Introducing Embedded Systems: A Cyber- Physical Systems Approach

Edward A. Lee

Robert S. Pepper Distinguished Professor
UC Berkeley

CPS PI Meeting

Education Keynote

National Harbor, Maryland

October 5, 2012

With special thanks to my collaborators:

- *Jeff Jensen, National Instruments*
- *Sanjit Seshia, UC Berkeley*

Background: Five Years of Experience with "Introduction to Embedded Systems"

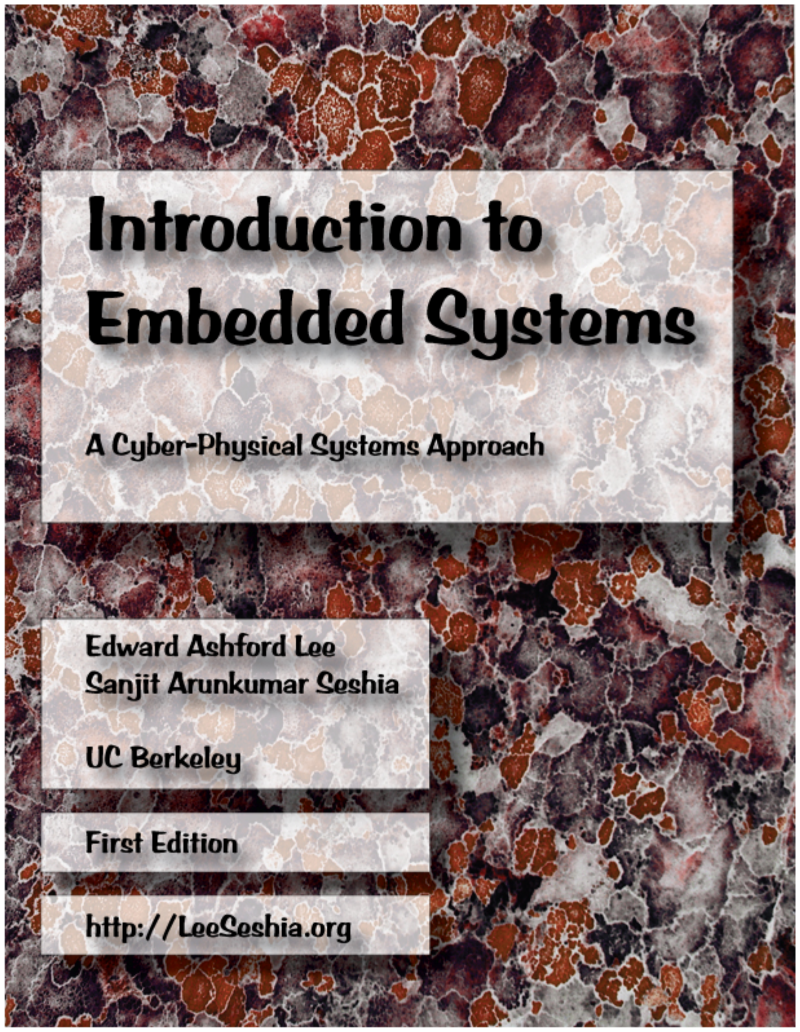
This course is intended to introduce students to the design and analysis of computational systems that interact with physical processes.

A major *theme of this course will be on the interplay of practical design with formal models of systems*, including both software components and physical dynamics. A major emphasis will be on building high confidence systems with real-time and concurrent behaviors.

The course has recently been extended to become a mezzanine-level course, aimed at advanced undergraduates and beginning graduate students. Dovetails with a new professional masters program with a focus on robotics and embedded systems.

- *Cyber-Physical Systems*
- *Model-Based Design*
- *Sensors and Actuators*
- *Interfacing to Sensors and Actuators*
- *Actors, Dataflow*
- *Modeling Modal Behavior*
- *Concurrency: Threads and Interrupts*
- *Hybrid Systems*
- *Simulation*
- *Specification; Temporal Logic*
- *Reachability Analysis*
- *Controller Synthesis*
- *Control Design for FSMs and ODEs*
- *Real-Time Operating Systems (RTOS)*
- *Scheduling: Rate-Monotonic and EDF*
- *Concurrency Models*
- *Execution Time Analysis*
- *Localization and Mapping*
- *Real-Time Networking*
- *Distributed Embedded Systems*

Approach: Interplay of Modeling, Design, and Analysis



Introduction to Embedded Systems

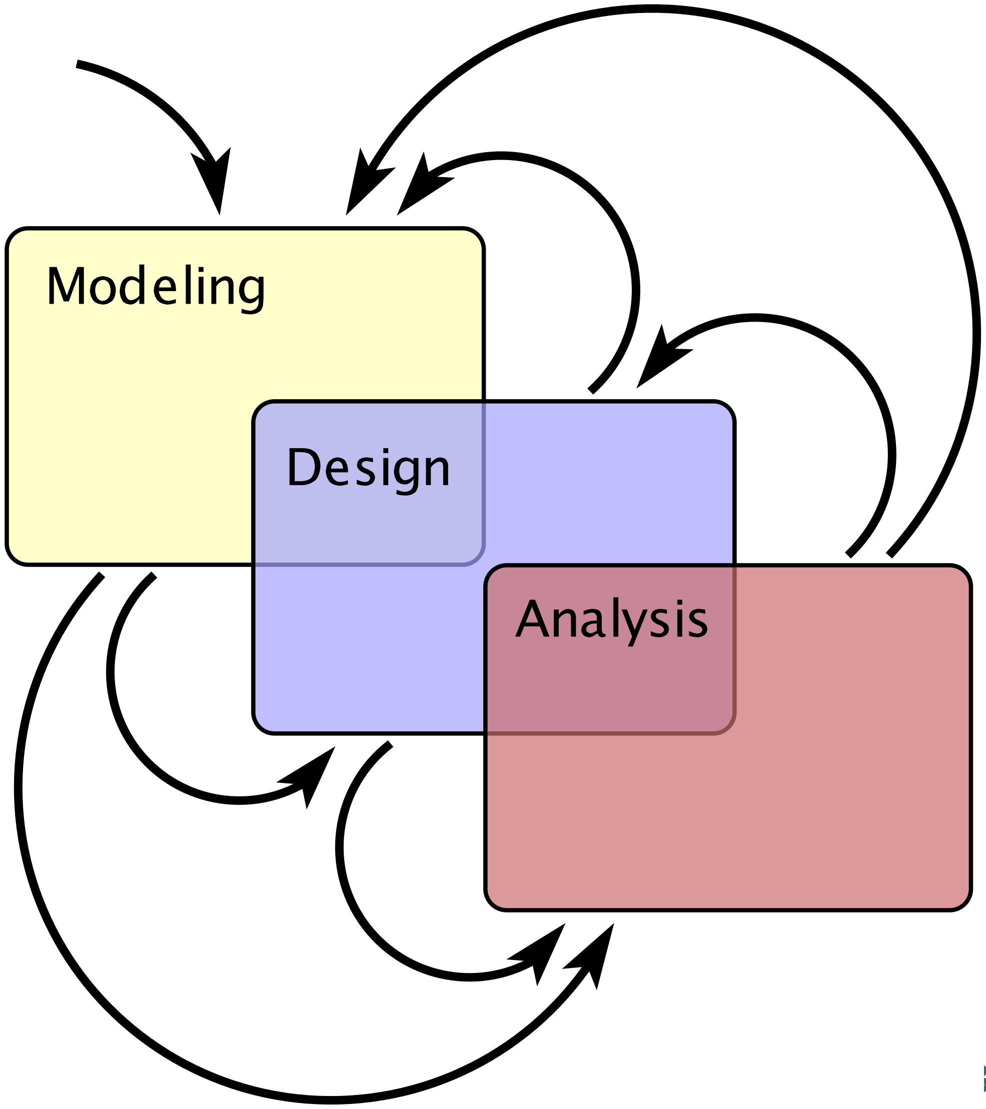
A Cyber-Physical Systems Approach

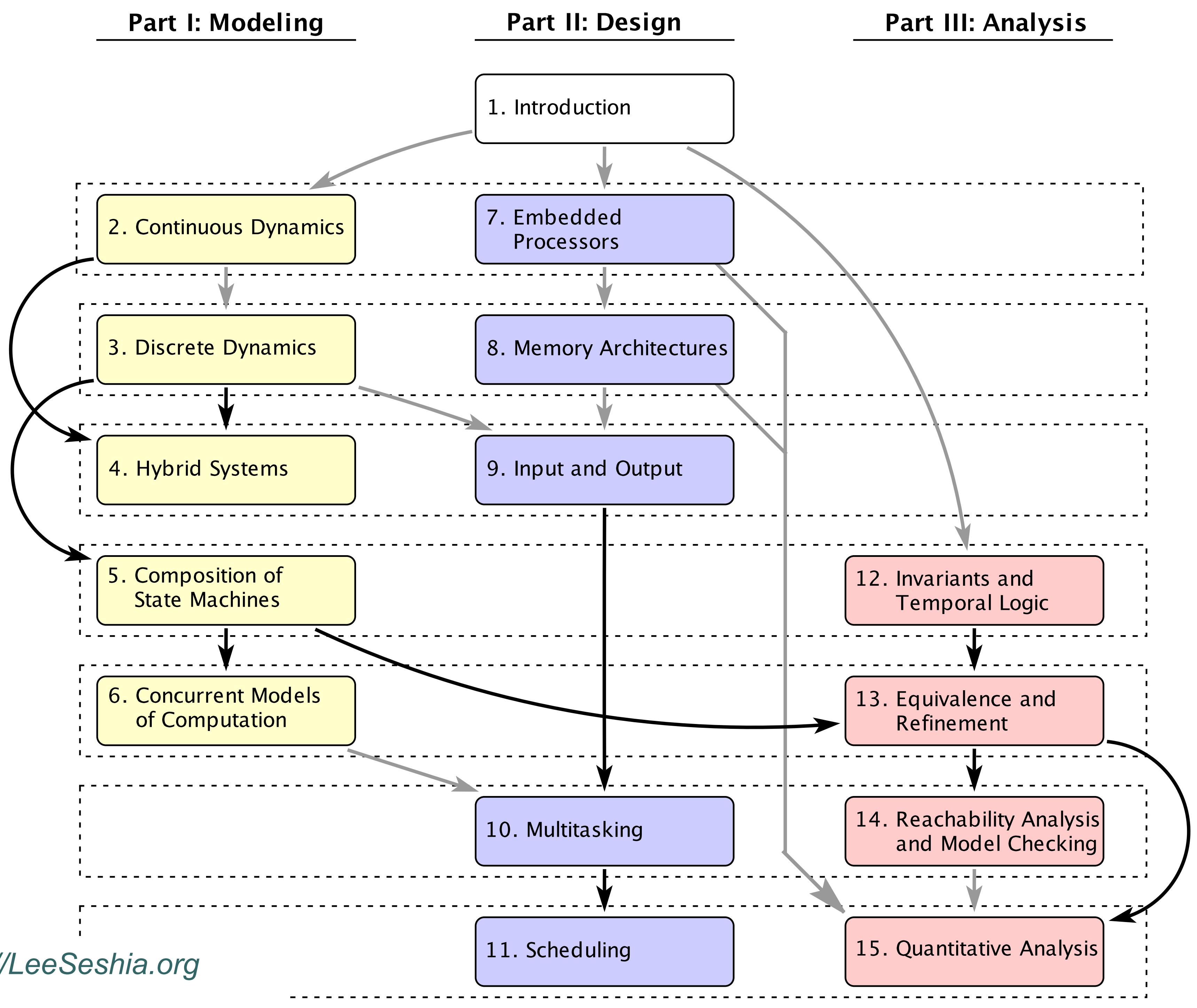
Edward Ashford Lee
Sanjit Arunkumar Seshia

UC Berkeley

First Edition

http://LeeSeshia.org




Part I: Modeling
Part II: Design
Part III: Analysis
1. Introduction
2. Continuous Dynamics
7. Embedded Processors
3. Discrete Dynamics
8. Memory Architectures
4. Hybrid Systems
9. Input and Output
5. Composition of State Machines
12. Invariants and Temporal Logic
6. Concurrent Models of Computation
13. Equivalence and Refinement
10. Multitasking
14. Reachability Analysis and Model Checking
11. Scheduling
15. Quantitative Analysis

This Talk: Focus on *Design* (and specifically, design of *software*)

- *Traditional design of embedded systems:*

  Embedded software is software on small computers. The technical problem is one of optimization (coping with limited resources and extracting performance).

- *CPS-based design of embedded systems:*

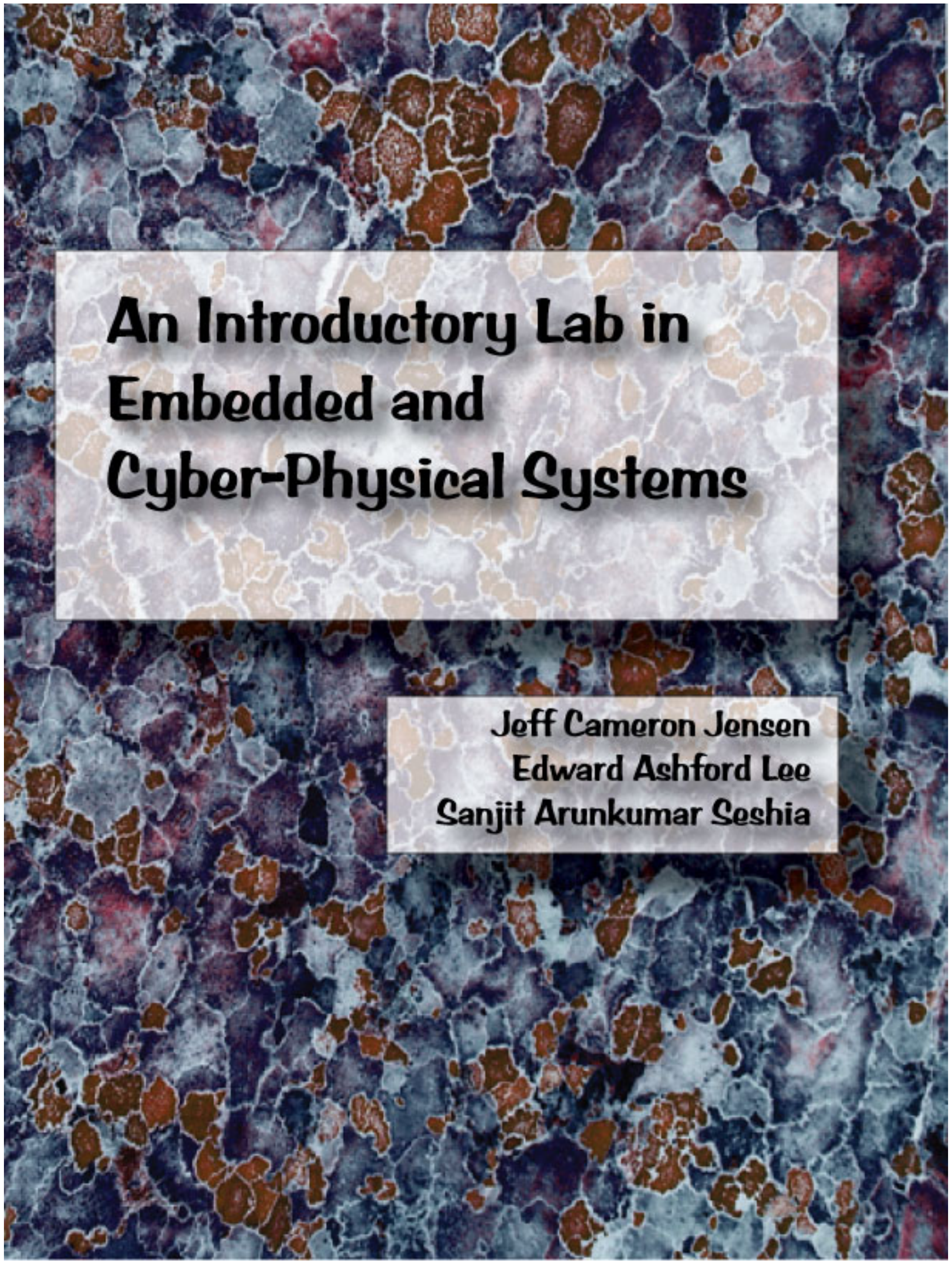
  Computation and networking integrated with physical processes. The technical problem is managing dynamics, time, and concurrency in networked computational + physical systems.

Our Approach:
Emphasis on Critical Thinking

"Our view is that the field of cyber-physical systems is very young, and it would not serve our students well to leave them with the illusion that completing the course equates to mastery of the subject."

[Lee, Seshia, Jensen, WESE 2012]

We teach them to *think critically* about today's technology, not just to master it.


An Introductory Lab in Embedded and Cyber-Physical Systems

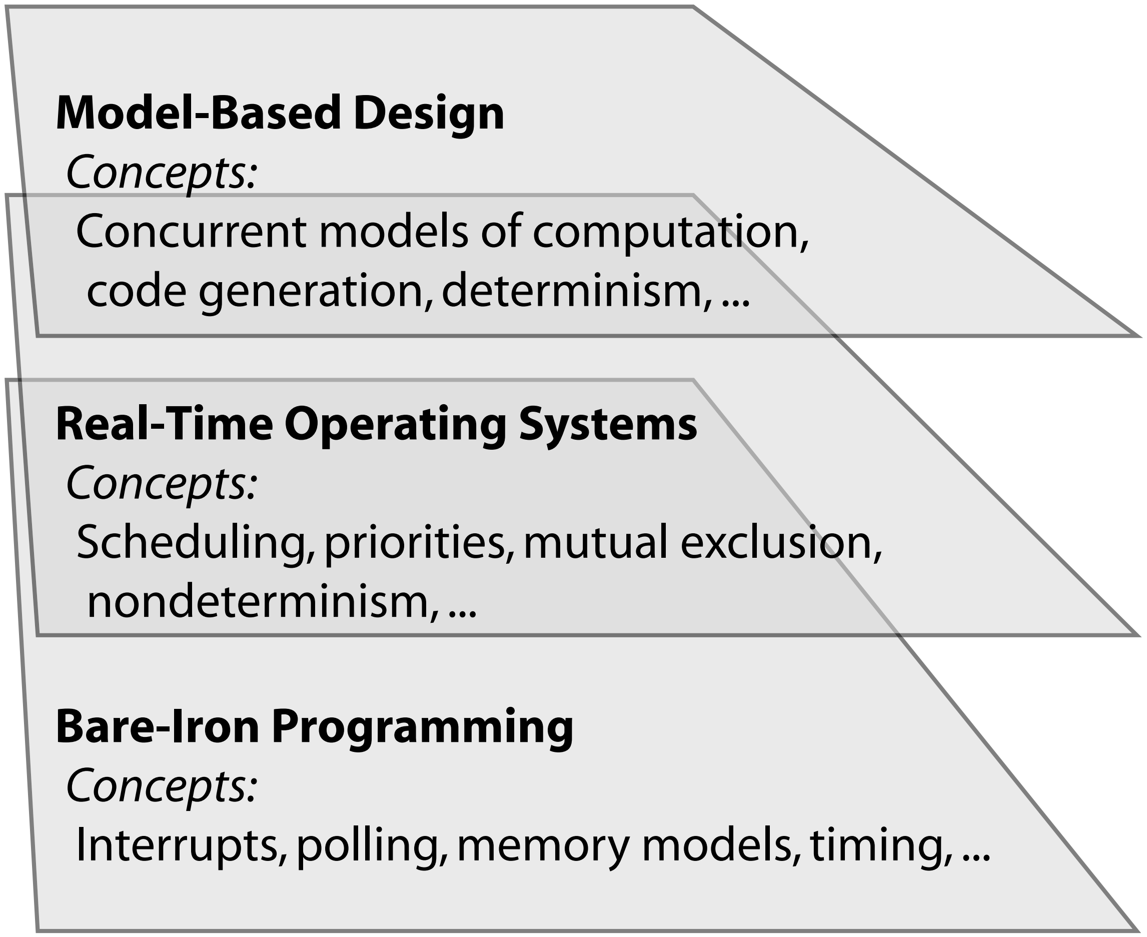
Jeff Cameron Jensen
Edward Ashford Lee
Sanjit Arunkumar Seshia

Design Lab: Structure

- 6 weeks of structured labs introducing students to some of the tools of the trade.
- 9 weeks of group projects.

Draft lab manual available…

The Tools of the Trade


Model-Based Design
Concepts:
Concurrent models of computation, code generation, determinism, ...
Real-Time Operating Systems
Concepts:
Scheduling, priorities, mutual exclusion, nondeterminism, ...
Bare-Iron Programming
Concepts:
Interrupts, polling, memory models, timing, ...

In the first six weeks, students get experience with three levels of abstraction in embedded software design.

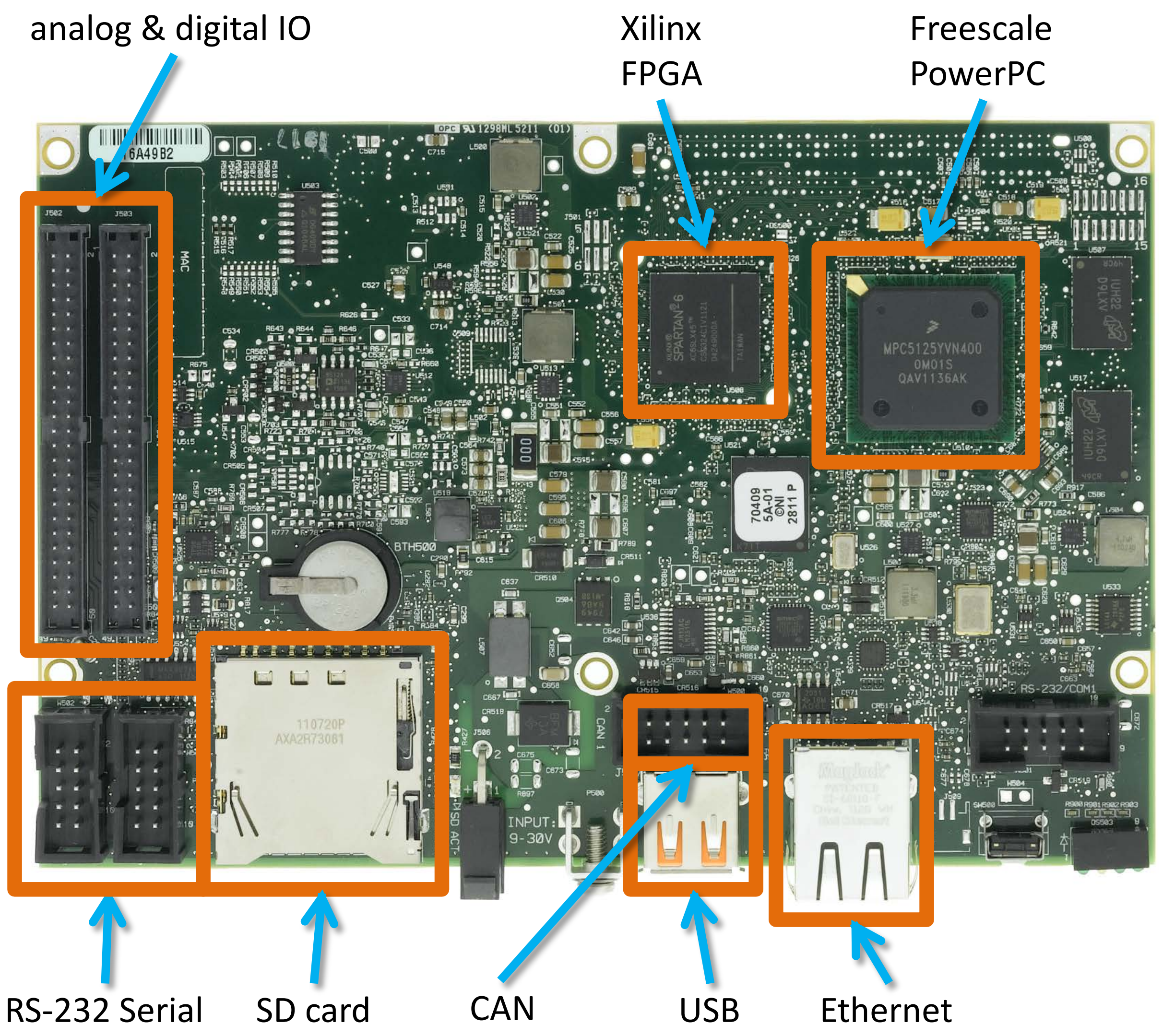
The Hardware Platform for the First Six Weeks

Modified iRobot Create with wireless networking, many built-in sensors, and a three-axis accelerometer.

Computational Platform: Single-Board Rio (National Instruments)

This board provides all three layers of abstraction:

- Bare iron C programming on a Xilinx Microblaze soft core.
- RTOS C programming on a PowerPC running VxWorks.
- LabVIEW model-based design with code generation.



The “Harry Potter” Approach to Embedded Software Design (Bare Iron Level)

On an Atmega 168 (a popular 8-bit microcontroller):

```
// Set timer1 to generate an interrupt every 1ms
TCCR1A = 0x00;
TCCR1B = (_BV(WGM12) | _BV(CS12));
OCR1A = 71;
```

Learn the right spells, and express them with conviction...

The Emphasis on *Critical Thinking* (Bare Iron Level)

```
TCCR1B = (_BV(WGM12) | _BV(CS12));
```

⇩ *Hunt for header files used by the compiler*

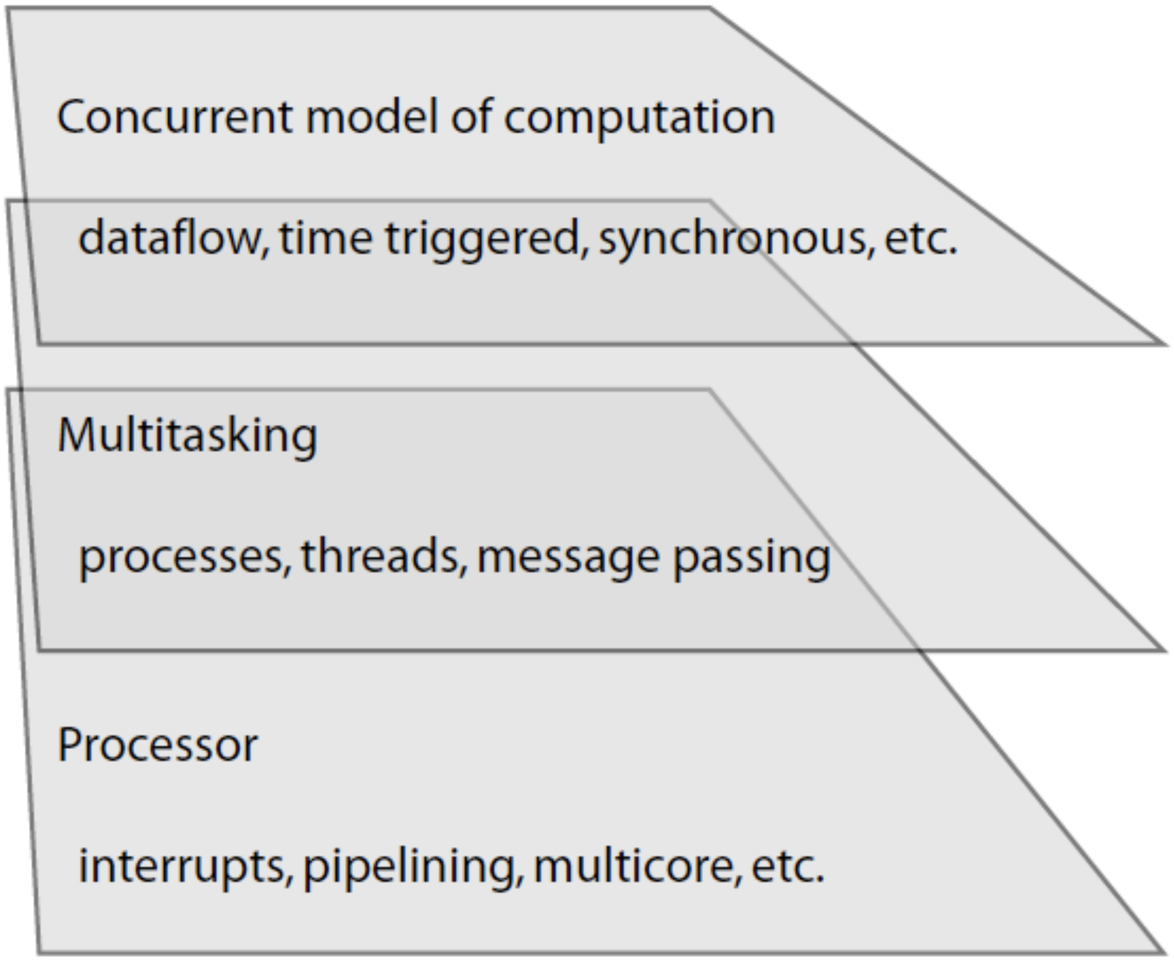
```
#define _MMIO_BYTE(mem_addr)(*(volatile uint8_t *)(mem_addr))
#define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr)
#define _BV(bit) (1 << (bit))
#define TCCR1B _SFR_MEM8 (0x81)
#define WGM12 3
#define CS12 2
```

⇩ *C preprocessor*

```
(*(volatile uint8_t *)(0x81)) = (1 << 3) | (1 << 2);
```

Although TCCR1B appears to be a C variable, it is not (and cannot be, since C provides no way to force a variable to reside at a particular memory address).
Evidently, C is not a perfect match for the problem at hand!

The Emphasis on *Critical Thinking* (RTOS Level)

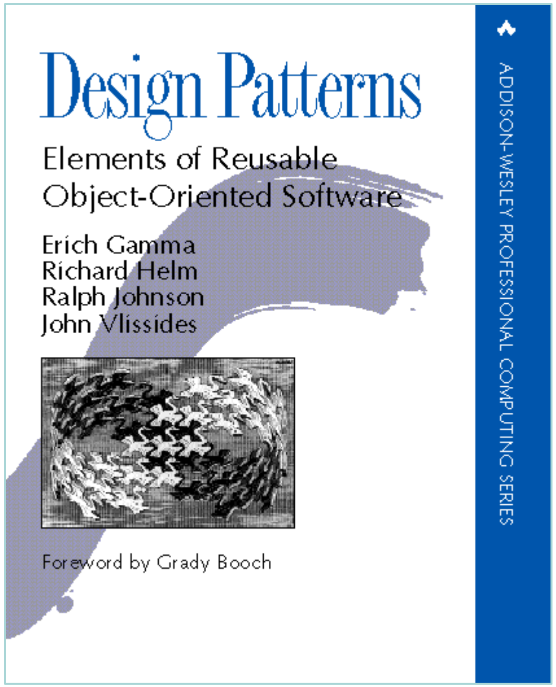

Concurrent model of computation
dataflow, time triggered, synchronous, etc.
Multitasking
processes, threads, message passing
Processor
interrupts, pipelining, multicore, etc.

Levels of abstraction for concurrent programs.

Critical thinking requires understanding pitfalls of scheduling and locks.

A Scenario

Under Integrated Modular Avionics, software in the aircraft engine continually runs diagnostics and publishes diagnostic data on the local network.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Proper software engineering practice suggests using the observer pattern.

An observer process updates the cockpit display based on notifications from the engine diagnostics.

Threads: the Prevailing Concurrency Model

```
#include <pthread.h>
...
int value;
pthread_mutex_t lock;

void addListener(notify listener) {
  pthread_mutex_lock(&lock);
  ... add the listener to the list ...
  pthread_mutex_unlock(&lock);
}

void update(int newValue) {
  pthread_mutex_lock(&lock);
  value = newValue;
  ... copy the list of listeners ...
  pthread_mutex_unlock(&lock);
  ... notify the listeners on the copy ...
}

int main(void) {
  pthread_mutex_init(&lock, NULL);
  ... start diagnostic & observer threads.
}
```

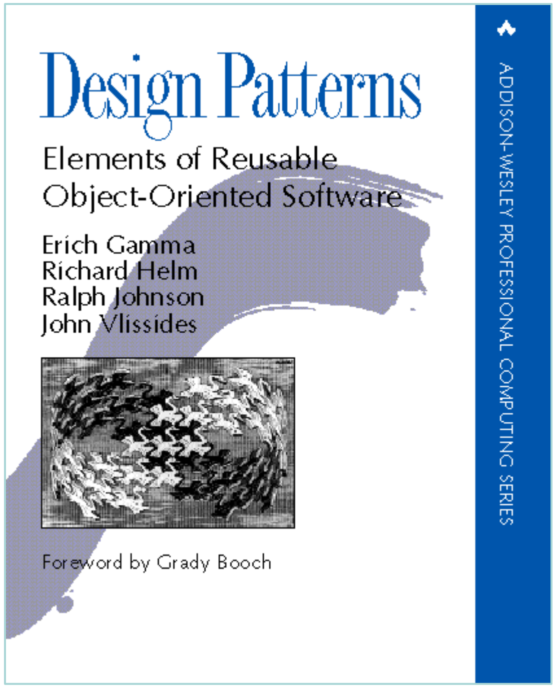
A carefully constructed "thread safe" multitasking solution.

It turns out it carries risk of lurking errors…

If multiple threads call update(), the updates will occur in some order. But there is no assurance that the listeners will be notified in the same order. Listeners may be mislead about the "final" value.

Recall the Scenario

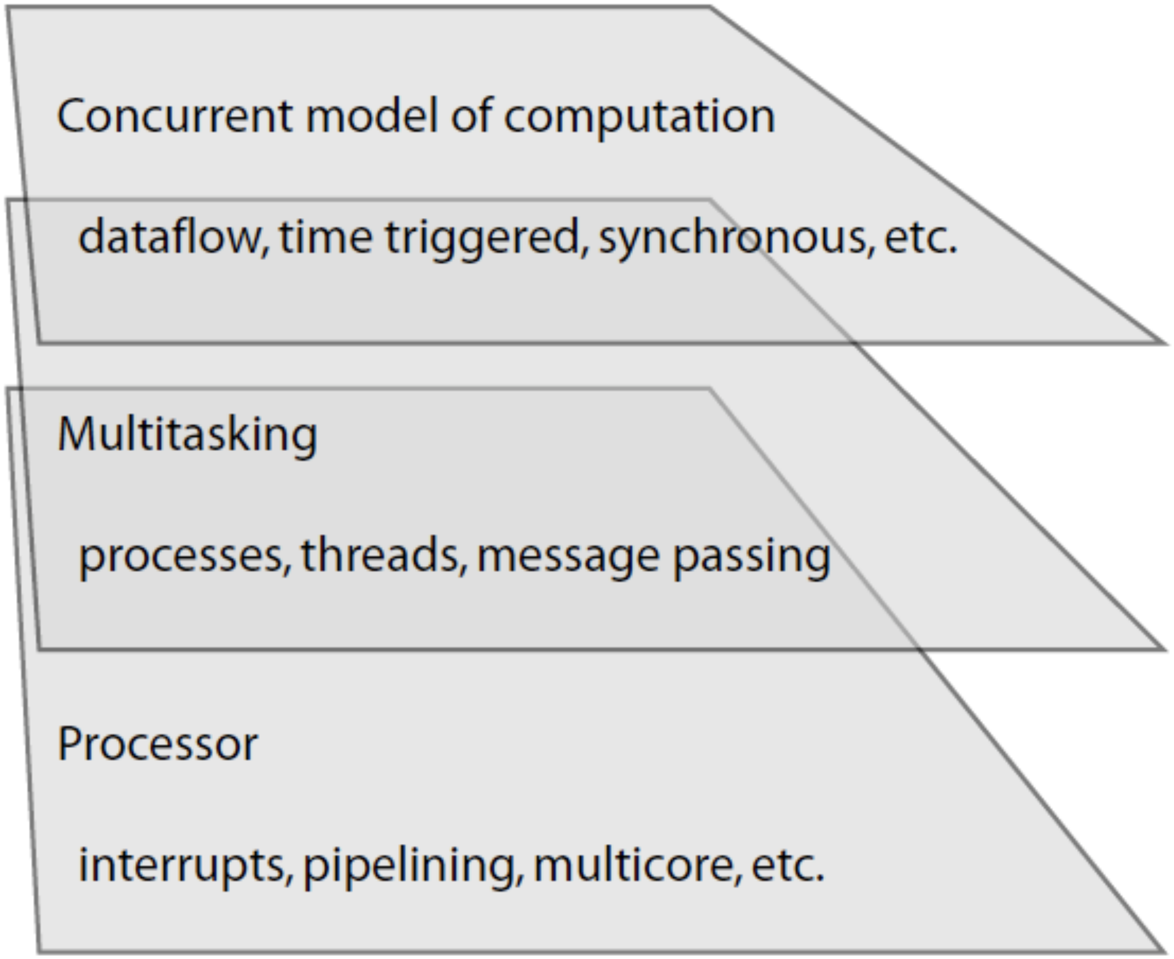
Under Integrated Modular Avionics, software in the aircraft engine continually runs diagnostics and publishes diagnostic data on the local network.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Proper software engineering practice suggests using the observer pattern.

An observer process updates the cockpit display based on notifications from the engine diagnostics.

The Emphasis on *Critical Thinking* (Model-Based Design Level)


Concurrent model of computation
dataflow, time triggered, synchronous, etc.
Multitasking
processes, threads, message passing
Processor
interrupts, pipelining, multicore, etc.

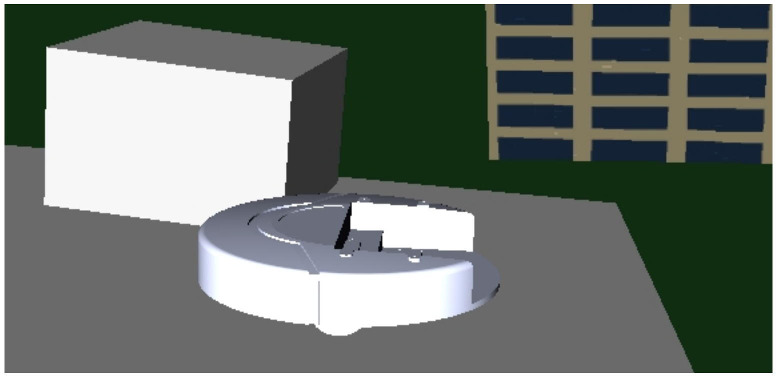
Levels of abstraction for concurrent programs.

Critical thinking requires understanding concurrent models of computation.

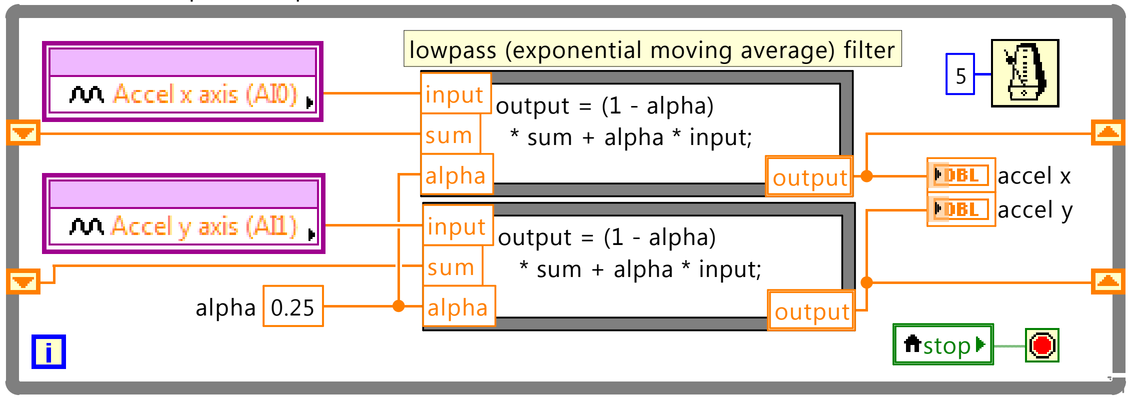
Model-Based Design

Emphasis on concurrency and timing.

Lab experience with LabVIEW, classroom discussion of other model-based design formalisms.



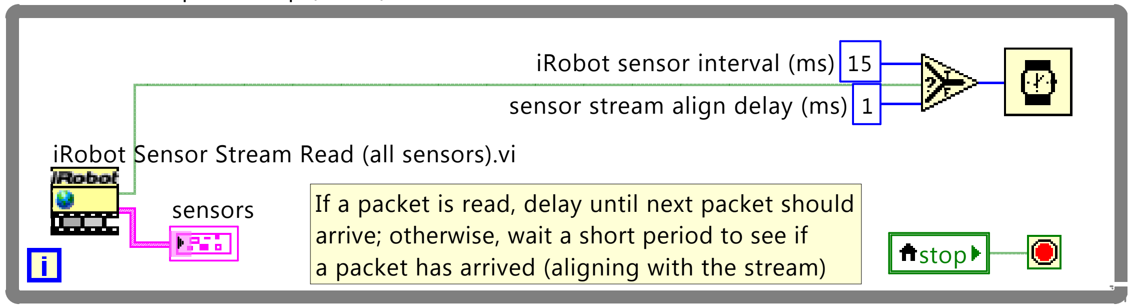
Accelerometer update loop (200 Hz)



```
output = (1 - alpha)
  * sum + alpha * input;
```

```
output = (1 - alpha)
  * sum + alpha * input;
```

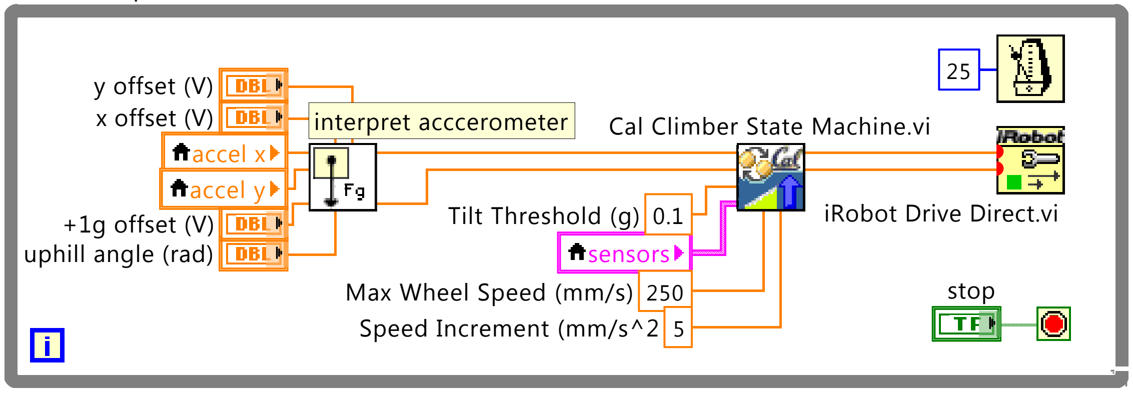
iRobot sensor update loop (67 Hz)



iRobot sensor interval (ms)
15
sensor stream align delay (ms)
1
iRobot Sensor Stream Read (all sensors).vi
iRobot
sensors
stop

If a packet is read, delay until next packet should arrive; otherwise, wait a short period to see if a packet has arrived (aligning with the stream)

Control loop (40 Hz)



Capstone Projects

- Cegway-like two-wheel robot
- Distributed Pacman
- Cooperative self-parking vehicles
- Face-tracking quadrotor
- Robotic convoys
- Elevator operator
- Automatic xylophone
- Dataglove gesture replicator
- Gesture-driven robot steering
- Mapping and localization
- Robotic summo wrestling
- …

Capstone Projects

Video here.

The Canon

Teachers should teach what they know.

Actually, the most valuable teachers are the ones who teach what is not known…

Conclusion

Our job isn't to get our students to replicate us.

Our job is to get our students to *replace* us.

If we succeed, our students will make us obsolete.

If we fail, *their* students will make us obsolete.

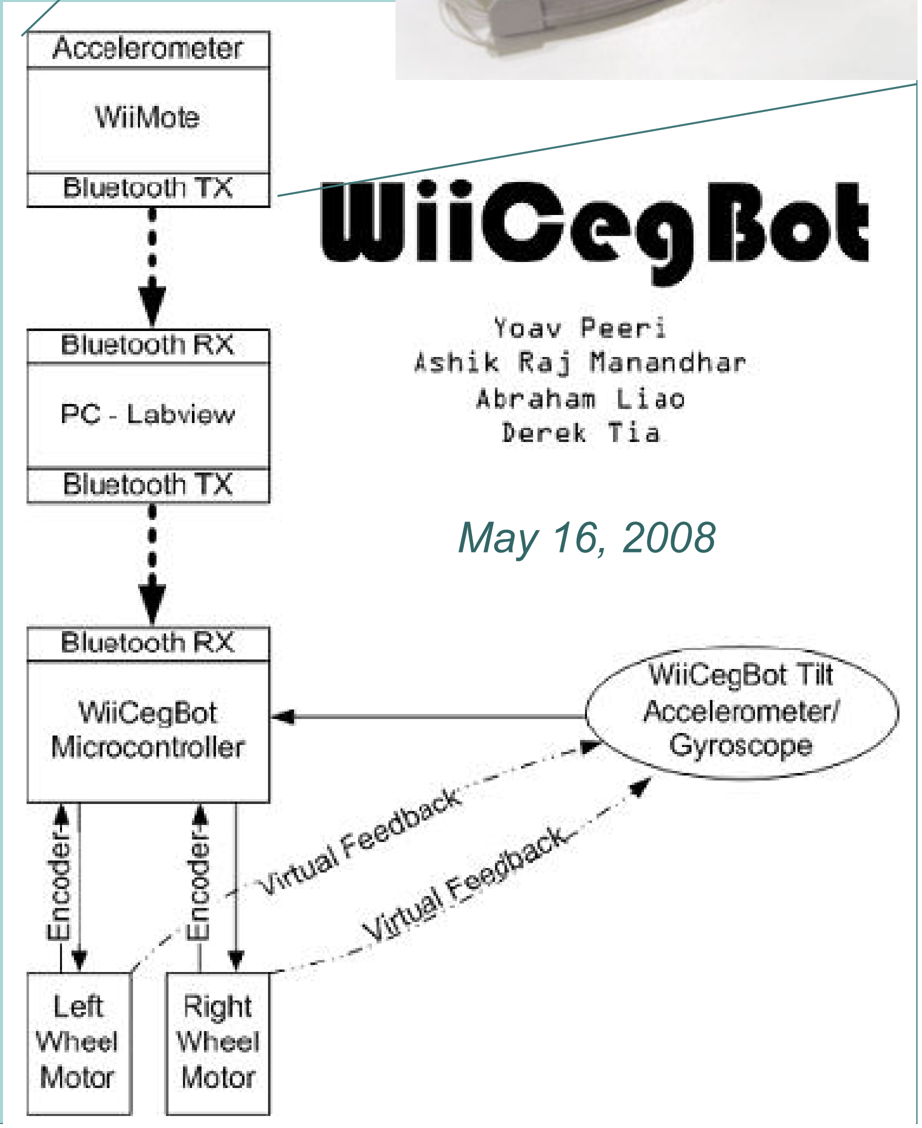
Backup Slides

Applications First? Or Foundations First?

Bottom-up:
- foundations first
- derive the applications

Top-down:
- applications first
- derive the foundations

Class Projects are Defined the Students. Example:



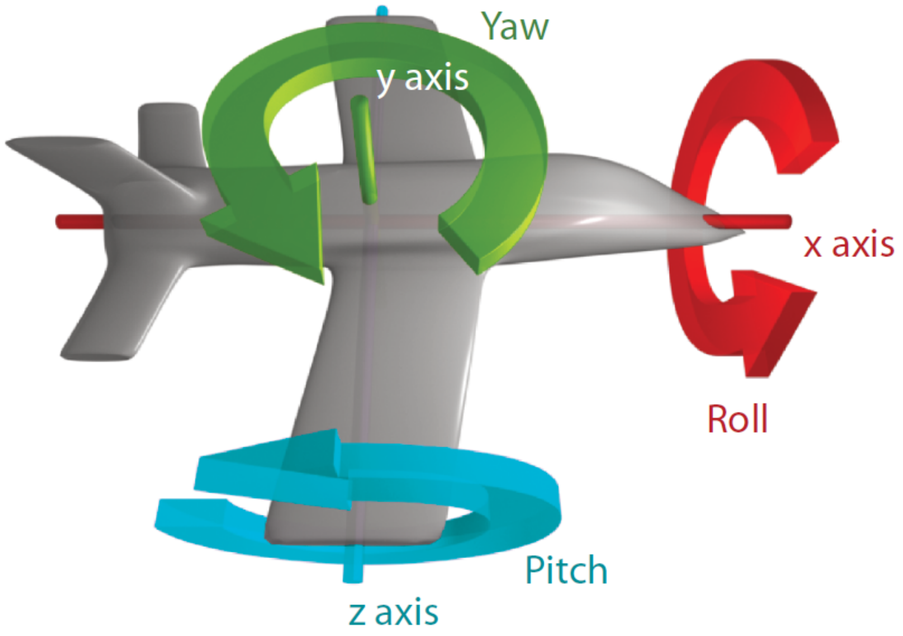
WiiCegBot

Yoav Peeri
Ashik Raj Manandhar
Abraham Liao
Derek Tia

May 16, 2008

One of the five project teams in 2008 developed a balancing robot inspired by the Segway. They used a Nintendo Wiimote as a controller communicating with a PC running LabVIEW, communicating with a Lego Mindstorm NXT, which they programmed in C.

Modeling Physical Dynamics

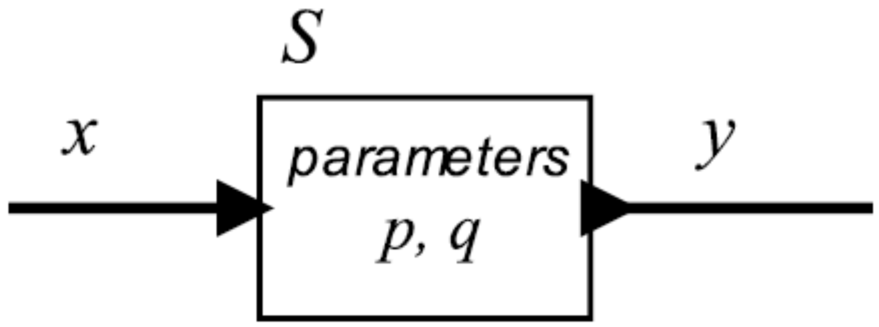


- Orientation: $\theta \colon \mathbb{R} \to \mathbb{R}^3$
- Angular velocity: $\dot{\theta} \colon \mathbb{R} \to \mathbb{R}^3$
- Angular acceleration: $\ddot{\theta} \colon \mathbb{R} \to \mathbb{R}^3$
- Torque: $\mathbf{T} \colon \mathbb{R} \to \mathbb{R}^3$

$$\theta(t) = \begin{bmatrix} \dot{\theta}_x(t) \\ \dot{\theta}_y(t) \\ \dot{\theta}_z(t) \end{bmatrix} = \begin{bmatrix} \text{roll} \\ \text{yaw} \\ \text{pitch} \end{bmatrix}$$

Actor Model of Systems

A *system* is a function that accepts an input *signal* and yields an output signal.



$$x : \mathbb{R} \to \mathbb{R}, \quad y : \mathbb{R} \to \mathbb{R}$$

$$S : X \to Y$$

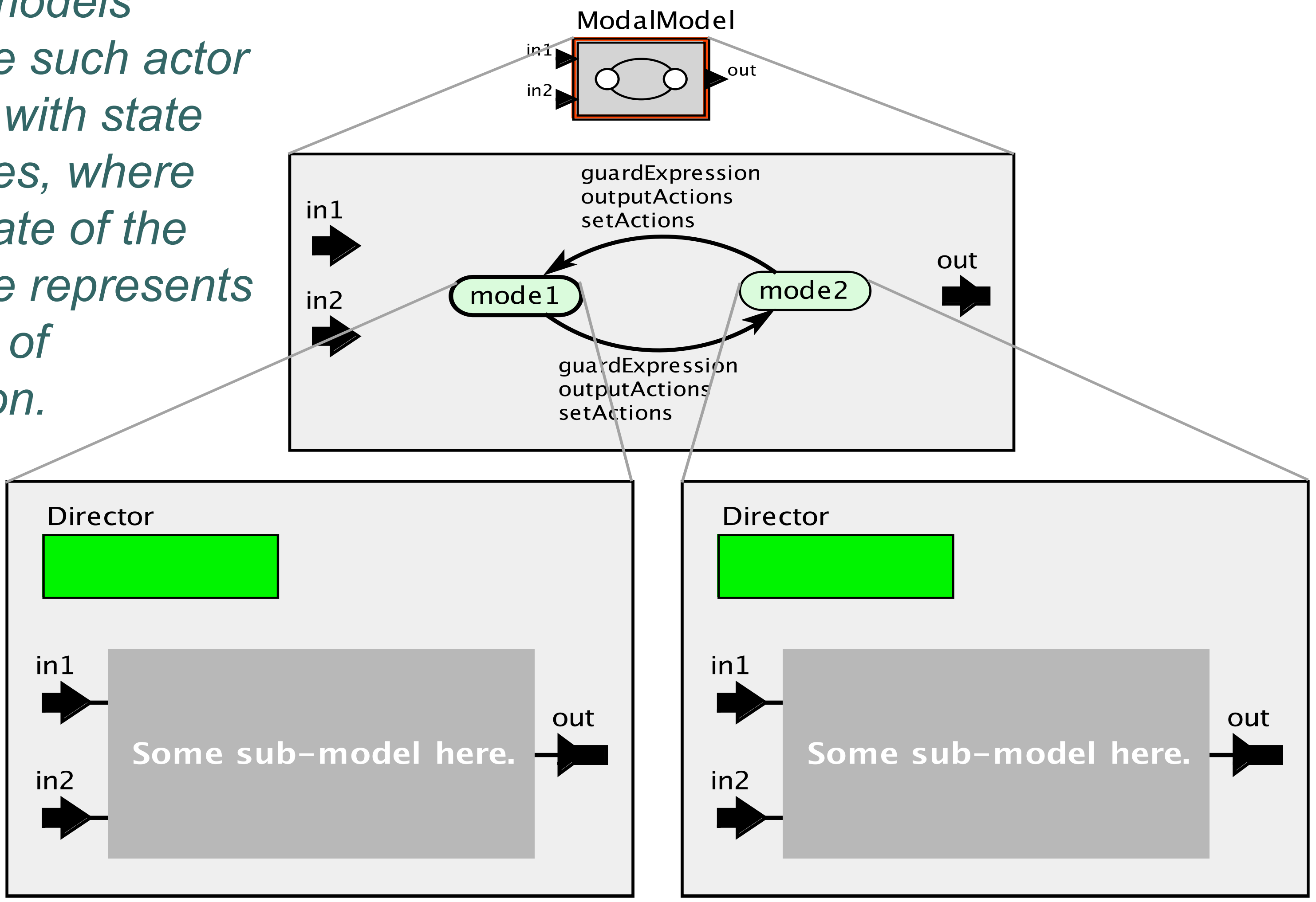
$$X = Y = (\mathbb{R} \to \mathbb{R})$$

The domain and range of the system function are sets of signals, which themselves are functions.

Parameters may affect the definition of the function *S*.

State Machines and Modal Models

Modal models combine such actor models with state machines, where each state of the machine represents a mode of operation.


ModalModel
in1
in2
out
guardExpression
outputActions
setActions
in1
in2
mode1
mode2
out
guardExpression
outputActions
setActions
Director
in1
in2
Some sub-model here.
out
Director
in1
in2
Some sub-model here.
out