Lecture 2: Modeling Discrete Systems

Foundations of Cyber-Physical Systems

Rupak Majumdar & Bjoern Brandenburg
Administrative Items

• Office: 422 MPI Building
• rupak@mpi-sws.org

• Office Hours: Thursdays 3-4 or by appointment

• TAs:
  – Arpan Gujarati arpanbg@mpi-sws.org
  – Susanne van den Elsen svanden@mpi-sws.org
Acknowledgements

Some of the slides are based on various lectures by (in no particular order)
    Alur (Upenn), Lee, Sangiovanni-Vincentelli, Seshia (UC Berkeley), Fainekos (ASU), Julius (RPI)
ca. 30 electric/electronic systems, 50-100 micro processors, >100 sensors within modern mid-size cars
Challenges in Automotive Electronics Development

Increasing functionality:
- Safety (active/passive)
- Fuel efficiency (hybrid)
- Reduced emissions (less CO2)
- Comfort

Increasing quality:
- 2000: ~1000 - 10ppm (per ECU)
- 2010: ~1 - 0ppm (per ECU)

Increasing value:
- Electronic Share (value):
  2004: 20% -> 2015: 40%
- Software Share (value):
  2000: 4.5% -> 2010: 13%

Reduce time to market:
- 2000: ~20 – 26 months
- 2010: < 18-20 month

Lexus 2006: ~100 CPUs, ~7M LOC
BMW: ~70-100CPUs, ~100M LOC!
Automotive Control Systems

Today: Federated Architectures

Each time a new function is required, the OEM starts a request to suppliers for a new ECU (an integrated HW/SW device realizing the function) to be integrated on the existing networks.

The device is developed by the supplier with its own choice of HW, RTOS, device drivers and communication layers (with some standardization).

The result is:
- Proliferation of ECUs (reaching 100)
- Complex distributed architectures with the need of high bandwidth and therefore multiple networks and gateways
- Complex functional and not-functional (timing) dependencies across the network, which OEMs struggle to control
- Missing opportunities for common set of libraries and (sub)functions
- Limited standardization, flexibility and extensibility
- Limited control on the execution platform by OEMs
From Federated to Integrated Architectures

Tomorrow?: Integrated Architectures

The execution architecture is completely selected and planned by the OEM. OEMs are free to standardize HW, drivers, RTOS and communication layers, leveraging competition among suppliers.

Each time a new function is required, the OEM starts a request to suppliers for new functional content (SW) to be integrated on the existing platform.

The challenges are:
- Moving from specifications of ECUs with message interfaces to the specs of SW components
- Standardize interoperability among components
- Standardize access to the platform services
- Define models that allow to predict the result of the composition (functional and not-functional)
How do we handle the complexity?
Systems and Models

- Calculate
- Abstract
- Build Model
- Test
- Model
- Predict
- Analyze Model
- System
- Mathematics
- Aircraft
# Why don’t Bridges Crash?

## Building Blocks

<table>
<thead>
<tr>
<th></th>
<th>Bridges</th>
<th>CPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Relevant facts</strong>*</td>
<td>Mass, Tensile Strength</td>
<td>?</td>
</tr>
<tr>
<td><strong>2. Model</strong></td>
<td>Free Body Diagram</td>
<td>?</td>
</tr>
<tr>
<td><strong>3. Analysis</strong></td>
<td>Solve Equations</td>
<td>?</td>
</tr>
</tbody>
</table>

* w.r.t. property of interest
What is Modeling?

- Develop insight about a system or artifact through imitation

- Model = artifact that imitates the system of interest

- Mathematical model = Model in the form of a set of definitions and formulas
One of the major applications of real-time systems: Real-time computer control systems

Today’s focus: Getting some understanding of models of hardware and software
Model-Based Design

• Create a mathematical model of all the parts of the embedded system
  – Physical world
  – Control system
  – Software environment
  – Hardware platform
  – Sensors and actuators

• Construct the implementation from the model
  – Ideally: Automatically, like a compiler
  – In practice: Only portions are auto generated
Readings

• Lee & Seshia, Chapters 3 and 5
Synchronous Reactive Component

Delay

bool in →

\[ \text{bool } x := 0 \]

\[ \text{out} := x \land x := \text{in} \]

bool out →

State variables:
Declaration + Initialization

Update code:
To be executed in each round

Inputs

Outputs
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 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...
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 $in$ of type bool
- The set of inputs is $\{0, 1\}$

Example: $I$ contains two variables: int $x$, bool $y$
- Each input is a pair of values (integer and 0/1)
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 $\{0, 1\}$
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
Initialization of state variables specified by Init
- Sequence of assignments to state variables

Semantics of initialization:
- The set \([\text{Init}]\) of initial states, which is a subset of \(Q_S\)

For Delay
- Init is given by the code fragment \(x := 0\)
- The set \([\text{Init}]\) 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 \rightarrow (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 $\text{out:=x ; x:=in}$
  - There are 4 reactions: 0 -0/0-> 0; 0 -1/0-> 1; 1 -0/1-> 0; 1 -1/1-> 1
Multiple Reactions

- During update, either \( x \) is updated to input \( \text{in} \), 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 \([\text{React}]\) of reactions now contains
    - \( 0 -0/0-> 0 \)
    - \( 0 -1/0-> 1; \ 0 -1/0-> 0 \)
    - \( 1 -0/1-> 0; \ 1 -0/1-> 1 \)
    - \( 1 -1/1-> 1 \)
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-> 1
  - 1 -0/1-> 1
Semantic Equivalence

- Both have identical sets of reactions
- Syntactically different but semantically equivalent
Synchronous Reactive Component Def

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

Synchronous languages in practice:

Richer syntactic features to describe React
Key to understanding: what happens in a single reaction?
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 \( n \):
  - Choose an input value \( i_n \) in \( Q_i \)
  - Execute React to produce output \( o_n \) and change state to \( s_{n+1} \)
    - that is, \( s_n - i_n / o_n \rightarrow s_{n+1} \) must be in [React]

- Sample execution:

  \[ s_0 \overset{i_0/o_0}{\rightarrow} s_1 \overset{i_1/o_1}{\rightarrow} s_2 \overset{i_2/o_2}{\rightarrow} s_3 \rightarrow \cdots \]
What does this component do?

bool x := 0; y := 0

if y then out := x
else out := 0;
x := in;
y := ~ y
Extended State Machines

Input: bool press

```plaintext
int x:=0
press=0 ?
  press=0 & x<10 → x:=x+1
  press=1 ?
press=1 | x>=10 → x:=0
```

`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
- **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)

- **State of the component Switch assigns values to mode and x**

- **press=0 ?**
  - press=0 \( \Rightarrow \) \( x:=x+1 \)

- **press=1 ?**
  - press=1 \( \land \) \( x\leq 10 \) \( \Rightarrow \) \( x:=x+1 \)
  - press=1 \( \land \) \( x>10 \) \( \Rightarrow \) \( x:=0 \)
Modified Switch: What executions are possible?

Input: bool press

- **off**
  - press=0 ?
  - press=1 ?
  - int x:=0
  - press=1 | x>=10 → x:=0

- **on**
  - press=0 & x<=10 → x:=x+1
  - press=1 ?

```plaintext
x := 0
```
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

But usually intractable!
Switch: Is it finite-state?

Input: bool press

```
Switch:
  Is it finite-state?

Input: bool press

int x:=0

 press=0 ?
    press=0 & x<10 
    -> x:=x+1
    press=1 ?

press=1 | x>=10 
    -> x:=0
```
Events

- Input/output variable can be of type `event`
- An event can be absent, or present, in which case has a value
  - `event x` means `x` ranges over `{present, absent}`
  - `event(bool) x` means `x` ranges over `{0, 1, absent}`
  - `event(nat) x` means `x` ranges over `{absent, 0, 1, 2, ...}`
- Syntax: `x?` means the test `(x != absent)`
- Syntax: `x!v` means the assignment `x := v`
- Event-based communication:
  - If no value is assigned to an output event, then it is absent (by default)
  - Event-triggered component executes only in those 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

```
int x := 0
if second? then {
  x:=x+1;
  if x==60 then {
    minute!;
    x :=0 }
}
```

- Event-Triggered Components
  - No need to execute in a round where triggering input events absent
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→ 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
What does this component do?

Arbiter

- req1? -> grant1!
- req2? -> grant2!
- ~ req1? & ~ req2?

Event req1

Event req2

Event grant1

Event grant2
A component is input-enabled if for every state $s$ and input $i$, there exists a state $t$ and an output $o$ such that $s$–$i/o$–$t$ 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
Structured modeling

- How do we build complex models from simpler ones
- What are basic operations on components?
Design a component with

- Input: bool in
- Output: bool out
- Output in round n should equal input in round n-2
Instantiation: Create 2 instances of Delay
  - Output of Delay1 = Input of Delay2 = Variable temp
Parallel composition: Concurrent execution of Delay1 and Delay2
Hide variable temp: Encapsulation
Delay1 = Delay[out -> temp]
- Explicit renaming of input/output variables
- Implicit renaming of state variables
- Components (I,O,S,Init,React) of Delay1 derived from Delay

Delay2 = Delay[in -> temp]
Parallel Composition

- DDelay = Delay1 || Delay2
  - Execute both concurrently

- When can two components be composed?
- How to define parallel composition precisely?
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
Output variables of Delay1 || Delay2 is \{temp, out\}

- Note: By default, every output is available to outside world

- If C1 has output vars O1 and C2 has output vars O2 then the product C1 || C2 has output vars O1 U O2
- Input variables of Delay1 || Delay2 is \{in\}
  - Even though temp is input of Delay2, it is not an input of product
- If C1 has input vars I1 and C2 has input vars I2 then the product \( C1 \mid \mid C2 \) has input vars \((I1 \cup I2) \setminus (O1 \cup O2)\)
  - 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 Delay1 || Delay2 equals \{x_1, x_2\}
- If C1 has state vars S1 and C2 has state vars S2 then the product C1 || C2 has state vars (S1 U S2)
  - A state of the product is a pair (s_1, s_2), where s_1 is a state of C1 and s_2 is a state of C2
  - If C1 has n_1 states and C2 has n_2 states then the product has n_1*n_2 states
The initialization code Init for Delay1 || 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 || C2 has initialization Init1; Init2
- Order does not matter
- [Init] is the product of sets [Init1] x [Init2]
Reactions of Product

- Execution of Delay1 || 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
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, cannot order the execution of the two
- In presence of such cyclic dependency, composition is disallowed
  - Intuition: Combinational cycles should be avoided
  - But there are exceptions: Constructive semantics (Esterel)
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
Example Interface

A1: \( x_1, \text{in1} \rightarrow y, x_1 \)

A2: \( x_2 \rightarrow \text{out2} \)

A3: \( x_1, \text{in1} \rightarrow \text{out1}, x_1 \)

A4: \( \text{in2}, y, \text{out2} \rightarrow \text{x2}, \text{out3} \)
Relay and Inverter are not compatible since there is a cycle in their combined await dependencies.
Component Compatibility Definition

- **Given:**
  - Component C1 with input vars I1, output vars O1, and awaits-dependency relation \( >_1 \)
  - Component C2 with input vars I2, output vars O2, and awaits-dependency relation \( >_2 \)

- The components C1 and C2 are compatible if
  - No common outputs: sets O1 and O2 are disjoint
  - The relation \( (>_1 \cup>_2) \) of combined await-dependencies is acyclic

- Parallel Composition is allowed only for compatible components
Defining the Product

```
Delay1

bool x1 := 0
A1 : in, x1 -> temp, x1
  temp:=x1 ; x1:= in

Delay2

bool x2 := 0
A2 : temp, x2 -> out, x2
  out:=x2 ; x2:= temp

Delay1 || Delay2

bool x1 := 0; x2:=0
A1 : in, x1 -> temp, x1
  temp:=x1 ; x1:= in
A2 : temp, x2 -> out, x2
  out:=x2 ; x2:= temp
```
Properties of Parallel Composition

- Commutative: C1 || C2 is same as C2 || C1
- Associative: Given C1, C2, C3, all of (C1 || C2) || C3, C1 || (C2 || C3), (C1 || C3) || C2, ... 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 C1 and C2 are finite-state, then so is product C1 || C2
  - If C1 has n1 states and C2 has n2 states then the product has n1*n2 states
- If both C1 and C2 are deterministic, then so is product C1 || C2
Given a component $C$, and an output variable $y$, the result of hiding $y$ in $C$, written as $C\backslash y$, 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

bool in

Delay1

bool x1 := 0
A1 : in, x1 -> temp, x1
  temp:=x1 ; x1:= in

Delay2

bool x2 := 0
A2 : temp, x2 -> out, x2
  out:=x2 ; x2:= temp

bool out

bool out

(Delay1 || Delay2) \ temp

bool x1 := 0; x2:=0
local bool temp
A1 : in, x1 -> temp, x1
  temp:=x1 ; x1:= in
A2 : temp, x2 -> out, x2
  out:=x2 ; x2:= temp

bool in

bool out
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

SyncNot

\[ \text{out} := \sim \text{in} \]

SyncAnd

\[ \text{out} := \text{in1} \& \text{in2} \]

SyncNot

\[ \text{out awaits in} \]

SyncAnd

\[ \text{out awaits in1, in2} \]
Exercise: Design Parity circuit
- Inputs: in₁, in₂, ... inₖ
- Output should be 1 if an even number of input variables equal 1
- How many SyncAnd/SyncNot gates are needed as a function of k?
Synchronous Latch

bool x := choose {0, 1}

A1: x -> out
out := x

A2: x, set, reset -> x
reset=1 -> x:=0
set=1 -> x:=1
(set=0 & reset=0) ?

Deterministic?
Input-enabled?
Designing Counter Circuit (1)

- Are awaits-dependencies acyclic?
Designing Counter Circuit (2)

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

- **CruiseController**
  - event
  - second
  - inc
  - dec
  - rotate
  - nat speed
  - event(nat) cruiseSpeed

- **ThrottleControl**
  - event(real) pressure

- **Clock**
  - event second

- **Sensor**
  - event rotate

- **Driver**
  - event cruise
  - event inc
  - event dec

- **Display**
  - nat speed
  - event(nat) cruiseSpeed
Decomposing CruiseController
Tracking Speed

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

- Recap: In a synchronous model, all components execute in a sequence of (logical) rounds in lock-step
- Asynchronous: Speeds at which different components execute are independent (or unknown)
  - Processes in a distributed system
  - Threads in a typical operating system such as Linux/Windows
- 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 empty/null, or hold 0/1 value
- **Initialization of state variables**: assignment x:=null
- **Input task $A_i$ for processing of inputs**: code: x:=in
- **Output task $A_o$ for producing outputs**: Guard: x != null; code: out:=x; x:=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 enabled, one of them is executed
- Sample Execution:
Example: Asynchronous Increments

```
<table>
<thead>
<tr>
<th>nat x:=0; y:=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_x: x := x+1</td>
</tr>
<tr>
<td>A_y: y := y+1</td>
</tr>
</tbody>
</table>
```

- Internal task: Does not involve input or output channels
  - Can have guard condition and update code
  - Execution of internal task: Internal action
- In each step, execute, either task $A_x$ or task $A_y$
- Sample Execution:
  
  $(0,0) \rightarrow (1,0) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow ... \rightarrow (1,105) \rightarrow (2, 105) ...$

- For every $m$, $n$, state $(x=m, y=n)$ is reachable
  - Interleaving model of concurrency
Asynchronous Merge

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

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

Sample Execution:

\[
([\ ], [\ ]) \rightarrow \text{in1?5} \rightarrow ([5], [\ ]) \rightarrow \text{in2?0} \rightarrow ([5], [0]) \rightarrow \text{out!0} \rightarrow ([5], [\ ]) \rightarrow \text{in1?6} \rightarrow ([5, 6], [\ ]) \rightarrow \text{in2?3} \rightarrow ([5, 6], [3]) \rightarrow \text{out!5} \rightarrow ([6], [3]) \ldots
\]
What does this process do?

int in1

int in2

int out

int+null x1 := null; x2 := null

A1: x1 = null → x1 := in1
A2: x2 = null → x2 := in2

B: (x1 != null) & (x2 != null) →
{ out := x1 + x2;
  x1 := null; x2 := null }
Definition: Asynchronous Process $P$

- 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 (contd): Asynchronous Process P

- 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 U {x} to write-set S
  - Defines a set of input actions of the form s – x?v -> 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 U {y}
  - Defines a set of output actions of the form s – y!v -> 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 – ε -> t
Visually the same as the synchronous case
  - Execution semantics different!
DoubleBuffer

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

- Instantiation: Create 2 instances of Buffer
  - Output of Buffer1 = Input of Buffer2 = Variable temp
- Parallel composition: Asynchronous concurrent execution of Buffer1 and Buffer2
- 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
- Production of output on temp by Buffer1 synchronized with consumption of input on temp by Buffer2
Compiled DoubleBuffer

Buffer1

{0,1,null} x1 := null
A1_i: x1 := in
A1_o: (x1 != null) →
{ temp:=x1; x1:=null}

bool in
->
bool temp
->
bool out

Buffer2

{0,1,null} x2 := null
A2_i: x2 := temp
A2_o: (x2 != null) →
{ out:=x2; x2:=null}

{0,1,null} x1 := null; x2 := null
A1_i: x1 := in1
A2_o: (x2 != null) →
{ out:=x2; x2:=null}
B (A1_o + A2_o): (x1 != null) →
{ local bool temp
temp:=x1; x1:=null; x2:= temp }
Definition of Asynchronous Composition

- Given asynchronous processes P1 and P2, how to define P1 | P2?
- Note: In each step of execution, only one task is executed
  - Concepts such as await-dependencies, compatibility of interfaces, are not relevant
- Sample case:
  - 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: Guard1 → Update1
  - A2 is an input task of P2 for y with code: Guard2 → Update2, then
  - Composition has an output task for y with code:
    (Guard1 & Guard2) → Update1 ; Update2
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
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
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 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 “P1 writes x” channel
  - To read y, P2 synchronizes with y on “P2 reads y” channel
By definition of our asynchronous model, each step of above is either internal to P1 or P2, or involves exactly one synchronization: either read or write of 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 incremented value
- But these two are separate steps, and register value can be changed in between by another process
Shared Memory Programs

AtomicReg nat x := 0

Process P1

nat y1:=0

y1 := x

x := y1 +1

Process P2

nat y2:=0

y2 := x

x := y2 +1

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

Can be formalized as asynchronous processes
Data Races

AtomicReg nat x := 0

Process P1

nat y1:=0
R1: y1 := x
W1: x := y1 +1

Process P2

nat y2:=0
R2: y2 := x
W2: x := y2 +1

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 shared object where the result depends on order of execution
Should be avoided!
What possible values can the shared register $x$ take?
Mutual Exclusion Problem

- 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

- **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: First Attempt

AtomicReg bool flag1 := 0; flag2 := 0

Process P1

Idle → Try
flag1 := 1
Try → Crit
flag2=0 ?
else
flag1 := 0

Process P2

Idle → Try
flag2 := 1
Try → Crit
flag1=0 ?
else
flag2 := 0

What's the bug?
Peterson’s Mutual Exclusion Protocol

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

Process P1

Idle → flag1 := 1
Try1 → turn := 1
Try2 → flag2=1? then Crit else turn := 1
Try3 → turn=2? then Crit else flag2 := 0

Process P2

Idle → flag2 := 1
Try1 → turn := 2
Try2 → flag1=1? then Crit else turn := 1
Try3 → turn=1? then Crit else flag1 := 0
Test&Set Register

- Beyond atomic registers:
  - In one (atomic) step, can do more than just read or write
  - Stronger synchronization primitives

- Test&Set Register: Holds a Booleans value
  - Reset operation: Changes the value to 0
  - Test&Set operation: Returns the old value and changes value to 1
  - If two processes are competing to execute Test&Set on a register with value 0, one will get back 0 and other will get back 1

- Modern processors support strong “atomic” operations
  - Compare-and-swap
  - Load-linked-store-conditional
  - Implementation is expensive (compared to read/write operations)!
Mutual Exclusion using Test&Set Register

Test&SetReg free:= 0

Process P1

Idle → Try (t&s(free)=0 ?) → Crit (else)
reset(free)

Process P2

Idle → Try (t&s(free)=0 ?) → Crit (else)
reset(free)

Is this correct?
Example from Industry: Engine Control

Source:
Delphi Automotive Systems (2001)
Next Time

• Specification and Analysis for Discrete Systems