) is the **lowest (least)** priority

Example (Priority Assignment)

Process	Period(T)	Priority (P)
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2



How to assign priorities (for FPS)

- Rate Monotonic Priority Assignment
- Each process is assigned a (unique) priority based on its period:
 $T_i < T_j \implies P_i > P_j$
- Note: priority 1 (one) is the **lowest (least)** priority

Theorem (Optimality)

Any process set scheduled with a fixed-priority assignment scheme can also be scheduled with a rate monotonic assignment scheme



Utility Based Analysis



Utility-Based Analysis

EPS

If $U > 1$ then unschedulable

- Assume rate monotonic priority assignment
- Sufficient schedulability test for $D = T$ task sets:

$$U \equiv \sum_{i=1}^N \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1)$$

- $U \leq 0.69$ as $N \rightarrow \infty$

Utilisation bounds

N	Utilisation Bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%



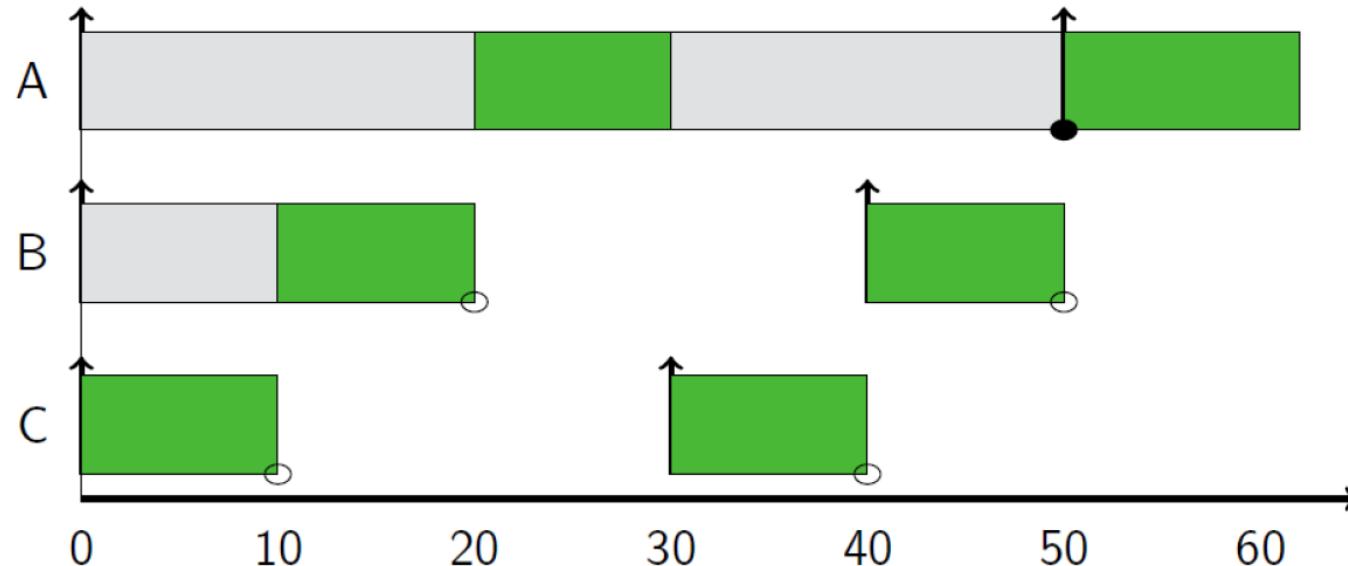
Process Set A: Utility-Based Schedulability test

Example (Utilisation Test for Process Set A)

Process	Period	Computation Time	Priority	Utilisation
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

- The combined utilisation is 0.82
- Above threshold for three processes (0.78): process set **failed** utilisation test

Gantt Chart for Process Set A



- At time $t = 50$ process A misses the deadline

Process	Period	Computation Time	Priority
a	50	12	1
b	40	10	2
c	30	10	3

Process Set B: Utility-Based Schedulability Test

Example (Utilisation Test for Process Set B)

Process	Period	Computation Time	Priority	Utilisation
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

- The combined utilisation is 0.775
- Below threshold for three processes (0.78): utilisation test **succeeded** (will meet all deadlines)



Process Set C

Example (Utilisation Test for Process Set C)

Process	Period	Computation Time	Priority	Utilisation
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

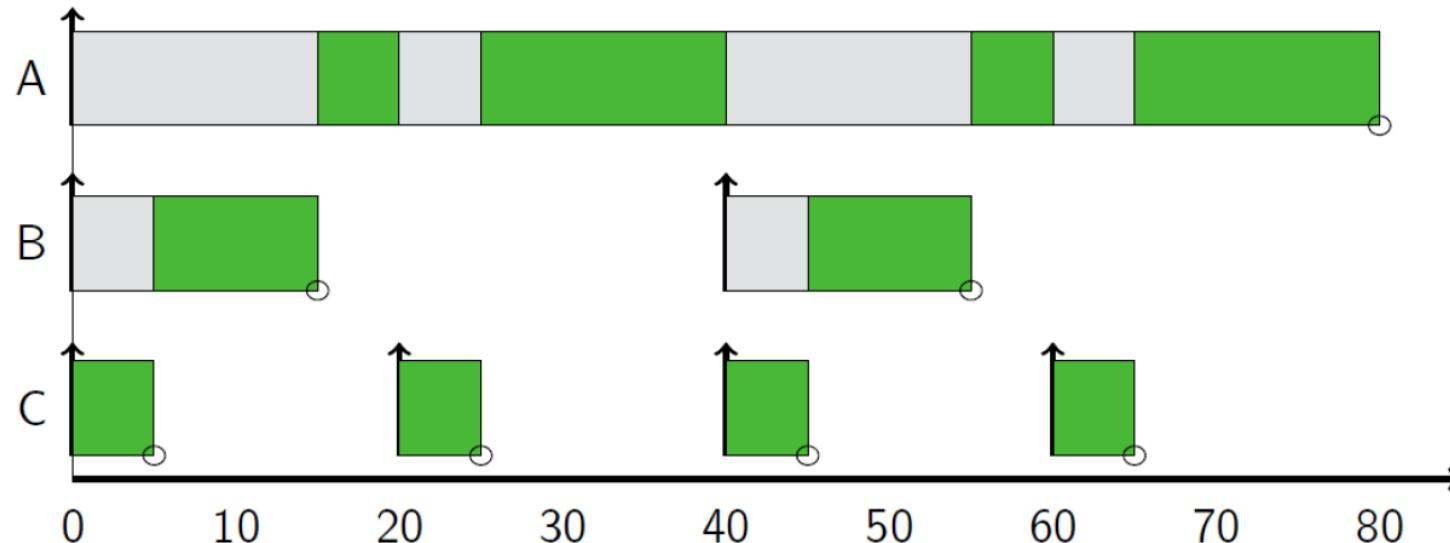
- The combined utilisation is 1.0
- Above threshold for three processes (0.78)... but the process set **will meet all its deadlines**

Utilisation Based Schedulability Test

Sufficient but **not** necessary



Gantt Chart for Process Set C



- Proof that task set C is schedulable

Process	Period	Computation Time	Priority
a	80	40	1
b	40	10	2
c	20	5	3

Utility Test for FPS

Problems

- Not exact
- Not general (only $T = D$)
- The test is sufficient but not necessary
- May lead to unacceptably low utilisation ($< 70\%$)

Advantages / “Fixes”

- Tests can be refined/improved
 - Consider **task families**
 - The Bini utilisation test

$$\prod_{i=1}^N \left(\frac{C_i}{T_i} + 1 \right) \leq 2$$

- Is quick and easy to perform: $\mathcal{O}(N)$



Utility Test for EDF

A much simpler test

$$\sum_{i=1}^N \frac{C_i}{T_i} \leq 1$$

- Superior to utilisation test for FPS; it can support high utilisation
- Other tests available: PDC (processor demand criteria) and QPA (quick processor-demand analysis)



Summary on Utility-Based Test

Utilisation Tests: Pros and Cons

- Easy to perform
- “Binary” answer
- Low utilisation (for FPS)

FPS vs. EDF

- FPS is easier to implement as priorities are static
- EDF requires more complex run-time system with higher overhead
- Easier to incorporate other factors into a priority than into a deadline
- During overload situations
 - FPS is more predictable; low priority processes miss their deadlines first
 - EDF is **unpredictable**; domino effect may occur: large number of processes miss deadlines



Response Time Analysis

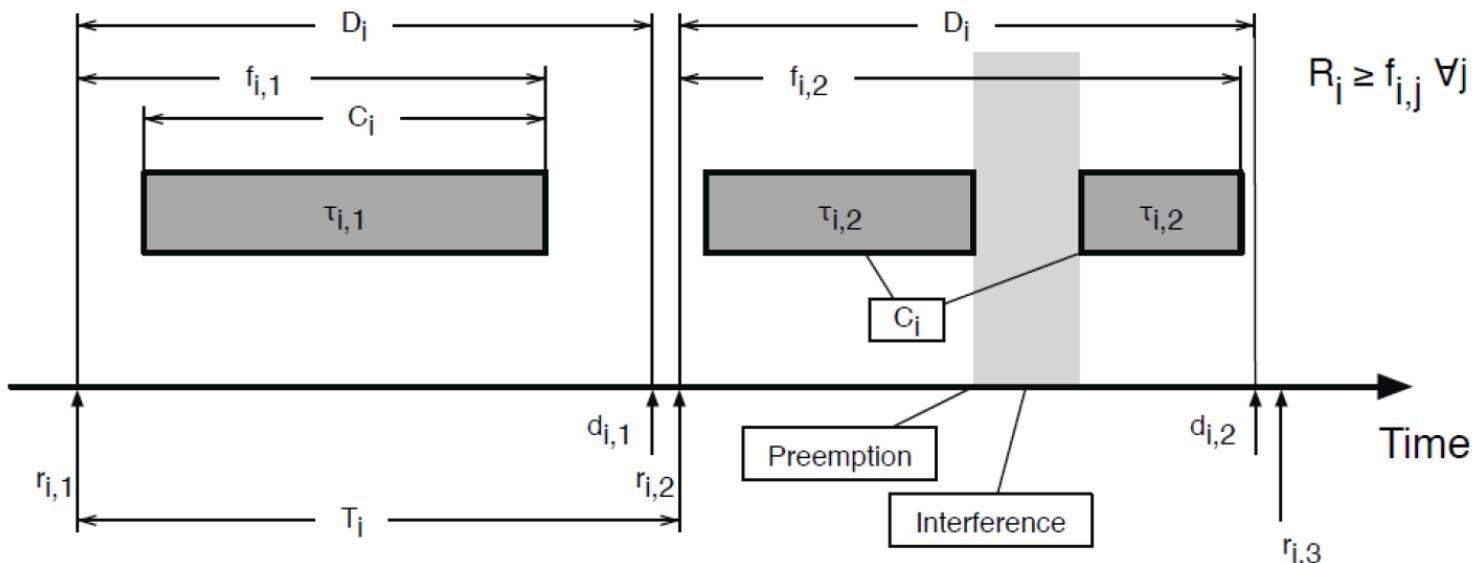
BREAK



Response Time Analysis

Calculating the Slowest Response

- Calculate i 's worst-case response time: $R_i = C_i + I$. Where I is the interference from higher priority tasks
- Check (trivially) if deadline is met $R_i \leq D_i$



Response Time Analysis

Calculating the Slowest Response

- Calculate i 's worst-case response time: $R_i = C_i + I$. Where I is the interference from higher priority tasks
- Check (trivially) if deadline is met $R_i \leq D_i$

Calculating I

- During R_i task j (with $P_j > P_i$) is released $\left\lceil \frac{R_i}{T_j} \right\rceil$ number of times.
- Total interference by task j is given by:

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- The ceiling function, $[x]$: the smallest integer greater than x , e.g., $[0.25] = 1$



Response-Time Equation

Worst Case Response Time

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of tasks with priority higher than task i

Solve by forming a recurrence relationship:

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

The set of values $R_i^0, R_i^1, R_i^2, \dots, R_i^n, \dots$ is monotonically non-decreasing.
When $R_i^n = R_i^{n+1}$ the solution to the equation has been found, R_i^0 , must
not be greater than R_i (use e.g., 0 or C_i)



Process Set C Revisited

$$R_c^0 = 5$$

$$R_b^0 = 10$$

$$R_a^0 = 40$$

	T(period)	C(exec.time)	P(priority)
--	-----------	--------------	-------------

a	80	40	1
b	40	10	2
c	20	5	3

$$R_i^{n+1} = C_i + \sum_{j>i} \left[\frac{R_j^n}{T_j} \right] \cdot C_j$$

Conclusion

$$R_c = 5 \leq 20 = T_c$$

$$R_b = 15 \leq 40 = T_b$$

$$R_a = 80 \leq 80 = T_a$$

So schedulable
!!



Process Set C Revisited

$$R_c^0 = 5$$

$$R_b^0 = 10$$

$$R_b^1 = 10 + \lceil \frac{10}{20} \rceil \cdot 5 = 10 + 5 = 15$$

$$R_b^2 = 10 + \lceil \frac{15}{20} \rceil \cdot 5 = 10 + 5 = 15$$

fixpoint

$$R_a^0 = 40$$

$$R_a^1 = 40 + \lceil \frac{40}{40} \rceil \cdot 10 + \lceil \frac{40}{20} \rceil \cdot 5 = 60$$

$$R_a^2 = 40 + \lceil \frac{60}{40} \rceil \cdot 10 + \lceil \frac{60}{20} \rceil \cdot 5 = 75$$

$$R_a^3 = 40 + \lceil \frac{75}{40} \rceil \cdot 10 + \lceil \frac{75}{20} \rceil \cdot 5 = 80$$

$$R_a^4 = 40 + \lceil \frac{80}{40} \rceil \cdot 10 + \lceil \frac{80}{20} \rceil \cdot 5 = 80$$



	T(period)	C(exec.time)	P(priority)
--	-----------	--------------	-------------

a	80	40	1
b	40	10	2
c	20	5	3

$$R_i^{n+1} = C_i + \sum_{j>i} \lceil \frac{R_j^n}{T_j} \rceil \cdot C_j$$

Conclusion

$$R_c = 5 \leq 20 = T_c$$

$$R_b = 15 \leq 40 = T_b$$

$$R_a = 80 \leq 80 = T_a$$

So schedulable



Process Set C Revisited

Example (Response Time Analysis for Process Set C)

Process	Period	Computation Time	Priority	Response Time
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

- The combined utilisation is 1.0
- This is **above** the (utilisation) threshold for three processes (0.78)
- The response time analysis shows that the process set will meet all its deadlines

Response Time Analysis

Necessary and sufficient

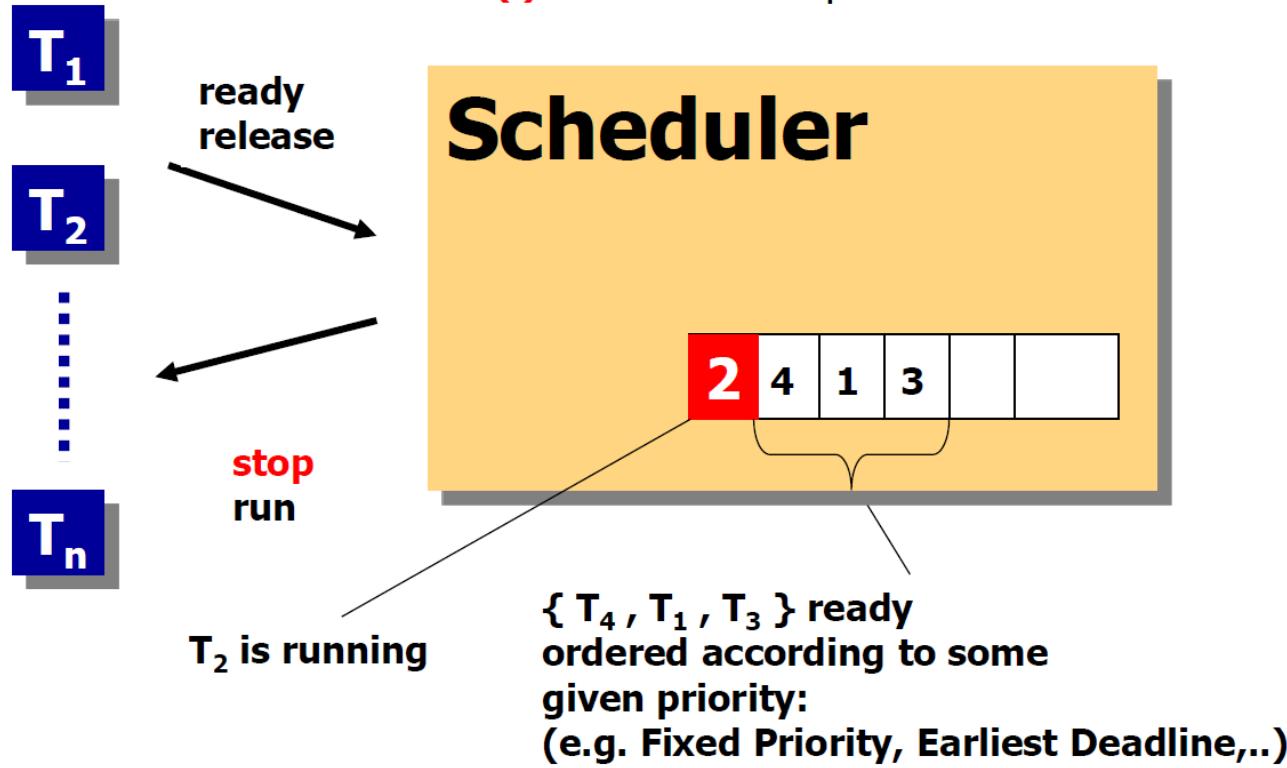


Schedulability Analysis in UPPAAL

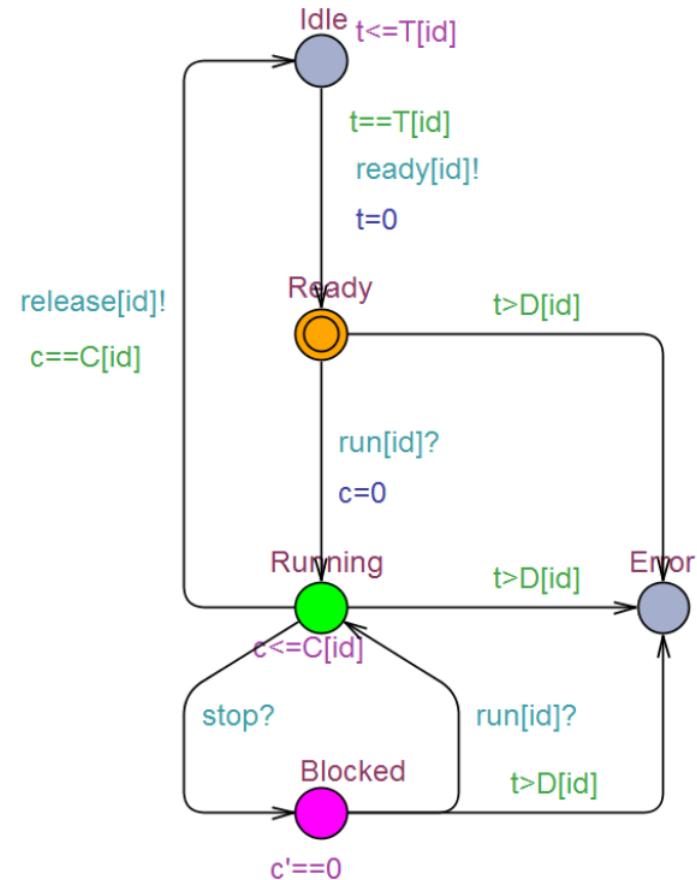
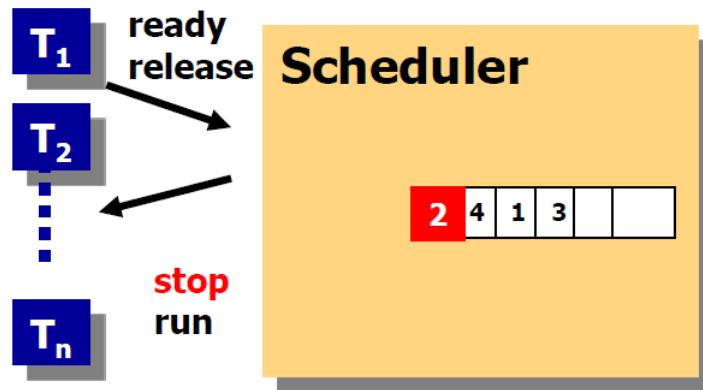


Task Scheduling

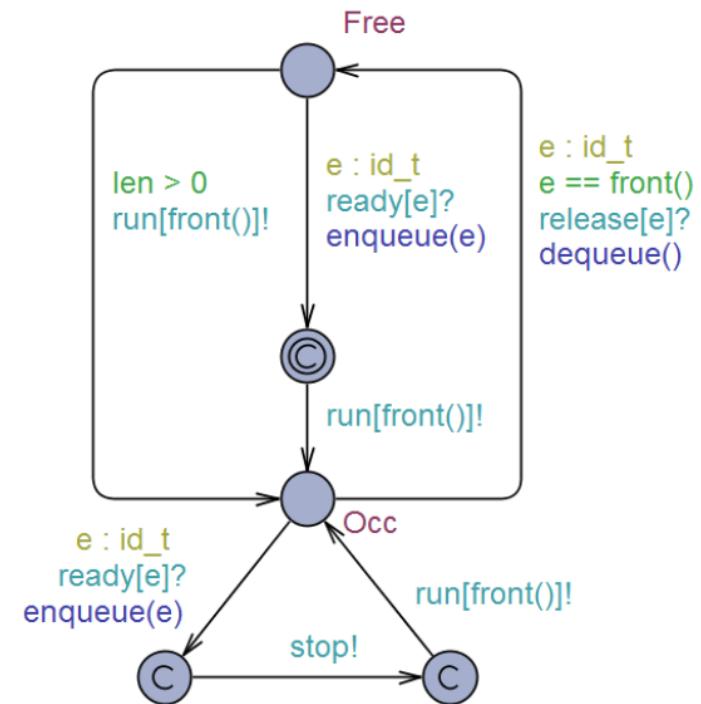
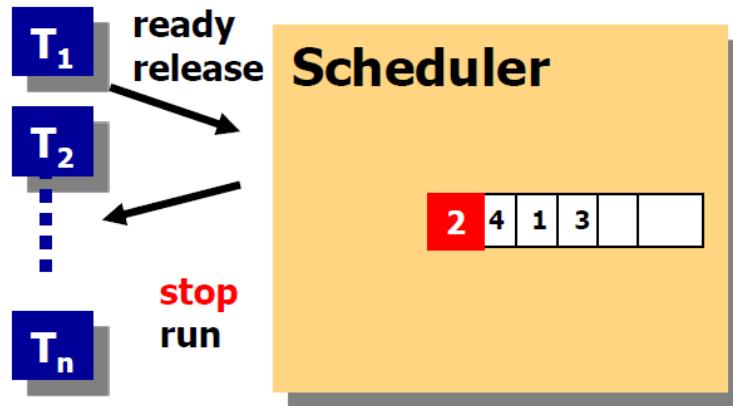
$T(i)$: period of task
 $C(i)$: execution time for T_i
 $D(i)$: deadline for T_i



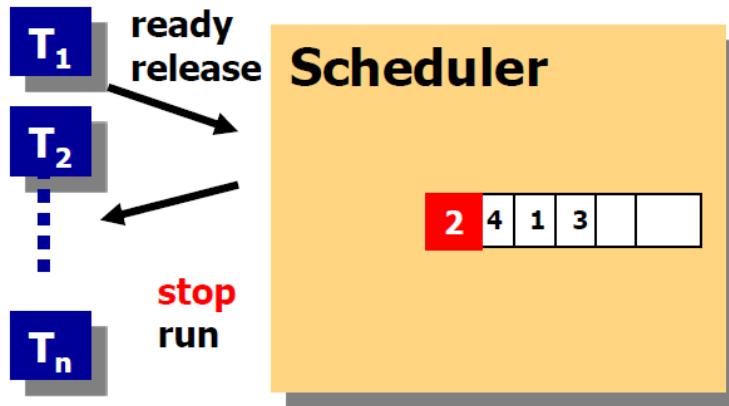
Modelling Task



Modelling Scheduler



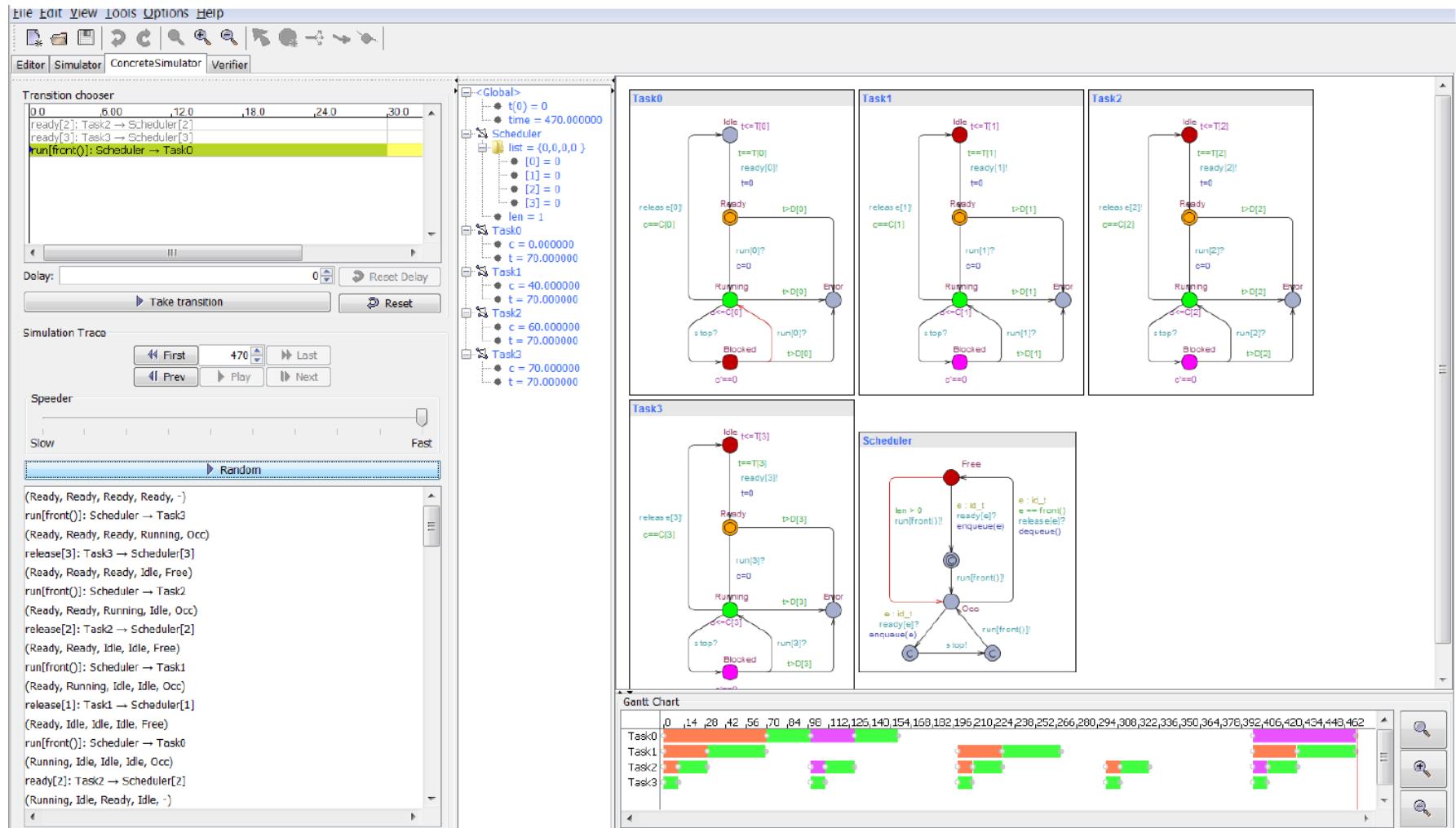
Modelling Queue



```
id_t list[N] = { 3, 2, 1, 0};  
int[N] len = N;  
  
// Put an element at the end of the queue  
void enqueue(id_t element)  
{  
    int tmp=0;  
    list[len++] = element;  
    if (len>0)  
    {  
        int i=len-1;  
        while (i>0 && P[list[i]]>P[list[i-1]])  
        {  
            tmp = list[i-1];  
            list[i-1] = list[i];  
            list[i] = tmp;  
            i--;  
        }  
    }  
}  
  
// Remove the front element of the queue  
void dequeue()  
{  
    int i = 0;  
    len -= 1;  
    while (i < len)  
    {  
        list[i] = list[i + 1];  
        i++;  
    }  
    list[i] = 0;  
}
```



Random Simulation



Verification

A[] not (Task0.Error or Task1.Error or Task2.Error or Task3.Error)



Herschel-Planck Scientific Mission ESA



Attitude and Orbit Control Software
TERMA A/S Steen Ulrik Palm, Jan Storbank Pedersen, Poul Hougaard

Herschel-Planck Mission

- **Application software (ASW)**
 - built and tested by Terma:
 - does attitude and orbit control, tele-commanding, fault detection isolation and recovery.
- **Basic software (BSW)**
 - low level communication and scheduling periodic events.
- **Real-time operating system (RTEMS)**
 - Priority Ceiling for ASW,
 - Priority Inheritance for BSW
- **Hardware**
 - single processor, a few buses, sensors and actuators

Application Software (ASW)

Basic Software (BSW)

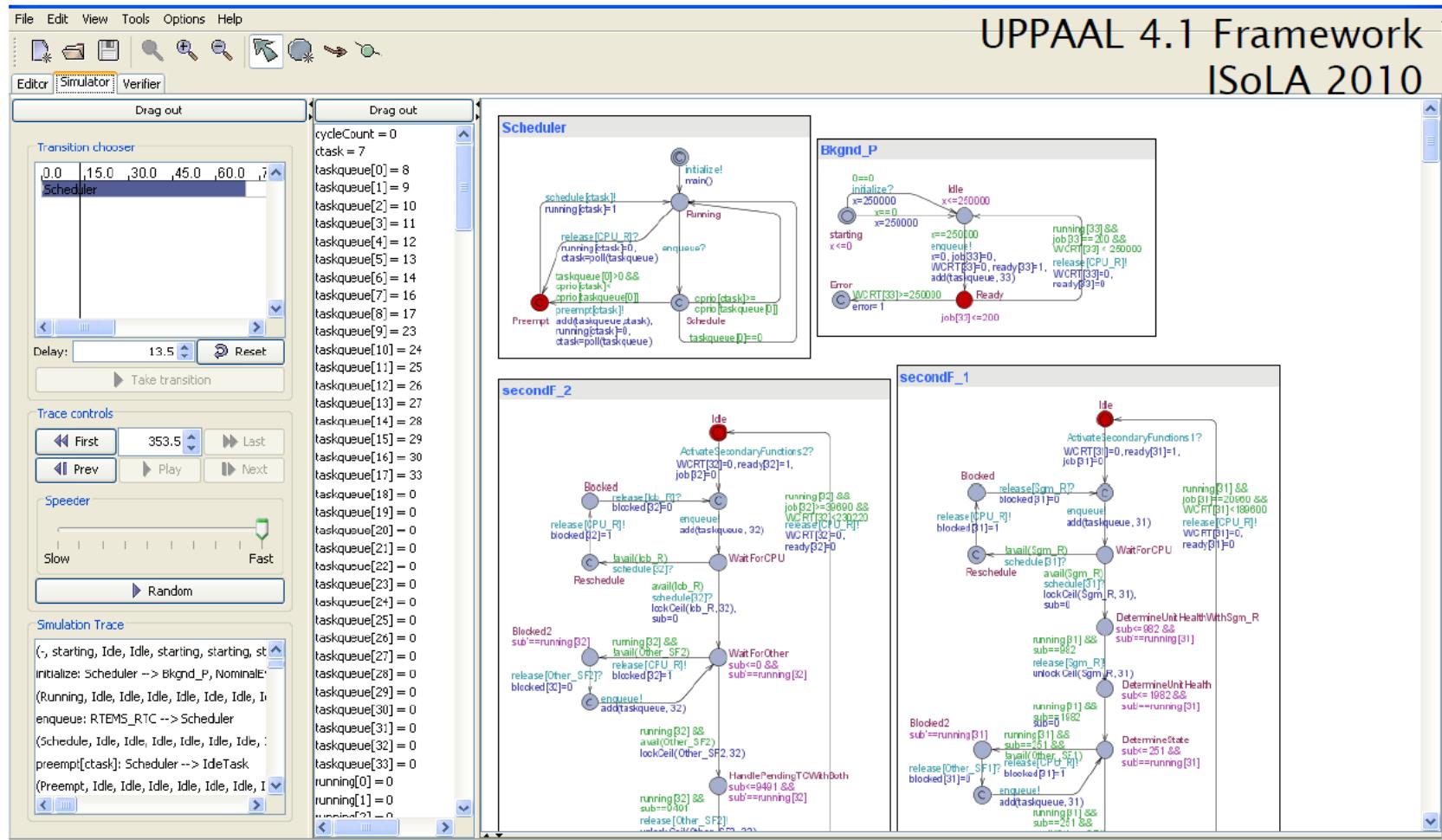
Hardware

Requirements:

Software tasks should be schedulable.
CPU utilization should not exceed 50% load



Modelling in UPPAAL



Blocking and Worst-Case Response-Time

ID	Task	Specification			Blocking times			WCRT		
		Period	WCET	Deadline	Terma	UPPAAL	Diff	Terma	UPPAAL	Diff
1	RTEMS_RTC	10.000	0.013	1.000	0.035	0	0.035	0.050	0.013	0.037
2	AswSync_SyncPulseIsr	250.000	0.070	1.000	0.035	0	0.035	0.120	0.083	0.037
3	Hk_SamplerIsr	125.000	0.070	1.000	0.035	0	0.035	0.120	0.070	0.050
4	SwCyc_CycStartIsr	250.000	0.200	1.000	0.035	0	0.035	0.320	0.103	0.217
5	SwCyc_CycEndIsr	250.000	0.100	1.000	0.035	0	0.035	0.220	0.113	0.107
6	Rt1553_Isr	15.625	0.070	1.000	0.035	0	0.035	0.290	0.173	0.117
7	Bc1553_Isr	20.000	0.070	1.000	0.035	0	0.035	0.360	0.243	0.117
8	Spw_Isr	39.000	0.070	2.000	0.035	0	0.035	0.430	0.313	0.117
9	Obdh_Isr	250.000	0.070	2.000	0.035	0	0.035	0.500	0.383	0.117
10	RtSdb_P_1	15.625	0.150	15.625	3.650	0	3.650	4.330	0.533	3.797
11	RtSdb_P_2	125.000	0.400	15.625	3.650	0	3.650	4.870	0.933	3.937
12	RtSdb_P_3	250.000	0.170	15.625	3.650	0	3.650	5.110	1.103	4.007
14	FdirEvents	250.000	5.000	230.220	0.720	0	0.720	7.180	5.153	2.027
15	NominalEvents_1	250.000	0.720	230.220	0.720	0	0.720	7.900	5.873	2.027
16	MainCycle	250.000	0.400	230.220	0.720	0	0.720	8.370	6.273	2.097
17	HkSampler_P_2	125.000	0.500	62.500	3.650	0	3.650	11.960	5.380	6.580
18	HkSampler_P_1	250.000	6.000	62.500	3.650	0	3.650	18.460	11.615	6.845
19	Acb_P	250.000	6.000	50.000	3.650	0	3.650	24.680	6.473	18.207
20	IoCyc_P	250.000	3.000	50.000	3.650	0	3.650	27.820	9.473	18.347
21	PrimaryF	250.000	34.050	59.600	5.770	0.966	4.804	65.470	54.115	11.355
22	RCSControlF	250.000	4.070	239.600	12.120	0	12.120	76.040	53.994	22.046
23	Obt_P	1000.000	1.100	100.000	9.630	0	9.630	74.720	2.503	72.217
24	Hk_P	250.000	2.750	250.000	1.035	0	1.035	6.800	4.953	1.847
25	StsMon_P	250.000	3.300	125.000	16.070	0.822	15.248	85.050	17.863	67.187
26	TmGen_P	250.000	4.860	250.000	4.260	0	4.260	77.650	9.813	67.837
27	Sgm_P	250.000	4.020	250.000	1.040	0	1.040	18.680	14.796	3.884
28	TcRouter_P	250.000	0.500	250.000	1.035	0	1.035	19.310	11.896	7.414
29	Cmd_P	250.000	14.000	250.000	26.110	1.262	24.848	114.920	94.346	20.574
30	NominalEvents_2	250.000	1.780	230.220	12.480	0	12.480	102.760	65.177	37.583
31	SecondaryF_1	250.000	20.960	189.600	27.650	0	27.650	141.550	110.666	30.884
32	SecondaryF_2	250.000	39.690	230.220	48.450	0	48.450	204.050	154.556	49.494
33	Bkgnd_P	250.000	0.200	250.000	0.000	0	0.000	154.090	15.046	139.044



Mini Project



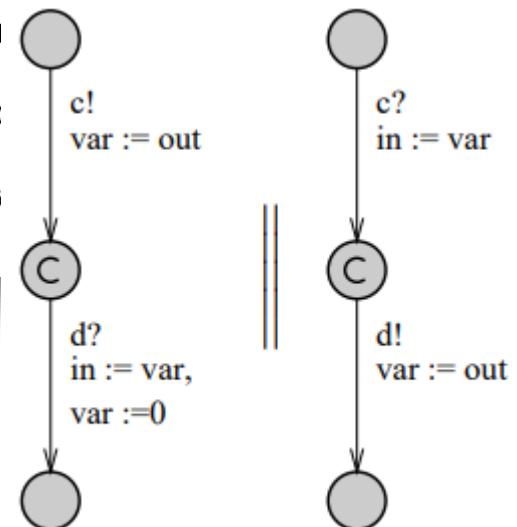
Gossiping Girls - April 9th



Model and analyze the following gossiping girls problem in UPPAAL. A number of girls initially know one distinct secret each. Each girl has access to a phone which can be used to call another girl to share their secrets. Each time two girls talk to each other they always exchange all secrets with each other (thus after the phone call they both know all secrets they knew together before the phone call). The girls can communicate only in pairs (no conference calls) but it is possible that different pairs of girls talk concurrently. For all of the tasks below you should consider the following three scenarios:

- Scenario 1: all girls may directly call any other girl, thus telephone calls can overlap.
- Scenario 2: only a single call in the entire network can take place at a time.
- Scenario 3: the girls are organized in a *linear chain* (with a first and last girl) and calls can only be made between neighboring persons.

Hints: Get inspiration at TRAIN gate.
Representation of secrets.
Value passing



Principles of Cyber-Physical Systems

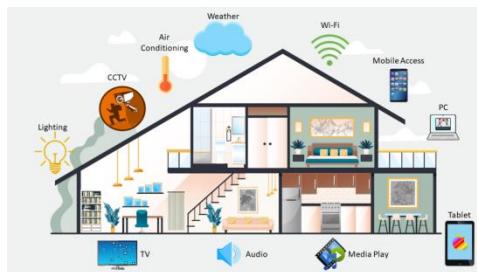
Dynamical Systems - Part 1

Instructor: Max Tschaikowski
tschaikowski@cs.aau.dk

Slides Courtesy of Rajeev Alur



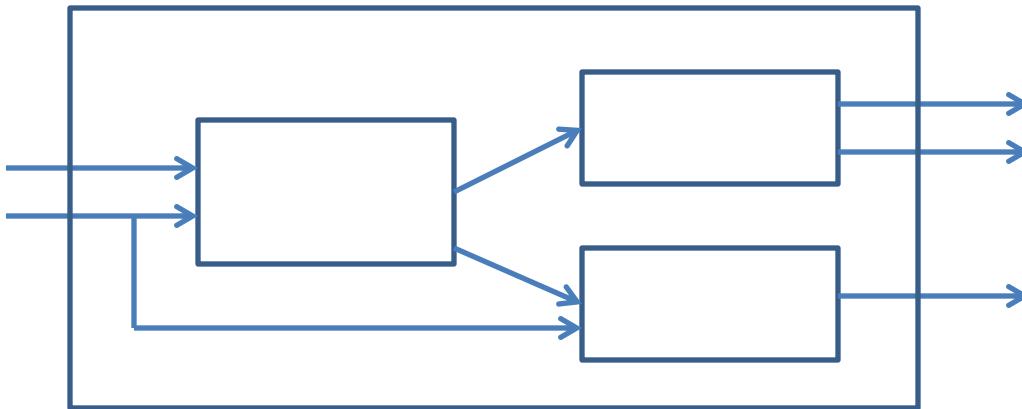
Dynamical Systems examples



Dynamical Systems

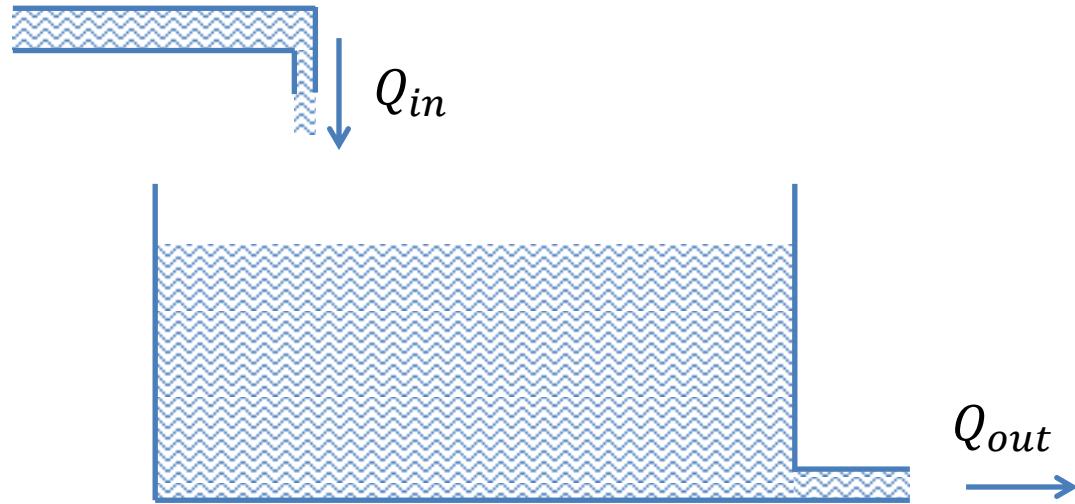
- Controller (software) interacting with the physical world via sensors and actuators
 - Thermostat controlling temperature
 - Cruise controller regulating speed of a car
- Variables: Physical quantities evolving continuously over time
 - Temperature, pressure, velocity ...
- Governed by nature's laws
 - Newton's laws
 - Laws of thermodynamics
 - Maxwell's equations
- Continuous-time models using differential equations

Model-Based Design



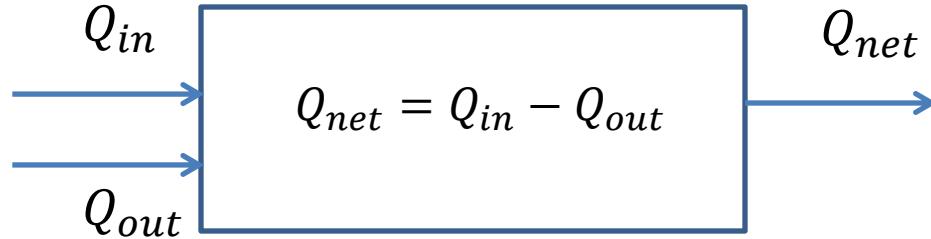
- Block Diagrams
 - Widely used in industrial design
 - Tools: Simulink, Modelica, RationalRose...
- Key question: what is the execution semantics?
 - Similar to synchronous model, but continuous-time instead of discrete-time

Example: Water Flow in a Tank



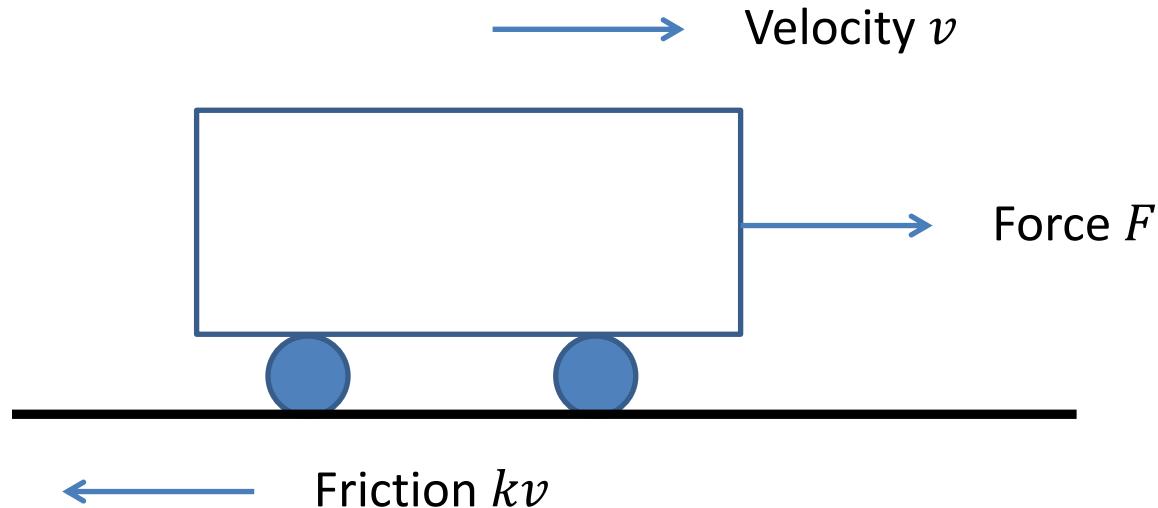
- Inflow of water Q_{in}
- Outflow of water Q_{out}

Example: Water Flow in a Tank



- Input variables: Q_{in} and Q_{out} of type real
- Output variable: Q_{net} of type real
- No state variables
- Signal: assignment of values to variables as function of time t
- At each time t , value of output signal $Q_{net}(t)$ equals $Q_{in}(t) - Q_{out}(t)$
- Output as a function of inputs/state specified using algebraic equations (as opposed to assignments)

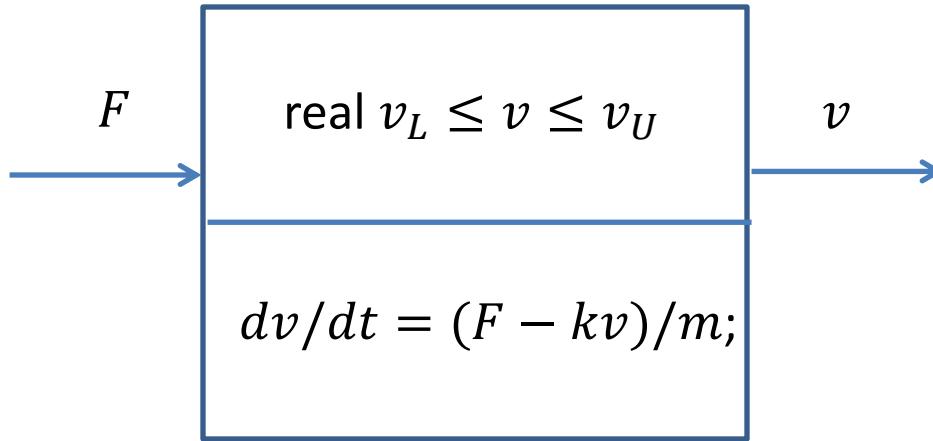
Example: Car Model



Newton's law of motion gives

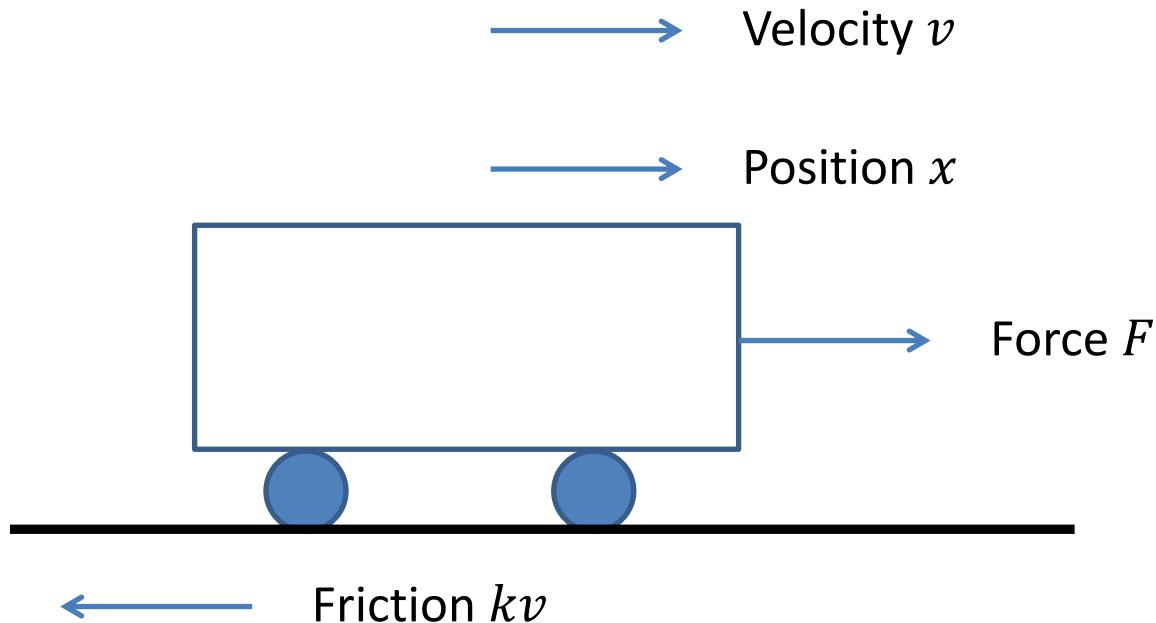
$$\begin{aligned} F_{net} &= ma \rightarrow \\ F - kv &= m dv/dt \end{aligned}$$

Continuous-time Component Car



- For state variable v , its rate of change dv/dt is defined using an expression over input and state variables
- For each output variable, its value is defined using an expression over input and state variables.

Example: Car Model with two variables

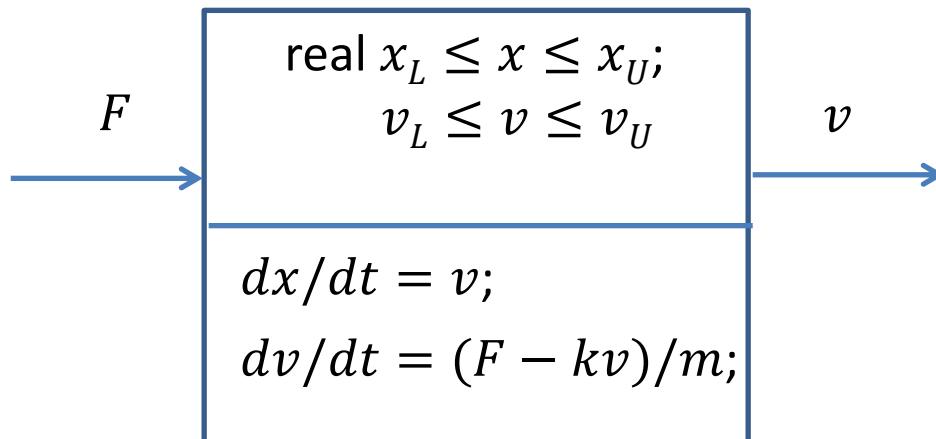


Newton's law of motion gives

$$F_{net} = ma \rightarrow$$
$$F - k\nu = m d^2x/dt$$

Continuous-time Component Car

- Remember that $a = dv/dt$ and $v = dx/dt$
- We can rewrite $F - kv = m d^2x/dt$ into the system below using state variables x and v .

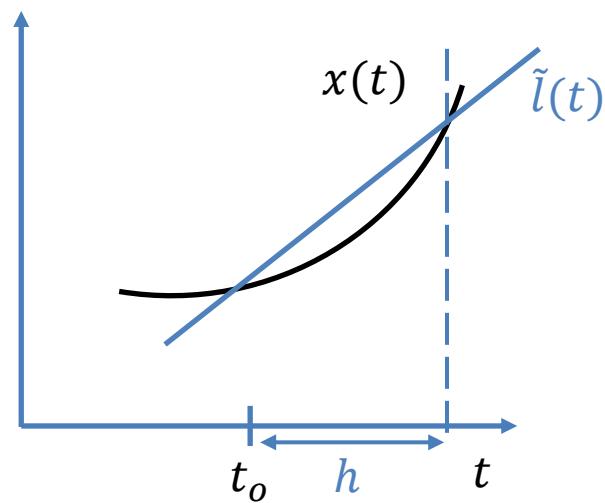
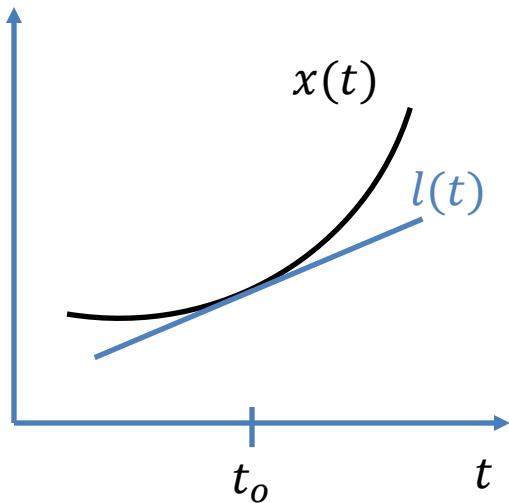


Executions of Car

- Input signal: Function $F(t) : \text{real}_{\geq 0} \rightarrow \text{real}$ that gives value of force as a function of time
 - Should be continuous or piecewise-continuous
- Given an initial state (x_0, v_0) and input signal $F(t)$, the execution of the system is defined by state-signals $x(t)$ and $v(t)$ (from $\text{real}_{\geq 0}$ to real) that satisfy the initial-value problem:
 - $x(0) = x_0; v(0) = v_0$
 - $dx(t)/dt = v(t)$
 - $dv(t)/dt = (F(t) - kv(t)) / m$

Calculus Brush Up

- $\frac{d}{dt}x(t)$ captures the rate of change of variable x at time t , i.e. the slope
- $\frac{d}{dt}x(t)$ is defined as the limit of the difference: $\frac{d}{dt}x(t) = \lim_{h \rightarrow 0} \frac{x(t+h)-x(t)}{h}$



$$l(t) = x(t_0) + \frac{d}{dt}x(t_0)(t - t_0)$$

$$l(t) \approx \tilde{l}(t) = x(t_0) + \left(\frac{x(t_0 + h) - x(t_0)}{h} \right) (t - t_0)$$

Algorithms for solving differential equations

- Exact formulation:

$$\frac{d}{dt}x(t) = f(x(t))$$

- Approximation:

$$\frac{x(t+h) - x(t)}{h} \approx f(x(t))$$

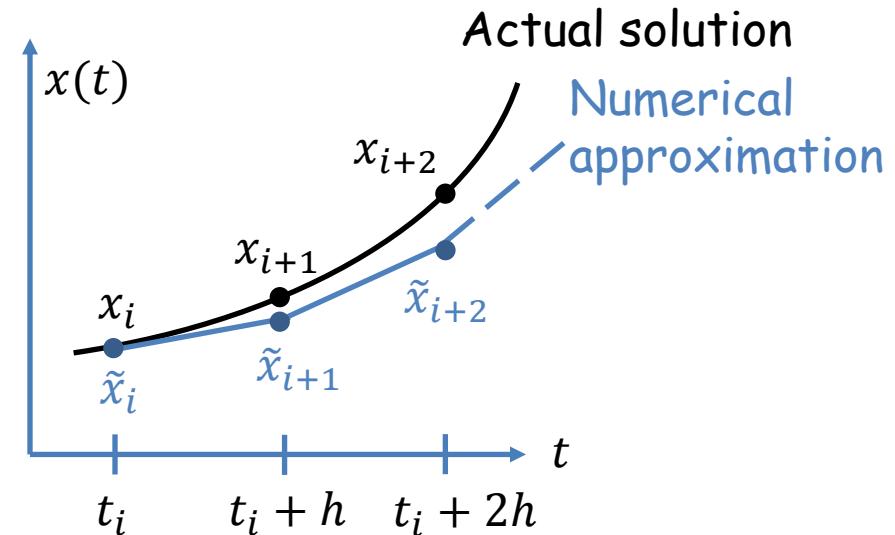
\Leftrightarrow

$$x(t+h) \approx x(t) + h \cdot f(x(t))$$

- Euler method:

Input: f, x_0, h, N

```
 $\tilde{x}_0 \leftarrow x_0$ 
for  $i \in \{0, 1, \dots, N\}$  do
    print  $\tilde{x}_i$ 
     $\tilde{x}_{i+1} \leftarrow \tilde{x}_i + h \cdot f(\tilde{x}_i)$ 
end
```



- Theorem

$\exists C > 0. \forall h > 0. \forall i \in \{0, 1, \dots, N\}:$

$$|\tilde{x}_i - x_i| \leq C \cdot h$$

So, the global truncation error is proportional to h .

Example of the Euler method

□ Let $\frac{d}{dt}x = t$, $x_0 = 0$, $h = 1$, and $N = 5$

□ The exact solution is $x(t) = \frac{1}{2}t^2$

Iteration i	Exact solution x_i	Numerical solution \tilde{x}_i
0	0	0
1	0.5	$0 + 1 \cdot 0 = 0$
2	2	$0 + 1 \cdot 1 = 1$
3	4.5	$1 + 1 \cdot 2 = 3$
4	8	$3 + 1 \cdot 3 = 6$
5	12.5	$6 + 1 \cdot 4 = 10$

□ Euler method:

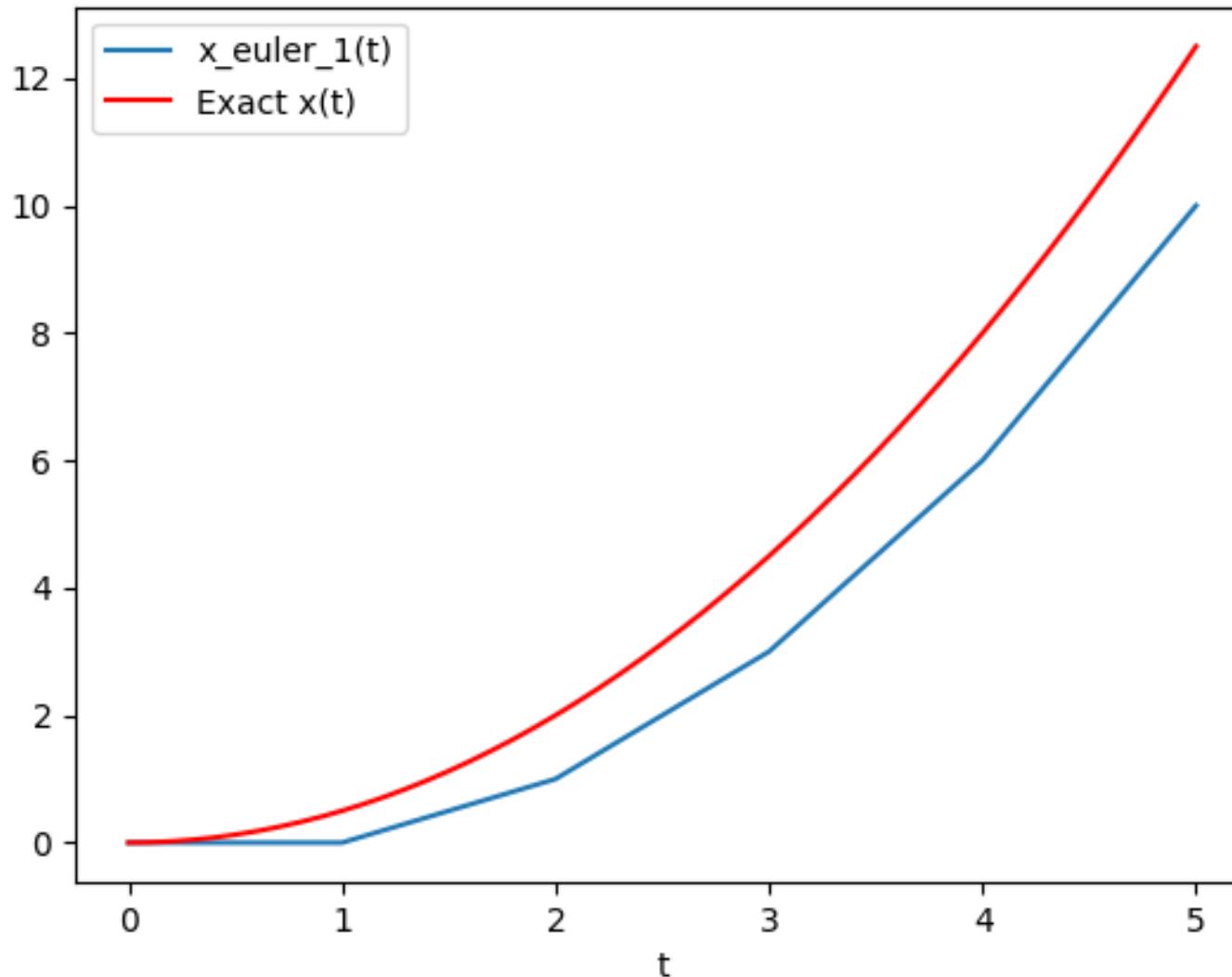
Input: f, x_0, h, N

```
 $\tilde{x}_0 \leftarrow x_0$ 
for  $i \in \{0, 1, \dots, N\}$  do
    print  $\tilde{x}_i$ 
     $\tilde{x}_{i+1} \leftarrow \tilde{x}_i + h \cdot f(\tilde{x}_i)$ 
end
```

□ Notice that $|\tilde{x}_i - x_i| = \frac{1}{2}i$

Precision of the Euler method

- Solution plotted with Euler method with different step sizes h



Algorithms for solving differential equations

- The Euler method assumes that the rate of change is constant during the interval, and is only based on the state at the beginning of the interval.
 - This is often *not* the case for dynamical systems.
 - We could take the endpoint into account as well.
 - Runge-Kutta methods are based on this idea
-
- Second-order Runge-Kutta method uses the following calculations:

$$\begin{aligned} k_1 &= f(x(t)) && \leftarrow \text{Slope at beginning of interval} \\ k_2 &= f(\tilde{x}_i + h \cdot k_1) && \leftarrow \text{Estimated slope at end of interval} \\ \tilde{x}_{i+1} &= \tilde{x}_i + h \cdot (k_1 + k_2)/2 && \leftarrow \text{Euler method, but now average slope} \end{aligned}$$

- Most commonly used method is the fourth-order Runge-Kutta method.

Algorithms for solving differential equations

- Most used method is the fourth-order Runge-Kutta method.
- It takes four slopes into account: start, two midpoints and endpoint.
- $\tilde{x}_{i+1} = \tilde{x}_i + h(k_1 + 2k_2 + 2k_3 + k_4)/6$
- See book for expressions of k_1, k_2, k_3, k_4 .
- Theorem

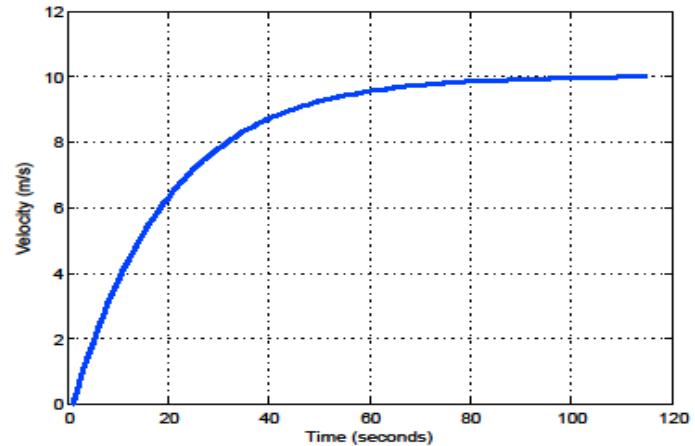
$\exists C > 0. \forall h > 0. \forall i \in \{0, 1, \dots, N\}$:

$$|\tilde{x}_i - x_i| \leq C \cdot h^4$$

- Note that for $0 < h < 1$: $h^4 \ll h$
- So, the fourth-order Runge-Kutta method is more efficient than the Euler method.

Back to Car Executions

- Suppose initial position is 0, initial velocity is 0, and force is constant F_0 . Then, to get executions, we need to solve:
 - $x(0) = 0; v(0) = 0$
 - $dx(t)/dt = v(t)$
 - $dv(t)/dt = (F_0 - k v(t)) / m$
- Compute the solution using MATLAB
 - Mass $m = 1000\text{kg}$
 - Coefficient of friction $k = 50$
 - Force $F_0 = 500$ Newton
 - Velocity converges to 10 m/s



Definition: Continuous-Time Component

- Set I of real-valued input variables
 - Type is either real or interval of real, $\text{real}[L, U]$
- Set O of real-valued output variables
- Set S of real-valued state variables
- Initialization Init specifying set $[\text{Init}]$ of initial states
- For each output var y , a real-valued expression h_y over $I \cup S$
- For each state variable x , a real-valued expression f_x over $I \cup S$
- Given an input-signal $I(t) : \text{real}_{>=0} \rightarrow \text{real}^{|I|}$, an execution consists of a *differentiable* state signal $S(t)$ and output signal $O(t)$ such that
 1. $S(0)$ is in $[\text{Init}]$
 2. For each output var y and time t , $y(t) = h_y(I(t), S(t))$
 3. For each state var x , $(d/dt)(x(t)) = f_x(I(t), S(t))$

Existence and Uniqueness

- Given an input signal $I(t)$, when are we guaranteed that the system has at least one execution? Exactly one execution?
- The input signal should be continuous (or at least piecewise continuous), but also depends on right-hand-sides of equations defining state and output dynamics
- Related to classical theory of ODEs (Ordinary Differential Equations)
- Consider the initial value problem
$$dx/dt = f(x); x(0) = x_0, x \text{ is } k\text{-dimensional signal}$$
- When does there exist a unique differentiable function $x(t)$ as a solution?

Existence

- Consider the initial value problem

$$dx/dt = f(x); x(0) = x_0, x \text{ is } k\text{-dimensional signal}$$

- There exists at least one solution $x(t)$ if the function f is a continuous function
- Definition of continuity relies on definition of distance between points (e.g. Euclidean distance)
- The function f is (uniformly) continuous if for all $\varepsilon > 0$, there exists $\delta > 0$ such that for all u, v in real^k , if $\|u - v\| < \delta$ then $\|f(u) - f(v)\| < \varepsilon$
- Example when solution does not exist:
$$dx/dt = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{else} \end{cases}$$
- Natural to require all right-hand-side expressions h_y and f_x in definition of a continuous-time component to be continuous
 - Discontinuous case \rightarrow Hybrid Systems

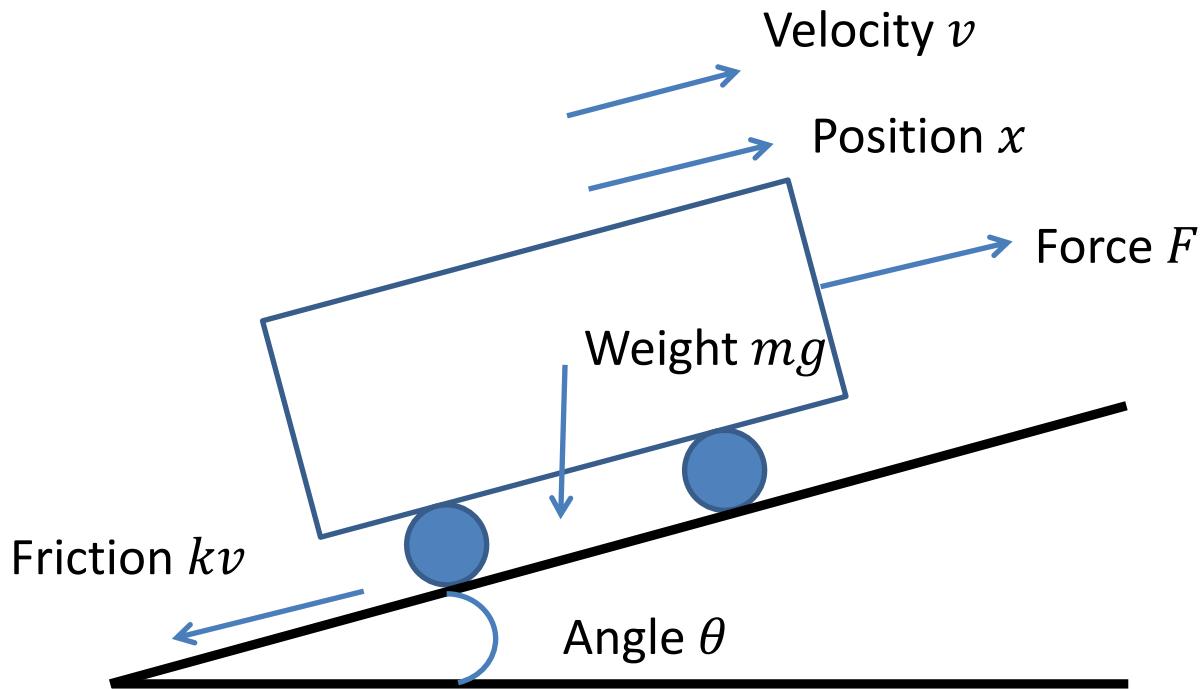
Uniqueness

- Consider the initial value problem
$$dx/dt = f(x); x(0) = x_0, x \text{ is } k\text{-dimensional signal}$$
- Picard-Lindelöf Theorem: There exists a unique solution $x(t)$ if the function f is Lipschitz-continuous
- Informally, Lipschitz-continuous means there is a constant upper bound on how fast a function changes
- Function f is Lipschitz-continuous if there exists a constant K such that for all u, v in real^k , $\|f(u) - f(v)\| \leq K\|u - v\|$
- Examples:
 - A linear function such as $(F - kv)/m$ is Lipschitz-cont
 - Quadratic function x^2 ? Lipschitz-cont if domain of x is bounded
- $x^{1/3}$ is not Lipschitz-cont: $dx/dt = x^{1/3}; x(0) = 0$ has multiple solutions
 - $x(t) = 0$
 - $x(t) = (2t/3)^{3/2}$

Lipschitz Continuous Component

- A continuous-time component has Lipschitz-continuous dynamics if
 - Each expression h_y corresponding to output variable y is a Lipschitz-continuous function of $I \cup S$
 - Each expression f_x corresponding to state variable x is a Lipschitz-continuous function over $I \cup S$
- If we supply a Lipschitz continuous component with a continuous input signal $I(t)$, then it has a unique response signals $S(t)$ and $O(t)$, both of which are continuous
- Note: Continuity of output signals means these can be fed to other components in a block diagram
- Henceforth, we will assume all components are Lipschitz continuous

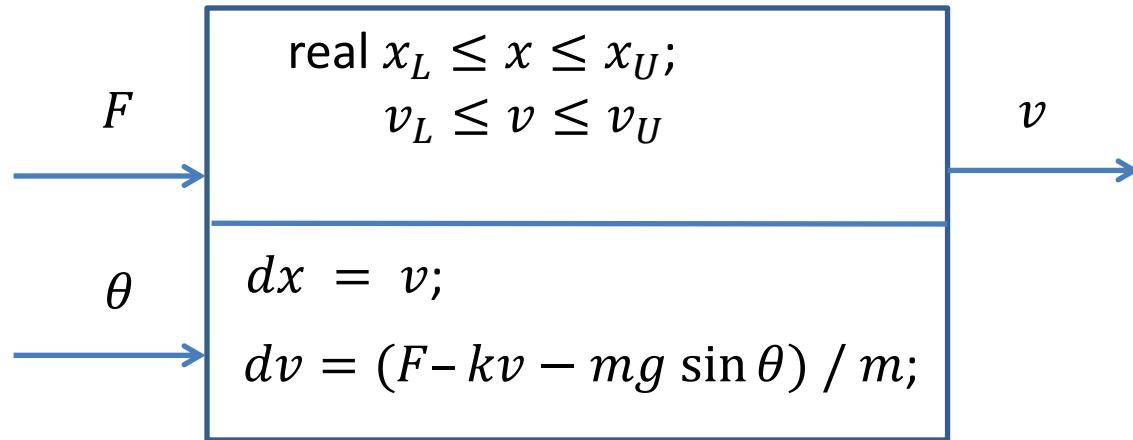
Car on a graded road



Newton's law of motion gives

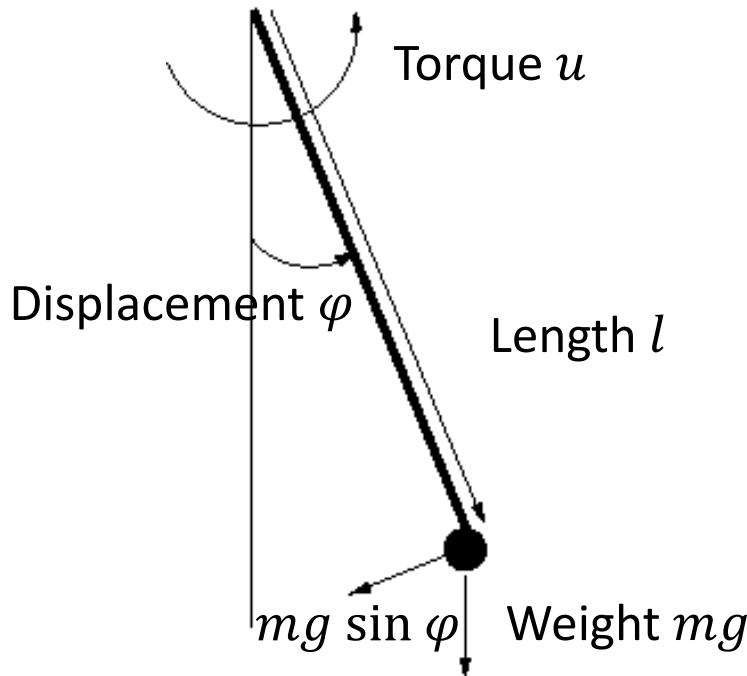
$$F - k\nu - mg \sin \theta = m d^2x/dt^2$$

Continuous-time Component Car2



- The grade of the road, denoted θ , models disturbance or an uncontrolled input

Simple Pendulum



- External torque applied by the motor at the pivot: u
- Dynamics captured by the second-order non-linear differential eqn:

$$ml^2 (d^2/dt^2)\varphi = u - mg l \sin \varphi$$

Pendulum Model

- Dynamics captured by the second-order non-linear differential eqn:

$$ml^2 (d^2/dt^2)\varphi = u - mgl \sin \varphi$$

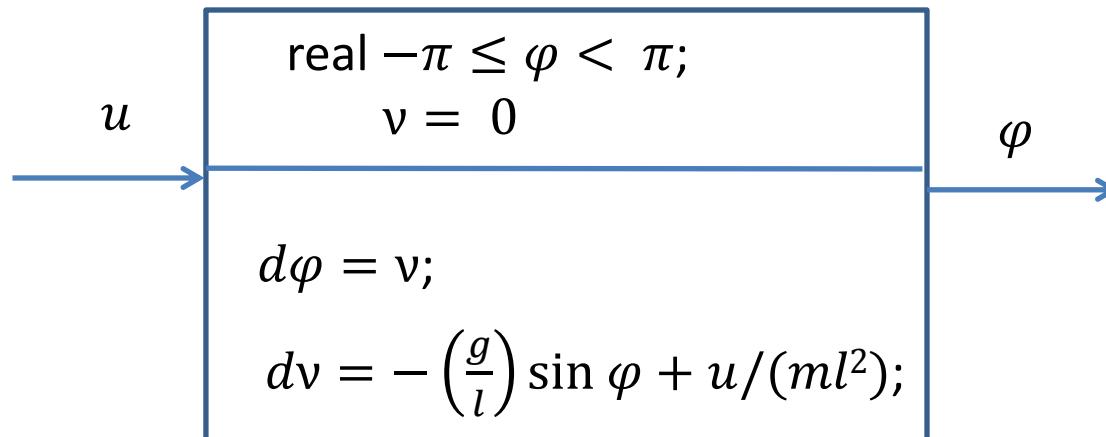
- Note that $\frac{d}{dt}\varphi = v$, so $\frac{d^2}{dt^2}\varphi = \frac{d}{dt}\left(\frac{d}{dt}\varphi\right) = \frac{d}{dt}v$

- With rewriting

$$ml^2 \left(\frac{d^2}{dt^2}\right)\varphi = u - mgl \sin \varphi$$

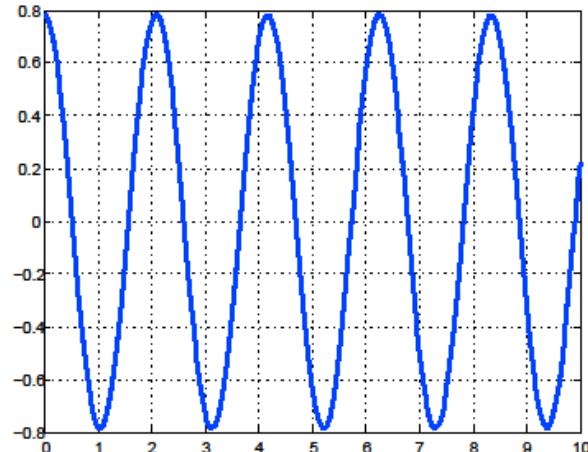
$$ml^2 \frac{d}{dt}v = u - mgl \sin \varphi$$

$$\frac{d}{dt}v = \frac{u}{ml^2} - \frac{g}{l} \sin \varphi$$



Angular Displacement

- External torque = 0; Initial displacement = $\pi/4$
- Oscillatory motion plotted by MATLAB
- Consider $dx/dt = f(x)$. An x satisfying $0 = f(x)$ is called equilibrium
- What are equilibria of this pendulum ?



Equilibria of Dynamical Systems

- Consider a closed (i.e. without inputs) continuous-time component H
 - If H has inputs, then we can analyze equilibria by setting inputs to a fixed value
 - Assume state x is k -dimensional, and dynamics is Lipschitz-continuous given by $dx/dt = f(x)$
- A state x_e is called an equilibrium of H if $f(x_e) = 0$
- If initial state of H equals an equilibrium state x_e , then the system stays in this state at all times

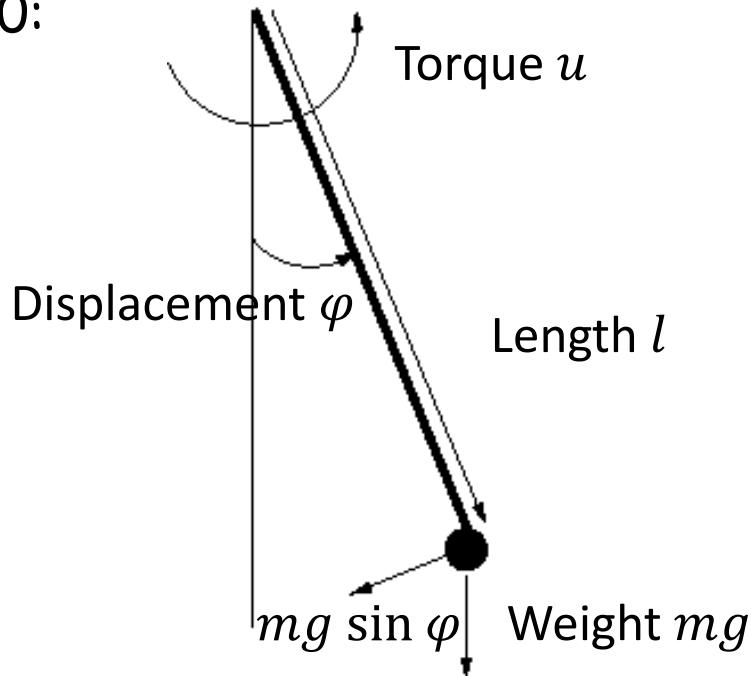
Pendulum Equilibria

Dynamics when external torque is 0:

$$d\varphi = v; dv = -\left(\frac{g}{l}\right) \sin \varphi$$

Equilibrium states:

$$v = 0; \sin \varphi = 0$$



Equilibrium state 1: $v = 0; \varphi = 0$; Pendulum is vertically downwards

Equilibrium state 2: $v = 0; \varphi = -\pi$; Pendulum is vertically upwards

Stability of Dynamical Systems

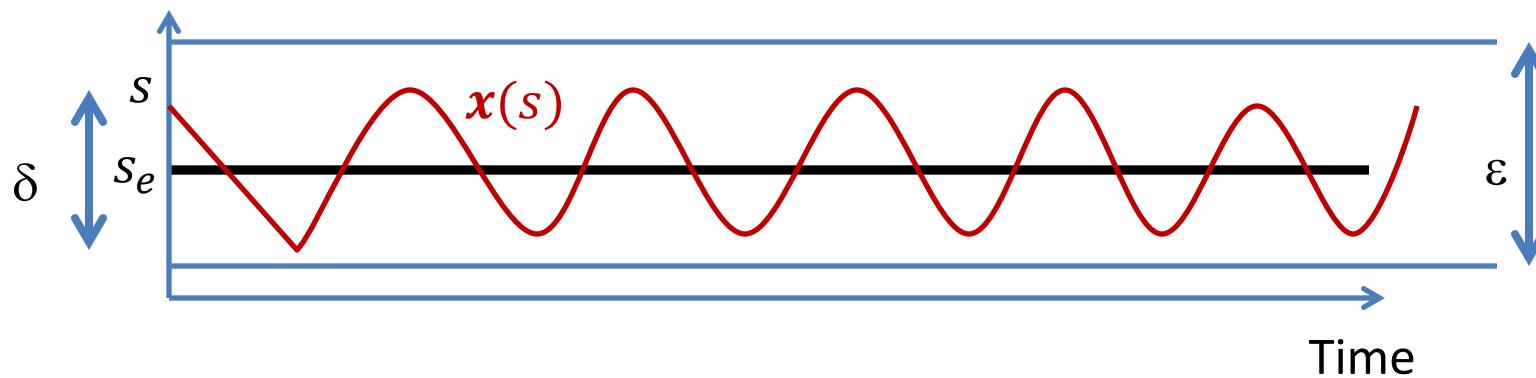
- Key correctness requirement for dynamical systems: stability
 - Small perturbations in the input values should not cause disproportionately large changes in the outputs
- For cruise controller, correctness requirements:
 - Safety: Speed should always be within certain threshold values
 - Liveness: Actual speed should eventually get close to desired speed
 - Stability: If grade of the road changes, speed should change only slowly
- Classical mathematical formalization of stability:
 - Lyapunov stability of equilibria

Lyapunov Stability

- Consider a closed continuous-time component H with Lipschitz-continuous dynamics $dx/dt = f(x)$
- Given an initial state s , let $x[s]$ denote the unique state response signal for the initial value problem $x(0) = s$ and $dx/dt = f(x)$
- Consider an equilibrium state s_e : if initial state is s_e then the response $x[s_e]$ is a constant function of time, always equal to s_e
- Stability of an equilibrium: when the system is in an equilibrium state, if we perturb its state slightly
 - As time passes, will the state stay close to the equilibrium state ?
 - As time passes, will the system eventually return to the equilibrium state?

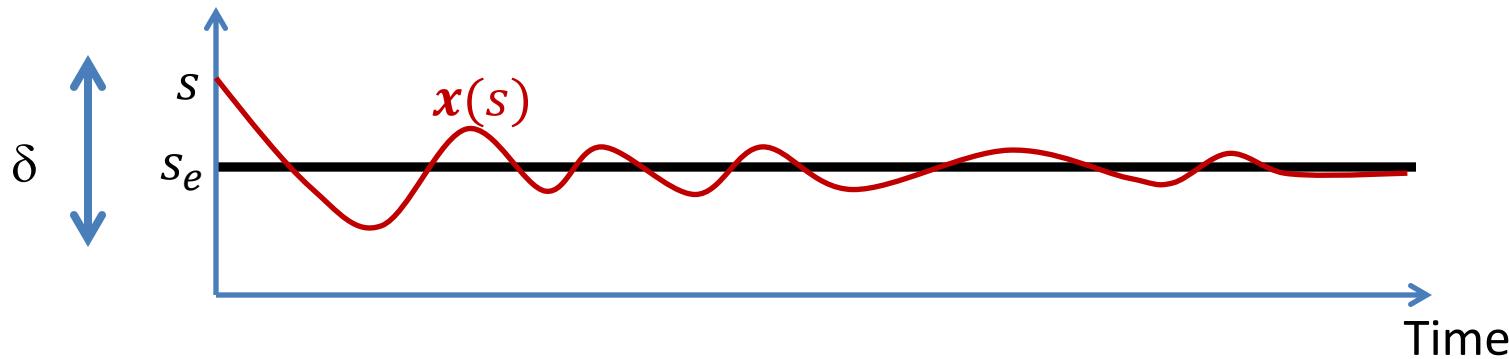
Lyapunov Stability Conditions

- Suppose the initial state s is *close* to an equilibrium state s_e , does the state along the response signal $x[s]$ stay close to s_e ?
 - If so, the equilibrium s_e is said to be **stable**
 - Formally, for every $\varepsilon > 0$, there exists $\delta > 0$ such that for all states s with $\|s - s_e\| < \delta$, for all times $t \geq 0$, $\|x[s](t) - s_e\| < \varepsilon$



Lyapunov Stability Conditions

- If in addition, the response signal $x[s]$ converges to the equilibrium state s_e , then the equilibrium is **asymptotically stable**
 - There exists $\delta > 0$ such that for all states s if $\|s - s_e\| < \delta$ then the limit $\lim_{t \rightarrow \infty} x(s)(t)$ exists and equals s_e
 - Note: This is a stronger condition than stable



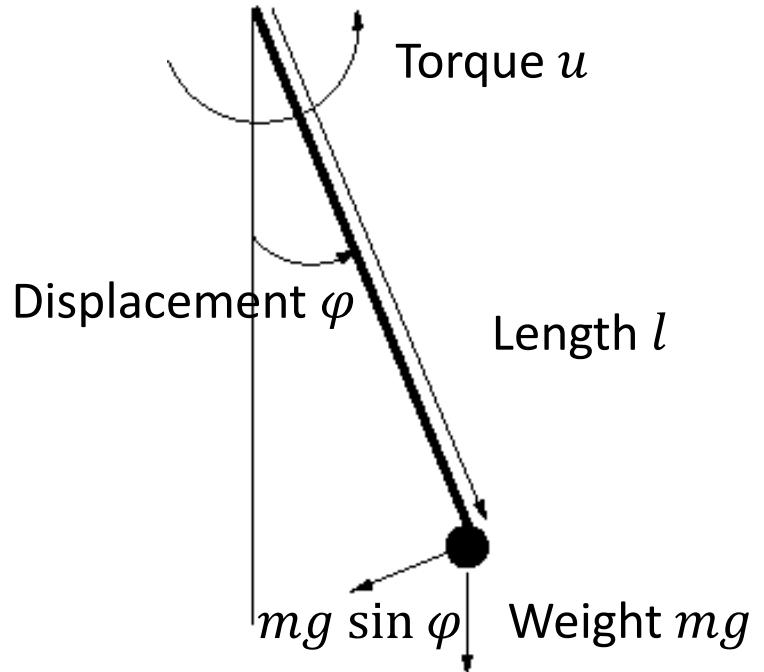
Pendulum Equilibria

Equilibrium state 1: $v = 0; \varphi = 0$;
Pendulum is vertically downwards.

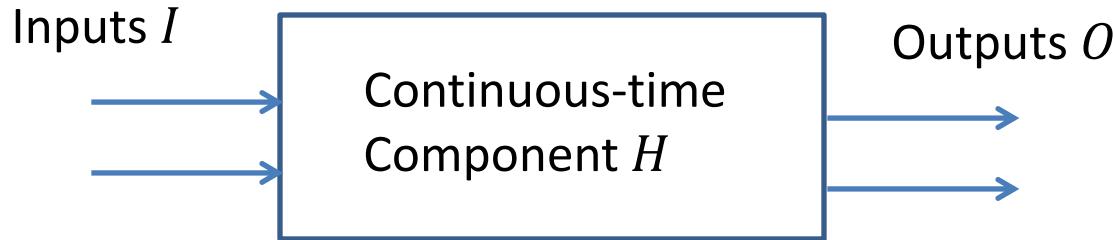
Stable, but not asymptotically stable

Equilibrium state 2: $v = 0; \varphi = -\pi$;
Pendulum is vertically upwards

Unstable !

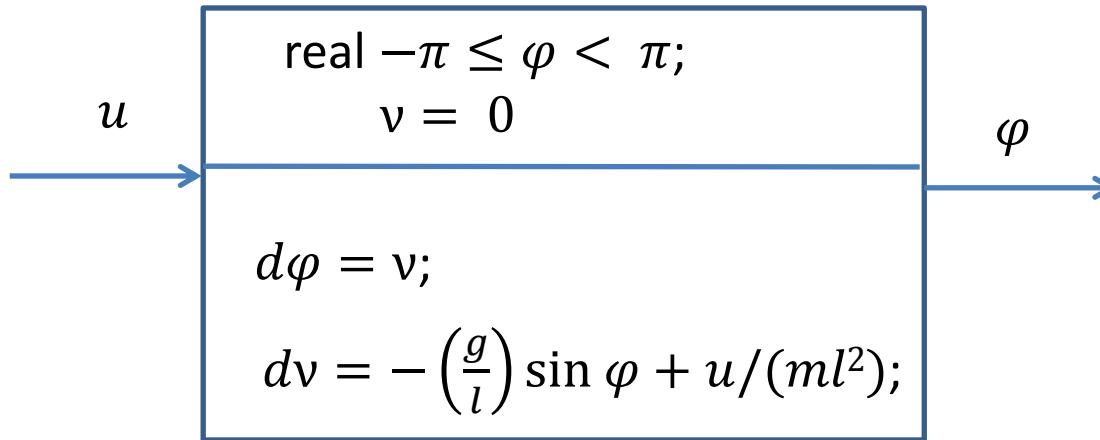


Input-Output Stability



- A continuous-time component H maps input signals $I(t)$ to output signals $O(t)$
- Input-output stability: If we change the input signal slightly, the output signal should change only slightly

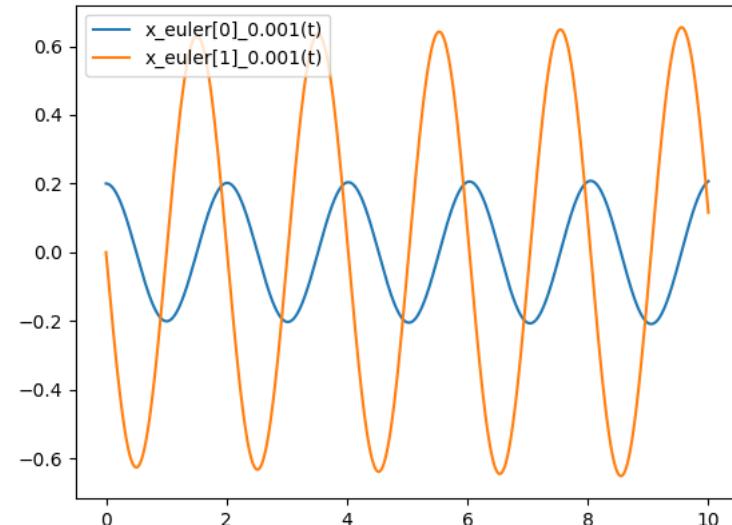
Simulation of the pendulum in Python



```
def f(x, t):
    # Implements the pendulum function
    without input
    g = 9.81
    l = 1
    result = [0, 0]

    result[0] = x[1]
    result[1] = -(g/l)*math.sin(x[0])

    return result
```



Equilibrium analysis with numerical methods

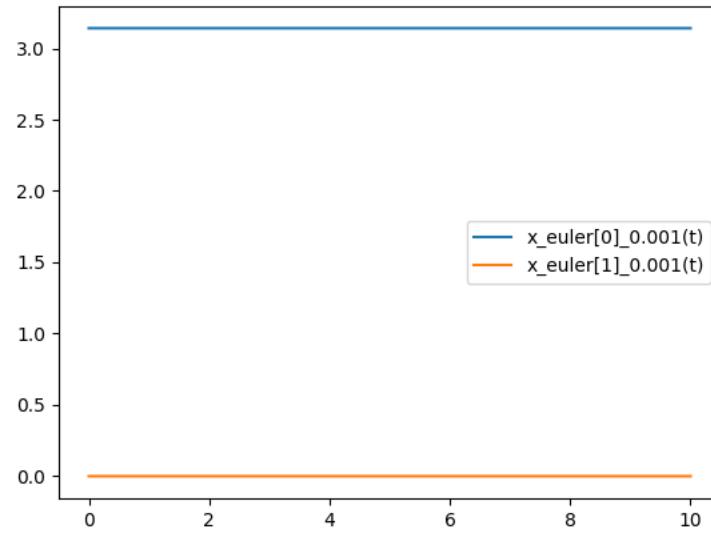
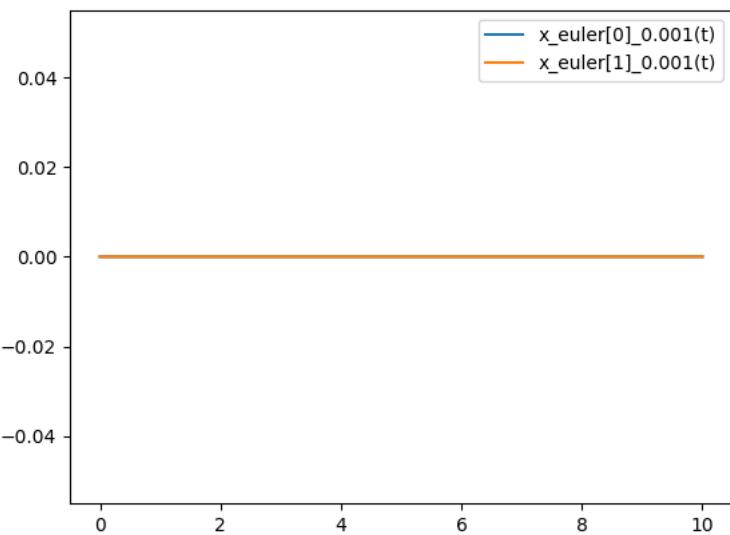
- Consider equilibrium state 1: $\varphi = 0; v = 0;$

In Python: $x0 = [0, 0]$

- Consider equilibrium state 2: $\varphi = -\pi; v = 0;$

In Python: $x0 = [3.14, 0]$

Better initial state: $x0 = [\text{math.pi}, 0]$



Instability

- If a continuous system is stable, it is considered to be correct.
- But what if a continuous system is unstable? Should we say to the mechanical engineers that they did a bad job at designing the dynamical system?
- Sometimes you can change the physical design to make it stable (towing a trailer example: https://youtu.be/6mW_gzdh6to?t=6), but this is not always possible.
- A Segway is an inverted pendulum (which is the unstable equilibrium point!)
- Solution: control software!
(next lecture)



Principles of Cyber-Physical Systems

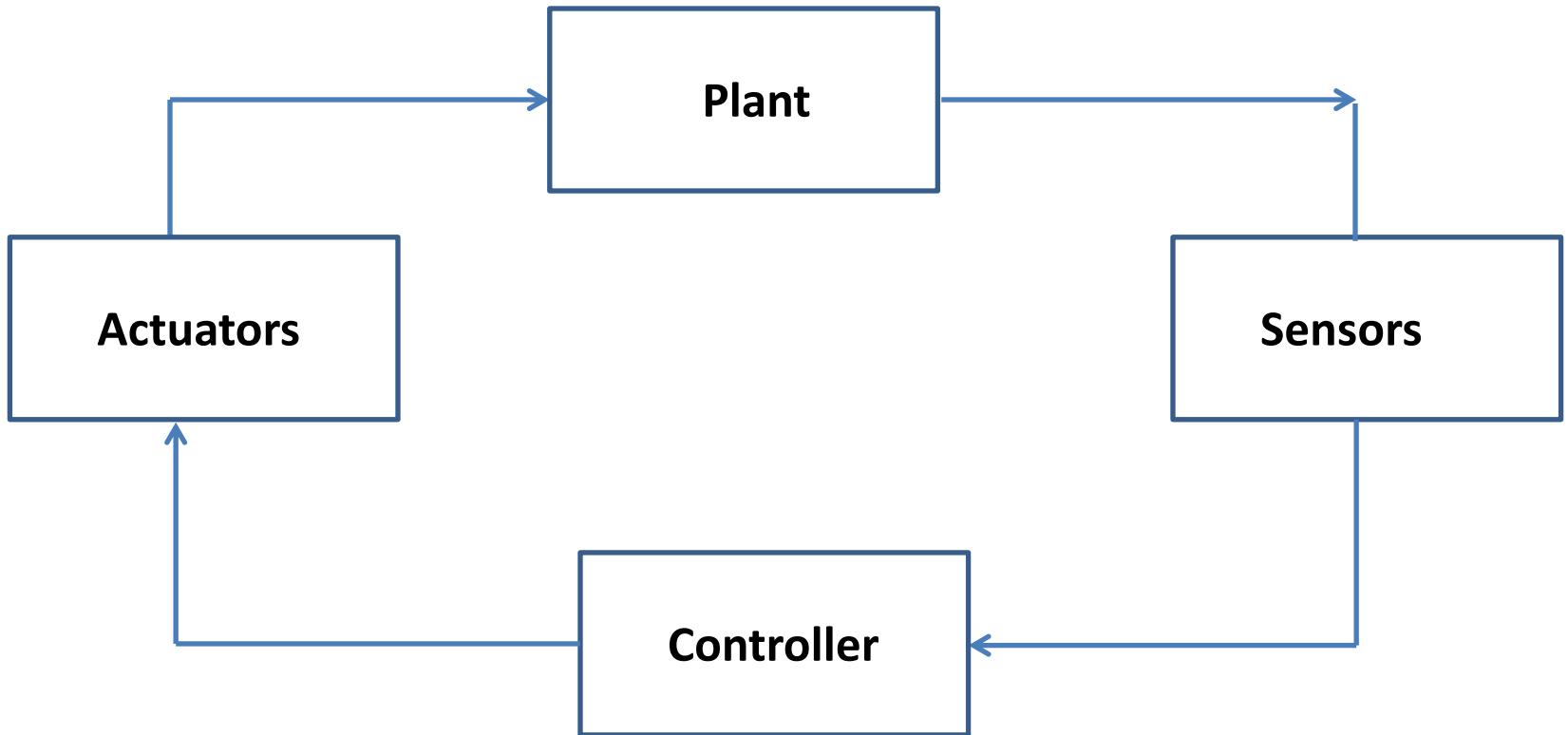
Dynamical Systems - Part 2

Instructor: Max Tschaikowski
tschaikowski@cs.aau.dk

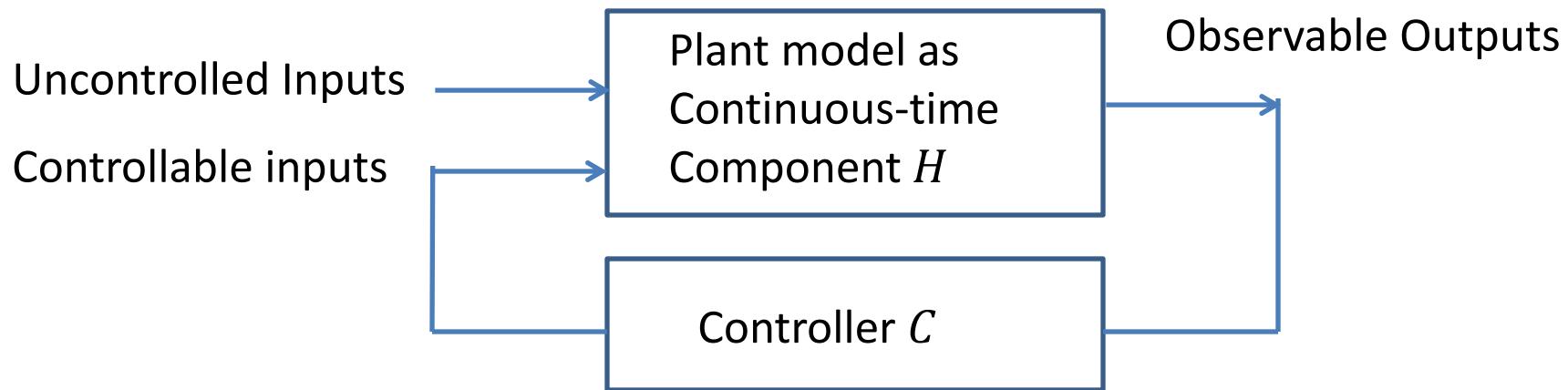
Slides Courtesy of Rajeev Alur



Traditional Feedback Control Loop

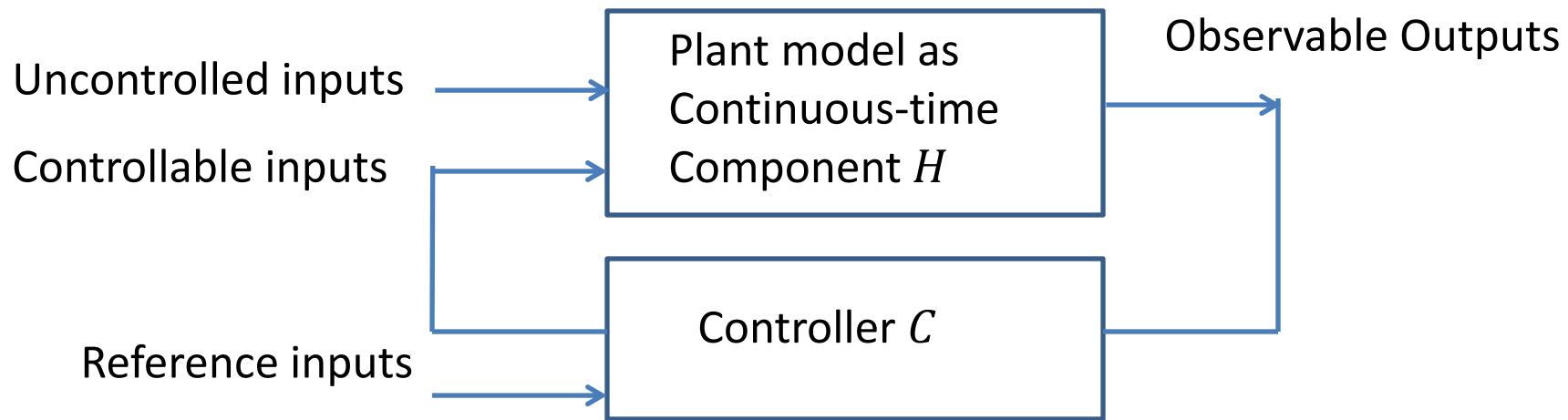


Control Design Problem



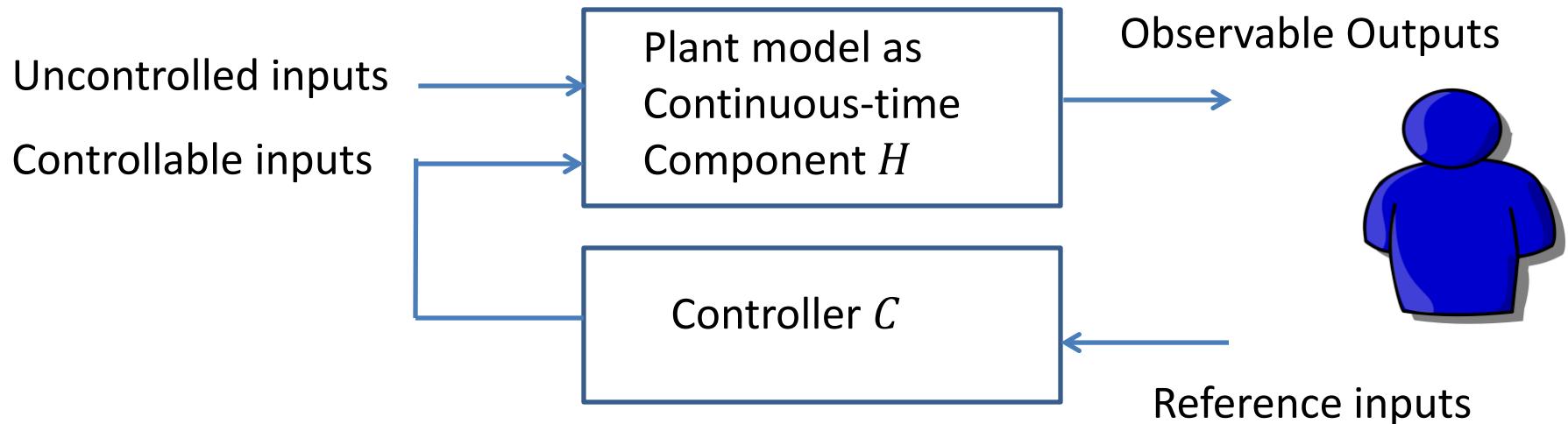
- Design a controller C so that the composed system $C||H$ is stable
- Is there a mathematical way to check when a system is stable?
- Is there a way to design C so that $C||H$ is stable ?
- Yes, if the plant model is **linear!** → Material for advanced Master's course.
- In this course, we will stick to analyzing controllers by simulation.

Control Design Problem



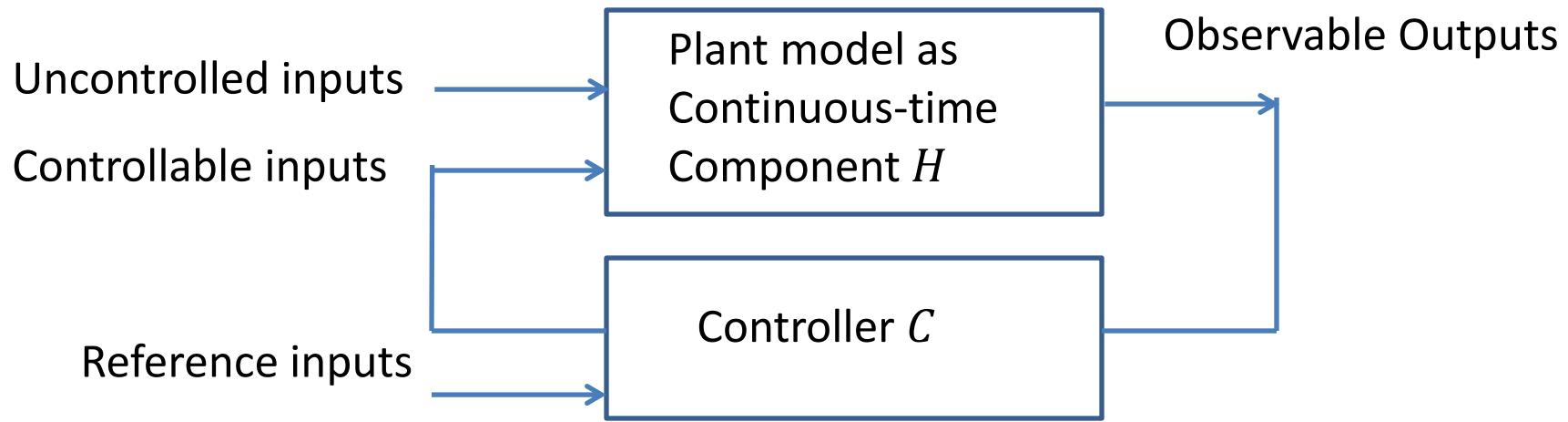
- Design a controller C so that the composed system $C||H$ is stable
- Reference inputs are high-level commands supplied by humans (e.g. desired speed of the car, temperature in the room)
 - Controller should satisfy additional safety/liveness requirements corresponding to reference inputs (e.g. speed of car eventually becomes close to desired cruising speed)

Open Loop Controller



- Plant outputs not fed to the controller
 - Benefit: Sensors not needed (less expensive)
- Controller simply maps reference inputs to controllable inputs
 - Knowledge of plant dynamics hard-coded in this algorithm
- Human intervention necessary to maintain acceptable performance

Feedback Controller



- Controller adjusts controllable inputs in response to observed outputs
 - Can respond better to variations in disturbances
 - Performance depends on how well outputs can be measured
- Two control design techniques
 - Mathematical based on theory of linear systems
 - PID controllers (widely used in practice)

Helicopter Model (Simplified)

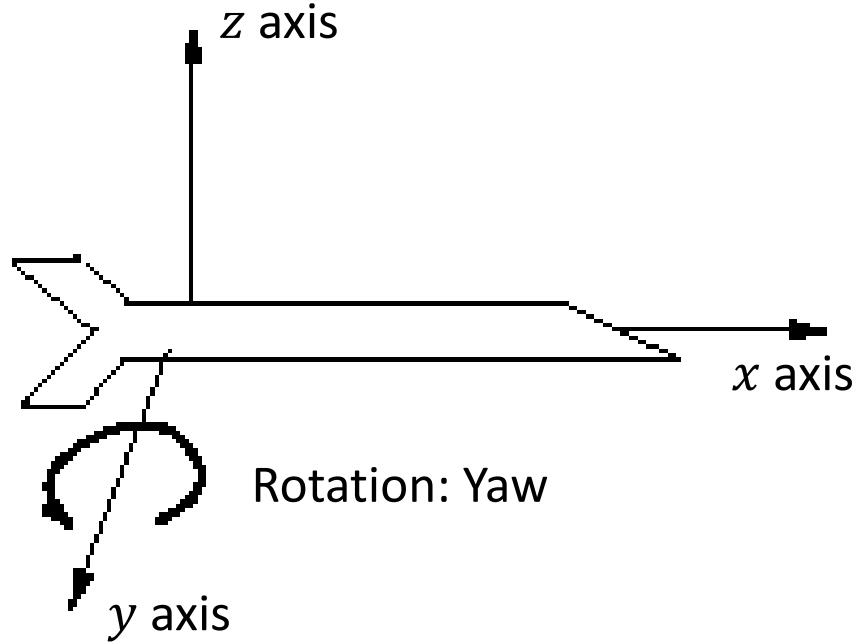
Design problem: What torque should the tail rotor apply to keep the helicopter from spinning?

$$\text{Yaw} = \theta$$

$$\text{Spin (rate of change of yaw)} = s$$

$$\text{Torque by rotor: } T$$

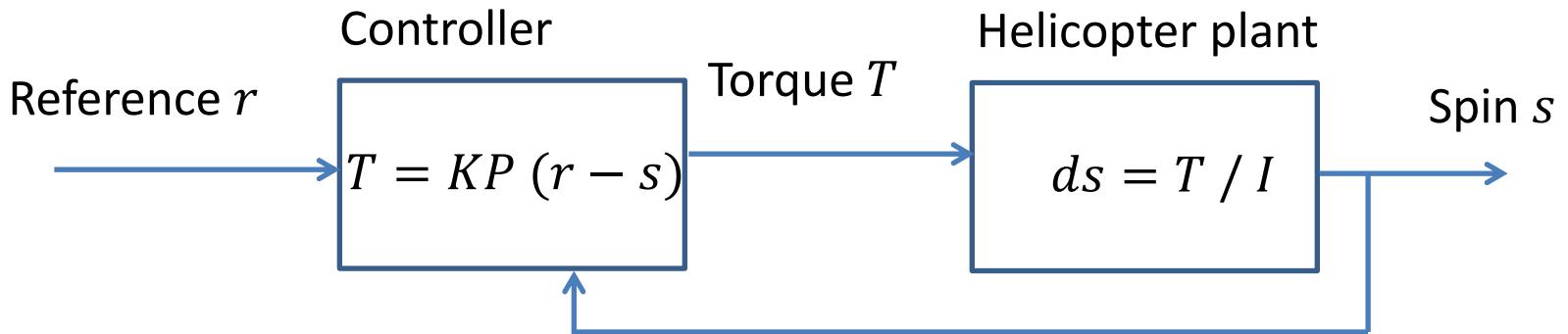
$$\text{Moment of inertia: } I$$



Equation of motion:

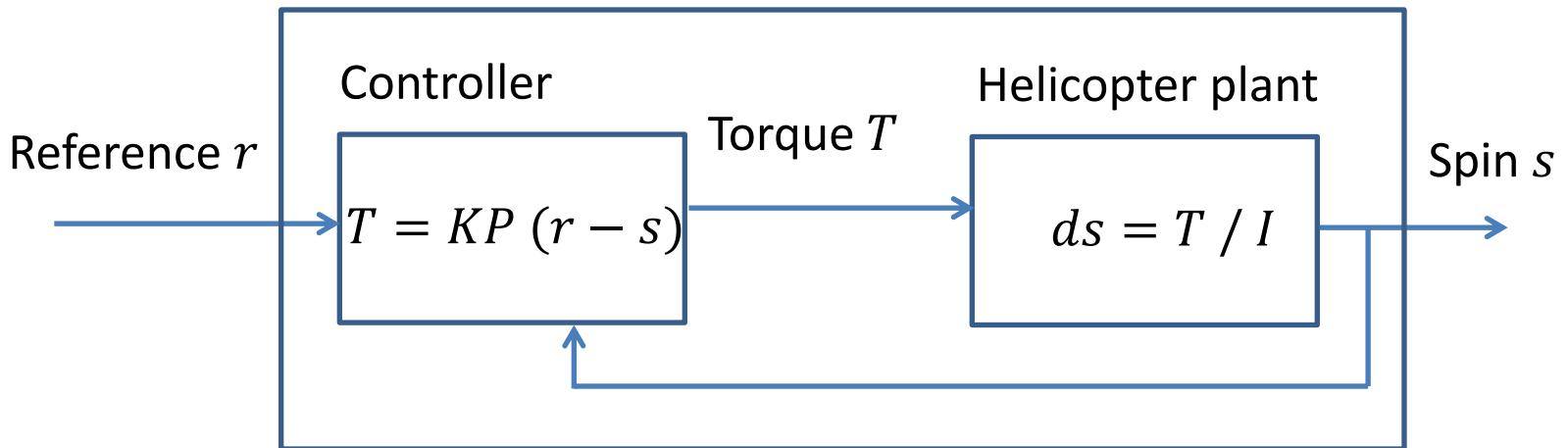
$$ds/dt = T / I$$

Feedback Controller for Helicopter Model



- Design controller so that composed system is stable
- Error $e = (r - s)$: difference in desired value and observed output
- Proportional controller: Its output is proportional to this error
- Constant K_P : Proportional Gain
- Note that the direction of torque changes with sign of the error

Stabilizing Controller for Helicopter Model

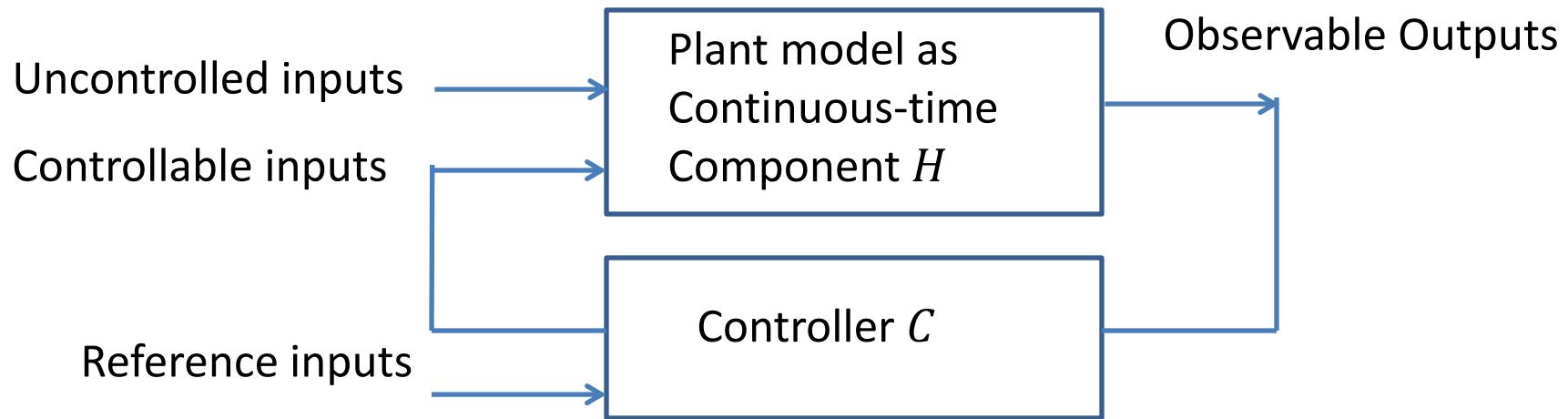


- Dynamics of the composed system:

$$ds/dt = K_P (r - s) / I$$

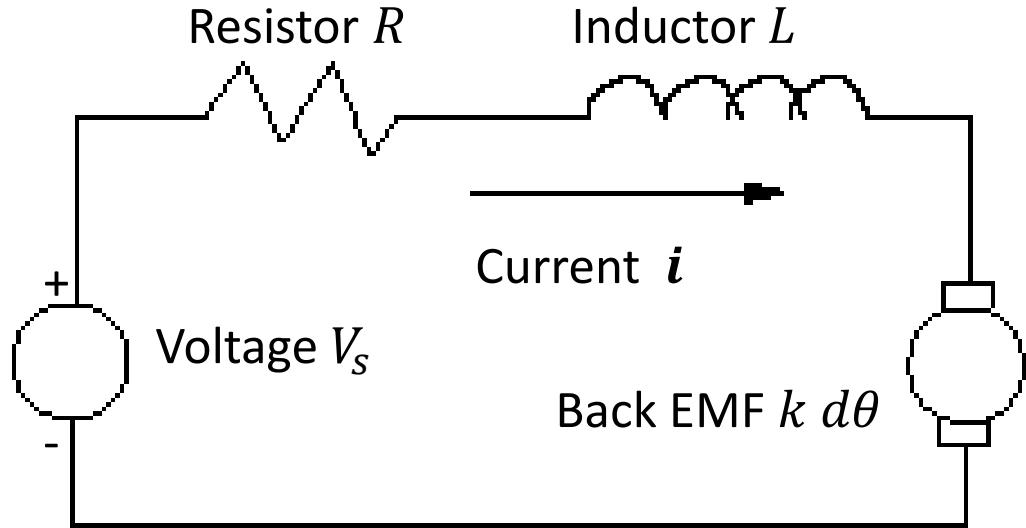
- When is this system asymptotically stable ?
- If the coefficient $-K_P/I$ is negative
- Control design: choose a positive gain constant K_P
 - Rate of convergence depends on its magnitude

PID Controllers



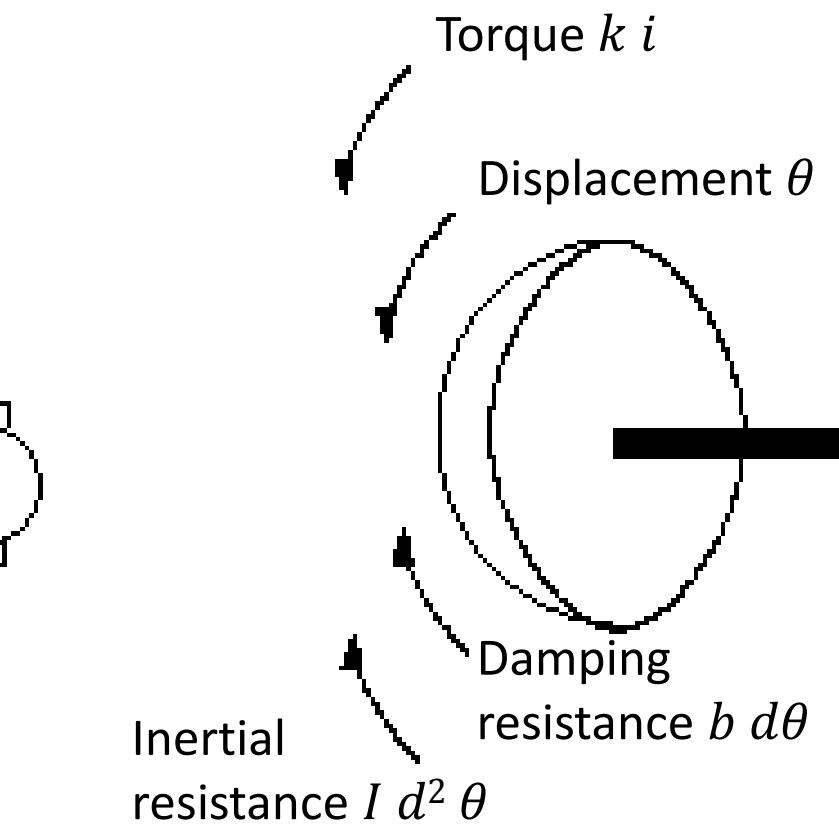
- Strategy for designing controllers that is widely used in practice
- Error = Reference Inputs - Observable Outputs
- Control output is sum of 3 terms:
 - Term proportional to error
 - Integral term to handle cumulative error
 - Derivative term in response to rate of change of error

DC Motor



Laws of electrical circuits:

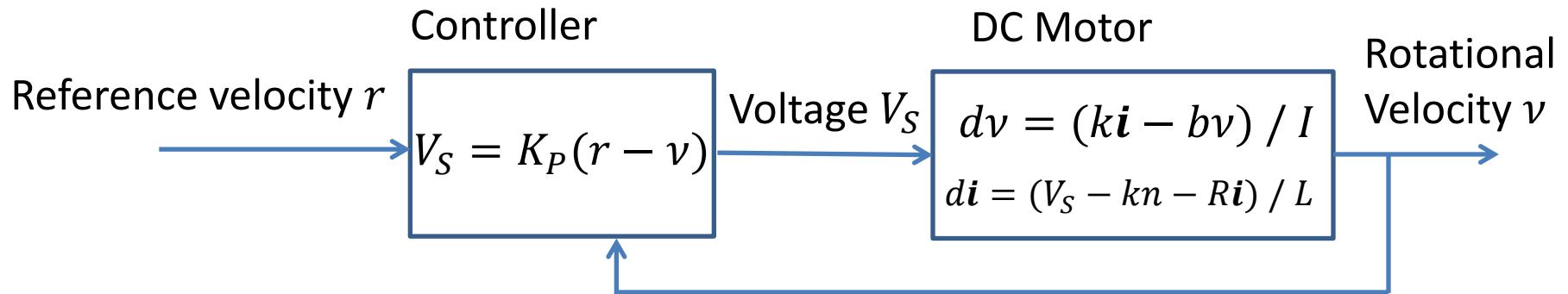
$$L \frac{di}{dt} + R i + k d\theta/dt = V_s$$



Laws of motion for the shaft:

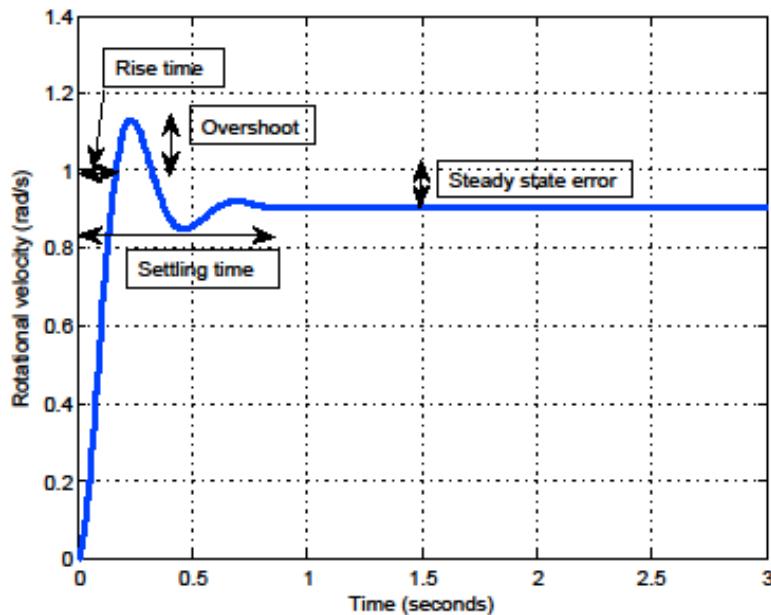
$$I d^2\theta/dt^2 + b d\theta/dt = k i$$

Proportional Controller for DC Motor



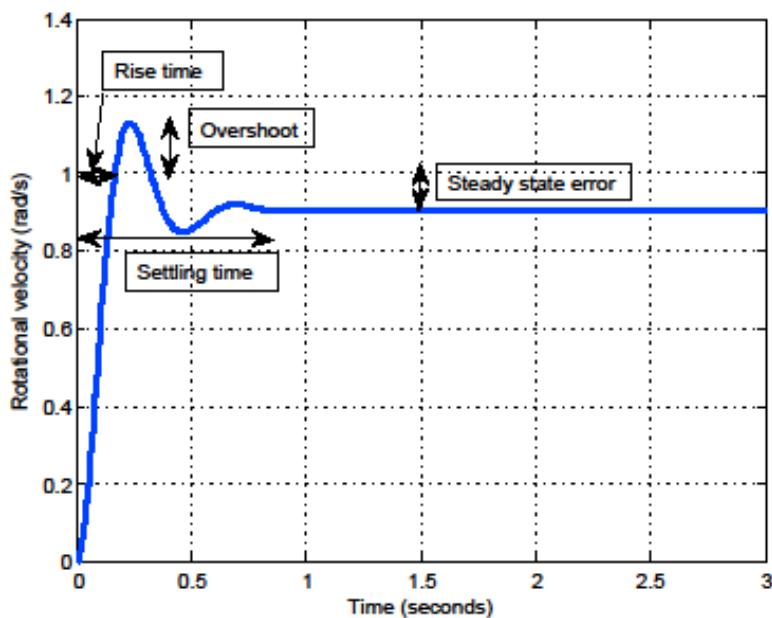
- DC Motor modeled as a linear system with 2 state variables, 1 input variable, and 1 output variable
- Feedback controller observes rotational velocity, and adjusts voltage to make it equal to desired rotational speed r
- First attempt: Proportional controller (P controller)

Step Response of P Controller



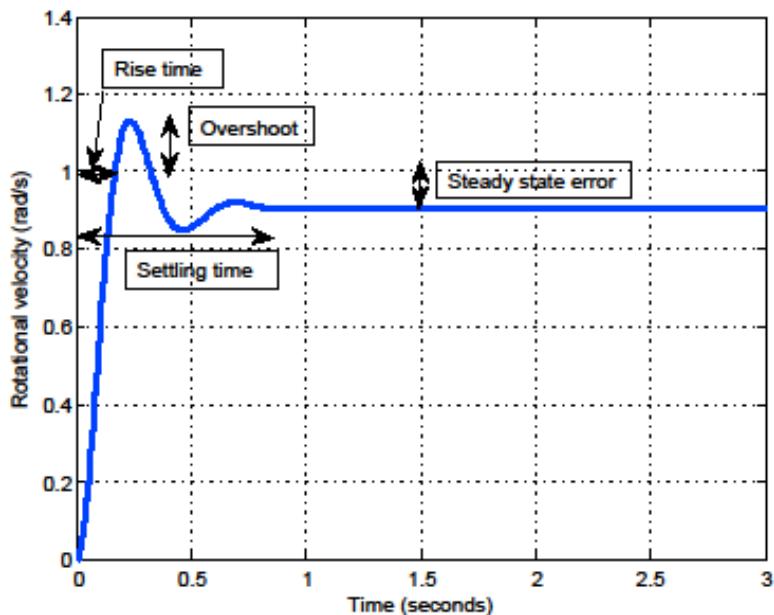
- Step response: How will system output change if at time 0, with $v = 0$, we change reference input r to 1?
- Plotted using MATLAB
- Beyond stability and convergence, what are desired characteristics of the response?

Characteristics of the Step Response



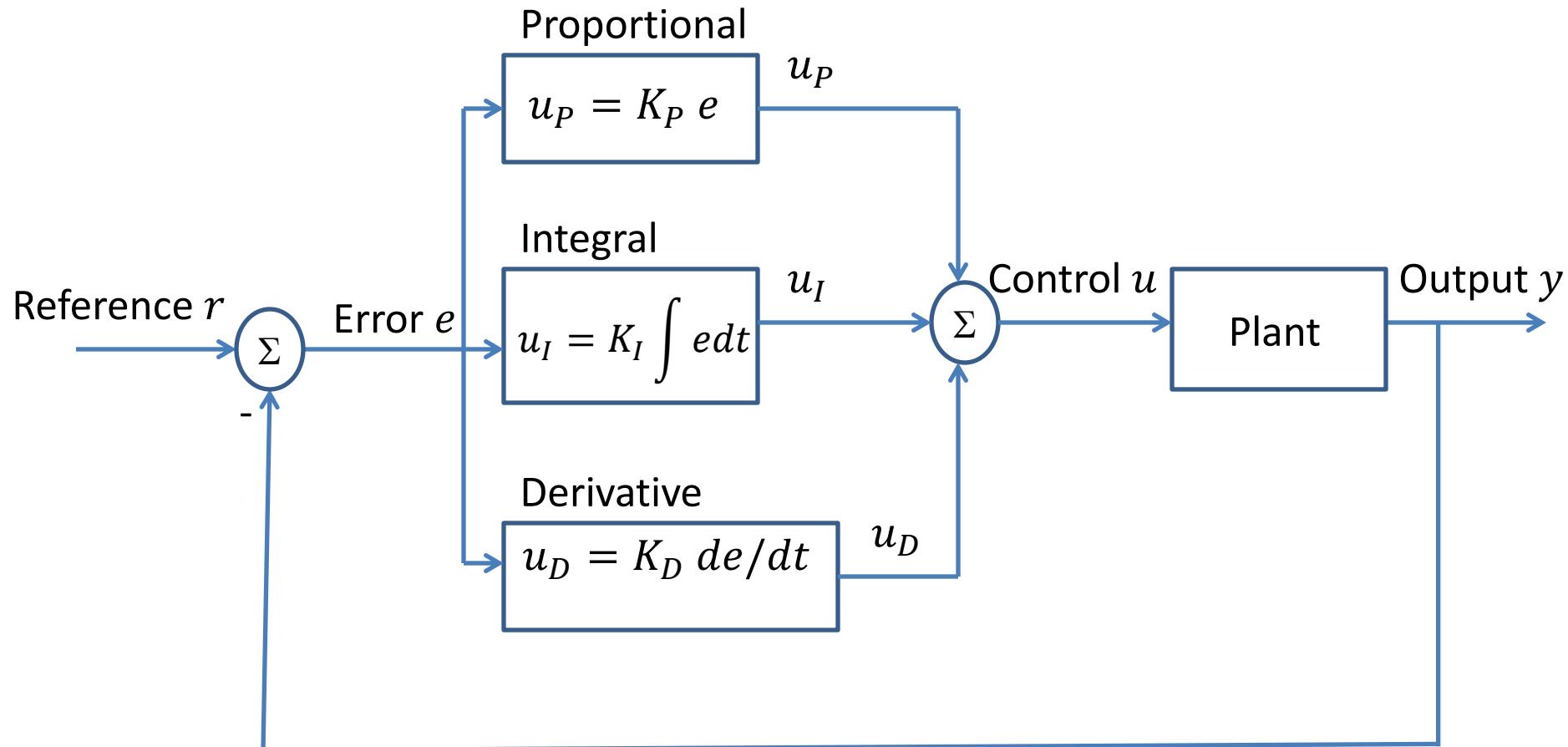
1. *Overshoot*: Difference between maximum output value and reference value (11% in this plot)
2. *Rise Time*: Time at which the output value crosses reference value (0.15sec in this plot)
3. *Settling Time*: Time at which output value reaches steady-state value (0.8sec in this plot)
4. *Steady State Error*: Difference between steady-state output value and reference (10% in this plot)

Improving the Step Response



- Performance of the P-controller depends on the value of the proportional gain constant K_P
- What happens if we increase it?
- Rise time decreases, but overshoot increases
- Steady-state error remains!
- Solution: Use Integral and Derivative Gains

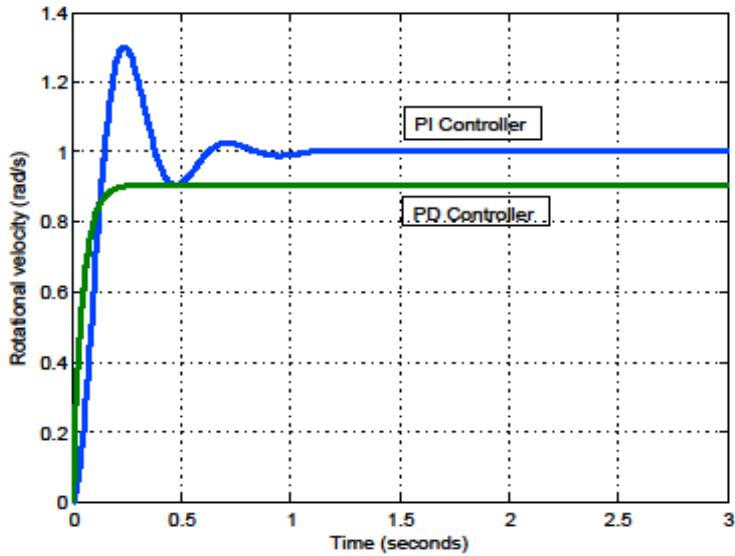
PID Controller



PID Controller

- If $e(t)$ is the error signal, then the output $u(t)$ of the PID controller is sum of 3 terms:
 - Proportional term: $K_P e(t)$, K_P is called proportional gain (response to current error)
 - Integral term: $K_I \int e dt$, K_I is integral gain (response to error accumulated so far)
 - Derivative term: $K_D d/dt e(t)$, K_D is derivative gain (response to current rate of change of error)
- Special cases of controllers: P, PD, PI

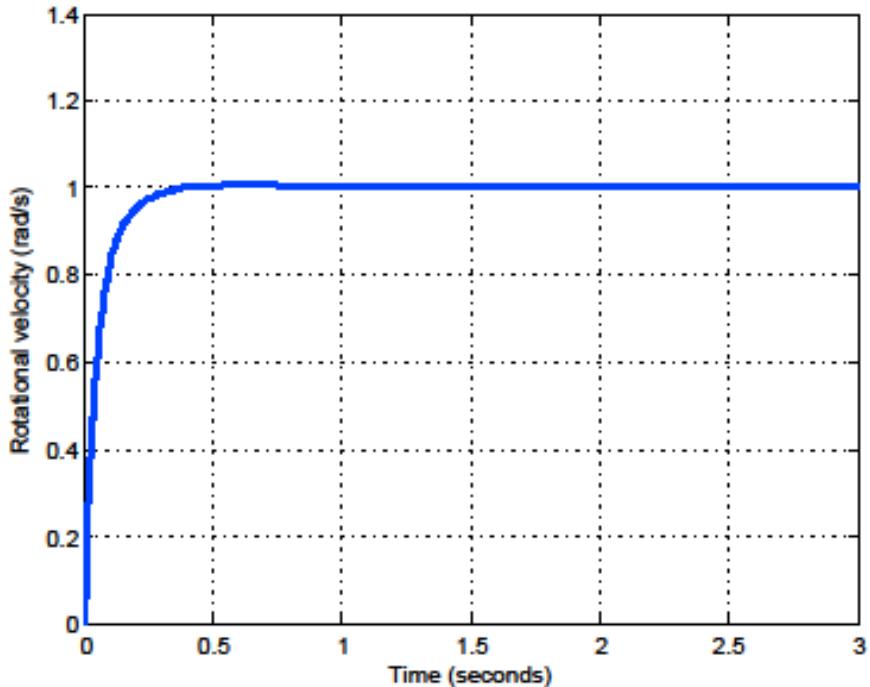
PI and PD Controllers for DC Motor



- PI Controller: adding integral term to proportional controller gets rid of steady state error
 - Overshoot, rise time, setting time increase

- PD controller: adding derivative term to proportional controller gets rid of overshoot
 - Steady state error remains

PID Controller for DC Motor

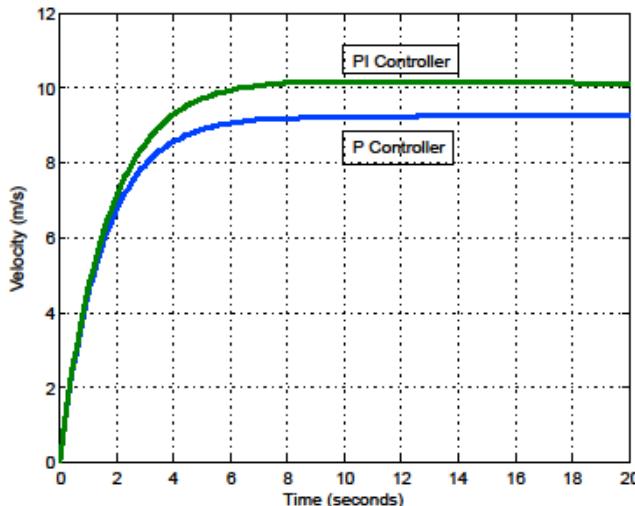


Excellent performance on all metrics: $K_P = 100$, $K_D = 10$, $K_I = 200$
Small rise time, settling time, negligible steady state error, no overshoot

Designing PID Controllers

- What are the effects of changing the gain constants K_P , K_D , K_I ?
- Broad co-relationships well understood
- Control toolboxes allow automatic tuning of parameters
- PID controllers seem to work well even when the actual system differs significantly from the plant model
 - Computation of control output depends only on the measured error, and not on the model!

PI Cruise Controller



- Desired change in velocity 10 m/s
- PI controller: $K_P = 600, K_I = 40$
- Settling time = 7s with negligible overshoot and steady-state error
- Works in a real car!

Principles of Cyber-Physical Systems

Hybrid Systems

Instructor: Max Tschaikowski
tschaikowski@cs.aau.dk

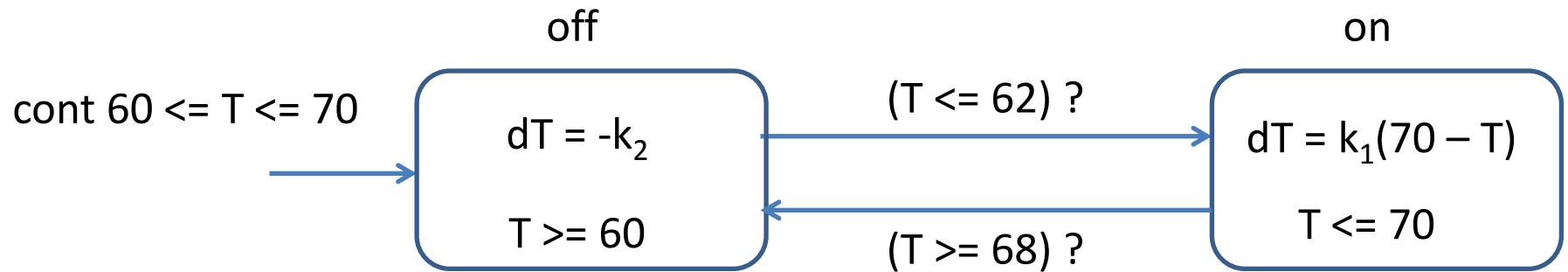
Slides Courtesy of Rajeev Alur



Models of Reactive Computation

- Continuous-time model for dynamical system
 - Synchronous, where time evolves continuously
 - Execution of system: Solution to algebraic / differential equations
- Timed model
 - Like asynchronous for communication of information
 - Clocks evolve continuously, and constraints on delays allow synchronous/global coordination
- Hybrid systems
 - Generalization of timed processes
 - During timed transitions, evolution of state/output variables specified using differential equations as in dynamical systems

Self-Regulating Switching Thermostat



State machine with two modes +

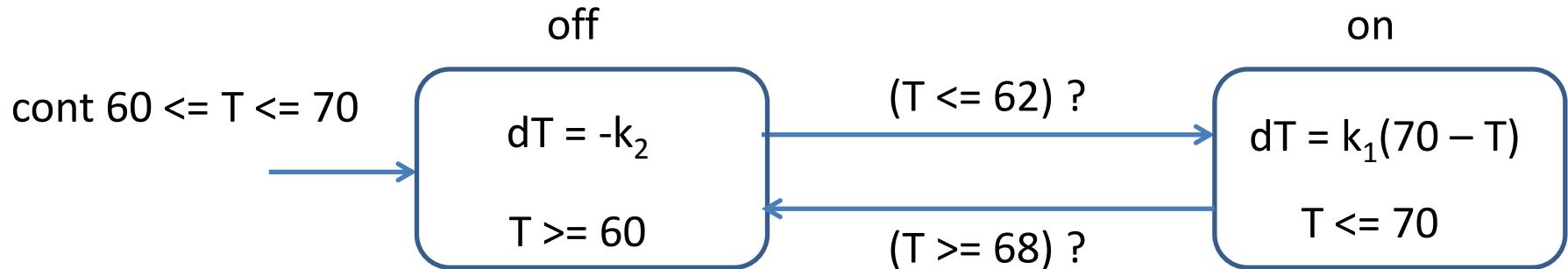
State variable T to model temperature: type **cont**

T can be tested and updated during mode-switches

T changes continuously during timed transitions given by differential equations

Invariants (as in timed model) constrain how long can a timed transition be

Executions of Thermostat



Initial state = (off, T_0) with T_0 in the interval [60,70]

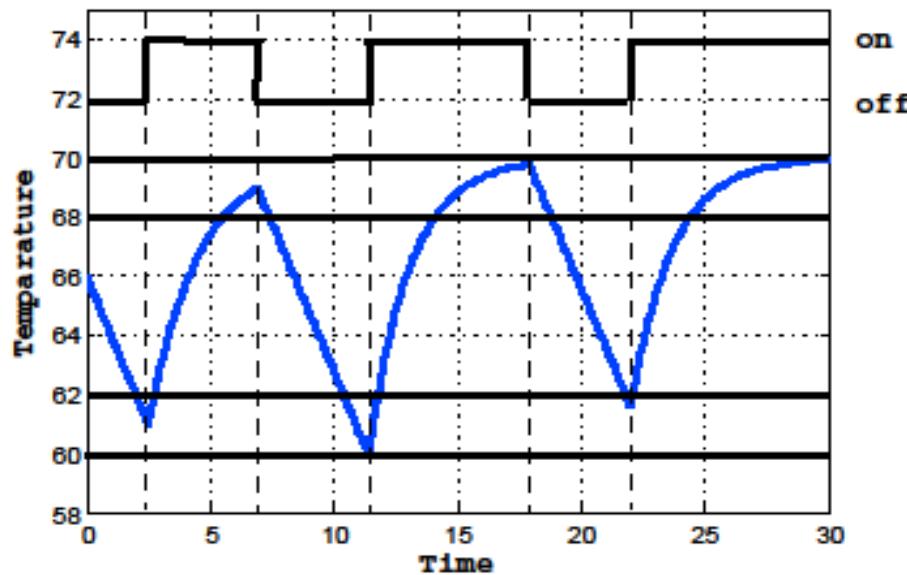
During a timed transition, T decreases continuously: $T(t) = T_0 - k_2 t$

Mode-switch to on enabled when $T \leq 62$, and must happen before T reaches 60

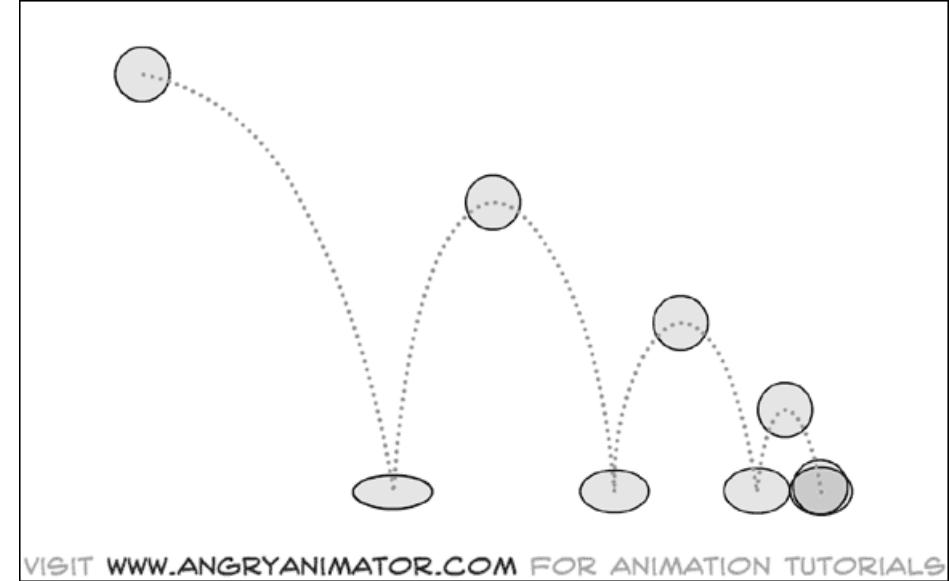
As time elapses in mode on, T increases according to $T(t) = 70 - (70 - T^*) e^{-k_1(t-t^*)}$,
 t^* , T^* : time and temperature upon entry to mode on

Mode-switch to off enabled when $T \geq 68$, and must happen before T reaches 70

Simulation Plot of an Execution

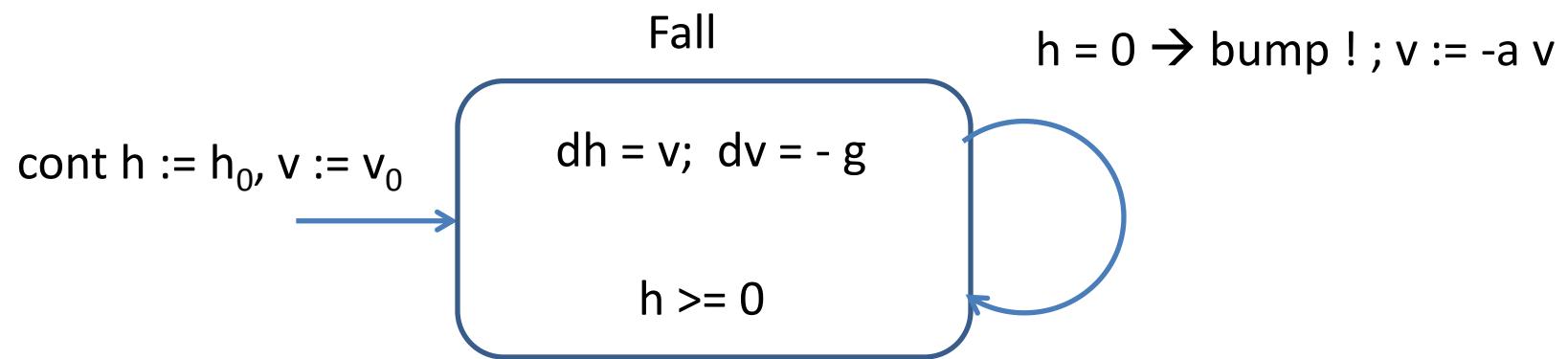


Modeling a Bouncing Ball



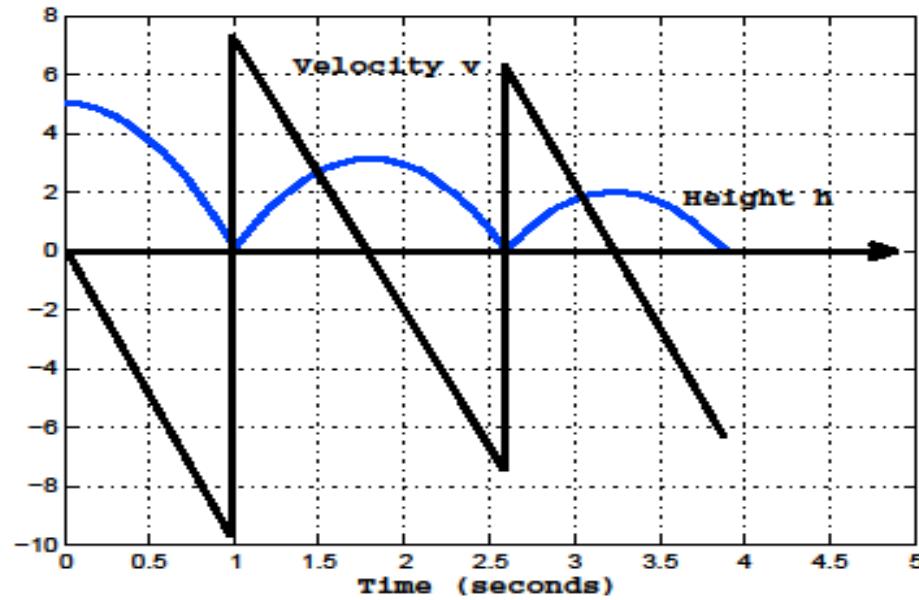
- Ball dropped from an initial height h_0 with an initial velocity v_0
- Velocity changes according to the differential equation $dv/dt = -g$
- When the ball hits the ground, that is, when height $h=0$, velocity changes discretely: $v := -a v$, where $0 < a < 1$ is dampening constant
- Modeled as a hybrid system: mix of discrete and continuous behaviors!

Hybrid Process for Bouncing Ball



Execution of the BouncingBall process

$$h_0 = 5; v_0 = 0$$



Definition of Hybrid Process: Syntax

- A hybrid process HP consists of
 1. An asynchronous process P, where some of the state variables can be type cont (ranging over real numbers)
 2. A continuous-time invariant CI which is a Boolean expression over the state variables of P
 3. For every output variable y of type cont, a Lipschitz-continuous real-valued expression that gives the value of y as a function of state variables and continuous input variables
 4. For every state variable x of type cont, a Lipschitz-continuous real-valued expression that gives the rate of change of x as a function of state variables and continuous input variables

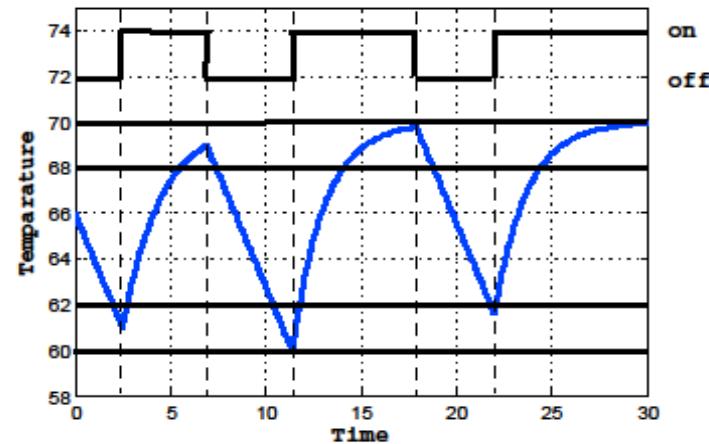
Definition of Hybrid Process: Semantics

- Define inputs, outputs, states, initial states, internal actions, input actions, output actions exactly the same as the asynchronous model
- Timed actions: Given a state s and real-valued time $\delta > 0$ and a continuous input signal $u(t)$ that gives values for continuous inputs over time interval $[0, \delta]$, the corresponding state/output signal over $[0, \delta]$ is uniquely defined so that
 1. Initial state $s(0)$ equals s
 2. Discrete (i.e. non-cont) state variables stay unchanged
 3. For each continuous output variable y , the value $y(t)$ satisfies the corresponding algebraic equation
 4. For each continuous state variable x , the derivative $dx(t)/dt$ of the signal satisfies the corresponding differential equation
 5. At all times t in $[0, \delta]$, the signal value $s(t)$ satisfies the invariant constraint CI

Executions of Hybrid Processes

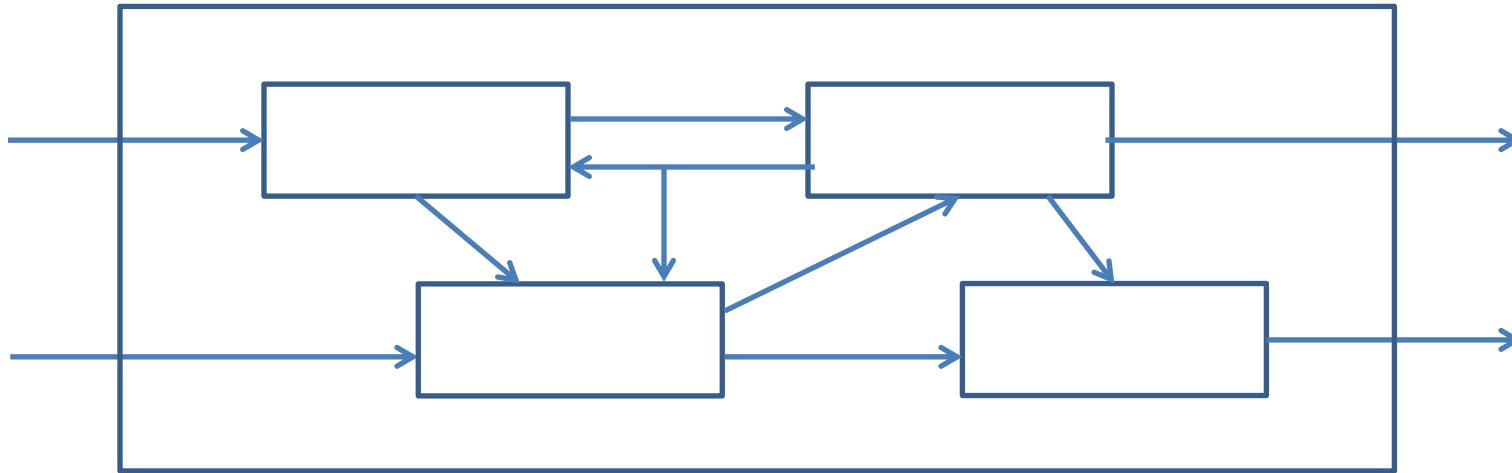
Starting from an initial state, execute either a discrete step (input, or output or internal action) or a timed step (need to solve system of differential equations)

(off, 66) -2.5 → (off, 61) → (on, 61)
-3.7 → (on, 69.02) → (off, 69.02)
-4.4 → (off, 60.22) → (on, 60.22)
-7.6 → (on, 69.9) → (off, 69.9)
-4.1 → (off, 61.7) → (on, 61.7) ...



Concepts based on transition systems such as reachable states, safety and liveness requirements, all apply to hybrid systems

Block Diagrams

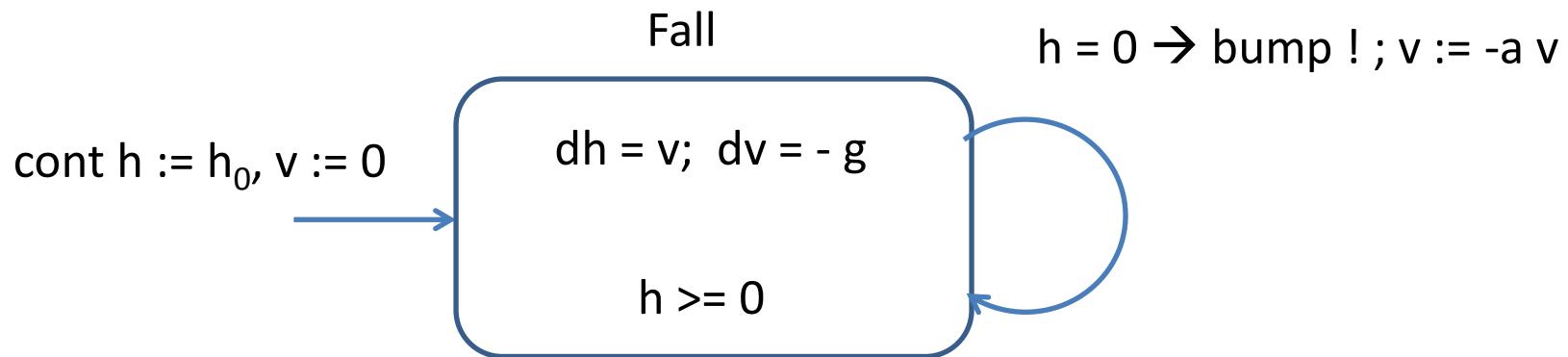


- Component processes can now be hybrid processes
 - Need to define Instantiation, Composition, Output Hiding
- Channels connecting processes of two types
 1. Sender/receiver communication of values during discrete steps as in the asynchronous model
 2. Continuously evolving signals during timed steps as in the model of continuous-time dynamical systems

Summary of the Model

- Generalizes timed model
 - Variables evolving continuously during a timed action can have complex dynamics, clocks being a very special case
- Generalizes continuous-time dynamical systems
 - Discontinuous changes to system state now can be modeled
- Generalizes asynchronous model
 - Distributed/multi-agent systems can be modeled
- Suitable for modeling of cyber-physical systems (in full generality)
- Existing commercial tool support: Modelica, Stateflow/Simulink
- Challenge for analysis
 - Even if dynamics in individual modes is linear, due to discrete changes, not possible to obtain closed-form solutions, or general theorems about stability

Analysis of Bouncing Ball Model



Change in height during first bounce: $h(t) = h_0 - gt^2/2$

Time at which first bump occurs: $t_1 = \sqrt{2h_0/g}$

Velocity just before first bump occurs: $- \sqrt{2g h_0} = -v_1$

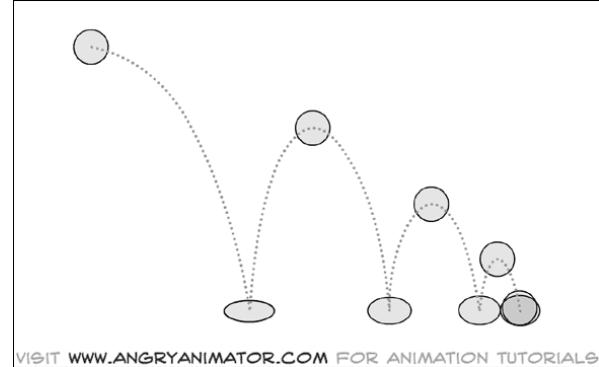
Velocity just after first bump: $v_2 = a v_1$

Evolution of height during second bounce: $h(t) = v_2 t - gt^2/2$

Time between first and second bump: $t_2 = 2v_2/g$

Velocity just before second bump occurs: $-v_2$ and after second bump $v_3 = a v_2$

Modeling a Bouncing Ball



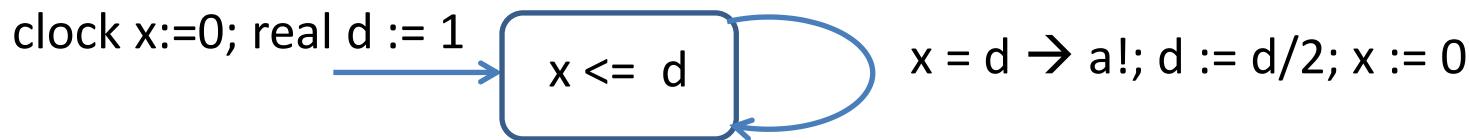
VISIT WWW.ANGRYANIMATOR.COM FOR ANIMATION TUTORIALS

- Velocity after k bumps = $a^k v_1$
- Duration between k -th and following bump $a^k v_1/g$
- Sum of all durations converges to some finite K
- Infinitely many discrete actions in finite time = Zeno behavior!
- An execution with infinitely many discrete actions describes behavior only up to time a certain time K and does not tell us the state of the system beyond K

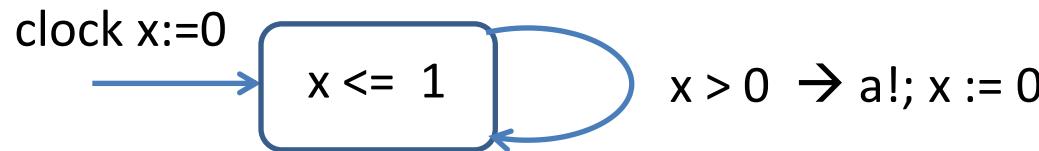
Formalization

- An infinite execution of a hybrid process HP is of the form $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow s_2 - t_3 \rightarrow s_3 \dots$, where t_i is duration of i-th step
 - Input/output/internal actions are instantaneous (duration 0)
- An infinite execution is called Zeno if the infinite sum of all the durations is bounded by a constant, and non-Zeno if the sum diverges
- A state s of the process HP is called
 - Zeno if every execution starting in state s is Zeno
 - Non-Zeno if there exists some non-Zeno execution starting in s
- A hybrid process HP is called non-Zeno if every reachable state of HP is non-Zeno
 - At every point during an execution it is possible for time to diverge
- Zeno system: Could end up in a state from which duration between successive steps must get smaller and smaller
- Thermostat: non-Zeno; Bouncing Ball: Zeno

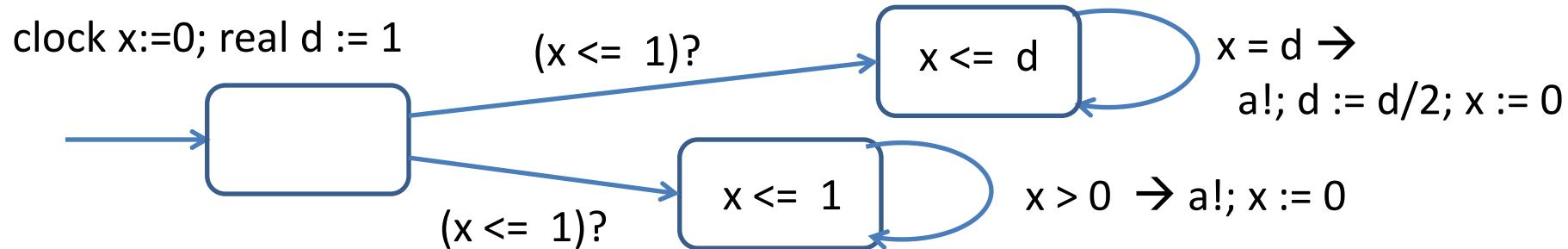
Zeno Vs Non-Zeno



Zeno ! Every possible execution is Zeno



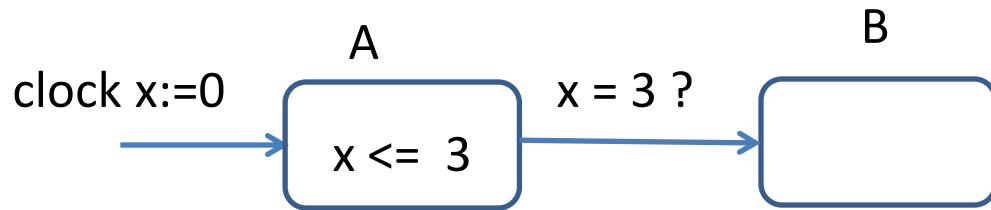
Non-Zeno ! Some executions are Zeno and some are non-Zeno



Zeno ! System may end up in a state from which only Zeno executions are possible

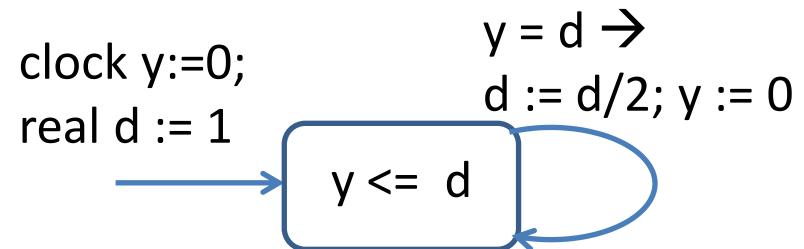
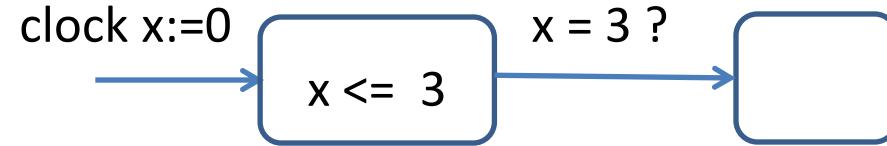
Zeno Processes and Reachability

- How does existence of Zeno processes influence analysis?
- Recall: A state s is said to be reachable if there exists a finite execution starting in an initial state and ending in state s
- Safety: A property φ is an invariant if all reachable states satisfy φ

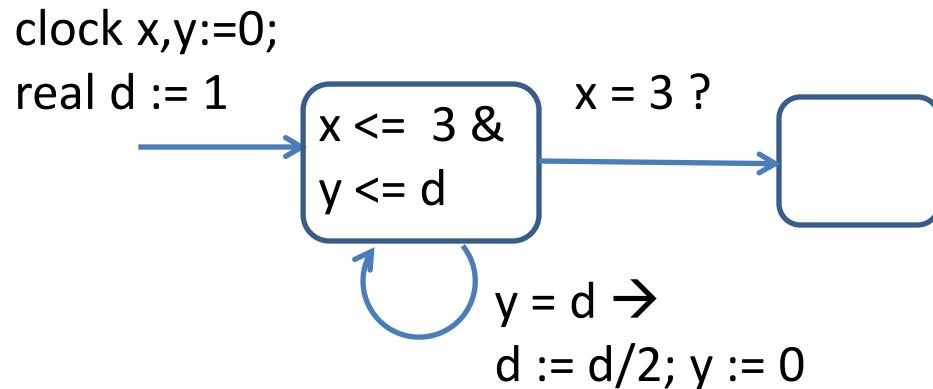


Is mode B reachable ?

Zeno Processes and Reachability

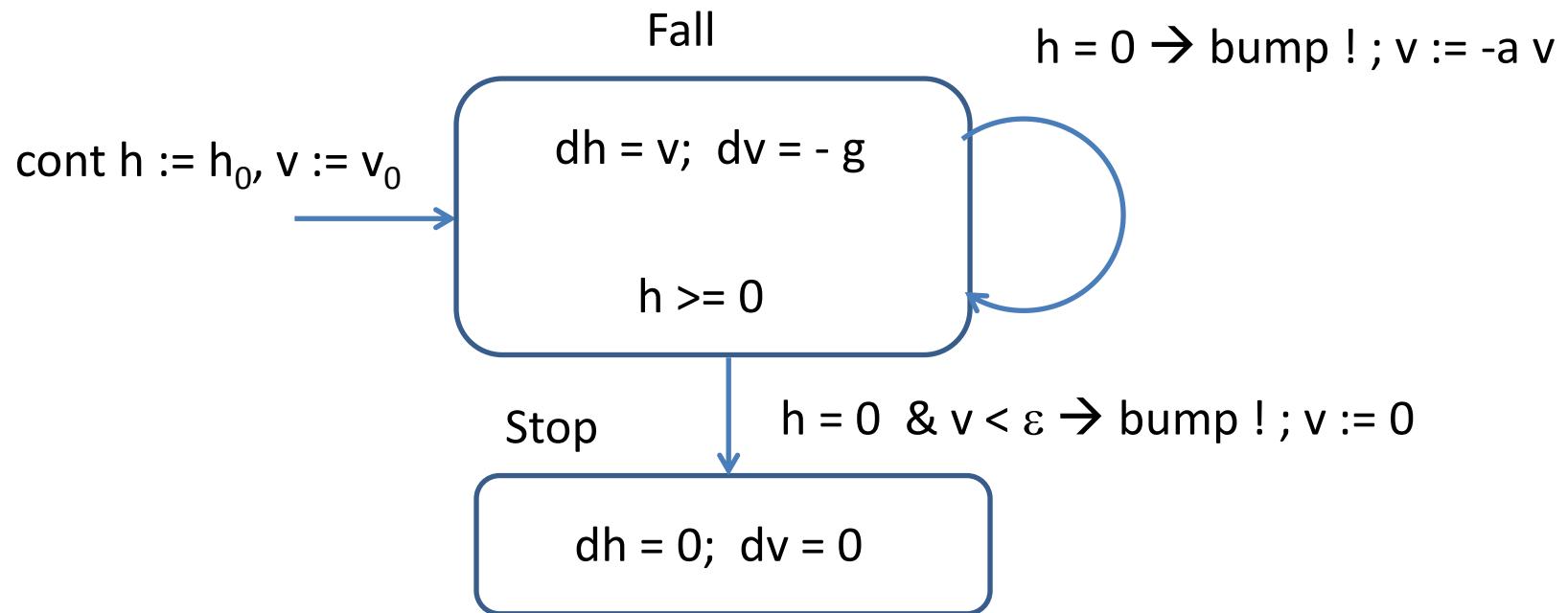


Is mode B reachable ?



Presence of a Zeno process in the system can stop time from advancing, and make states of other processes unreachable !

Making Bouncing Ball Non-Zeno



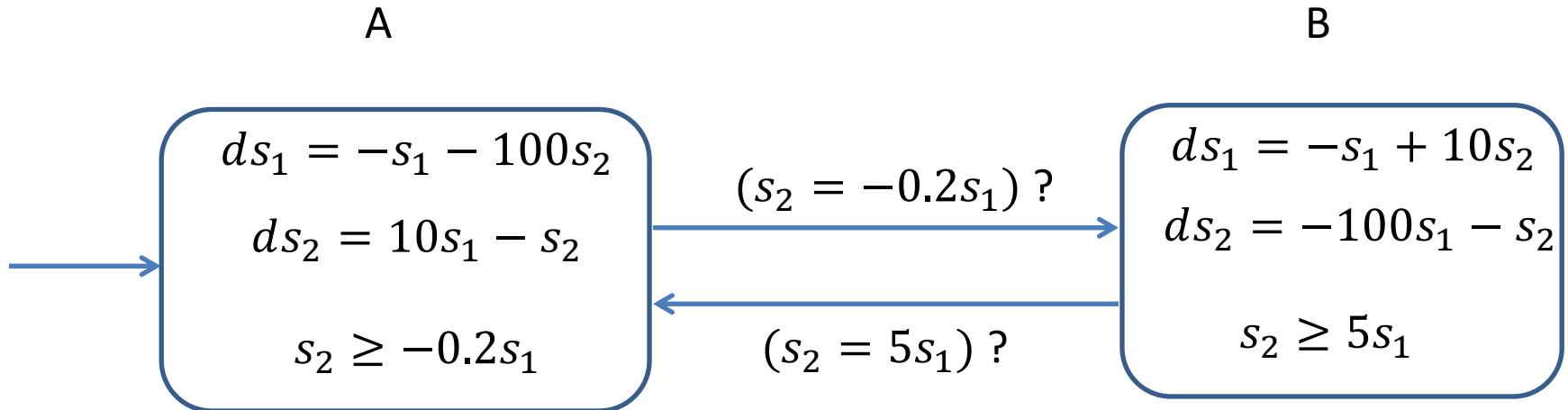
If velocity is too small, stop modeling dynamics accurately

In this model, there is a lower bound on duration between successive bumps

Stability analysis



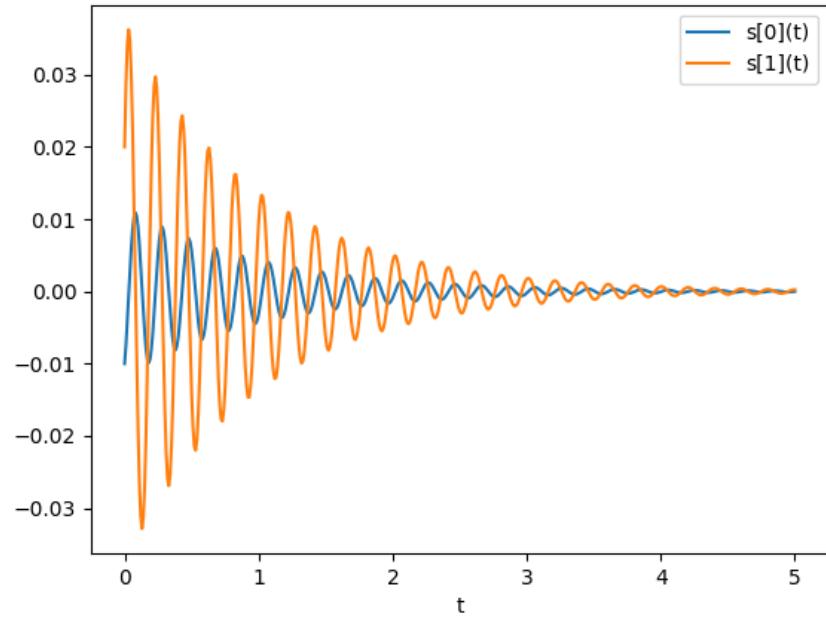
Stability of Hybrid Systems



Is the dynamics in mode A stable?

Is the dynamics in mode B stable?

Both modes have stable dynamics, b



Stability of Hybrid Systems

