

CIS 673: Lecture 4: Sept 18

- Events are like messages
- An event is modeled by toggling of a boolean variable:
 - $x!$ stands for $x' := \neg x$; issuing an event
 - $x?$ stands to $x' \neq x$; checking for presence
- $x : \mathbf{event}$ declares x to be of type event: value of x is unimportant, only changes are.
- In general, a channel is modeled by an event x and contents c :
 - Sending a message v : $x!; c' := v$
 - Receiving a message into y : $x? \rightarrow y' := c'$

Passive Modules

- Intuition: If environment does not change, then the module also may do nothing
- Different from asynchrony:
 - Interface variables can be synchronized with external variables
 - But no way to count rounds (i.e. no explicit notion of time)
- Suitable for discrete events systems
- Modeled using nondeterminism

Closed versus Open Modules

- A module is closed if it has no external variables
- Behavior of a closed module is completely determined, and not influenced by the environment
- A module may be closed by taking parallel composition with another module that models the environment

Determinism

- Deterministic atom: Precisely one guard true at any time (initial and update commands define one-to-one mappings)
- Deterministic module: only deterministic atoms
- Private determinism: all private vars are controlled by deterministic atoms (state determined by observation)
- Nondeterministic: even private vars may change nondeterministically (hard to analyze)

Alternative Models

- Literature has many models for concurrency.
Important issues:
 - What constitutes a state?
 - What constitutes a round?
 - How do components interact in a round?
 - When are two systems equivalent?
- Examples:
 - Automata-style: Moore/Mealy machines, I/O automata
 - Process algebras: CSP, CCS, ACSR, pi-calculus...
 - Petri nets
 - Synchronous languages: ESTEREL, Lustre, Signal, Statecharts
 - Interleaving shared memory: Unity, Promela,

Significant Semantic Differences

- Blocking interaction (eg. CSP):
 - send ($x!$) is enabled only when there is a matching receiver, if not wait
 - Strong synchronization primitive
 - Influences theory of process equivalence
- Dynamic resolution of dependencies in Esterel:
 - there may be loops in awaits dependencies
 - but in every state, there is a consistent way to order atoms
- Interleaving composition:
 - Composing atoms corresponds to merging guarded commands
 - Less transitions
 - Write-shared variables allowed
 - Modular reasoning is difficult

Significant Syntactic Restrictions

- Instead of guarded commands, richer languages with sequencing, looping, and conditionals
- Many languages allow mixing sequencing and parallel composition eg. $a; b; (c \parallel d)$
- Synchronous languages allow interrupts
- Promela allows dynamic creation of processes
- Mobile calculi: instead of simple values, communication allows exchange of functions or processes

Interleaving mutual exclusion

module *Pete* **is**

interface $pc_1, pc_2: \{outC, reqC, inC\}$

private $x: \{1, 2\}$

init

$\parallel true \rightarrow pc'_1 := outC; pc'_2 := outC$

update

$\parallel pc_1 = outC \rightarrow pc'_1 := reqC; x' := 1$

$\parallel pc_1 = reqC \wedge pc_2 = outC \rightarrow pc'_1 := inC$

$\parallel pc_1 = reqC \wedge x = 2 \rightarrow pc'_1 := inC$

$\parallel pc_1 = inC \rightarrow pc'_1 := outC$

$\parallel pc_2 = outC \rightarrow pc'_2 := reqC; x' := 2$

$\parallel pc_2 = reqC \wedge pc_1 = outC \rightarrow pc'_2 := inC$

$\parallel pc_2 = reqC \wedge x = 1 \rightarrow pc'_2 := inC$

$\parallel pc_2 = inC \rightarrow pc'_2 := outC$

Transition Graphs

- Operational semantics of reactive modules
- A transition graph G consists of
 - State space: set Σ of vertices
 - initial region: a subset $\sigma^I \subseteq \Sigma$ of the vertices
 - transition relation: a binary relation $\rightarrow \subseteq \Sigma^2$ on the vertices
- With every module P , we will associate a transition graph G_P

State space of a module

- A state of a module P is a valuation for the set X_P of module variables.
- Σ_P : the set of states of P
- Σ_P is always nonempty
- $Pete$ has 36 states
- Sample state s of $Pete$:
($pc_1 = outCS, x_1 = true, pc_2 = outCS, x_2 = true$)

Initial Region of a Module

- Intuition: all states that are obtained by executing initial commands of all atoms
- Formally, a state s of P is initial if for every atom U of P ,

$$(awaitX'_U[s'], ctrX'_U[s']) \in \llbracket Init_U \rrbracket$$

- σ_P^I : the set of initial states of P
- Thm: For every valuation s_e of the external vars, there is some initial state s with $extlX[s] = s_e$
- *Pete* has 4 initial states s_1, s_2, s_3, s_4 :

$$\begin{aligned} pc_1[s_1] &= outC, x_1[s_1] = T, pc_2[s_1] = outC, x_2[s_1] = T; \\ pc_1[s_2] &= outC, x_1[s_2] = T, pc_2[s_2] = outC, x_2[s_2] = F; \\ pc_1[s_3] &= outC, x_1[s_3] = F, pc_2[s_3] = outC, x_2[s_3] = T; \\ pc_1[s_4] &= outC, x_1[s_4] = F, pc_2[s_4] = outC, x_2[s_4] = F. \end{aligned}$$

Transitions of a Module

- The state pair (s, t) is a transition if t can be obtained from s by executing the update commands of all the atoms.

- Formally, (s, t) is a transition of P if for every atom U of P ,

$$(\text{read}X_U[s] \cup \text{await}X'_U[t'], \text{ctr}X'_U[t']) \in \llbracket \text{update}_U \rrbracket$$

- \rightarrow_P : the set of transitions of P

- The state s_1 of *Pete* has 4 successors: s_5, s_6, s_7 :

$$pc_1[s_5] = reqC, x_1[s_5] = T, pc_2[s_5] = outC, x_2[s_5] = T;$$

$$pc_1[s_6] = outC, x_1[s_6] = T, pc_2[s_6] = reqC, x_2[s_6] = F;$$

$$pc_1[s_7] = reqC, x_1[s_7] = T, pc_2[s_7] = reqC, x_2[s_7] = F$$

- Thm: Every state has some successor (in fact, for every possible way of updating the external vars).

From Modules to Transition Graphs

- The module P defines the transition graph $G_P = (\Sigma_P, \sigma_P^I, \rightarrow_P)$.
- The transition graph of a module is serial:
 - At least one initial state
 - Every state has at least one successor
- The transition graph of a closed module is finitely branching:
 - Finitely many initial states
 - Every state has only finitely many successors
 - Means nondeterminism within a module is bounded
- If all vars have finite types then transition graph is finite
- Important: Transitions correspond to rounds and not subrounds

Trajectories

- A trajectory of G is a nonempty word $\bar{s}_{1..m}$ over the alphabet Σ of states such that for all $1 \leq i < m$, $s_i \rightarrow s_{i+1}$.
- Initialized trajectory: First state (source) is an initial state
- In serial graphs, for every natural number i , there is an initialized trajectory of length i .
- Module execution: Simulator or interpreter that produces initialized trajectories
- Interpreter has many choices and can produce many outputs
- Random simulation or user-guided step-by-step simulation

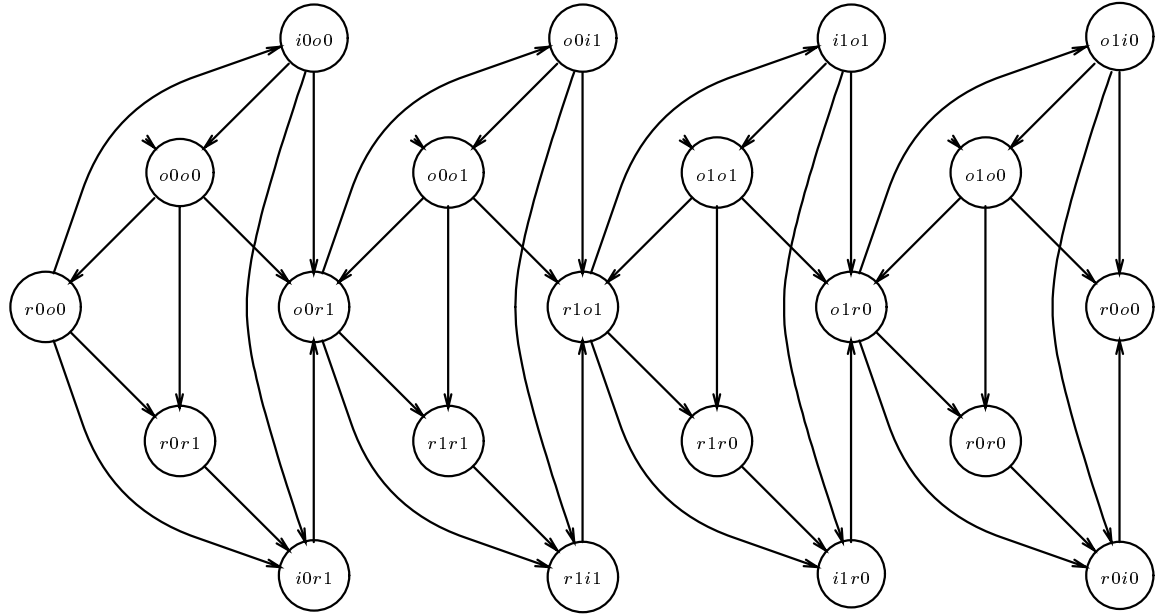
Sample initialized trajectory of Pete

	pc_1	x_1	pc_2	x_2
s_1	<i>outC</i>	<i>T</i>	<i>outC</i>	<i>T</i>
s_2	<i>outC</i>	<i>T</i>	<i>reqC</i>	<i>F</i>
s_3	<i>reqC</i>	<i>F</i>	<i>inC</i>	<i>F</i>
s_4	<i>reqC</i>	<i>F</i>	<i>inC</i>	<i>F</i>
s_5	<i>reqC</i>	<i>F</i>	<i>outC</i>	<i>F</i>
s_6	<i>reqC</i>	<i>F</i>	<i>reqC</i>	<i>T</i>
s_7	<i>inC</i>	<i>F</i>	<i>reqC</i>	<i>T</i>
s_8	<i>outC</i>	<i>F</i>	<i>reqC</i>	<i>T</i>
s_9	<i>outC</i>	<i>F</i>	<i>inC</i>	<i>T</i>
s_{10}	<i>reqC</i>	<i>T</i>	<i>outC</i>	<i>T</i>
s_{11}	<i>inC</i>	<i>T</i>	<i>outC</i>	<i>T</i>
s_{12}	<i>outC</i>	<i>T</i>	<i>outC</i>	<i>T</i>

Reachability Problem

- A state t of G is a reachable state if there is an initialized trajectory that ends in t
- Reachable Subgraph G^R :
 - States: reachable region σ^R of G
 - Initial region unchanged
 - Transitions: Reachable transitions \rightarrow^R (the restriction of \rightarrow to σ^R)
- Reachability problem:
 - Input: A transition graph G and a target region σ^T
 - Output: Yes if some state in σ^T is reachable, and No otherwise
 - Witness: Initialized trajectory leading to σ^T

Reachable subgraph of Pete



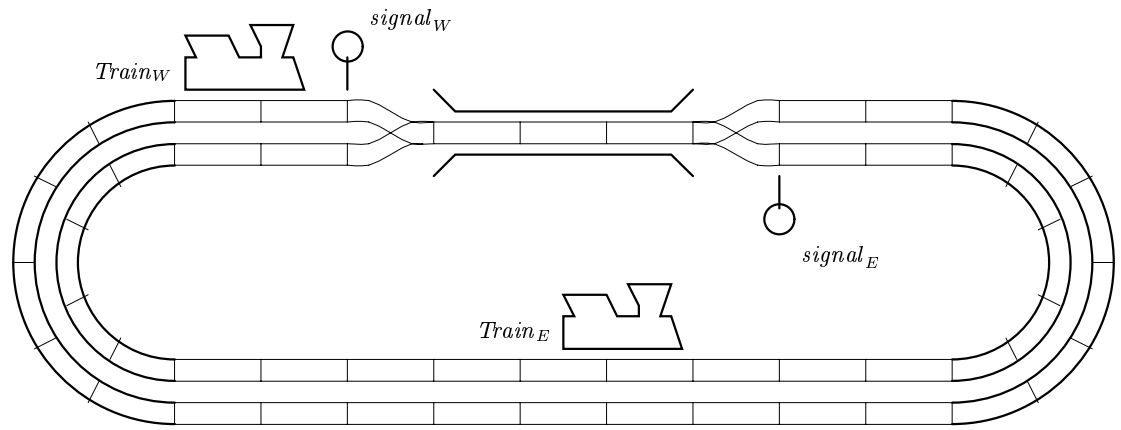
Invariants

- Simplest form of requirements: system should always stay within the good or safe region
- How to specify safe region?
observation predicates
- State predicate:
 - Boolean expression over module vars
 - Each state predicate p defines the region $\llbracket p \rrbracket$
 - p is an observation predicate if it refers only to interface and external vars
- The observation predicate p is an invariant of the module P if all reachable states satisfy p
- Mutual exclusion requirement of Pete: Is
$$\neg(pc_1 = inC \wedge pc_2 = inC)$$
an invariant of Pete?

Invariant Verification

- Input: Module P and observation predicate p
- Output:
 - Yes if p is an invariant of P
 - No otherwise.
 - Error-trajectory: initialized trajectory leading to a state violating p (what went wrong?)
- Invariant verification problem (P, p) reduces to graph reachability problem $(G_P, \llbracket \neg p \rrbracket)$
- Invariant-verification is undecidable in general (halting problem of Turing machines), but decidable for finite modules
- Is $out = in_1 \wedge in_2$ an invariant of SyncAnd? of AsyncAnd?

Railroad Controller



Modeling each train

module *Train* **is**

interface *pc*: {*away*, *wait*, *bridge*}; *arrive*, *leave*: **event**

external *signal*: {*green*, *red*}

lazy atom controls *arrive* **reads** *pc*

update

$\parallel pc = away \rightarrow arrive!$

lazy atom controls *leave* **reads** *pc*

update

$\parallel pc = bridge \rightarrow leave!$

atom controls *pc*

reads *pc*, *arrive*, *leave*, *signal* **awaits** *arrive*, *leave*

init

$\parallel true \rightarrow pc' := away$

update

$\parallel pc = away \wedge arrive? \rightarrow pc' := wait$

$\parallel pc = wait \wedge signal = green \rightarrow pc' := bridge$

$\parallel pc = bridge \wedge leave? \rightarrow pc' := away$

Design Problem

- Train traveling clockwise:

module $Train_W$ **is**
 $Train[pc, arrive, signal, leave :=$
 $pc_W, arrive_W, signal_W, leave_W]$

- Train traveling anticlockwise:

module $Train_E$ **is**
 $Train[pc, arrive, signal, leave :=$
 $pc_E, arrive_E, signal_E, leave_E]$

- Safety requirement:

$safe: \quad \neg(pc_W = bridge \wedge pc_E = bridge)$

- Design Controller so that $safe$ is an invariant of the compound module

$RailroadSystem = \mathbf{hide} \text{ } arrive_W, arrive_E, leave_W, leave_E \mathbf{in}$
 $\parallel Train_W$
 $\parallel Train_E$
 $\parallel Controller.$

First Attempt at Controller Design

```
module Controller1 is  
  interface  $signal_W, signal_E: \{green, red\}$   
  external  $arrive_W, arrive_E, leave_W, leave_E: \mathbf{event}$   
  passive atom controls  $signal_W, signal_E$   
    reads  $signal_W, signal_E, arrive_W, arrive_E, leave_W, leave_E$   
    awaits  $arrive_W, arrive_E, leave_W, leave_E$   
  init  
     $\parallel true \rightarrow signal'_W := green; signal'_E := green$   
  update  
     $\parallel arrive_W? \rightarrow signal'_E := red$   
     $\parallel arrive_E? \rightarrow signal'_W := red$   
     $\parallel leave_W? \rightarrow signal'_E := green$   
     $\parallel leave_E? \rightarrow signal'_W := green$ 
```

Invariant Verification

- Does RailroadSystem with Controller1 satisfy the invariant?
- Answer is No.
- Error-trajectory:

<i>pc_W</i>	<i>away</i>	<i>wait</i>	<i>bridge</i>	<i>away</i>	<i>away</i>	<i>wait</i>	<i>bridge</i>
<i>arrive_W</i>		◇				◇	
<i>leave_W</i>				◇			
<i>pc_E</i>	<i>away</i>	<i>wait</i>	<i>wait</i>	<i>wait</i>	<i>bridge</i>	<i>bridge</i>	<i>bridge</i>
<i>arrive_W</i>		◇					
<i>leave_W</i>							
<i>signal_W</i>	<i>green</i>	<i>green</i>	<i>green</i>	<i>green</i>	<i>green</i>	<i>green</i>	<i>green</i>
<i>signal_E</i>	<i>green</i>	<i>red</i>	<i>red</i>	<i>green</i>	<i>green</i>	<i>red</i>	<i>red</i>

A Second Attempt

```
module Controller2 is  
  private  $near_W, near_E$ : bool  
  interface  $signal_W, signal_E$ : {green, red}  
  external  $arrive_W, arrive_E, leave_W, leave_E$ : event  
  passive atom controls  $near_W$   
  reads  $near_W, arrive_W, leave_W$  awaits  $arrive_W, leave_W$   
  init  
     $\parallel true \rightarrow near'_W := false$   
  update  
     $\parallel arrive_W? \rightarrow near'_W := true$   
     $\parallel leave_W? \rightarrow near'_W := false$   
  passive atom controls  $near_E...$   
  lazy atom controls  $signal_W, signal_E$   
  reads  $near_W, near_E, signal_W, signal_E$   
  init  
     $\parallel true \rightarrow signal'_W := red; signal'_E := red$   
  update  
     $\parallel near_W \wedge signal_E = red \rightarrow signal'_W := green$   
     $\parallel near_E \wedge signal_W = red \rightarrow signal'_E := green$   
     $\parallel \neg near_W \rightarrow signal'_W := red$   
     $\parallel \neg near_E \rightarrow signal'_E := red$ 
```

Beyond Invariants

- Not all properties can be formulated as invariants
- What if correctness refers to the order in which the states appear on possible trajectories?
- Solution: add a monitoring module M and verify invariants of $P \parallel M$
- Monitor: The module M is a monitor of P if
 - M is compatible with P
 - $\text{intf}X_M \cap \text{extl}X_P = \emptyset$
- Monitor only observes, but does not influence the behavior of P .

Monotonicity Example

- Suppose module P has an interface var x , and we want to checkk that in every trajectory of P , x always increases from one round to the next.
- Module P meets the given requirement iff the compound module $P||M$ has the invariant that no alarm is sounded by the monitor M .

```
module MonMonitor is  
  interface alert: {0, 1}  
  external x: nat  
  passive atom controls alert reads x awaits x  
  init  
     $\parallel$  true  $\rightarrow$  alert' := 0  
  update  
     $\parallel$  x'  $\geq$  x  $\rightarrow$  alert' := 0  
     $\parallel$  x' < x  $\rightarrow$  alert' := 1
```

Back to Railroad Control

- Is Controller2 fair to the trains in opposite directions?
- Equal-opportunity requirement: while a train is waiting at a red signal, it is not possible that the signal at the opposite entrance to the bridge turns from green to red and back to green
- Cannot be formulated as an invariant, but we can add a monitor

Monitoring for Equal Opportunity

```
module EqOppMonitor is  
  interface a: {0, 1, 2, 3}  
  external p: {away, wait, bridge}; s1, s2: {green, red}  
  atom controls a reads a, p, s1, s2  
  init  
     $\parallel$  true  $\rightarrow$  a' := 0  
  update  
     $\parallel$  a = 0  $\wedge$  p = wait  $\wedge$  s1 = red  $\wedge$  s2 = green  $\rightarrow$  a' := 1  
     $\parallel$  a = 1  $\wedge$  s1 = green  $\rightarrow$  a' := 0  
     $\parallel$  a = 1  $\wedge$  s1 = red  $\wedge$  s2 = red  $\rightarrow$  a' := 2  
     $\parallel$  a = 2  $\wedge$  s1 = green  $\rightarrow$  a' := 0  
     $\parallel$  a = 2  $\wedge$  s1 = red  $\wedge$  s2 = green  $\rightarrow$  a' := 3
```

Verifying Equal Opportunity

- Monitor for clockwise train:

module *EqOppMonitor_W* **is**
EqOppMonitor[*a*, *p*, *s*₁, *s*₂ := *alert_W*, *pc_W*, *signal_W*, *signal_E*]

- Monitor for anticlockwise train:

module *EqOppMonitor_E* **is**
EqOppMonitor[*a*, *p*, *s*₁, *s*₂ := *alert_E*, *pc_E*, *signal_E*, *signal_W*]

- Verify if

$$\neg(\mathit{alert}_W = 3 \vee \mathit{alert}_E = 3)$$

is an invariant of

RailroadSystem || *EqOppMonitor_W* || *EqOppMonitor_E*.

Error-trajectory for Equal Opportunity

pc_W	pc_E	$signal_W$	$signal_E$	$alert_W$	$alert_E$
<i>away</i>	<i>away</i>	<i>red</i>	<i>red</i>	0	0
<i>wait</i>	<i>away</i>	<i>red</i>	<i>red</i>	0	0
<i>wait</i>	<i>wait</i>	<i>red</i>	<i>red</i>	0	0
<i>wait</i>	<i>wait</i>	<i>red</i>	<i>green</i>	0	0
<i>wait</i>	<i>bridge</i>	<i>red</i>	<i>green</i>	1	0
<i>wait</i>	<i>away</i>	<i>red</i>	<i>red</i>	1	0
<i>wait</i>	<i>wait</i>	<i>red</i>	<i>red</i>	2	0
<i>wait</i>	<i>wait</i>	<i>red</i>	<i>green</i>	2	0
<i>wait</i>	<i>bridge</i>	<i>red</i>	<i>green</i>	3	0