Session 3A—System-Level Design and Specification

Heterogeneously-Specified Synchronous Controllers

Stephen Edwards
Edward A. Lee

{sedwards,eal}@eecs.berkeley.edu

http://ptolemy.eecs.berkeley.edu/~{sedwards,eal}

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

March 14th, 1996

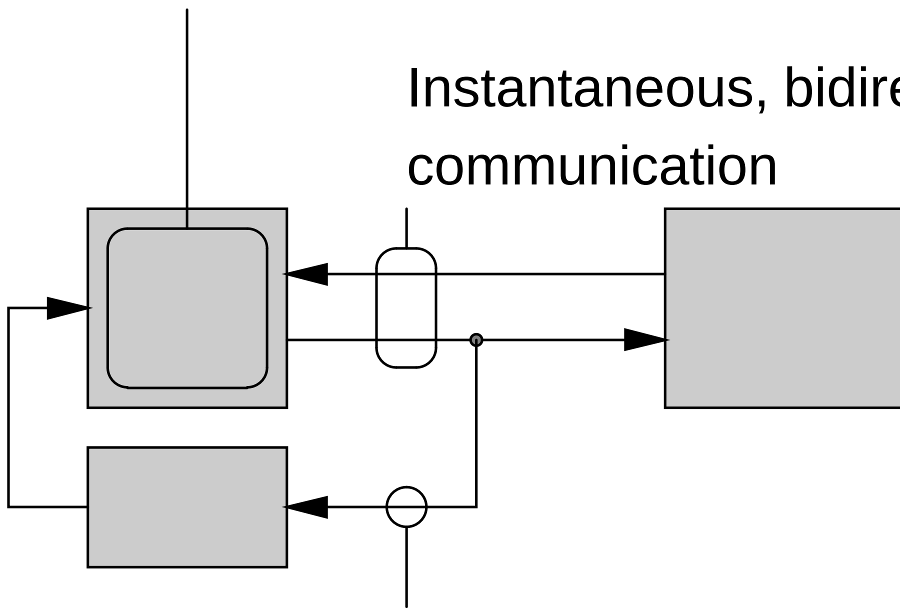
Controllers are reactive systems

- Maintain an ongoing dialog with their environment—listen, don't terminate
- *When* things happen as important as *what* happens
- Discrete-valued, time-varying
- Examples:
  - Systems with user interfaces
    - Digital watches
    - CD players
  - Real-time controllers
    - Anti-lock braking systems
    - Industrial process controllers

Our systems: Networks of concurrently executing modules communicating synchronously

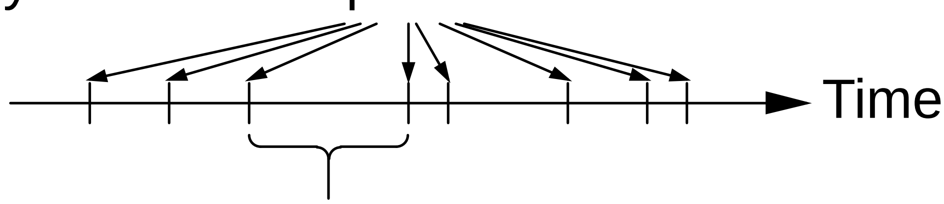
Opaque, zero-delay modules compute functions

Instantaneous, bidirectional communication



Single driver, multiple receiver “wires” (no buffering)

Every module computes once each instant



Nothing happens between instants

Outline

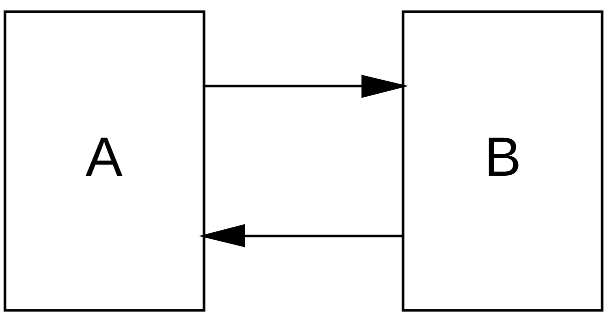
- Defining behavior in an instant:
  A fixed point

- Ensuring determinism:
  Monotonic module functions

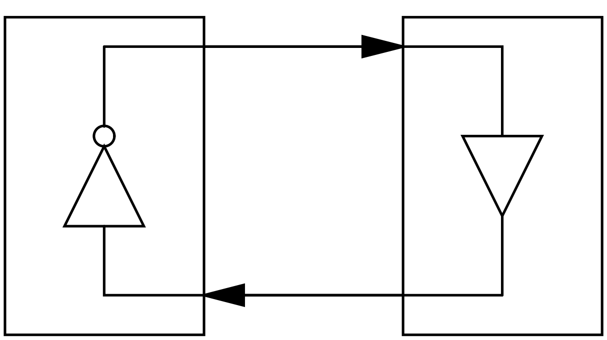
- Efficient software implementation:
  Iterating to a fixed point

Zero delay, Determinism, Heterogeneity, and Cycles together: A challenge.

Most schemes relax one of these requirements.



Which goes first?
Need an order-invariant semantics



Contradictory!
Need to attach meaning to such systems without *looking inside modules*

Fixed-point semantics are natural for synchronous specifications with cycles

Why a fixed point?

Self-reference:
The essence of a cycle

$$f(x_t) = x_t$$

System function (composition of module functions)

Wire values at time t (zero delay)

fixed point $\Longleftrightarrow$ stable state

determinism $\Longleftrightarrow$ unique solution

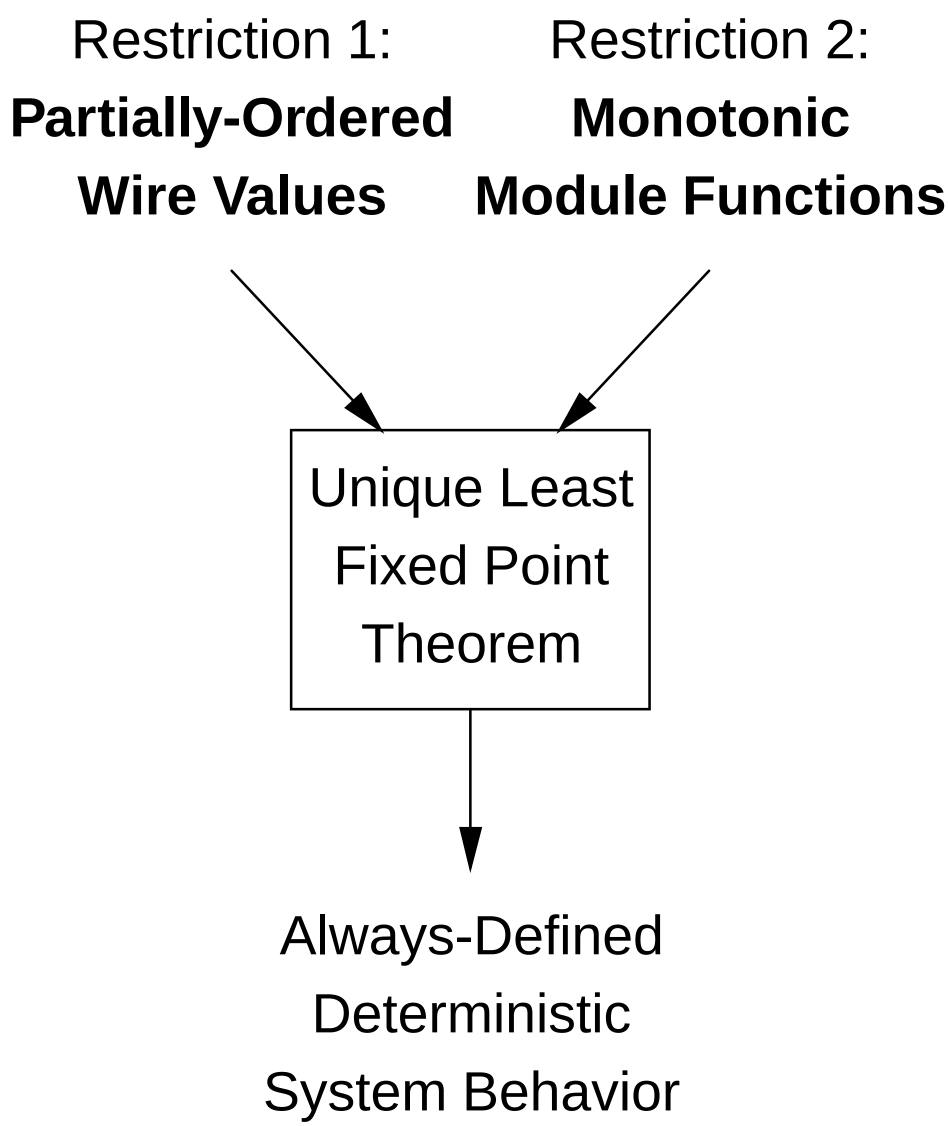
Outline

- Defining behavior in an instant:
  A fixed point

- Ensuring determinism:
  Monotonic module functions

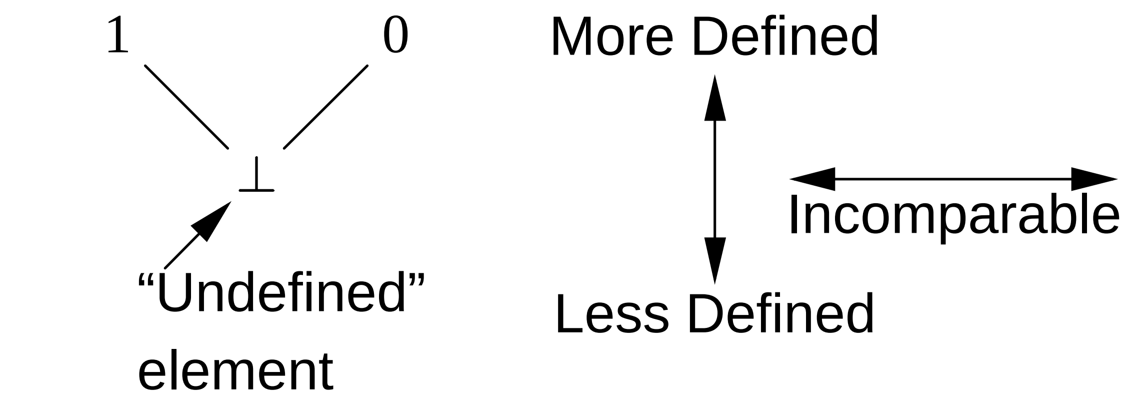
- Efficient software implementation:
  Iterating to a fixed point

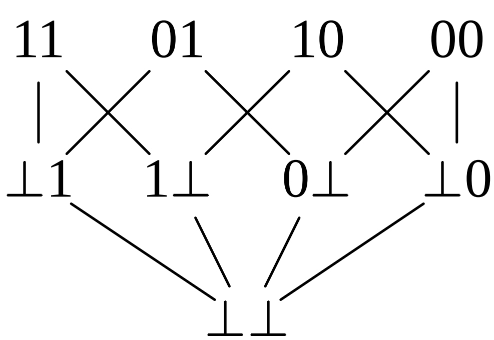
Two restrictions make these systems deterministic



Restriction 1: Partially ordered wire values

Values along an upward path grow more defined.



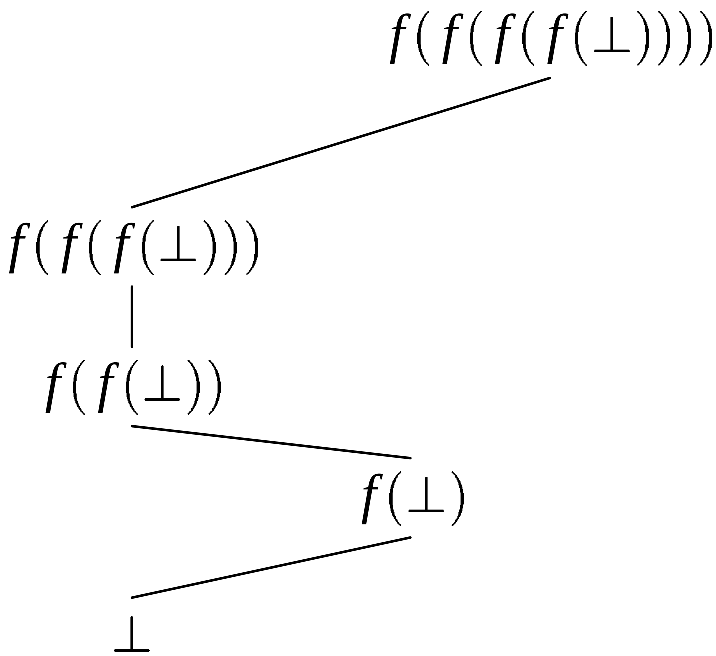


vector-valued extension

Formally, $x \sqsubseteq y$ if y is at least as defined as x.

Restriction 2: Monotonic module functions

A monotonic function never gives a less defined or incomparable result.



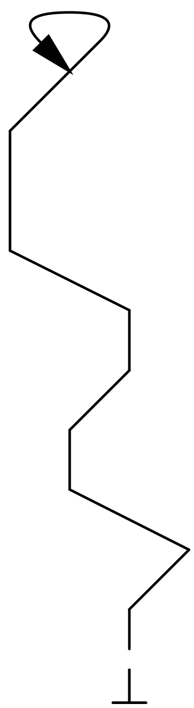
Formally, $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

Closed under composition: if $f(x)$ and $g(x)$ are monotonic, then $f(g(x))$ is.

Implication: Composing monotonic functions builds a monotonic network.

The least fixed point theorem ensures determinism

Well-known theorem: A monotonic function on a partial order has a unique least fixed point.



Behavior in an instant: The least fixed point of the (monotonic) system function

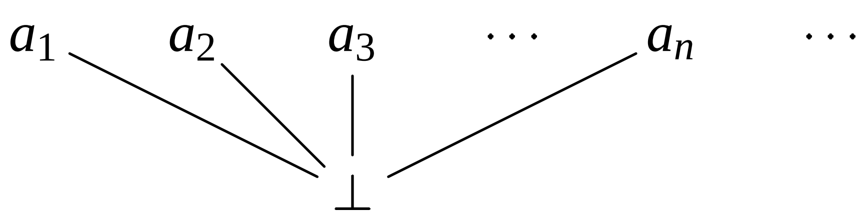
Implications:

- unique
- always defined
- quickly computed
- heterogeneous
  (only need monotonicity)

Meeting the conditions for determinism is easy

- **Partially-ordered wire values**

  Any set $\{a_1, a_2, \ldots, a_n, \ldots\}$ can easily be "lifted" to give a flat partial order:



- **Monotonic module functions**

  Ways to ensure monotonicity:

  – Strict functions are monotonic

  – Most functions in "X-valued simulation" are monotonic

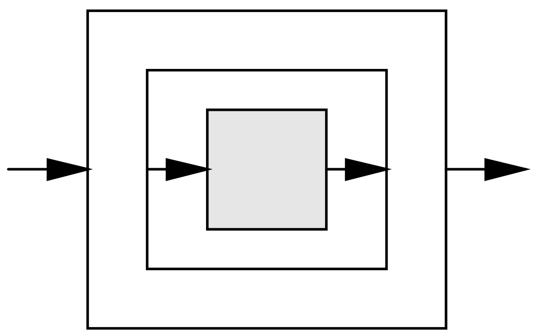
  – The composition of monotonic functions is monotonic

Many languages use strict functions, which are monotonic

A strict function:

$$g(\underbrace{\ldots,\bot,\ldots}_{\text{input wires}}) = (\underbrace{\bot,\ldots,\bot}_{\text{output wires}})$$

Outside:
A strict monotonic function



Inside:
Simple “function call” semantics

Common languages with strict functions:

- C/C++
- Synchronous Dataflow (SDF)

Danger: *Cycles of strict functions deadlock—fixed point is all ⊥—need some non-strict functions.*

Outline

- Defining behavior in an instant:
  A fixed point

- Ensuring determinism:
  Monotonic module functions

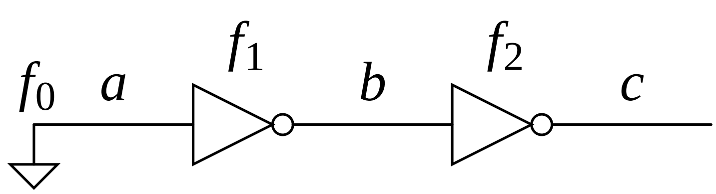
- Efficient software implementation:
  Iterating to a fixed point

The fixed point theorem suggests a simulation algorithm

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq \mathsf{LFP} = \mathsf{LFP} = \cdots$$

For each instant,

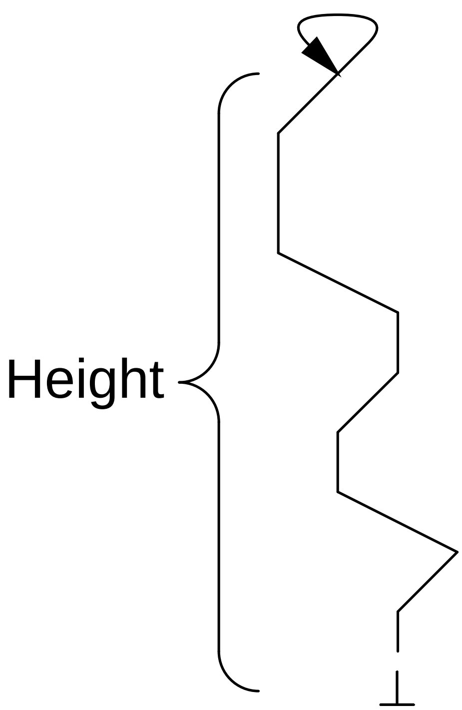
1. Start with all wires at $\bot$
2. Evaluate all module functions (in some order)
3. If any change their outputs, repeat Step 2



$$\begin{aligned}
(a,b,c) &= (\bot,\bot,\bot) \\
f_0(\bot,\bot,\bot) &= (0,\bot,\bot) \\
f_1(0,\bot,\bot) &= (0,1,\bot) \\
f_2(0,1,\bot) &= (0,1,0) \\
f_2(f_1(f_0(0,1,0))) &= (0,1,0)
\end{aligned}$$

Iterating to a fixed point is efficient and predictable

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq \text{LFP} = \text{LFP} = \cdots$$



A simple bound:

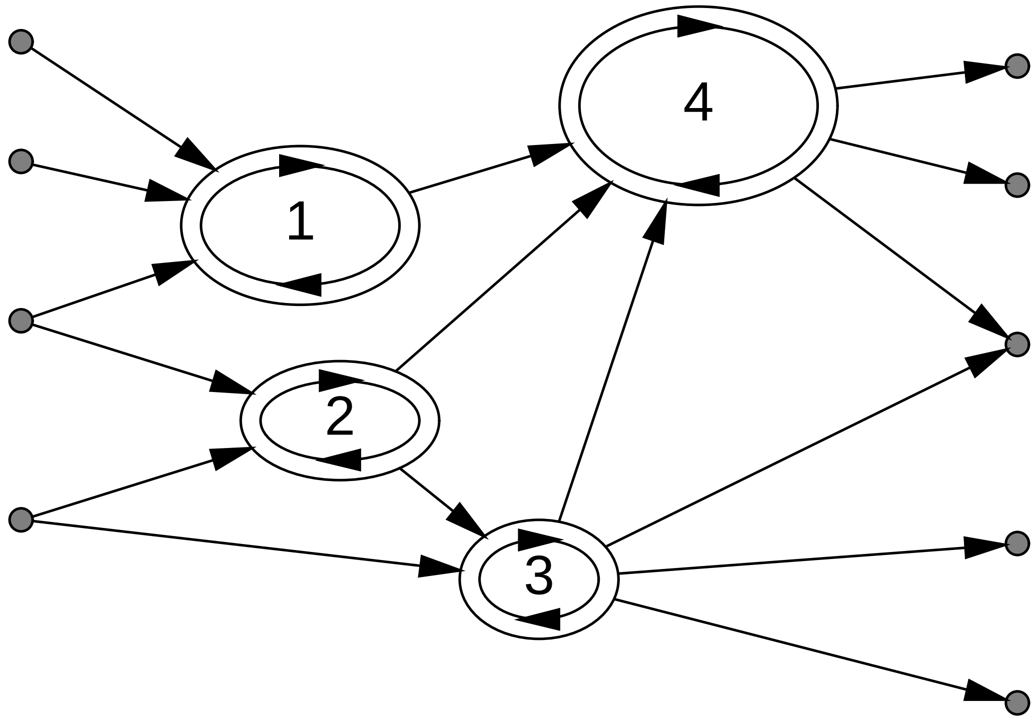
- Height is linear in the number of wires
- Each module evaluated once per step

$O(W \cdot M)$ module evaluations per instant

Can be scheduled statically: module evaluation order fixed at compile-time.

No wire tests required: just make iterations = height.

Many optimizations are possible



- Evaluate strongly-connected components in a topological order
- Form reactive clusters and bypass idle ones
- Cache the more-expensive-to-compute functions

Summary

- A way to specify synchronous controllers heterogeneously

  Synchronous = Zero Delay
  Heterogeneous = Modules are Opaque

- Behavior defined as a fixed point

  Fixed points natural for describing cycles

- Determinism through monotonic functions on partial orders

  Least fixed point theorem ensures unique behavior always defined

- Iterating to a fixed point efficient and predictable

  Statically schedulable
  $O(W \cdot M)$ *worst-case execution time*