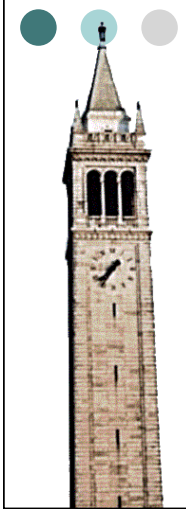


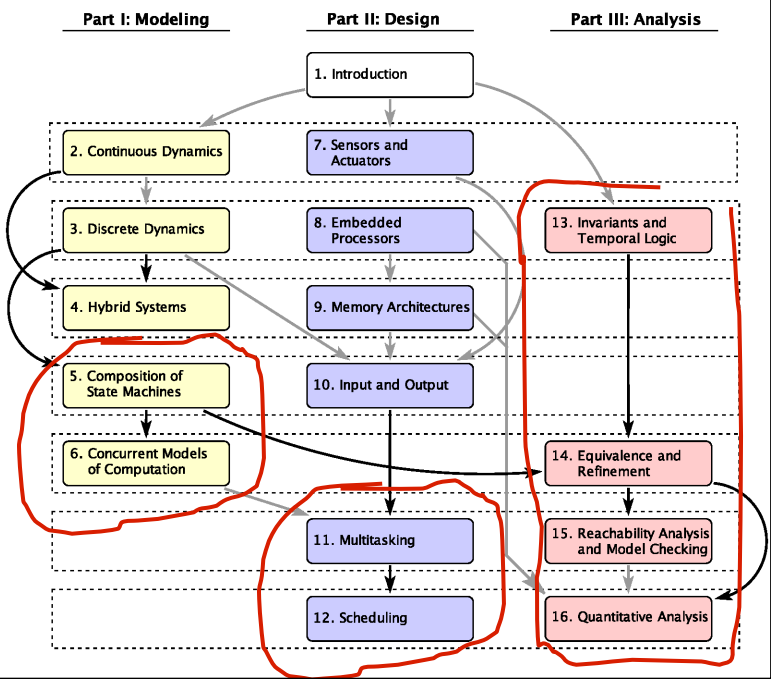
Introduction to Embedded Systems

Edward A. Lee
Alberto Sangiovanni-Vincentelli

UC Berkeley
EECS 149/249A
Fall 2014

Wrapup

Where We've Been



Emphasis for Midterm 2

Cyber-Physical Systems

Orchestrating networked computational resources and physical systems.

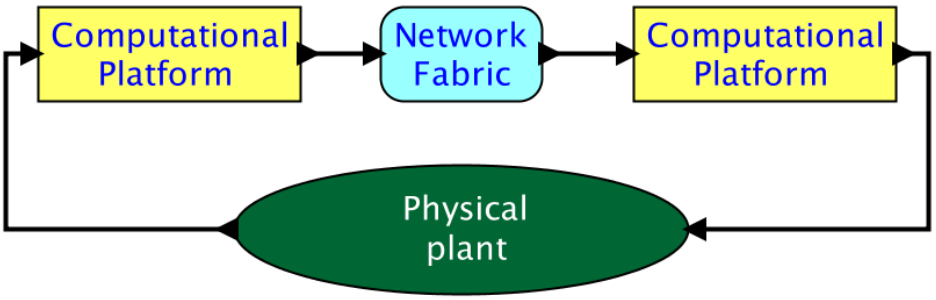
Image: Wikimedia Commons

Roots:

- Coined around 2006 by Helen Gill at the National Science Foundation in the US
- ***Cyberspace**: attributed William Gibson, who used the term in the novel Neuromancer.*
- ***Cybernetics**: coined by Norbert Wiener in 1948, to mean the* conjunction of control and communication.

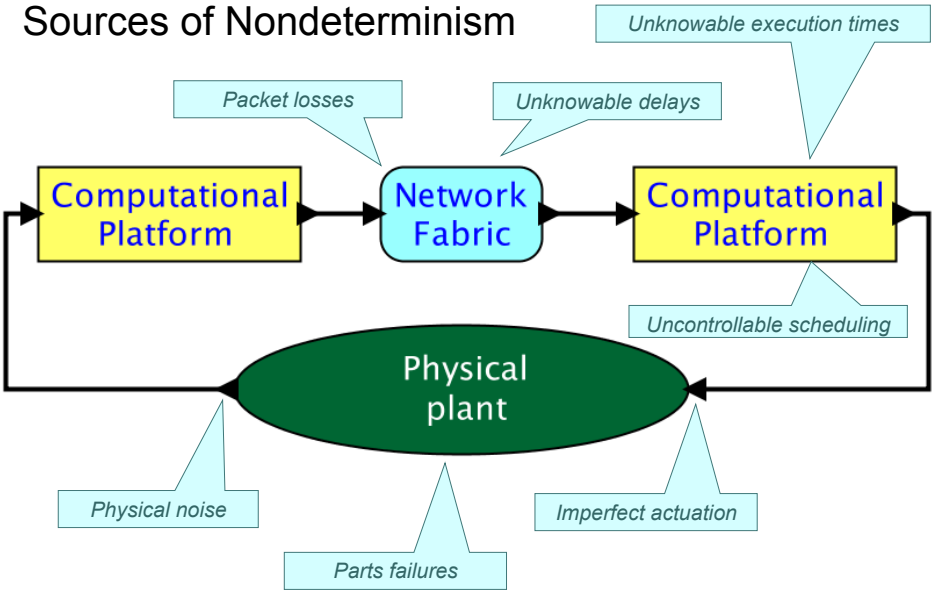
3

Schematic of a simple CPS:



4

Sources of Nondeterminism



5

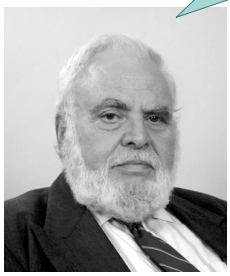
In the face of such nondeterminism, does it make sense to talk about deterministic models for cyber-physical systems?

6

Models vs. Reality

Solomon Golomb: Mathematical models – Uses and limitations. Aeronautical Journal 1968

You will never strike oil by drilling through the map!



Solomon Wolf Golomb (1932) mathematician and engineer and a professor of electrical engineering at the University of Southern California. Best known to the general public and fans of mathematical games as the inventor of polyominoes, the inspiration for the computer game Tetris. He has specialized in problems of combinatorial analysis, number theory, coding theory and communications.

7

But this does not, in any way, diminish the value of a map!

8

The Kopetz Principle

Prof. Dr. Hermann Kopetz

Many (predictive) properties that we assert about systems (determinism, timeliness, reliability, safety) are in fact not properties of an *implemented* system, but rather properties of a *model* of the system.

We can make definitive statements about *models*, from which we can *infer* properties of system realizations. The validity of this inference depends on *model fidelity*, which is always approximate.

(paraphrased)

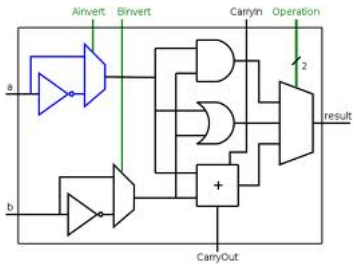
9

Deterministic Models of Nondeterministic Systems

Physical System

Image: Wikimedia Commons

Model



Synchronous digital logic

10

Deterministic Models of Nondeterministic Systems

Physical System

Image: Wikimedia Commons

Model

```
/** Reset the output receivers, which are the inside receivers of
 *  the output ports of the container.
 *  @exception IllegalActionException If getting the receivers fails.
 */
private void _resetOutputReceivers() throws IllegalActionException {
    List<IOPort> outputs = ((Actor) getContainer()).outputPortList();
    for (IOPort output : outputs) {
        if (_debugging) {
            _debug("Resetting inside receivers of output port: "
                    + output.getName());
        }
        Receiver[][] receivers = output.getInsideReceivers();
        if (receivers != null) {
            for (int i = 0; i < receivers.length; i++) {
                if (receivers[i] != null) {
                    for (int j = 0; j < receivers[i].length; j++) {
                        if (receivers[i][j] instanceof FSMReceiver) {
                            receivers[i][j].reset();
                        }
                    }
                }
            }
        }
    }
}
```

Single-threaded imperative programs

11

Deterministic Models of Nondeterministic Systems

Physical System

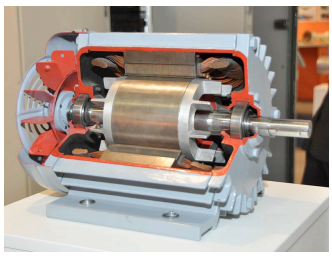


Image: Wikimedia Commons

Model

$$\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \mathbf{F}(\tau)d\tau$$

Differential Equations

12

A Major Problem for CPS:
Combinations of these models are Nondeterministic

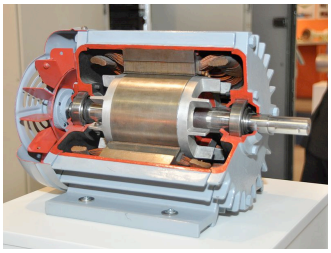


Image: Wikimedia Commons

```
/** Reset the output receivers, which are the inside receivers of
 *  the output ports of the container.
 *  @exception IllegalActionException If getting the receivers fails.
 */
private void _resetOutputReceivers() throws IllegalActionException {
    List<IOPort> outputs = ((Actor) getContainer()).outputPortList();
    for (IOPort output : outputs) {
        if (_debugging) {
            _debug("Resetting inside receivers of output port: "
                    + output.getName());
        }
        Receiver[][] receivers = output.getInsideReceivers();
        if (receivers != null) {
            for (int i = 0; i < receivers.length; i++) {
                if (receivers[i] != null) {
                    for (int j = 0; j < receivers[i].length; j++) {
                        if (receivers[i][j] instanceof FSMReceiver) {
                            receivers[i][j].reset();
                        }
                    }
                }
            }
        }
    }
}
```

$$\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \mathbf{F}(\tau)d\tau$$

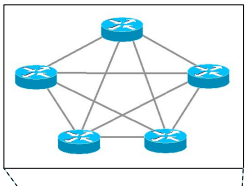
A Key Challenge:
Timing is not Part of Software Semantics

Correct *execution of a program in C, C#, Java, Haskell, OCaml, Esterel, etc. has nothing to do with how long it takes to do anything. Nearly all our computation and networking abstractions are built on this premise.*

Programmers have to step *outside* the programming abstractions to specify timing behavior.

Programmers have no map!

The Model



```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    ... // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```

Computational Platform → Network Fabric → Computational Platform

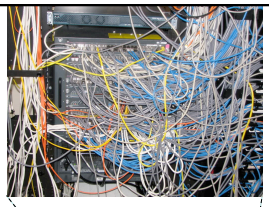
Physical plant

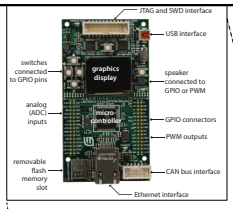
$$\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \mathbf{F}(\tau)d\tau$$

Image: Wikimedia Commons

15

The Reality





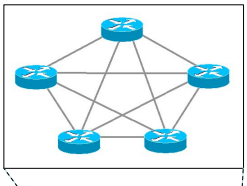
Computational Platform → Network Fabric → Computational Platform

Physical plant

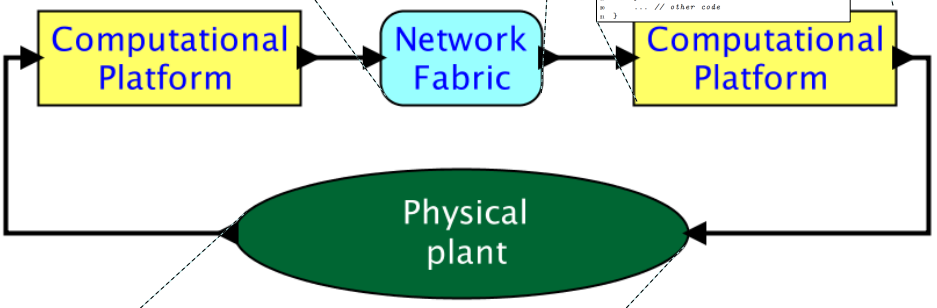
Image: Wikimedia Commons

16

The Model is not much more deterministic than the reality



```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    ... // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```



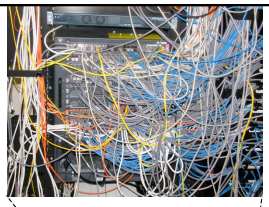
$$\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \mathbf{F}(\tau)d\tau$$

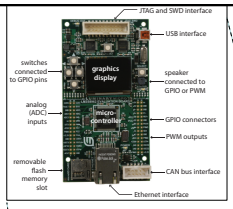
The modeling languages have disjoint, incompatible semantics

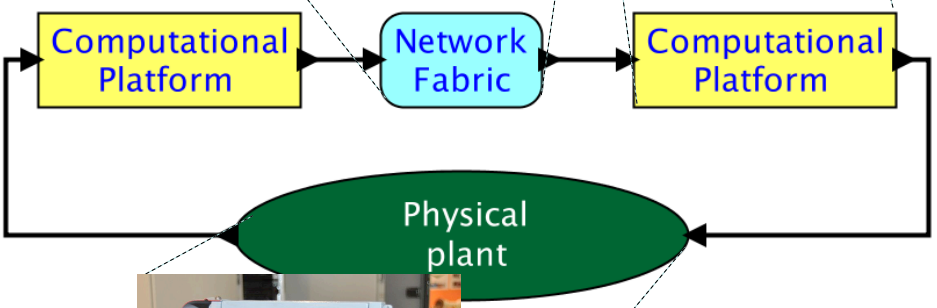
Image: Wikimedia Commons

17

System dynamics emerges from the physical realization







… leading to a "prototype and test" style of design

Image: Wikimedia Commons

18

Our Attempts to Understand the Problem

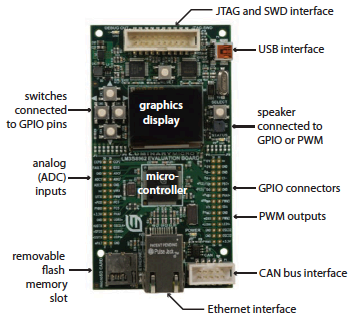


Figure 10.1: Stellaris® LM3S8962 evaluation board (Luminary Micro®, 2008a). (Luminary Micro was acquired by pointerTexas Instruments in 2009.)

10.1.1 Pulse Width Modulation

Pulse width modulation (PWM) is a technique for delivering a variable amount of power efficiently to external hardware devices. It can be used to control for example the speed of electric motors, the brightness of an LED light, and the temperature of a heating element. In general, it can deliver varying amounts of power to devices that tolerate rapid and abrupt changes in voltage and current.

PWM hardware uses only digital circuits, and hence is easy to integrate on the same chip with a microcontroller. Digital circuits, by design, produce only two voltage levels, high and low. A PWM signal rapidly switches between high and low at some fixed frequency, varying the amount of time that it holds the signal high. The duty cycle is the proportion

Chapter 10: I/O

This chapter reviews hardware and software mechanisms used to get sensor data into processors and commands from the processor to actuators. The emphasis is on understanding the principles behind the mechanisms, with a particular focus on the bridging between the sequential world of software and the parallel physical world. This chapter also covers the analog/digital interface from a signal processing perspective, emphasizing the artifacts that may be introduced by quantization, noise, and sampling.

.

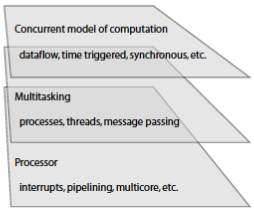


Figure 11.1: Layers of abstraction for concurrency in programs.

taneously on multiple processors or cores. This is also a timing issue, since it presumes that it is better to complete tasks earlier than later. A third reason is to directly control the timing of external interactions. A program may need to perform some action, such as updating a display, at particular times, regardless of what other tasks might be executing at that time.

We have already discussed concurrency in a variety of contexts. Figure 11.1 shows the relationship between the subject of this chapter and those of other chapters. Chapters 8 and 10 cover the lowest layer in Figure 11.1, which represents how hardware provides concurrent mechanisms to the software designer. Chapters 5 and 6 cover the highest layer, which consists of abstract models of concurrency, including synchronous composition, dataflow, and time-triggered models. This chapter bridges these two layers. It describes mechanisms that are implemented using the low-level mechanisms and can provide infrastructure for realizing the high-level mechanisms. Collectively, these mid-level techniques are called **multitasking**, meaning the simultaneous execution of multiple tasks.

Embedded system designers frequently use these mid-level mechanisms directly to build applications, but it is becoming increasingly common for designers to use instead the high-level mechanisms. The designer constructs a model using a software tool that supports a model of computation (or several models of computation). The model is then automatically or semi-automatically translated into a program that uses the mid-level or

Chapter 11: Multitasking

This chapter focuses on mid-level abstractions for concurrent programs, above the level of interrupts and parallel hardware, but below the level of concurrent models of computation. Specifically, it explains threads, mutual exclusion, and semaphores. It shows that threads are fraught with peril, and that writing correct multithreaded programs is extremely difficult. Message passing schemes avoid some of the difficulties, but not all, at the expense of being somewhat more constraining. In the long run, designers will be better off using higher-levels of abstraction, as discussed in Chapter 6.

Example 12.3: The Mars Pathfinder, shown in Figure 12.8, landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. Engineers on the ground diagnosed the problem as priority inversion, where a low priority meteorological task was holding a lock and blocking a high-priority task, while medium priority tasks executed. (**Source**: What Really Happened on Mars? Mike Jones, RISKS-19.49 on the comp.programming.threads newsgroup, Dec. 07, 1997, and What Really Happened on Mars? Glenn Reeves, Mars Pathfinder Flight Software Cognizant Engineer, email message, Dec. 15, 1997.)

Priority inversion is a scheduling anomaly where a high-priority task is blocked while unrelated lower-priority tasks are executing. The phenomenon is illustrated in Figure 12.9. In the figure, task 3, a low priority task, acquires a lock at time 1. At time 2, it is preempted by task 1, a high-priority task, which then at time 3 blocks trying to acquire the same lock. Before task 3 reaches the point where it releases the lock, however, it gets preempted by an unrelated task 2, which has medium priority. Task 2 can run for an unbounded amount

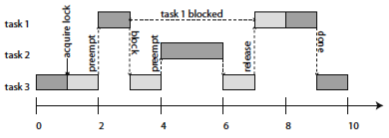


Figure 12.9: Illustration of priority inversion. Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.

Chapter 12: Scheduling

This chapter gives an overview of some of the basic techniques for scheduling real-time tasks and parallel scheduling. It explains some of the pitfalls, such as priority inversion and scheduling anomalies. A designer that is aware of the pitfalls is better equipped to guard against them..

Our Attempts to Solve the Problem

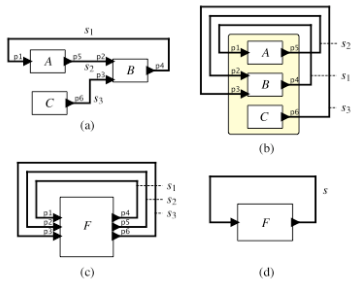


Figure 6.1: Any interconnection of actors can be modeled as a single (side-by-side composite) actor with feedback.

6.1 Structure of Models

In this chapter, we assume that models consist of fixed interconnections of actors like that shown in Figure 6.1(a). The interconnections between actors specify communication paths. The communication itself takes the form of a **signal**, which consists of one or more **communication events**. For the discrete signals of Section 3.1, for example, a signal s has the form of a function

$$s : \mathbb{R} \quad V_s \quad absent \ ,$$

where V_s is a set of values called the type of the signal s. A communication event in this case is a non-absent value of s.

Chapter 6: Concurrent Models of Computation

This chapter provides a whirlwind tour of a rather large topic. It begins with synchronous-reactive models, which are closest to the synchronous composition of state machines considered in the previous chapter. It then considers dataflow models, where execution can be more loosely coordinated. Only data precedences impose constraints on the order of actor computations. The chapter then concludes with a quick view of a few models of computation that explicitly include a notion of time. Such MoCs are particularly useful for modeling cyber-physical systems.

Example 13.9: In Figure 13.1(b), $\mathbf{a}\mathbf{U}x$ is true for any trace for which $\mathbf{F}x$ holds. Since this does not include all traces, $\mathbf{a}\mathbf{U}x$ does not hold for the state machine.

Some authors define a weaker form of the **U** operator that does not require ϕ_2 to hold. Using our definition, this can be written

$$(\mathbf{G}\phi_1) \vee (\phi_1 \mathbf{U} \phi_2) .$$

Probing Further: Alternative Temporal Logics

Amir Pnueli (1977) was the first to formalize temporal logic as a way of specifying program properties. For this he won the 1996 ACM Turing Award, the highest honor in Computer Science. Since his seminal paper, temporal logic has become widespread as a way of specifying properties for a range of systems, including hardware, software, and cyber-physical systems.

In this chapter, we have focused on LTL, but there are several alternatives. LTL formulas apply to individual traces of an FSM, and in this chapter, by convention, we assert than an LTL formula holds for an FSM if it holds for all possible traces of the FSM. A more general logic called **computation tree logic** (**CTL***) explicitly provides quantifiers over possible traces of an FSM (Emerson and Clarke (1980); Ben-Ari et al. (1981)). For example, we can write a CTL* expression that holds for an FSM if there exists *any* trace that satisfies some property, rather than insisting that the property must hold *for all* traces. CTL* is called a **branching-time logic** because whenever a reaction of the FSM has a nondeterministic choice, it will simultaneously consider all options. LTL, by contrast, considers only one trace at a time, and hence it is called a **linear-time logic**. Our convention of asserting that an LTL formula holds for an FSM if it holds for all traces cannot be expressed directly in LTL, because LTL does not include quantifiers like "for all traces." We have to step outside the logic to apply this convention. With CTL*, this convention is expressible directly in the logic.

Other temporal logic variants include **real-time temporal logics** (e.g., **timed computation tree logic** or **TCTL**), for reasoning about real-time systems (Alur et al., 1991); and **probabilistic temporal logics**, for reasoning about probabilistic models such as Markov chains or Markov decision processes (see, for example, Hansson and Jonsson (1994)).

Chapter 13: Temporal Logic

Dependability and correctness are central concerns in embedded systems design. Formal specifications, in turn, are central to achieving these goals. This chapter studies temporal logic, one of the main approaches for writing formal specifications. This chapter provides techniques for precisely stating properties that must hold over time for a system. It specifically focuses on linear temporal logic, which is able to express many safety and liveness properties of systems.

A can accept from the environment, and that any environment that can accept any output A can produce can also accept any output that B can produce.

To make the problem concrete, assume an actor model for A and B, as shown in Figure 14.1. In that figure, A has three ports, two of which are input ports represented by the set $P_A = \{x, w\}$, and one of which is an output port represented by the set $Q_A = \{y\}$. These ports represent communication between A and its environment. The inputs have type V_x and V_w, which means that at a reaction of the actor, the values of the inputs will be members of the sets V_x or V_w.

If we want to replace A by B in some environment, the ports and their types impose four constraints:

1. The first constraint is that B does not require some input signal that the environment does not provide. If the input ports of B are given by the set P_B, then this is guaranteed by

$$P_B \subseteq P_A. \tag{14.1}$$

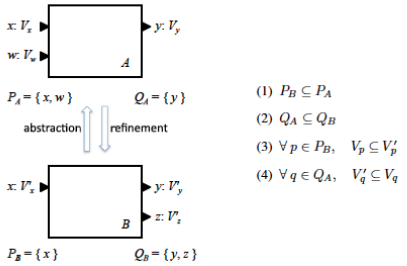


Figure 14.1: Summary of type refinement. If the four constraints on the right are satisfied, then B is a type refinement of A.

Chapter 14: Equivalence and Refinement

This chapter considers three increasingly strong abstraction-refinement relations for FSMs. These relations enable designers to determine when one design can safely replace another, or when one design correctly implements a specification. The relations are type refinement, language refinement, simulation (and bisimulation) relations.

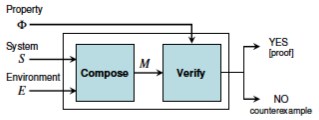


Figure 15.2: Formal verification procedure.

Techniques for formal verification are typically applied to a model of the closed system M obtained by composing the model of the system S that is to be verified with a model of its environment E. S and E are typically open systems, where all inputs to S are generated by E and vice-versa. Thus, as shown in Figure 15.2, there are three inputs to the verification process:

- A model of the system to be verified, S;
- A model of the environment, E, and
- The property to be verified Φ.

The verifier generates as output a YES/NO answer, indicating whether or not S satisfies the property Φ in environment E. Typically, a NO output is accompanied by a counterexample, also called an **error trace**, which is a trace of the system that indicates how Φ is violated. Counterexamples are very useful aids in the debugging process. Some formal verification tools also include a proof or certificate of correctness with a YES answer; such an output can be useful for **certification** of system correctness.

The form of composition used to combine system model S with environment model E depends on the form of the interaction between system and environment. Chapters 5 and 6 describe several ways to compose state machine models. All of these forms of composition can be used in generating a verification model M from S and E. Note that M can be non-deterministic.

For simplicity, in this chapter we will assume that system composition has already been performed using one of the techniques presented in Chapters 5 and 6. All algorithms discussed in the following sections will operate on the combined verification model M,

Chapter 15: Verification

This chapter gives some basic algorithms for formal verification, including model checking, a technique for verifying if a finite-state system satisfies a property specified in linear temporal logic. The first key concept is that of reachability analysis, which verifies properties of the form **Gp**. The concept of abstraction, central to the scalability of model checking, is also discussed in this chapter. This chapter also shows how explicit-state model checking algorithms can handle liveness properties, where a crucial concept is the correspondence between properties and automata.

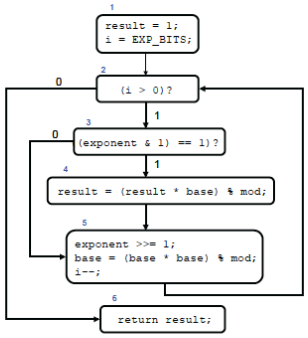


Figure 16.1: Control-flow graph for the `modexp` function of Example 16.1. All incoming edges at a node indicate transfer of control to the start of the basic block for that node, and all outgoing edges from a node indicate an exit from the end of the basic block for that node. For clarity, we label the outgoing edges from a branch statement with `0` or `1` indicating the flow of control in case the branch evaluates to false or true, respectively. An ID number for each basic block is noted above the node for that block; IDs range from 1 to 6 for this example.

Chapter 16: Quantitative Analysis

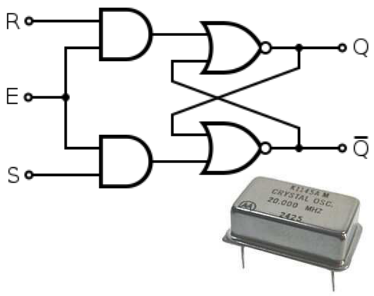
Quantitative properties, involving physical parameters or specifying resource constraints, are central to embedded systems. This chapter gives an introduction to basic concepts in quantitative analysis. The chapter focuses on execution time analysis, considering loop bounds, path feasibility, path explosion, and cache effects.

Further Work: Research

29

The hardware out of which we build computers is capable of delivering “correct” computations and precise timing…

The synchronous digital logic abstraction removes the messiness of transistors.



… but the overlaying software abstractions discard the timing precision.

```
// Perform the convolution.
for (int i=0; i<10; i++) {
  x[i] = a[i]*b[j-i];
  // Notify listeners.
  notify(x[i]);
}
```

30

http://chess.eecs.berkeley.edu/pret

PRET Machines – Giving Programs the Capabilities their Hardware Already Has.

- **PRE**cision-**T**imed processors = **PRET**
- **P**redictable, **RE**peatable **T**iming = **PRET**
- **P**erformance *with* **RE**peatable **T**iming = **PRET**

```
// Perform the convolution.
for (int i=0; i<10; i++) {
  x[i] = a[i]*b[j-i];
  // Notify listeners.
  notify(x[i]);
}
```

Computing

+

With time

= PRET

31

Major Challenges

- Pipelines
  - fine-grain multithreading
- Memory hierarchy
  - memory controllers with controllable latency
- I/O
  - threaded interrupts, with bounded effects on timing

32

Leading to Major Opportunities

- Improved determinism
- Substitutable hardware
- Reduced energy consumption
- Reduced overdesign (cost, weight)
- Better testability
- Improved confidence and safety

33

Example of one sort of mechanism we would like:

```
tryin (500ms) {
   // Code block
} catch {
    panic();
}
```

```
jmp_buf  buf;

if ( !setjmp(buf) ){
  set_time r1, 500ms
  exception_on_expire r1, 0
  // Code block
  deactivate_exception 0
} else {
   panic();
}

exception_handler_0 () {
    longjmp(buf)
}
```

Pseudocode showing the mechanism in a mix of C and assembly.

If the code block takes longer than 500ms to run, then the panic() procedure will be invoked.

But then we would like to verify that panic() is never invoked!

Extending an ISA with Timing Semantics

[V1] Best effort:

```
set_time r1, 1s
// Code block
delay_until r1
```

[V2] Late miss detection

```
set_time r1, 1s
// Code block
branch_expired r1, <target>
delay_until r1
```

[V3] Immediate miss detection

```
set_time r1, 1s
exception_on_expire r1, 1
// Code block
deactivate_exception 1
delay_until r1
```

[V4] Exact execution:

```
set_time r1, 1s
// Code block
MTFD r1
```

Leading to Major Opportunities

- Improved determinism
- Substitutable hardware
- Reduced energy consumption
- Reduced overdesign (cost, weight)
- Better testability
- Improved confidence and safety

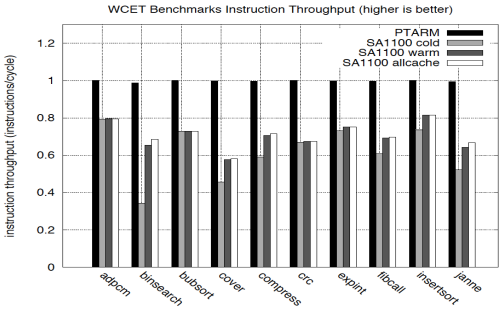
So this must come at a cost in performance, right?

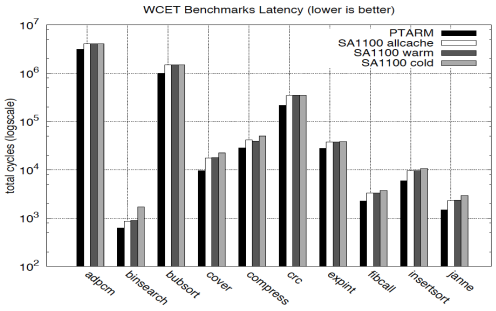
NO!!!

36

The Bottom Line

In microarchitecture design, we have shown that you do not need to sacrifice performance to get control over timing.

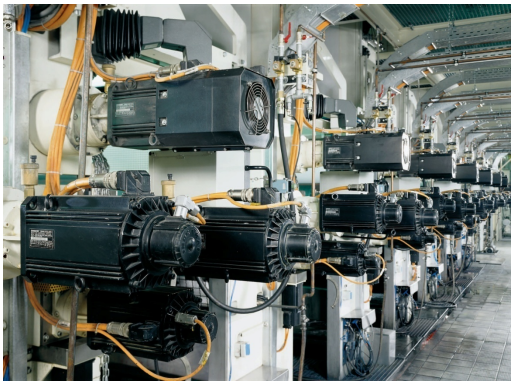




[Isaac Liu, PhD Thesis, May, 2012] 37

Next Challenge: Distributed Cyber-Physical Systems

Example: Printing Press



Bosch-Rexroth

Hundreds of microcontrollers and an Ethernet network are orchestrated with precisions on the order of microseconds.

Software for such systems can be developed in a completely new way.

Clock synchronization enables tightly coordinated actions and reliable networking with bounded latency, despite using TCP/IP.

38

EE 249B: Design of Embedded Systems: Models, Validation and Synthesis

This class merges theoretical aspects with applications to give students a grasp of the fundamentals of system design as well as a taste of the problems posed by complex applications. It presents theories, design methods and tools that help handle the growing complexity and heterogeneity of embedded and cyber-physical systems. Approaches to a new "system science" are demonstrated, where heterogeneity, concurrency, multiple levels of abstraction play a fundamental role and where a set of correct-by-construction refinement and composition techniques are used to substantially reduce design time and errors.

39

Related Classes

- EE125: Introduction to Robotics
- EE128 / ME 134: Feedback Control Systems
- ME 135: Design of Microprocessor-Based Mechanical Systems
- EE144: Fundamental Algorithms for Systems Modeling, Analysis
- CS150: Components and Design Techniques for Digital Systems
- CS152: Computer Architecture and Engineering
- CS162: Operating Systems and System Programming
- CS169: Software Engineering
- EE192: Mechatronics

40

Followup (Advanced) Classes

- EE 219A: Numerical Simulation and Modeling
- EE 219C: Computer-Aided Verification
- EE 144/244: Fundamental Algorithms for Systems Modeling, Analysis
- EE 249B: Design of Embedded Systems: Models, Validation and Synthesis
- CS 260A: User Interface Design and Development
- CS 262A: Advanced Topics in Computer Systems
- CS 287: Advanced Robotics

41

Conclusion

Today, timing behavior in programs and networks emerges from the physical realization.

Tomorrow, timing behavior will be part of the programming abstractions and the hardware realizations.

Raffaello Sanzio da Urbino – The Athens School *Image: Wikimedia Commons*

42