CONCURRENT MODELS OF COMPUTATION FOR EMBEDDED SOFTWARE

by

Edward A. Lee

Memorandum No. UCB/ERL M05/2

22 December 2004

CONCURRENT MODELS OF COMPUTATION FOR EMBEDDED SOFTWARE

by

Edward A. Lee

Memorandum No. UCB/ERL M05/2

22 December 2004

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Concurrent Models of Computation for Embedded Software

Edward A. Lee

Department of
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720, USA

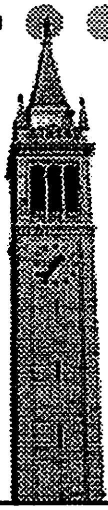
UCB ERL Technical Memorandum UCB/ERL M05/2
December 22, 2004

This document collects the lecture notes that I used when teaching EECS 290n in the Fall of 2004. This course is an advanced graduate course with a nominal title of *Advanced Topics in Systems Theory*. This instance of the course studies models of computation used for the specification and modeling of concurrent real-time systems, particularly those with relevance to embedded software. Current research and industrial approaches are considered, including real-time operating systems, process networks, synchronous languages (such as used in SCADE, Esterel, and Statecharts), timed models (such as used in Simulink, Giotto, VHDL, and Verilog), and dataflow models (such as a used in Labview and SPW). The course combines an experimental approach with a study of formal semantics. The objective is to develop a deep understanding of the wealth of alternative approaches to managing concurrency and time in software.

The experimental portion of the course uses Ptolemy II as the software laboratory. The formal semantics portion of the course builds on the mathematics of partially ordered sets, particularly as applied to prefix orders and Scott orders. It develops a framework for models of computation for concurrent systems that uses partially ordered tags associated with events. Discrete-event models, synchronous/reactive languages, dataflow models, and process networks are studied in this context. Basic issues of computability, boundedness, determinacy, liveness, and the modeling of time are studied. Classes of functions over partial orders, including continuous, monotonic, stable, and sequential functions are considered, as are semantics based on fixed-point theorems.

More details about this course can be found on its website:
`http://embedded.eecs.berkeley.edu/concurrency`

Contents



Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall, 2004

Lecture 1: Current Trends in Embedded Software

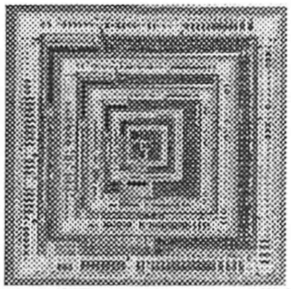
Are Resource Limitations the Key Defining Factor for Embedded Software?

- small memory
- small data word sizes
- relatively slow clocks

To deal with these problems, emphasize efficiency:
- write software at a low level (in assembly code or C)
- avoid operating systems with a rich suite of services
- develop specialized computer architectures
  - programmable DSPs
  - network processors

This is how embedded SW has been designed for 25 years

Why hasn't Moore's law changed all this in 25 years?



Hints that Embedded SW Differs Fundamentally from General Purpose SW

- object-oriented techniques are rarely used
  - classes and inheritance
  - dynamic binding
- processors avoid memory hierarchy
  - virtual memory
  - dynamically managed caches
- memory management is avoided
  - allocation/de-allocation
  - garbage collection

To be fair, there are some applications: e.g. Java in cell phones, but mainly providing the services akin to general purpose software.

More Hints: Fundamentally Different Techniques Applied to Embedded SW.

method-call models of computation

nesC/TinyOS
- developed for programming very small programmable sensor nodes called "motes"

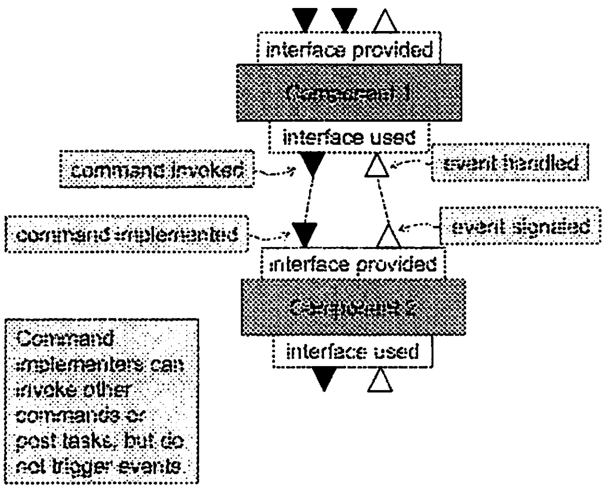
Click
- created to support the design of software-based network routers

actor-oriented models of computation

Simulink with Real-Time Workshop
- created for embedded control software and widely used in the automotive industry

Lustre/SCADE
- created for safety-critical embedded software (e.g. avionics software)

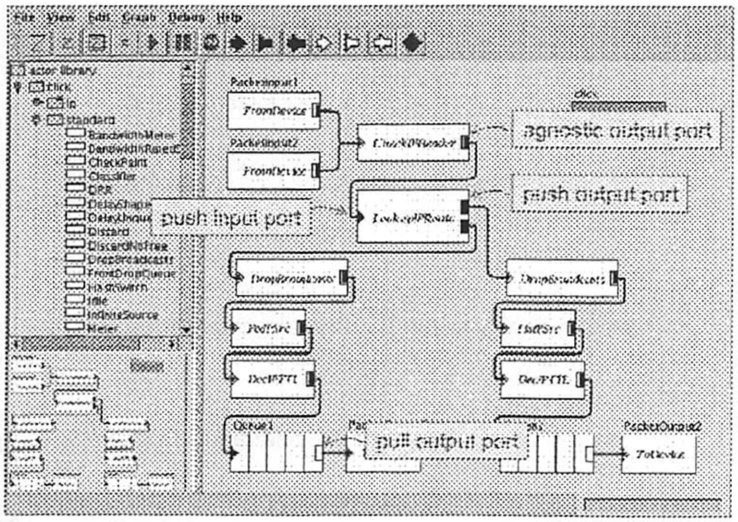
Alternative Concurrency Models: First example: nesC and TinyOS



Command implementers can invoke other commands or post tasks, but do not trigger events.

Typical usage pattern:
- hardware interrupt signals an event.
- event handler posts a task.
- tasks are executed when machine is idle.
- tasks execute atomically w.r.t. one another.
- tasks can invoke commands and signal events.
- hardware interrupts can interrupt tasks.
- exactly one monitor, implemented by disabling interrupts.

Alternative Concurrency Models: Second example: Click



Typical usage pattern:

- queues have push input, pull output.
- schedulers have pull input, push output.
- thin wrappers for hardware have push output or pull input only.

Implementation of Click with a visual syntax in Mescal (Keutzer, et al)

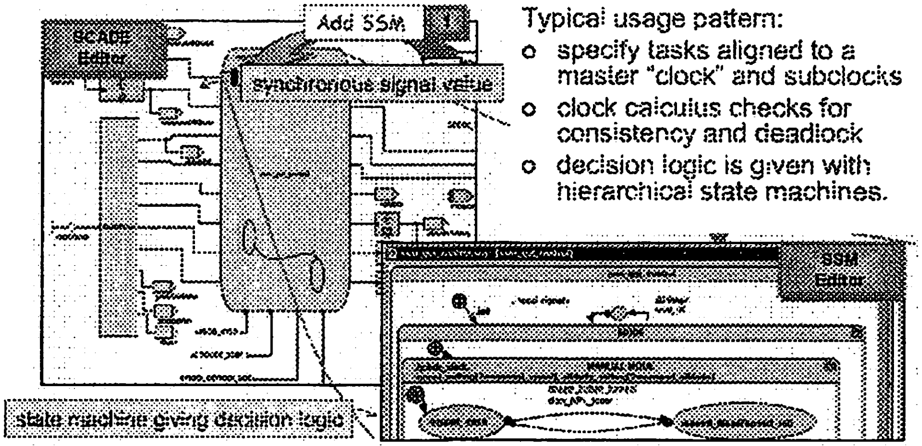
Observations about nesC/TinyOS & Click

- Very low overhead
- Bounded stack sizes
- No (unintended) race conditions
- No threads or processes
- Access to timers
- Can create thin wrappers around hardware

But rather specialized

- Unfamiliar to programmers
- No preemption (tasks must be decomposed)

Alternative Concurrency Models: Third example: Lustre/SCADE



Typical usage pattern:
- specify tasks aligned to a master "clock" and subclocks
- clock calculus checks for consistency and deadlock
- decision logic is given with hierarchical state machines.

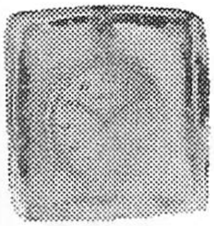
from http://www.esterel-technologies.com/

Observations about Lustre/SCADE

- Very low overhead
- Bounded stack sizes
- No (unintended) race conditions
- No threads or processes
- Verifiable (finite) behavior
- Certified compiler (for use in avionics)

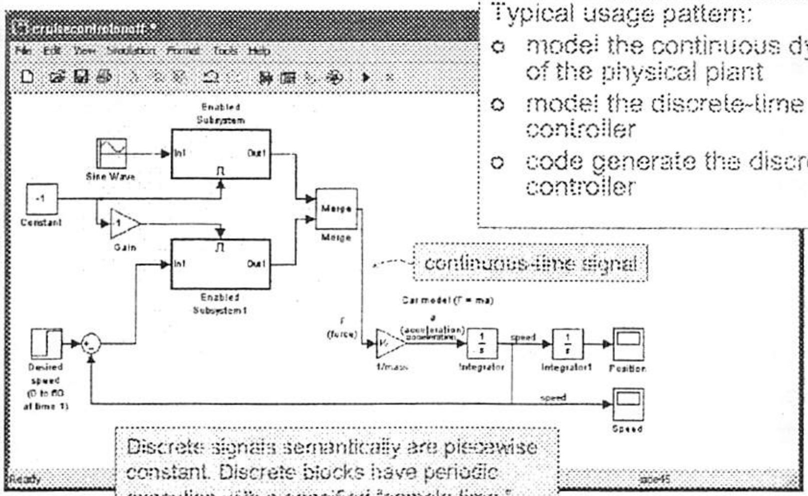
But rather specialized
  - Unfamiliar to programmers
  - No preemption
  - No time

The Real-Time Problem



- Programming languages have no time in their core semantics
- Temporal properties are viewed as "non-functional"
- Precise timing is poorly supported by hardware architectures
- Operating systems provide timed behavior on a best-effort basis (e.g. using priorities).
- Priorities are widely misused in practice

Alternative Concurrency Models: Fourth example: Simulink



Typical usage pattern:

- model the continuous dynamics of the physical plant
- model the discrete-time controller
- code generate the discrete-time controller

continuous-time signal

Discrete signals semantically are piecewise constant. Discrete blocks have periodic execution with a specified "sample time."

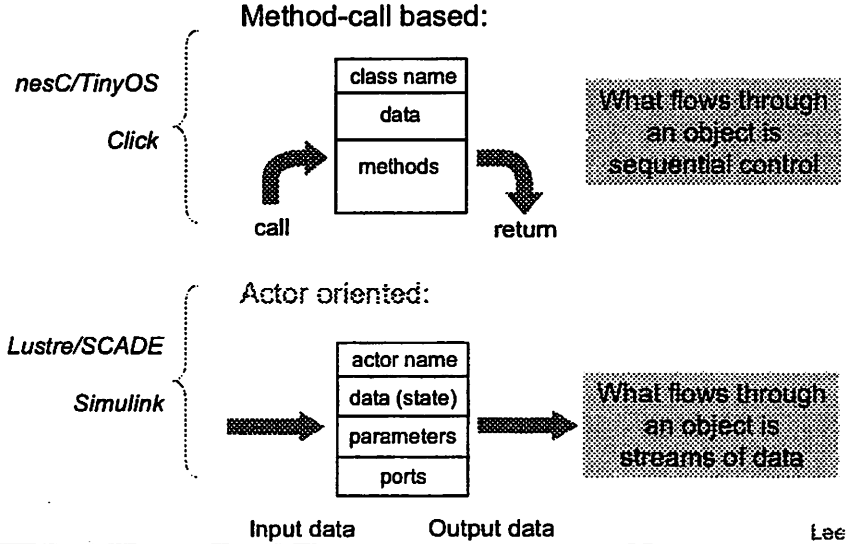
Observations about Simulink

- Bounded stack sizes
- Deterministic (no race conditions)
- Timing behavior is explicitly given
- Efficient code generator (for periodic discrete-time)
- Supports concurrent tasks
- No threads or processes visible to the programmer
  - But cleverly leverages threads in an underlying O/S.

But rather specialized

- Periodic execution of all blocks
- Accurate schedulability analysis is difficult

Two Distinct Component Interaction Mechanisms



Method-call based:

nesC/TinyOS

Click

What flows through an object is sequential control

Actor oriented:

Lustre/SCADE

Simulink

What flows through an object is streams of data

Terminology Problem

Of these, only nesC is recognized as a "programming language."

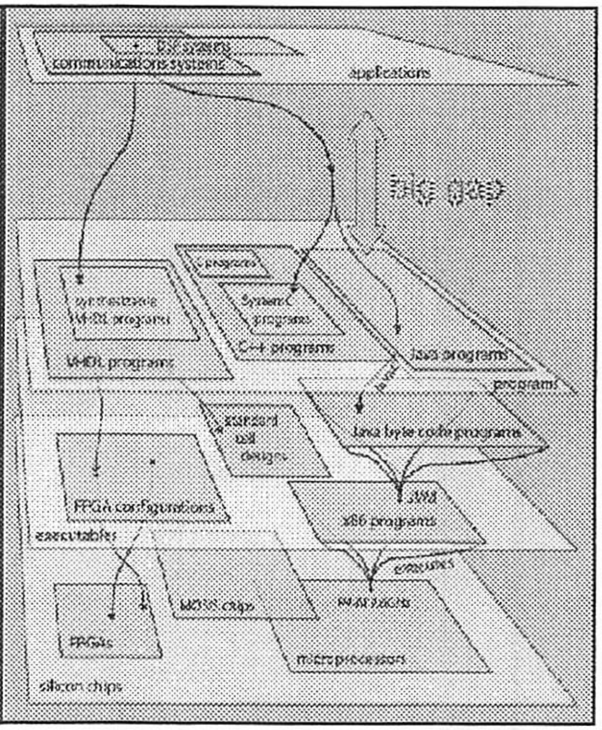
I will call them "platforms"

- A platform is a set of possible designs:
  - the set of all nesC/TinyOS programs
  - the set of all Click configurations
  - the set of all SCADE designs
  - the set of all Simulink block diagrams

Platforms

A *platform* is a set of designs.

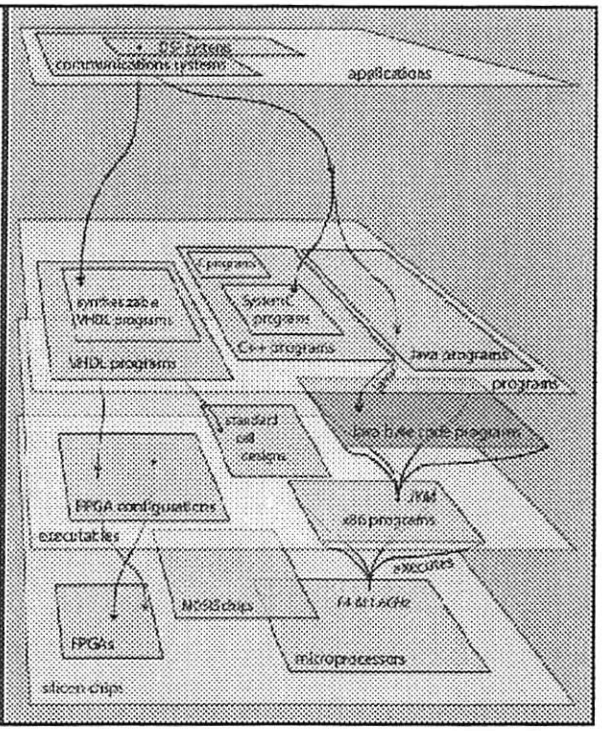
Relations between platforms represent design processes.



Progress

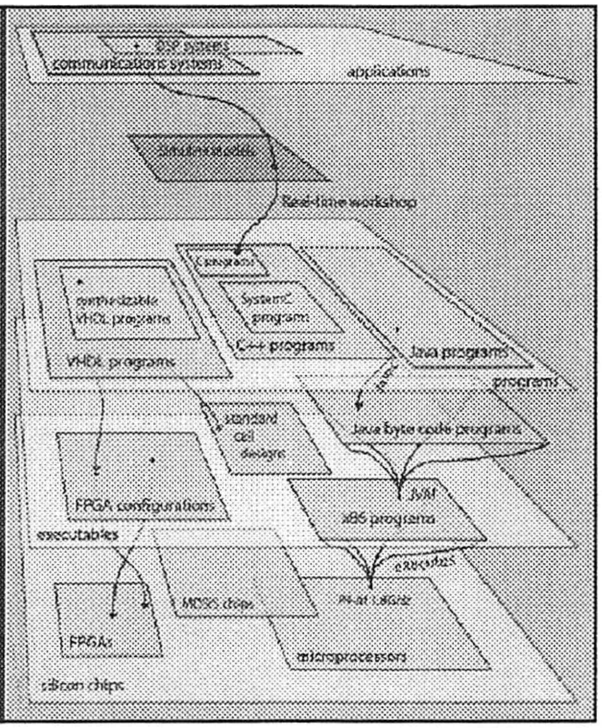
Many useful technical developments amount to creation of new platforms.

- o microarchitectures
- o operating systems
- o virtual machines
- o processor cores
- o configurable ISAs



Better Platforms

Platforms with modeling properties that reflect requirements of the application, not accidental properties of the implementation.



How to View This Design

From above: Signal flow graph with linear, time-invariant components.

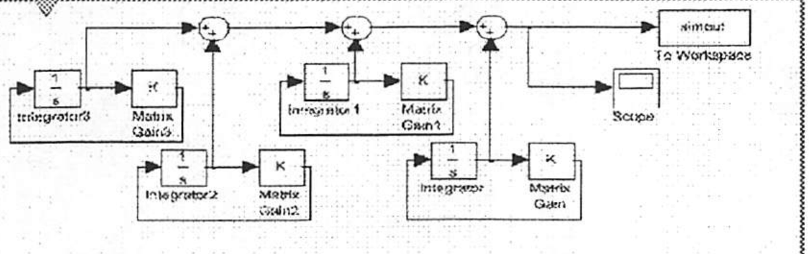


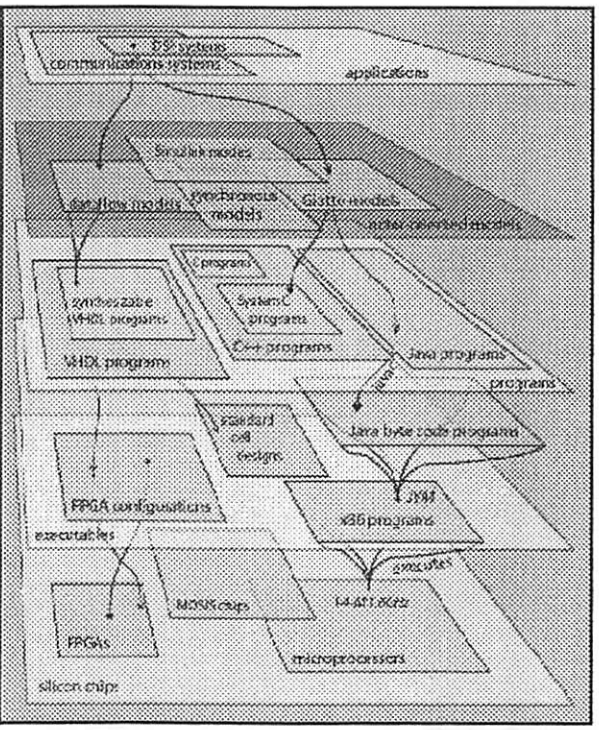
Figure C.12: A block diagram generating a plucked string sound with a fundmental and three harmonics.

From below: Synchronous concurrent composition of components

Actor-Oriented Platforms

Actor oriented models compose concurrent components according to a model of computation.

Time and concurrency become key parts of the programming model.



How Many More (Useful) Models of Computation Are There?

Here are a few actor-oriented platforms:

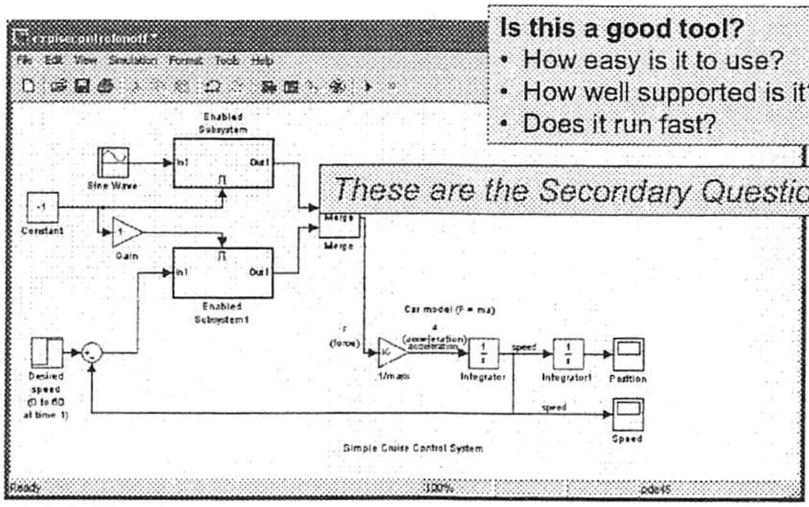
- Labview (synchronous dataflow)
- Modelica (continuous-time, constraint-based)
- CORBA event service (distributed push-pull)
- SPW (synchronous dataflow)
- OPNET (discrete events)
- VHDL, Verilog (discrete events)
- SDL (process networks)
- …

Many Variants –
Consider Dataflow Alone:

- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980's]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- …

Many tools, software frameworks, and hardware architectures have been built to support one or more of these.

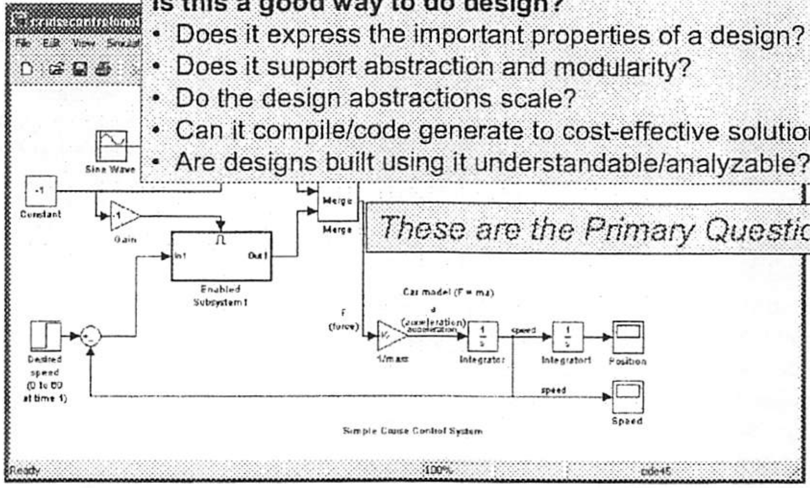
How to Choose a Platform: Tools Focus



Is this a good tool?
- How easy is it to use?
- How well supported is it?
- Does it run fast?

These are the Secondary Questions!

How to Choose a Platform: Abstraction Focus



Is this a good way to do design?
- Does it express the important properties of a design?
- Does it support abstraction and modularity?
- Do the design abstractions scale?
- Can it compile/code generate to cost-effective solutions?
- Are designs built using it understandable/analyzable?

These are the Primary Questions!

The Meta Question

How can we objectively evaluate the alternatives?

Meta Platforms
Supporting Multiple Models of Computation

- Ptolemy Classic and Ptolemy II (UC Berkeley)
- GME (Vanderbilt)
- Metropolis (UC Berkeley)
- ROOM (Rational)
- SystemC (Synopsys and others)

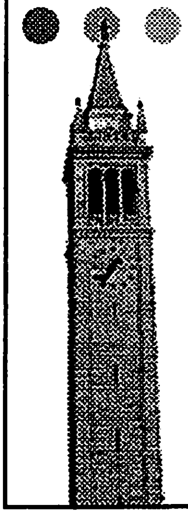
To varying degrees, each of these provides an *abstract semantics* that gets specialized to deliver a particular model of computation.

ROOM is evolving into an OMG standard (composite structures in UML 2)

Conclusion

- Embedded software is an immature technology
- Focus on "platforms" not "languages"
- Platforms have to:
  - **expose hardware (with thin wrappers)**
  - **embrace time in the core semantics**
  - **embrace concurrency in the core semantics**
- API's over standard SW methods won't do
- Ask about the "abstractions" not the "tools"

Many questions remain...



Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall, 2004

Lecture 2: Threads

Prevailing Software Practice

- Processes for concurrent execution of multiple apps
  - **Processes interact through files, pipes, sockets**
- Threads for concurrency within an application
  - **Threads share memory, processes do not**
- Remote procedure calls (RPC) for distributed apps
  - **Assumes reliable communication**
- Middleware (e.g. CORBA) built on top of RPC
  - **Inherits requirement for reliable communication**
- Real-time operating systems (RTOS): thread scheduling
  - **Priority tweaking and bench testing**

Problems with Threads:
Example: Simple Observer Pattern

```
public void addListener(listener) {...}

public void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

What's wrong with this?

Thanks to Mark S. Miller, HP Labs, for the details of this example.

Example: Simple Observer Pattern
With Mutual Exclusion (Mutexes) using Monitors

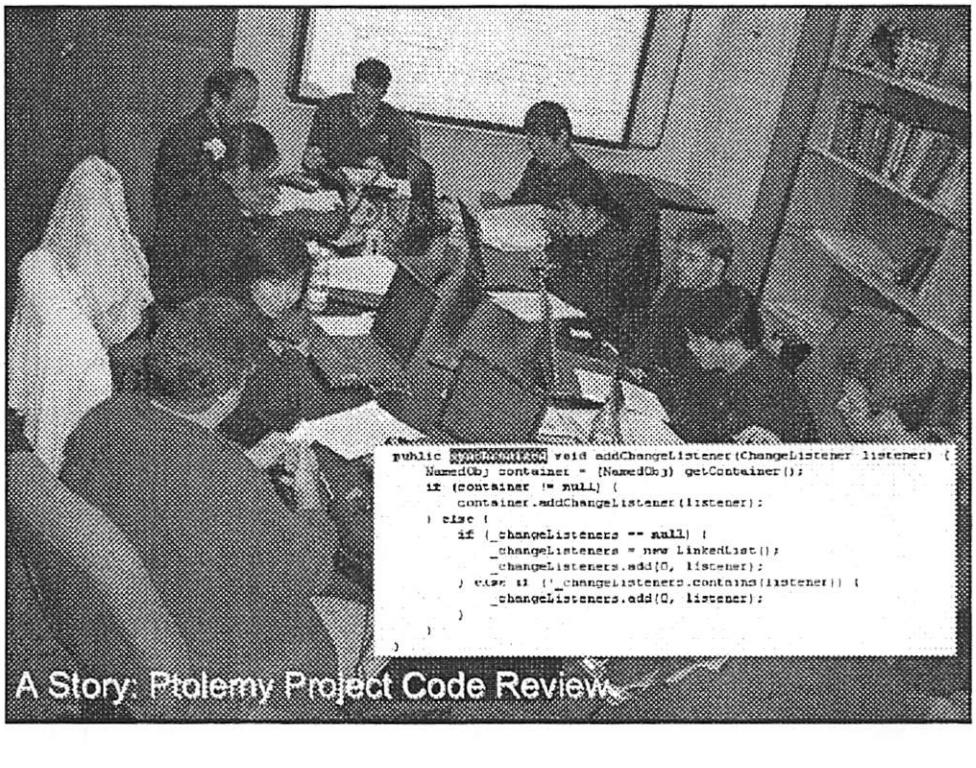
```
public synchronized void addListener(listener) {...}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

Javasoft recommends against this.
What's wrong with it?

Mutexes using Monitors are Minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(), deadlock!



```
public synchronized void addChangeListener(ChangeListener listener) {
    NamedObj container = (NamedObj) getContainer();
    if (container != null) {
        container.addChangeListener(listener);
    } else {
        if (_changeListeners == null) {
            _changeListeners = new LinkedList();
            _changeListeners.add(0, listener);
        } else if (!_changeListeners.contains(listener)) {
            _changeListeners.add(0, listener);
        }
    }
}
```

A Story: Ptolemy Project Code Review

Ptolemy Project Code Review
A Typical Story

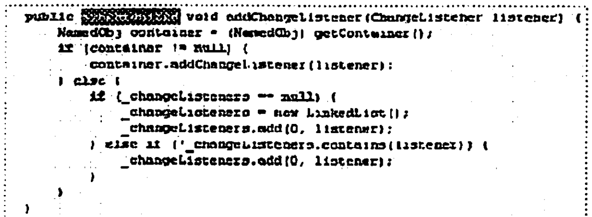
Code review discovers that a method needs to be synchronized to ensure that multiple threads do not reverse each other's actions.

No problems had been detected in 4 years of using the code.

Three days after making the change, users started reporting deadlocks caused by the new mutex.

Analysis and correction of the deadlock is hard.

But code review successfully identified the flaw.



```
public synchronized void addChangeListener(ChangeListener listener) {
    NamedObj container = (NamedObj) getContainer();
    if (container != null) {
        container.addChangeListener(listener);
    } else {
        if (_changeListeners == null) {
            _changeListeners = new LinkedList();
            _changeListeners.add(0, listener);
        } else if (!_changeListeners.contains(listener)) {
            _changeListeners.add(0, listener);
        }
    }
}
```

Simple Observer Pattern Becomes
Not So Simple

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make copy of listeners to avoid race conditions

notify each listener <u>outside</u> of synchronized block to avoid deadlock

This still isn't perfect.
What's wrong with it?

Simple Observer Pattern:
Is it Even Possible to Make It Right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value changes in the wrong order!

A Stake in the Ground

Nontrivial concurrent programs based on threads and mutexes are incomprehensible to humans.

- No amount of process improvement will help
    - the human brain doesn't work this way
- Formal methods may help
    - scalability?
    - understandability?
- Better concurrency abstractions will help more

Diagnosing What's Wrong With Threads: Some Notation

Set: $S = \{a, b, c, \dots\}$
Natural numbers: $N = \{1, 2, 3, \dots\}$
Counting set: $N_M = \{1, 2, \dots, M\}$
Nonnegative integers: $N_+ = \{0, 1, 2, 3, \dots\}$
Function: $f : S \to S'$ (Domain: S Codomain: S')
Finite sequence: $s : N_M \to S, \quad M \in N$
Infinite sequence: $s : N \to S$
Set of functions: $F = [S \to S']$
Set of finite sequences: $S^* = [\, N_M \to S, \;\; M \in N \,]$
Set of finite and infinite sequences: $S^{**} = [\, N \to S \,] \cup S^*$

A Model of Threads

Binary digits: $B = \{0, 1\}$
State space: B^{**}
Instruction (atomic action): $a : B^{**} \to B^{**}$
Instruction (action) set: $A \subset [B^{**} \to B^{**}]$
Thread (non-terminating): $t : N \to A$
Thread (terminating): $t : \{1, \dots, n\} \to A, \quad n \in N$

A thread is a sequence of atomic actions.

Programs

A program is a finite representation of a family of threads (one for each initial state b_0).

Machine control flow: $c : B^{**} \rightarrow N_+$ (e.g. program counter) where $c(b) = 0$ is interpreted as a "stop" command.

Let m be the program length. Then a program is:

$$p : \{1, \dots , m\} \rightarrow A$$

A program is an ordered sequence of m instructions.

Execution (Operational Semantics)

Given initial state $b_0 \in B^{**}$, then execution is:

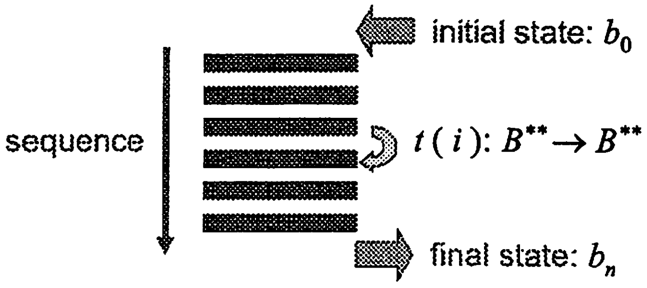
$$b_1 = p(c(b_0))(b_0) \quad = t(1)(b_0)$$
$$b_2 = p(c(b_1))(b_1) \quad = t(2)(b_1)$$
$$\dots$$
$$b_n = p(c(b_{n-1}))(b_{n-1}) \quad = t(n)(b_{n-1})$$
$$c(b_n) = 0$$

Execution defines a *partial function* (defined on a subset of the domain) from the initial state to final state:

$$e_p : B^{**} \rightarrow B^{**}$$

This function is undefined if the thread does not terminate.

Threads as Sequences of State Changes



initial state: b_0

sequence

$t(i): B^{**} \to B^{**}$

final state: b_n

- Time is irrelevant
- All actions are ordered
- The thread sequence depends on the program and the state

Expressiveness

Given a finite action set: $A \subset [B^{**} \to B^{**}]$
Execution: $e_p \in [B^{**} \to B^{**}]$

Can all functions in $[B^{**} \to B^{**}]$ be defined by a program?

Compare the cardinality of the two sets:
set of functions: $[B^{**} \to B^{**}]$
set of programs: $[\{1, \ldots, m\} \to A,\ m \in N]$

Programs Cannot Define All Functions

Cardinality of this set: $[\{1, \dots, m\} \to A,\ m \in N]$ is the same as the cardinality of the set of integers (put the elements of the set into a one-to-one correspondence with the integers). The set is countable.

This set is larger: $[B^{**} \to B^{**}]$.

Proof: Choose the subset of *constant functions*,

$$C \subset [B^{**} \to B^{**}]$$

This set is not countable (use Cantor's diagonal argument to show this).

Simpler: Choose a Smaller State Space

Smaller state space (natural numbers): $N = \{1, 2, 3, \dots\}$

Set of all functions: $F = [N \to N]$

Finite action set: $A \subset [N \to N]$

Set of all programs: $[\{1, \dots, m\} \to A,\ m \in N]$

Again, the set of all functions is uncountable and the set of all programs is countable, so clearly not all functions can be given by programs.

With a "good" choice of action set, we get programs that implement a well-defined subset of functions.

Taxonomy of Functions

Functions from initial state to final state:

$$F = [N \to N]$$

Partial recursive functions:

$$PR \subset [N \to N]$$

(Those functions for which there is a program that terminates for zero or more initial states).

Total recursive functions:

$$TR \subset P \subset [N \to N]$$

(There is a program that terminates for all initial states).

Church's Thesis

Every function $f: N \to N$ that is computable by any practical computer is in PR.

There are many "good" choices of finite action sets that yield the same definition of PR.

Evidence that this set is fundamental is that Turing machines, lambda calculus, PCF (a basic recursive programming language), and all practical computer instruction sets yield the same set PR.

Key Results in Computation

Turing: Instruction set with 7 instructions is enough to write programs for all partial recursive functions.

- **A program using this instruction set is called a Turing machine**
- **A *universal Turing machine* is a Turing machine that can execute a binary encoding of any Turing machine.**

Church: Instructions are a small set of transformation rules on strings called the lambda calculus.

- **Equivalent to Turing machines.**

Turing Completeness

A *Turing complete* instruction set is a finite subset of PR (and probably of TR) whose transitive closure is PR.

Many choices of underlying instruction sets $A \subset [N \to N]$ are Turing complete and hence equivalent.

This can be generalized to the larger state space B^{**} by encoding the integers in it.

Equivalence

Any two programs that implement the same partial recursive function are equivalent.

- Terminate for the same initial states.
- End up in the same final states.

NOTE: Big problem for embedded software:

- All non-terminating programs are equivalent.
- All programs that terminate in the same "exception" state are equivalent.

Limitations of the 20-th Century Theory of Computation

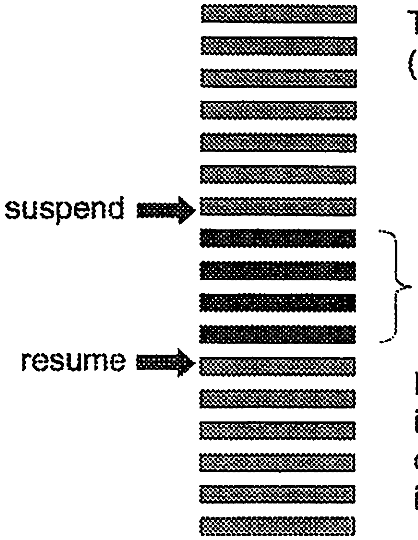
- Only terminating computations are handled.

This is not very useful…

But it gets even worse:

- There is no concurrency.

Concurrency: Interactions Between Threads



suspend

resume

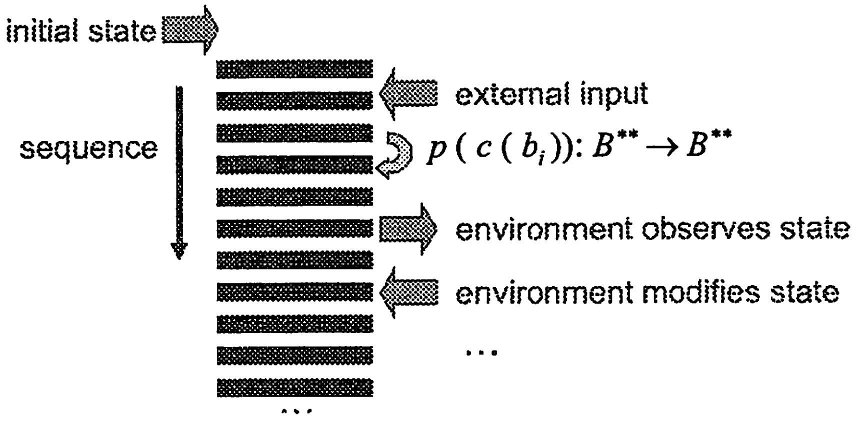
The operating system (typically) provides:

- suspend/resume
- mutual exclusion
- semaphores

another thread can change the state

Recall that for a thread, which instruction executes next depends on the state, and what it does depends on the state.

Nonterminating and/or Interacting Threads: Allow State to be Observed and Modified



initial state

sequence

external input

$p(c(b_i)): B^{**} \to B^{**}$

environment observes state

environment modifies state

. . .

. . .

Recall Execution of a Program

Given initial state $b_0 \in B^{**}$, then execution is:

$$
\begin{aligned}
b_1 &= p(c(b_0))(b_0) &&= t(1)(b_0) \\
b_2 &= p(c(b_1))(b_1) &&= t(2)(b_1) \\
&\dots \\
b_n &= p(c(b_{n-1}))(b_{n-1}) &&= t(n)(b_{n-1}) \\
c(b_n) &= 0
\end{aligned}
$$

When a thread executes alone, execution is a composition of functions:

$$t(n) \circ \ldots \circ t(2) \circ t(1)$$

Interleaved Threads

Consider two threads with functions:

$$t_1(1), t_1(2), \ldots, t_1(n)$$
$$t_2(1), t_2(2), \ldots, t_2(m)$$

These functions are arbitrarily interleaved.

Worse: The *i*-th action executed by the machine, if it comes from program $c(b_{i-1})$, is:

$$t(i) = p(c(b_{i-1}))$$

which depends on the state, which may be affected by the other thread.

Equivalence of Pairs of Programs

For concurrent programs p_1 and p_2 to be equivalent under threaded execution to programs p_1' and p_2', we need for each arbitrary interleaving of the thread functions produced by that interleaving to terminate and to compose to the same function as all other interleavings for both programs.

This is hopeless, except for trivial concurrent programs!

Equivalence of Individual Programs

If program p_1 is to be executed in a threaded environment, then without knowing what other programs will execute with it, there is no way to determine whether it is equivalent to program p_1' except to require the programs to be identical.

This makes threading nearly useless, since it makes it impossible to reason about programs.

Determinacy

For concurrent programs p_1 and p_2 to be *determinate* under threaded execution we need for each arbitrary interleaving of the thread functions produced by that interleaving to terminate and to compose to the same function as all other interleavings.

This is again hopeless, except for trivial concurrent programs!

Moreover, without knowing what other programs will execute with it, we cannot determine whether a given program is determinate.

Manifestations of Problems

- Race conditions
  - Two threads modify the same portion of the state. Which one gets there first?
- Consistency
  - A data structure with interdependent data is updated in multiple atomic actions. Between these actions, the state is inconsistent.
- Deadlock
  - Fixes to the above two problems result in threads waiting for each other to complete an action that they will never complete.

Improving the Utility of the Thread Model

Brute force methods for making threads useful:

- Segmented memory (processes)
  - Pipes and file systems provide mechanisms for sharing data.
  - Implementation of these requires a thread model, but this implementation is done by operating system expert, not by application programmers.
- Functions (no side effects)
  - Disciplined programming design pattern, or…
  - Functional languages (like Concurrent ML)
- Single assignment of variables
  - Avoids race conditions

Mechanisms for Achieving Determinacy

Less brute force (but also weaker):

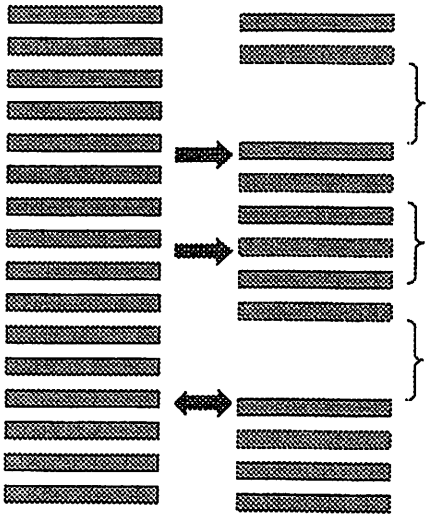
- Semaphores
- Mutual exclusion locks (*mutexes*, *monitors*)
- Rendezvous

All require an atomic test-and-set operation, which is not in the Turing machine instruction set.

Mechanisms for Interacting Threads

Potential for race conditions, inconsistency, and deadlock severely compromise software reliability.

These methods date back to the 1960's (Dijkstra).



semaphore or monitor used to stall a thread

race condition

rendezvous is more symmetric use of semaphores

Deadlock

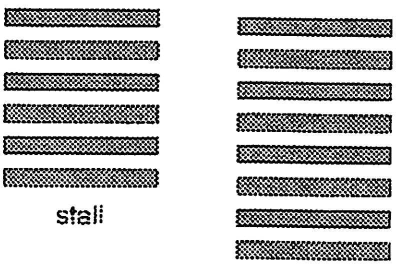
"Acquire lock *x*" means the following atomic action:

if *x* is false, set it to true,
else stall until it is false.

where *x* is Boolean variable (a "semaphore").

"Release lock *x*" means:

set *x* to false.



Simple Rule for Avoiding Deadlock [Lea]

“Always acquire locks in the same order.”

However, this is very difficult to apply in practice:

- Method signatures do not indicate what locks they grab (so you need access to all the source code of methods you use).
- Symmetric accesses (where either thread can initiate an interaction) become more difficult.

Deadlock Risk can Lurk for Years in Code

```
/**
CrossRefList is a list that maintains pointers to other CrossRefLists.
-
@author Geroncio Galicia, Contributor: Edward A. Lee
@version $Id: CrossRefList.java,v 1.78 2004/04/29 14:50:00 eal Exp $
@since Ptolemy II 0.2
@Pt.ProposedRating Green (eal)
@Pt.AcceptedRating Green (bart)
*/
public final class CrossRefList implements Serializable  {
    ..
    protected class CrossRef implements Serializable{
        ..
        // NOTE: It is essential that this method not be
        // synchronized, since it is called by _farContainer(),
        // which is.  Having it synchronized can lead to
        // deadlock.  Fortunately, it is an atomic action,
        // so it need not be synchronized.
        private Object _nearContainer() {
            return _container;
        }

        private synchronized Object _farContainer() {
            if (_far != null) return _far._nearContainer();
            else return null;
        }
        -
    }
}
```

Code that had been in use for four years, central to Ptolemy II, with an extensive test suite, design reviewed to yellow, then code reviewed to green in 2000, causes a deadlock during a demo on April 26, 2004.

And Doubts Remain…

```
/**
CrossRefList is a list that maintains pointers to other CrossRefLists.
..
@author Geroncio Galicia, Contributor: Edward A. Lee
@version $Id: CrossRefList.java,v 1.78 2004/04/29 14:50:00 eal Exp $
@since Ptolemy II 0.2
@Pt.ProposedRating Green (eal)
@Pt.AcceptedRating Green (bart)
*/
public final class CrossRefList implements Serializable  {
    ..
    protected class CrossRef implements Serializable{
        ..
        private synchronized void _dissociate() {
            _unlink(); // Remove this
            // NOTE: Deadlock risk here!  If _far is waiting
            // on a lock to this CrossRef, then we will get
            // deadlock. However, this will only happen if
            // we have two threads simultaneously modifying a
            // model. At the moment (4/29/04), we have no
            // mechanism for doing that without first
            // acquiring write permission the workspace().
            // Two threads cannot simultaneously hold that
            // write access.
            if (_far != null) _far._unlink(); // Remove far
        }
    }
```

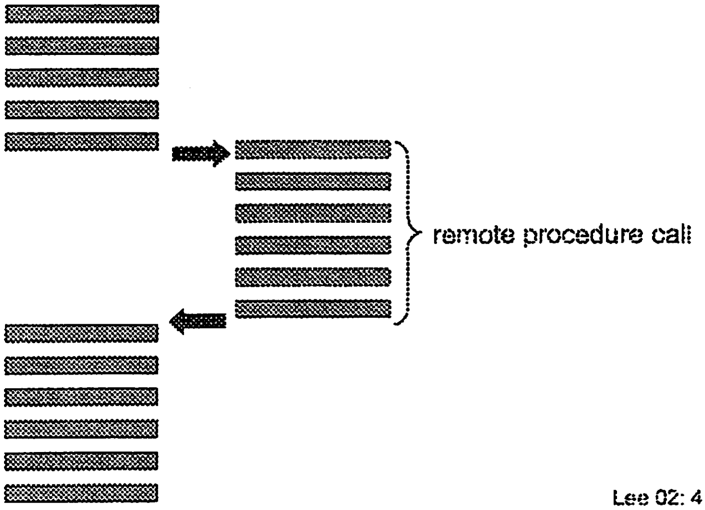
Safety of this code depends on policies maintained by entirely unconnected classes. The language and synchronization mechanisms provide no way to talk about these systemwide properties.

What it Feels Like to Use the *synchronized* Keyword in Java

Image "borrowed" from an Iomega advertisement for Y2K software and disk drives, *Scientific American*, September 1999.

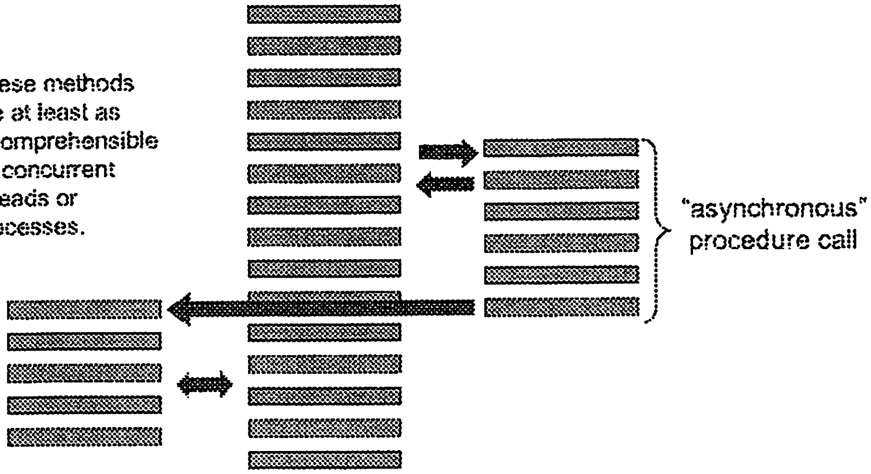
Distributed Computing: In Practice, Mostly Based on Remote Procedure Calls (RPC)

Force-fitting the sequential abstraction onto parallel hardware.



Combining Processes and RPC –
Split-Phase Execution, Futures, Asynchronous Method Calls, Callbacks, …

These methods are at least as incomprehensible as concurrent threads or processes.

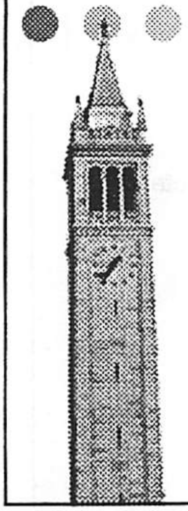


Summary

- Theory of computation supports well only
  - terminating
  - non-concurrent

  computation

- Threads are a poor concurrent model of computation
  - weak formal reasoning possibilities
  - incomprehensibility
  - race conditions
  - inconsistent state conditions
  - deadlock risk



Concurrent Models of Computation for Embedded Software

Edward A. Lee

Professor, UC Berkeley

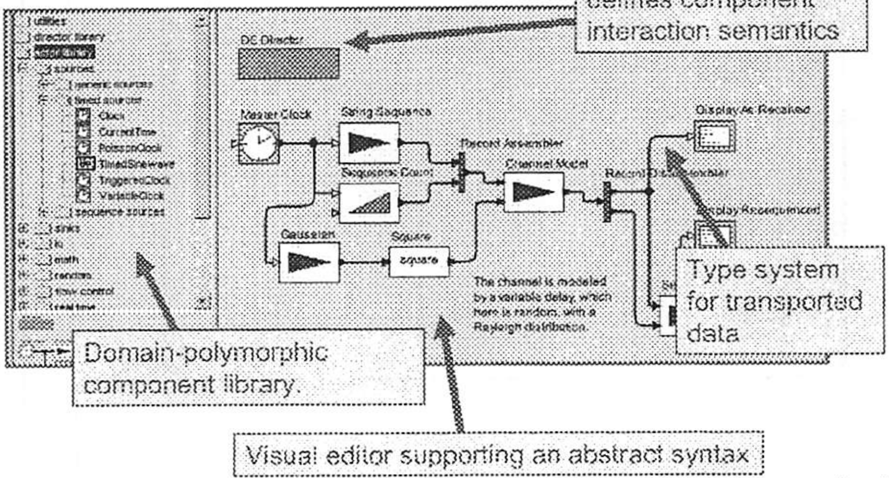
EECS 290n -- Advanced Topics in Systems Theory

Fall, 2004

Lecture 3: Overview of Actor-Oriented Models of Computation

Ptolemy II: Framework for Experimenting with Alternative Concurrent Models of Computation

Basic Ptolemy II infrastructure:



The channel is modeled by a variable delay, which here is random, with a Rayleigh distribution.

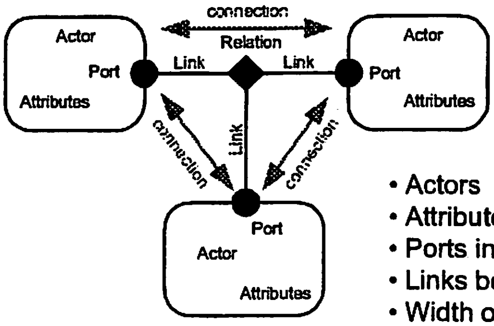
Director from a library defines component interaction semantics

Type system for transported data

Domain-polymorphic component library.

Visual editor supporting an abstract syntax

The Basic Abstract Syntax

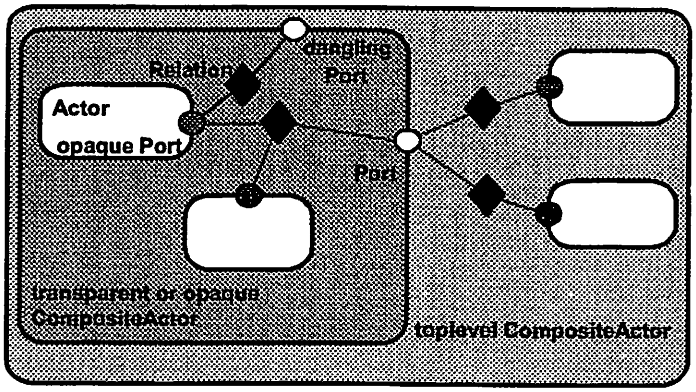


- Actors
- Attributes on actors (parameters)
- Ports in actors
- Links between ports
- Width on links (channels)
- Hierarchy

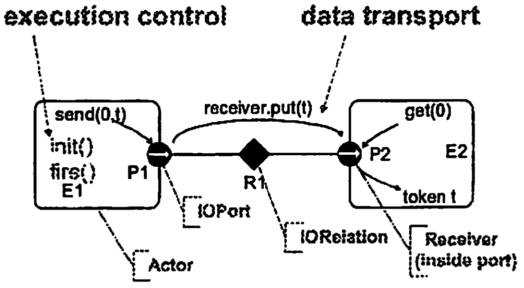
Concrete syntaxes:

- XML
- Visual pictures
- Actor languages (Cal, StreamIT, ...)

Hierarchy - Composite Components



Abstract Semantics
of *Actor-Oriented* Models of Computation

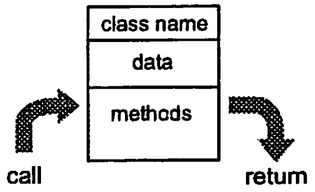


Actor-Oriented Models of Computation that we have implemented:

- dataflow (several variants)
- process networks
- distributed process networks
- Click (push/pull)
- continuous-time
- CSP (rendezvous)
- discrete events
- distributed discrete events
- synchronous/reactive
- time-driven (several variants)
- ...

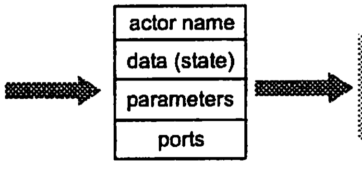
What is an *Actor-Oriented* MoC?

Traditional component interactions:



What flows through an object is sequential control

Actor oriented:



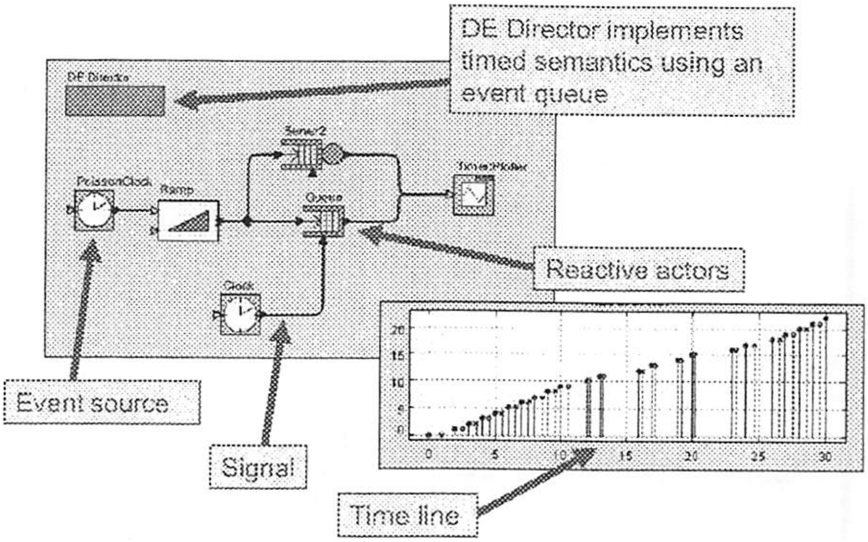
What flows through an object is streams of data

Models of Computation Implemented in Ptolemy II

- CI – Push/pull component interaction
- Click – Push/pull with method invocation
- CSP – concurrent threads with rendezvous
- CT – continuous-time modeling
- DE – discrete-event systems
- DDE – distributed discrete events
- FSM – finite state machines
- DT – discrete time (cycle driven)
- Giotto – synchronous periodic
- GR – 2-D and 3-D graphics
- PN – process networks
- DPN – distributed process networks
- SDF – synchronous dataflow
- SR – synchronous/reactive
- TM – timed multitasking

Most of these are actor oriented.

Discrete Event Models



DE Director implements timed semantics using an event queue

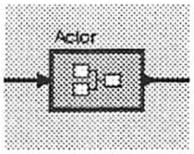
Reactive actors

Event source

Signal

Time line

Semantics of DE Signals



signal in

F

signal out

Note: A signal is not a single event but all the events that flow on a path.

A signal is a partial function:

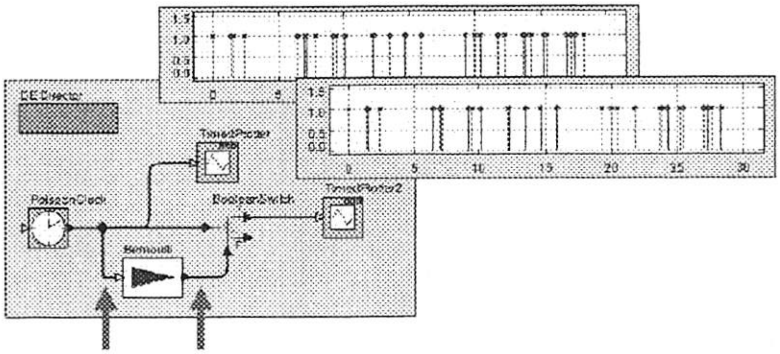
$$F : \Re \times N \to T$$

Data type (set of values)

Real numbers (approximated by doubles)

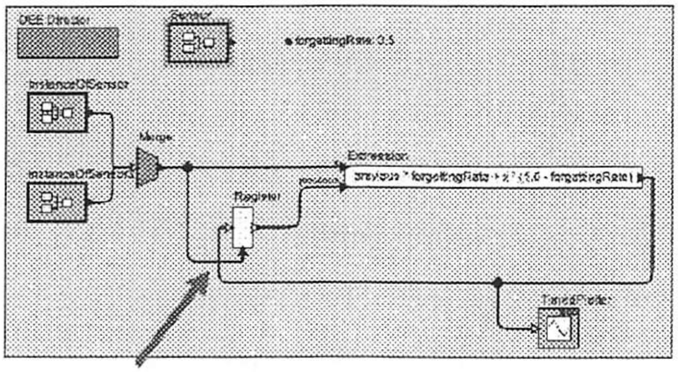
Natural numbers (allowing for simultaneous events in a signal)

Subtleties: Simultaneous Events



By default, an actor produces events with the same time as the input event. But in this example, we expect (and need) for the BooleanSwitch to "see" the output of the Bernoulli in the same "firing" where it sees the event from the PoissonClock. Events with identical time stamps are also ordered, and reactions to such events follow data precedence order.

Subtleties: Feedback



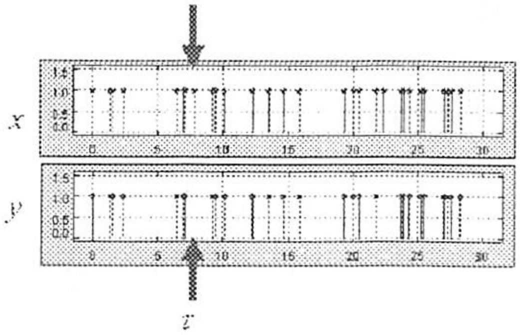
Data precedence analysis has to take into account the non-strictness of this actor (that an output can be produced despite the lack of an input).

Discrete-Event Semantics

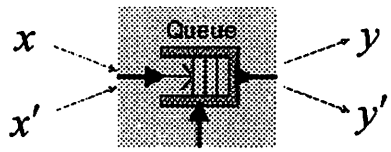
Cantor metric:

$$d(x, y) = 1/2^{\tau}$$

where τ is the earliest time where x and y differ.



Causality



Causal:

$$d(y,y') \le d(x,x')$$

Strictly causal:

$$d(y,y') < d(x,x')$$

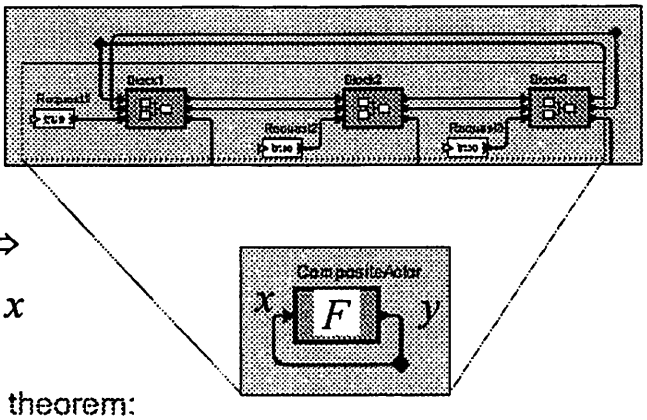
Delta causal:

$$\exists \delta < 1,$$
$$d(y,y') \le \delta\, d(x,x')$$

A delta-causal component is a "contraction map."

Semantics of Composition

If the components are deterministic, the composition is deterministic.

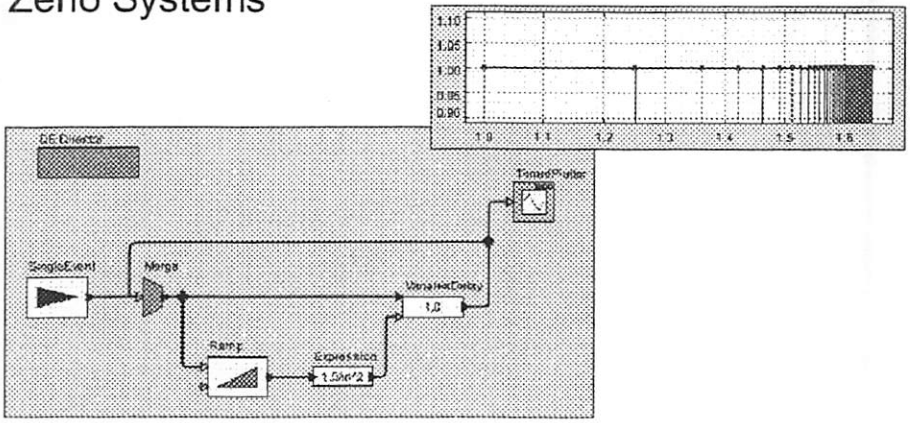


$$x = y \Rightarrow$$
$$F(x) = x$$

Banach fixed point theorem:

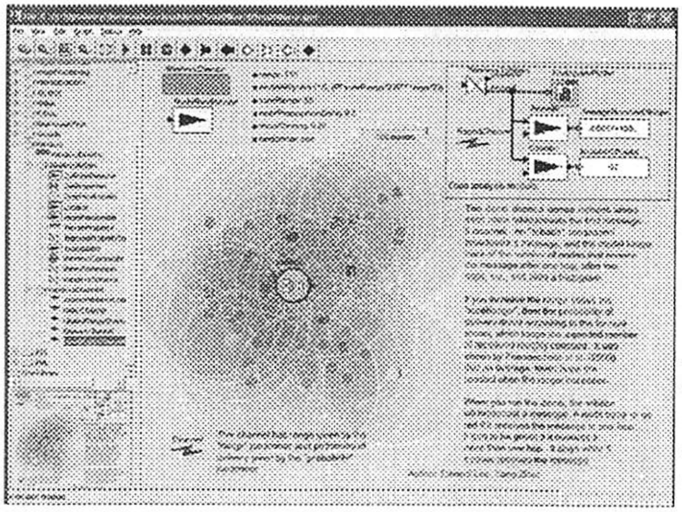
- Contraction map has a unique fixed point
- Execution procedure for finding that fixed point
- Successive approximations to the fixed point

Zeno Systems



Theorem: If every directed cycle contains a delta-causal component, then the system is non-Zeno.

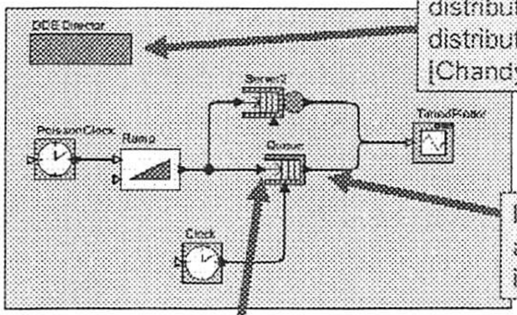
Extension of Discrete-Event Modeling for Wireless Sensor Nets



VisualSense extends the Ptolemy II discrete-event domain with communication between actors representing sensor nodes being mediated by a *channel*, which is another actor.

The example at the left shows a grid of nodes that relay messages from an *initiator* (center) via a channel that models a low (but non-zero) probability of long range links being viable.

Distributed Discrete Event Models as Currently Implemented in Ptolemy II



DDE Director supports distributed execution and a distributed notion of time [Chandy & Misra 1979]

Local notion of time in each actor, advancing only when input is received

Blocking read at input ports prevents time from locally advancing without "pemission" from a source

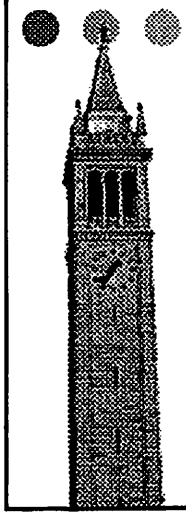
This is the "Chandy and Misra" style of distributed discrete events [1979], which compared to Croquet and Time Warp [Jefferson, 1985], is "conservative."

Other Interesting Possibilities for Distributed Discrete Events

- Time-Warp (Jefferson)
  - Optimistic computation
  - Backtracking
- Croquet (Reed)
  - Optimistic computation
  - Replication of computation
  - Voting algorithm (Lamport)

Conclusion

- There are many alternative concurrent MoCs
- The ones you know are the tip of the iceberg
- Ptolemy II is a lab for experimenting with them

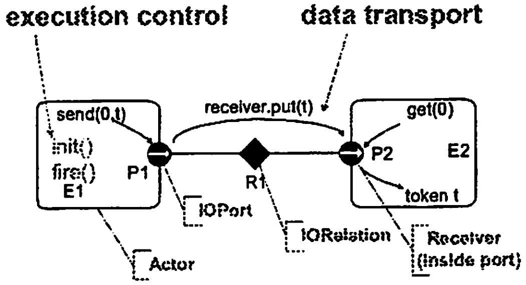


Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall, 2004

Lecture 4: Implementing Process Networks

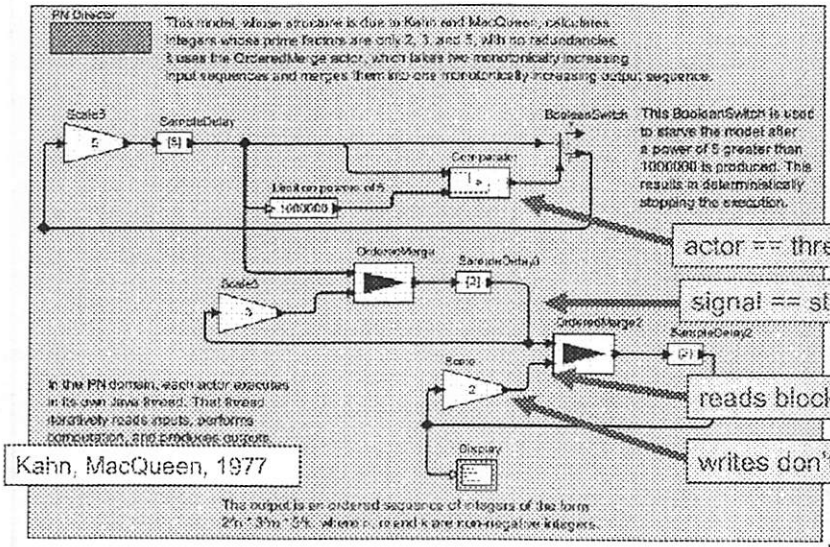
Abstract Semantics
of *Actor-Oriented* Models of Computation



Actor-Oriented Models of Computation that we have implemented:

- dataflow (several variants)
- process networks
- distributed process networks
- Click (push/pull)
- continuous-time
- CSP (rendezvous)
- discrete events
- distributed discrete events
- synchronous/reactive
- time-driven (several variants)
- ...

Process Networks (PN)



This model, whose structure is due to Kahn and MacQueen, calculates integers whose prime factors are only 2, 3, and 5, with no redundancies. It uses the OrderedMerge actor, which takes two monotonically increasing input sequences and merges them into one monotonically increasing output sequence.

This BooleanSwitch is used to starve the model after a power of 5 greater than 1000000 is produced. This results in deterministically stopping the execution.

In the PN domain, each actor executes in its own Java thread. That thread iteratively reads inputs, performs computation, and produces outputs.

The output is an ordered sequence of integers of the form 2^n * 3^m * 5^k, where n, m and k are non-negative integers.

actor == thread

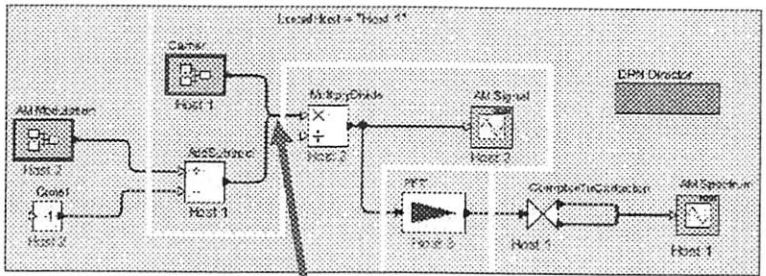
signal == stream

reads block

writes don't

Kahn, MacQueen, 1977

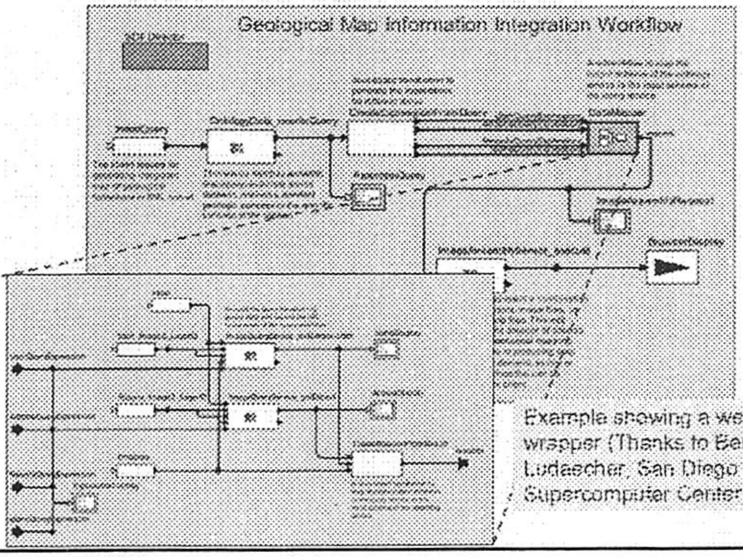
Distributed Process Networks



Transport mechanism between hosts is provided by the director. Transparently provides guaranteed delivery and ordered messages.

Created by Dominique Ragot, Thales Communications

Kepler: Extensions to Ptolemy II for Scientific Workflows

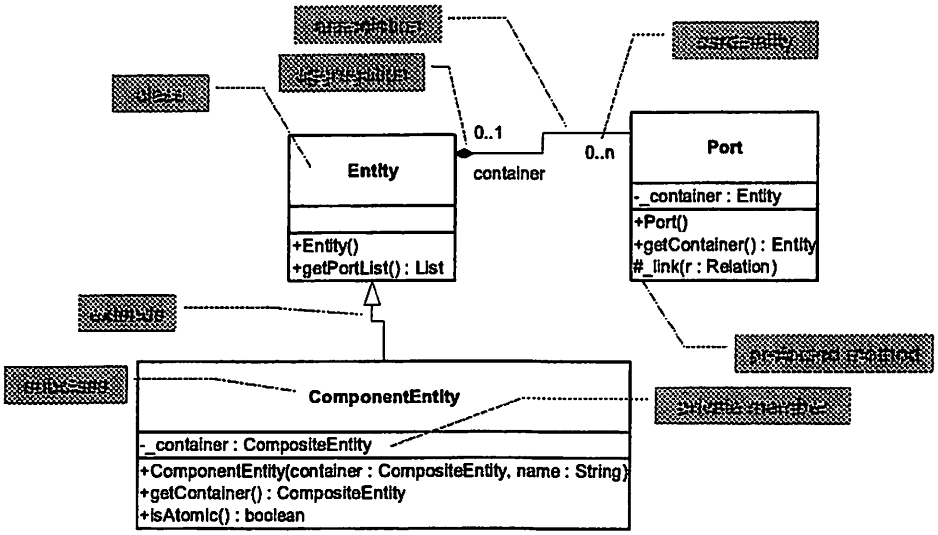


Example showing a web service wrapper (Thanks to Bertram Ludaescher, San Diego Supercomputer Center)

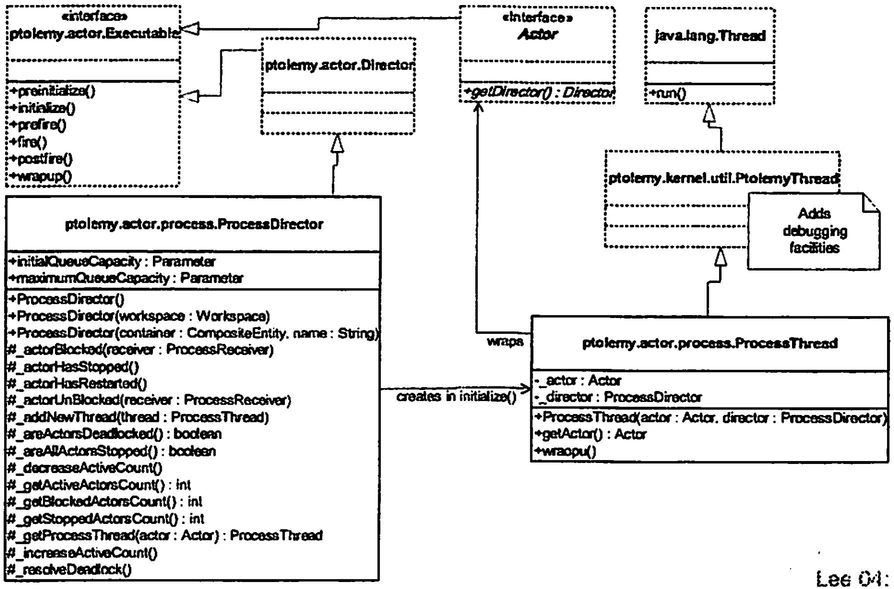
Coarse History

- Semantics for a very general form of PN were given by Gilles Kahn in 1974.
  - Fixed points of continuous and monotonic functions
- More limited but more easily implemented form given by Kahn and MacQueen in 1977.
  - Blocking reads and nonblocking writes.
- Many attempts to generalize the semantics to nondeterministic systems
  - Kosinski [1978], Stark [1980s], …
- Bounded memory execution strategy given by Parks in 1995.
  - Solves an undecidable problem.

Notation: UML Static Structure Diagrams



Instance of ProcessThread Wraps Every Actor



ProcessThread Implementation (Outline)

```
_director._increaseActiveCount();
try {
    _actor.initialize();
    boolean iterate = true;
    while (iterate) {
        if (_actor.prefire()) {
            _actor.fire();
            iterate = _actor.postfire();
        }
    }
} finally {
    try {
        wrapup();
    } finally {
        _director._decreaseActiveCount();
    }
}
```

Subtleties:

- The threads may never terminate on their own (a common situation).
- The model may deadlock (all active actors are waiting for input data)
- Execution may be paused by pushing the pause button.
- An actor may be deleted while it is executing.
- Any actor method may throw an exception.
- Buffers may grow without bound.

Typical fire() Method of an Actor

```
/** Compute the absolute value of the input.
 *  If there is no input, then produce no output.
 *  @exception IllegalActionException If there is
 *   no director.
 */
public void fire() throws IllegalActionException {
    if (input.hasToken(0)) {
        ScalarToken in = (ScalarToken)input.get(0);
        output.send(0, in.absolute());
    }
}
```

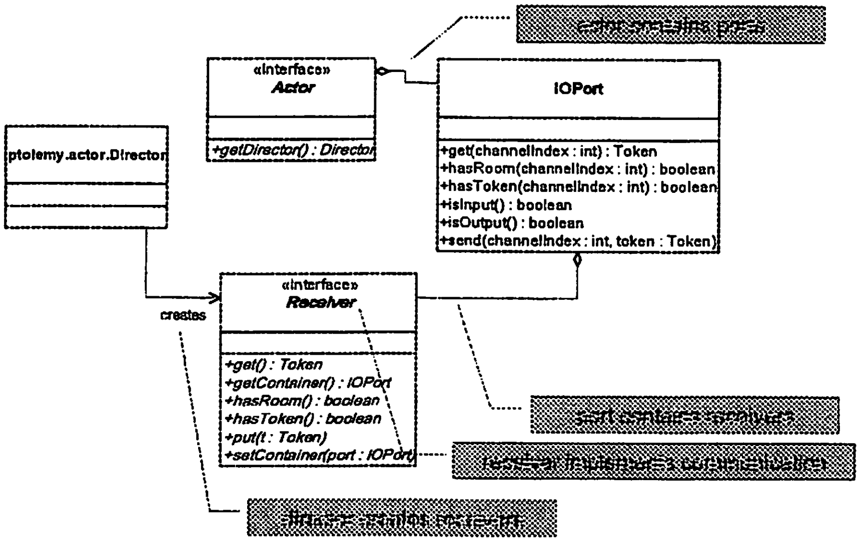
The get() method is behaviorally polymorphic: what it does depends on the director.

In PN, hasToken() always returns *true*, and the get() method blocks if there is no data.

Sketch of get() and send() Methods of IOPort

```
public Token get(int channelIndex) {
    Receiver[] localReceivers = getReceivers();
    return localReceivers[channelIndex].get();
}

public void send(int channelIndex, Token token) {
    Receiver[] farReceivers = getRemoteReceivers();
    farReceivers[channelIndex].put(token);
}
```

Ports and Receivers



Process Networks Receiver Outline

```
public class PNQueueReceiver extends QueueReceiver
        implements ProcessReceiver {

    private boolean _readBlocked;

    public boolean hasToken() {
        return true;
    }

    public synchronized Token get() {
        ...
    }

    public synchronized void put(Token token) {
        ...
    }
}
```

flag indicating whether the consumer thread is blocked.

always indicate that a token is available

acquire a lock on the receiver before executing put or get

get() Method (Simplified)

super class returns true only if there is a token in the queue

```
    public synchronized Token get() {
        PNDirector director = ... get director ...;
        while (!super.hasToken()) {
            _readBlocked = true;
            director._actorBlocked(this);
            while (_readBlocked) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    throw new TerminateProcessException("");
                }
            }
        }
        return result = super.get();
    }
```

notify the director that the consumer thread is blocked

release the lock on the receiver and stall the thread

use this exception to stop execution of the actor thread

super class returns the first token in the queue.

put() Method (Simplified)

```
public synchronized void put(Token token) {
    PNDirector director = ... get director ...;
    super.put(token);
    if (_readBlocked) {
        director._actorUnBlocked(this);
        _readBlocked = false;
        notifyAll();
    }
}
```

[illegible]

[illegible] blocked on wait().

Subtleties

- Director must be able to detect deadlock.
  - It keeps track of blocked threads
- Stopping execution is tricky
  - When to stop a thread?
  - How to stop a thread?
- Non-blocking writes are problematic in practice
  - Unbounded memory usage
  - Use Parks' strategy:
    - Bound the buffers
    - Block on writes when buffer is full
    - On deadlock, increase buffers sizes for actors blocked on writes
    - Provably executes in bounded memory if that is possible (subtle).

Stopping Threads

"Why is Thread.stop deprecated?

Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be *damaged*. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future."

Java JDK 1.4 documentation.

Thread.suspend() and resume() are similarly deprecated.

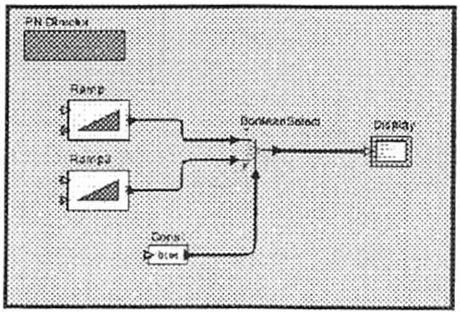
Thread.destroy() is unimplemented.

Properties of PN (Two Big Topics)

- Assuming "well-behaved" actors, a PN network is determinate in that the sequence of tokens on each arc is independent of the thread scheduling strategy.
  - Making this statement precise, however, is nontrivial.
- PN is Turing complete.
  - Given only boolean tokens, memoryless functional actors, Switch, Select, and initial tokens, one can implement a universal Turing machine.
  - Whether a PN network deadlocks is undecidable.
  - Whether buffers grow without bound is undecidable.

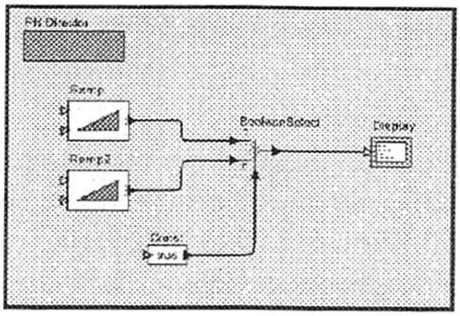
Question 1:
Is "Fair" Thread Scheduling a Good Idea?

In the following model, what happens if every thread is given an equal opportunity to run?



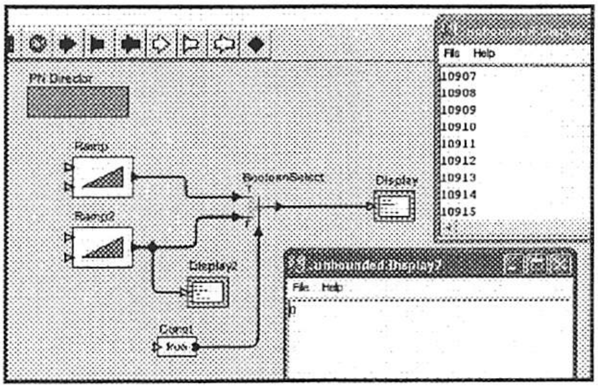
Question 2:
Is "Data-Driven" Execution a Good Idea?

In the following model, if threads are allowed to run when they have input data on connected inputs, what will happen?



Question 3:
When are Outputs Required?

Is the execution shown for the following model the "right" execution?

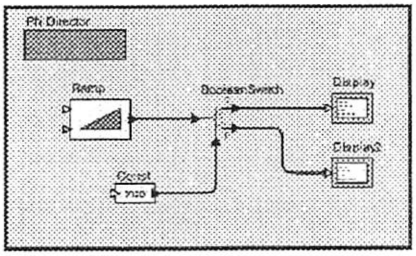


```
10907
10908
10909
10910
10911
10912
10913
10914
10915
```

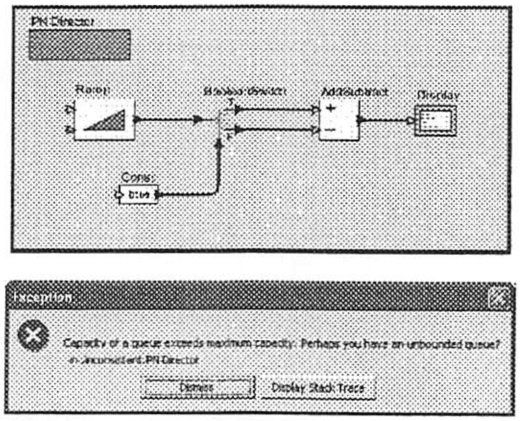
```
0
```

Question 4:
Is "Demand-Driven" Execution a Good Idea?

In the following model, if threads are allowed to run when another thread requires their outputs, what will happen?

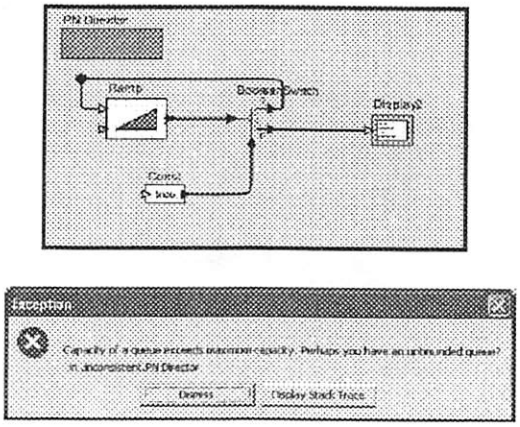


Question 5:
What is the “Correct” Execution of This Model?



Capacity of a queue exceeds maximum capacity. Perhaps you have an unbounded queue?
in .inconsistent.PN Director

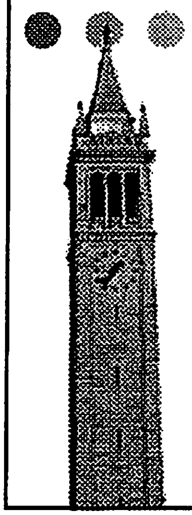
Question 6:
What is the Correct Behavior of this Model?



Capacity of a queue exceeds maximum capacity. Perhaps you have an unbounded queue?
in .inconsistent.PN Director

Summary

- Process Networks (PN) are an attractive concurrent model of computation.
- Basics of an implementation using monitors is straightforward, but there are some subtleties:
  - How to detect deadlock
  - How to keep memory usage bounded
  - How (or whether) to get fairness
  - What thread scheduling policies are correct?
  - What does "correct" mean?



Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall, 2004

Lecture 5: Extending Ptolemy II

Background for Ptolemy II

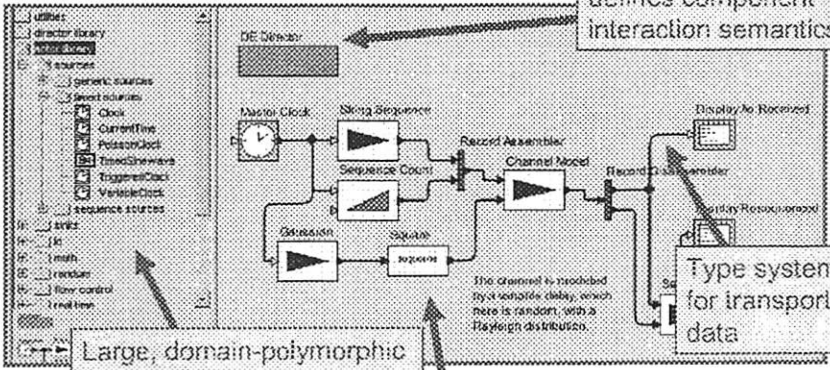
- Gabriel (1986-1991)
  - Written in Lisp
  - Aimed at signal processing
  - Synchronous dataflow (SDF) block diagrams
  - Parallel schedulers
  - Code generators for DSPs
  - Hardware/software co-simulators
- Ptolemy Classic (1990-1997)
  - Written in C++
  - Multiple models of computation
  - Hierarchical heterogeneity
  - Dataflow variants: BDF, DDF, PN
  - C/VHDL/DSP code generators
  - Optimizing SDF schedulers
  - Higher-order components
- Ptolemy II (1996-2022)
  - Written in Java
  - Domain polymorphism
  - Multithreaded
  - Network integrated
  - Modal models
  - Sophisticated type system
  - CT, HDF, CI, GR, etc.

Each of these served us, first and foremost, as a laboratory for investigating design.

- PtPlot (1997-??)
  - Java plotting package
- Tycho (1996-1998)
  - Itcl/Tk GUI framework
- Diva (1998-2000)
  - Java GUI framework

All open source
All truly free software (cf. FSF)

Framework Infrastructure that Supports Diverse Experiments with Models of Computation



The channel is modeled by a variable delay, which here is random, with a Rayleigh distribution.

Concurrency management supporting dynamic model structure.

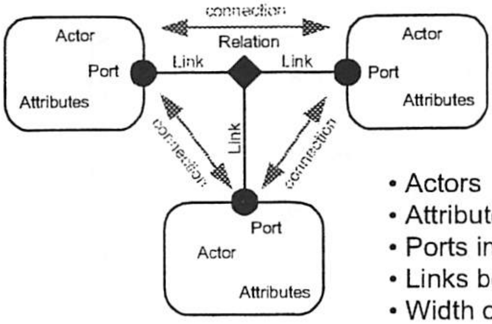
Director from a library defines component interaction semantics

Type system for transported data

Large, domain-polymorphic component library.

Visual editor supporting an abstract syntax

The Basic Abstract Syntax



- Actors
- Attributes on actors (parameters)
- Ports in actors
- Links between ports
- Width on links (channels)
- Hierarchy

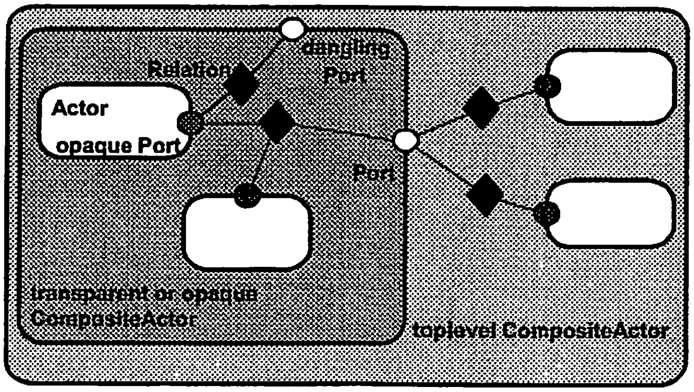
Concrete syntaxes:
- XML
- Visual pictures
- Actor languages (Cal, StreamIT, …)

MoML
XML Schema for this Abstract Syntax

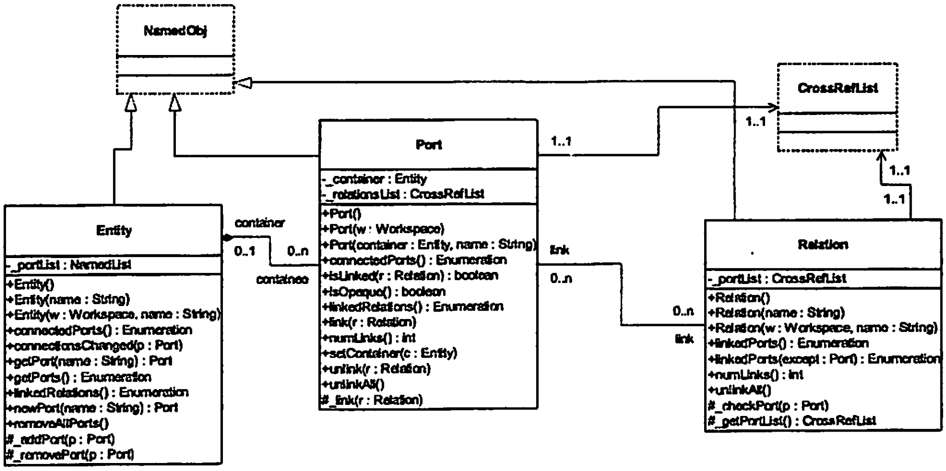
Ptolemy II designs are represented in XML:

```
...
<entity name="FFT" class="ptolemy.domains.sdf.lib.FFT">
    <property name="order" class="ptolemy.data.expr.Parameter" value="order">
    </property>
    <port name="input" class="ptolemy.domains.sdf.kernel.SDFIOPort">
        ...
    </port>
    ...
</entity>
...
<link port="FFT.input" relation="relation"/>
<link port="AbsoluteValue2.output" relation="relation"/>
...
```

Hierarchy - Composite Components



Kernel Classes
Support the Abstract Syntax



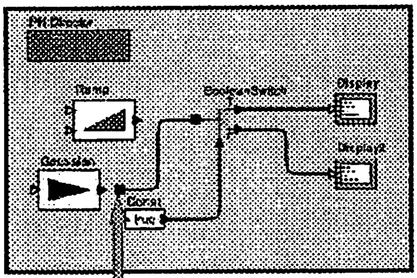
Concurrency Management Supporting Dynamic Model Structure

Changes to a model while the model is executing:

- Change parameter values
- Change model structure

How can this be made safe?

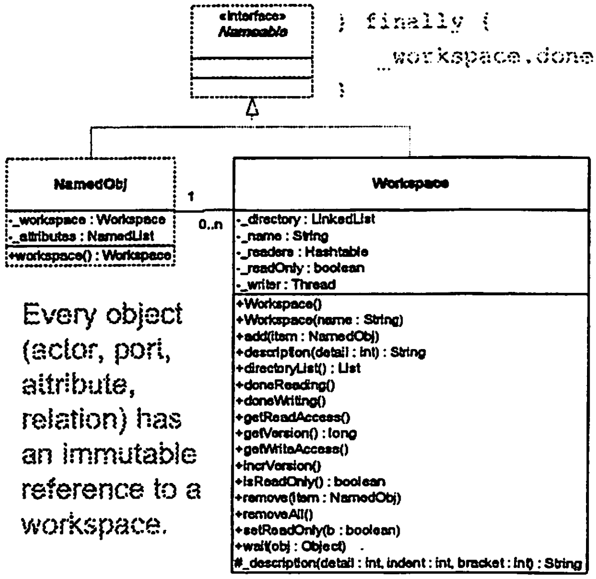
- Workspace class
- ChangeRequest class
- stopFire() method



Can dynamically modify the model while it executes… safely.

Workspace

```
try {
   _workspace.getReadAccess();
   … actions depending on model structure
} finally {
   _workspace.doneReading();
}
```



«Interface»
Nameable

NamedObj
-_workspace : Workspace
-_attributes : NamedList
+workspace() : Workspace

1

0..n

Workspace
-_directory : LinkedList
-_name : String
-_readers : Hashtable
-_readOnly : boolean
-_writer : Thread
+Workspace()
+Workspace(name : String)
+add(item : NamedObj)
+description(detail : int) : String
+directoryList() : List
+doneReading()
+doneWriting()
+getReadAccess()
+getVersion() : long
+getWriteAccess()
+incrVersion()
+isReadOnly() : boolean
+remove(item : NamedObj)
+removeAll()
+setReadOnly(b : boolean)
+wait(obj : Object)
#_description(detail : int, indent : int, bracket : int) : String

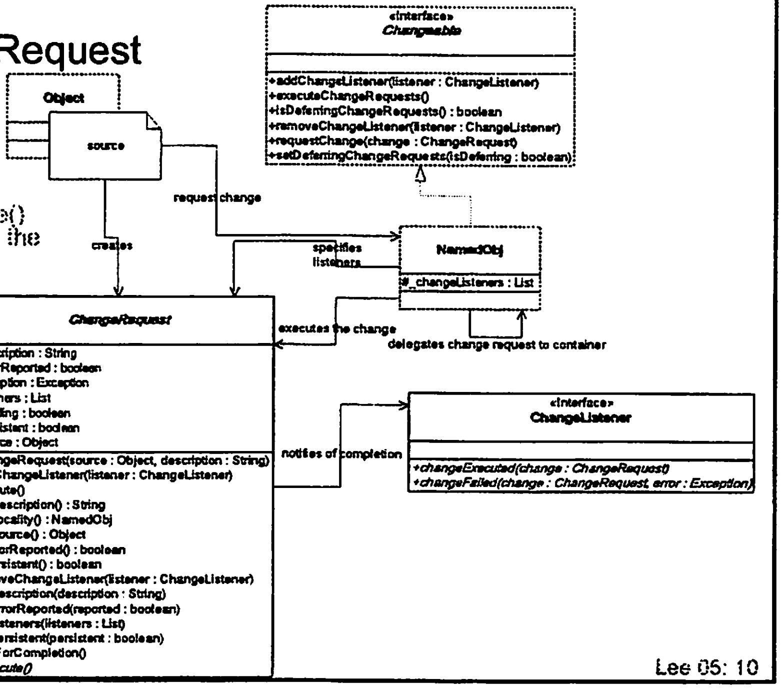
Every object (actor, port, attribute, relation) has an immutable reference to a workspace.

Many threads can have read access at the same time. Only one thread can have write access, and only if no other thread has read access.

Specialized wait(Object) method releases the locks during the wait.

ChangeRequest

Changes to a model are requested by calling requestChange() on an object in the model



Object

source

«Interface»
Changeable
+addChangeListener(listener : ChangeListener)
+executeChangeRequests()
+isDeferringChangeRequests() : boolean
+removeChangeListener(listener : ChangeListener)
+requestChange(change : ChangeRequest)
+setDeferringChangeRequests(isDeferring : boolean)

request change

creates

specifies listeners

NamedObj
#_changeListeners : List

executes the change

delegates change request to container

ChangeRequest
-_description : String
-_errorReported : boolean
-_exception : Exception
-_listeners : List
-_pending : boolean
-_persistent : boolean
-_source : Object
+ChangeRequest(source : Object, description : String)
+addChangeListener(listener : ChangeListener)
+execute()
+getDescription() : String
+getLocality() : NamedObj
+getSource() : Object
+isErrorReported() : boolean
+isPersistent() : boolean
+removeChangeListener(listener : ChangeListener)
+setDescription(description : String)
+setErrorReported(reported : boolean)
+setListeners(listeners : List)
+setPersistent(persistent : boolean)
+waitForCompletion()
#_execute()

notifies of completion

«Interface»
ChangeListener
+changeExecuted(change : ChangeRequest)
+changeFailed(change : ChangeRequest, error : Exception)

When to Execute Change Requests

In many models of computation, there is a natural time: between iterations.

In PN, this is not a trivial question…

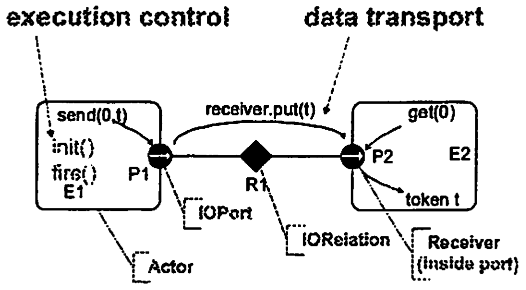
- All threads must be stopped (blocked)
  - On reads
  - On writes to full buffers
  - Or block themselves with a wait()
- What happens when the model structure changes during a call to get()?

ProcessThread with Pauses for Mutations

```
while(iterate) {
    if (_director.isStopFireRequested()) {
        synchronized (_director) {
            _director._actorHasStopped();
                while (_director.isStopFireRequested()) {
                    try {
                        workspace.wait(_director);
                    } catch (InterruptedException ex) {
                        break;
                    }
                }
                _director._actorHasRestarted();
            }
        }
    }
    boolean iterate = true;
    while (iterate) {
        if (_actor.prefire()) {
            _actor.fire();
            iterate = _actor.postfire();
        }
    }
```

Specialized wait() method releases workspace locks while the thread is suspended.

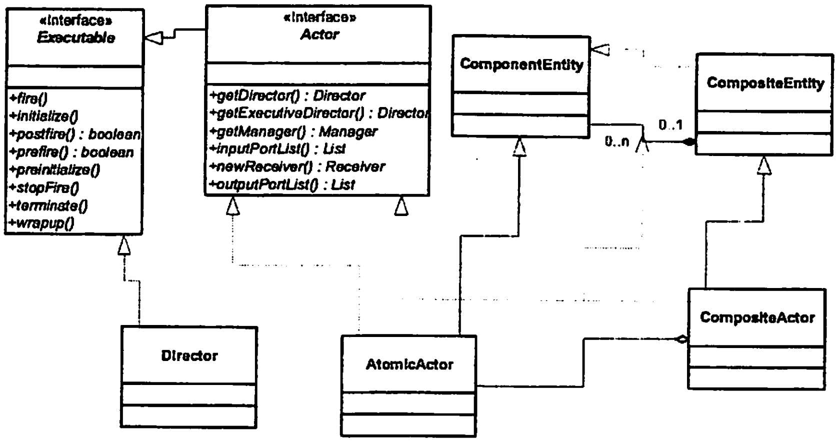
Abstract Semantics
of *Actor-Oriented* Models of Computation



Actor-Oriented Models of Computation that we have implemented:

- dataflow (several variants)
- process networks
- distributed process networks
- Click (push/pull)
- continuous-time
- CSP (rendezvous)
- discrete events
- distributed discrete events
- synchronous/reactive
- time-driven (several variants)
- ...

Object Model for Executable Components


«Interface»
Executable

+fire()
+initialize()
+postfire() : boolean
+prefire() : boolean
+preinitialize()
+stopFire()
+terminate()
+wrapup()

«Interface»
Actor

+getDirector() : Director
+getExecutiveDirector() : Directo
+getManager() : Manager
+inputPortList() : List
+newReceiver() : Receiver
+outputPortList() : List

ComponentEntity

CompositeEntity

0..1

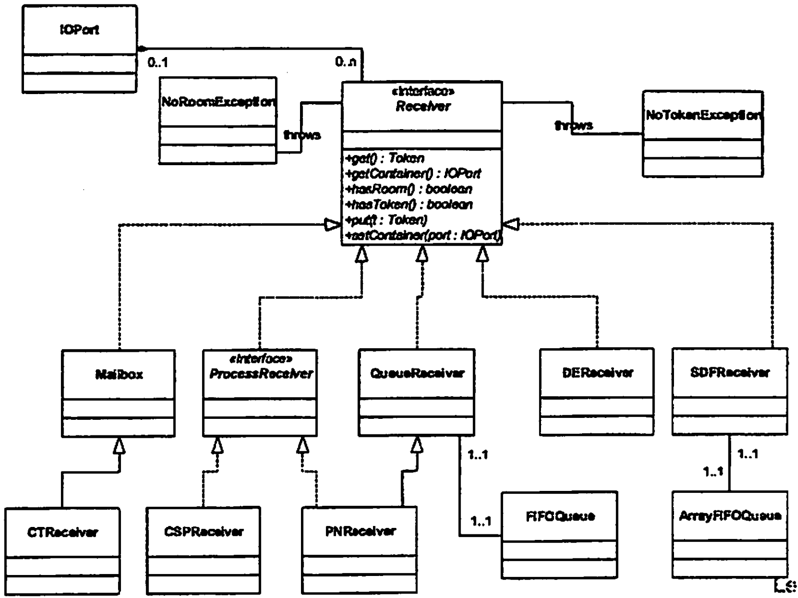
0..n

Director

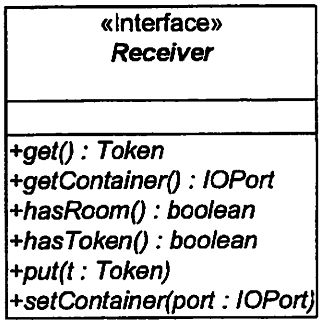
AtomicActor

CompositeActor

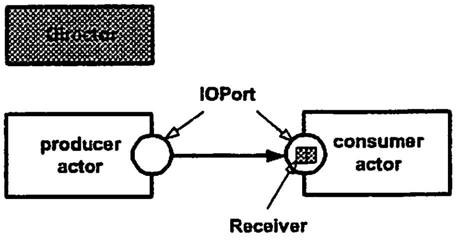
Object Model (Simplified) for Communication Infrastructure



Object-Oriented Approach to Achieving Behavioral Polymorphism



These polymorphic methods implement the communication semantics of a domain in Ptolemy II. The receiver instance used in communication is supplied by the director, not by the component.



Recall: Behavioral polymorphism is the idea that components can be defined to operate with multiple models of computation and multiple middleware frameworks.

Extension Exercise

Build a director that subclasses PNDirector to allow ports to alter the "blocking read" behavior. In particular, if a port has a parameter named "tellTheTruth" then the receivers that your director creates should "tell the truth" when hasToken() is called. That is, instead of always returning true, they should return true only if there is a token in the receiver.

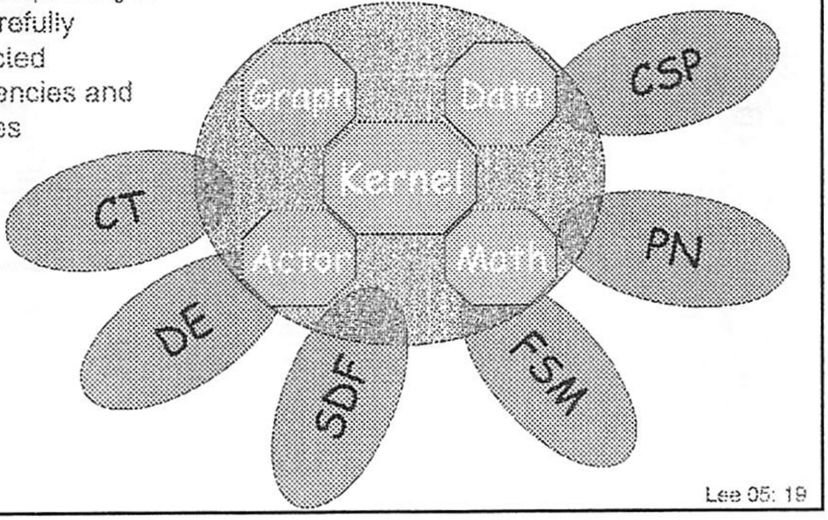
Parameterizing the behavior of a receiver is a simple form of communication refinement, a key principle in, for example, Metropolis.

Implementation of the New Model of Computation

```
package experiment;

import …

public class NondogmaticPNDirector extends PNDirector {
    public NondogmaticPNDirector(CompositeEntity container, String name)
           throws IllegalActionException, NameDuplicationException {
       super(container, name);
    }
    public Receiver newReceiver() {
        return new FlexibleReceiver();
    }
    public class FlexibleReceiver extends PNQueueReceiver {
        public boolean hasToken() {
            IOPort port = getContainer();
            Attribute attribute = port.getAttribute("tellTheTruth");
            if (attribute == null) {
                 return super.hasToken();
            }
            // Tell the truth...
            return _queue.size() > 0;
        }
    }
}
```

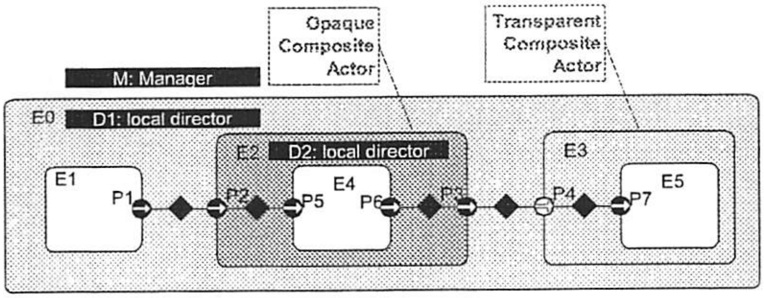
Ptolemy II Software Architecture Built for Extensibility

Ptolemy II packages have carefully constructed dependencies and interfaces

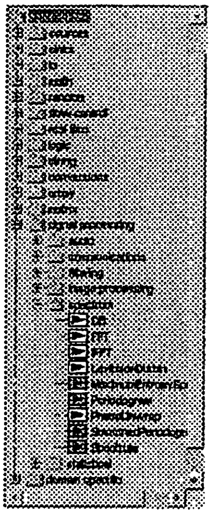


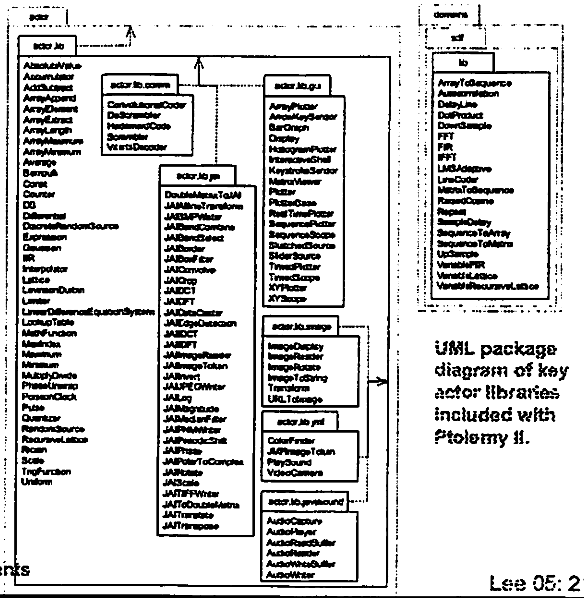
Hierarchical Heterogeneity

Directors are domain-specific. A composite actor with a director becomes opaque. The Manager is domain-independent.



Ptolemy II Component Library

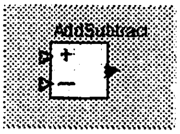




UML package diagram of key actor libraries included with Ptolemy II.

- Data polymorphic components
- Behaviorally polymorphic components

Polymorphic Components - Component Library Works Across Data Types and Domains



Data polymorphism:

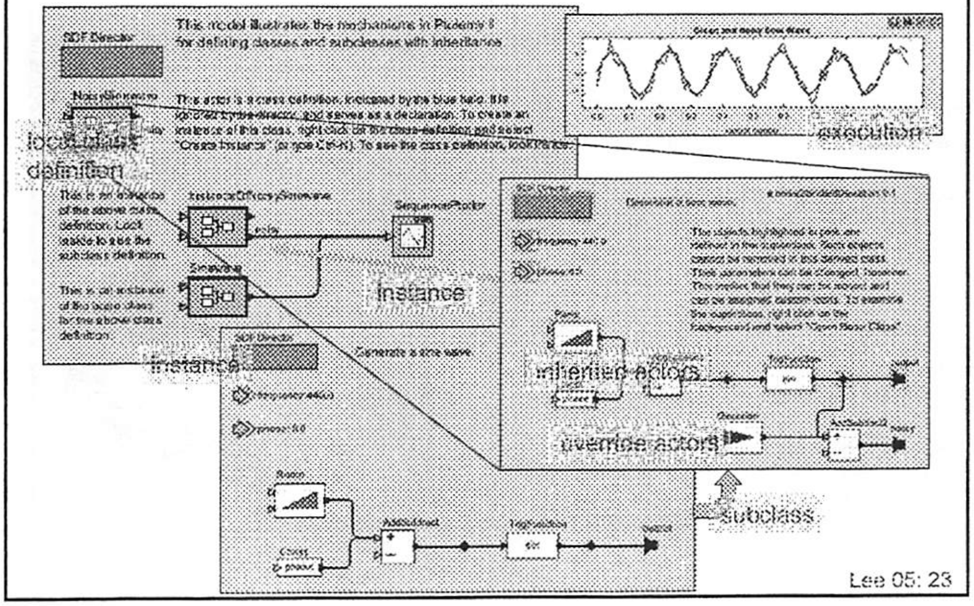
- Add numbers (int, float, double, Complex)
- Add strings (concatenation)
- Add composite types (arrays, records, matrices)
- Add user-defined types

Behavioral polymorphism:

- In dataflow, add when all connected inputs have data
- In a time-triggered model, add when the clock ticks
- In discrete-event, add when any connected input has data, and add in zero time
- In process networks, execute an infinite loop in a thread that blocks when reading empty inputs
- In CSP, execute an infinite loop that performs rendezvous on input or output
- In push/pull, ports are push or pull (declared or inferred) and behave accordingly
- In real-time CORBA, priorities are associated with ports and a dispatcher determines when to add

By not choosing among these when defining the component, we get a huge increment in component re-usability. But how do we ensure that the component will work in all these circumstances?

Shared Infrastructure
Modularity Mechanisms



local class definition

execution

instance

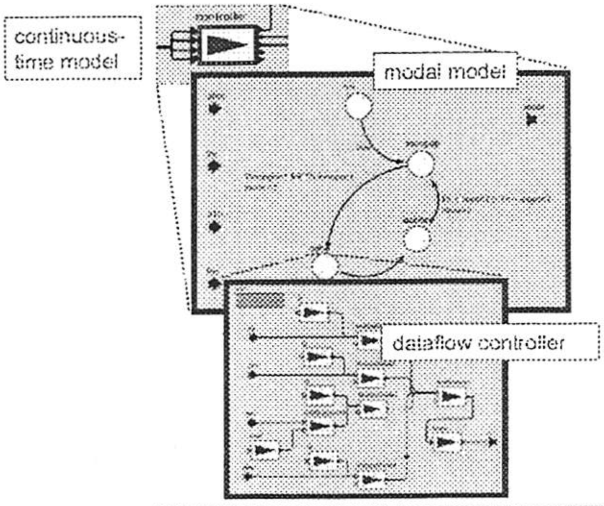
instance

inherited actors

override actors

subclass

More Shared Infrastructure: Hierarchical Heterogeneity and Modal Models



continuous-time model

modal model

dataflow controller

example Ptolemy II model: hybrid control system

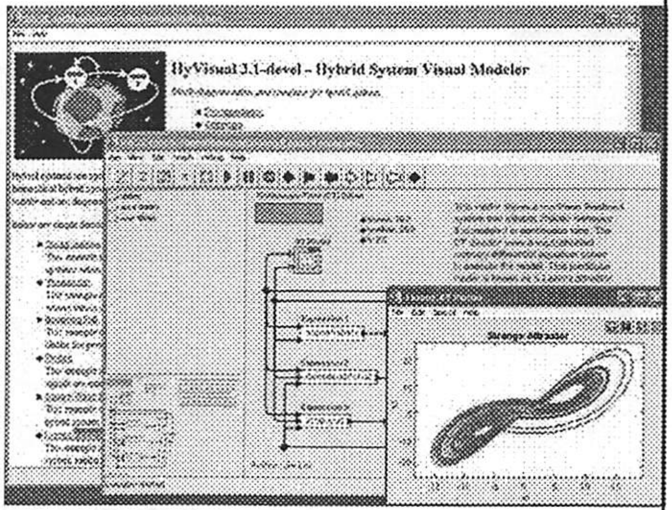
Branding

Ptolemy II *configurations* are Ptolemy II models that specify

- welcome window
- help menu contents
- library contents
- File->New menu contents
- default model structure
- etc.

A configuration can identify its own "brand" independent of the "Ptolemy II" name and can have more targeted objectives.

An example is HyVisual, a tool for hybrid system modeling. VisualSense is another tool for wireless sensor network modeling.

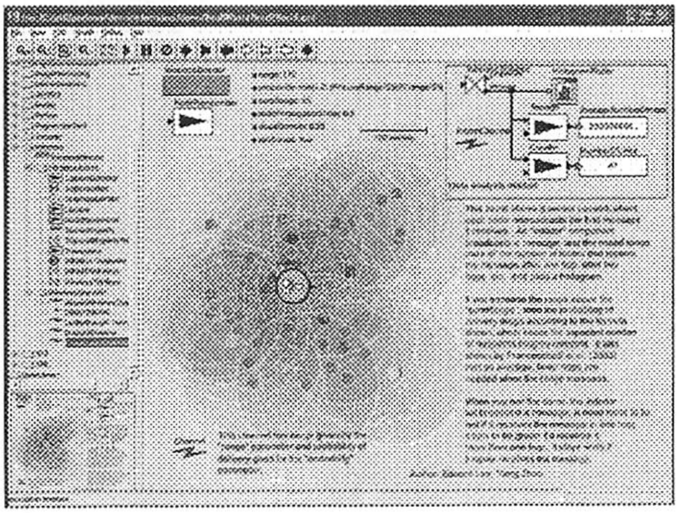


Ptolemy II Extension Points

- Define actors
- Interface to foreign tools (e.g. Python, MATLAB)
- Interface to verification tools (e.g. Chic)
- Define actor definition languages
- Define directors (and models of computation)
- Define visual editors
- Define textual syntaxes and editors
- Packaged, branded configurations

All of our "domains" are extensions built on a core infrastructure.

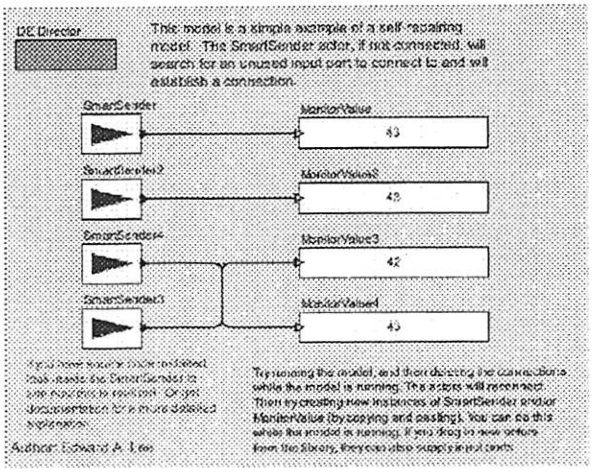
Example Extension: VisualSense



- Branded
- Customized visualization
- Customized model of computation (an extension of DE)
- Customized actor library
- Motivated some extensions to the core (e.g. classes, icon editor).

Example Extensions: Self-Repairing Models

Concept demonstration built together with Boeing to show how to write actors that adaptively reconstruct connections when the model structure changes.



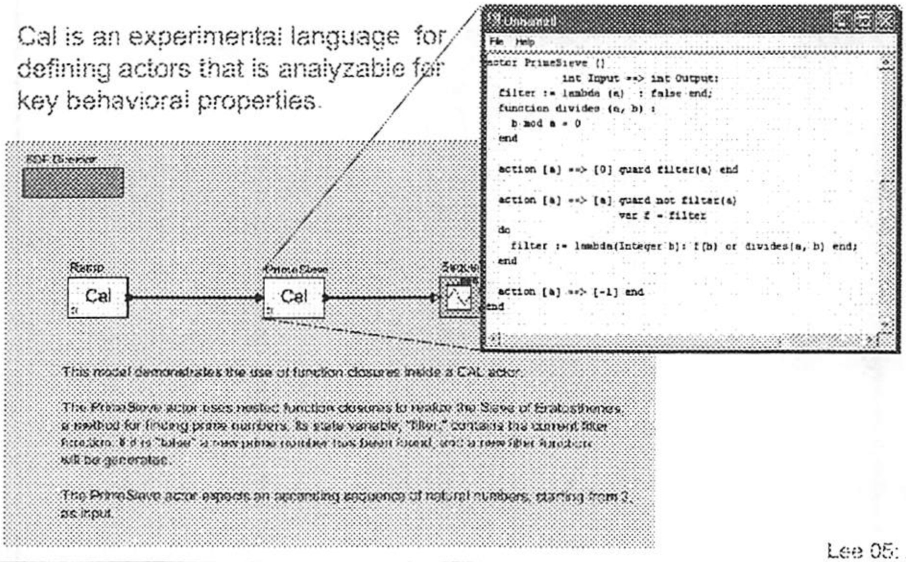
This model is a simple example of a self-repairing model. The SmartSender actor, if not connected, will search for an unused input port to connect to and will establish a connection.

Author: Edward A. Lee

Try running the model, and then deleting the connections while the model is running. The actors will reconnect. Then try creating new instances of SmartSender and/or MonitorValue (by copying and pasting). You can do this while the model is running. If you drag in new actors from the library, they can also supply input ports.

Example Extensions
Python Actors and Cal Actors

Cal is an experimental language for defining actors that is analyzable for key behavioral properties.



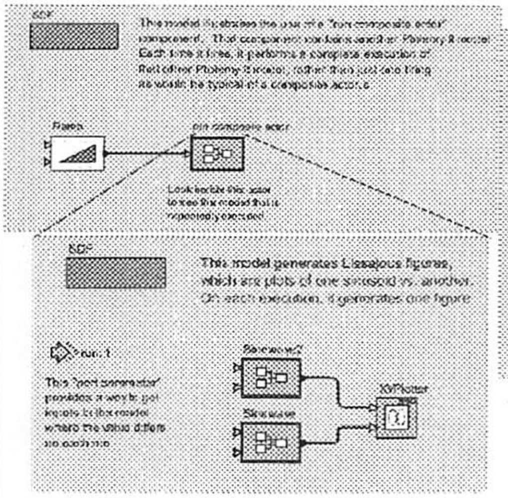
```
actor PrimeSieve ()
          int Input ==> int Output:
  filter := lambda (a)  : false end;
  function divides (a, b) :
    b mod a = 0
  end

  action [a] ==> [0] guard filter(a) end

  action [a] ==> [a] guard not filter(a)
                    var f = filter
  do
    filter := lambda(Integer b): f(b) or divides(a, b) end;
  end

  action [a] ==> [-1] end
end
```

This model demonstrates the use of function closures inside a CAL actor.

The PrimeSieve actor uses nested function closures to realize the Sieve of Eratosthenes, a method for finding prime numbers. Its state variable, "filter," contains the current filter function. If it is "false" a new prime number has been found, and a new filter function will be generated.

The PrimeSieve actor expects an ascending sequence of natural numbers, starting from 3, as input.

Example Extensions
Using Models to Control Models

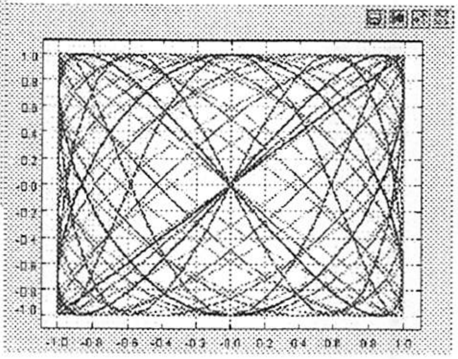


This model illustrates the use of a "run composite actor" component. That component contains another Ptolemy II model. Each time it fires, it performs a complete execution of that other Ptolemy II model, rather than just one firing as would be typical of a composite actor.

Look inside this actor to see the model that is repeatedly executed.

This model generates Lissajous figures, which are plots of one sinusoid vs. another. On each execution, it generates one figure.

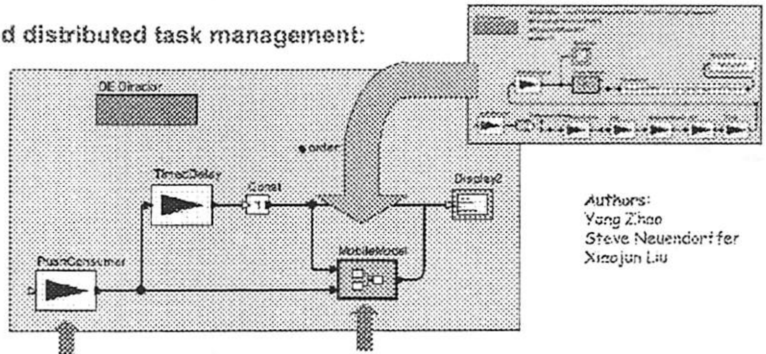
This "port parameter" provides a way to get inputs to the model where the value differs on each run.



This is an example of a "higher-order component," or an actor that references one or more other actors.

Examples of Extensions
Mobile Models

Model-based distributed task management:

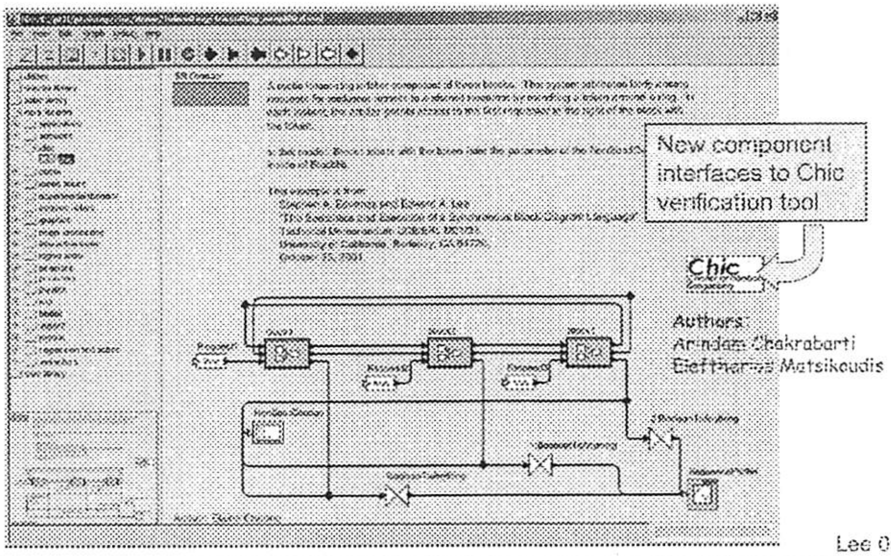


Authors:
Yang Zhao
Steve Neuendorffer
Xiaojun Liu

PushConsumer actor receives pushed data provided via CORBA, where the data is an XML model of a signal analysis algorithm.

MobileModel actor accepts a StringToken containing an XML description of a model. It then executes that model on a stream of input data.

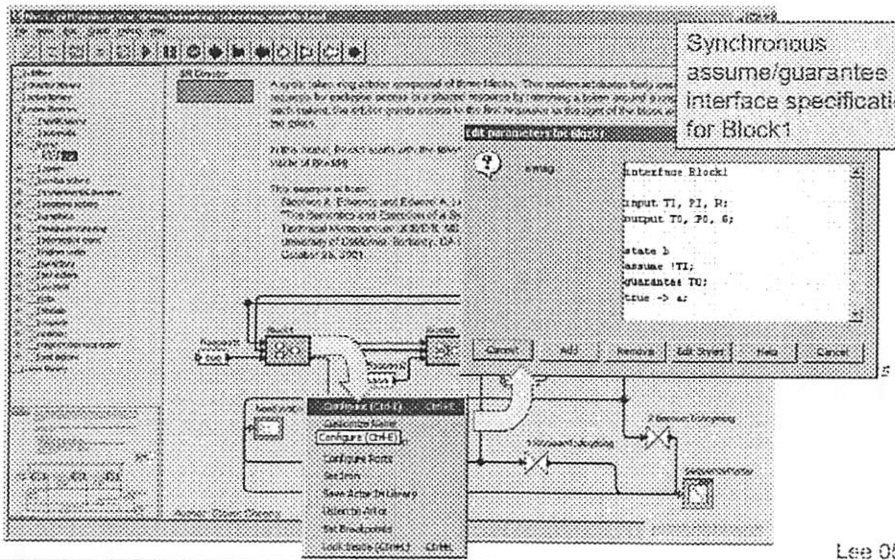
Examples of Extensions
Hooks to Verification Tools



New component interfaces to Chic verification tool

Authors:
Arindam Chakrabarti
Eleftherios Matsikoudis

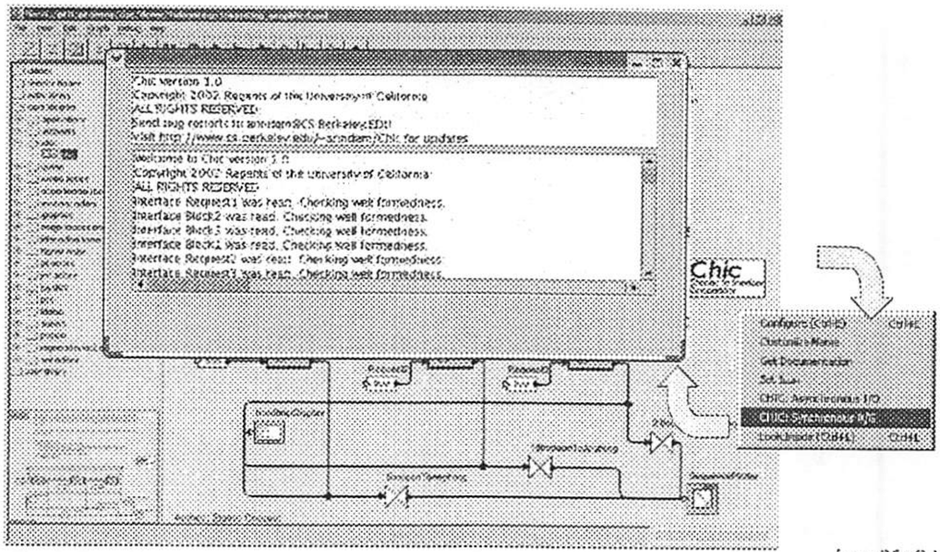
Examples of Extensions
Hooks to Verification Tools



```
interface Block1

input TI, PI, R;
output TO, PO, G;

state b
assume !TI;
guarantee TO;
true -> a;
```

Synchronous assume/guarantee interface specification for Block1

Examples of Extensions
Hooks to Verification Tools



```
Chic version 1.0
Copyright 2002 Regents of the University of California
ALL RIGHTS RESERVED
Send bug reports to arindam@CS.Berkeley.EDU
Visit http://www.cs.berkeley.edu/~arindam/Chic for updates
```

```
Welcome to Chic version 1.0
Copyright 2002 Regents of the University of California
ALL RIGHTS RESERVED
Interface Request1 was read. Checking well formedness.
Interface Block2 was read. Checking well formedness.
Interface Block3 was read. Checking well formedness.
Interface Block1 was read. Checking well formedness.
Interface Request2 was read. Checking well formedness.
Interface Request3 was read. Checking well formedness.
```

Getting More Information: Design Document

PTOLEMY II

HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

VOLUME 1: INTRODUCTION TO PTOLEMY II

Volume 1:
User-Oriented

PTOLEMY II

HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

VOLUME 2: PTOLEMY II SOFTWARE ARCHITECTURE

Volume 2:
Developer-Oriented

PTOLEMY II

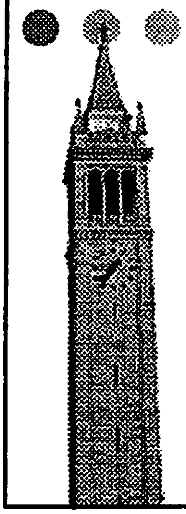
HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

VOLUME 3: PTOLEMY II DOMAINS

Volume 3:
Researcher-Oriented

Summary

Ptolemy II provides considerable infrastructure for experimenting with models of computation.

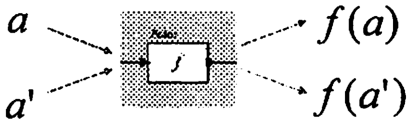


Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall, 2004

Lecture 6: Process Networks Semantics

PN Semantics
Where This is Going



A signal is a sequence of values

Define a prefix order:

$$a \sqsubseteq a'$$

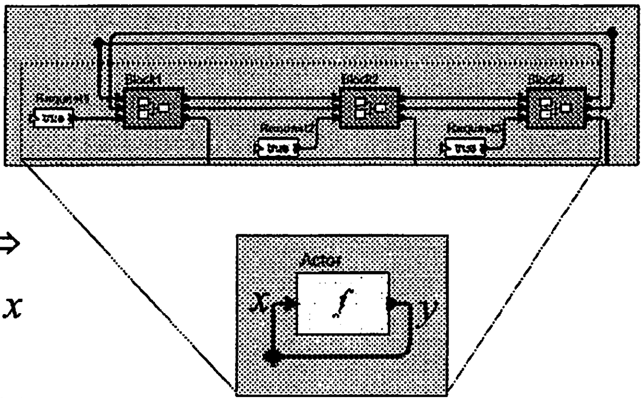
means that x is a prefix of y.

Actors are *monotonic* functions:

$$a \sqsubseteq a' \Rightarrow f(a) \sqsubseteq f(a')$$

Stronger condition: Actors are *continuous* functions (intuitively: they don't wait forever to produce outputs).

PN Semantics of Composition (Kahn, '74)
This Approach to Semantics is "Tarskian"



If the components are deterministic, the composition is deterministic.

$$x = y \Rightarrow$$
$$f(x) = x$$

Fixed point theorem:

- Continuous function has a unique least fixed point
- Execution procedure for finding that fixed point
- Successive approximations to the fixed point

What is Order?

Intuition:

1. 0 < 1
2. 1 < ∞
3. child < parent
4. child > parent
5. 11,000/3,501 is a better approximation to π than 22/7
6. integer n is a divisor of integer m.
7. Set A is a subset of set B.

Which of these are *partial* orders?

Relations

- A *relation* R from A to B is a subset of $A \times B$
- A *function* F from A to B is a relation where
  $(a, b) \in R$ and $(a, b') \in R \Rightarrow b = b'$
- A *binary relation* R on A is a subset of $A \times A$
- A *binary relation* R on A is *reflexive* if
  $\forall\, a \in A,\ (a, a) \in R$
- A *binary relation* R on A is *symmetric* if
  $(a, b) \in R \Rightarrow (b, a) \in R$
- A *binary relation* R on A is *antisymmetric* if
  $(a, b) \in R$ and $(b, a) \in R \Rightarrow a = b$
- A *binary relation* R on A is *transitive* if
  $(a, b) \in R$ and $(b, c) \in R \Rightarrow (a, c) \in R$

Infix Notation for Binary Relations

- $(a, b) \in R$ can be written $a\, R\, b$
- A symbol can be used instead of R. For examples:
  - $\leq \subset N \times N$ is a relation.
  - $(a, b) \in\ \leq$ is written $a \leq b$
- A function $f \in (A, B)$ can be written $f\colon A \rightarrow B$

Partial Orders

A *partial order* on the set A is a binary relation $\leq$ that is:
For all $a, b, c \in A$,

- reflexive: $a \leq a$
- antisymmetric: $a \leq b$ and $b \leq a \Rightarrow a = b$
- transitive: $a \leq b$ and $b \leq c \Rightarrow a \leq c$

A *partially ordered set* (*poset*) is a set A and a binary relation $\leq$, written $(A, \leq)$.

Strict Partial Order

For every partial order $\leq$ there is a *strict partial order* $<$ where $a < b$ if and only if $a \leq b$ and $a \neq b$.

A *strict poset* is a set and a strict partial order.

Total Orders

Elements *a* and *b* of a poset $(A, \leq)$ are *comparable* if either $a \leq b$ or $b \leq a$. Otherwise they are *incomparable*.

A poset $(A, \leq)$ is *totally ordered* if every pair of elements is comparable.

Totally ordered sets are also called *linearly ordered sets* and *chains*.

A *well-ordered set* is a chain such that every non-empty subset has a least element.

Quiz

1. Is the set of integers with the usual numerical ordering a well-ordered set?
2. Given a set A and its *powerset* (set of all subsets) $P(A)$, is $(P(A), \subseteq)$ a poset? A chain?
3. For $A = \{a, b, c\}$ (a set of three letters), find a well-ordered subset of $(P(A), \subseteq)$.

Answers

1. Is the set of integers with the usual numerical ordering a well-ordered set?
   No. The set itself is a chain with no least element.

2. Given a set A and its *powerset* (set of all subsets) $P(A)$, is $(P(A), \subseteq)$ a poset? A chain?
   It is a poset, but not a chain.

3. For $A = \{a, b, c\}$ (a set of three letters), find a well-ordered subset of $(P(A), \subseteq)$.
   One possibility: $\{\emptyset, \{a\}, \{a, b\}, \{a, b, c\}\}$

Pertinent Example: Prefix Orders

Let A be a type (a set of values).
Let A^{**} be the set of all finite and infinite sequences of elements of A, including the empty sequence $\perp$ (bottom).

Let $\sqsubseteq$ be a binary relation on A^{**} such that $a \sqsubseteq b$ if a is a *prefix* of b. That is, for all n in N such that $a(n)$ is defined, then $b(n)$ is defined and $a(n) = b(n)$.

This is called a *prefix order.*

During execution, any output of a PN actor is a well-ordered subset of $(A^{**}, \sqsubseteq)$.

Join (Least Upper Bound)

An *upper bound* of a subset $B \subseteq A$ of a poset $(A, \leq)$ is an element $a \in A$ such that for all $b \in B$ we have $b \leq a$.

A *least upper bound* (LUB) or *join* of B is an upper bound a such that for all other upper bounds a' we have $a \leq a'$.

The *join* of B is written $\vee B$.

When the join of B exists, then B is said to be *joinable*.

Meet (Greatest Lower Bound)

A *lower bound* of a subset $B \subseteq A$ of a poset $(A, \leq)$ is an element $a \in A$ such that for all $b \in B$ we have $a \leq b$.

A *greatest lower bound* (GLB) or *meet* of B is a lower bound a such that for all other lower bounds a' we have $a' \leq a$.

The *meet* of B is written $\wedge B$.

When the meet of B exists and is in B, then B is said to be *well-founded*. In this case, we call $\wedge B$ the "bottom" of B and often write it $\perp$.

Example of Join and Meet

Example: Given a set A and its *powerset* (set of all subsets) $P(A)$, then $(P(A), \subseteq)$ is a poset. For any $B \subseteq P(A)$, we have

$\vee B = \cup B$ (the union of the subsets) and

$\wedge B = \cap B$ (the intersection of the subsets)

Complete Partial Order

A *complete partial order* (CPO) is a well-founded partially ordered set where every chain is joinable.

Example: $(N, \leq)$ is not a CPO.

Example: $(N \cup \{\infty\}, \leq)$ is a CPO.

Example: $(A^{**}, \sqsubseteq)$ is a CPO.

- The bottom element is the empty sequence.
- The join of any infinite chain is an infinite sequence.

Example: $(A^{*}, \sqsubseteq)$ is not a CPO.

- A^{*} is the set of all finite sequences.

Monotonic (Order Preserving) Functions

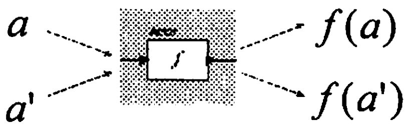
Let $(A, \leq)$ a d $(B, \leq)$ be posets.

A function f $A \to B$ is called *monotonic* if

$$a \leq a' \;\Rightarrow\; f(a) \leq f(a')$$

Example: P actors are monotonic with the prefix order.

PN Actors are Monotonic Functions on a CPO



Set of signal s with the prefix order is a CPO.

Actors are m onotonic functions:

$$a \sqsubseteq a' \;\Rightarrow\; f(a) \sqsubseteq f(a')$$

This is a tim less *causality* condition.

Example of a Non-Monotonic but Functional Actor

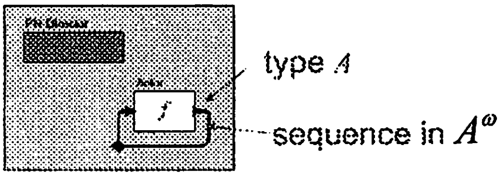
Unfair merge $f: A \times A \to A$ where $(A, \sqsubseteq)$ is a poset

$$f(a,b) = \begin{cases} a & \text{if } a \text{ is infinite} \\ a.b & \text{otherwise} \end{cases}$$

where the period indicates concatenation.

Exercise: show that this function is not monotonic under the prefix order.

Fixed Point Semantics

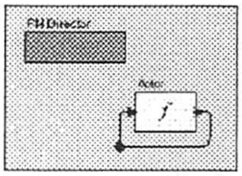


- Start with the empty sequence.
- Apply the (monotonic) function.
- Apply the function again to the result.
- Repeat forever.

The result "converges" to the least fixed point.

Fixed Point Theorem 2

Let $f : A \to A$ be a monotonic function on CPO A.
Then f has a least fixed point.



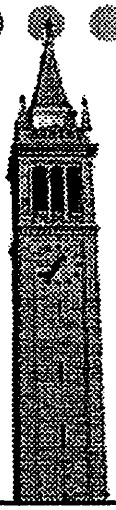
Take the "meaning" or "semantics" of this process network to be that the (one and only) signal in the system is the least fixed point of f.

Conclusion

PN actors that are "causal" are monotonic functions on the CPO of sequences with the prefix order.

The semantics of a PN model with an actor feeding its own output back to its input is the least fixed point of the actor function.

Next time: Give a procedure for finding the fixed point and generalize to arbitrary process networks.

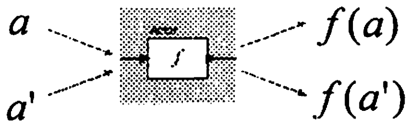


Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall, 2004

Lecture 7: Continuous Functions and PN Composition

PN Actors are Monotonic Functions on a CPO



Set of signals with the prefix order is a CPO.

Actors are *monotonic* functions:

$$a \sqsubseteq a' \Rightarrow f(a) \sqsubseteq f(a')$$

This is a timeless *causality* condition.

Continuous (Limit Preserving) Functions

Let $(A, \leq)$ and $(B, \leq)$ be CPOs.

A function $f: A \to B$ is called *continuous* if for all chains $C \subseteq A$,

$$f(\vee C) = \vee \hat{f}(C)$$

Notation: Given a function $f: A \to B$, define a new function $\hat{f}: P(A) \to P(B)$, where for any $C \subseteq A$,

$$\hat{f}(C) = \{b \in B \mid \exists c \in C \text{ s.t. } f(c) = b\}$$

Continuous vs. Monotonic

Fact: Every continuous function is monotonic.

- **Easy to show (consider chains of length 2)**

Fact: If every chain in A is finite, then every monotonic function $f: A \to B$ is continuous.

But: If A has infinite chains, the monotonic does not imply continuous.

Counterexample Showing that Monotonic Does Not Imply Continuous

Let $A = (N \cup \{\infty\}, \leq)$ (a CPO).
Let $f \colon A \to A$ be given by

$$f(a) = \begin{cases} 1 & \text{if } a \text{ is finite} \\ 2 & \text{otherwise} \end{cases}$$

This function is obviously monotonic. But it is not continuous. To see that, let $C = \{1, 2, 3, \ldots\}$, and note that $\vee C = \infty$. Hence,

$$f(\vee C) = 2$$
$$\vee f(C) = 1$$

which are not equal.

Intuition

Under the prefix order, for any monotonic functions that is not continuous, there is a continuous function that yields the same result for every finite input.

For practical purposes, we can assume that any monotonic function is continuous, because the only exceptions will be functions that wait for infinite input before producing output.

Fixed Point Theorem 1

Let $(A, \leq)$ be a CPO with bottom $\perp$
Let $f: A \to A$ be a monotonic function
Let $C = \{f^n(\perp), n \in N\}$

- If f is continuous, then $\vee C = f(\vee C)$
- If $\vee C = f(\vee C)$, then $\vee C$ is the *least* fixed point of f

Intuition: The least fixed point of a continuous function is obtained by applying the function first to the empty sequence, then to the result, then to that result, etc.

Proof (Continuous Part)

Note that C is a chain in a CPO (show this) and hence has a LUB $\vee C$.

Let $C' = C \cup \{\perp\}$ and note that $\vee C = \vee C'$.
Note further that $\hat{f}(C') = C$ and hence $\vee \hat{f}(C') = \vee C$
By continuity, $\vee \hat{f}(C') = f(\vee C') = f(\vee C)$
Hence $\vee C = f(\vee C)$

QED ($\vee C$ is a fixed point of f)

Proof (Least Fixed Point Part)

NOTE: This part does not require continuity.

Let a be another fixed point: $f(a) = a$

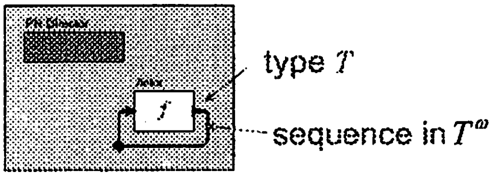
Show that $\vee C$ is the least fixed point: $\vee C \leq a$

Since f is monotonic:

$$\bot \leq a$$
$$f(\bot) \leq f(a) = a$$
$$\dots$$
$$f^k(\bot) \leq f^k(a) = a$$

So a is an upper bound of the chain C, hence $\vee C \leq a$.

Fixed Point Semantics



- Start with the empty sequence.
- Apply the (continuous) function.
- Apply the function again to the result.
- Repeat forever.

The result "converges" to the least fixed point.

Fixed Point Theorem 2

Let $f: A \to A$ be a monotonic function on CPO $(A, \leq)$. Then f has a least fixed point.

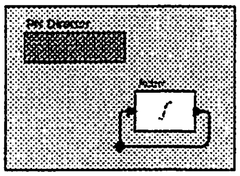
Intuition: If a function is monotonic (but not continuous), then it has a least fixed point, but the execution procedure of starting with the empty sequence and iterating may not converge to that fixed point.

This is obvious, since monotonic but not continuous means it waits forever to produce output.

Example 1: Identity Function

Let $A = T^{**}$ and $f: A \to A$ be such that $\forall a \in A$, $f(a) = a$.
This is obviously continuous (and hence monotonic) under the prefix order.

Then the model below has many fixed points, but only one least fixed point (the empty sequence).

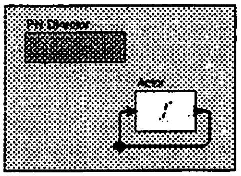


Example 2: Delay Function

Let $A = T^{**}$ and $f: A \to A$ be such that $\forall a \in A$, $f(a) = t.a$ (concatenation), where $t \in T$.

This is obviously continuous (and hence monotonic) under the prefix order.

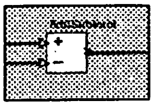
Then the model below has only one fixed point, the infinite sequence $(t, t, t, \ldots)$



Why is this called a "delay?" In the feedback loop, it functions like Const.

Multiple Inputs or Outputs

What about actors with multiple inputs or outputs?



Cartesian Products of Posets

Let $(A, \leq)$ and $(B, \leq)$ be CPOs.
Then $A \times B$ is a CPO under the pointwise order.

Pointwise order: $(a_1, b_1) \leq (a_2, b_2) \Leftrightarrow a_1 \leq a_2$ and $b_1 \leq b_2$

Contrast with lexicographic order:
$(a_1, b_1) \leq (a_2, b_2) \Leftrightarrow a_1 \leq a_2$ or $a_1 = a_2$ and $b_1 \leq b_2$

Exercise (homework): Determine whether $A \times B$ is a CPO under the lexicographic order.

More Cartesian Products and Projections

Let $(A, \leq)$ be a CPO.
Let A^n denote $A \times A \times \ldots \times A$, n times

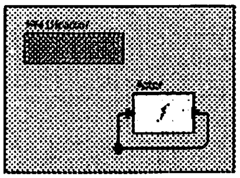
Then $(A^n, \leq)$ is a CPO under the pointwise order for any natural number n.

For any $a = \{a_1, \ldots , a_n\} \in A^n$ and $i \in \{1, \ldots n\}$, define the *projection* on *i* to be:

$$\pi_i(a) = \{a_1, \ldots , a_n\}$$

Composing Actors

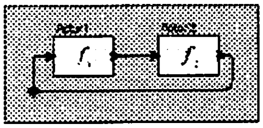
So far, our theory applies only to a single actor in a feedback loop:



What about more interesting models?

Cascade Composition

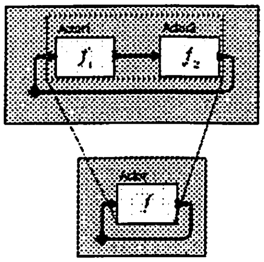
Consider cascade composition: .



If $f_1 : A \to B$ and $f_2 : B \to C$ are monotonic (or continuous) functions on CPOs A, B, C, then $f_1 \circ f_2$ is monotonic (or continuous) (show this).

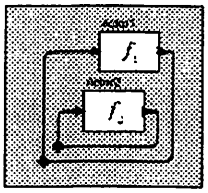
Hence, the execution procedure works for cascade composition.

Cascade Composition Reduces to the Previous Case



Parallel Composition

Consider parallel composition:



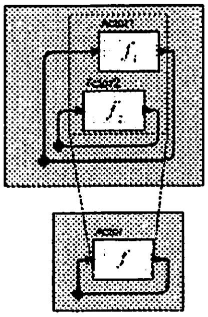
If $f_1 : A \to B$ and $f_2 : C \to D$ are monotonic (or continuous) functions on CPOs A, B, C, D, then $f_1 \times f_2$ is monotonic (or continuous) on CPOs $A \times B, C \times D$.

Cartesian Products of Functions

If $f_1 : A \to B$ and $f_2 : C \to D$ then the Cartesian product is $f_1 \times f_2 : A \times B \to C \times D$.

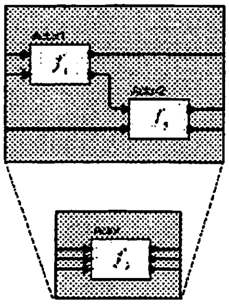
If A, B, C, D are CPOs then so are $A \times B$ and $C \times D$ under the pointwise order.

Parallel Composition Reduces to the Previous Case



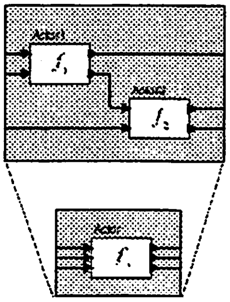
More Interesting Feedback Compositions

Assuming f_1 and f_2 are monotonic, is f_3 monotonic?
Assuming f_1 and f_2 are continuous, is f_3 continuous?
Assuming f_1 and f_2 are sequential, is f_3 sequential?

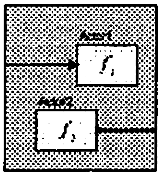


More Interesting Feedback Compositions

Assuming f_1 and f_2 are monotonic, is f_3 monotonic? *yes*
Assuming f_1 and f_2 are continuous, is f_3 continuous? *yes*
Assuming f_1 and f_2 are sequential, is f_3 sequential? *no*



Source and Sink Actors

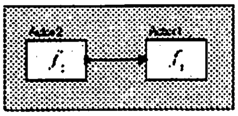


Consider Actor1. Its function is $f_1 \colon A^1 \to A^0$ where A^0 is a *singleton set* (a set with one element). Such a function is always monotonic (and continuous, and sequential).

Consider Actor2. Its function is $f_1 \colon A^0 \to A^1$. Such a function is again always monotonic (and continuous, and sequential). In fact, the function can only yield one possible output sequence, since its domain has size 1.

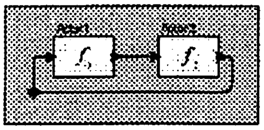
Composing Sources and Sinks

What about the following interconnection?

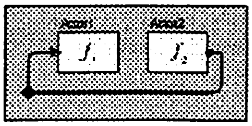


Composing Sources and Sinks

Recall cascade composition:



Reorganized, this looks like cascade composition:



The codomain of f_1 and domain of f_2 are singleton sets, so there is no need to show any signal.

Complicated Compositions

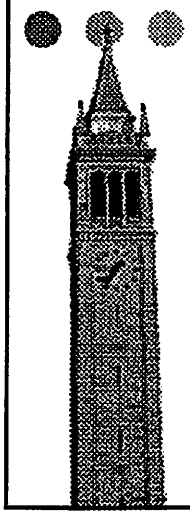
Simple procedure:

- Bring all n signals out as outputs.
- Feed back all n signals as inputs.
- The resulting $f: A^n \to A^n$ will be continuous if the component functions are continuous.
- Hence the model will have a least fixed point that can be found by starting with all sequences being empty and repeatedly applying the function f.

Conclusion

Continuous functions compose, sequential functions do not.

Implementing sequential functions is easy (blocking reads). Implementing continuous functions can be hard.

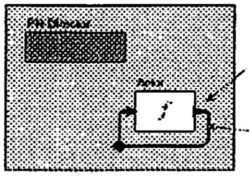


Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall. 2004

Lecture 8: Execution of Process Networks

Semantics of a PN Model is the Least Fixed Point of a Monotonic Function



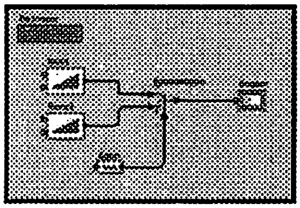
type A

sequence in A^ω

- Chain: $C = \{f(\perp), f(f(\perp)), \dots, f^n(\perp), \dots\}$
- Continuity: $f(\vee C) = \vee \hat{f}(C)$

Limits

Applying This In Practice



- Model is a composition of actors
- Each actor implements a monotonic function
- The composition is a monotonic function
- All signals are part of the "feeback"
- Execution approximates the semantics by
  - starting with empty sequences on all signals
  - allowing actors to react to inputs and build output signals
- Actors execute in their own thread.
- Reads of empty inputs block.

Kahn-MacQueen Blocking Reads

Following Kahn-MacQueen [1977], actors are threads that implement *blocking reads*, which means that when they attempt to read from an empty input, the thread stalls.

- This restricts expressiveness more than continuity
- This still leaves open the question of thread scheduling

Blocking Reads Realize Sequential Functions [Vuillemin]

Let $f: A^n \to A^m$ be an n input, m output function.

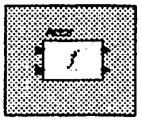
Then f is *sequential* if it is continuous and for any $a, b \in A^n$ where $a \le b$ there exists an $i \in \{1, \ldots n\}$, where:

$$\pi_i(a) = \pi_i(b) \Rightarrow f(a) = f(b)$$

Intuitively: At all times during an execution, there is an input channel that blocks further output. This is the Kahn-MacQueen blocking read!

Continuous Function that is not Sequential

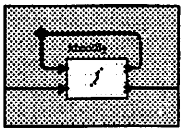
Two input identity function is not sequential:



Let $f: A^2 \to A^2$ such that for all $a \in A^2$, $f(a) = a$.
Then f is not sequential.

Cannot Implement the Two-Input Identity with Blocking Reads

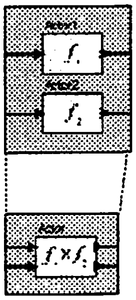
Consider the following connection:



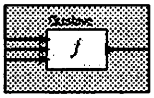
This has a well-defined behavior, but an implementation of the two-input identity with blocking reads will fail to find that behavior.

Sequential Functions do not Compose

If $f_1 : A \to B$ and $f_2 : C \to D$ are sequential then $f_1 \times f_2$ may or may not be sequential. Simple example: suppose f_1 and f_2 are identity functions in the following:



Gustave Function
Non Sequential but Continuous



Let $A = T^{**}$ where $T = \{t, f\}$.

Let $f : A^3 \to N^{**}$ such that for all $a \in A^3$,

$$f(a) = \begin{cases} (1) & \text{if } ((t),(f),\bot) \sqsubseteq a \\ (2) & \text{if } (\bot,(t),(f)) \sqsubseteq a \\ (3) & \text{if } ((f),\bot,(t)) \sqsubseteq a \end{cases}$$

This function is continuous but not sequential.

Linear Functions [Erhard]

Function $f : A \to B$ on CPOs is *linear* if for all joinable sets $C \subseteq A$, $\hat{f}(C)$ is joinable and

$$\vee \hat{f}(C) = f(\vee C)$$

Intuition: If two possible inputs can be extended to a common input, then the two corresponding outputs can be extended to the common output.

Fact: Sequential functions are linear.

Fact: Linear functions are continuous (trivial)

Stable Functions [Berry]

Function $f: A \to B$ on CPOs is *stable* if it is continuous and for all joinable sets $C \subseteq A$, $\hat{f}(C)$ is joinable and

$$\wedge \hat{f}(C) = f(\wedge C) \quad \longleftarrow \text{NOTE. meet! not join!}$$

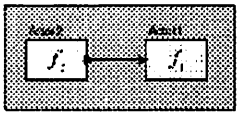
Intuition: If two possible inputs do not contain contradictory information, then neither will the two corresponding outputs.

Fact: Sequential functions are stable.

Practical Questions

- When a process suspends, how should you decide which process to activate next?
- If a process does not (voluntarily) suspend, when should you suspend it?
- How can you ensure "fairness"? In fact, what does "fairness" mean?
  - All inputs to a process are eventually consumed?
  - All outputs that a process can produce are eventually produced?
  - All processes are given equal opportunity to run? What does "equal opportunity" mean?

Consider a Simple Example



How can we prevent Actor2 from never suspending, thus starving Actor1 and causing memory usage to explode?

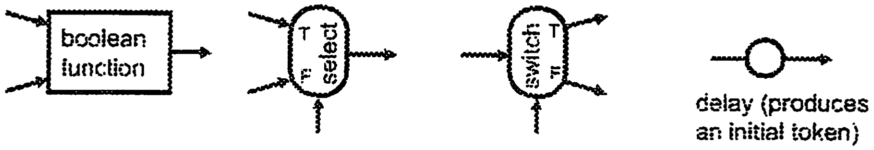
How can we prevent buffers from growing infinitely (data is produced a higher rate than it is consumed)?

Naïve answers:

- Fair execution: Give both actors equal time slices
- Data-driven execution: When Actor2 produces, execute Actor1
- Demand-driven execution: When Actor1 needs, execute Actor2
- Bound the buffer between them and implement blocking writes.

Undecidability [Buck, 1993]

Given the following four actors, and boolean data types on the ports, you can construct a universal Turing machine:



Consequence: The following questions are undecidable:

- Will a PN model deadlock?
- Can a PN model be executed in bounded memory?

Consequences

It is undecidable whether a PN model can execute in bounded memory, so no terminating algorithm can identify (for all PN models) bounds that are safe to use on the channels.

A PN model *terminates* if every signal is finite in the least fixed point semantics.

It is undecidable whether a PN model terminates.

A Practical Policy

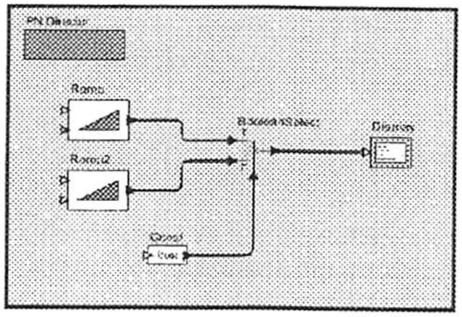
- Define a *correct execution* to be any execution for which after any finite time every signal is a prefix of the LUB signal given by the semantics.
- Define a *useful execution* to be a correct execution that satisfies the following criteria:
  1. **For every non-terminating PN model, after any finite time, a useful execution will extend at least one signal in finite (additional) time.**
  2. **If a correct execution satisfying criterion (1) exists that executes with bounded buffers, then a useful execution will execute with bounded buffers.**

Parks' Strategy [Parks, 1995]

- Start with an arbitrary bound on the capacity of all buffers.
- Execute with both blocking reads and blocking writes (which prevent buffers from overflowing).
- If deadlock occurs and at least one actor is blocked on a write, increase the capacity of at least one buffer to unblock at least one write.
- Continue executing, repeatedly checking for deadlock.

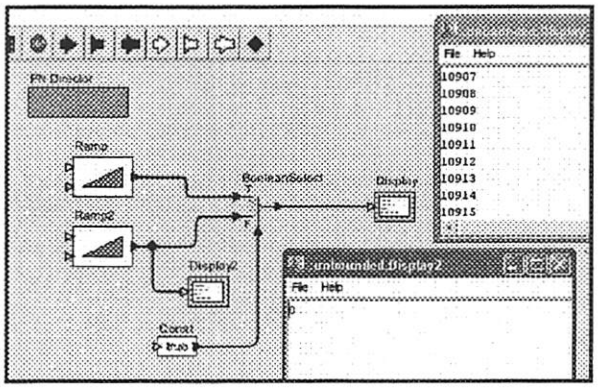
This is the strategy implemented in the PN domain in Ptolemy II. Notice that it "solves" two undecidable problems, but does so in infinite time.

Questions 1 & 2: (from lecture 4)
Is "Fair" Thread Scheduling a Good Idea?



A "useful execution" will allow Ramp2 to produce only finite output.

Question 3: (from lecture 4)
When are Outputs Required?

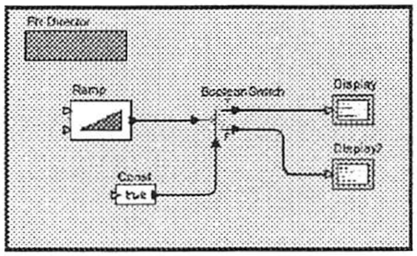


```
10907
10908
10909
10910
10911
10912
10913
10914
10915
```

```
0
```

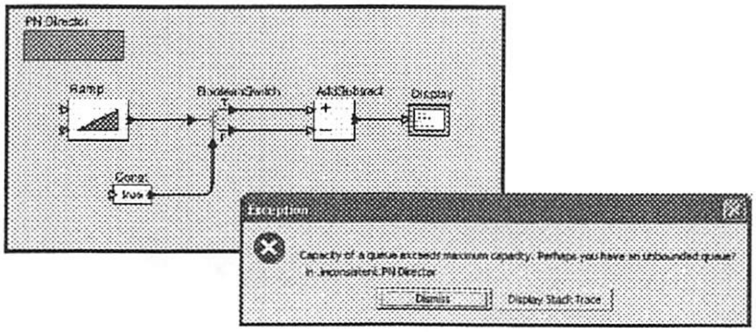
The "useful execution" is not changed by the mere act of observing a signal.

Question 4: (from lecture 4)
Is "Demand-Driven" Execution a Good Idea?



A useful execution of this is not frustrated by the lack of data to Display2.

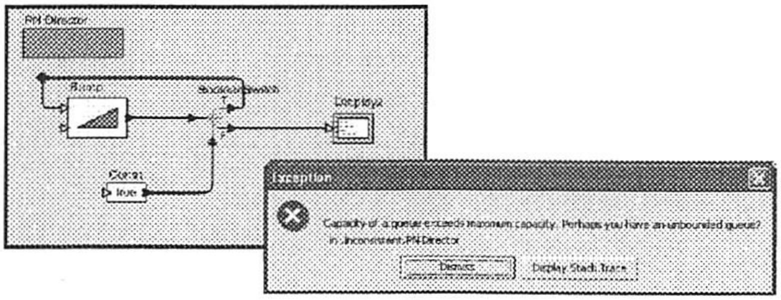
Question 5: (from lecture 4)
What is the "Correct" Execution of This Model?



Capacity of a queue exceeds maximum capacity. Perhaps you have an unbounded queue?
in .inconsistent.PN Director

The PN Director optionally allows you to specify an overall bound on buffer sizes. This is a debugging tool, not a change in the semantics!

Question 6:
What is the Correct Behavior of this Model?

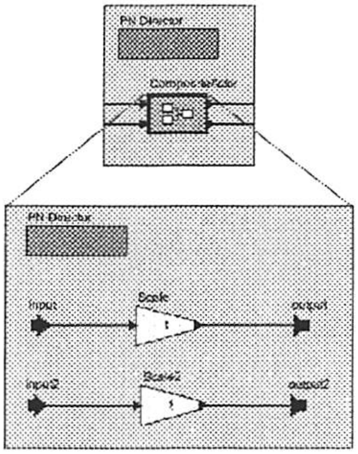


Capacity of a queue exceeds maximum capacity. Perhaps you have an unbounded queue?
in .inconsistent.PN Director

A correct behavior of this model (like the previous one) requires unbounded buffers.

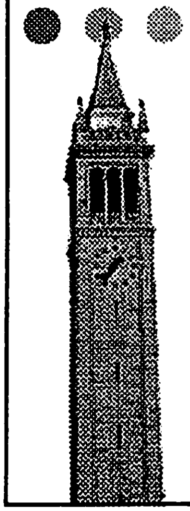
A Deeper Question

How can process networks be composed?



Conclusion

- Processes with blocking reads realize sequential functions, a subset of monotonic functions.
- Sequential functions are (regrettably) not compositional.
- Deadlock and memory requires are undecidable for PN.
- Correct and useful executions can be practically achieved despite this fact using Parks' strategy.
- Compositionality questions still have to be addressed.



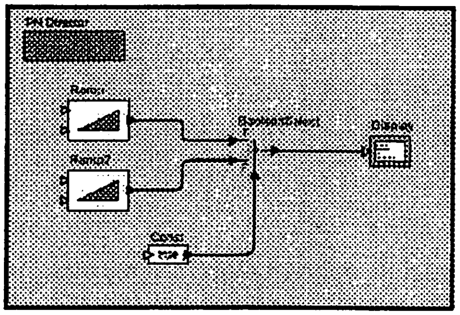
Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n -- Advanced Topics in Systems Theory
Fall, 2004

Lecture 9: Convergence and Introduction to Synchronous Models

---

The Convergence Question



- *Correct execution:* after any finite time every signal is a prefix of the LUB signal given by the semantics.
- *Useful execution:* a correct execution that:
  1. Does not stop if at least one signal has not reach the LUB.
  2. Executes with bounded buffers if this is possible.

The Question: Does this execution "converge" to the LUB?

Convergence in the Reals

Consider a sequence of real numbers:

$$s : N \rightarrow \Re$$

This sequence is said to *converge* to a real number a if for all open sets A containing a there exists an integer n such that for all $m > n$ the following holds:

$$s(m) \in A$$

Standard Topology in the Reals

An *open neighborhood* around a in the reals is

$$\{ x \in \Re \mid a - \varepsilon < x < a + \varepsilon \}$$

for some positive real number ε.

An *open set* A in the reals is a subset of $\Re$ such that for all $a \in A$, there is an open neighborhood around a that is a subset of A.

The collection of open sets in the reals is called a *topology*.

Topology

Let X be any set. A collection τ of subsets of X is called a *topology* if:

- X and $\varnothing$ are members of τ
- The intersection of any two members of τ is in τ
- The union of any family of members of τ is in τ

For any topology τ, the members of τ are called its open sets.

The set of open sets in the reals is a *topology*.

Scott Topology

Consider a set T and the set T^{**} of all finite and infinite sequences of elements of T.

Given a *finite* sequence $t \in T^{**}$, an *open neighborhood* around t is the set

$$N_t = \{\, t' \in T \mid t' \sqsubseteq t \}$$

Let τ be the collection of all sets that arbitrary unions of open neighborhoods.

Fact: τ is a topology.

Limit of a Sequence of Sequences (Convergence in the Scott Topology)

Consider a sequence of sequences:

$$s : N \to T^{**}$$

This sequence is said to *converge* to a sequence a if for all open sets A containing a there exists an integer n such that for all $m > n$ the following holds:

$$s(m) \in A$$

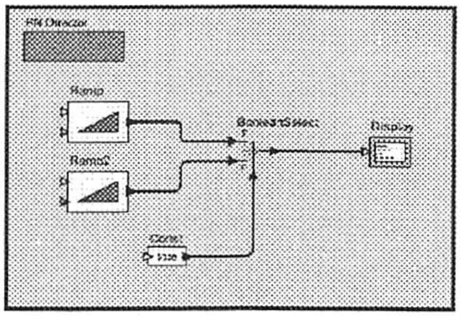
Intuition: For any finite prefix $p \sqsubseteq a$, the sequences in s eventually all have prefix p .

Consequences for Process Networks

- "Correct" executions of process networks do not necessarily converge to the LUB semantics.
- This is because "correct" executions allow any signal to be evaluated only to a finite prefix of the LUB semantics.
- But if leaving the execution at a finite prefix were "incorrect," then it would be incorrect for Ptolemy II to stop the execution when you push the stop button.

  This would be counterintuitive.

Convergent Execution vs. Correct Execution



- A "convergent" execution of the above model is impossible with finite memory.
- A "correct" and "useful" execution is possible and practical.

Which do you prefer?

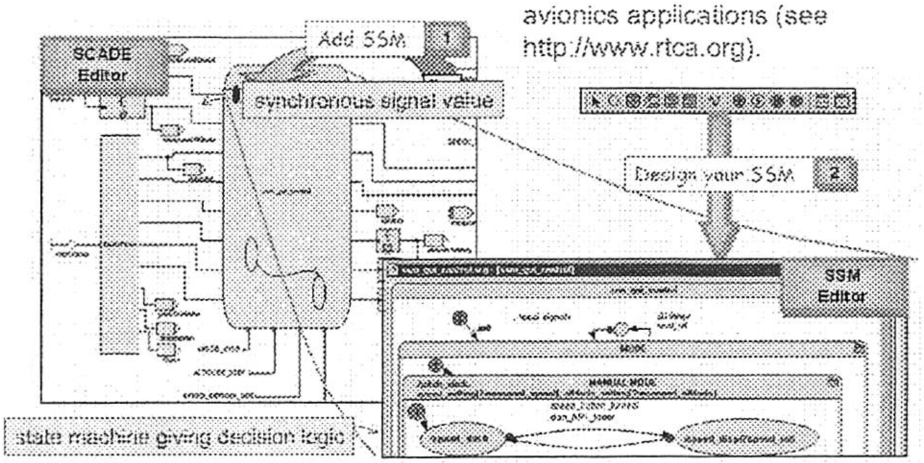
Synchronous Languages

- Esterel
- Lustre
- SCADE (visual editor for Lustre)
- Signal
- Statecharts (some variants)
- Ptolemy II SR domain

The model of computation is called *synchronous reactive* (SR). It has strong formal properties (many key questions are decidable).

Lustre/SCADE

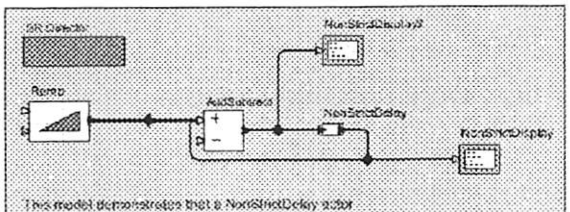
The SCADE tool has a code generator that produces C or ADA code that is compliant with the DO-178B Level A standard, which allows it to be used in critical avionics applications (see http://www.rtca.org).



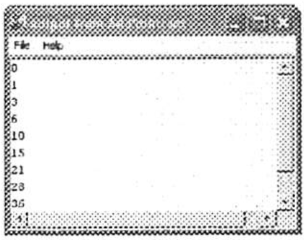
from http://www.esterel-technologies.com/

SR Domain in Ptolemy II

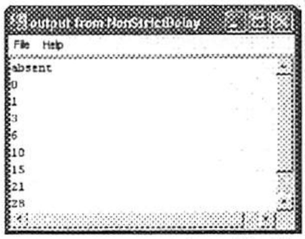
At each tick of a global "clock," every signal has a value or is absent.



This model demonstrates that a NonStrictDelay actor breaks a feedback loop in a SR model.



```
0
1
3
6
10
15
21
28
36
```



output from NonStrictDelay

```
absent
0
1
3
6
10
15
21
28
```

The job of the SR director is to find the value at each tick.

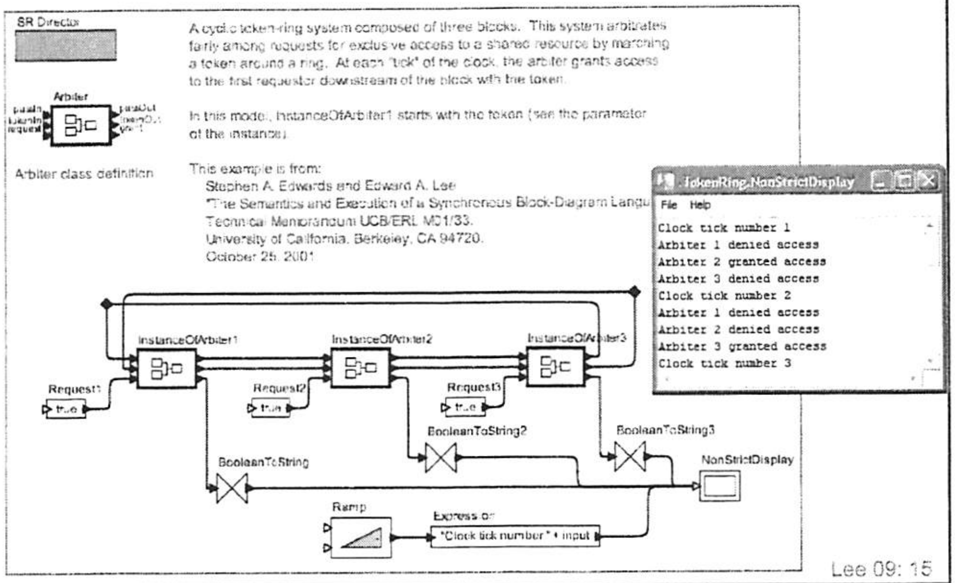
The Synchronous Abstraction

- "Model time" is discrete: Countable ticks of a clock.
- WRT model time, computation does not take time.
- All actors execute "simultaneously" and "instantaneously" (WRT to model time).
- There is an obviously appealing mapping onto real time, where the real time between the ticks of the clock is constant. Good for specifying periodic real-time tasks.

Properties

- Buffer memory is bounded (obviously).
- Hence the model of computation is not Turing complete.
  - ... or bounded memory would be undecidable ...
- Causality loops are possible, where at a tick, the value of one or more signals cannot be determined.

Practical Application – Token Ring Arbitration



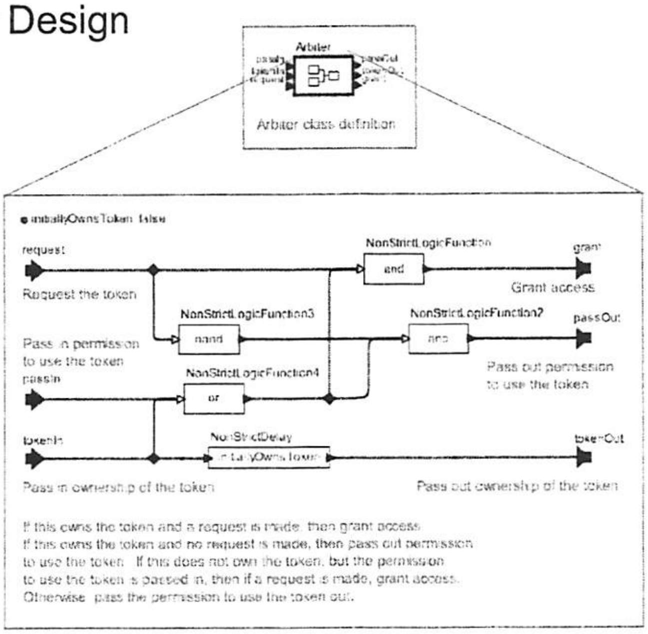
A cyclic token-ring system composed of three blocks. This system arbitrates fairly among requests for exclusive access to a shared resource by marching a token around a ring. At each "tick" of the clock, the arbiter grants access to the first requestor downstream of the block with the token.

In this model, InstanceOfArbiter1 starts with the token (see the parameter of the instance).

This example is from:
Stephen A. Edwards and Edward A. Lee
"The Semantics and Execution of a Synchronous Block-Diagram Langu
Technical Memorandum UCB/ERL M01/33,
University of California, Berkeley, CA 94720,
October 25, 2001

```
Clock tick number 1
Arbiter 1 denied access
Arbiter 2 granted access
Arbiter 3 denied access
Clock tick number 2
Arbiter 1 denied access
Arbiter 2 denied access
Arbiter 3 granted access
Clock tick number 3
```

Arbiter Design

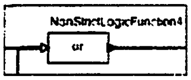


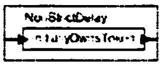
If this owns the token and a request is made, then grant access.
If this owns the token and no request is made, then pass out permission
to use the token. If this does not own the token, but the permission
to use the token is passed in, then if a request is made, grant access.
Otherwise, pass the permission to use the token out.

Cycles

Note that there are cycles in this graph, so that if you require that all inputs be known to find the output, then this cannot execute.

The "non strict" actors are key: They do not need to know all their inputs to determine the outputs.





Simple Execution Policy

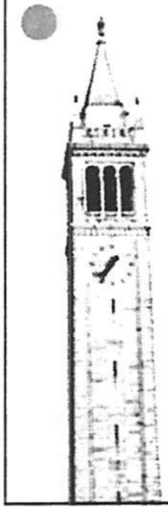
At each tick, start with all signals "unknown." Evaluate non-strict actors and source actors. Then keep evaluating any actors that can be evaluated until all signals become known or until no further progress can be made.

Q: How do we know this will work?

A: Least fixed point semantics.

Conclusion and Open Issues

- "Correct" and "useful" executions of process networks do not necessarily converge to the denotational semantics of the model.
- But insisting on convergence may cause an execution to fail on a finite memory machine that could have executed forever.
- Synchronous/Reactive languages are promising alternatives where termination and boundedness are decidable.



Concurrent Models of Computation for Embedded Software

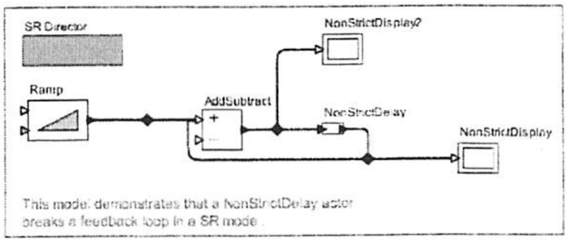
Edward A. Lee

Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 10: Synchronous Reactive Semantics

SR Domain in Ptolemy II

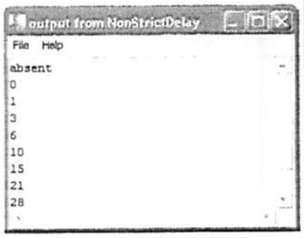
At each tick of a global "clock," every signal has a value or is absent.



This model demonstrates that a NonStrictDelay actor breaks a feedback loop in a SR model.

The job of the SR director is to find the value at each tick.

```
0
1
3
6
10
15
21
28
36
```

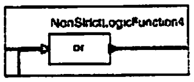


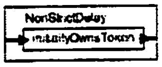
```
absent
0
1
3
6
10
15
21
28
```

Cycles

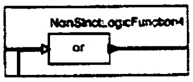
Note that there are cycles in this graph, so that if you require that all inputs be known to find the output, then this cannot execute.

The "non strict" actors are key: They do not need to know all their inputs to determine the outputs.





Non-Strict Logical Or



The non-strict or (often called the "parallel or") can produce a known output even if the input is not completely know. Here is a table showing the output as a function of two inputs:

input 1

input 2

	⊥	ε	F	T
⊥	⊥	⊥	⊥	T
ε	⊥	ε	F	T
F	⊥	F	F	T
T	T	T	T	T

Simple Execution Policy

At each tick, start with all signals "unknown." Evaluate non-strict actors and source actors. Then keep evaluating any actors that can be evaluated until all signals become known or until no further progress can be made.

Q: How do we know this will work?

A: Least fixed point semantics.

The Flat CPO

Consider a set of possible values $T = \{t_1, t_2, \dots\}$. Let

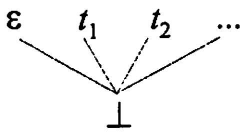
$$A = T \cup \{\perp, \varepsilon\}$$

where $\perp$ represents "unknown" and ε represents "absent."

Let $(A, \leq)$ be a partial order where:

- $\perp \leq \varepsilon$
- for all t in T, $\perp \leq t$
- all other pairs are incomparable

Hasse Diagram for the Flat CPO



Note that this is obviously a CPO
(all chains have a LUB)

All chains have length 2.

Monotonic Functions on This CPO

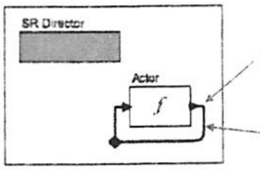
In this CPO, any function $f: A \rightarrow A$ is monotonic if

$$f(\bot) = a \neq \bot \;\Rightarrow\; f(b) = a \text{ for all } b \in A$$

I.e., if the function yields a "known" output when the input is unknown, then it will not change its mind about the output once the input becomes known.

Since all chains are finite, every monotonic function is continuous.

Applying Fixed Point Theorem 1



type T

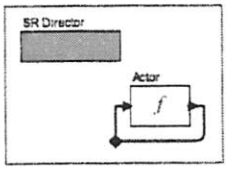
value in $A = T \cup \{ \bot, \varepsilon \}$

At each tick of the clock

- Start with signal value $\bot$
- Evaluate $f(\bot)$
- Evaluate $f(\, f(\bot))$
- Stop when a fixed point is reached

  Unlike PN, a fixed point is always reached in a finite number of steps (one, in this case).

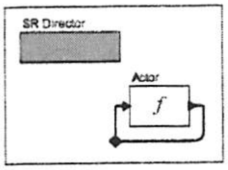
Causality Loops



What is the behavior in the following cases?

- f is the identity function.
- f is the logical NOT function.
- f is the nonstrict delay function with initial value 0.
- f is the nonstrict delay function with no initial value.

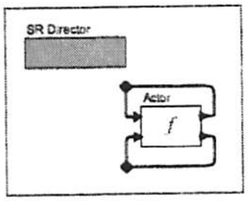
Causality Loops

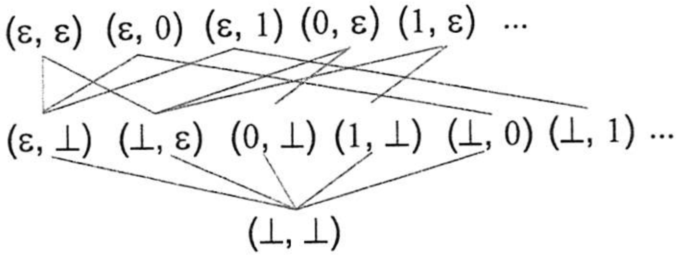


What is the behavior in the following cases?

- f is the identity function: $\perp$
- f is the logical NOT function: $\perp$
- f is the nonstrict delay function with initial value 0: 0
- f is the nonstrict delay function with no initial value: ε

Generalizing to Multiple Signals





$(\varepsilon, \varepsilon)$ $(\varepsilon, 0)$ $(\varepsilon, 1)$ $(0, \varepsilon)$ $(1, \varepsilon)$ ...

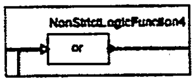
$(\varepsilon, \perp)$ $(\perp, \varepsilon)$ $(0, \perp)$ $(1, \perp)$ $(\perp, 0)$ $(\perp, 1)$ ...

$(\perp, \perp)$

product CPO assuming $T = \{0, 1\}$.

- The Cartesian product of flat CPOs under pointwise ordering is also a CPO.
- All chains are still finite.
- Can now apply to any composition, as done with PN.

Non-Strict Logical Or is Monotonic



The non-strict or is a monotonic function $f: A \times A \rightarrow A$ where $A = \{ \perp, \varepsilon, \mathrm{T}, \mathrm{F} \}$ as can be verified from the truth table:

input 1

input 2

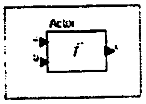
	⊥	ε	F	T
⊥	⊥	⊥	⊥	T
ε	⊥	ε	F	T
F	⊥	F	F	T
T	T	T	T	T

Compositional Reasoning

So far, with both PN and SR, we deal with composite systems by reducing them to a monotonic function of all the signals. An alternative approach is to convert an arbitrary composition to a continuous function.

Example to Use for Compositional Reasoning

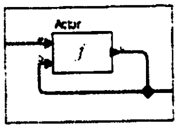
Consider an actor:



Assume $a, b, c \in A$, where A is a CPO.

Assume that the actor function $f: A \times A \to A$ is continuous

Consider the following composition:

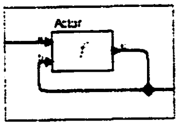


We would like to consider this a function from a to c.

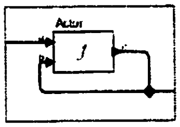
First Option: Currying (Named after Haskell Curry)

Given a function $f: A \times A \to A$, we can alternatively think of this in stages as $f_1 : A \to [A \to A]$, where $[A \to A]$ is the set of all functions from A to A.

For the following example, for each given value of a we get a new function $f_1(a)$ for which we can find the least fixed point. That least fixed point is the value of c.



Example: Non-Strict OR



Suppose f is a non-strict logical OR function. Then:

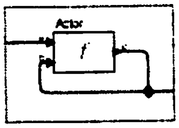
- If $a = \textit{true}$, then the resulting function $f_1(a)$ always returns $\textit{true}$, for all values of the input b.

  In this case, the least fixed point yields $c = \textit{true}$.

- If $a = \textit{false}$, then the resulting function $f_1(a)$ always returns b, for all values of the input b.

  In this case, the least fixed point yields $c = \bot$.

Second Option: Lifting (Named after Heavy Lifting)



Given a function $f : A \times A \to A$. we are looking for a function $g : A \to A$ such that

$$c = g(a)$$

In the model we have $b = c$ and $c = f(a, b)$ so

$$g(a) = f(a, g(a))$$

This looks like a fixed point problem, but the "unknown" on both sides is g, a function not a value. If we can find the function g that satisfies this equation, then we can use it always to calculate c given a.

Posets of Functions

Suppose $(A, \leq)$ and $(B, \leq)$ are CPOs.
Consider functions $f, g \in [A \to B]$.
Define the *pointwise order* on these functions to be

$$f \leq g \Leftrightarrow \forall\, a \in A,\ f(a) \leq g(a)$$

Let $X \subset [A \to B]$ be the set of all continuous total functions from A to B.

Theorem: $(X, \leq)$ is a CPO under the pointwise order.

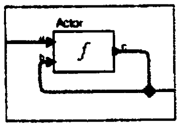
Proof: See handout.

Least Function in the CPO of Functions

Let $X \subset [A \to B]$ be the set of all continuous total functions from A to B. Since X is a CPO, it must have a bottom. The bottom is a function $\perp_X: A \to B$ where for all $a \in A$,

$$\perp_X(a) = \perp_B \in B$$

Consequence of this Theorem



Given a continuous function $f: A \times A \to A$, the function $g: A \to A$ such that

$$c = g(a)$$

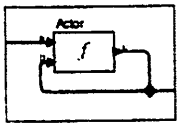
is the least fixed point of a continuous function

$$F: X \to X$$

where $X \subset [A \to A]$ is the set of all continuous total functions from A to A.

We need to now determine the continuous function F.

Consequence of this Theorem (Continued)



We need to find a function that g satisfies:

$$g(a) = f(a, g(a))$$

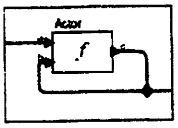
Let $X \subset [A \to A]$ be the set of all continuous total functions from A to A and let F be a continuous function $F: X \to X$.

Then $g \in X$ is the least function such that $F(g) = g$ where all $a \in A$,

$$(F(g))(a) = f(a, g(a))$$

The theorem, with fixed point theorem 1, tells us that F has a least fixed point, and tells us how to find it.

Example: Non-Strict OR



Suppose f is a non-strict logical OR function. Then:

$$(F(g))(a)=\begin{cases} true & \text{if} \quad a = true \\ g(a) & \text{otherwise} \end{cases}$$

The least fixed point of this is the function f given by:

$$g(a)=\begin{cases} true & \text{if} \quad a = true \\ \perp & \text{otherwise} \end{cases}$$

To find this, start with $F(\perp)$, then find $F(F(\perp))$, etc., until you get a fixed point (which happens immediately).

Showing that F is Continuous

Need to show that given a chain of continuous total functions $C = \{ g_1, g_2 \dots \}$ that:

$$F(\vee C) = \vee \hat{F}(C)$$

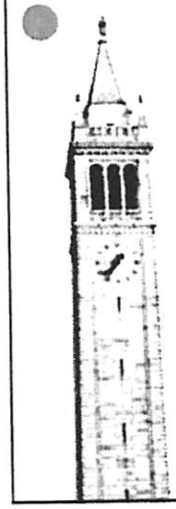
For all $a \in A$:

$$\begin{aligned} (F(\vee C))(a) &= f(a, (\vee C)(a)) \\ &= f(a, \vee\{g_1(a), g_2(a), \dots\}) && \text{because each } g_i \text{ is continuous} \\ &= \vee \hat{f}(a, \{g_1(a), g_2(a), \dots\}) && \text{because } f \text{ is continuous} \\ &= (\vee \hat{F}(C))(a) && \text{QED} \end{aligned}$$

Conclusion and Open Issues

- In SR, fixed point semantics is simpler than in PN because the CPO has only finite chains.
- The fancier techniques of Currying and Lifting can be applied equal well to PN. but we introduce them here because the simpler CPO makes them easier to understand.

- The fixed point semantics of SR talks only about the behavior at a tick of the clock. The behavior across ticks of the clock will require a *clock calculus*.



Concurrent Models of Computation for Embedded Software

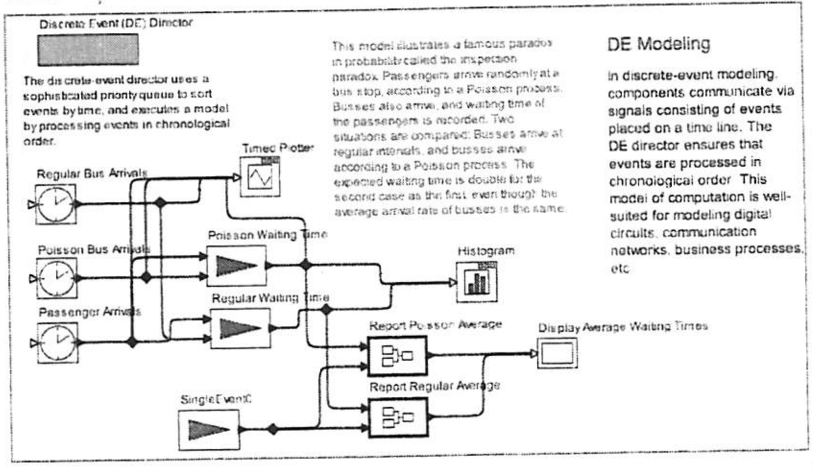
Edward A. Lee

Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 11: Discrete Event Systems

Design of Discrete-Event Models

Example: Model of a transportation system:



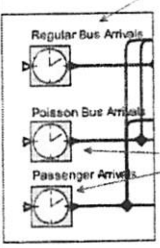
The discrete-event director uses a sophisticated priority queue to sort events by time, and executes a model by processing events in chronological order.

This model illustrates a famous paradox in probability called the inspection paradox. Passengers arrive randomly at a bus stop, according to a Poisson process. Busses also arrive, and waiting time of the passengers is recorded. Two situations are compared: Busses arrive at regular intervals, and busses arrive according to a Poisson process. The expected waiting time is double for the second case as the first, even though the average arrival rate of busses is the same.

DE Modeling

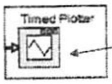
In discrete-event modeling, components communicate via signals consisting of events placed on a time line. The DE director ensures that events are processed in chronological order. This model of computation is well-suited for modeling digital circuits, communication networks, business processes, etc.

Event Sources and Sinks



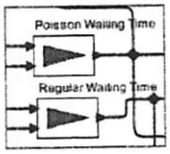
The Clock actor produces events at regular intervals. It can repeat any finite pattern of event values and times.

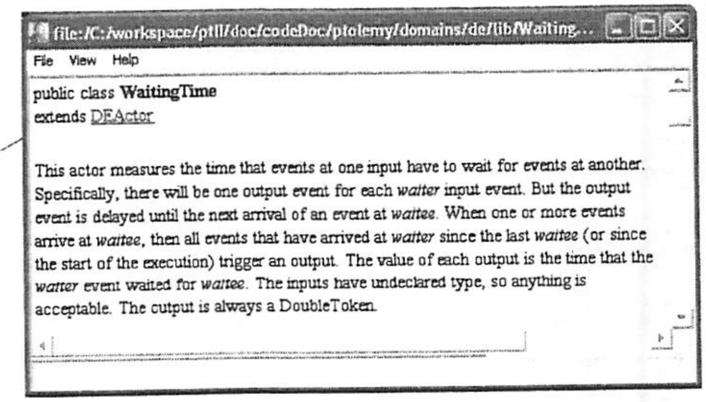
The PoissonClock actor produces events at random intervals. The time between events is given by an exponential random variable. The resulting output random process is called a Poisson process. It has the property that at any time, the expected time until the next event is constant (this is called the *memoryless* property because it makes no difference what events have occurred before that time).



The TimedPlotter actor plots double-valued events as a function of time.

Actors that Use Time



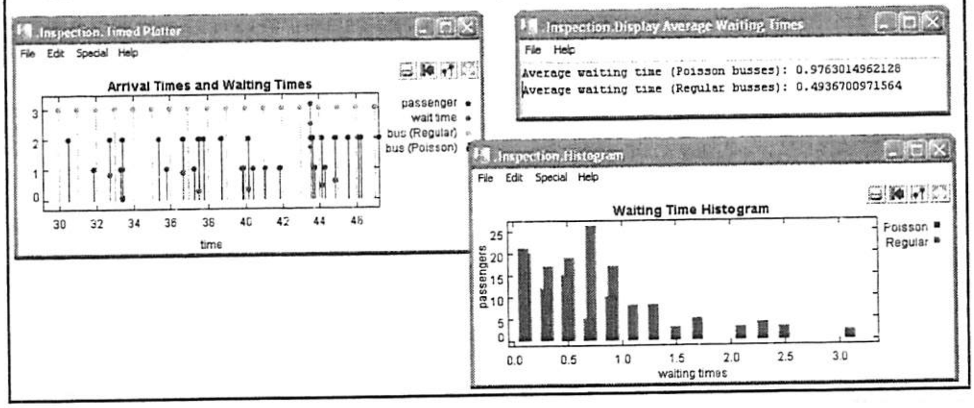


public class **WaitingTime**
extends DEActor

This actor measures the time that events at one input have to wait for events at another. Specifically, there will be one output event for each *waiter* input event. But the output event is delayed until the next arrival of an event at *waitee*. When one or more events arrive at *waitee*, then all events that have arrived at *waiter* since the last *waitee* (or since the start of the execution) trigger an output. The value of each output is the time that the *waiter* event waited for *waitee*. The inputs have undeclared type, so anything is acceptable. The output is always a DoubleToken.

Execution of the Transportation System Model

These displays show that the average time that passengers wait for a bus is smaller if the busses arrive at regular intervals than if they arrive random intervals, even when the average arrival rate is the same. This is called the *inspection paradox*.



```
Average waiting time (Poisson busses): 0.9763014962128
Average waiting time (Regular busses): 0.4936700971564
```

Uses for Discrete-Event Modeling

- Modeling timed systems
  - transportation, commerce, business, finance, social, communication networks, operating systems, wireless networks, …
- Designing digital circuits
  - VHDL, Verilog
- Designing real-time software
  - Music systems (Max, …)

Using DE to Model Real-Time Software

Consider a real-time program on an embedded computer that is connected to two sensors *A* and *B*, each providing a stream of data at a normalized rate of one sample per time unit (exactly). The data from the two sensors is deposited by an interrupt service routine into a register.

Assume a program that looks like this:

```
while(true) {
    wait for new data from A;
    wait a fixed amount of time T;
    observe registered data from B;
    average data from A and B;
}
```

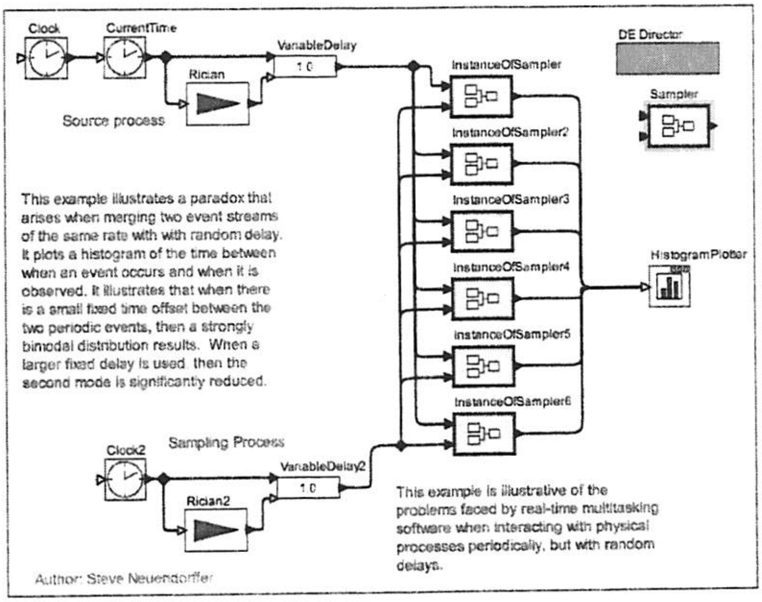
The Design Question

Assume that there are random delays in the software (due to multitasking, interrupt handling, cache management, etc.) for both the above program and the interrupt service routines.

What is the best choice for the value for T?

One way to frame the question: How old is the data from *B* that will be averaged with the data from *A*?

A Model that Measures for Various Values of T



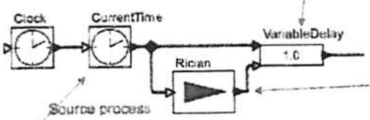
This example illustrates a paradox that arises when merging two event streams of the same rate with with random delay. It plots a histogram of the time between when an event occurs and when it is observed. It illustrates that when there is a small fixed time offset between the two periodic events, then a strongly bimodal distribution results. When a larger fixed delay is used then the second mode is significantly reduced.

This example is illustrative of the problems faced by real-time multitasking software when interacting with physical processes periodically, but with random delays.

Author: Steve Neuendorffer

Modeling Random Delay in Sensor Data

Given an event with time stamp t on the upper input, the VariableDelay actor produces an output with the same value but time stamp $t + t'$, where t' is the value of the most recently seen event on the lower input.

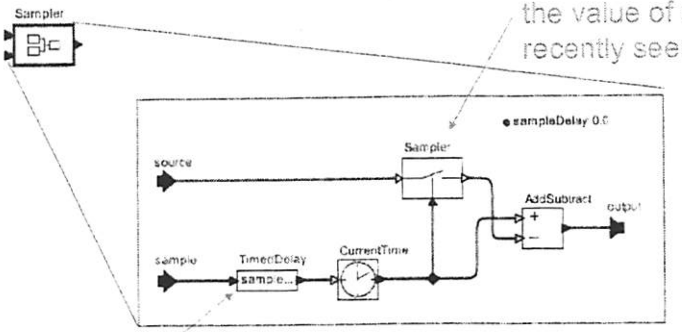


The Rician actor, when triggered, produces an output event with a non-negative random value and with time stamp equal to that of the trigger event.

Given an input event at time t with any value, the CurrentTime actor outputs the double t with time stamp t.

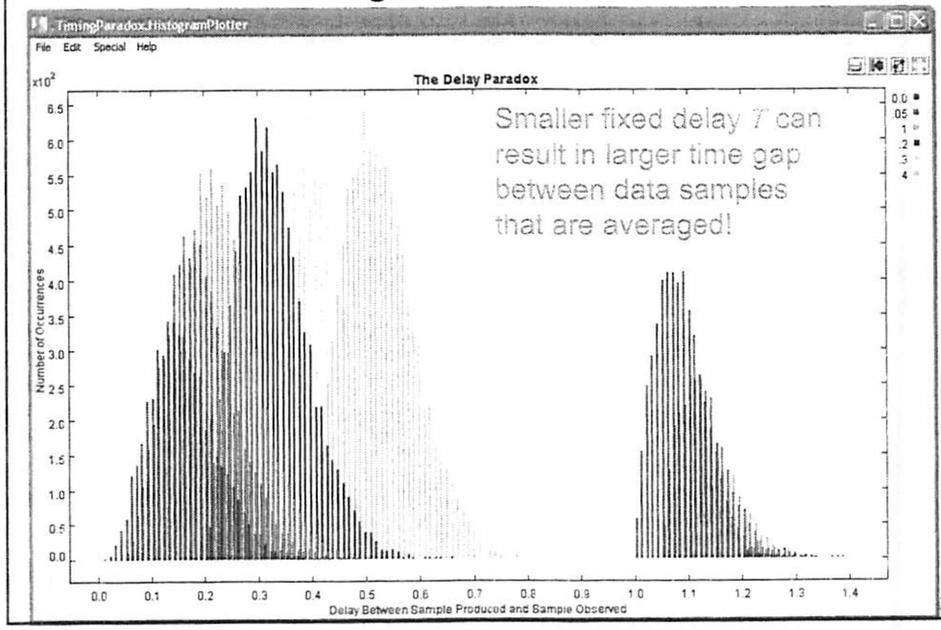
Actor-Oriented Sampler Class

Given a trigger event with time stamp *t* the Sampler actor produces an output event with value equal to the value of the most recently seen input event.



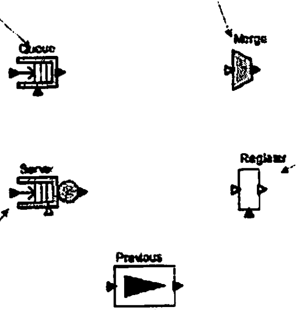
The TimedDelay actor transfers every input event to the output with a fixed increment in the time stamp. Here. the value is sampleDelay, a parameter of the composite actor.

Result of Executing this Model



Smaller fixed delay *T* can result in larger time gap between data samples that are averaged!

Design in DE: Other Useful Actors



When a token is received on the input port, it is stored in the queue. When the trigger port receives a token, the oldest element in the queue is output. If there is no element in the queue when a token is received on the trigger port, then no output is produced.

Merge is deterministic in DE.

Like a register in digital circuits.

Like the Queue, except that a *serviceTime* parameter provides a lower bound on the time between outputs.

When triggered by an input, output the previous input. Is this useful in feedback loops?

Signals in DE

A signal in DE is a partial function $a : T \rightarrow A$, where A is a set of possible event values (a data type and an element indicating "absent"), and T is a totally ordered set of *tags* that represent *time stamps* and ordering of events at the same time stamp.

In a DE model, all signals share the same domain T, but they may have different ranges A.

Executing Discrete Event Systems

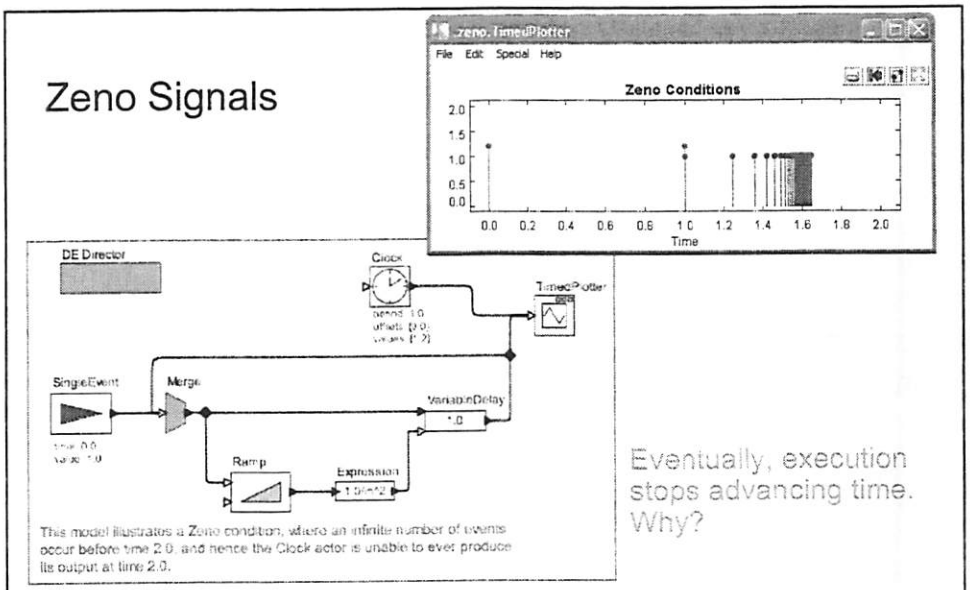
- Maintain an *event queue*, which is an ordered set of events.
- Process the least event in the event queue by sending it to its destination port and firing the actor containing that port.

- Questions:

  How to get fast execution when there are many events in the event queue…

  What to do when there are multiple simultaneous events in the event queue…

Zeno Signals



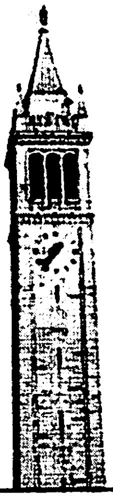
This model illustrates a Zeno condition, where an infinite number of events occur before time 2.0, and hence the Clock actor is unable to ever produce its output at time 2.0.

Eventually, execution stops advancing time. Why?

Note that if the Ramp is set to produce integer outputs, then eventually the output will overflow and become negative, which will cause an exception.

Conclusion and Open Issues

- The discrete-event model of computation is useful for modeling and design of time-based systems.
- In DE models, signals are time-stamped events, and events are processed in chronological order.
- Simultaneous events and Zeno conditions create subtleties that the semantics will have to deal with.



Concurrent Models of Computation for Embedded Software

Edward A. Lee

Professor, UC Berkeley

EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 12: Tags and Discrete Signals

Tags, Time Stamps, and Events

The DE Tag system
- $T = R \times N$, real and natural numbers.
- Lexicographic order using natural ordering of R and N.

This is a totally ordered set.

- *Event:* a pair $e = (t, v) \in T \times V$ where V is a set of *values* and $t = (\tau, n)$ is a tag.
- *Time stamp:* of an event e is $\tau = \pi_1(\pi_1(e))$ (projection)
- *Index*: of an event e is $n = \pi_2(\pi_1(e))$ allowing distinct events with the same time stamp.

Note that events in a signal are totally ordered.

Signals

Signal: a set s of events with distinct tags.

Equivalently: a signal s is a partial function

$$s : T \to V$$

Tag Sets

A signal: $s = \{ e_1, e_2, \dots \} = \{ (t_1, v_1), (t_2, v_2), \dots \}$

Its tags: $\hat{\pi}_1(s) = \{ t_1, t_2, \dots \}$

A *system:* $S = \{ s_1, s_2, \dots \}$ is a set of signals.

Its tags: $\hat{\hat{\pi}}_1(S) = \pi_1(s_1) \cup \pi_1(s_2) \cup \dots$

Discrete Signals

A signal s is *discrete* if there is an *order embedding* from its tag set $\pi_1(s)$ to the integers (under their usual order).

A system S (a set of signals) is *discrete* if there is an *order embedding* from its tag set $\pi_1(s)$ to the integers (under their usual order).

Terminology: Order Embedding

Given two posets A and B, an *order embedding* is a function $f: A \to B$ such that for all $a, a' \in A$,

$$a \le a' \Leftrightarrow f(a) \le f(a')$$

Exercise: Show that if A and B are two posets, and $f: A \to B$ is an order embedding, then f is *one-to-one*.

Examples

1. Suppose we have a signal s whose tag set is

$$\{(\tau, 0) \mid \tau \in R\}$$

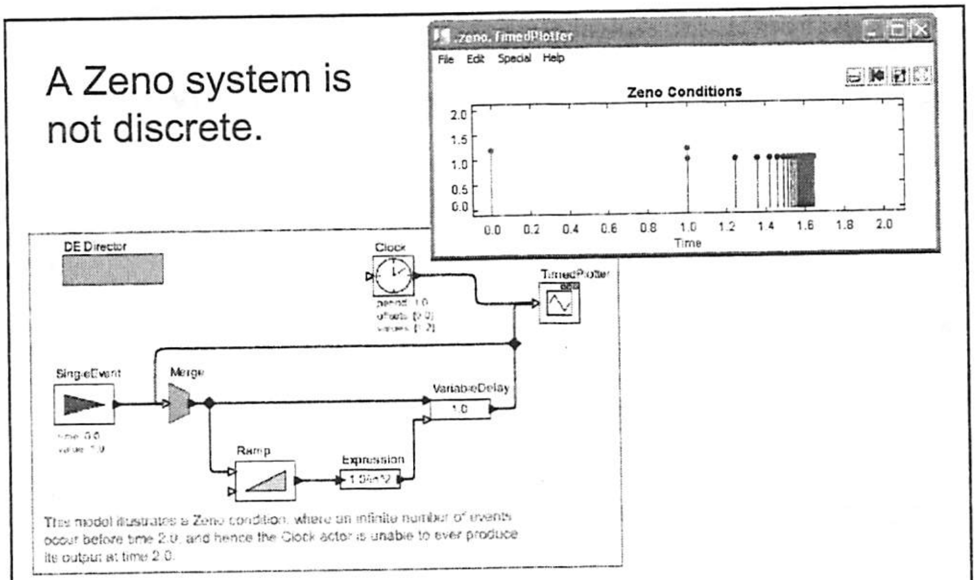
(this is a *continuous-time* signal). This signal is not discrete.

2. Suppose we have a signal s whose tag set is

$$\{(\tau, 0) \mid \tau \in Rationals\}$$

This signal is also not discrete.

A Zeno system is not discrete.

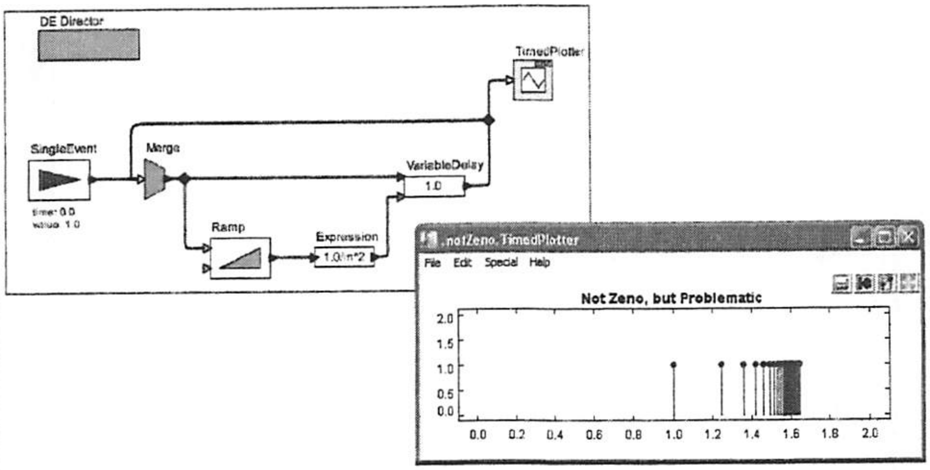


This model illustrates a Zeno condition, where an infinite number of events occur before time 2.0, and hence the Clock actor is unable to ever produce its output at time 2.0.

The tag set here includes $\{0, 1, 2, \ldots\}$ and $\{1, 1.25, 1.36, 1.42, \ldots\}$.

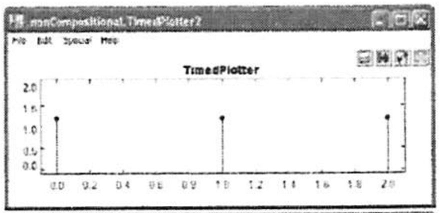
Exercise: Prove that this system is not discrete.

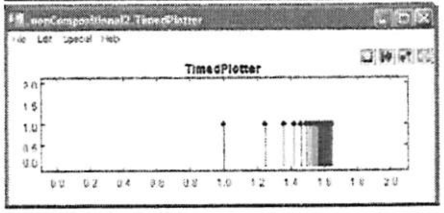
Is the following system discrete?



Discreteness is Not a Compositional Property

Given two discrete signals s, s' it is not necessarily true that $S = \{ s, s' \}$ is a discrete system.

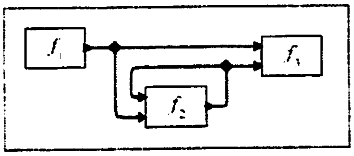




Putting these two signals in the same model creates a Zeno condition.

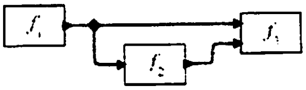
Question 1:

Can we find necessary and/or sufficient conditions to avoid Zeno systems?



Question 2:

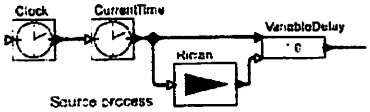
In the following model, if f_2 has no delay, should f_3 see two simultaneous input events with the same tag? Should it react to them at once, or separately?



In Verilog, it is nondeterministic. In VHDL, it sees a sequence of two distinct events separated by "delta time" and reacts twice, once to each input. In the Ptolemy II DE domain, it sees the events together and reacts once.

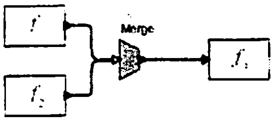
Example

In the following segment of a model, clearly we wish that the VariableDelay see the output of Rician when it processes an input from CurrentTime.



Question 3:

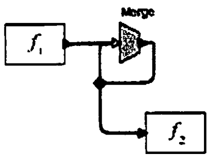
What if the two sources in the following model deliver an event with the same tag? Can the output signal have distinct events with the same tag?



Recall that we require that a signal be a partial function $s: T \to V$, where V is a set of possible event values (a data type), and T is a totally ordered set of *tags*.

Question 4:

What does this mean?



The Merge presumably does not introduce delay, so what is the meaning of this model?

Mathematical Framework

Let the set of all signals be $A = [T \to V]$ where T is a totally ordered set and V is a set of values. Let an actor

be a function $f : A^n \to A^m$. What are the constraints on these functions such that:

- Compositions of actors are determinate.
- Feedback compositions have a meaning.
- We can rule out Zeno behavior.

Can We Re-Use Prefix Orders?

Since tags are totally ordered, signals can be thought of as sequences. Can we just re-use PN semantics?

Signals as Sequences of Events

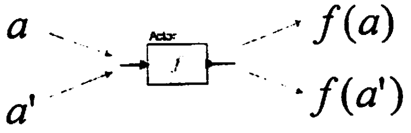
A discrete signal s is a set of events with distinct tags where there is an order embedding from the tags to the integers. Thus, a signal is equivalently a *sequence* s' of events, a partial function

$$s' : N \rightarrow T \times V$$

where the tags are ordered,

$$n < m \;\Rightarrow\; \pi_1(s'(n)) < \pi_1(s'(m))$$

Prefix Order on Signals



Consider using the prefix order on signals and requiring actors to be monotonic functions:

$$a \sqsubseteq a' \;\Rightarrow\; f(a) \sqsubseteq f(a')$$

Will this be an adequate basis for DE semantics?

First Problem: Ensuring that Tags are Distinct

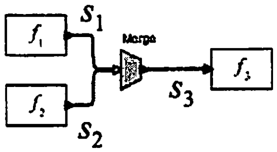
Consider an actor:

where, for each input event e it produces the output $((0, 0), 0)$, an event with tag $(0, 0)$. The output sequence does not have distinct tags. But the function is monotonic in the prefix order.

Simple solution: Do not allow actors to specify the index. The output sequence becomes:

$(((0, 0), 0), ((0, 1), 0), ((0, 2), 0), \ldots)$

Example: Merge Actor



The output cannot be defined to be simply the union of the input events, because the output may then have duplicate tags.

Define the Merge actor so that if the inputs have events with the same time stamp *t*:

$$s_1 = \{\ldots\ ((t, 0), v_1), ((t, 1), v_2), \ldots)$$

$$s_2 = \{\ldots\ ((t, 0), q_1), ((t, 1), q_2), \ldots)$$

the output will interleave these as follows:

$$s_3 = \{\ldots\ ((t, 0), v_1), ((t, 1), q_1), ((t, 2), v_2), ((t, 3), q_2), \ldots)$$

Second Problem: Causality

Consider an actor:

where, for each input event *e* with time stamp τ it produces an output event with time stamp $\tau - 1$. This actor is monotonic in the prefix order, but could be used to build time travel machines.

Looks like a prefix order alone won't do the job…

Conclusion and Open Issues

- A *discrete system* is one where the there is an order embedding from the set of tags in the system to the integers.

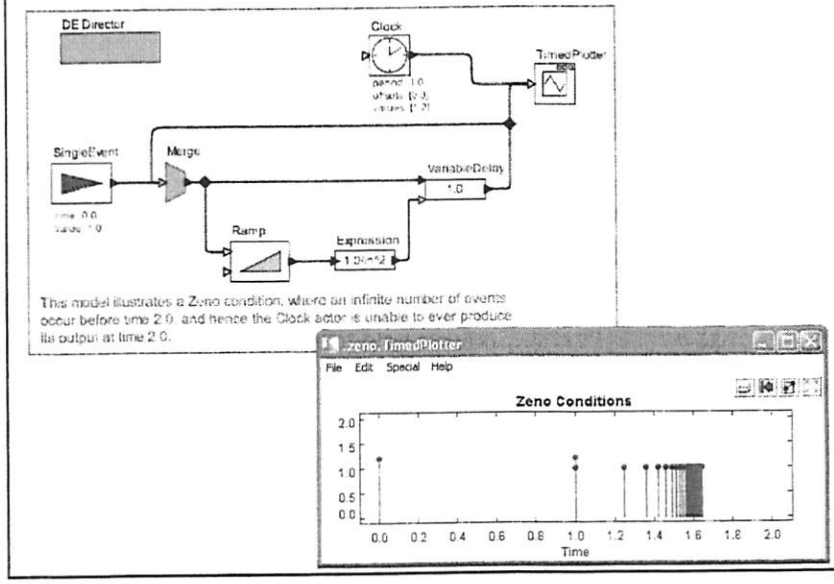
- Monotonic functions on a prefix order does not appear to be sufficient for DE semantics.

Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 13: Metric Space Semantics

We Seek Semantics that Give Meaning to Feedback and Help Rule Out Zeno



This model illustrates a Zeno condition, where an infinite number of events occur before time 2.0, and hence the Clock actor is unable to ever produce its output at time 2.0.

Mathematical Framework

Let the set of all signals be $A=[T\to V]$ where $T=R\times N$ is a totally ordered tag set and V is a set of values. Let an actor

be a function $f: A^n \to A^m$. What are the constraints on these functions such that:

Compositions of actors are determinate.
Feedback compositions have a meaning.
We can rule out Zeno behavior.

Metric

A *metric* on a set A is a function $d: A\times A\to R$ where for all $a,b,c\in A$

1. $d(a,b)=d(b,a)$
2. $d(a,b)=0 \Leftrightarrow a=b$
3. $d(a,b)+d(b,c)\geq d(a,c)$

Exercise: Show that these properties imply that for all $a,b\in A,\ d(a,b)\geq 0$

Metric space: (A,d)

Variations on Metrics

Ultrametric: Replace property 3 with:

3. $\max\,(d(a,b), d(b,c)) \geq d(a,c)$

Exercise: Prove that an ultrametric is a metric.

Partial Metric: Replace properties 2 and 3 with:

2. $d(a,a) \leq d(a,b)$
3. $d(a,b) + d(b,c) - d(b,b) \geq d(a,c)$

In a partial metric, a is the "closest" object to itself.

The Cantor Metric

Given the tag set $T = R \times N$ use only the time stamps. Let

$$d\colon [T \to V] \times [T \to V] \to R$$

such that for all $s, s' \in [T \to V]$,

$$d(s, s') = 1/2^{\tau}$$

where τ is the time stamp of the least tag t where $s(t) \neq s'(t)$. That is, either one is defined and the other not at t or both are defined but are not equal.

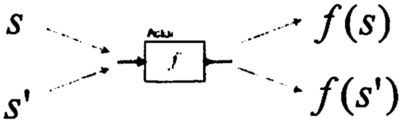
The Cantor Metric is an Ultrametric

Need to show that for all signals $a, b, c \in [T \to V]$,

1. $d(a,b) = d(b,a)$
2. $d(a,b) = 0 \Leftrightarrow a = b$
3. $\max(d(a,b), d(b,c)) \geq d(a,c)$

(1) and (2) are obvious. To show (3), assume without loss of generality that $d(a,b) \geq d(b,c)$. This means that *a* and *b* differ earlier than *b* and *c*. Suppose that *a* and *b* differ first at time τ. Since *a* and *b* differ earlier than *b* and *c*, then prior to τ, *b* and *c* are identical. Thus, *a* and *c* must be identical prior to τ so $d(a,c)$ must be smaller than or equal to $d(a,b)$. QED

Causality



Causal: For all signals *s* and *s'*

$$d(f(s), f(s')) \leq d(s, s')$$

Strictly causal: For all signals *s* and *s'*

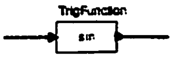
$$s \neq s' \;\Rightarrow\; d(f(s), f(s')) < d(s, s')$$

Delta causal: There exists a real number $\delta < 1$ such that for all signals *s* and *s'*

$$s \neq s' \;\Rightarrow\; d(f(s), f(s')) \leq \delta d(s, s')$$

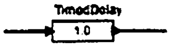
Examples

Simple functional actor:



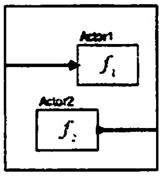
This actor is causal but not strictly causal or delta causal.

Time delay with non-zero delay:



This actor is delta causal.

Source and Sink Actors



Consider Actor1. Its function is $f_1 : A^1 \to A^0$ where A^0 is a *singleton set* (a set with one element). Such a function is always delta causal with $\delta = 0$.

Consider Actor2. Its function is $f_1 : A^0 \to A^1$. Such a function is again always delta causal with $\delta = 0$.
In fact, the function can only yield one possible output signal, since its domain has size 1.

Extending to Multiple Inputs/Outputs

Consider a function $f: A^n \to A^m$, where $A = [T \to V]$

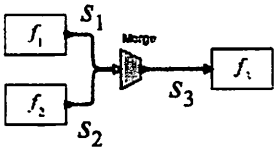
The input is a tuple of signals $(a_1, a_2, \ldots, a_n)$.

Extend the Cantor metric to handle tuples:

$$d((a_1, a_2, \ldots, a_n),(b_1, b_2, \ldots, b_n))$$
$$= \min(d(a_1, b_1), \ldots, d(a_n, b_n))$$

The resulting function is still an ultrametric.

Example: Merge Actor



Recall that for input

$$s_1 = \{\ldots ((t, 0), v_1), ((t, 1), v_2), \ldots)$$
$$s_2 = \{\ldots ((t, 0), q_1), ((t, 1), q_2), \ldots)$$

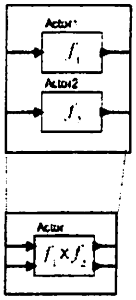
the output is:

$$s_3 = \{\ldots ((t, 0), v_1), ((t, 1), q_1), ((t, 2), v_2), ((t, 3), q_2), \ldots)$$

This actor is causal but not strictly causal, and the operations on indexes do not appear in the semantics.

Parallel Composition of Actors

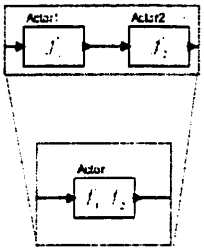
If f_1 and f_2 are causal (strictly causal, delta causal), then so is $f_1 \times f_2$.



What if f_1 is causal and f_2 is delta causal?

Cascade Composition of Actors

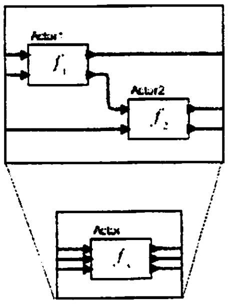
If f_1 and f_2 are causal (strictly causal, delta causal), then so is $f_1 \circ f_2$.



What if f_1 is causal and f_2 is delta causal?

More Interesting Composition

If f_1 and f_2 are causal (strictly causal, delta causal), then so is the following composition:



Question: What if f_1 is causal and f_2 is delta causal?

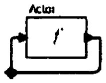
Technicality

In the set $S = [T \to V]$, we could have a signal s that has, for example, an event at all integer time stamps (positive and negative), and we could compare it against a signal s' that has no events at all.

$$d(s, s') = \infty$$

This is problematic. We can avoid these problems by excluding from the set S all signals that have infinite distance from the empty signal. All such signals have an earliest event.

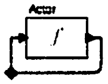
Feedback: Fixed Point Semantics



Since monotonicity on the prefix order is not very useful, we can't use fixed-point theorem 1.

Use instead fixed-point theorems on metric spaces.

Fixed Point Theorem 3



Let $(S^n = [T \to V]^n, d)$ be a metric space and $f: S^n \to S^n$ be a strictly causal function. Then f has at most one fixed point.

Proof: It is enough to show that

$$s \neq s' \Rightarrow f(s) \neq s \text{ or } f(s') \neq s'.$$

Suppose to the contrary that

$$s \neq s' \text{ and } f(s) = s \text{ and } f(s') = s'$$

But this is not possible because it would imply that

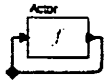
$$d(s, s') = d(f(s), f(s')) < d(s, s').$$

Determinacy

Fixed-Point Theorem 3 takes care of determinacy. There can be no more than one behavior.

Can we find that behavior?

Fixed Point Theorem 4
(Banach Fixed Point Theorem)



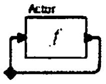
Let $(S^n = [T \to V]^n, d)$ be a *complete* metric space and $f : S^n \to S^n$ be a delta causal function. Then f has a unique fixed point, and for any point $s \in S^n$, the following sequence converges to that fixed point:

$$s_1 = s,\ s_2 = f(s_1),\ s_3 = f(s_2),\ \ldots$$

This means no Zeno! Two issues:

- Any starting point?
- Complete metric space?

Construction of a Fixed Point: Example



Suppose f is a delay by one time unit, such that

$$s' = f(s)$$

where for each event $e = (t, v) \in s$ where $t = (\tau , n)$, there is an event $e' = (t', v) \in s'$ where $t' = (\tau + 1 , n)$.

Suppose we start with a "lucky guess" $s = \emptyset$. This is the only fixed point, so we converge immediately.

Suppose we start with an "unlucky guess" $s = \{((0,0), 0)\}$. As we iterate f, the event gets further out in the future, and the signal "converges" to $s = \emptyset$.

Complete Metric Spaces

A *Cauchy sequence* $\{s_1, s_2, \ldots\}$ is an infinite sequence where

$$d(s_n, s_m) \to 0 \text{ as } n, m \to \infty$$

A *complete metric space* (X, d) is one where every Cauchy sequence has a limit in X.

Example 1

Consider a sequence $\{s_1, s_2, \ldots\}$ where

$$s_n = \{((n, 0), v)\}$$

Is this sequence Cauchy?

Does the sequence converge? To what?

Example 1

Consider a sequence $\{s_1, s_2, \ldots\}$ where

$$s_n = \{((n, 0), v)\}$$

Is this sequence Cauchy? **Yes**

$$d(s_n, s_m) = 1/2^{\min(m, n)} \rightarrow 0$$

Does the sequence converge? To what? **Yes. To ∅**

$$\lim(s_n) = \varnothing$$

Example 2

Consider a sequence $\{s_1, s_2, \ldots\}$ where

$$s_n = \{((i, 0), v) \mid i \in \{1, 2, \ldots, n\}\}$$

Is this sequence Cauchy?

Does the sequence converge? To what?

Example 2

Consider a sequence $\{s_1, s_2, \ldots\}$ where

$$s_n = \{((i, 0), v) \mid i \in \{1, 2, \ldots, n\}\}$$

Is this sequence Cauchy? **Yes**

$$d(s_n, s_m) = 1/2^{\min(m, n) + 1} \to 0$$

Does the sequence converge? To what? **Yes. To**

$$\{((i, 0), v) \mid i \in \{1, 2, \ldots\}\}$$

Example 3

Consider a sequence $\{s_1, s_2, \ldots\}$ where

$$s_n = \{((\tau_i, 0), v) \mid i \in \{1, 2, \ldots, n\}, \tau_i = 1 - 1/i\}$$

Is this sequence Cauchy?

Does the sequence converge? To what?

Example 3

Consider a sequence $\{s_1, s_2, \ldots\}$ where

$$s_n = \{((\tau_i, 0), v) \mid i \in \{1, 2, \ldots, n\}, \tau_i = 1 - 1/i\}$$

Is this sequence Cauchy? **No**

$$d(s_n, s_m) > 1/2$$

Does the sequence converge? To what? **No. Exercise.**

Completeness of DE Signals

The set of n-tuples of discrete-event signals under the Cantor metric is a complete metric space.

Proof (sketch): We need to show that every Cauchy sequence converges. Given a Cauchy sequence $\{s_1, s_2, \ldots\}$, for any tag t with time stamp $\tau > 0$, there is a subsequence $\{s_n, s_{n+1}, \ldots\}$, for some $n > 0$, of signals that are identical up to and including tag t. Let s be the sequence obtained by letting its value at each tag t be that identical value (or absence, if all signals in the subsequence have no event at t). This is clearly a signal (or tuple of signals). Then it is easy to show that the Cauchy sequence converges to s.

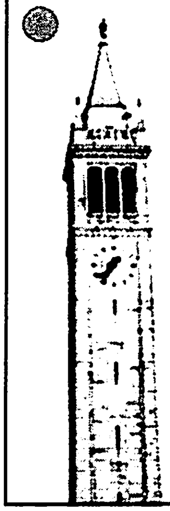
Thanks to Adam Cataldo for this proof.

Operational Semantics

1. Topologically sort actors according to paths that do not increment tags.
2. Start with a set of events on signals taken from the event queue that all have the same tag.
3. Iterate to find a fixed-point value for all signals at that tag (absent or having a value).
4. Continue with the next smallest tag in the event queue.

Conclusions and Open Issues

- Ignoring the index, strictly causal functions in a feedback loop have at most one fixed point, and hence are determinate.
- Delta causal functions in a feedback loop have exactly one fixed point, and that fixed point can be found by starting with any initial signal(s) and iterating to the fixed point. This guarantees no Zeno.
- Convergence in DE is achieved when time stamps approach infinity.
- Within a time stamp, use SR semantics and iterate to a fixed point.



Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley

EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 14: Dataflow Process Networks

Firings

Dataflow is a variant of Kahn Process Networks where a process is computed as a sequence of atomic *firings*, which are finite computations enabled by a *firing rule*.

In a firing, an actor consumes a finite number of input tokens and produces a finite number of outputs.

A possibly infinite sequence of firings is called a *dataflow process*.

Firing Rules

Let $F : S^n \rightarrow S^m$ be a dataflow process.

Let $U \subset S^n$ be a set of *firing rules* with the constraints:

1. Every $u \in U$ is finite, and
2. No two elements of U are joinable.

This implies that for all $s \in S^n$ there is at most one $u \in U$ where $u \sqsubseteq s$. (exercise)

When $u \sqsubseteq s$ there is a unique s' such that $s = u.s'$ where the period denotes concatenation of sequences.

Firing Function

Let $f : S^n \rightarrow S^m$ be a (possibly partial) firing function with the constraint that for all $u \in U$, $f(u)$ is defined and is finite.

Then the dataflow process $F : S^n \rightarrow S^m$ is given by

$$F(s) = \begin{cases} f(u).F(s') & \text{if there is a } u \in U \text{ such that } s = u.s' \\ \perp_n & \text{otherwise} \end{cases}$$

where $\perp_n \in S^n$ is the n-tuple of empty sequences.
Note that this is self referential. Seek a fixed point F.

Fixed Point Definition of Dataflow Process (cf. Lifting Formulation in SR)

Define $\phi : [S^n \to S^m] \to [S^n \to S^m]$ by:

$$(\phi(F))(s) = \begin{cases} f(u).F(s') & \text{if there is a } u \in U \text{ such that } s = u.s' \\ \bot_n & \text{otherwise} \end{cases}$$

Fact: ϕ is continuous (see handout). This means that it has a unique least fixed point, and that we can constructively find that fixed point by starting with the bottom of the CPO. The bottom of the CPO is the function $F_0 : S^n \to S^m$ that returns $\bot_n$.

Executing a Dataflow Process is the Same as Finding the Least Fixed Point

Suppose $s \in S^n$ is a concatenation of firing rules,

$$s = u_1.\, u_2.\, u_3 \quad \ldots$$

Then the procedure for finding the least fixed point of ϕ yields the following sequence of approximations to the dataflow process:

$$F_0(s) = \bot_n$$

$$F_1(s) = (\phi(F_0))(s) = f(u_1)$$

$$F_2(s) = (\phi(F_1))(s) = f(u_1).f(u_2)$$

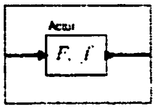
$$\ldots$$

This exactly describes the operational semantics of repeated firings governed by the firing rules!

The LUB of this Sequence of Functions is Continuous

The chain $\{F_0(s), F_1(s), \dots\}$ will be finite for some s (certainly for finite s, but also for any s for which after some point no more firing rules match), and infinite for other s. Since each F_i is a continuous function, and the set of continuous functions is a CPO, then the LUB is continuous, and hence describes a valid Kahn process that guarantees determinacy, and can be put into a feedback loop.

Example 1



Suppose $V = \{0, 1\}$ and $S = V^{**}$ is the set of finite and infinite sequences of elements from V.

Consider a dataflow process with one input and one output, $F : S \to S$. Its firing rules are $U \subset S$. The following are all valid firing rules:

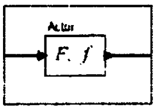
$$U = \{\perp\}$$

$$U = \{(0)\}$$

$$U = \{(0), (1)\}$$

$$U = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$$

Example 2 : Valid Firing Rule?



Suppose $V = \{0, 1\}$ and $S = V^{**}$ is the set of finite and infinite sequences of elements from V.

Consider a dataflow process with one input and one output, $F: S \rightarrow S$. Its firing rules are $U \subset S$. Is the following set a valid set of firing rule?

$$U = \{\perp, (0), (1)\}$$

Example 2 : Valid Firing Rule?

Suppose $V = \{0, 1\}$ and $S = V^{**}$ is the set of finite and infinite sequences of elements from V.

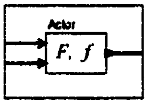
Consider a dataflow process with one input and one output, $F: S \rightarrow S$. Its firing rules are $U \subset S$. Is the following set a valid set of firing rule?

$$U = \{\perp, (0), (1)\}$$

No. There are joinable pairs.

Intuition: The same input sequence can lead to multiple executions. Nondeterminacy!

Example 3



Consider $F : S^2 \to S$. Its firing rules are $U \subset S^2$. Which of the following are valid sets of firing rules?

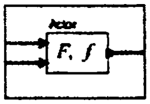
$$\{((0), (0)), ((0), (1)), ((1), (0)), ((1), (1))\}$$

$$\{((0), \perp), ((1), \perp), (\perp, (0)), (\perp, (1))\}$$

$$\{((0), \perp), ((1), (0)), ((1), (1))\}$$

$$\{((0), \perp), ((1), \perp)\}$$

Example 3



Consider $F : S^2 \to S$. Its firing rules are $U \subset S^2$. Which of the following are valid sets of firing rules?

$$\{((0), (0)), ((0), (1)), ((1), (0)), ((1), (1))\}$$

Yes. Consume one token from each input.

$$\{((0), \perp), ((1), \perp), (\perp, (0)), (\perp, (1))\}$$

No. Nondeterminate merge.

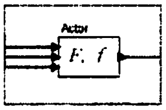
$$\{((0), \perp), ((1), (0)), ((1), (1))\}$$

Yes. Consume from the second input if the first is 1.

$$\{((0), \perp), ((1), \perp)\}$$

Yes. Consume only from the first input.

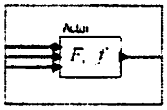
Example 4



Consider $F : S^3 \to S$. Its firing rules are $U \subset S^3$. Is the following a valid set of firing rules?

$$\{((1), (0), \perp), ((0), \perp, (1)), (\perp, (1), (0))\}$$

Example 4



Consider $F : S^3 \to S$. Its firing rules are $U \subset S^3$. Is the following a valid set of firing rules?

$$\{((1), (0), \perp), ((0), \perp, (1)), (\perp, (1), (0))\}$$

Yes. Dataflow version of the Gustave function!

Conclusions and Open Issues

- Dataflow processes are Kahn processes composed of atomic *firings*.
- Firing rules that are not joinable lead to simple fixed point semantics.

Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 15: Generalized Firing Rules

Firing Rules from Last Lecture

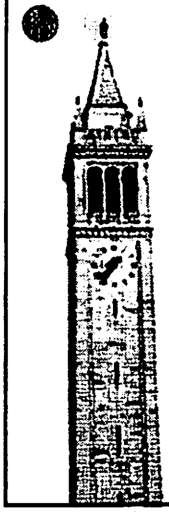
Let $F : S^n \to S^m$ be a dataflow process.

Let $U \subset S^n$ be a set of *firing rules* with the constraints:

1. Every $u \in U$ is finite, and
2. No two elements of U are joinable.

This implies that for all $s \in S^n$ there is at most one $u \in U$ where $u \sqsubseteq s$. (exercise)

When $u \sqsubseteq s$ there is a unique s' such that $s = u.s'$ where the period denotes concatenation of sequences.



Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 15: Generalized Firing Rules

Firing Rules from Last Lecture

Let $F : S^n \to S^m$ be a dataflow process.

Let $U \subset S^n$ be a set of *firing rules* with the constraints:

1. Every $u \in U$ is finite, and
2. No two elements of U are joinable.

This implies that for all $s \in S^n$ there is at most one $u \in U$ where $u \sqsubseteq s$. (exercise)

When $u \sqsubseteq s$ there is a unique s' such that $s = u.s'$ where the period denotes concatenation of sequences.

Source and Sink Actors

Sink actor: $F : S^n \to S^0$ with firing function $f : S^n \to S^0$.

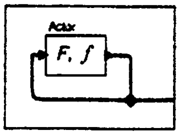
In this case, if $S^0 = \{\sigma\}$ then $f(u) = \sigma$ is the single element. Define concatenation in S^0 so that $\sigma.\sigma = \sigma$. Then everything works (e.g., let $\sigma = \perp$).

Source actor: $F : S^0 \to S^m$ with firing function $f : S^0 \to S^m$. Firing rules $U = S^0$ (singleton set) have the constraints trivially satisfied.

Source Actors Too Limited?

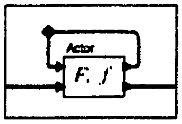
With the above definitions, the dataflow process produces the sequence $f(\sigma) . f(\sigma) . f(\sigma) \ldots$ where $U = S^0 = \{\sigma\}$.

If is non-empty, this is infinite and periodic. This may seem limiting for dataflow processes that act as sources, but in fact it is not, because a source with a more complicated output sequence can be constructed using feedback composition.



More Generally:
Is a Single Firing Function Too Restrictive?

Not really. Use a self loop:

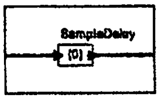


Let the data type of the feedback loop be $V = \{1, 2, \ldots, n\}$

Then the first argument to the firing function can represent n different "states" of the actor, where in each state the output is a different function of the input.

But how can you get this started?

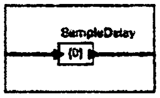
A Possible Problem:
Sample Delay Actor



Can the sample delay be represented with the following firing rules?

$$\{\perp, (0), (1)\}$$

A Possible Problem: Sample Delay Actor



Can the sample delay be represented with the following firing rules?

$$\{\perp, (0), (1)\}$$

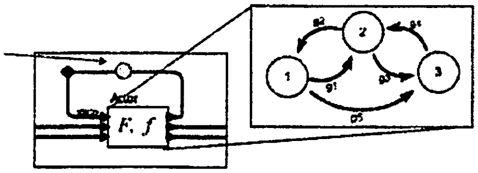
No. These are not joinable.

Instead, we require that initial tokens on an arc be a primitive concept in dataflow. This is implemented in Ptolemy II by outputting the initial token prior to any firings.

Firing Rules Defined by a State Machine

Feedback path data type: V = $\{1, 2, \ldots, n\}$ where there are n states:

initial state $i \in V$



In each state $i \in V$, there is a set of firing rules

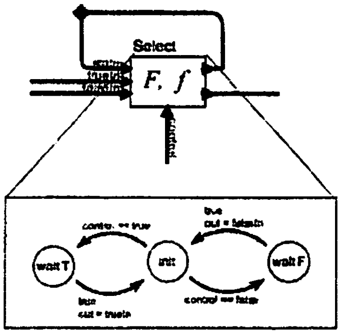
$$U_i = \{(i,\ldots), (i,\ldots), \ldots\}$$

where every member is finite and no two members are joinable. Then the total set of firing rules is

$$U = U_1 \cup \ldots \cup U_n$$

Every member is finite and no two members are joinable.

Example: Select Actor



- In the *init* state, read input from the *control* port.
- In the *waitT* state, read input from the *trueIn* port.
- In the *waitF* state, read input from the *falseIn* port.

$$U_{init} = \{(init, \perp, \perp, *)\}$$

$$U_{waitT} = \{(waitT, *, \perp, \perp)\}$$

$$U_{waitF} = \{(waitF, \perp, *, \perp)\}$$

shorthand to match any input token

Sequential Functions

Any sequential function can be implemented by a state machine that in each state has firing rules that match the state identifier in the state input port and match any token in exactly one other input port.

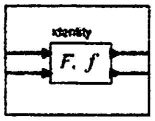
Each state could also (in effect) implement a different firing function (one firing function with the state identifier as an input can model this).

Generalize Further to get the Cal Actor Language

Partition the firing rules and associate a distinct firing function with each partition of the firing rules. Each such firing function is called an *action*.

This is similar to the pattern matching in some functional languages such as Haskell.

Another Possible Problem: Cannot Implement Identity Functions!

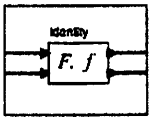


Will the following firing rules work?

$$\{((0), \bot), ((1), \bot), (\bot, (0)), (\bot, (1))\}$$

$$\{((0), (0)), ((0), (1)), ((1), (0)), ((1), (1))\}$$

Cannot Implement Identity Functions!



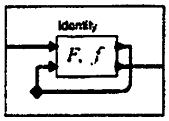
Will the following firing rules work?

$$\{((0), \perp), ((1), \perp), (\perp, (0)), (\perp, (1))\}$$

No. Nondeterminate merge.

$$\{((0), (0)), ((0), (1)), ((1), (0)), ((1), (1))\}$$

No. Try feeding back one output to one input. E.g.:



Generalized Firing Rules

We previously defined the firing rules $U \subset S^n$ with:

1. Every $u \in U$ is finite, and
2. No two elements of U are joinable.

We now replace constraint 2 with:

3. For any two elements of $u, u' \in U$ that are joinable, we require that:

$$u \wedge u' = \perp_n$$

$$f(u) . f(u') = f(u') . f(u)$$

I.e., when two firing rules are enabled, they can be applied in either order without changing the output.

Examining Rule 3

3. For any two elements of $u, u' \in U$ that are joinable, we require that:

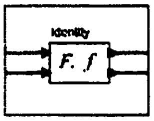
$$u \wedge u' = \perp_n$$

I.e., no two joinable firing rules have a common prefix.

$$f(u) . f(u') = f(u') . f(u)$$

I.e., when two firing rules are enabled, they can be applied in either order without changing the output.

Applying Rule 3 to Identity Functions



With these firing rules

$$U = \{((0), \perp), ((1), \perp), (\perp, (0)), (\perp, (1))\}$$

and for all $u \in U$,

$$f(u) = u$$

rule 3 is satisfied. Exercise: Show that rule 3 is not satisfied by the nondeterminate merge.

Fixed Point Semantics Under Rule 3

Let $Q(s) = \{u_1, u_2, \dots, u_q\} \subset U$ be the set of all firing rules that are a prefix of s. This could be empty. Then define

$$(\phi'(F))(s) = \begin{cases} f(u_1).f(u_2).....f(u_q).F(s') & \text{if } Q(s) \neq \varnothing \\ \perp_n & \text{otherwise} \end{cases}$$

Where $s = \vee Q(s).s'$
(exercise to show that s' always exists).

The function ϕ' is continuous, and all previous results hold.

Conclusions and Open Issues

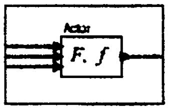
- Dataflow processes are Kahn processes composed of atomic *firings*.
- Firing rules that are not joinable lead to simple fixed point semantics.
- Simple semantics leaves out delays, two-input identity functions, and other compositions.
- Generalized firing rules allow joinable pairs under certain circumstances.

Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 16: Statically Schedulable Dataflow

Execution Policy for a Dataflow Actor



Suppose $s \in S^n$ is a concatenation of firing rules,

$$s = u_1 . u_2 . u_3 \quad \ldots$$

Then the output of the actor is the concatenation of the results of a sequence of applications of the firing function:

$$F_0(s) = \perp_n$$

$$F_1(s) = (\phi(F_0))(s) = f(u_1)$$

$$F_2(s) = (\phi(F_1))(s) = f(u_1).f(u_2)$$

$$\ldots$$

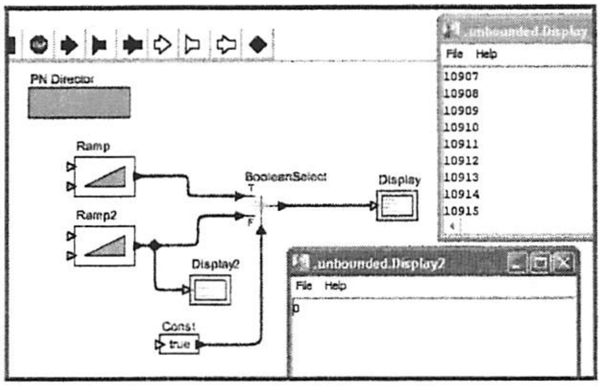
The problem we address now is *scheduling*: how to choose which actor to fire when there are choices.

Apply the Same Policy as for PN

- Define a *correct execution* to be any execution for which after any finite time every signal is a prefix of the LUB signal given by the semantics.
- Define a *useful execution* to be a correct execution that satisfies the following criteria:
  - For every non-terminating PN model, after any finite time, a useful execution will extend at least one signal in finite (additional) time.
  - If a correct execution satisfying criterion (1) exists that executes with bounded buffers, then a useful execution will execute with bounded buffers.

Policies that Fail

- Fair scheduling
- Demand driven
- Data driven



```
10907
10908
10909
10910
10911
10912
10913
10914
10915
```

```
0
```

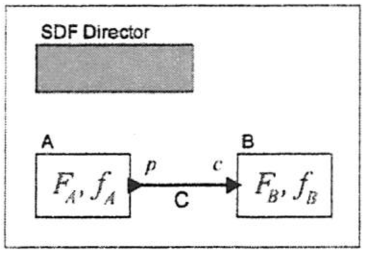
Adapting Parks' Strategy to Dataflow

- Require that the scheduler "know" how many tokens a firing will produce on each output port before that firing is invoked.
- Start with an arbitrary bound on the capacity of all buffers.
- Execute enabled actors that will not overflow the buffers on their outputs.
- If deadlock occurs and at least one actor is blocked on a enabled, increase the capacity of at least one buffer to allow an actor to fire.
- Continue executing, repeatedly checking for deadlock.

But Often the Firing Sequence can be Statically Determined! A History of Attempts:

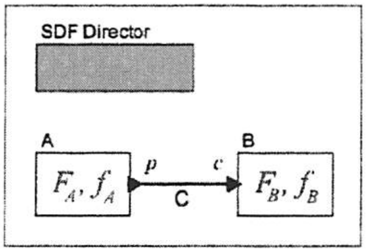
- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986] **today**
- Structured dataflow [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980's]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- Parameterized dataflow [Bhattacharya and Bhattacharyya 2001]
- ...

Statically Schedulable Dataflow – SSDF
Historically called: Synchronous Dataflow (SDF)



If the number of tokens consumed and produced by the firing of an actor is constant, then static analysis can tell us whether we can schedule the firings to get a useful execution, and if so, then a finite representation of a schedule for such an execution can be created.

Balance Equations



Let q_A, q_B be the number of firings of actors A and B.
Let p_C, c_C be the number of token produced and consumed on a connection C.
Then the system is *in balance* if for all connections C

$$q_A p_C = q_B c_C$$

where A produces tokens on C and B consumes them.

Relating to Infinite Firings

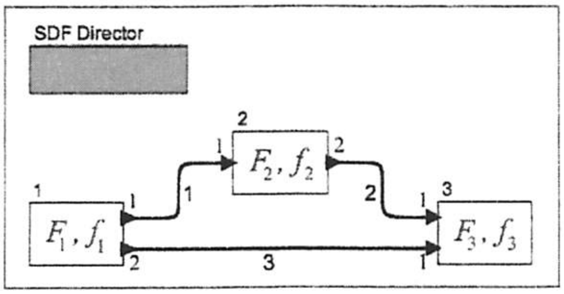
Of course, if $q_A = q_B = \infty$, then the balance equations are trivially satisfied.

By keeping a system in balance as an infinite execution proceeds, we can keep the buffers bounded.

Whether we can have a bounded infinite execution turns out to be decidable for SSDF models.

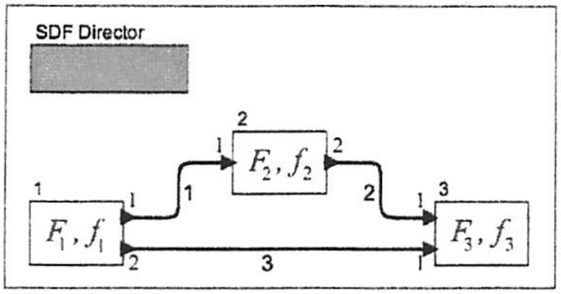
Example

Consider this example, where actors and arcs are numbered:



The balance equations imply that actor 3 must fire twice as often as the other two actors.

Compactly Representing the Balance Equations



production/consumption matrix

Connector 1

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$

Actor 1

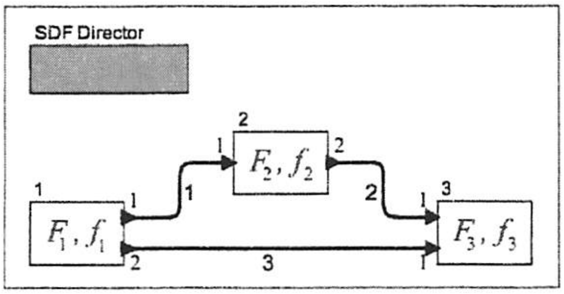
$$q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

firing vector

balance equations

$$\Gamma q = \vec{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Example

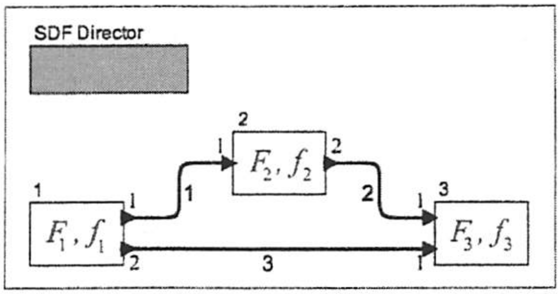


A solution to balance equations:

$$q = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \qquad \Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \qquad \Gamma q = \vec{0}$$

This tells us that actor 3 must fire twice as often as actors 1 and 2.

Example

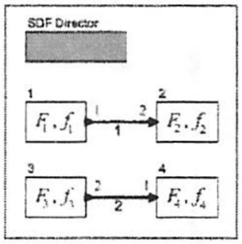


But there are many solutions to the balance equations:

$$q=\begin{bmatrix}1\\1\\2\end{bmatrix}\quad q=\begin{bmatrix}0\\0\\0\end{bmatrix}\quad q=\begin{bmatrix}2\\2\\4\end{bmatrix}\quad q=\begin{bmatrix}-1\\-1\\-2\end{bmatrix}\quad q=\begin{bmatrix}\pi\\\pi\\2\pi\end{bmatrix}\qquad \Gamma q=\vec{0}$$

We will see that for "well-behaved" models, there is a unique least positive solution.

Disconnected Models



For a disconnected model with two connected components, solutions to the balance equations have the form:

Solutions are linear combinations of the solutions for each connected component:

$$\Gamma=\begin{bmatrix}1 & -2 & 0 & 0\\0 & 0 & 2 & -1\end{bmatrix}\qquad q=\begin{bmatrix}2n\\n\\m\\2m\end{bmatrix}=n\begin{bmatrix}2\\1\\0\\0\end{bmatrix}+m\begin{bmatrix}0\\0\\1\\2\end{bmatrix}$$

$$\Gamma q=\vec{0}$$

Disconnected Models are Just Separate Connected Models

Define a *connected model* to be one where there is a path from any actor to any other actor, and where every connection along the path has production and consumption numbers greater than zero.

It is sufficient to consider only connected models, since disconnected models are disjoint unions of connected models. A schedule for a disconnected model is an arbitrary interleaving of schedules for the connected components.

Least Positive Solution to the Balance Equations

Note that if p_C, c_C , the number of tokens produced and consumed on a connection C, are non-negative integers, then the balance equation,

$$q_A p_C = q_B c_C$$

implies:

- q_A is rational if an only if q_B is rational.
- q_A is positive if an only if q_B is positive.

Consequence: Within any connected component, if there is any solution to the balance equations, then there is a unique least positive solution.

Rank of a Matrix

The rank of a matrix Γ is the number of linearly independent rows or columns. The equation

$$\Gamma q = \vec{0}$$

is forming a linear combination of the columns of G. Such a linear combination can only yield the zero vector if the columns are linearly dependent (this is what is means to be linearly dependent).

If Γ has a rows and b columns, the rank cannot exceed $\min(a, b)$. If the columns or rows of Γ are re-ordered, the resulting matrix has the same rank as Γ.

Rank of the Production/Consumption Matrix

Let a be the number of actors in a connected graph. Then the *rank* of the production/consumption matrix Γ must be a or $a - 1$.

Γ has a columns and at least $a - 1$ rows. If it has only $a - 1$ columns, then it cannot have rank a.

If the model is a *spanning tree* (meaning that there are barely enough connections to make it connected) then Γ has a rows and $a - 1$ columns. Its rank is $a - 1$. (Prove by induction).

Consistent Models

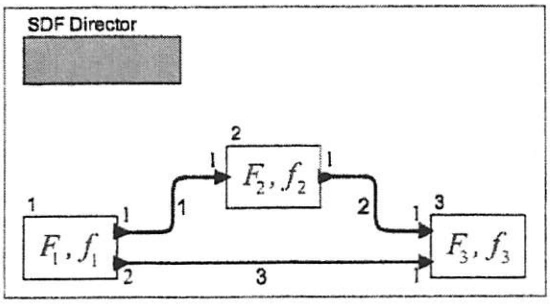
Let a be the number of actors in a connected model. The model is *consistent* if Γ has rank $a - 1$.

If the rank is a, then the balance equations have only a trivial solution (zero firings).

When Γ has rank $a - 1$, then the balance equations always have a non-trivial solution.

Example of an Inconsistent Model: No Non-Trivial Solution to the Balance Equations

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$



This production/consumption matrix has rank 3, so there are no nontrivial solutions to the balance equations.

Dynamics of Execution

Consider a model with 3 actors. Let the *schedule* be a sequence $v : N_0 \to B^3$ where $B = \{0, 1\}$ is the binary set. That is,

$$v(n) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

to indicate firing of actor 1, 2, or 3.

Buffer Sizes and Periodic Admissible Sequential Schedules (PASS)

Assume there are m connections and let $b : N_0 \to N^m$ indicate the buffer sizes prior to the each firing. That is, $b(0)$ gives the initial number of tokens in each buffer, $b(1)$ gives the number after the first firing, etc. Then

$$b(n+1) = b(n) + \Gamma v(n)$$

A *periodic admissible sequential schedule* (PASS) of length K is a sequence

$$v(0) \dots v(K-1)$$

such that $b(n) \geq \vec{0}$ for each $n \in \{0, \dots K-1\}$, and

$$b(K) = b(0) + \Gamma[v(0) + \dots + v(K-1)] = b(0)$$

Periodic Admissible Sequential Schedules

Let $q = v(0) + ... + v(K-1)$
and note that we require that $\Gamma q = \vec{0}$.

A PASS will bring the model back to its initial state, and hence it can be repeated indefinitely with bounded memory requires.

A necessary condition for the existence of a PASS is that the balance equations have a non-zero solution. Hence, a PASS can only exist for a consistent model.

☆

SSDF Theorem 1

We have proved:

For a connected SSDF model with *a* actors, a *necessary* condition for the existence of a PASS is that the model be consistent.

SSDF Theorem 2

We have also proved:

For a consistent connected SSDF model with production/consumption matrix Γ, we can find an integer vector q where every element is greater than zero such that

$$\Gamma q = \vec{0}$$

Furthermore, there is a unique least such vector q.

SSDF Sequential Scheduling Algorithms

Given a consistent connected SSDF model with production/consumption matrix Γ, find the least positive integer vector q such that $\Gamma q = \vec{0}$.

Let $K = \mathbf{1}^{\mathrm{T}} q$, where $\mathbf{1}^{\mathrm{T}}$ is a row vector filled with ones. Then for each of $n \in \{0, \ldots K-1\}$, choose a firing vector

$$v(n) = \left\{ \begin{bmatrix} 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \cdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 1 \end{bmatrix} \right\}$$

The number of rows in $v(n)$ is a.

SSDF Sequential Scheduling Algorithms (Continued)

.. such that $b(n+1) = b(n) + \Gamma v(n) \geq \vec{0}$ (each element is non-negative), where $b(0)$ is the initial state of the buffers, and

$$\sum_{n=0}^{K-1} v(n) = q$$

The resulting *schedule* ($v(0)$, $v(1)$, ..., $v(K-1)$) forms one cycle of an infinite periodic schedule.

Such an algorithm is called an *SSDF Sequential Scheduling Algorithm (SSSA).*

SSDF Theorem 3

If an SSDF model has a correct infinite sequential execution that executes in bounded memory, then any SSSA will find a schedule that provides such an execution.

Proof outline: Must show that if an SSDF has a correct, infinite, bounded execution, then it has a PASS of length K. See Lee & Messerschmit [1987]. Then must show that the schedule yielded by an SSSA is correct, infinite, and bounded (trivial).

Note that every SSSA terminates.

Creating a Scheduler

Given a connected SSDF model with actors $A_1, \ldots, A_a$:

Step 1: Solve for a rational q. To do this, first let $q_1 = 1$. Then for each actor A_i connected to A_1, let $q_i = q_1 m/n$, where m is the number of tokens A_1 produces or consumes on the connection to A_i, and n is the number of tokens A_i produces or consumes on the connection to A_1. Repeat this for each actor A_j connected to A_i for which we have not already assigned a value to q_j. When all actors have been assigned a value q_j, then we have a found a rational vector q such that $\Gamma q = \vec{0}$.

Creating a Scheduler (continued)

Step 2: Solve for the least integer q. Use Euclid's algorithm to find the least common multiple of the denominators for the elements of the rational vector q. Then multiply through by that least common multiple to obtain the least positive integer vector q such that

$$\Gamma q = \vec{0}$$

Let $K = \mathbf{1}^T q$.

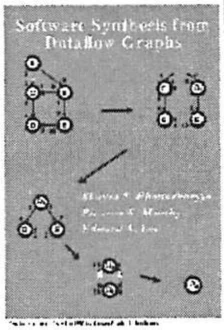
Creating a Scheduler (continued)

Step 3: For each $n \in \{0, \dots, K-1\}$:

1. Given buffer sizes $b(n)$, determine which actors have firing rules that are satisfied (every source actor will have such a firing rule).
2. Select one of these actors that has not already been fired the number of times given by q. Let $v(n)$ be a vector with all zeros except in the position of the chosen actor, where its value is 1.
3. Update the buffer sizes:

$$b(n+1) = b(n) + \Gamma v(n)$$

A Key Question: If More Than One Actor is Fireable in Step 2, How do I Select One?



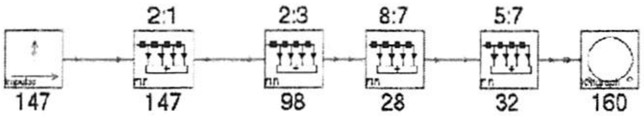
Optimization criteria that might be applied:

- Minimize buffer sizes.
- Minimize the number of actor activations.
- Minimize the size of the representation of the schedule (code size).

See S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs,* Kluwer Academic Press, 1996.

Minimum Buffer Schedule

CD to DAT sample rate conversion

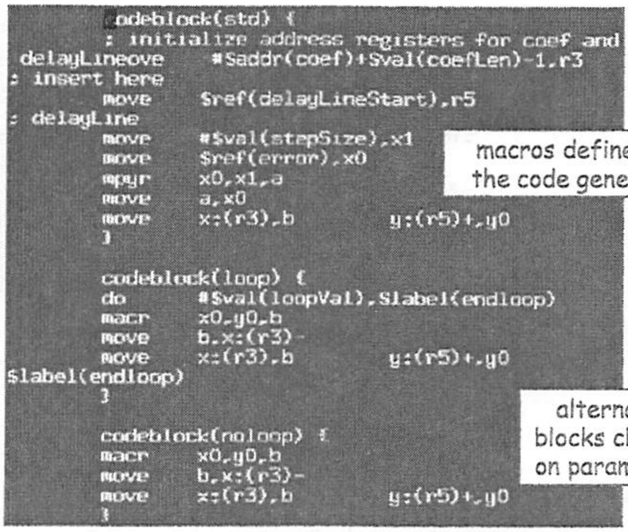


```
ABABCABCABABCABCDEAFFFFFBABCABCABABCDE
AFFFFFBCABABCABCABABCDEAFFFFFBCABABCABC
DEAFFFFFBABCABCABABCABCDEAFFFFFBABCABCA
BABCDEAFFFFFBCABABCABCABABCDEAFFFFFEBCA
FFFFFBABCABCDEAFFFFFBABCABCABABCABCDEAF
FFFFBABCABCABABCDEAFFFFFBCABABCABCABABC
DEAFFFFFBCABABCABCDEAFFFFFBABCABCABABCA
BCDEAFFFFFBABCABCABABCDEAFFFFFEBCAFFFFFB
ABCABCABABCDEAFFFFFBCABABCABCDEAFFFFFBA
BCABCABABCABCDEAFFFFFBABCABCABABCDEAFFF
FFBCABABCABCABABCDEAFFFFFBCABABCABCDEAF
FFFFBABCABCABABCABCDEAFFFFFEBAFFFFFBCABC
ABABCDEAFFFFFBCABABCABCABABCDEAFFFFFBCA
BABCABCDEAFFFFFBABCABCABABCAECDEAFFFFFB
ABCABCABABCDEAFFFFFBCABABCABCABABCDEAF
FFFFBCABABCABCDEFFFFFEFFFFF
```

Source: Shuvra Bhattacharyya

Code Generation (Circa 1992)

Block specification for DSP code generation in Ptolemy Classic:



```
        codeblock(std) {
        ; initialize address registers for coef and
 delayLineove    #$addr(coef)+$val(coefLen)-1,r3
; insert here
        move     $ref(delayLineStart),r5
; delayLine
        move     #$val(stepSize),x1
        move     $ref(error),x0
        mpyr     x0,x1,a
        move     a,x0
        move     x:(r3),b          y:(r5)+,y0
        }

        codeblock(loop) {
        do       #$val(loopVal),$label(endloop)
        macr     x0,y0,b
        move     b,x:(r3)-
        move     x:(r3),b          y:(r5)+,y0
$label(endloop)
        }

        codeblock(noloop) {
        macr     x0,y0,b
        move     b,x:(r3)-
        move     x:(r3),b          y:(r5)+,y0
        }
```

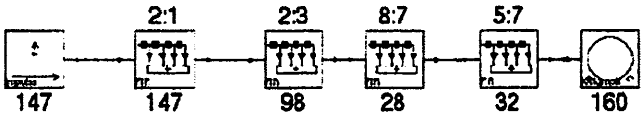
macros defined by the code generator

alternative code blocks chosen based on parameter values

Scheduling Tradeoffs

(Bhattacharyya, Parks, Pino)

CD to DAT sample rate conversion



Scheduling strategy	Code	Data
Minimum buffer schedule, no looping	13735	32
Minimum buffer schedule, with looping	9400	32
Worst minimum code size schedule	170	1021
Best minimum code size schedule	170	264

Source: Shuvra Bhattacharyya

Parallel Scheduling

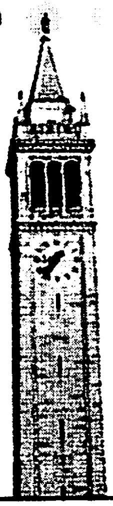
It is easy to create an SSSA that as it produces a PASS, it constructs an *acyclic precedence graph* (APG) that represents the dependencies that an actor firing has on prior actor firings.

Given such an APG, the parallel scheduling problem is a standard one where there are many variants of the optimization criteria and scheduling heuristics.

See many papers on the subject on the Ptolemy website.

Conclusions and Open Issues

- SSDF models have actors that produce and consume a fixed (constant) number of tokens on each arc.
- A periodic admissible sequential schedule (PASS) is a finite sequence of firings that brings buffers back to their initial state and keeps buffer sizes non-negative.
- A necessary condition for the existence of a PASS is that the balance equations have a non-trivial solution.
- A class of algorithms has been identified that will always find a PASS if one exists.



Concurrent Models of Computation for Embedded Software

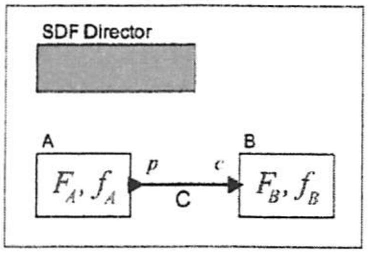
Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 17: Generalizations of SSDF

History of Dataflow Models of Computation

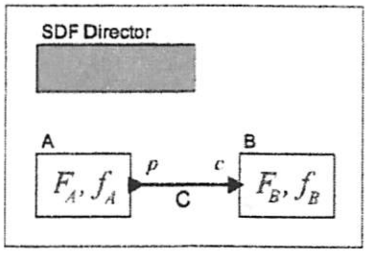
- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980's]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993] **today**
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- Parameterized dataflow [Bhattacharya and Bhattacharyya 2001]
- …

Statically Schedulable Dataflow – SSDF
Historically called: Synchronous Dataflow (SDF)



If the number of tokens consumed and produced by the firing of an actor is constant, then static analysis can tell us whether we can schedule the firings to get a useful execution, and if so, then a finite representation of a schedule for such an execution can be created.

Balance Equations



Let q_A, q_B be the number of firings of actors A and B.
Let p_C, c_C be the number of token produced and consumed on a connection C.
Then the system is *in balance* if for all connections C

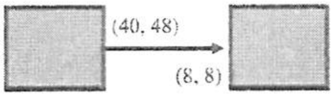
$$q_A p_C = q_B c_C$$

where A produces tokens on C and B consumes them.

Multidimensional SSDF

(Lee, 1993)

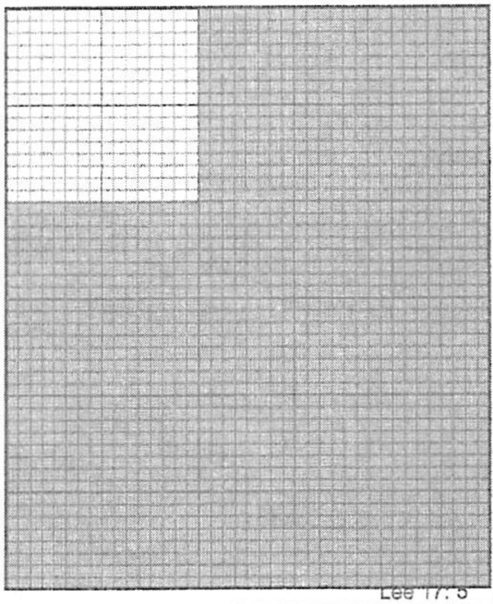
Production and consumption of *N*-dimensional arrays of data:



Balance equations and scheduling policies generalize.

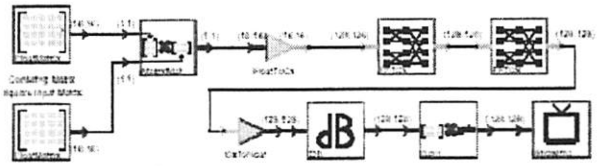
Much more data parallelism is exposed.

Similar (but dynamic) multidimensional streams have been implemented in Lucid.



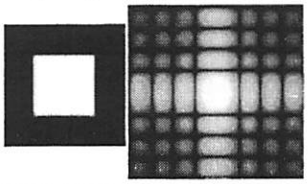
More interesting Example

Two dimensional FFT constructed out of one-dimensional actors.

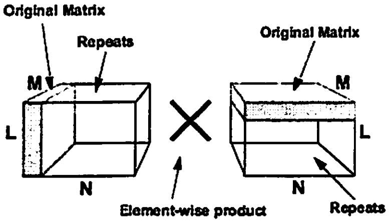


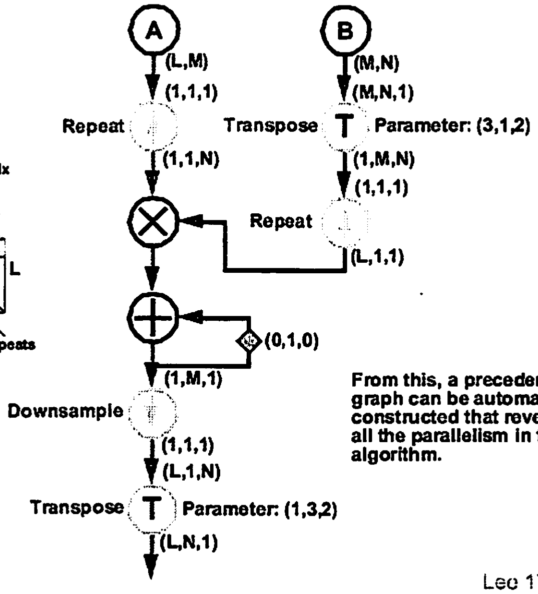
```
MDSDF SCHEDULE:
fft_of_square2.FloatMatrix1, firing range: (0,0)
fft_of_square2.FloatMatrix2, firing range: (0,0)
fft_of_square2.Mult1, firing range: (0,0) - (15,15)
fft_of_square2.FloatToCx1, firing range: (0,0)
fft_of_square2.FFTCx2, firing range: (0,0) - (15,0)
fft_of_square2.FFTCx1, firing range: (0,0) - (0,127)
fft_of_square2.CxToFloat1, firing range: (0,0)
fft_of_square2.DB1, firing range: (0,0)
fft_of_square2.Gain1, firing range: (0,0)
fft_of_square2.ShowImg1, firing range: (0,0)
```

Figure 6. Screen dump of 2D-FFT system, the associated schedule, and outputs.



MDSSDF Structure Exposes Fine-Grain Data Parallelism





From this, a precedence graph can be automatically constructed that reveals all the parallelism in the algorithm.

However, such programs are extremely hard to write (and to read).

Extensions of MDSSDF

Extended to non-rectangular lattices and connections to number theory:

P. K. Murthy, "Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow," Technical Memorandum UCB/ERL M96/79, Ph.D. Thesis, EECS Department, University of California, Berkeley, CA 94720, December 1996.

Praveen K. Murthy and Edward A. Lee, "Multidimensional Synchronous Dataflow ," *IEEE Transactions on Signal Processing*, volume 50, no. 8, pp. 2064 -2079, July 2002.

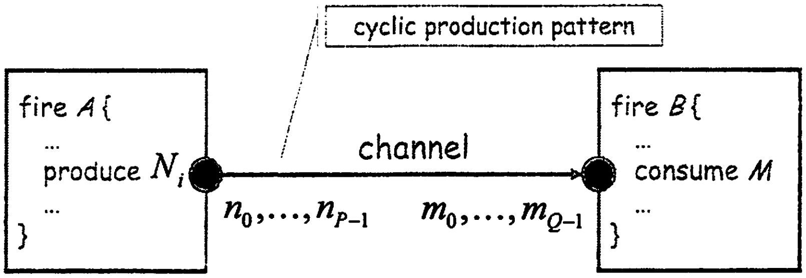
Cyclostatic Dataflow (CSDF)

(Lauwereins et al., TU Leuven, 1994)

Actors cycle through a regular production/consumption pattern.

Balance equations become:

$$q_A \sum_{i=0}^{R-1} n_{i \bmod P} = q_B \sum_{i=0}^{R-1} m_{i \bmod Q};\ R = lcm(P,Q)$$



cyclic production pattern

fire *A* {

...

produce N_i

...

}

channel

$n_0, \ldots, n_{P-1}$

$m_0, \ldots, m_{Q-1}$

fire *B* {

...

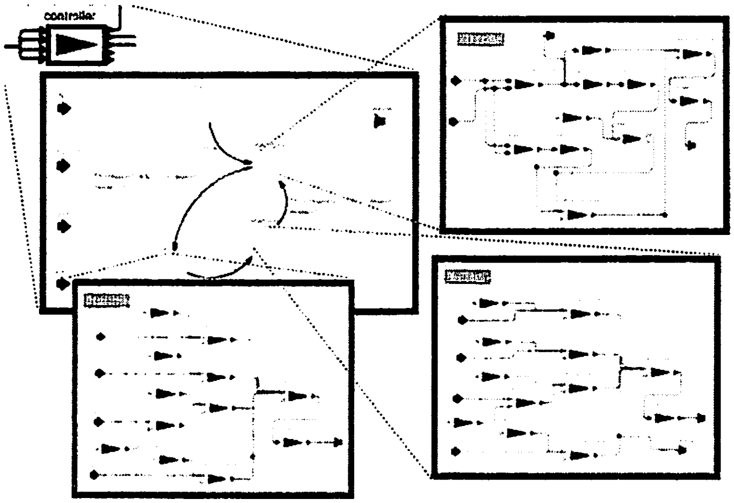
consume *M*

...

}

Heterochronous Dataflow (HDF)

(Girault, Lee, & Lee, 1997)



An actor consists of a state machine and refinements to the states that define behavior.

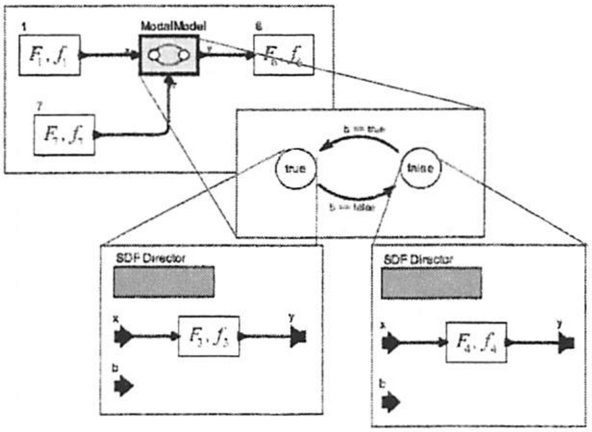
Heterochronous Dataflow (HDF)

(Girault, Lee, and Lee, 1997)

Related to "parameterized dataflow" of Bhattacharya and Bhattacharyya (2001).

- An interconnection of actors.
- An actor is either SDF or HDF.
- If HDF, then the actor has:
  - a state machine
  - a refinement for each state
  - where the refinement is an SDF or HDF actor
- Operational semantics:
  - with the state of each state machine fixed, graph is SDF
  - in the initial state, execute one complete SDF iteration
  - evaluate guards and allow state transitions
  - in the new state, execute one complete SDF iteration
- HDF is decidable if state machines are finite
  - but complexity can be high

If-Then-Else Using Heterochronous Dataflow



Imperative equivalent:

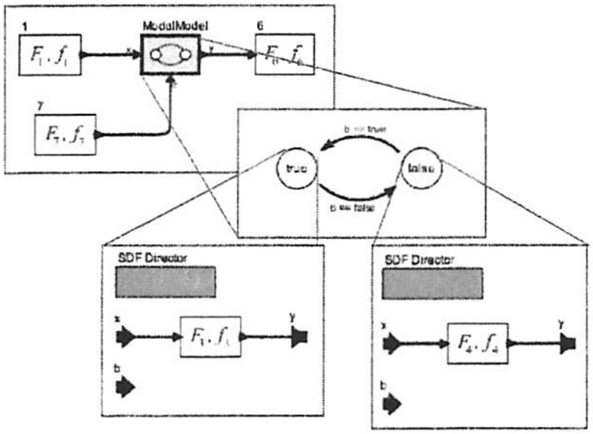
```
b = true;
while (true) {
    x = f1();
    if (b) {
      y = f3(x);
    } else {
      y = f4(x);
    }
    f6(y);
    b = f7();
}
```

Semantics of HDF:

-Execute SDF model for one complete iteration in current state

-Take state transitions to get a new SDF model.

If-Then-Else Using Heterochronous Dataflow

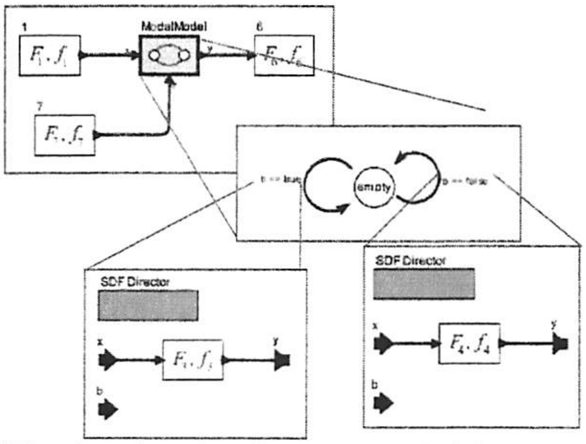


Imperative equivalent:

```
b = true;
while (true) {
    x = f1();
    if (b) {
      y = f3(x);
    } else {
      y = f4(x);
    }
    f6(y);
    b = f7();
}
```

Note that if these two refinements have the same production/consumption parameters, then this is simply hierarchical SDF, where one static schedule suffices.

Hierarchical SDF Using Transition Refinements



Imperative equivalent:

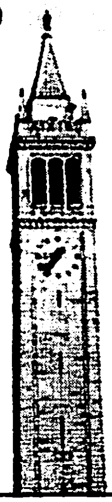
```
while (true) {
    x = f1();
    b = f7();
    if (b) {
      y = f3(x);
    } else {
      y = f4(x);
    }
    f6(y);
}
```

This only works under rather narrow constraints:

- Exactly one outgoing transition from any state is enabled.
- The transition refinements on all transitions have the same production/consumption patterns.
- The state has no refinement.

Conclusions and Open Issues

- Generalizations to SSDF improve expressiveness while preserving decidability.
- Usable languages for many of these extensions have yet to be created.



Concurrent Models of Computation for Embedded Software

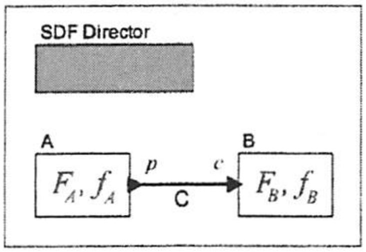
Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 18: Boolean Dataflow

History of Dataflow Models of Computation

- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980's]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993] **today**
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- Parameterized dataflow [Bhattacharya and Bhattacharyya 2001] **Friday**
- ...

Statically Schedulable Dataflow – SSDF
Historically called: Synchronous Dataflow (SDF)



If the number of tokens consumed and produced by the firing of an actor is constant, then static analysis can tell us whether we can schedule the firings to get a useful execution, and if so, then a finite representation of a schedule for such an execution can be created.

Expressiveness Limitations in SSDF

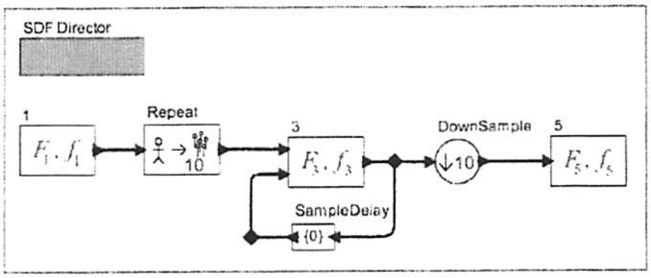
SSDF cannot express data-dependent flow of tokens:

- If-then-else
- Do-while
- Recursion

Hierarchical SSDF can do some of this…

A more general solution is dynamically scheduled dataflow. We now explore DDF, and in particular, how to use static analysis to achieve similar results to those of SSDF.

Manifest Iteration in SSDF

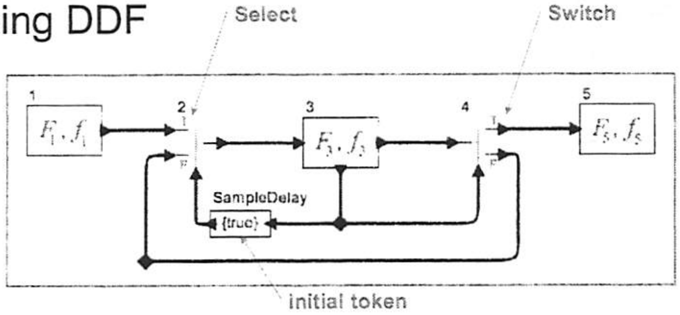


Imperative equivalent:

```
while (true) {
   x = f1();
   y = 0;
   for i in (1..10) {
      y = f3(x, y);
   }
   f5(y);
}
```

Manifest iteration (where the number of iterations is a fixed constant) is expressible in SSDF. But data-dependent iteration is not.

Do-While Using DDF



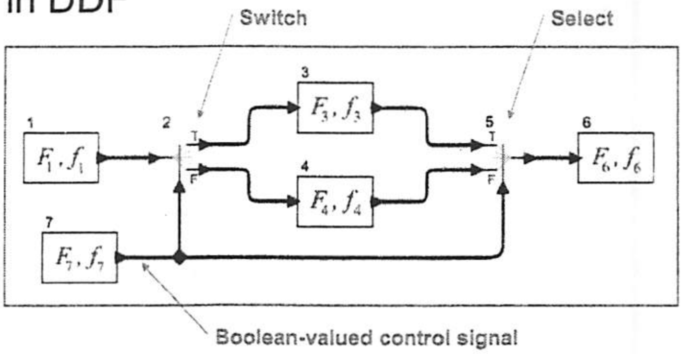
Imperative equivalent:

```
while (true) {
   x = f1();
   b = false;
   while(!b) {
      (x, b) = f3(x);
   }
   f5(x);
}
```

This model uses conditional routing of tokens to iterate a function a data-dependent number of times.

Exercise: Can this be done with HDF? Hierarchical SDF?

If-Then-Else in DDF



Switch

Select

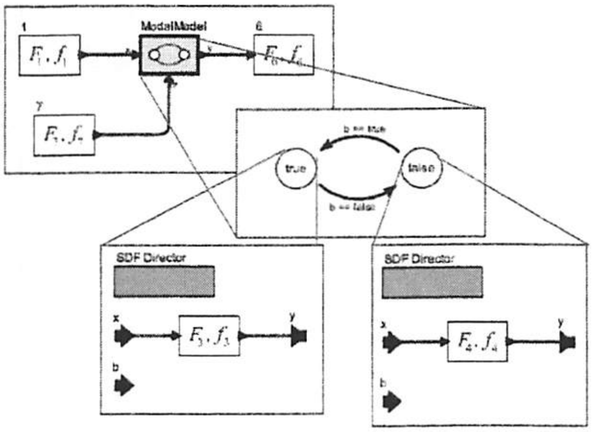
Boolean-valued control signal

Imperative equivalent:

```
while (true) {
   x = f1();
   b = f7();
   if (b) {
     y = f3(x);
   } else {
     y = f4(x);
   }
   f6(y);
}
```

This model uses conditional routing of tokens to route each token in a stream through one of two actors.

Aside: Compare With If-Then-Else Using Heterochronous Dataflow



Imperative equivalent:

```
b = true;
while (true) {
    x = f1();
    if (b) {
      y = f3(x);
    } else {
      y = f4(x);
    }
    f6(y);
    b = f7();
}
```

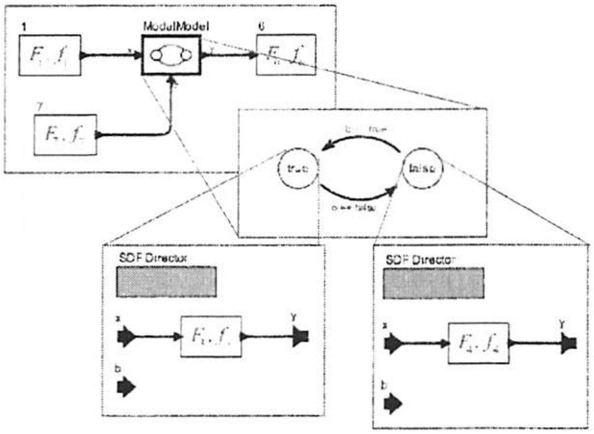
Note that this is not quite the same as the previous version…

Semantics of HDF:

-Execute SDF model for one complete iteration in current state

-Take state transitions to get a new SDF model.

Aside: Compare With
If-Then-Else Using Heterochronous Dataflow

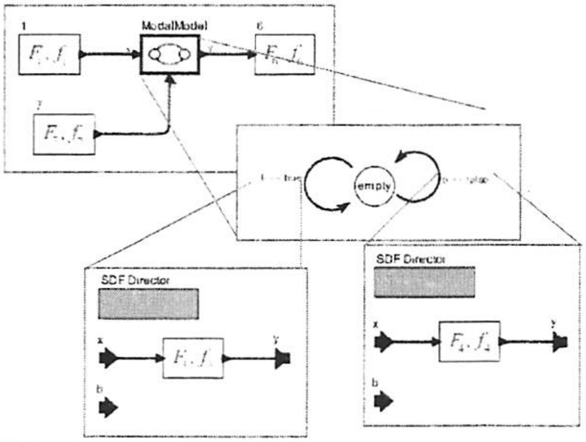


Imperative equivalent:

```
b = true;
while (true) {
    x = f1();
    if (b) {
        y = f3(x);
    } else {
        y = f4(x);
    }
    f6(y);
    b = f7();
}
```

Note that if these two refinements have the same production/consumption parameters, then this is simply hierarchical SDF, where one static schedule suffices.

Hierarchical SDF Using Transition Refinements



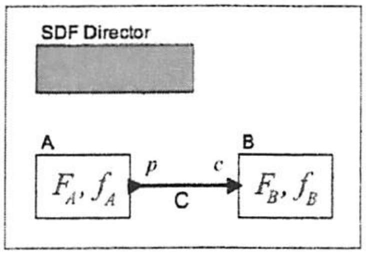
Imperative equivalent:

```
while (true) {
    x = f1();
    b = f7();
    if (b) {
        y = f3(x);
    } else {
        y = f4(x);
    }
    f6(y);
}
```

This only works under rather narrow constraints:

- Exactly one outgoing transition from any state is enabled.
- The transition refinements on all transitions have the same production/consumption patterns.
- The state has no refinement.

Balance Equations



Let q_A, q_B be the number of firings of actors A and B.

Let p_C, c_C be the number of token produced and consumed on a connection C.

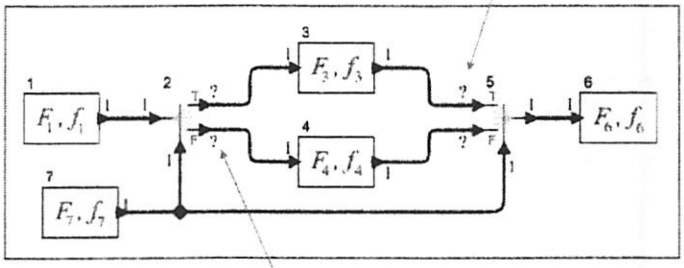
Then the system is *in balance* if for all connections C

$$q_A p_C = q_B c_C$$

where A produces tokens on C and B consumes them.

If-Then-Else in DDF

What consumption rate?



What production rate?

Imperative equivalent:

```
while (true) {
    x = f1();
    b = f7();
    if (b) {
      y = f3(x);
    } else {
      y = f4(x);
    }
    f6(y);
}
```

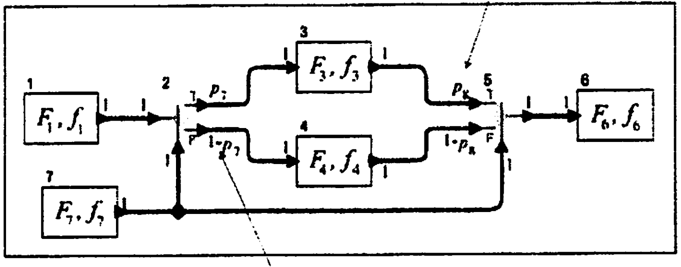
The if-then-else model is not SDF. But we can clearly give a bounded *quasi-static* schedule for it:

(1, 7, 2, b?3, !b?4, 5, 6)

guard

Symbolic Rates

Symbolic consumption rate.



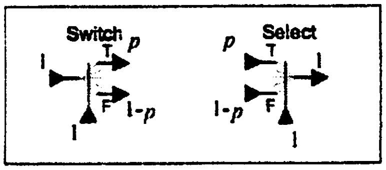
Symbolic production rate.

Imperative equivalent:

```
while (true) {
   x = f1();
   b = f7();
   if (b) {
     y = f3(x);
   } else {
     y = f4(x);
   }
   f6(y);
}
```

Production and consumption rates are given symbolically in terms of the values of the Boolean control signals consumed at the control port.

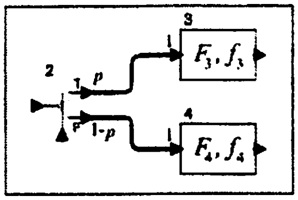
Interpretations of Symbolic Rates



- General interpretation: p is a symbolic placeholder for an unknown.
- Probabilistic interpretation: p is the probability that a Boolean control input is *true.*
- Proportion interpretation: p is the proportion of *true* values at the control input in one *complete cycle*.

NOTE: We do not need numeric values for p. We always manipulate it symbolically.

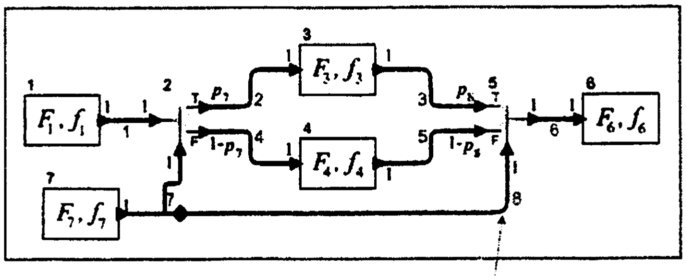
Symbolic Balance Equations



The two connections above imply the following balance equations:

$$q_2 p = q_3$$
$$q_2 (1-p) = q_4$$

Symbolic Rates



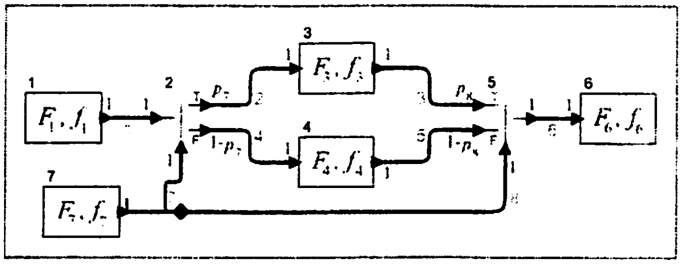
Label the arcs

Imperative equivalent:

```
while (true) {
   x = f1();
   b = f7();
   if (b) {
      y = f3(x);
   } else {
      y = f4(x);
   }
   f6(y);
}
```

Production and consumption rates are given symbolically in terms of the values of the Boolean control signals consumed at the control port.

Production/Consumption Matrix for If-Then-Else



Symbolic variables:

$$\vec{p} = \begin{bmatrix} p_7 \\ p_8 \end{bmatrix}$$

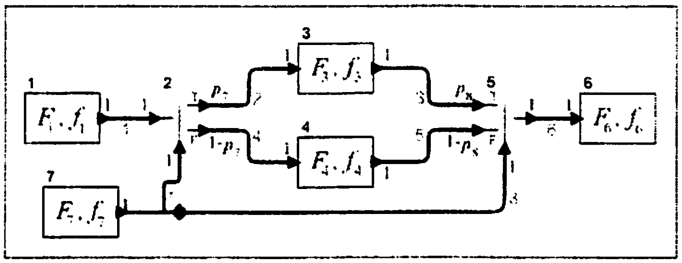
$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & p_7 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -p_8 & 0 & 0 \\ 0 & 1-p_7 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -(1-p_8) & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

Balance equations:

$$\Gamma(\vec{p})q(\vec{p}) = \vec{0}$$

Note that the solution $q(\vec{p})$ now depends on the symbolic variables $\vec{p}$

Production/Consumption Matrix for If-Then-Else



The balance equations have a solution $q(\vec{p})$ if an only if $\Gamma(\vec{p})$ has rank 6. This occurs if and only if $p_7 = p_8$, which happens to be true by construction because signals 7 and 8 come from the same source. The solution is given at the right.

$$q(\vec{p}) = \begin{bmatrix} 1 \\ 1 \\ p_7 \\ 1-p_7 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Strong and Weak Consistency

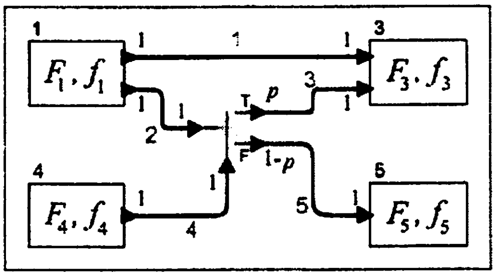
A *strongly consistent* dataflow model is one where the balance equations have a solution that is provably valid without concern for the values of the symbolic variables.

- **The if-then-else dataflow model is strongly consistent.**

A *weakly consistent* dataflow model is one where the balance equations cannot be proved to have a solution without constraints on the symbolic variables that cannot be proved.

- **Note that whether a model is strongly or weakly consistent depends on how much you know about the model.**

Weakly Consistent Model

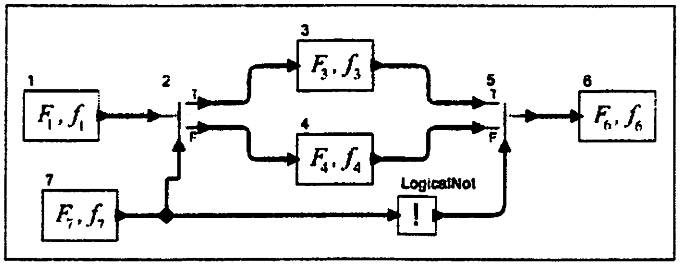


$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & p & -1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 1-p & 0 & 0 & -1 \end{bmatrix}$$

This production/consumption matrix has full rank unless $p = 1$.

Unless we know f_4, this cannot be verified at compile time.

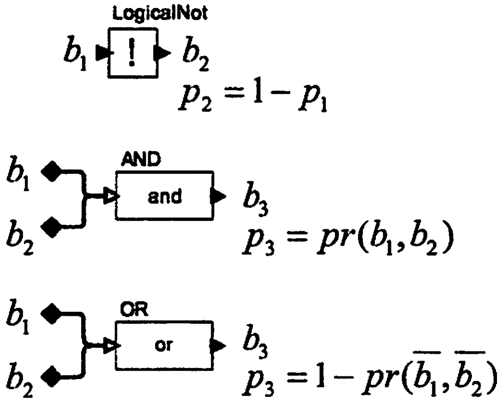
Another Example of a Weakly Consistent Model



This one requires that actor 7 produce half true and half false (that $p = 0.5$) to be consistent. This fact is derived automatically from solving the balance equations.

Use Boolean Relations

Symbolic variables across logical operators can be related as shown.



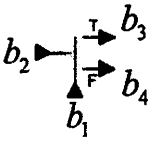
$p_2 = 1 - p_1$

$p_3 = pr(b_1, b_2)$

$p_3 = 1 - pr(\overline{b_1}, \overline{b_2})$

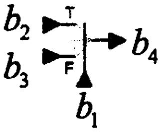
Routing of Boolean Tokens

Symbolic variables across switch and select can be related as shown.



$$p_3 = pr(b_2 \mid b_1)$$

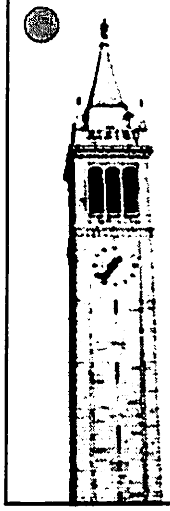
$$p_4 = pr(b_2 \mid \overline{b_1})$$



$$p_4 = pr(b_2 \mid b_1) + pr(b_3 \mid \overline{b_1})$$

Conclusions and Open Issues

- BDF generalizes the idea of balance equations to include symbolic variables.
- Whether balance equations have a solution may depend on the relationships between symbolic variables.



Concurrent Models of Computation for Embedded Software

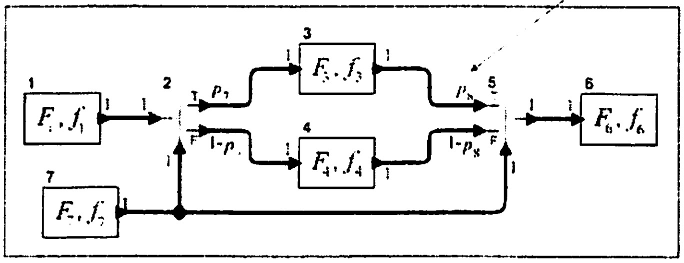
Edward A. Lee

Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 19: Scheduling Boolean Dataflow

Recall If-Then-Else Pattern

Symbolic consumption rate.



Solution to the symbolic balance equations:

$$q(\bar{p}) = \begin{bmatrix} 1 \\ 1 \\ p_7 \\ 1-p_7 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The if-then-else model is strongly consistent and we can give a *quasi-static* schedule for it:

(1, 7, 2, b?3, !b?4, 5, 6)

guard

$$\bar{p} = \begin{bmatrix} p_7 \\ p_8 \end{bmatrix}$$

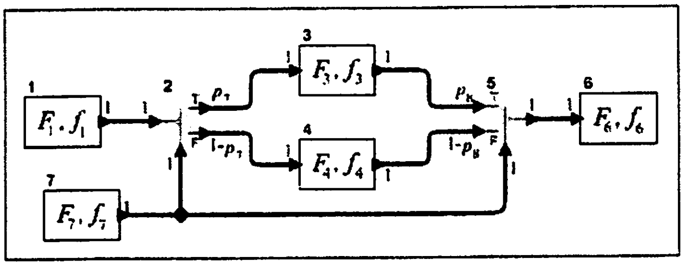
Quasi-Static Schedules & Traces

A *quasi-static schedule* is a finite list of guarded firings where:

- The number of tokens on each arc after executing the schedule is the same as before, regardless of the outcome of the Booleans.
- If any arc has a Boolean token prior to the execution of the schedule, then it will have a Boolean token with the same value after execution of the schedule.
- Firing rules are satisfied at every point in the schedule.

A *trace* is a particular execution sequence.

Quasi-Static Schedules & Traces

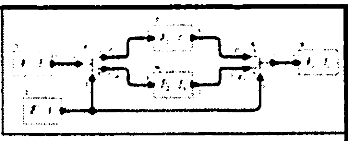


Solution to the symbolic balance equations:

$$q(\vec{p}) = [1 \quad 1 \quad p_7 \quad 1-p_7 \quad 1 \quad 1 \quad 1]^T$$

Quasi-static schedule: (1, 7, 2, b?3, !b?4, 5, 6)
Possible trace: (1, 7, 2, 3, 5, 6)
Another possible trace: (1, 7, 2, 4, 5, 6)

Proportion Vectors



- Let S be a trace. E.g. (1, 7, 2, 3, 5, 6)
- Let q_S be a repetitions vector for S. E.g.
$$q_S = [1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1]^T$$
- Let $t_{i,S}$ be the number of TRUEs consumed from Boolean stream b_i in S. E.g. $t_{7,S} = 1,\ t_{8,S} = 1.$
- Let $n_{i,S}$ be the number of tokens consumed from Boolean stream b_i in S. E.g. $n_{7,S} = 1,\ n_{8,S} = 1.$
- Let
$$\vec{p}_S = \begin{bmatrix} t_{7,S} / n_{7,S} \\ t_{8,S} / n_{8,S} \end{bmatrix} \longleftarrow \text{proportion vector}$$
- We want a quasi-static schedule s.t. for every trace S we have $\Gamma(\vec{p}_S)q_S = \vec{0}$.

Proportion Interpretation

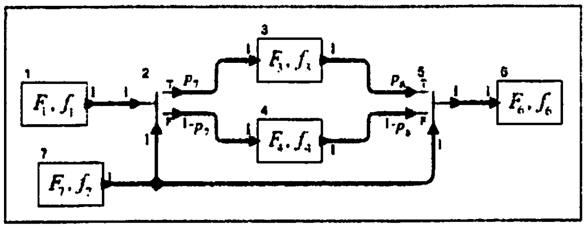
Recall the balance equations depend on $\vec{p}$, a vector with one symbolic variable for each Boolean stream that affects consumption production rates:

$$\Gamma(\vec{p})q(\vec{p}) = \vec{0}$$

Under a proportion interpretation, for a trace S, $\vec{p}_S$ represents the *proportion* of TRUEs in S. We seek a schedule that always yields traces that satisfy

$$\Gamma(\vec{p}_S)q_S = \vec{0}$$

Proportion Interpretation for If-Then-Else



$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & p_7 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -p_8 & 0 & 0 \\ 0 & 1-p_7 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -(1-p_8) & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

Quasi-static schedule: (1, 7, 2, b?3, !b?4, 5, 6)

Possible trace: $S = (1, 7, 2, 3, 5, 6)$

$$\vec{p} = [1 \quad 1]^T \qquad q_S = [1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1]^T$$

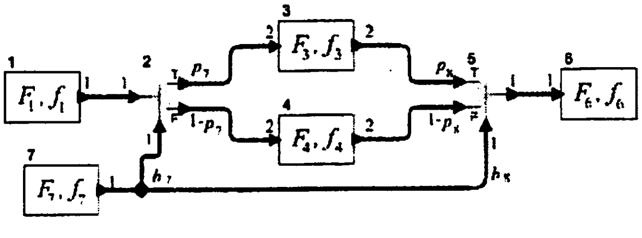
Another possible trace: (1, 7, 2, 4, 5, 6)

$$\vec{p} = [0 \quad 0]^T \qquad q_S = [1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1]^T$$

Both satisfy the balance equations.

Limitations of Consistency

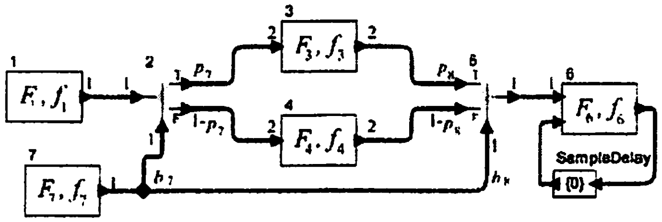
Consistency is necessary but not sufficient for a dataflow graph to have a bounded-memory schedule. Consider:



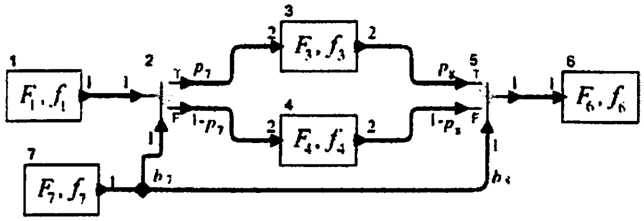
[Gao et al. '92]. This model is strongly consistent. But there is no bounded schedule (e.g., suppose $b_7 = (F, T, T, T, \ldots)$.

Limitations of Consistency

Even out-of-order execution (as supported by tagged-token scheduling [Arvind et al.] doesn't solve the problem:



Gao's Example has no Quasi-Static Schedule



Solution to the symbolic balance equations is

$$q(\bar{p}) = [2 \quad 2 \quad p_7 \quad 1-p_7 \quad 2 \quad 2 \quad 2]^T$$

A trace S with N firings (N even) of actor 1 must have

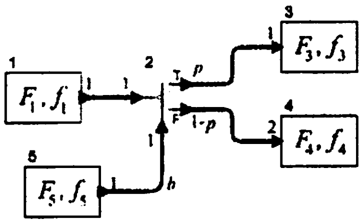
$$q_S = [N \quad N \quad t_{7,S}/2 \quad (N-t_{7,S})/2 \quad N \quad N \quad N]^T$$

But this cannot be unless $t_{7,S}$ is even. There is no assurance of this.

Another Example

The model is strongly consistent.
Solution to symbolic equations:

$$q(\bar{p}) = [2 \quad 2 \quad 2p \quad 1-p \quad 2]^T$$



A trace S with N firings (N even) of actor 1 must have:

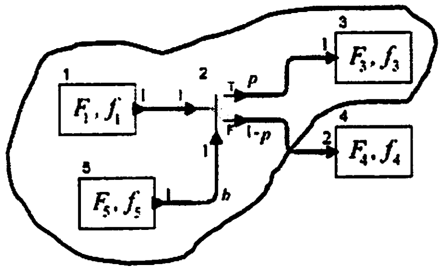
$$q_S = [N \quad N \quad t \quad (N-t)/2 \quad N]^T$$

where t is the number of TRUEs consumed. There is no finite N where this is assured of being an integer vector.

Clustered Quasi-Static Schedules

Consider the *clustered schedule:*

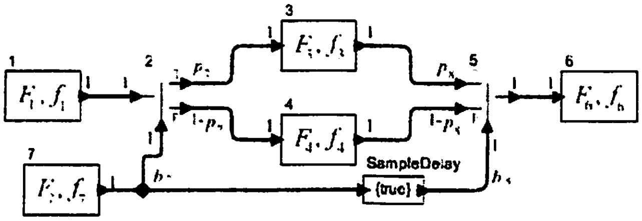
```
n = 0;
do {
    fire 1;
    fire 5;
    fire 2;
    if (b) {
       fire 3;
    } else {
       n += 1;
    }
} while (n < 2);
fire 4;
```



This schedule either fails to terminate or yields an integer vector of the form:

$$q_S = [N \quad N \quad t \quad (N-t)/2 \quad N]^T$$

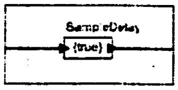
Delays Can Also Cause Trouble



This model is weakly consistent, where the balance equations have a non-trivial solution only if $p_7 = p_8$, in which case the solution is:

$$q(\vec{p}) = [1 \quad 1 \quad p_7 \quad 1-p_7 \quad 1 \quad 1 \quad 1]^T$$

Relating Symbolic Variables Across Delays

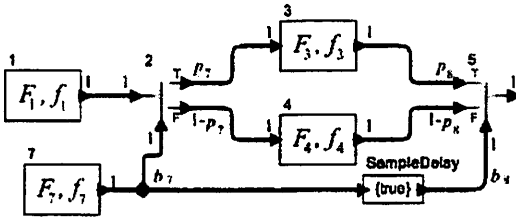
For the sample delay: 

What is the relationship between p_1 and p_2?

Since consistency is about behavior in the limit, under the probabilistic of the interpretation for the symbolic variables, it is reasonable to assume $p_1 = p_2$.

Is this reasonable under the proportion interpretation?

Delays Cause Trouble with the Proportion Interpretation



Solution to the symbolic balance equations is

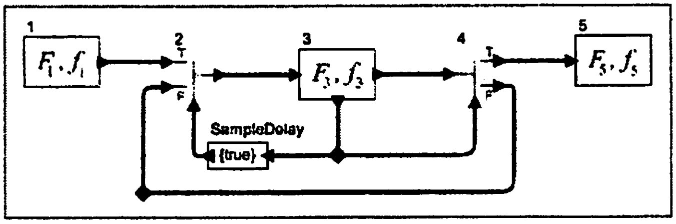
$$q(\vec{p}) = [1 \quad 1 \quad p_7 \quad 1-p_7 \quad 1 \quad 1 \quad 1]^T$$

A trace S with N firings of actor 1 must have

$$q_S = [N \quad N \quad t_{7,S} \quad (N - t_{7,S}) \quad N \quad N \quad N]^T$$

But for no value of N is there any assurance of being able to fire actor 5 N times. This schedule won't work.

Do-While Relies on a Delay

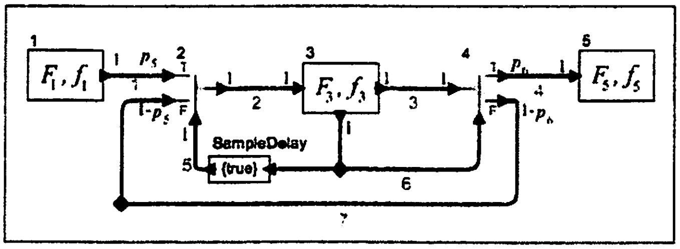


Imperative equivalent:

```
while (true) {
   x = f1();
   b = false;
   while(!b) {
      (x, b) = f3(x);
   }
   f5(x);
}
```

Is this model strongly consistent? Weakly consistent? Inconsistent?

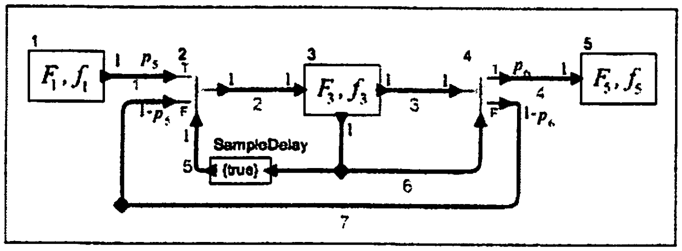
Checking Consistency of Do-While



$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & -p_5 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & p_6 & -1 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & -(1-p_5) & 0 & 1-p_6 & 0 \end{bmatrix}$$

This model is consistent if and only if $p_5 = p_6$, which is true under the probabilistic interpretation, but not under the proportion interpretation.

Checking Consistency of Do-While

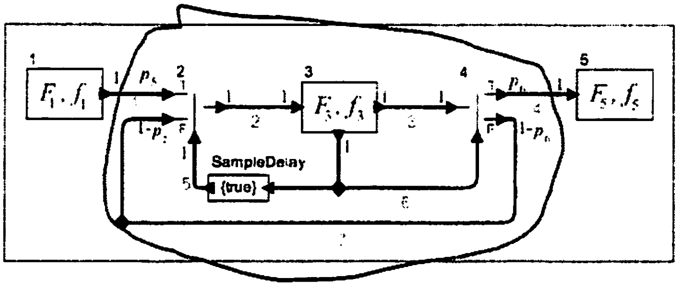


$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & -p_5 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & p_6 & -1 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & -(1-p_5) & 0 & 1-p_6 & 0 \end{bmatrix}$$

Let $p = p_5 = p_6$, then the solution to the balance equations is:

$$q(\vec{p}) = [1 \quad 1/p \quad 1/p \quad 1/p \quad 1]^T$$

Clustering Solution for Do-While



Clustered Schedule:

```
fire 1;
do {
    fire 2;
    fire 3;
    fire 4;
} while(!b);
fire 5;
```

This schedule yields traces S for which $p_5 = p_6 = 1/N$ and

$$q_S = [1 \quad N \quad N \quad N \quad 1]^T$$

compare:

$$q(\vec{p}) = [1 \quad 1/p \quad 1/p \quad 1/p \quad 1]^T$$

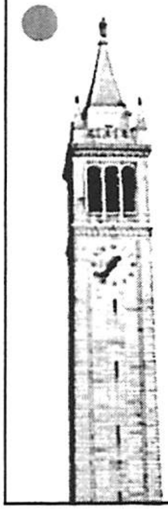
Extensions

- State enumeration scheduling approach: Seek a finite set of finite guarded schedules that leave the model in a finite set of states (buffer states), and for which there is a schedule starting from each state.
- Integer dataflow (IDF [Buck '94]): Allow symbolic variables to have integer values, not just Boolean values. Extension is straightforward in concept, but reasoning about consistency becomes harder.

Conclusions and Open Issues

- BDF and IDF generalize the idea of balance equations and introduce *quasi-static scheduling.*
- BDF and IDF are Turing complete, so existence of quasi-static schedules is undecidable.
- Can often construct quasi-static schedules anyway.
- Tricks like clustered schedules make the set of manageable models larger.
- Are Switch and Select like unrestricted GOTO?

- **Fully usable languages have yet to be created.**

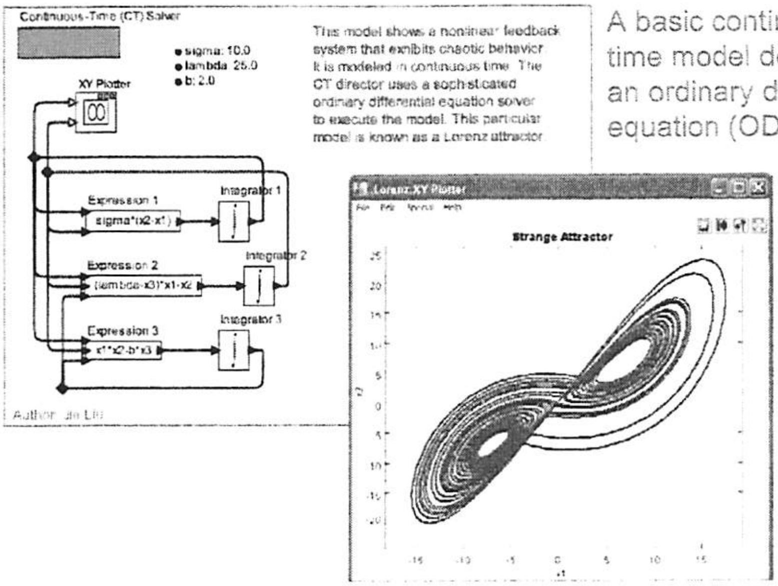


Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 20: Continuous-Time Models

Basic Continuous-Time Modeling

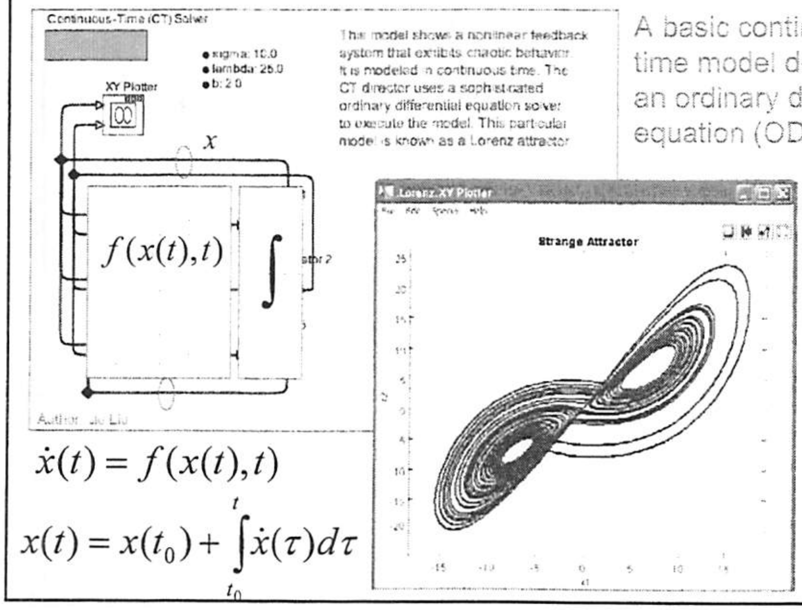


- sigma: 10.0
- lambda: 25.0
- b: 2.0

This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

A basic continuous-time model describes an ordinary differential equation (ODE).

Basic Continuous-Time Modeling



Continuous-Time (CT) Solver

- sigma: 10.0
- lambda: 25.0
- b: 2.0

This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor

A basic continuous-time model describes an ordinary differential equation (ODE).

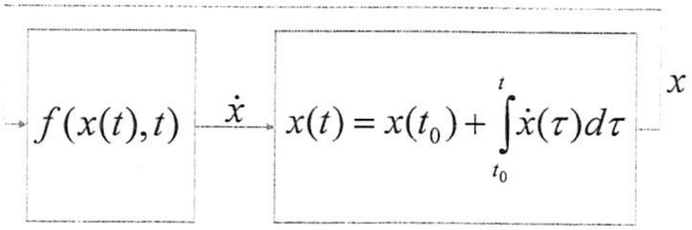
$$\dot{x}(t) = f(x(t),t)$$

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau)d\tau$$

Basic Continuous-Time Modeling

The state trajectory is modeled as a vector function of time,

$$x : T \to R^n \qquad T = [t_0, \infty) \subset R$$



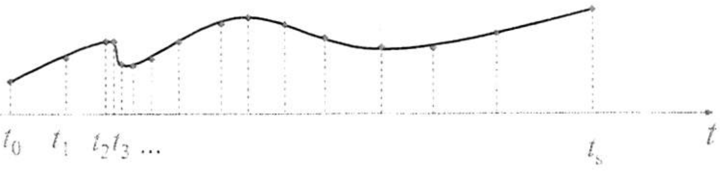
$$\dot{x}(t) = f(x(t),t)$$

$$f : R^m \times T \to R^m$$

ODE Solvers

Numerical solution approximates the state trajectory of the ODE by estimating its value at discrete time points:

$$\{t_0, t_1, \ldots\} \subset T$$



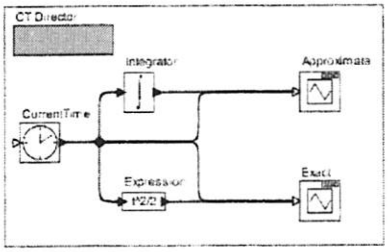
Reasonable choices for these points depend on the function *f*.

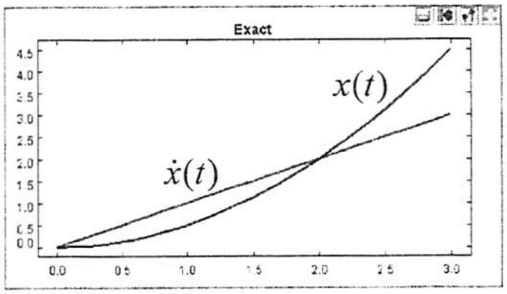
Using such solvers, signals are discrete-event signals.

Simple Example

This simple example integrates a ramp, generated by the CurrentTime actor. In this case, it is easy to find a closed form solution,

$$\dot{x}(t) = t \;\Rightarrow\; x(t) = t^2 / 2$$

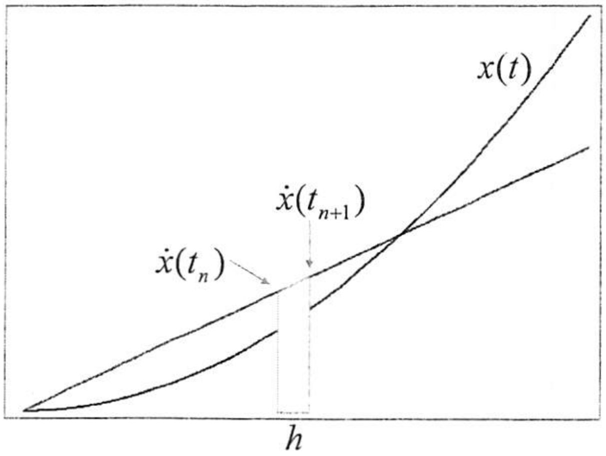




Trapezoidal Method

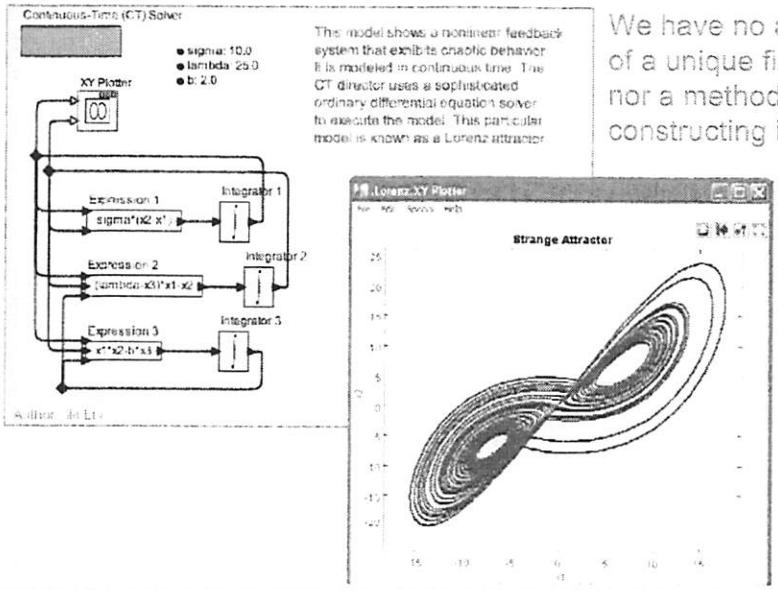
Classical method estimates the area under the curve by calculating the area of trapezoids.

However, with this method, an integrator is only causal, not strictly causal or delta causal.



$$x(t_{n+1}) = x(t_n) + h(\dot{x}(t_n) + \dot{x}(t_{n+1}))/2$$

Trapezoidal Method is Problematic with Feedback



- sigma: 10.0
- lambda: 25.0
- b: 2.0

This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

We have no assurance of a unique fixed point, nor a method for constructing it.

Forward Euler Solver

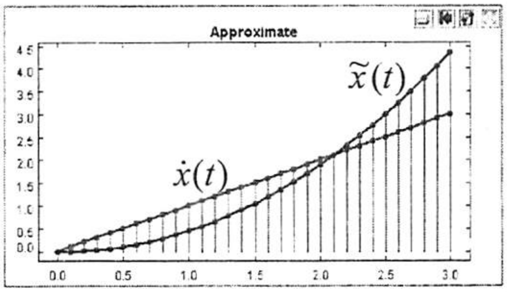
Given $x(t_n)$ and a time increment h, calculate:

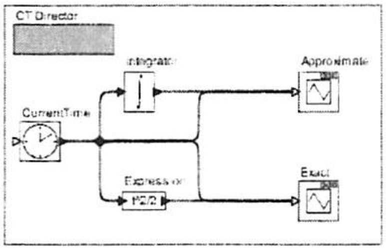
$$t_{n+1} = t_n + h$$
$$x(t_{n+1}) = x(t_n) + h\, f(x(t_n), t_n)$$

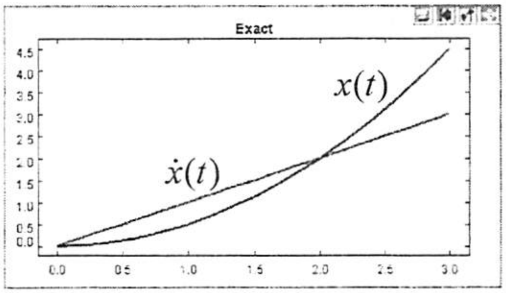
This method is strictly causal, or, with a lower bound on the step size h, delta causal. It can be used in feedback systems. The solution is unique an non-Zeno.

Forward Euler on Simple Example

In this case, we have used a fixed step size $h = 0.1$. The result is close, but diverges over time.







Runge-Kutta 2-3 Solver (RK2-3)

Given $x(t_n)$ and a time increment h, calculate

$$K_0 = f(x(t_n), t_n) \longleftarrow \dot{x}(t_n)$$

$$K_1 = f(x(t_n) + 0.5hK_0, t_n + 0.5h) \longleftarrow \text{estimate of } \dot{x}(t_n + 0.5h)$$

$$K_2 = f(x(t_n) + 0.75hK_1, t_n + 0.75h) \longleftarrow \text{estimate of } \dot{x}(t_n + 0.75h)$$

then let

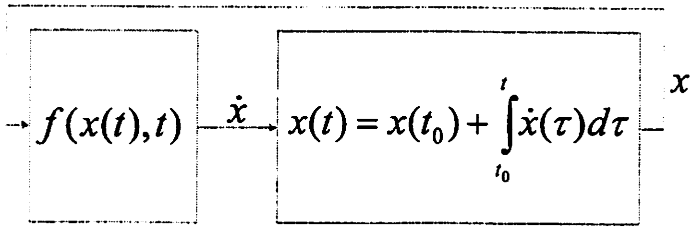
$$t_{n+1} = t_n + h$$

$$x(t_{n+1}) = x(t_n) + (2/9)hK_0 + (3/9)hK_1 + (4/9)hK_2$$

Note that this is strictly (delta) causal, but requires three evaluations of f at three different times with three different inputs.

Operational Requirements

In a software system, the blue box below can be specified by a program that. given $x(t)$ and t calculates $f(x(t), t)$. But this requires that the program be functional (have no side effects).



$$\dot{x}(t) = f(x(t), t)$$

$$f : R^m \times T \to R^m$$

Adjusting the Time Steps

For time step given by $t_{n+1} = t_n + h$, let

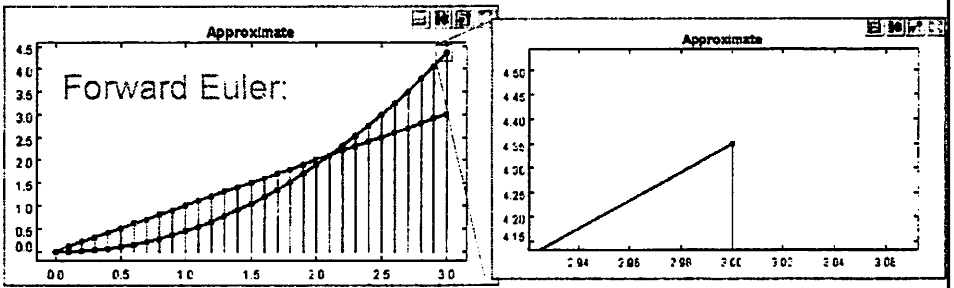
$$K_3 = f(x(t_{n+1}), t_{n+1})$$
$$\varepsilon = h((-5/72)K_0 + (1/12)K_1 + (1/9)K_2 + (-1/8)K_3)$$

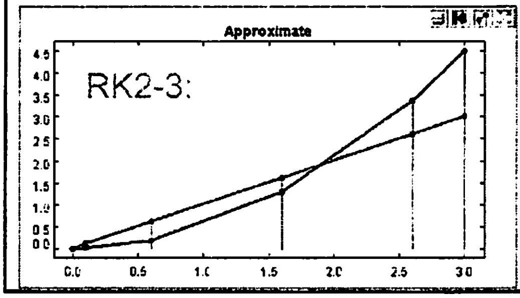
If ε is less than the "error tolerance" e, then the step is deemed "successful" and the next time step is estimated at:

$$h' = 0.8\sqrt[3]{e/\varepsilon}$$

If ε is greater than the "error tolerance," then the time step h is reduced and the whole thing is tried again.

Comparing RK2-3 to Forward Euler

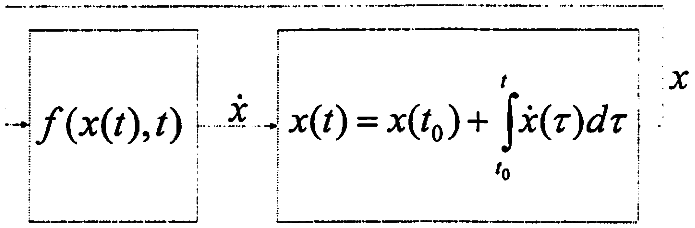




For this example, RK2-3 is exact at 3.0, while Forward Euler undershoots by a significant amount.

Accumulating Errors

In feedback systems, the errors of FE accumulate more rapidly than those of RK2-3.



$f(x(t),t)$

$\dot{x}$

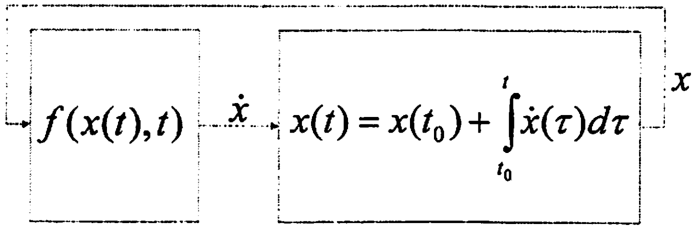
$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau) d\tau$

x

$$\dot{x}(t) = f(x(t),t)$$

$$f : R^m \times T \to R^m$$

Examining This Computationally



$f(x(t),t)$

$\dot{x}$

$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau) d\tau$

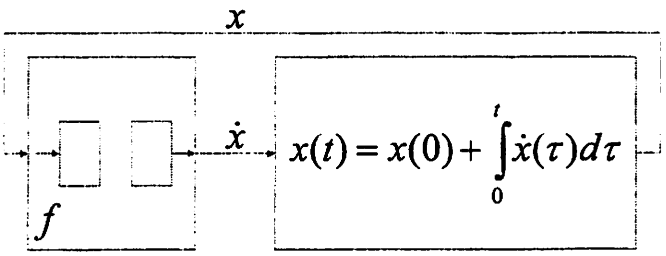
x

At each discrete time t_n, given a time increment $t_{n+1} = t_n + h$, we can estimate $x(t_{n+1})$ by repeatedly evaluating f with different values for the arguments. We may then decide that h is too large and reduce it and redo the process.

How General Is This Model?

Does it handle:

Systems without feedback? yes

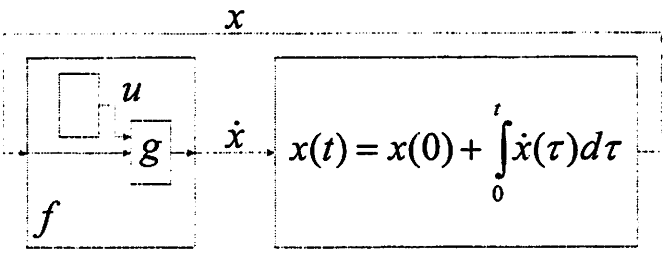


$$\dot{x}(t) = f(x(t), t)$$

How General Is This Model?

Does it handle:

External inputs? yes



$$\dot{x}(t) = f(x(t), t) = g(u(t), x(t), t)$$

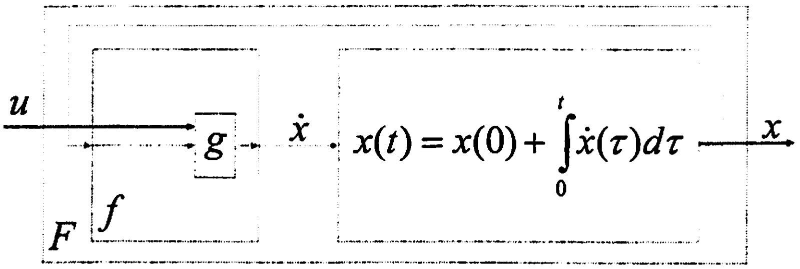
The Model Itself as a Function

Note that the model function has the form:

$$F:[T \to R^m] \to [T \to R^m]$$

Which does not match the form:

$$f: R^m \times T \to R^m$$

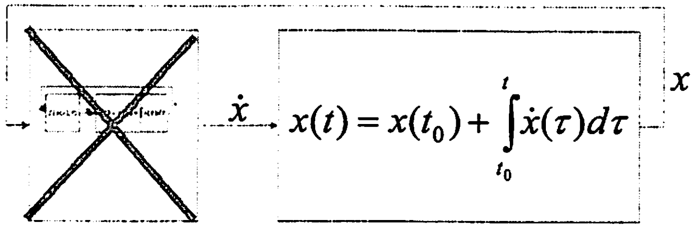


(This assumes certain technical requirements on *f* and u that ensure existence and uniqueness of the solution.)

Consequently, the Model is Not Compositional!

In general, the behavior of the inside dynamical system cannot be given by a function of form:

$$f: R^m \times T \to R^m$$

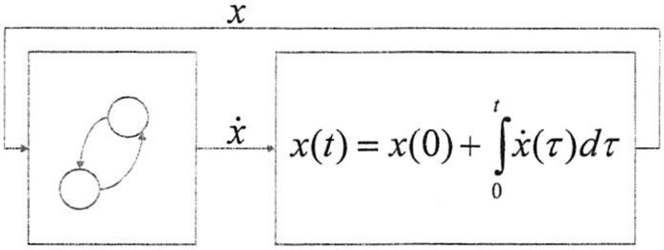


To see this, just note that the output must depend only on the current value of the input and the time to conform with this form.

So How General Is This Model?

Does it handle:

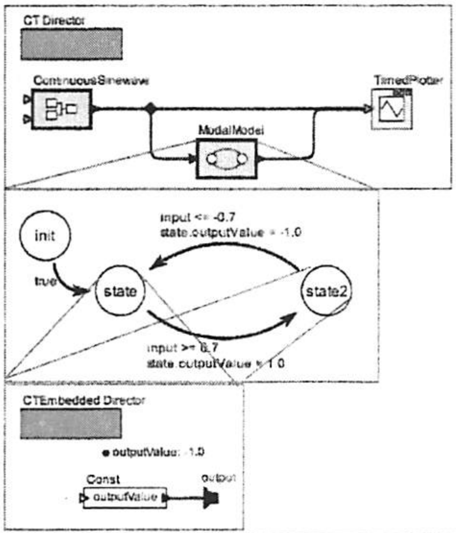
- State machines? No… The model needs work…



$$x(t) = x(0) + \int_0^t \dot{x}(\tau)d\tau$$

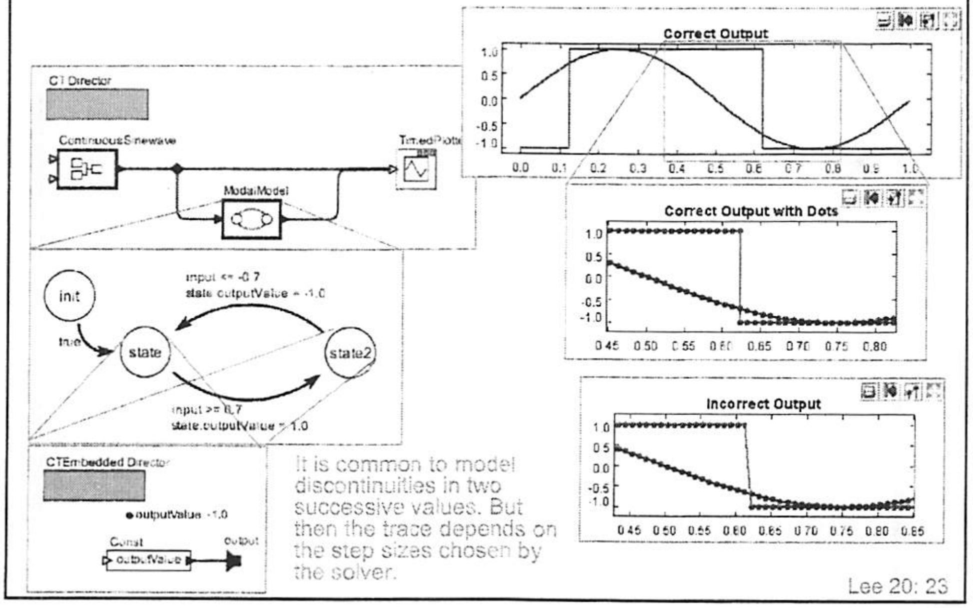
Since this model is itself a state machine, the inability to put a state machine in the left box explains the lack of composability.

Start with Simple State Machines Hysteresis Example



This model shows the use of a two-state FSM to model hysteresis. Semantically, the output of the ModalModel block is discontinuous. If transitions take zero time, this is modeled as a signal that has two values *at the same time*, and *in a particular order*.

Hysteresis Example



It is common to model discontinuities in two successive values. But then the trace depends on the step sizes chosen by the solver.

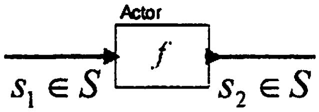
Requirements

The hysteresis example illustrates two requirements:

- A signal may have more than one value at a particular time, and the values it has have an order.
- The times at which the solver evaluates signals must precisely include the times at which interesting events happen, like a guard becoming true.

Both Requirements Are Dealt With By an Abstract Semantics

Previously

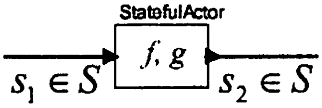


$S = [T \to R]$

$f : R^m \times T \to R^m$

$\forall t \in T, \quad s_2(t) = f(s_1(t), t)$

Now we need:



$S = [T \times N \to R]$

$f : \Sigma \times R^m \times T \to R^m$

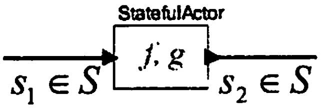
$g : \Sigma \times R^m \times T \to \Sigma$

state space

$\forall (t,n) \in T \times N, \quad s_2(t,n) = ?$

The new function f gives outputs in terms of inputs and the current state. The function g updates the state at the specified time.

Abstract Semantics



$S = [T \times N \to R]$

$f : \Sigma \times R^m \times T \to R^m$

$g : \Sigma \times R^m \times T \to \Sigma$

At each $t \in T$ the output is a *sequence* of one or more values where given the current state $\sigma(t) \in \Sigma$ and the input $s_1(t)$ we evaluate the procedure

$$s_2(t,0) = f(\sigma(t), s_1(t,0), t)$$
$$\sigma_1(t) = g(\sigma(t), s_1(t,0), t)$$
$$s_2(t,1) = f(\sigma_1(t), s_1(t,1), t)$$
$$\sigma_2(t) = g(\sigma_1(t), s_1(t,1), t)$$

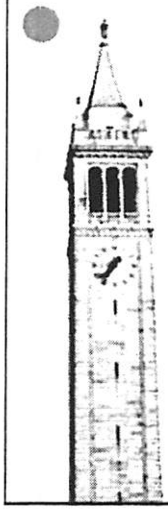
...

until the state no longer changes. We use the final state on any evaluation at later times.

This deals with the first requirement.

Conclusion and Open Issues

- The basic model assumed by many ODE solvers does not lend itself easily to reasonable software architectures.
- A generalized model supports signals with multiple, ordered values at a time value.
- An abstract semantics for components can be defined that supports these multiple values and also is amenable to reasonable software realizations.
- Compositionality remains an open issue.



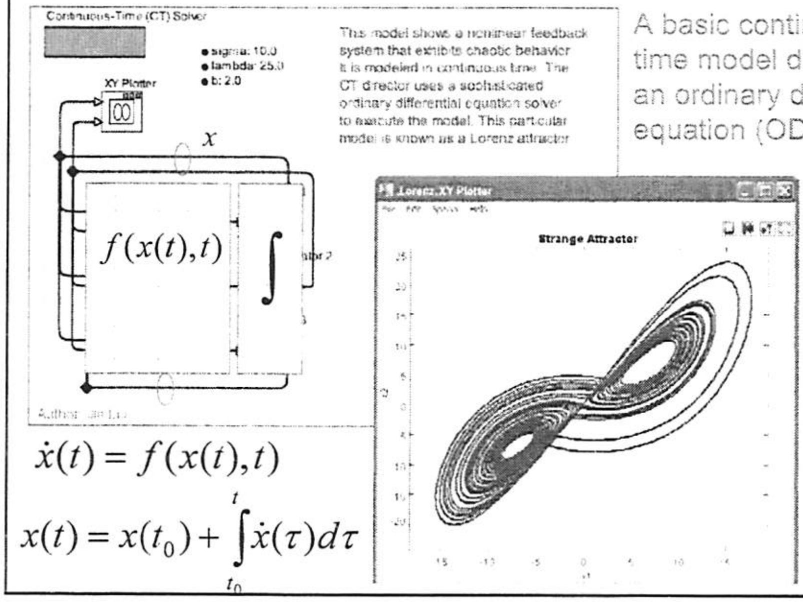
Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley

EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 21: Mixed Signal Models and Hybrid Systems

Basic Continuous-Time Modeling



This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

A basic continuous-time model describes an ordinary differential equation (ODE).

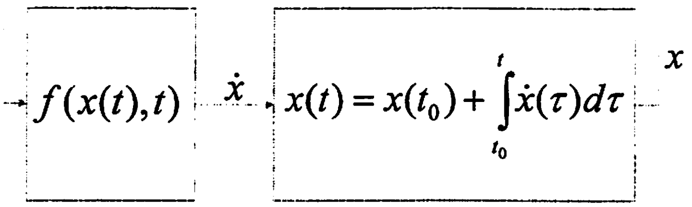
$$\dot{x}(t) = f(x(t),t)$$

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau)d\tau$$

Basic Continuous-Time Modeling

The state trajectory is modeled as a vector function of time,

$$x : T \to R^n \qquad T = [t_0, \infty) \subset R$$



$$f(x(t),t) \xrightarrow{\dot{x}} x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau)d\tau \rightarrow x$$

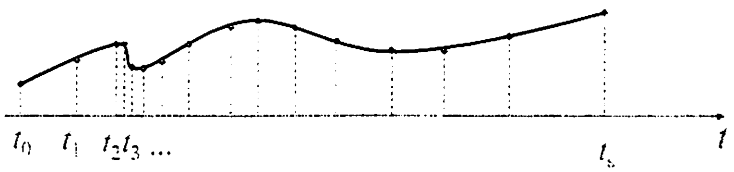
$$\dot{x}(t) = f(x(t),t)$$

$$f : R^m \times T \to R^m$$

ODE Solvers

Numerical solution approximates the state trajectory of the ODE by estimating its value at discrete time points:

$$\{t_0, t_1, \ldots\} \subset T$$



Reasonable choices for these points depend on the function *f*.

Using such solvers, signals are discrete-event signals.

Requirements

We have two requirements:

- A signal may have more than one value at a particular time, and the values it has have an order.
- The times at which the solver evaluates signals must precisely include the times at which interesting events happen, like a guard becoming true, or any point of discontinuity in a signal (a time where it has more than one value).

Ideal Solver Semantics

Given an interval $I = [t_i, t_{i+1}]$ and an initial value $x(t_i)$ and a function $f : R^m \times T \to R^m$ that is Lipschitz in x on the interval (meaning that there exists an $L \geq 0$ such that

$$\forall t \in I, \quad \|f(x(t),t) - f(x'(t),t)\| \leq L\|x(t) - x'(t)\|$$

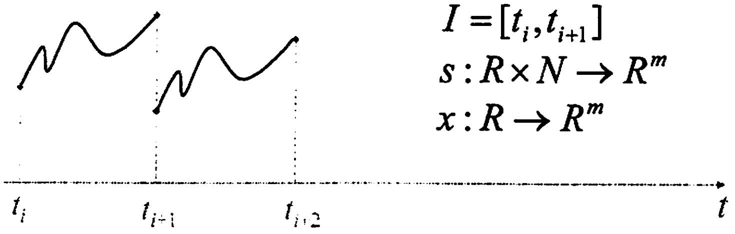
then the following equation has a unique solution x satisfying the initial condition where

$$\forall t \in I, \quad \dot{x}(t) = f(x(t),t)$$

The ideal solver yields the exact value of $x(t_{i+1})$.

Piecewise Lipschitz Systems

In our CT semantics, signals have multiple values at the times of discontinuities. Between discontinuities, a necessary condition that we can impose is that the function *f* be Lipschitz, where we choose the points at the discontinuities to ensure this:



$$I = [t_i, t_{i+1}]$$
$$s : R \times N \to R^m$$
$$x : R \to R^m$$

RK2-3 Solver Approximates Ideal Solver

Given $x(t_n)$ and a time increment h, calculate

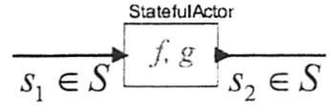
$$K_0 = f(x(t_n), t_n) \longleftarrow \dot{x}(t_n)$$
$$K_1 = f(x(t_n) + 0.5hK_0, t_n + 0.5h) \longleftarrow \text{estimate of } \dot{x}(t_n + 0.5h)$$
$$K_2 = f(x(t_n) + 0.75hK_1, t_n + 0.75h) \longleftarrow \text{estimate of } \dot{x}(t_n + 0.75h)$$

then let

$$t_{n+1} = t_n + h$$
$$x(t_{n+1}) = x(t_n) + (2/9)hK_0 + (3/9)hK_1 + (4/9)hK_2$$

Note that this is strictly (delta) causal, but requires three evaluations of *f* at three different times with three different inputs.

Abstract Semantics



$$S = [T \times N \to R]$$
$$f : \Sigma \times R^m \times T \to R^m$$
$$g : \Sigma \times R^m \times T \to \Sigma$$

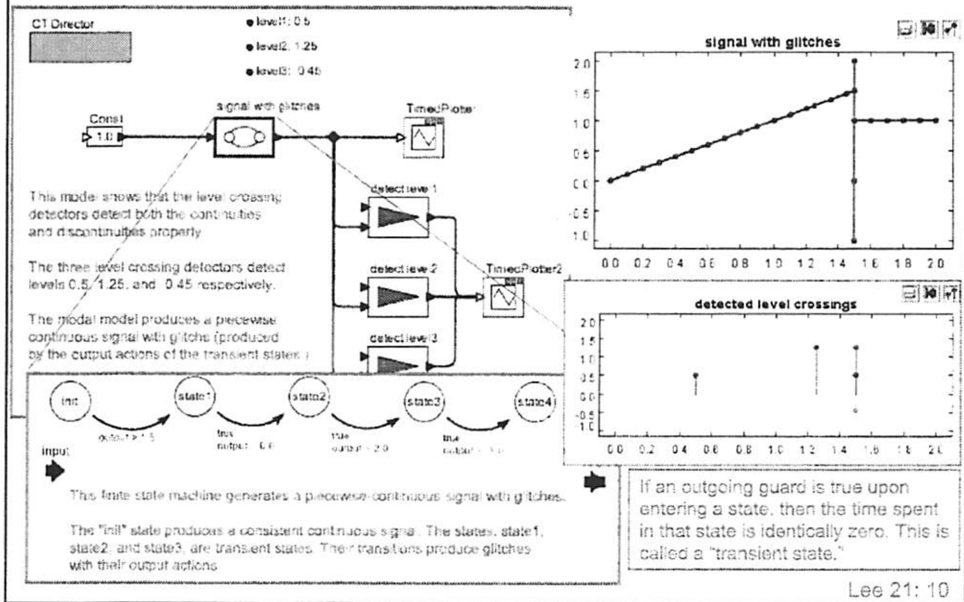
At each $t \in T$ the output is a *sequence* of one or more values where given the current state $\sigma(t) \in \Sigma$ and the input $s_1(t)$ we evaluate the procedure

$$s_2(t,0) = f(\sigma(t), s_1(t), t)$$
$$\sigma_1(t) = g(\sigma(t), s_1(t), t)$$
$$s_2(t,1) = f(\sigma_1(t), s_1(t), t)$$
$$\sigma_2(t) = g(\sigma_1(t), s_1(t), t)$$
$$\dots$$

until the state no longer changes. We use the final state on any evaluation at later times.

This deals with the first requirement.

Generalizing: Multiple Events at the Same Time using Transient States



This model shows that the level crossing detectors detect both the continuities and discontinuities properly.

The three level crossing detectors detect levels 0.5, 1.25, and -0.45 respectively.

The modal model produces a piecewise continuous signal with glitchs (produced by the output actions of the transient states.)

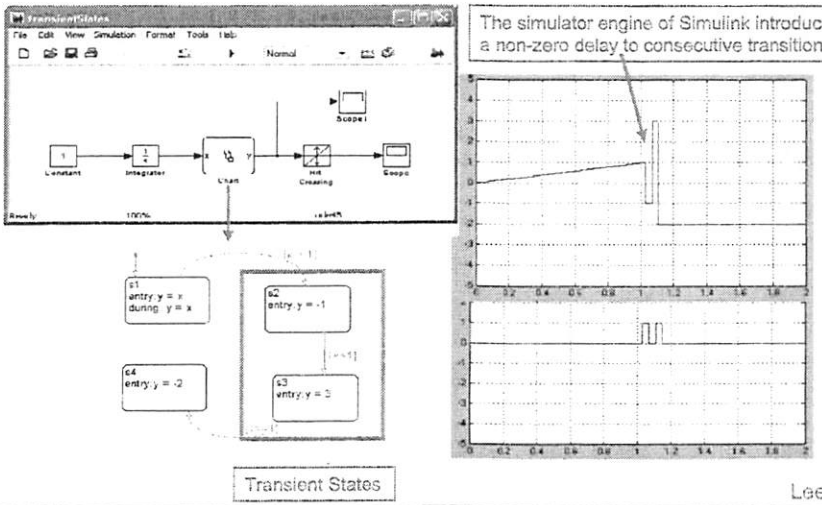
This finite state machine generates a piecewise-continuous signal with glitches.

The "init" state produces a consistent continuous signal. The states, state1, state2, and state3, are transient states. Their transitions produce glitches with their output actions.

If an outgoing guard is true upon entering a state, then the time spent in that state is identically zero. This is called a "transient state."

Contrast with Simulink/Stateflow

In Simulink semantics, a signal can only have one value at a given time. Consequently, Simulink introduces solver-dependent behavior.



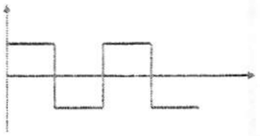
The simulator engine of Simulink introduces a non-zero delay to consecutive transitions.

Transient States

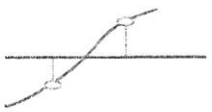
Second Requirement: Simulation Times Must Include Event Times

Event times are sometimes predictable (e.g. the times of discontinuous outputs of a clock) and sometimes unpredictable without running the solver (e.g. the time at which a continuous-time crosses a threshold). In both cases, the solver must not step over the event time.

- Predictable Breakpoints:
  - Known beforehand.
  - Register to a Breakpoint Table in advance.
  - Use breakpoints to adjust step sizes.

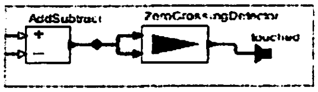


- Unpredictable Breakpoints:
  - Known only after they have been missed.
  - Requires being able to backtrack and re-execute with a smaller step size.

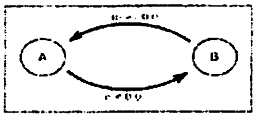


Event Times

In continuous-time models, Ptolemy II can use *event detectors* to identify the precise time at which an event occurs:

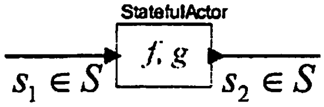


or it can use Modal Models, where guards on the transitions specify when events occur. In the literature, you can find two semantic interpretations to guards: *enabling* or *triggering*.



If only enabling semantics are provided, then it becomes nearly impossible to give models whose behavior does not depend on the step-size choices of the solver.

The Abstract Semantics Supports the Second Requirement as Well



$$S = [T \times N \to R]$$

$$f : \Sigma \times R^m \times T \to R^m$$

$$g : \Sigma \times R^m \times T \to \Sigma$$

At each $t \in T$ the calculation of the output given the input is separated from the calculation of the new state. Thus, the state does not need to updated until after the step size has been decided upon.

In fact, the variable step size solver relies on this, since any of several integration calculations may result in refinement of the step size because the error is too large.

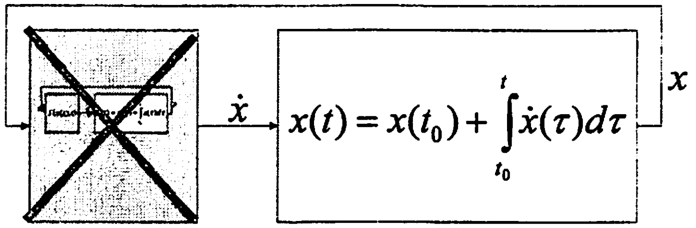
This deals with the second requirement.

However, Getting Compositional Semantics Requires More Work

In general, to give the behavior of the inside solver in the following form requires storing considerable state:

$$f : \Sigma \times R^m \times T \to R^m$$

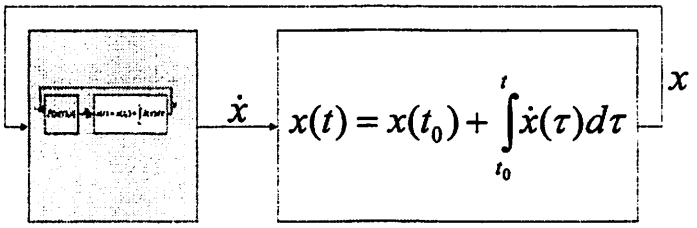
$$g : \Sigma \times R^m \times T \to \Sigma$$



The state space must include the state of all components, since backtracking of the entire subsystem may be required.

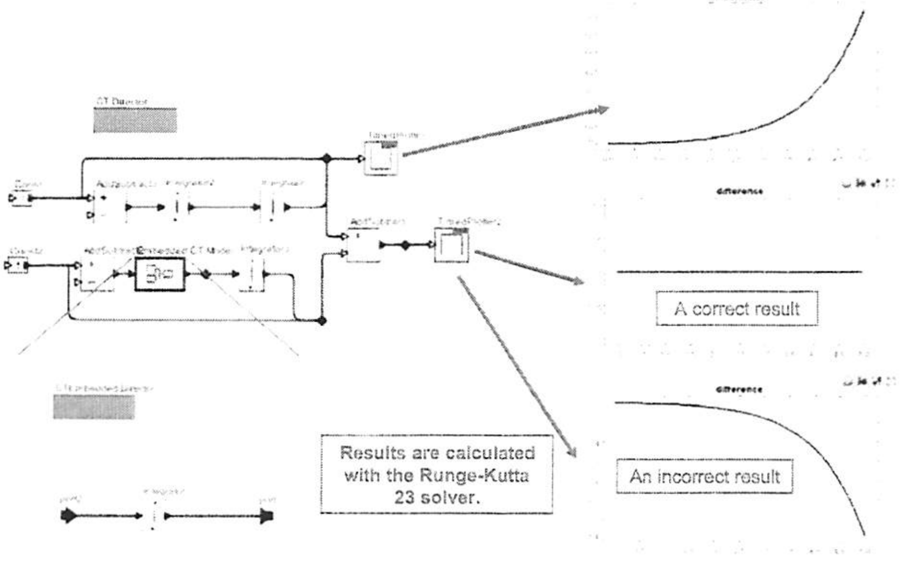
Third Requirement: Compositional Semantics

We require that the system below yield an execution that is identical to a flattened version of the same system. That is, despite having two solvers, it must behave as if it had one.



Achieving this appears to require that the two solvers coordinate quite closely. This is challenging when the hierarchy is deeper.

Hierarchical Executions



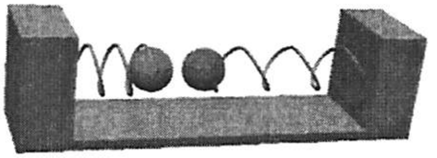
A correct result

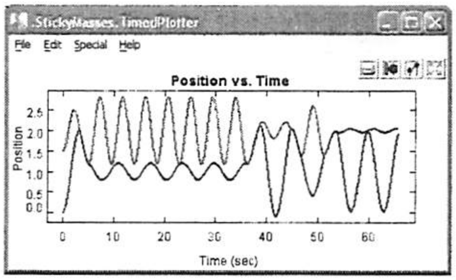
An incorrect result

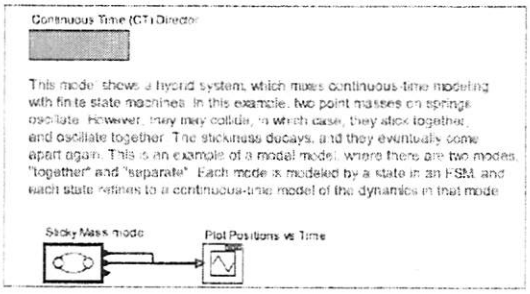
Results are calculated with the Runge-Kutta 23 solver.

The "Right Design" Supports Deeper Hierarchies

Masses on Springs



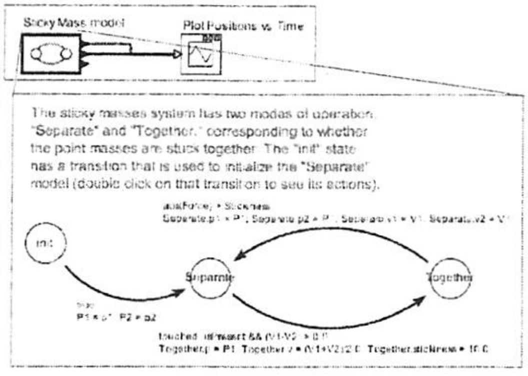




This model shows a hybrid system, which mixes continuous-time modeling with finite state machines. In this example, two point masses on springs oscillate. However, they may collide, in which case, they stick together, and oscillate together. The stickiness decays, and they eventually come apart again. This is an example of a modal model, where there are two modes, "together" and "separate". Each mode is modeled by a state in an FSM and each state refines to a continuous-time model of the dynamics in that mode.

Consider two masses on springs which, when they collide, will stick together with a decaying stickiness until the force of the springs pulls them apart again.

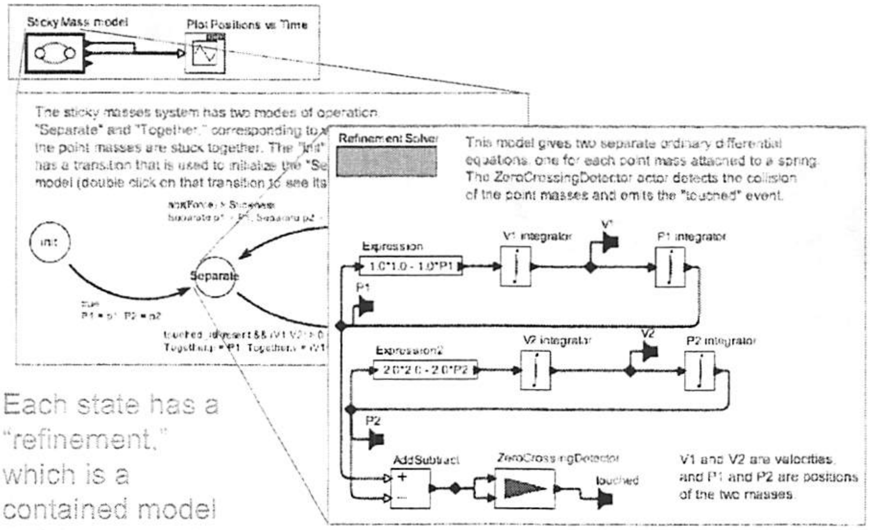
Structure of the Spring-Masses Model



The sticky masses system has two modas of operation. "Separate" and "Together," corresponding to whether the point masses are stuck together. The "init" state has a transition that is used to initialize the "Separate" model (double click on that transition to see its actions).

A component in a continuous-time model is defined by a finite state machine.

Structure of the Spring-Masses Model



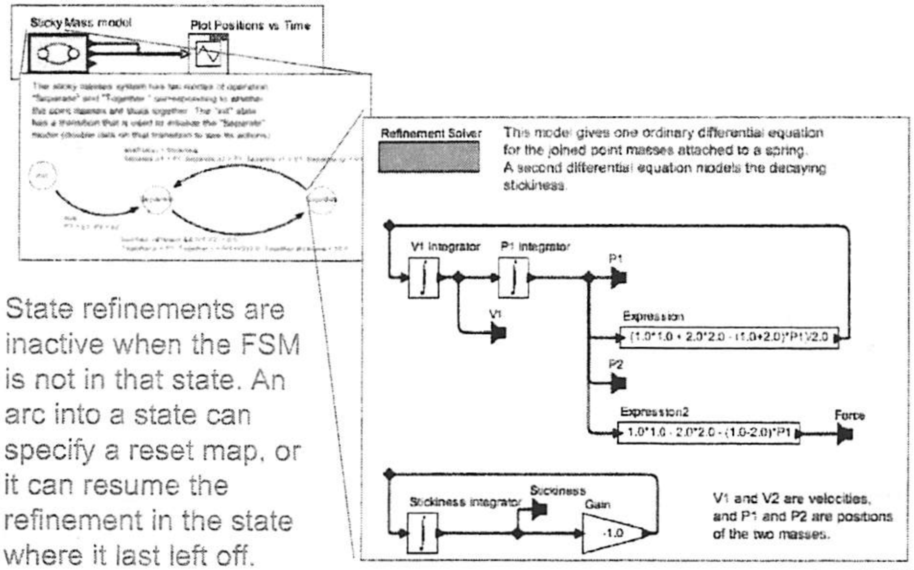
This model gives two separate ordinary differential equations, one for each point mass attached to a spring. The ZeroCrossingDetector actor detects the collision of the point masses and emits the "touched" event.

V1 and V2 are velocities, and P1 and P2 are positions of the two masses.

Each state has a "refinement," which is a contained model defining behavior.

Notice that we need compositionality.

Structure of the Spring-Masses Model



This model gives one ordinary differential equation for the joined point masses attached to a spring. A second differential equation models the decaying stickiness.

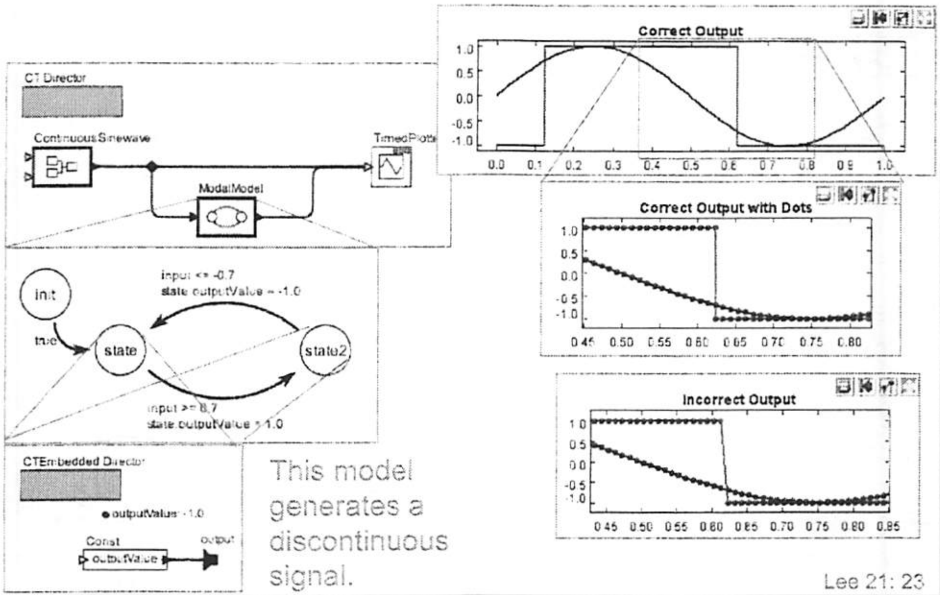
V1 and V2 are velocities, and P1 and P2 are positions of the two masses.

State refinements are inactive when the FSM is not in that state. An arc into a state can specify a reset map, or it can resume the refinement in the state where it last left off.

Consider Corner Cases

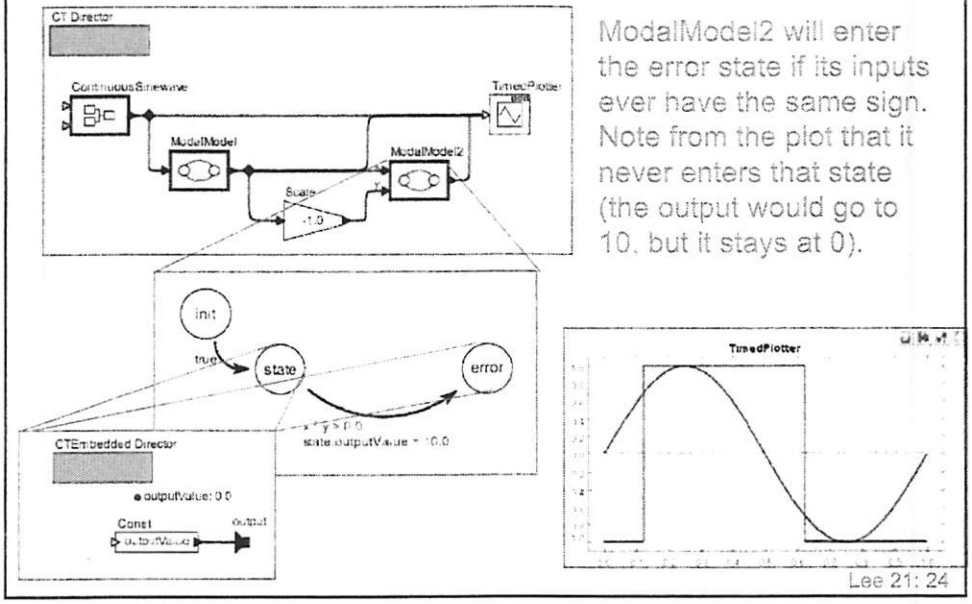
- When triggering transitions based on predicates on discontinuous signals, how should the discontinuity affect the transition?
- What should samples of discontinuous signals be?

Recall Hysteresis Example



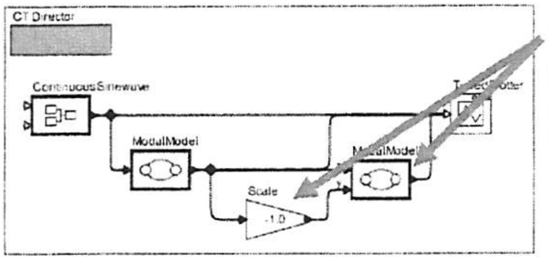
This model generates a discontinuous signal.

Observing the Discontinuous Signal



ModalModel2 will enter the error state if its inputs ever have the same sign. Note from the plot that it never enters that state (the output would go to 10, but it stays at 0).

Simultaneous Events: The Order of Execution Question



The output of the Scale actor has the same tag as its input, so ModelModel2 sees only two values with opposite signs.

Semantics of a signal:

$$s : T \times N \rightarrow R$$

In HyVisual, every continuous-time signal has a value at $(t, 0)$ for any $t \in T$. This yields deterministic execution of the above model.

Alternative Interpretations

- *Nondeterministic*: Some hybrid systems languages (e.g. Charon) declare this to be nondeterministic, saying that perfectly zero time delays never occur anyway in physical systems. Hence, ModalModel2 may or may not see the output of ModalModel before Scale gets a chance to negate it.
- *Delta Delays*: Some models (e.g. VHDL) declare that every block has a non-zero delay in the index space. Thus, ModalModel2 will see an event with time duration zero where the inputs have the same sign.

Disadvantages of These Interpretations

- *Nondeterministic*:
  - Constructing deterministic models is extremely difficult
  - What should a simulator do?
- *Delta Delays*:
  - Changes in one part of the model can unexpectedly change behavior elsewhere in the model.

Nondeterministic Ordering

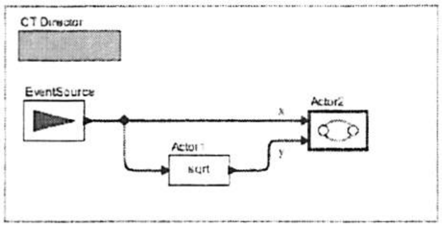
In favor

- Physical systems have no true simultaneity
- Simultaneity in a model is artifact
- Nondeterminism reflects this physical reality

Against

- It surprises the designer
  - counters intuition about causality
- It is hard to get determinism
  - determinism is often desired (to get repeatability)
- Getting the desired nondeterminism is easy
  - build on deterministic ordering with nondeterministic FSMs
- Writing simulators that are trustworthy is difficult
  - It is incorrect to just pick one possible behavior!

Consider Nondeterministic Semantics

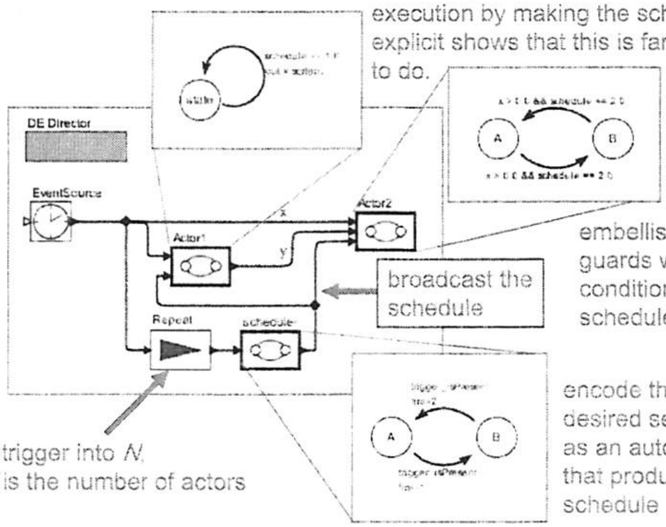


Under nondeterministic semantics, we could modify the model to explicitly schedule the firings.

Suppose we want deterministic behavior in the above (rather simple) model. How could we achieve it?

Non-Deterministic Interaction is the Wrong Answer

An attempt to achieve deterministic execution by making the scheduling explicit shows that this is far too difficult to do.



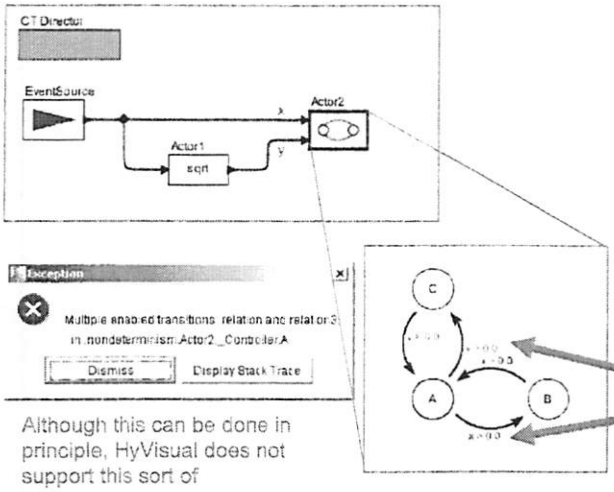
broadcast the schedule

embellish the guards with conditions on the schedule

encode the desired sequence as an automaton that produces a schedule

turn one trigger into *N*, where *N* is the number of actors

OTOH: Nondeterminism is Easily Added in a Deterministic Modeling Framework

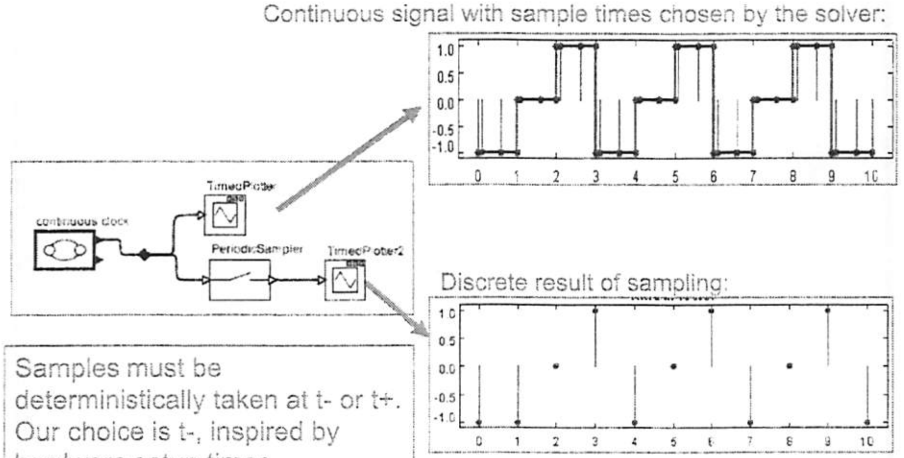


Multiple enabled transitions: relation and relation3 in .nondeterminism.Actor2._Controller.A

Although this can be done in principle, HyVisual does not support this sort of nondeterminism. What execution trace should it give?

At a time when the event source yields a positive number, both transitions are enabled.

Sampling Discontinuous Signals



Continuous signal with sample times chosen by the solver:

Discrete result of sampling:

Samples must be deterministically taken at t- or t+. Our choice is t-, inspired by hardware setup times.

Note that in HyVisual, unlike Simulink, discrete signals have no value except at discrete points.

Conclusion and Open Issues

- Compositionality across levels of the hierarchy appears to require that solvers coordinate rather tightly. Does the abstract semantics adequately support this coordination? Is this abstract semantics implementable in a cost-effective way?
- When considering discontinuous signals, have to consider corner cases... Give them a well-defined semantics, any well-defined semantics!

Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 22: Clocks in Synchronous Languages

Synchronous Languages

- Esterel
- Lustre
- SCADE (visual editor for Lustre)
- Signal
- Statecharts (some variants)
- Ptolemy II SR domain

The model of computation is called *synchronous reactive* (SR). It has strong formal properties (many key questions are decidable).

The Synchronous Abstraction

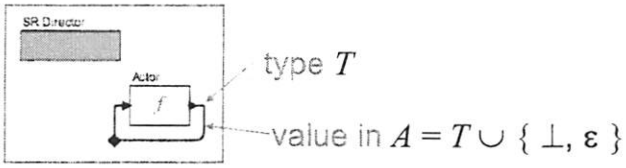
- "Model time" is discrete: Countable ticks of a clock.
- WRT model time, computation does not take time.
- All actors execute "simultaneously" and "instantaneously" (WRT to model time).
- There is an obviously appealing mapping onto real time, where the real time between the ticks of the clock is constant. Good for specifying periodic real-time tasks.

Simple Execution Policy

At each tick, start with all signals "unknown." Evaluate non-strict actors and source actors. Then keep evaluating any actors that can be evaluated until all signals become known or until no further progress can be made.

Note that signals will resolve to a value or to "absent" if there are no causality loops.

Fixed Point Semantics



type T

value in $A = T \cup \{ \perp, \varepsilon \}$

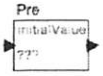
At each tick of the clock

- Start with signal value $\perp$ (unknown)
- Evaluate $f(\perp)$
- Evaluate $f(\,f(\perp))$
- Stop when a fixed point is reached

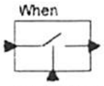
A fixed point is always reached in a finite number of steps (one, in this case).

Synchronous/Reactive Actors

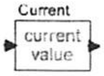
Key SR Actors



Pre: When the input is present, the output is the previous present input value

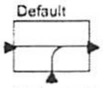


When: When the bottom input is present and true, the output equals the input. Otherwise, the output is absent.

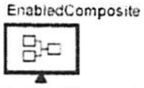


Current: The output equals the most recent present input value.

NonStrictDelay: The output is equal to the input in the previous clock tick.



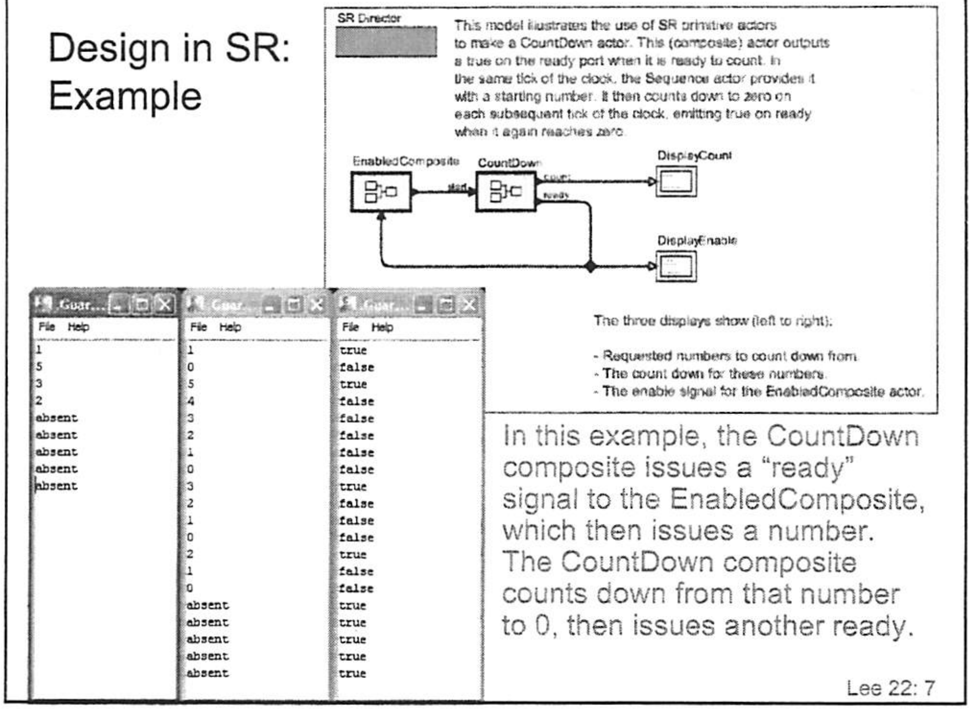
Default: The output equals the left input, if it is present, and the bottom input otherwise.



EnabledComposite: Composite actor whose internal clock ticks only when the bottom input is present and true.

Use of some of these can be quite subtle.

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

The three displays show (left to right):

- Requested numbers to count down from.
- The count down for these numbers.
- The enable signal for the EnabledComposite actor.

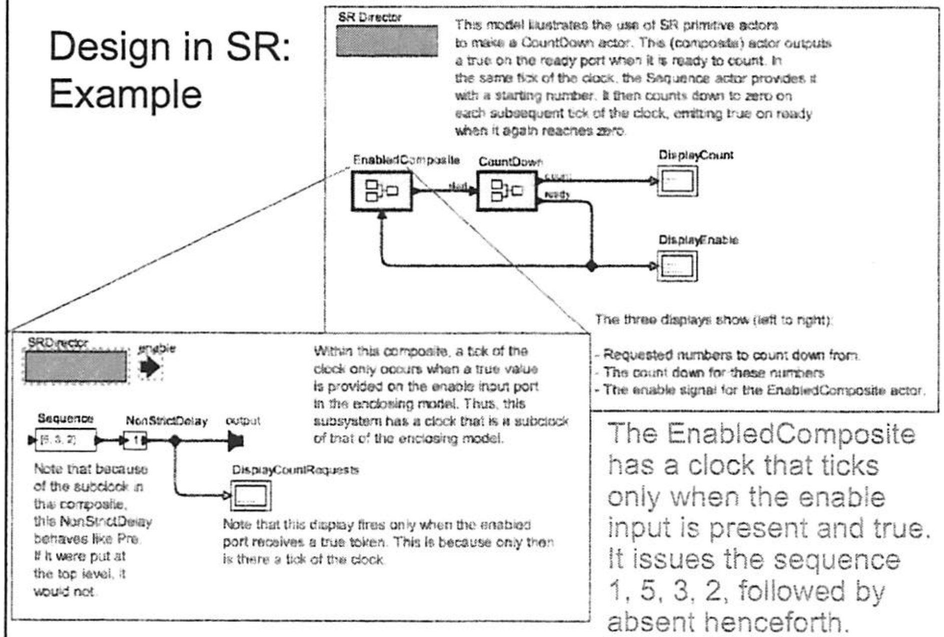
```
1
5
3
2
absent
absent
absent
absent
absent
```

```
1
0
5
4
3
2
1
0
3
2
1
0
2
1
0
absent
absent
absent
absent
absent
```

```
true
false
true
false
false
false
false
false
true
false
false
false
true
false
false
true
true
true
true
true
```

In this example, the CountDown composite issues a "ready" signal to the EnabledComposite, which then issues a number. The CountDown composite counts down from that number to 0, then issues another ready.

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

The three displays show (left to right):

- Requested numbers to count down from.
- The count down for these numbers.
- The enable signal for the EnabledComposite actor.

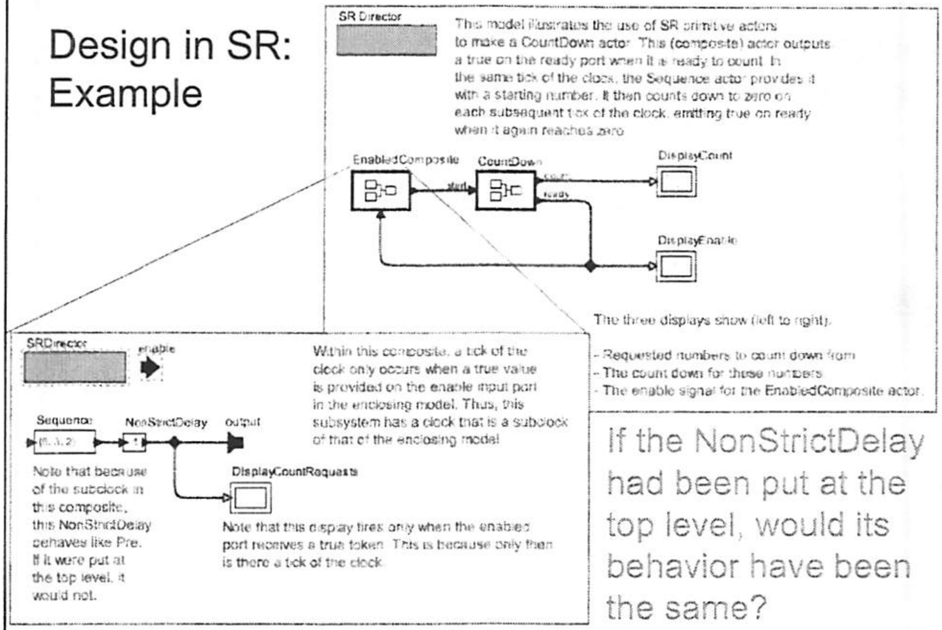
Within this composite, a tick of the clock only occurs when a true value is provided on the enable input port in the enclosing model. Thus, this subsystem has a clock that is a subclock of that of the enclosing model.

Note that because of the subclock in this composite, this NonStrictDelay behaves like Pre. If it were put at the top level, it would not.

Note that this display fires only when the enabled port receives a true token. This is because only then is there a tick of the clock.

The EnabledComposite has a clock that ticks only when the enable input is present and true. It issues the sequence 1, 5, 3, 2, followed by absent henceforth.

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

The three displays show (left to right):

- Requested numbers to count down from
- The count down for these numbers
- The enable signal for the EnabledComposite actor.

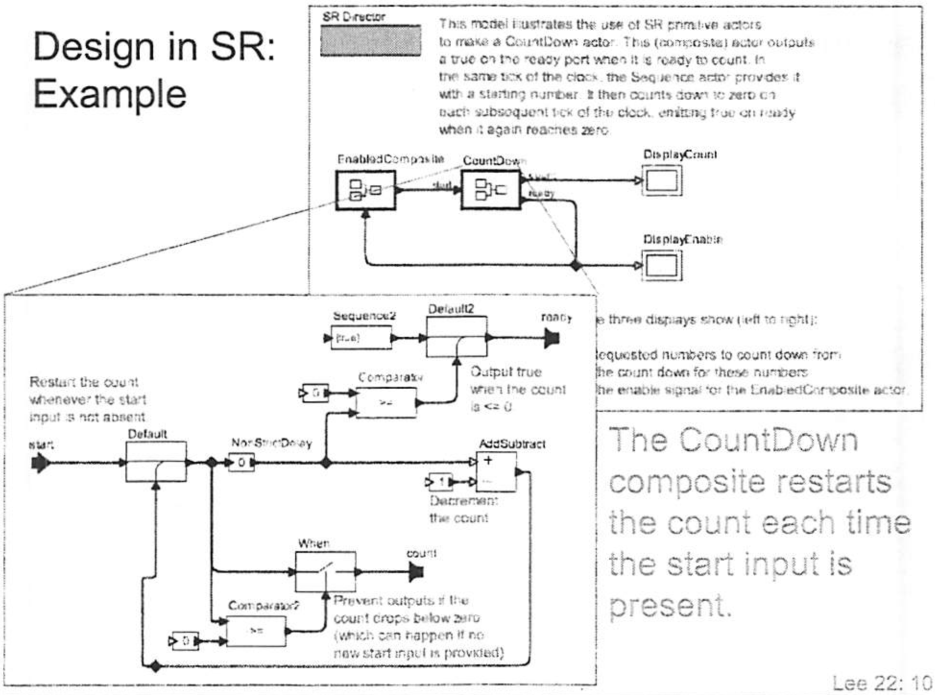
Within this composite, a tick of the clock only occurs when a true value is provided on the enable input port in the enclosing model. Thus, this subsystem has a clock that is a subclock of that of the enclosing model.

Note that because of the subclock in this composite, this NonStrictDelay behaves like Pre. If it were put at the top level, it would not.

Note that this display fires only when the enabled port receives a true token. This is because only then is there a tick of the clock.

If the NonStrictDelay had been put at the top level, would its behavior have been the same?

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

e three displays show (left to right):

equested numbers to count down from
he count down for these numbers
he enable signal for the EnabledComposite actor.

Restart the count whenever the start input is not absent

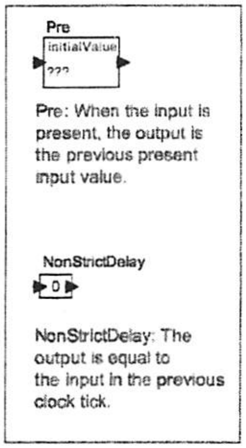
Output true when the count is <= 0

Decrement the count

Prevent outputs if the count drops below zero (which can happen if no new start input is provided)

The CountDown composite restarts the count each time the start input is present.

Subtleties: Pre vs. NonStrictDelay



Pre: When the input is present, the output is the previous present input value.

NonStrictDelay: The output is equal to the input in the previous clock tick.

Pre: True one-sample delay. The behavior is not affected by insertion of an arbitrary number of ticks with "absent" inputs between present inputs.

NonStrictDelay: One-tick delay (vs. one-sample). The output in each tick equals the input in the previous tick (whether absent or not).

Illustration of this Subtlety

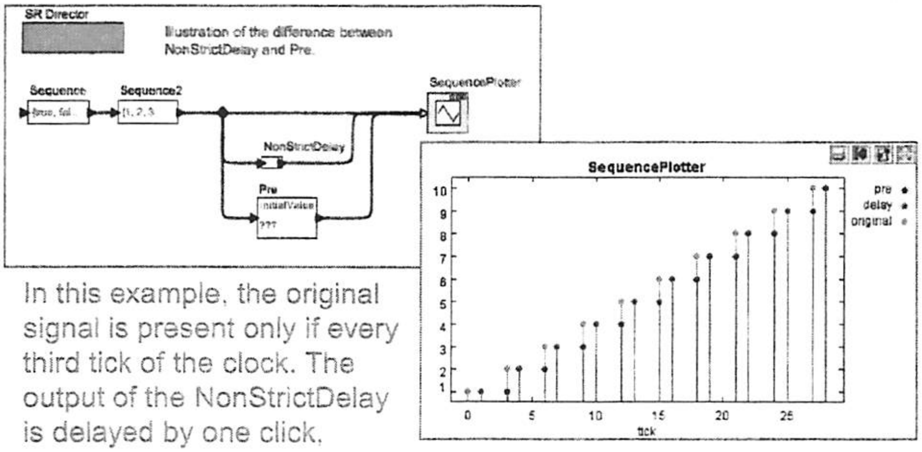
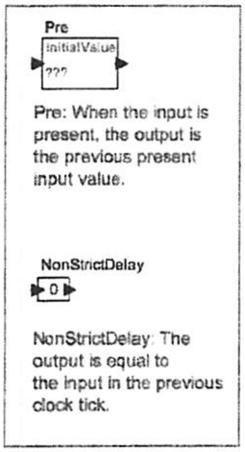


Illustration of the difference between NonStrictDelay and Pre.

In this example, the original signal is present only if every third tick of the clock. The output of the NonStrictDelay is delayed by one click, whereas the output the Pre actor is delayed by one (present) sample.

Consequences: Pre vs. NonStrictDelay



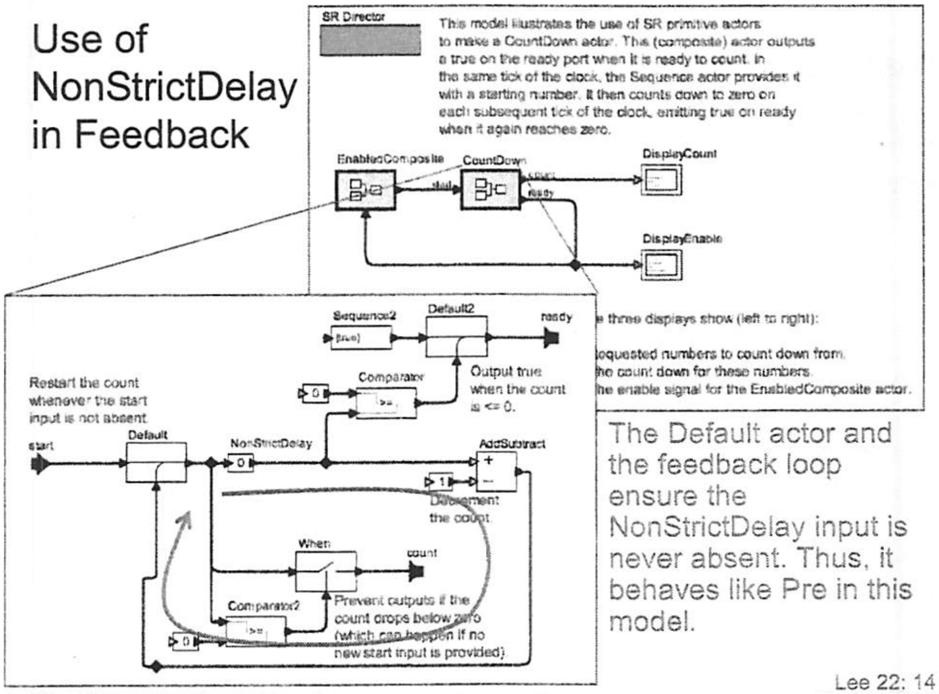
Pre: When the input is present, the output is the previous present input value.

NonStrictDelay: The output is equal to the input in the previous clock tick.

Pre: This actor is *strict*. It must know whether the input is present before it can determine the output. Hence, it cannot be used to break feedback loops.

NonStrictDelay: This actor is *nonstrict*. It need not know whether the input is present nor what its value is before it can determine the output. Hence, it can be used to break feedback loops.

Use of NonStrictDelay in Feedback



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

e three displays show (left to right):

equested numbers to count down from.
he count down for these numbers.
he enable signal for the EnabledComposite actor.

Restart the count whenever the start input is not absent.

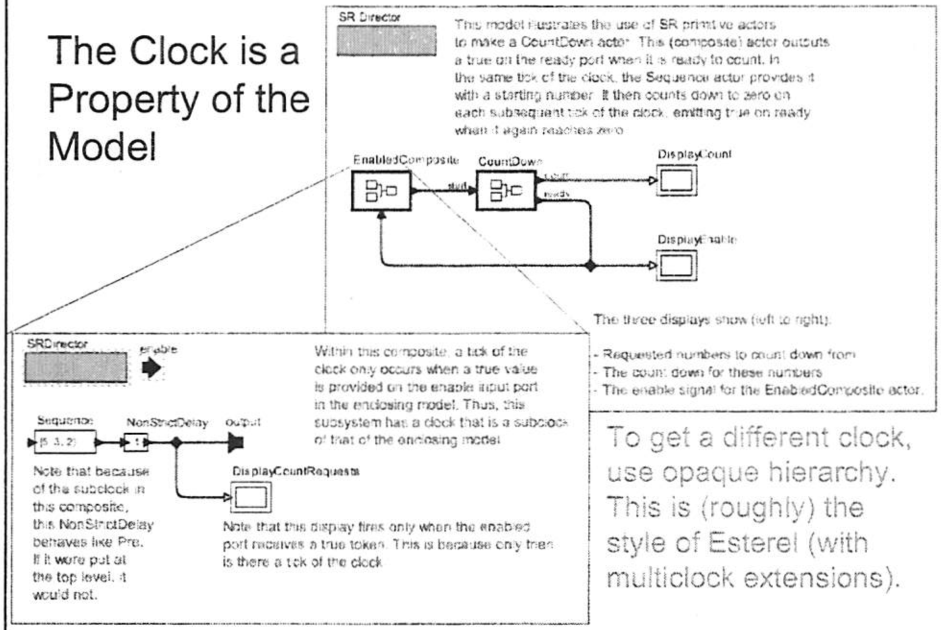
Output true when the count is <= 0.

Decrement the count.

Prevent outputs if the count drops below zero (which can happen if no new start input is provided).

The Default actor and the feedback loop ensure the NonStrictDelay input is never absent. Thus, it behaves like Pre in this model.

The Clock is a Property of the Model



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

The three displays show (left to right):

- Requested numbers to count down from
- The count down for these numbers
- The enable signal for the EnabledComposite actor.

Within this composite, a tick of the clock only occurs when a true value is provided on the enable input port in the enclosing model. Thus, this subsystem has a clock that is a subclock of that of the enclosing model.

Note that because of the subclock in this composite, this NonStrictDelay behaves like Pre. If it were put at the top level, it would not.

Note that this display fires only when the enabled port receives a true token. This is because only then is there a tick of the clock.

To get a different clock, use opaque hierarchy. This is (roughly) the style of Esterel (with multiclock extensions).

Hierarchical Clock Domains

Opaque hierarchy can do:

- Conditioning an internal tick on an external signal
  - Like a conditional
  - If the internal component is an instance of the external, then this amounts to recursion
- Multiple internal ticks per external tick
  - Like a do-while
- Iterated internal ticks over a data structure (use IterateOverArray higher-order actor)
  - Like a for

Alternative Semantics:
The Clock is a Property of the Signal

In Lustre and Signal, a clock is a property of a signal, and Pre and NonStrictDelay could (in theory) behave identically. They would only "tick" when the clock of the input signal ticked.

However, this model has problems with decidability. Clocks cannot always be inferred.

Clock Calculus

- Let T be a well founded totally ordered set of tags.
- Let s: $T \rightarrow V \cup \{\varepsilon\}$ be a signal of type V, where ε means "absent."
- Let c: $T \rightarrow \{-1, 0, 1\}$ be a *clock* associated with s where

$$s(t) = \varepsilon \Rightarrow c(t) = 0$$
$$s(t) = \mathit{true} \Rightarrow c(t) = 1$$
$$s(t) = \mathit{false} \Rightarrow c(t) = -1$$

If V is not boolean, then when $s(t)$ is present, $c(t)$ has value or 1 or -1 (we will make no distinction).

Operations on Clocks

Arithmetic on clocks is in GF-3 (a Galois field with 3 elements), as follows:

$$0 + x = x \qquad 0 \cdot x = 0$$

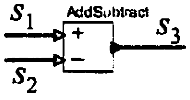
$$1 + 1 = -1 \qquad 1 \cdot x = x$$

$$-1 + -1 = 1 \qquad -1 \cdot x = -x$$

$$-1 + 1 = 0$$

Clock Relations: Simple Synchrony

Most actors require that the clocks on all signals be the same. For example:

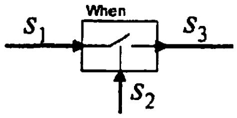


$$\forall t \in T, \quad c_1^2(t) = c_2^2(t) = c_3^2(t)$$

This means that either all are present, or all are absent.

Clock Relations: When Operator

Assuming that s_1 is a boolean-valued signal (which it must be), the clocks on signals interacting through the when operator are related as follows:



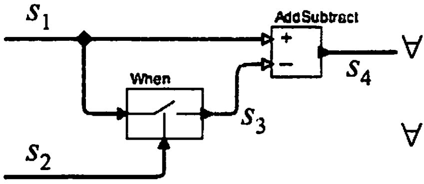
$$\forall t \in T, \quad c_3(t) = c_1(t)(-c_2(t) - c_2^2(t))$$

This means:

- If s_1 is absent, then s_3 is absent.
- If s_2 is false, then s_3 is absent.
- If s_2 is true, then s_3 is the same as s_1.

Consistency Checking

Consider the following model:



$$\forall t \in T, \quad c_1^2(t) = c_4^2(t) = c_3^2(t)$$

$$\forall t \in T, \quad c_3(t) = c_1(t)(-c_2(t) - c_2^2(t))$$

These two together imply that:

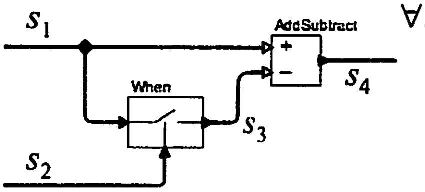
$$\forall t \in T, \quad c_2^2(t)(1 + c_1^2(t)) = -c_2(t)c_1^2(t)$$

where we have used the fact that:

$$(-c_2(t) - c_2^2(t))^2 = (-c_2(t) - c_2^2(t))$$

Interpretation of Consistency Result

Consistency check implies that:



$$\forall t \in T, \quad c_2^2(t)(1+c_1^2(t)) = -c_2(t)c_1^2(t)$$

This means:

s_1 is absent if and only if s_2 is absent.

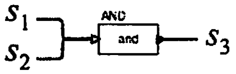
if s_2 is present, then s_2 is true.

Logic Operators Affect Clocks

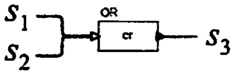
The output of the When actor has a clock that depends on the Boolean control signal. Clocks of Boolean-valued signals reflect the signal value as follows:

$$\forall t \in T,$$

$$c_2(t) = -c_1(t)$$



$$c_3(t) = (c_1(t)c_2(t))^2(-(c_1(t)+1)(c_2(t)+1)-1)$$

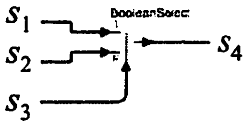


$$c_3(t) = (c_1(t)c_2(t))^2((c_1(t)-1)(c_2(t)-1)+1)$$

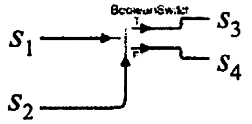
Token Routing Also Affects Clocks

Switch and Select affect the clocks as follows:

$$\forall t \in T,$$

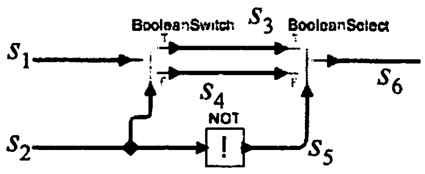


$$c_4(t) = c_3(t)(c_2(t)(1-c_3(t)) - c_1(t)(1+c_3(t)))$$
$$-(c_3(t)+1)c_3(t) = c_1^2(t)$$
$$-(c_3(t)-1)c_3(t) = c_2^2(t)$$



$$c_3(t) = -c_2(t)(c_2(t)+1)c_1(t)$$
$$c_4(t) = c_2(t)(1-c_2(t))c_1(t)$$
$$c_2^2(t) = c_1^2(t)$$

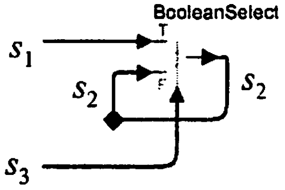
Example 1 Using Switch and Select



What can you infer about the clock of s_6?

$$c_6(t) = 0$$

Example 2 Using Switch and Select



What can you infer about the clocks?

$$c_1(t) = 0 \text{ and}$$
$$\text{either } \; c_3(t) = 0 \; \text{ or } \; 1 + c_3(t) = 0$$

This means that s_1 is absent and s_3 is either absent or false.

What About Delays?

Pre: When the input is present, the output is the previous present input value.

NonStrictDelay: The output is equal to the input in the previous clock tick.

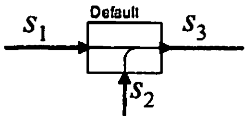
Clock relations across the delays become dependent on the tags. E.g., if *T* is the natural numbers, then we get a nonlinear dynamical system:

$$c_1^2(t) = c_2^2(t) \text{ and}$$
$$c(0) = \text{initial state}$$
$$c(t+1) = (1 - c_1^2(t))c(t) + c_1(t)$$
$$c_2(t) = c_1^2(t)c(t)$$

This makes clock analysis very difficult, in general.

Default Operator

Default: The output equals the left input, if it is present, and the bottom input otherwise:



$$\forall t \in T, \quad c_3(t) = c_1(t) + c_2(t)(1 - c_1^2(t))$$

This means the clock of s_3 is equal to the clock of s_1, if it is present, and to the clock of s_2 otherwise.

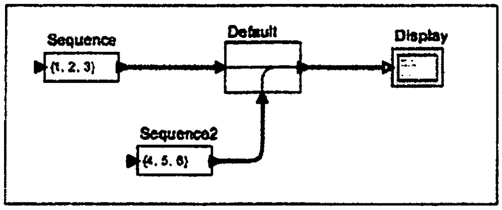
SIGNAL Clock System

In the SIGNAL language, the clock system is richer:

- Let T be a partially ordered set of tags.
- A signal s: $T \rightarrow V \cup \{\varepsilon\}$ of type V is a *partial function* defined on a totally ordered subset of T, where again ε means "absent."

Default Operator in SIGNAL is Nondeterministic

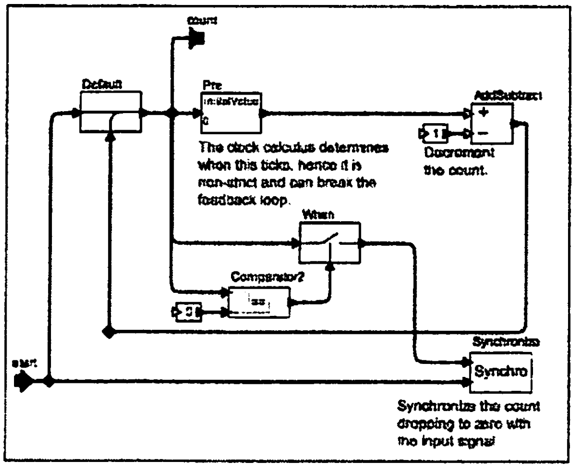
In SIGNAL semantics, the following model has many behaviors:



The two generated sequences have independent clocks (defined over incomparable values of $t \in T$), and the output sequence is any interleaving that preserves the ordering.

Guarded Count in SIGNAL

Instead of generating a "ready" signal, in SIGNAL, the count hitting zero can be synchronized with the input being present.



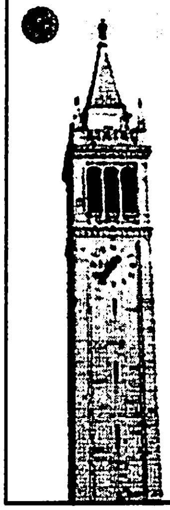
The clock calculus determines when this ticks, hence it is non-strict and can break the feedback loop.

Decrement the count.

Synchronize the count dropping to zero with the input signal

Conclusion and Open Issues

- When clocks are a property of the model, the result is structured synchronous models, where differences between clocks are explicit and no consistency checks are necessary.
- When clocks are a property of a signal, the result is similar to Boolean Dataflow (BDF). It is arguable that clock operators like "when," "default," "switch," and "select" become analogous to unstructured gotos. Clock consistency checking becomes undecidable.
- When further extended as in SIGNAL to partially ordered clock ticks, models easily become nondeterministic.



Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

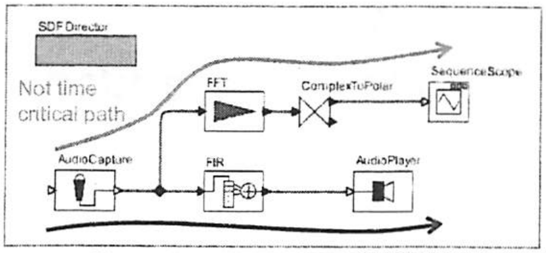
Lecture 23: Time Triggered Models

The Synchronous Abstraction Has a Serious Drawback

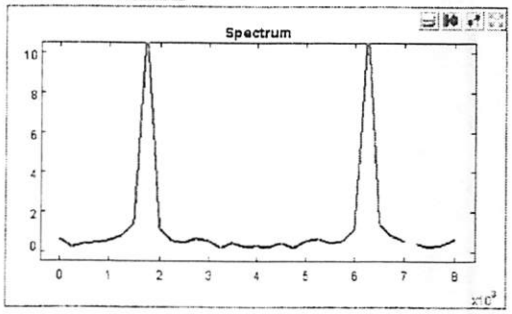
- "Model time" is discrete: Countable ticks of a clock.
- **WRT model time, computation does not take time.**
- **All actors execute "simultaneously" and "instantaneously" (WRT to model time).**

  As a consequence, long-running tasks determine the maximum clock rate of the *fastest* clock, irrespective of how frequently those tasks must run.

Simple Example: Spectrum Analysis

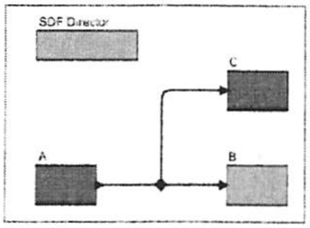


Time critical path

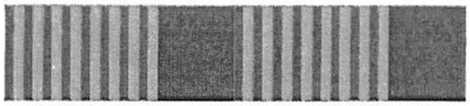


How do we keep the non-time critical path from interfering with the time-critical path?

Abstracted Version of the Spectrum Example

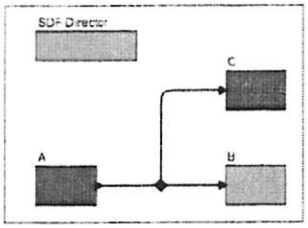


Suppose that C requires 8 data values from A to execute. Suppose further that C takes much longer to execute than A or B. Then a schedule might look like this:

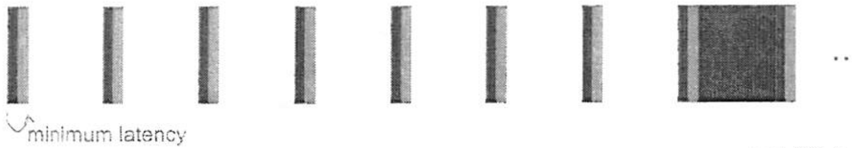


...

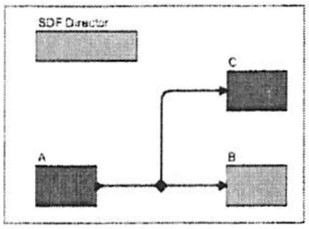
Uniformly Timed Schedule



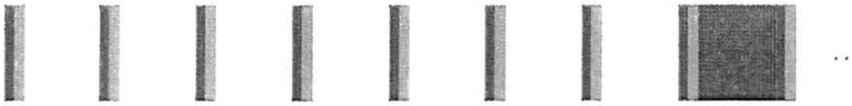
A preferable schedule would space invocations of A and B uniformly in time, as in:



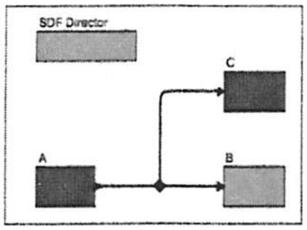
Non-Concurrent Uniformly Timed Schedule



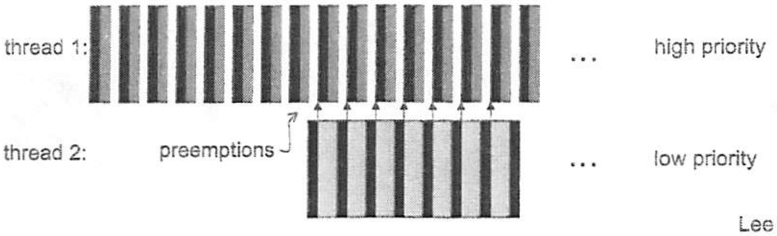
Notice that in this schedule, the rate at which A and B can be invoked is limited by the execution time of C.



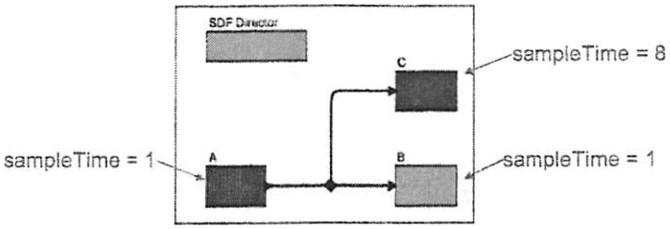
Concurrent Uniformly Timed Schedule



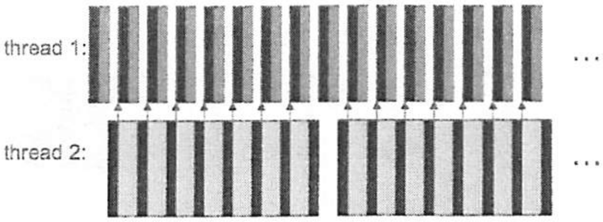
With preemptive multitasking, the rate at which A and B can be invoked is limited only by total computation:



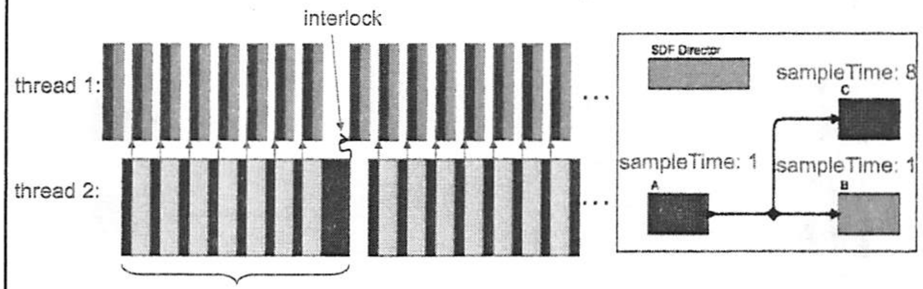
Ignoring Initial Transients, Abstract to Periodic Tasks



In steady-state, the execution follows a simple periodic pattern:



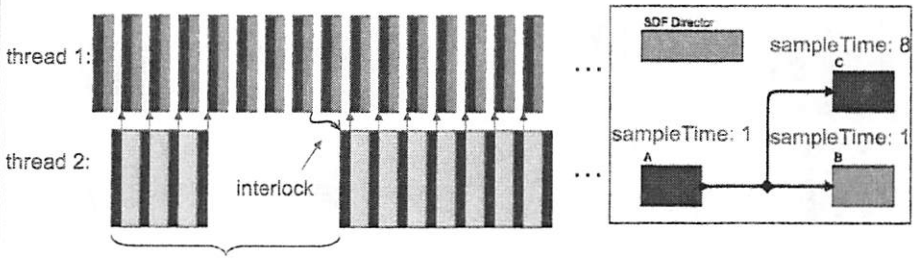
Requirement 1 for Determinacy: Periodicity



If the execution of C runs longer than expected, data determinacy requires that thread 1 be delayed accordingly. This can be accomplished with semaphore synchronization. But there are alternatives:

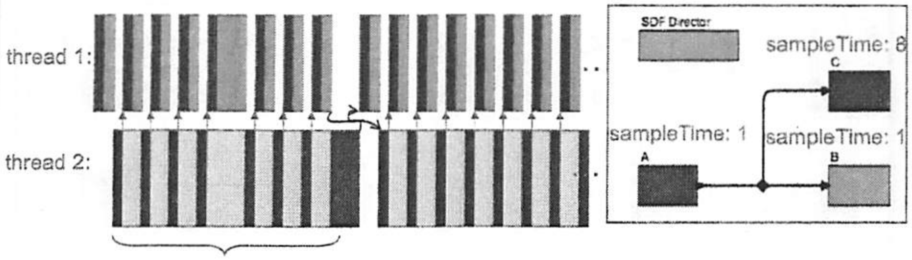
- Throw an exception to indicate timing failure.
- "Anytime" computation: use incomplete results of C

Requirement 1 for Determinacy: Periodicity



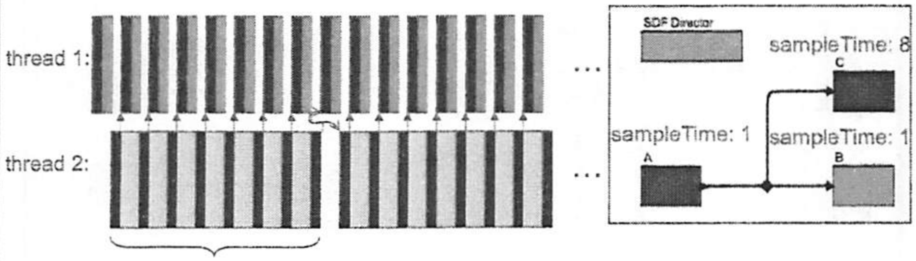
If the execution of C runs shorter than expected, data determinacy requires that thread 2 be delayed accordingly. That is, it must not start the next execution of C before the data is available.

Semaphore Synchronization Required Exactly Twice Per Major Period



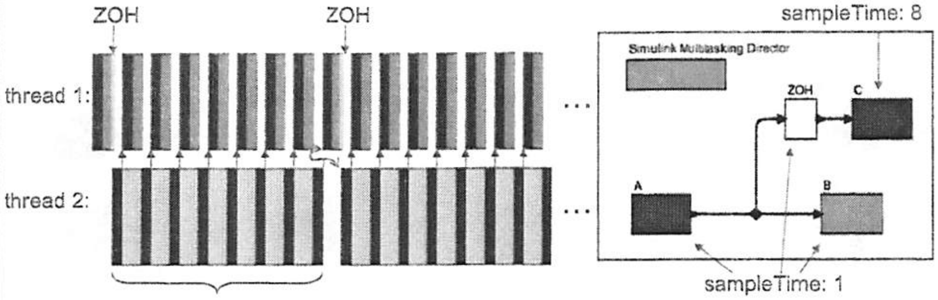
Note that semaphore synchronization is *not* required if actor B runs long because its thread has higher priority. Everything else is automatically delayed.

Requirement 2 for Determinacy: Data Integrity



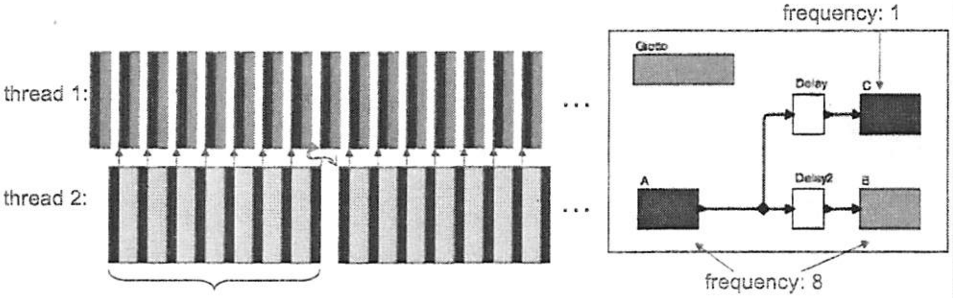
During one execution of C, it is essential that any data it reads from its inputs not depend on any executions of A that are concurrent with that execution of C. This is because execution times are estimates, so when the preemption occurs within the code of C is best modeled as random.

Simulink Strategy for Preserving Determinacy



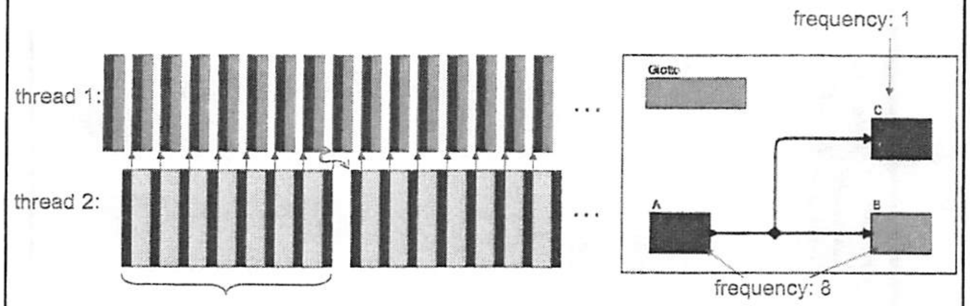
In "Multitasking Mode," Simulink requires a Zero-Order Hold (ZOH) block at any downsampling point. The ZOH runs at the slow rate, but at the priority of the fast rate. The ZOH holds the input to C constant for an entire execution.

Giotto Strategy for Preserving Determinacy



First execution of C operates on initial data in the delay. Second execution operates on the result of the 8-th execution of A.

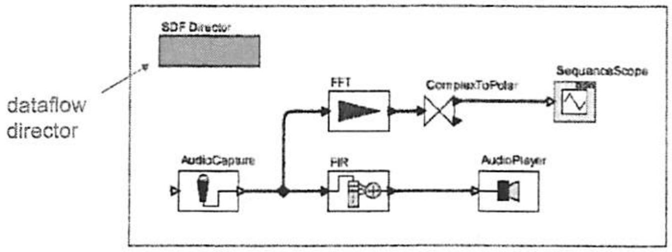
Giotto: A Delay on Every Arc



Since Giotto has a delay on every connection, there is no need to show it. It is implicit.

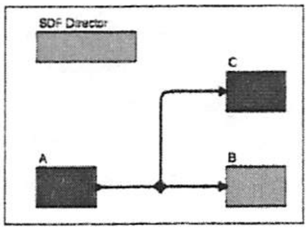
Is a delay on every arc a good idea?

Note that Neither the Simulink nor the Giotto Strategy Works for Our Example



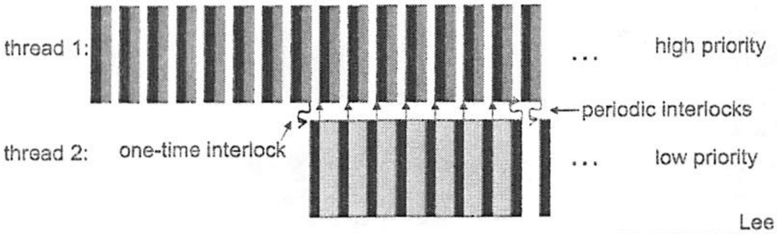
The data from the AudioCapture actor is buffered in a FIFO queue for the FFT actor. There is no danger of data being overwritten by the AudioCapture actor. The Simulink strategy would present only the first of each 8 samples from the AudioCapture block to the FFT block.

One-Time Interlock for Dataflow



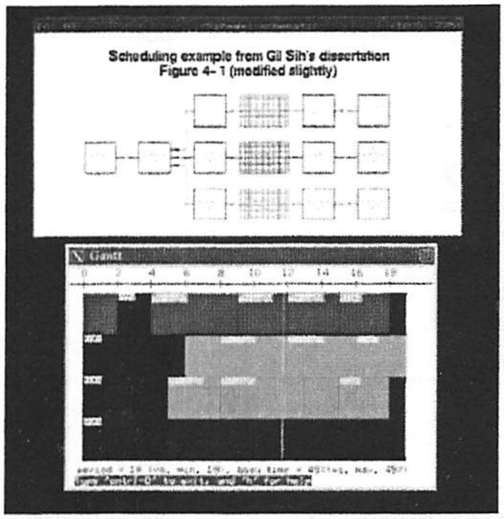
No ZOH block is required!

For dataflow, a one-time interlock ensures sufficient data at the input of C:

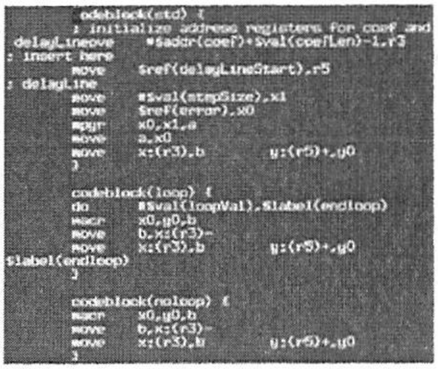


Aside: Ptolemy Classic Code Generator Used Such Interlocks (since about 1990)

SSDF model, parallel schedule, and synthesized DSP assembly code



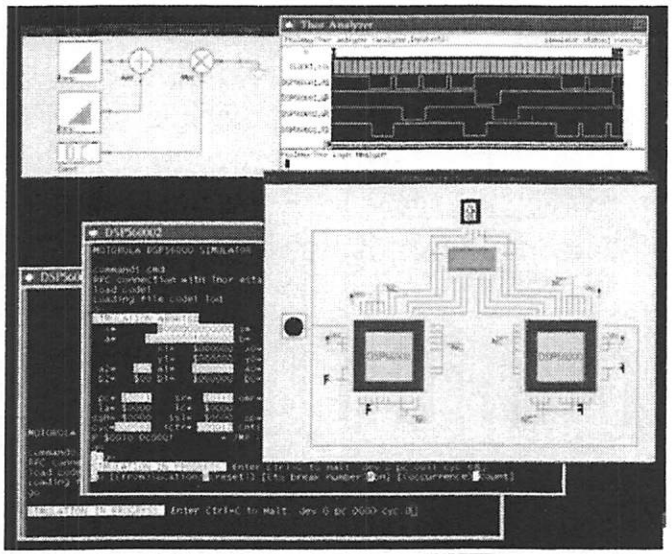
Scheduling example from Gil Sih's dissertation
Figure 4-1 (modified slightly)



```
	odeblock(std) {
	; initialize address registers for coef and
delayLineove	#$addr(coef)+$val(coefLen)-1,r3
; insert here
	move	$ref(delayLineStart),r5
; delayLine
	move	#$val(stepSize),x1
	move	$ref(error),x0
	mpyr	x0,x1,a
	move	a,x0
	move	x:(r3),b		y:(r5)+,y0
	}

	codeblock(loop) {
	do	#$val(loopVal),$label(endloop)
	macr	x0,y0,b
	move	b,x:(r3)-
	move	x:(r3),b		y:(r5)+,y0
$label(endloop)
	}

	codeblock(noloop) {
	macr	x0,y0,b
	move	b,x:(r3)-
	move	x:(r3),b		y:(r5)+,y0
	}
```

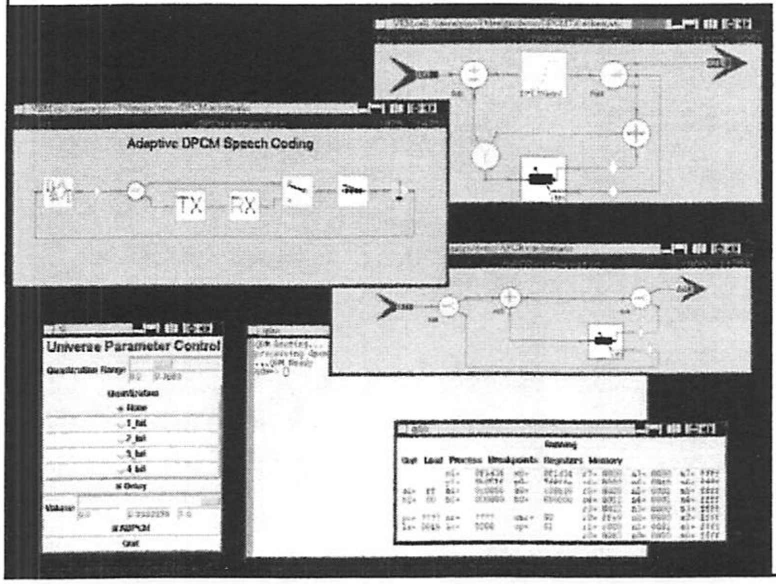
It is an interesting (and rich) research problem to minimize interlocks in complex multirate applications.

Aside: Ptolemy Classic Development Platform (1990)



An SSDF model, a "Thor" model of a 2-DSP architecture, a "logic analyzer" trace of the execution of the architecture, and two DSP code debugger windows, one for each processor.

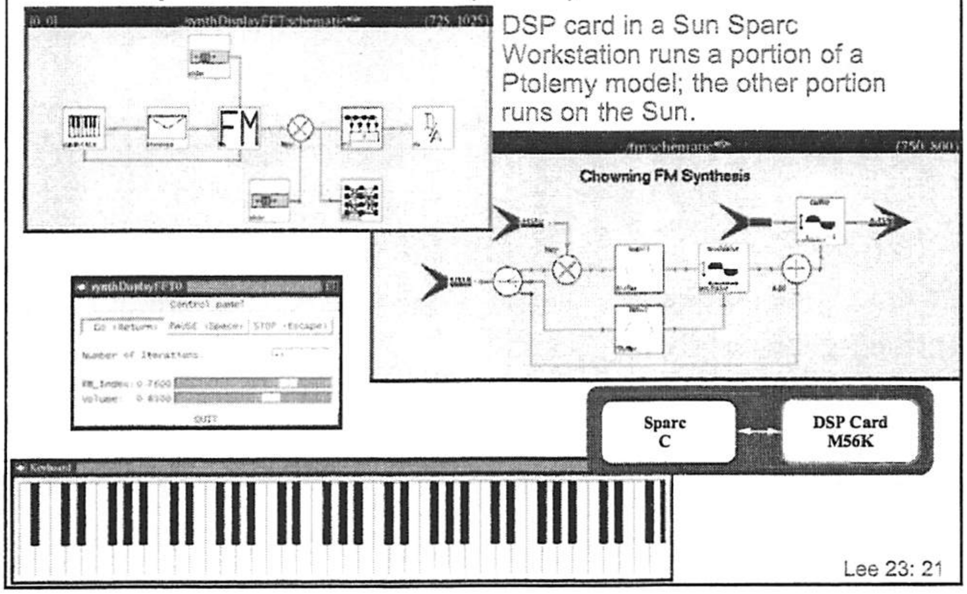
Aside: Application to ADPCM Speech Coding (1993)



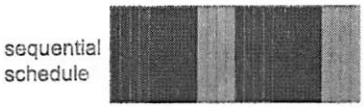
Note updated DSP debugger interface with host/DSP interaction.

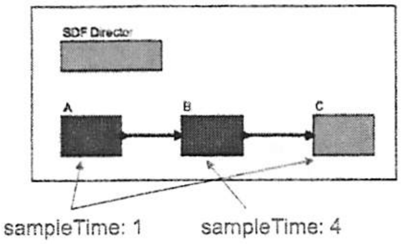
Aside: Heterogeneous Architecture with DSP and Sun Sparc Workstation (1995)

DSP card in a Sun Sparc Workstation runs a portion of a Ptolemy model; the other portion runs on the Sun.



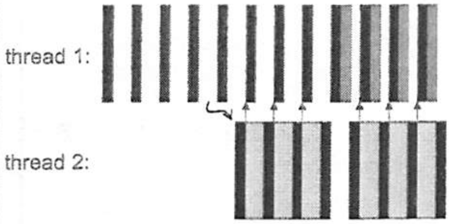
Consider a Low-Rate Actor Sending Data to a High-Rate Actor

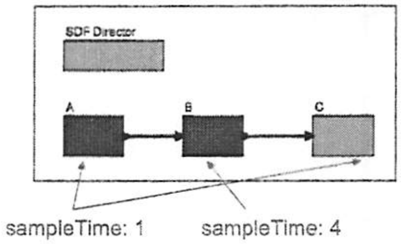




Note that data precedences make it impossible to achieve uniform timing for A and C with the periodic non-concurrent schedule indicated above.

Overlapped Iterations Can Solve This Problem

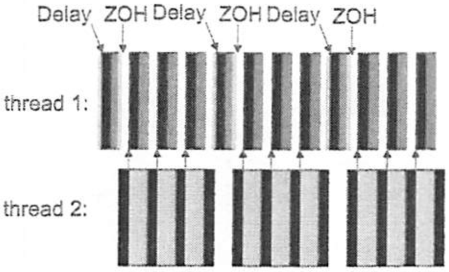


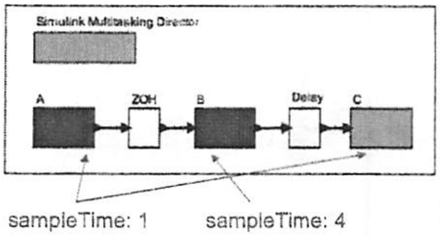


This solution takes advantage of the intrinsic buffering provided by dataflow models.

For dataflow, this requires the initial interlock as before, and the same periodic interlocks.

Simulink Strategy

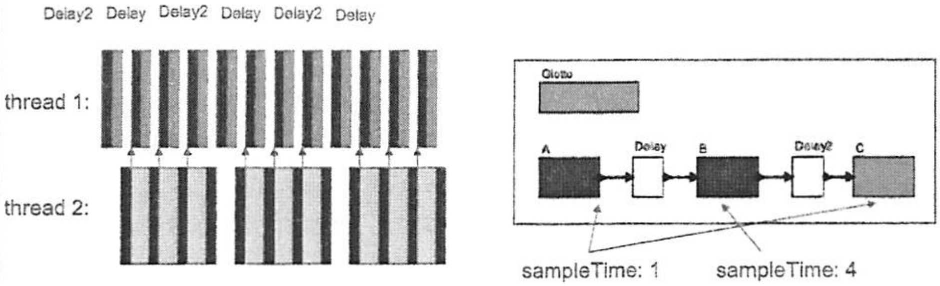




The Delay provides just one initial sample to C (there is no buffering in Simulink). The Delay and ZOH run at the rates of the slow actor, but at the priority of the fast ones.

Part of the objective seems to be to have no initial transient. Why?

Giotto Strategy



Giotto uses delays on all connections. The effect is the same, except that there is one additional sample delay from input to output.

Discussion Questions

- What about more complicated rate conversions (e.g. a task with sampleTime 2 feeding one with sampleTime 3)?
- What are the advantages and disadvantages of the Giotto delays?
- Could concurrent execution be similarly achieved with synchronous languages?
- How does concurrent execution of dataflow compare to Giotto and Simulink?
- Which of these approaches is more attractive from the application designer's perspective?
- How can these ideas be extended to non-periodic execution? (modal models, Timed Multitasking, xGiotto)

Conclusions and Open Questions

- Giotto, Simulink, and TM, all achieve data determinism with snapshot of inputs and delayed commit of outputs.
- Giotto introduces a unit delay in any communication. Simulink introduces a unit delay only on downwards sample rate changes.
- By exploiting uses of Pre in synchronous languages, concurrent execution can be similarly achieved.
- Dataflow does not introduce a unit delay.
- Considerable confusion remains.

Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

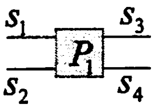
Lecture 24: The Tagged Signal Model

Tags, Values, Events, and Signals

- A set of *values* V and a set of *tags* T
- An *event* is $e \in T \times V$
- A *signal* s is a set of events. I.e. $s \subset T \times V$
- The set of all signals $S = P(T \times V)$
- A *functional signal* is a (partial) function $s: T \rightarrow V$
- A tuple of signals $\mathbf{s} \in S^n$
- The empty signal $\lambda = \emptyset \in S$
- The empty tuple of signals $\Lambda \in S^n$

Processes

A *process* is a subset of signals $P \subset S^n$



$$P_1 \subset S^4$$

The *sort* of a process is the identity of its signals. That is, two processes P_1 and P_2 are of the same sort if

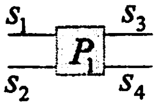
$$\forall i \in \{1,\ldots,n\}, \quad \pi_i(P_1) = \pi_i(P_2)$$

projection

Alternative Notation

Instead of tuples of signals, let X be a set of variables. E.g.

$$X = \{s_1, s_2, s_3, s_4\}$$

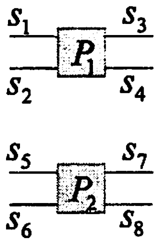


$$P_1 \subset [X \to S] = S^X$$

This is a better notation because it is explicit about the *sort*. This notation was introduced by [Benveniste, et al., 2003]. We will nonetheless stick to the original notation in [Lee, Sangiovanni 1998].

Process Composition

To compose processes, they may need to be augmented to be of the same sort:

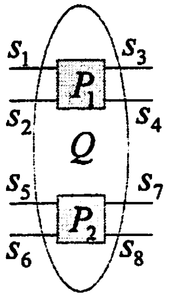


$P_1 \subset S^4$ $\quad$ $P_1' = P_1 \times S^4 \subset S^8$

$P_2 \subset S^4$ $\quad$ $P_2' = S^4 \times P_2 \subset S^8$

Process Composition

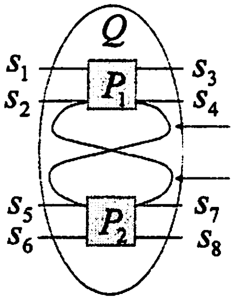
To compose processes, they may need to be augmented to be of the same sort:



$$\begin{tikzcd}
P_1' = P_1 \times S^4 \subset S^8 \arrow[dr] & \\
& Q = P_1' \cap P_2' = P_1 \times P_2 \\
P_2' = S^4 \times P_2 \subset S^8 \arrow[ur] &
\end{tikzcd}$$

Connections

Connections simply establish that signals are identical:



$$C_{4,5} = \{s \in S^8 \mid \pi_4(s) = \pi_5(s)\}$$

$$C_{2,7} = \{s \in S^8 \mid \pi_2(s) = \pi_7(s)\}$$

$$Q = P_1' \cap P_2' \cap C_{4,5} \cap C_{2,7}$$

Projections (Hiding and Renaming)

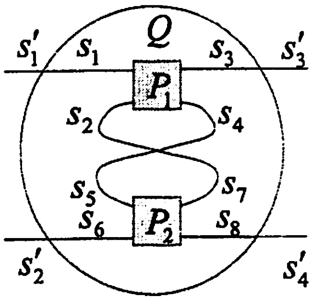
Given an m-tuple of indexes: $I \in \{1,\ldots,n\}^m$

the following projection accomplishes
hiding and/or renaming:

$$\pi_I(P) = (\pi_{\pi_1(I)}(P),\ldots,\pi_{\pi_m(I)}(P))$$

Example of Projections (Hiding)

Projections change the sort of a process:



$$I=(1,3,6,8)$$

$$Q=\pi_I(P_1'\cap P_2'\cap C_{4,5}\cap C_{2,7})\subset S^4$$

Inputs

Given a process $P\subset S^n$, an *input* is a subset of the same sort, $A\subset S^n$, that constrains the behaviors of the process to

$$P'=P\cap A$$

An input could be a single event in a signal, an entire signal, or any combination of events and signals. A particular process may “accept” only certain inputs, in which case the process is defined by $P\subset S^n$ and $B\subset P(S^n)$, where any input A is required to be in B,

$$A\in B$$

Closed System (no Inputs)

A process $P \subset S^n$ with input set $B \subset P(S^n)$ is *closed* if

$$B = \{S^n\}$$

This means that the only possible input (constraint) is:

$$A = S^n$$

which imposes no constraints at all in

$$P' = P \cap A$$

Functional Processes

Model for a process $P \subset S^n$ that has m input signals and p output signals (exercise: what is the input set B?)

- Define two index sets for the input and output signals:

$$I \in \{1,\ldots,n\}^m, \quad O \in \{1,\ldots,n\}^p$$

- The process is *functional* w.r.t. (I, O) if

$$\forall s, s' \in P, \quad \pi_I(s) = \pi_I(s') \Rightarrow \pi_O(s) = \pi_O(s')$$

- In this case, there is a (possibly partial) function

$$F : S^m \to S^p \quad s.t. \quad \forall s \in P, \quad \pi_O(s) = F(\pi_I(s))$$

Determinacy

A process P with input set B is determinate if for any input $A \in B$,

$$|P \cap A| \in \{0,1\}$$

That is, given an input, there is no more than one behavior.

Note that by this definition, a functional process is assured of being determinate if all its signals are visible on the output.

Refinement Relations

A process (with input constraints) (P', B') is a *refinement* of the process (P, B) if

$$B \subseteq B'$$

and

$$\forall A \in B, \;\; P' \cap A \subseteq P \cap A$$

That is, the refinement accepts any input that the process it refines accepts, and for any input it accepts, its behaviors are a subset of the behaviors of the process it refines with the same input.

Tags for Discrete-Event Systems

For DE, let $T = R \times N$ with a total order (the lexical order) and an ultrametric (the Cantor metric). Recall that we have used the structure of this tag set to get nontrivial results:

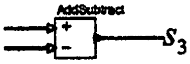
If processes are functional and causal and every feedback path has at least one delta-causal process, then compositions of processes are determinate and we have a procedure for identifying their behavior.

Synchrony

- Two events are synchronous if they have the same tag.
- Two signals are synchronous if all events in one a synchronous with an event in the other.
- A process is synchronous if for in every behavior in the process, every signal is synchronous with every other signal.

Tags for Process Networks

- The tag set T is a poset.
- The tags $T(s)$ on each signal s are totally ordered.
- A *sequential* process has a signal associated with it that imposes ordering constraints on the other signals. For example:



$$s_1 = \{(v_{1,1}, t_{1,1}), (v_{1,2}, t_{1,2}), \ldots\}$$

$$s_2 = \{(v_{2,1}, t_{2,1}), (v_{2,2}, t_{2,2}), \ldots\}$$

$$s_3 = \{(v_{3,1}, t_{3,1}), (v_{3,2}, t_{3,2}), \ldots\}$$

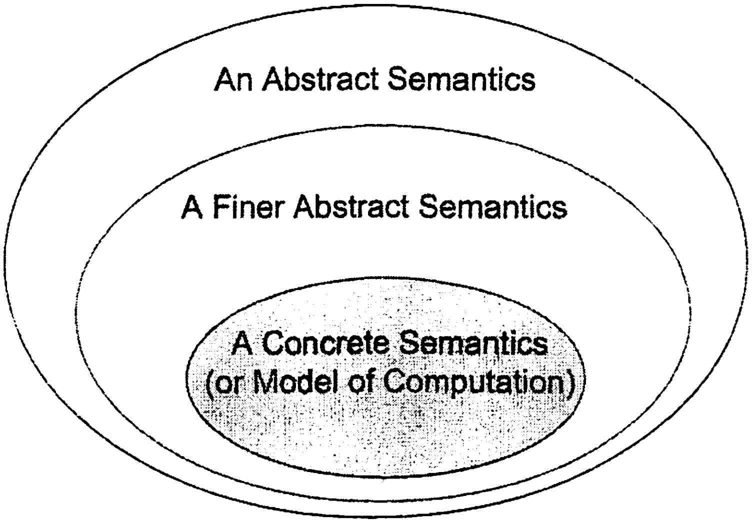
$$s_4 = \{(x, t_{4,1}), (x, t_{4,2}), \ldots\}$$

$$t_{i,j} < t_{i,j+1} \qquad t_{1,j} < t_{4,2j}, \;\; t_{2,j} < t_{4,2j+1}, \;\; t_{3,j} > t_{4,2j+1}$$

Tags Can Model …

- Dataflow firing
- Rendezvous in CSP
- Ordering constraints in Petri nets
- etc. (see paper)

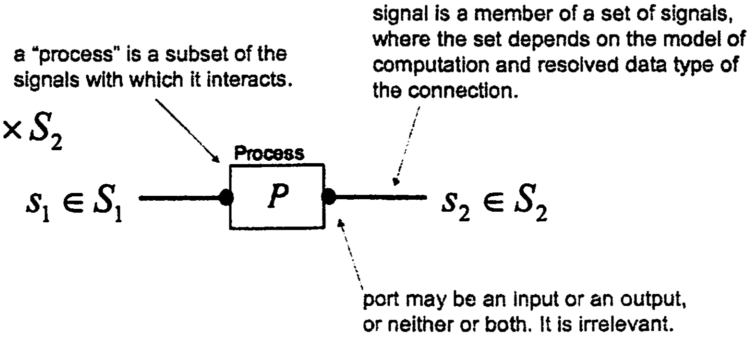
The Tagged Signal Model can be used to Define *Abstract Semantics*



Tagged Signal Abstract Semantics

Tagged Signal Abstract Semantics:

$$P \subset S_1 \times S_2$$



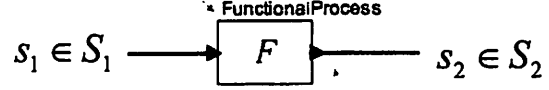
This outlines a general *abstract semantics* that gets specialized. When it becomes concrete you have a *model of computation.*

A Finer Abstraction Semantics

Functional Abstract Semantics:

a process is now a function from input signals to output signals.

$$F : S_1 \rightarrow S_2$$



port is now either an input or an output (or both).

This outlines an *abstract semantics* for deterministic producer/consumer actors.

Uses for Such an Abstract Semantics

Give structure to the sets of signals

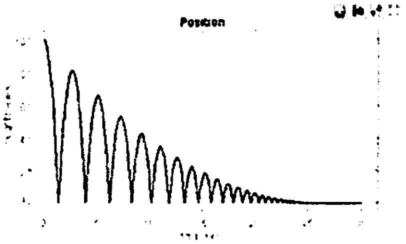
- **e.g. Use the Cantor metric to get a metric space.**

Give structure to the functional processes

- **e.g. Contraction maps on the Cantor metric space.**

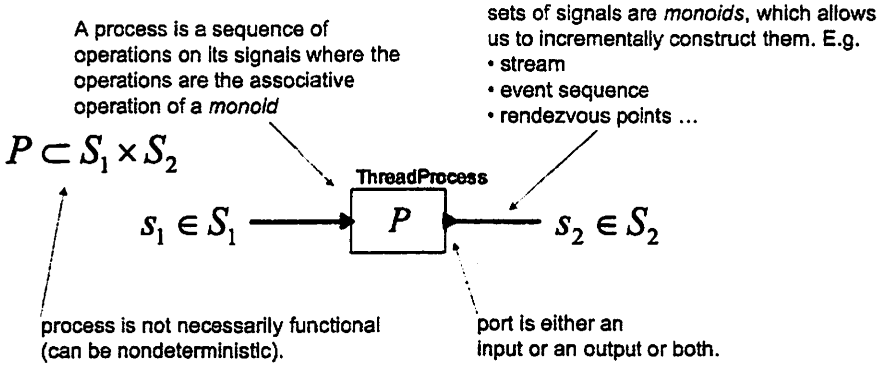
Develop static analysis techniques

- **e.g. Conditions under which a hybrid systems is provably non-Zeno.**



Another Finer Abstract Semantics

Process Networks Abstract Semantics:



A process is a sequence of operations on its signals where the operations are the associative operation of a *monoid*

sets of signals are *monoids*, which allows us to incrementally construct them. E.g.

- stream
- event sequence
- rendezvous points ...

$P \subset S_1 \times S_2$

process is not necessarily functional (can be nondeterministic).

port is either an input or an output or both.

This outlines an abstract semantics for actors constructed as processes that incrementally read and write port data.

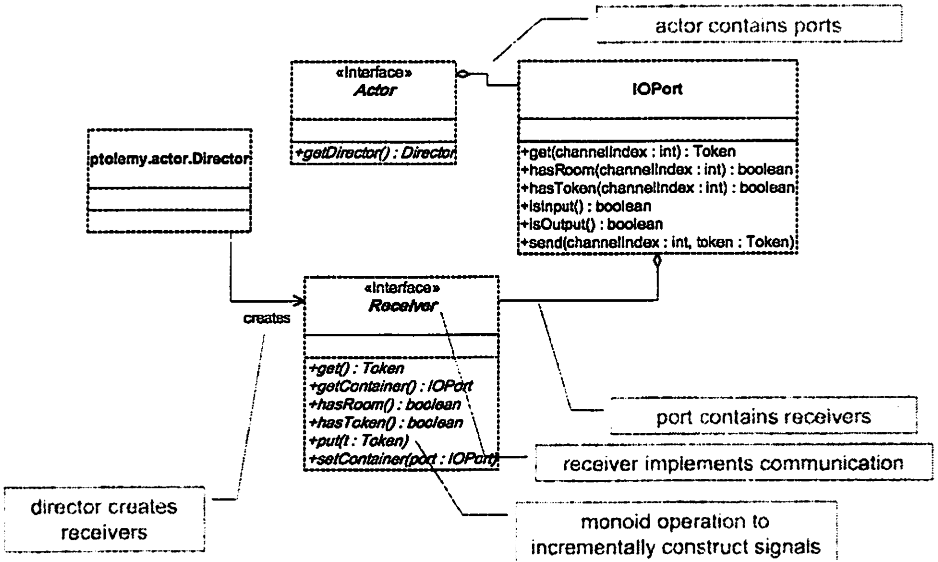
Concrete Semantics that Conform with the Process Networks Abstract Semantics

- Communicating Sequential Processes (CSP) [Hoare]
- Calculus of Concurrent Systems (CCS) [Milner]
- Kahn Process Networks (KPN) [Kahn]
- Nondeterministic extensions of KPN [Various]
- Actors [Hewitt]

Some Implementations:

- Occam, Lucid, and Ada languages
- Ptolemy Classic and Ptolemy II (PN and CSP domains)
- System C
- Metropolis

Process Network Abstract Semantics in Ptolemy II



actor contains ports

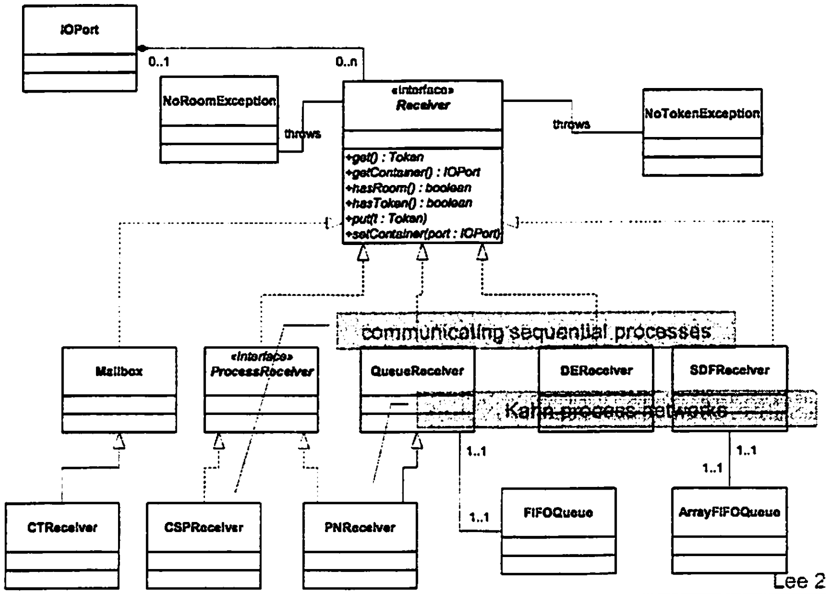
port contains receivers

receiver implements communication

director creates receivers

monoid operation to incrementally construct signals

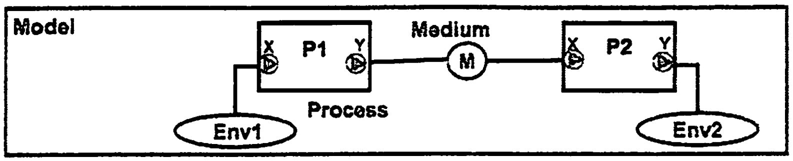
Several Concrete Semantics Refine this Abstract Semantics



communicating sequential processes

Kahn process networks

Process Network Abstract Semantics in Metropolis



```
process P{
  port reader X;
  port writer Y;
  thread(){
   while(true){
    ...
    z = f(X.read());
    Y.write(z);
   }}}
```

```
interface reader extends Port{
   update int read();
   eval int n();
}
```

```
interface writer extends Port{
   update void write(int i);
   eval int space();
}
```

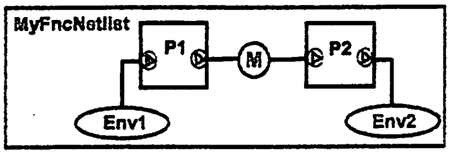
```
medium M implements reader, writer{
  int storage;
  int n, space;
  void write(int z){
     await(space>0; this.writer ; this.writer)
          n=1; space=0; storage=z;
  }
  word read(){ ... }
}
```

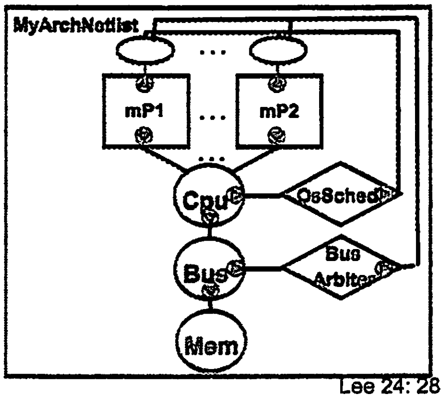
Thanks to
Doug Densmore

Leveraging Abstract Semantics for Joint Modeling of Architecture and Application

MyMapNetlist

B( ) B( ) E( ) E( )
B( ) B( ) E( ) E( )
B( ) B( ) E( ) E( )
B( ) B( ) E( ) E( )



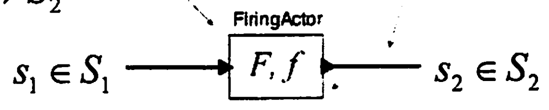


The abstract semantics provides natural points of the execution (where the monoid operations are invoked) that can be synchronized across models. Here, this is used to model operations of an application on a candidate implementation architecture.

A Finer Abstract Semantics

Firing Abstract Semantics:

$$F : S_1 \rightarrow S_2$$



a process still a function from input signals to output signals, but that function now is defined in terms of a firing function.

signals are in monoids (can be incrementally constructed) (e.g. streams, discrete-event signals).

port is still either an input or an output.

The process function F is the least fixed point of a functional defined in terms of f.

Models of Computation that Conform to the Firing Abstract Semantics

- Dataflow models (all variations)
- Discrete-event models
- Time-driven models (Giotto)

In Ptolemy II, actors written to the *firing abstract semantics* can be used with directors that conform only to the process network abstract semantics.

Such actors are said to be *behaviorally polymorphic*.

Actor Language for the Firing Abstract Semantics: Cal

Cal is an experimental actor language designed to provide statically inferable actor properties w.r.t. the firing abstract semantics. E.g.:

```
actor Select () S, A, B ==> Output:

    action S: [sel], A: [v] ==> [v]
    guard sel end

    action S: [sel], B: [v] ==> [v]
    guard not sel end
end
```

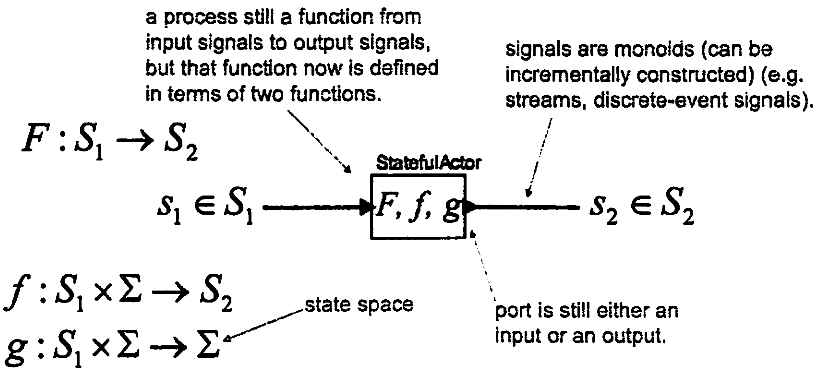
Inferable firing rules and firing functions:

$$U_1 = \{\langle(\text{true}),(v),\bot\rangle : v \in \mathbf{Z}\}, f_1 : \langle(\text{true}),(v),\bot\rangle \mapsto (v)$$
$$U_2 = \{\langle(\text{false}),\bot,(v)\rangle : v \in \mathbf{Z}\}, f_2 : \langle(\text{false}),\bot,(v)\rangle \mapsto (v)$$

Thanks to Jorn Janneck, Xilinx

A Still Finer Abstract Semantics

Stateful Firing Abstract Semantics:



a process still a function from input signals to output signals, but that function now is defined in terms of two functions.

signals are monoids (can be incrementally constructed) (e.g. streams, discrete-event signals).

$F : S_1 \to S_2$

StatefulActor

$s_1 \in S_1 \longrightarrow \boxed{F, f, g} \longrightarrow s_2 \in S_2$

$f : S_1 \times \Sigma \to S_2$

$g : S_1 \times \Sigma \to \Sigma$

state space

port is still either an input or an output.

The function f gives outputs in terms of inputs and the current state. The function g updates the state.

Models of Computation that Conform to the Stateful Firing Abstract Semantics

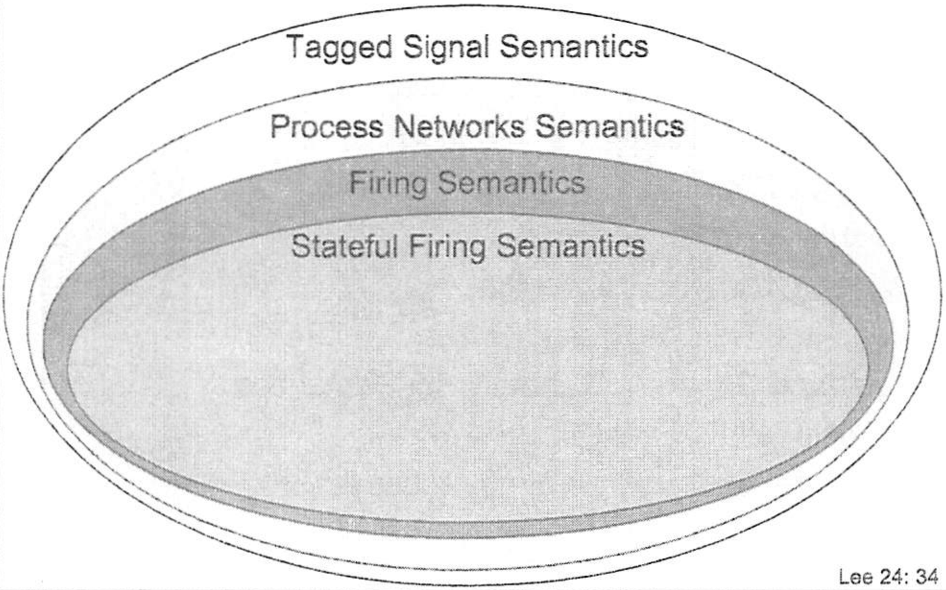
- Synchronous reactive
- Continuous time
- Hybrid systems

Stateful firing supports iteration to a fixed point, which is required for hybrid systems modeling.

In Ptolemy II, actors written to the stateful firing abstract semantics can be used with directors that conform only to the firing abstract semantics or to the process network abstract semantics.

Such actors are said to be *behaviorally polymorphic*.

Where We Are



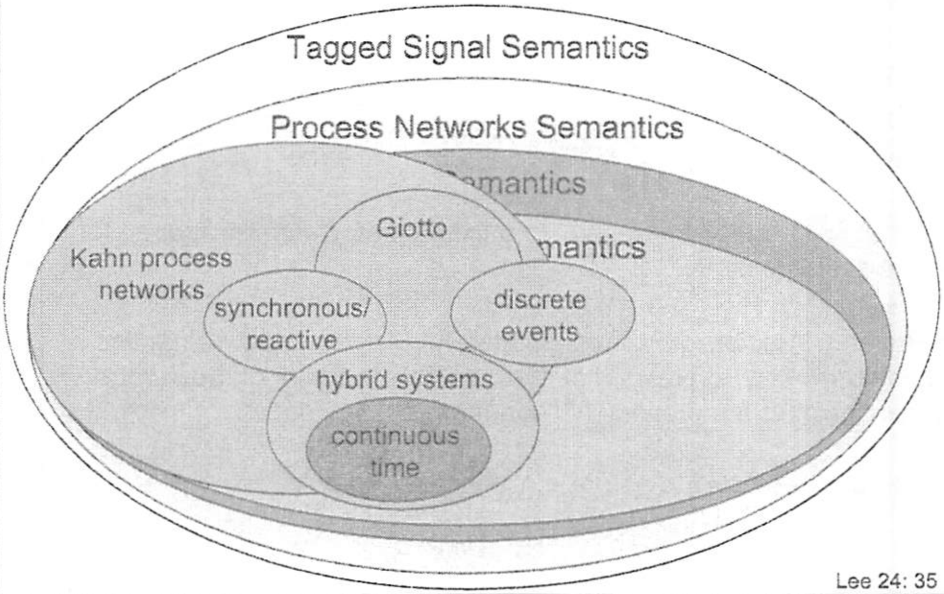
Tagged Signal Semantics

Process Networks Semantics

Firing Semantics

Stateful Firing Semantics

Where We Are



Related Work

- Abramsky, et al., Interaction Categories
- Agha, et al., Actors
- Hoare, CSP
- Mazurkiewicz, et al., Traces
- Milner, CCS and Pi Calculus
- Reed and Roscoe, Metric Space Semantics
- Scott and Strachey, Denotational Semantics
- Winskel, et al., Event Structures
- Yates, Networks of real-time processes

Conclusion and Open Issues

- The *tagged signal model* provides a very general conceptual framework for comparing and reasoning about models of computation,
- The tagged signal model provides a natural model of *design refinement*, which offers the possibility of type-system-like formal structures that deal with dynamic behavior, and not just static structure.
- The idea of *abstract semantics* offers ways to reason about multi-model frameworks like Ptolemy II and Metropolis, and offers clean definitions of behaviorally polymorphic components.

Concurrent Models of Computation for Embedded Software

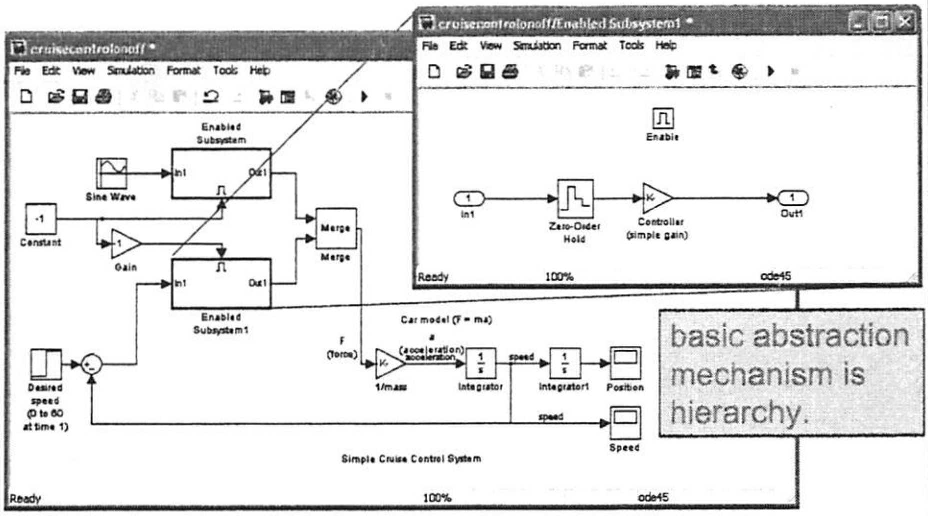
Edward A. Lee
Professor, UC Berkeley
EECS 290n – Advanced Topics in Systems Theory
Fall, 2004

Lecture 25: Actor-Oriented Type Systems

Does Actor-Oriented Design Offer Best-Of-Class SW Engineering Methods?

- Abstraction
  - procedures/methods
  - classes
- Modularity
  - subclasses
  - inheritance
  - interfaces
  - polymorphism
  - aspects
- Correctness
  - type systems

Example of an Actor-Oriented Framework: Simulink



basic abstraction mechanism is hierarchy.

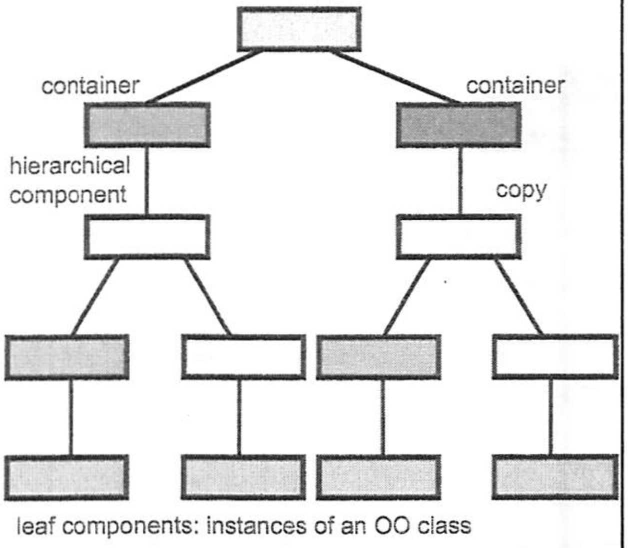
Observation

By itself, hierarchy is a very weak abstraction mechanism.

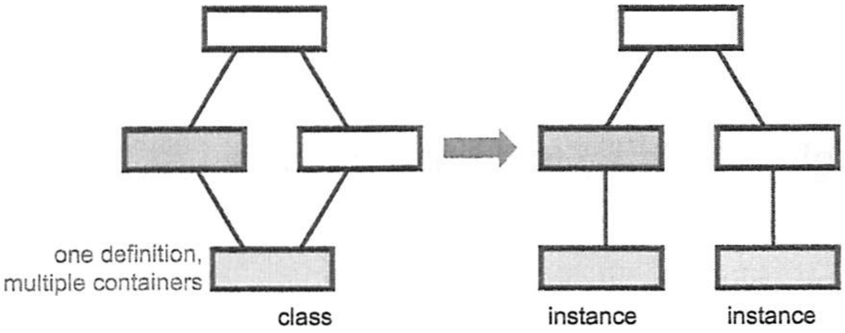
Tree Structured Hierarchy

Does not represent common *class* definitions. Only instances.

Multiple instances of the same hierarchical component are *copies.*



Alternative Hierarchy: Roles and Instances



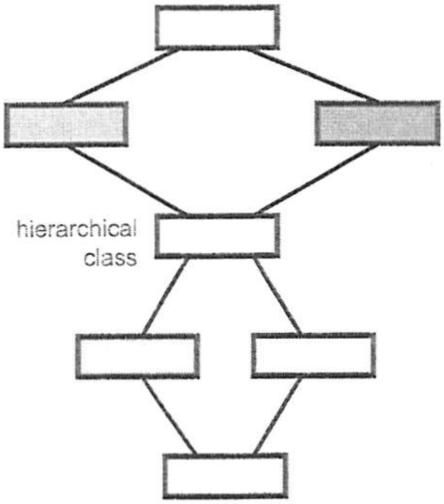
role hierarchy ("design-time" view)

instance hierarchy ("run time" view)

Role Hierarchy

Multiple instances of the same hierarchical component are represented by *classes* with multiple containers.

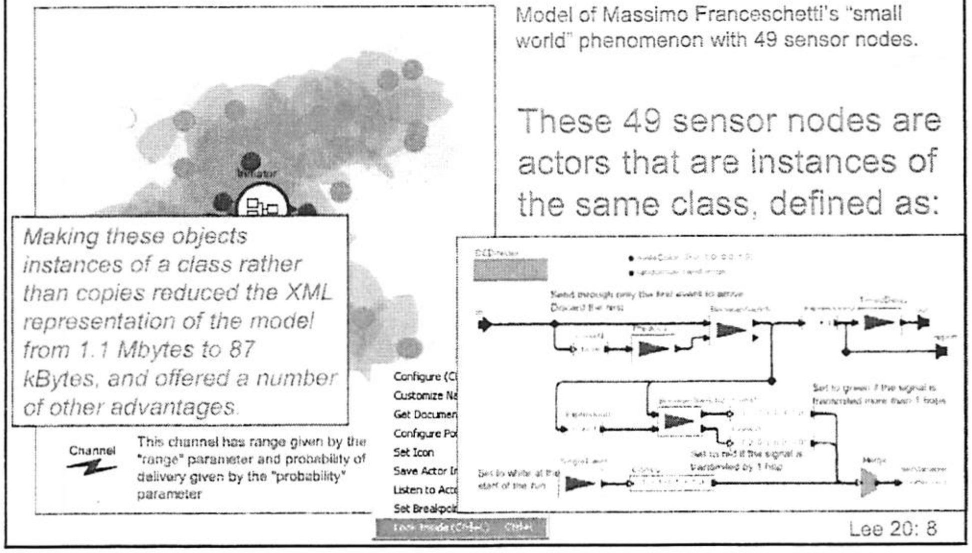
This makes hierarchical components more like leaf components.



A Motivating Application: Modeling Sensor Networks

Model of Massimo Franceschetti's "small world" phenomenon with 49 sensor nodes.

These 49 sensor nodes are actors that are instances of the same class, defined as:



This channel has range given by the "range" parameter and probability of delivery given by the "probability" parameter

Send through only the first event to arrive. Discard the rest.

Set to green if the signal is transmitted more than 1 hops

Set to red if the signal is transmitted by 1 hop

Set to white at the start of the run

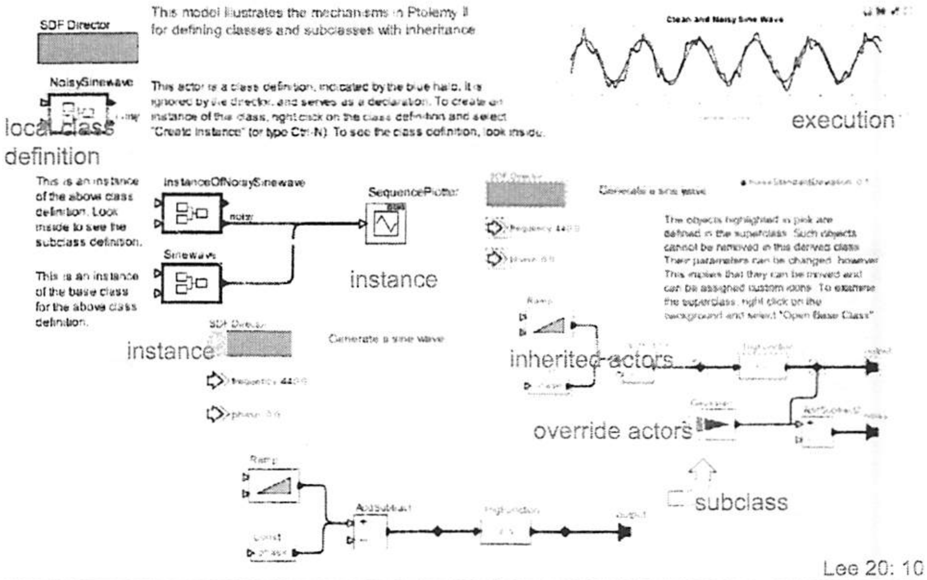
Making these objects instances of a class rather than copies reduced the XML representation of the model from 1.1 Mbytes to 87 kBytes, and offered a number of other advantages.

Subclasses, Inheritance? Interfaces, Subtypes? Aspects?

Now that we have classes, can we bring in more of the modern programming world?

- subclasses?
- inheritance?
- interfaces?
- subtypes?
- aspects?

Example Using AO Classes



This model illustrates the mechanisms in Ptolemy II for defining classes and subclasses with inheritance

This actor is a class definition, indicated by the blue halo. It is ignored by the director, and serves as a declaration. To create an instance of the class, right click on the class definition and select "Create instance" (or type Ctrl-N). To see the class definition, look inside.

This is an instance of the above class definition. Look inside to see the subclass definition.

This is an instance of the base class for the above class definition.

Generate a sine wave

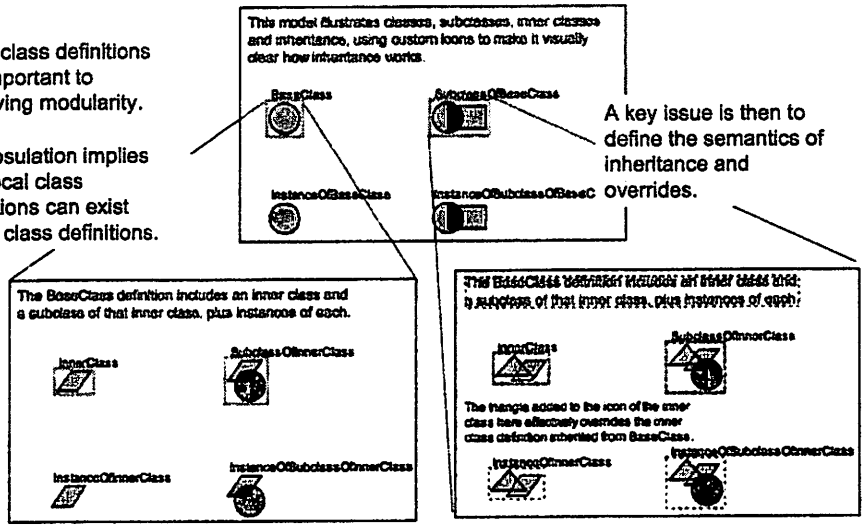
Generate a sine wave

The objects highlighted in pink are defined in the superclass. Such objects cannot be removed in this derived class. Their parameters can be changed, however. This implies that they can be moved and can be assigned custom icons. To examine the superclass, right click on the background and select "Open Base Class"

Inner Classes

Local class definitions are important to achieving modularity.

Encapsulation implies that local class definitions can exist within class definitions.



This model illustrates classes, subclasses, inner classes and inheritance, using custom icons to make it visually clear how inheritance works.

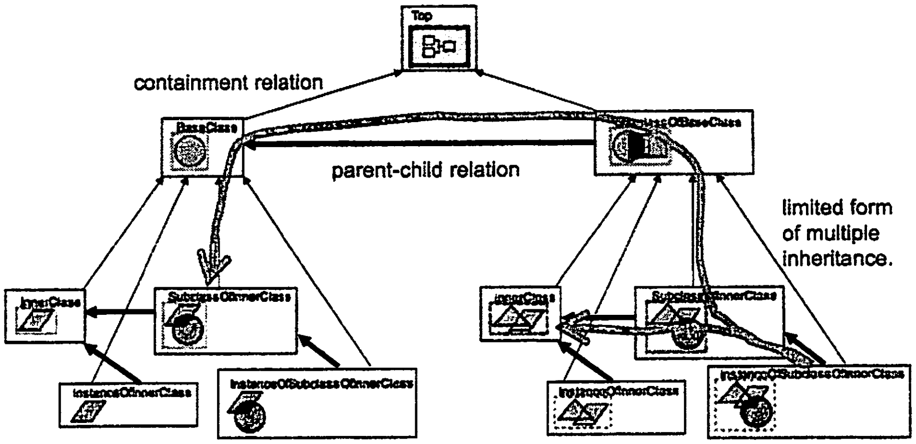
The BaseClass definition includes an inner class and a subclass of that inner class, plus instances of each.

The BaseClass definition includes an inner class and a subclass of that inner class, plus instances of each.

The triangle added to the icon of the inner class here effectively overrides the inner class definition inherited from BaseClass.

A key issue is then to define the semantics of inheritance and overrides.

Ordering Relations



containment relation

parent-child relation

limited form of multiple inheritance.

Mathematically, this structure is a *doubly-nested diposet*, the formal properties of which help to define a clean inheritance semantics. The principle we follow is that *local* changes override *global* changes.

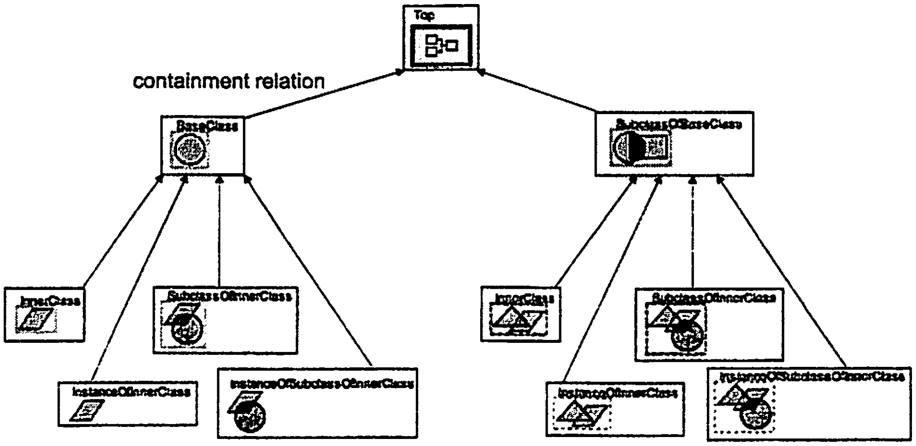
Formal Structure: Containment

- Let D be the set of *derivable objects* (actors, composite actors, attributes, and ports).
- Let $c: D \rightarrow D$ be a partial function (*containment*).
- Let $c^{+} \subset D \times D$ be the transitive closure of c (*deep containment*). When $(x, y) \in c^{+}$ we say that x **is deeply contained by** y.
- Disallow circular containment (anti-symmetry):

$$(x, y) \in c^{+} \Rightarrow (y, x) \notin c^{+}$$

So (D, c^{+}) is a strict poset.

Containment Relation



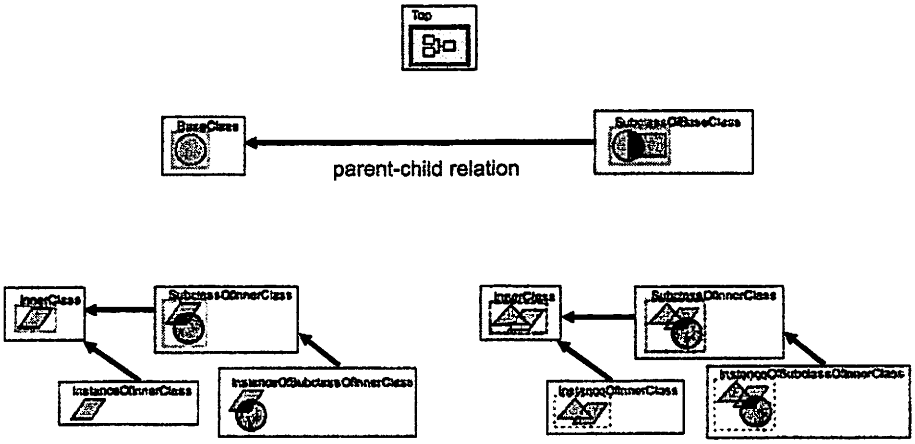
Formal Structure: Parent-Child

- Let $p: D \to D$ be a partial function (*parent*).
- Interpret $p(x) = y$ to mean y is the parent of x, meaning that either x is an instance of class y or x is a subclass of y . We say x is a child of y.
- Let $p^{+} \subset D \times D$ be the transitive closure of p (*deep containment*). When $(x,\ y) \in p^{+}$ we say that x **is descended from** y.
- Disallow circular containment (anti-symmetry):

$$(x,y) \in p^{+} \Rightarrow (y,x) \notin p^{+}$$

Then (D, p^{+}) is a strict poset.

Parent-Child Relation



Structural Constraint

We require that

$$(x,y) \in p^+ \Rightarrow (x,y) \notin c^+ \text{ and } (y,x) \notin c^+$$
$$(x,y) \in c^+ \Rightarrow (x,y) \notin p^+ \text{ and } (y,x) \notin p^+$$

That is, if x is deeply contained by y, then it cannot be descended from y, nor can y be descended from it.

Correspondingly, if x is descended from y, then it cannot be deeply contained by y, nor can y be deeply contained by it.

This is called a *doubly nested diposet* [Davis, 2000]

Labeling

- Let L be a set of identifying labels.
- Let $l: D \rightarrow L$ be a labeling function.
- Require that if $c(x) = c(y)$ then $l(x) \neq l(y)$.
  (Labels within a container are unique).

  Labels function like file names in a file system, and they can be appended to get "full labels" which are unique for each object within a single model (but are not unique across models).

Derived Relation

- Let $d \subset D \times D$ be the least relation so that $(x,\ y) \in d$ implies either that:

$$(x,\ y) \in p^{+}$$

or

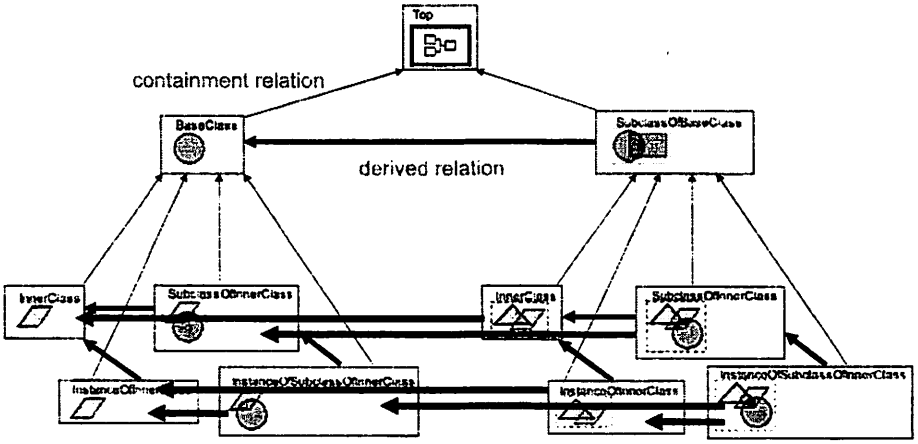
$$(c(x),\ c(y)) \in d \ \text{ and } \ l(x) = l(y)$$

x is derived from y if either:

x is descended from y or

x and y have the same label and the container of x is derived from the container of y.

Derived Relation



this object is derived from more than one other object: multiple inheritance.

Implied Objects and the Derivation Invariant

We say that y **is implied by** z in D if

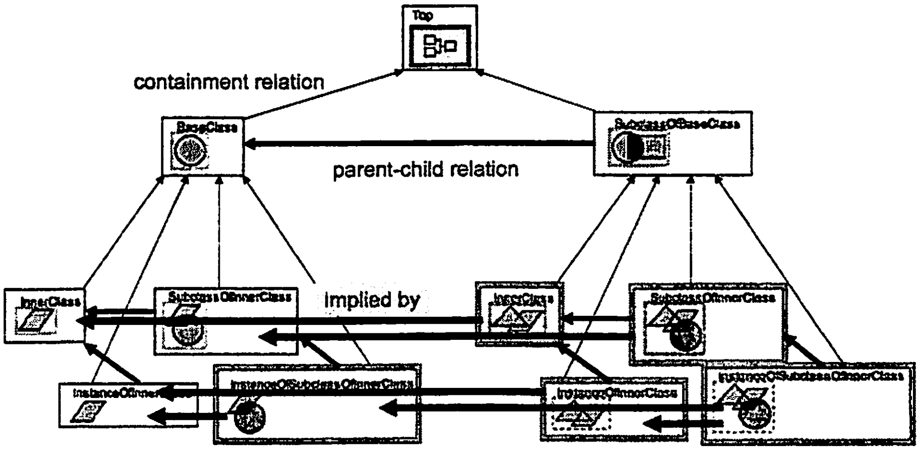
$$(y, z) \in d \text{ and } (y, z) \notin p^+.$$

I.e., y is implied by z if it is derived but is not a descendant.

Consequences:

- There is no need to represent implied objects in a persistent representation of the model, unless they somehow *override* the object from which they are derived.

Implied Objects



Derivation Invariant

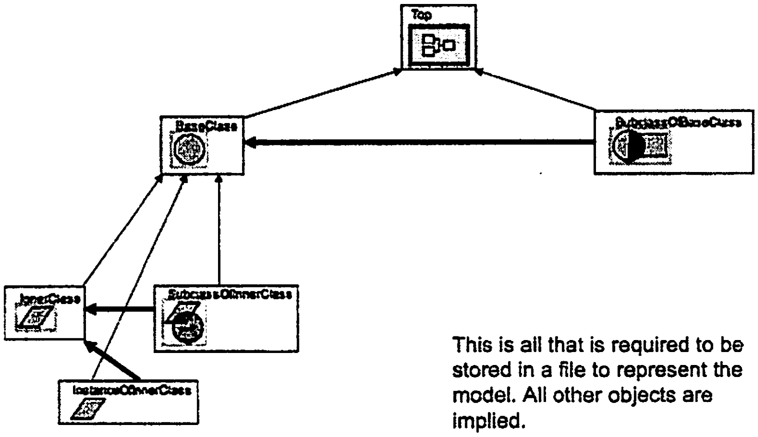
If x is derived from y then for all z where $c(z) = y$, there exists a z' where $c(z') = x$ and $l(z) = l(z')$ and either

1. $p(z)$ and $p(z')$ are undefined, or
2. $(p(z), p(z')) \in d$, or
3. $p(z) = p(z')$ and both $(p(z), y) \notin c^+$ and $(p(z'), x) \notin c^+$

I.e. z' is implied by z, and it is required that either

1. z' and z have no parents
2. the parent of z is derived from the parent of z' or
3. z' and z have the same parent, not contained by x or y

Persistent Representation



This is all that is required to be stored in a file to represent the model. All other objects are implied.

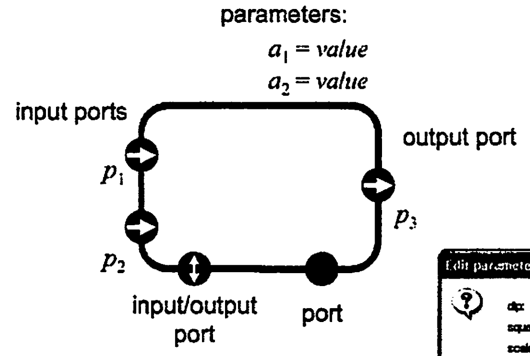
Values and Overrides

- Derived objects can contain more than the objects from which they derive (but not less).
- Derived objects can override their *value*.
- Since there may be multiple derivation chains from one object to an object derived from it, there are multiple ways to specify the value of the derived object.
- A reasonable policy is that more local overrides supercede less local overrides. Ensuring this is far from simple (but it is doable! see paper and/or Ptolemy II code).

Advanced Topics

- Interfaces and interface refinement
- Types, subtypes, and component composition
- Abstract actors
- Aspects
- Recursive containment

Defining Actor Interfaces: Ports and Parameters



parameters:

$a_1 = value$

$a_2 = value$

input ports

p_1

p_2

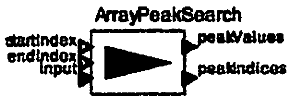
output port

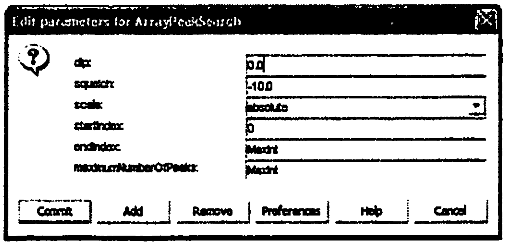
p_3

input/output port

port

Example:





dip: 0.0

squelch: -10.0

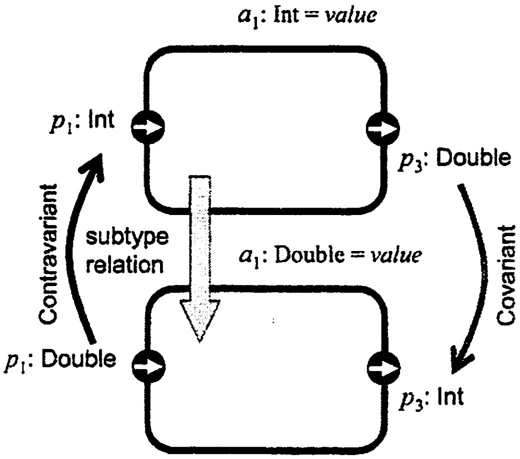
scale: absolute

startIndex: 0

endIndex: MaxInt

maximumNumberOfPeaks: MaxInt

Actor Subtypes



a_1: Int = *value*

p_1: Int

p_3: Double

subtype relation

a_1: Double = *value*

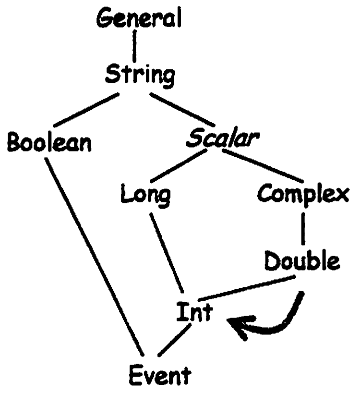
Contravariant

Covariant

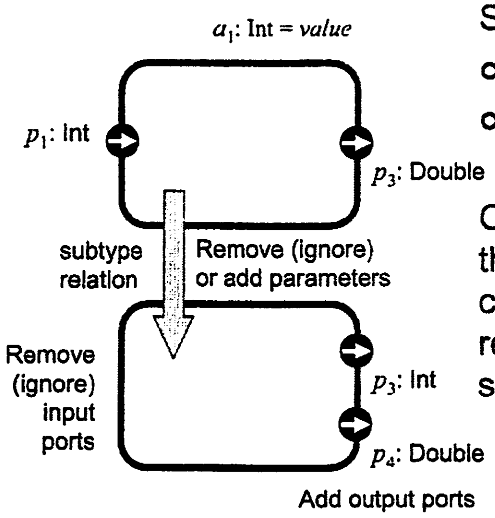
p_1: Double

p_3: Int

Example of a simple type lattice:



Actor Subtypes (cont)



Subtypes can have:

- Fewer input ports
- More output ports

Of course, the types of these can have co/contravariant relationships with the supertype.

Observations

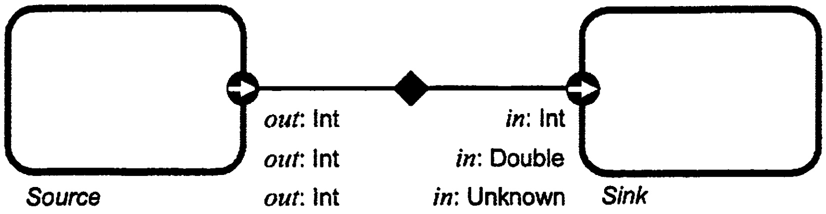
- Subtypes can remove (or ignore) parameters and also add new parameters because parameters always have a default value (unlike inputs, which a subtype cannot add)
- Subtypes cannot modify the types of parameters (unlike ports). Co/contravariant at the same time.
- PortParameters are ports with default values. They can be removed or added just like parameters because they provide default values.

Are there similar exceptions to co/contravariance in OO languages?

Composing Actors

A connection implies a type constraint. Can:

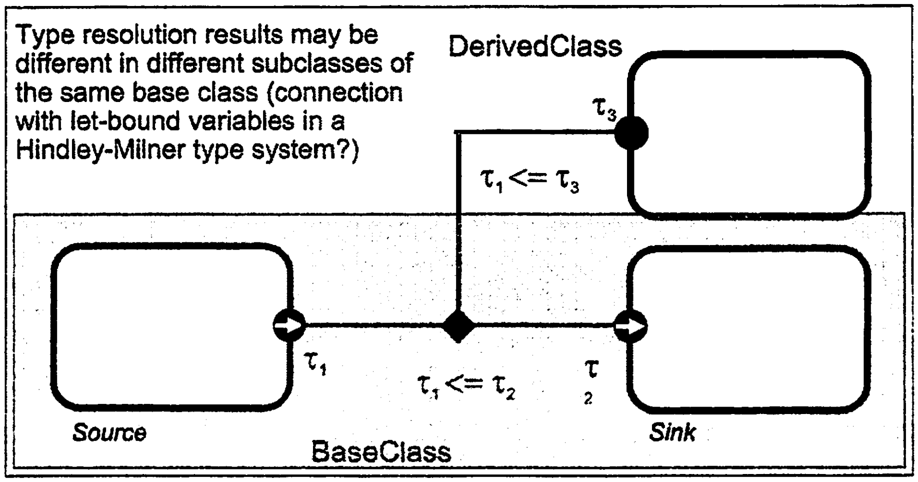
- check compatibility
- perform conversions
- infer types



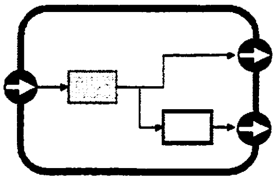
The Ptolemy II type system does all three.

What Happens to Type Constraints When a Subclass Adds Connections?

Type resolution results may be different in different subclasses of the same base class (connection with let-bound variables in a Hindley-Milner type system?)



Abstract Actors?



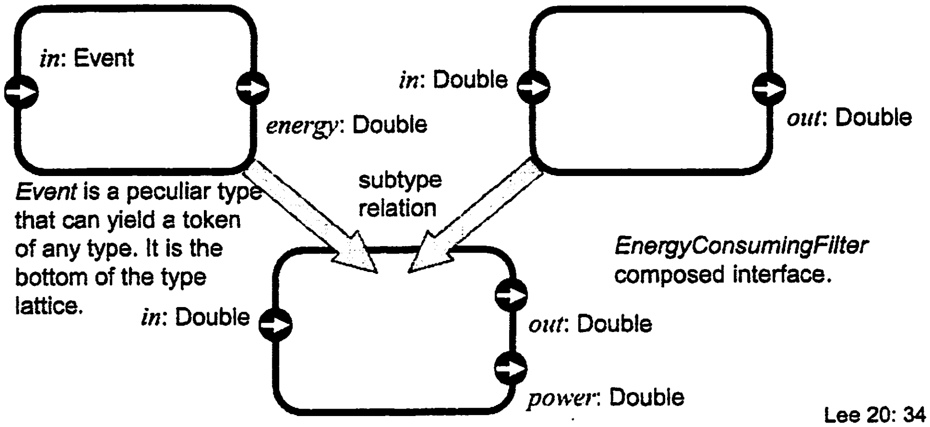
Suppose one of the contained actors is an interface only. Such a class definition cannot be instantiated (it is abstract). Concrete subclasses would have to provide implementations for the interface.

Is this useful?

Implementing Multiple Interfaces An Example

EnergyConsumer interface has a single output port that produces a Double representing the energy consumed by a firing.

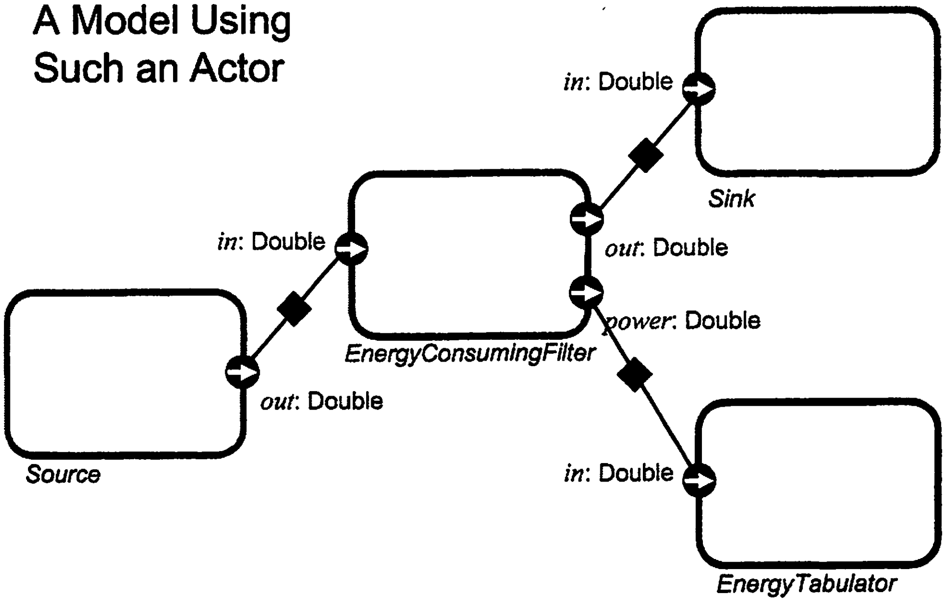
Filter interface for a stream transformer component.



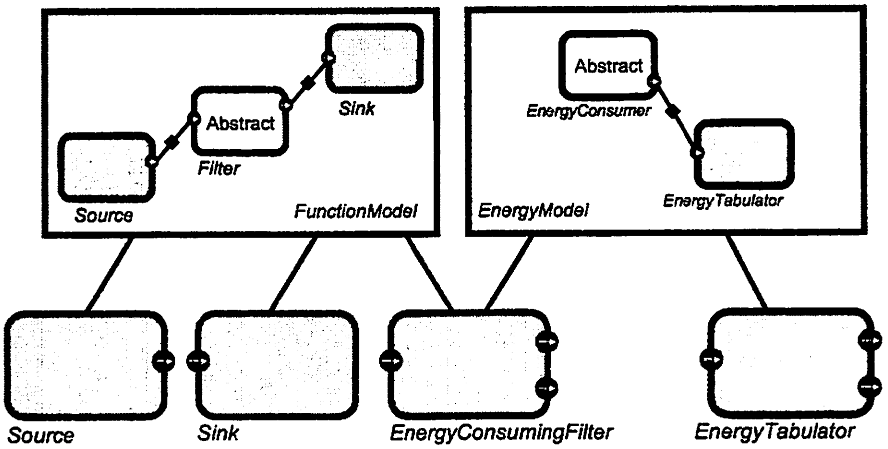
Event is a peculiar type that can yield a token of any type. It is the bottom of the type lattice.

EnergyConsumingFilter composed interface.

A Model Using Such an Actor



Heterarchy? Multi-View Modeling? Aspects?

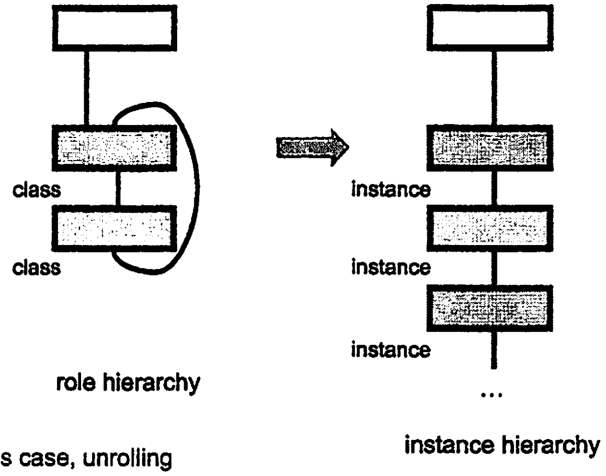


This is *multi-view modeling*, similar to what GME (Vanderbilt) can do.

Is this an *actor-oriented* version of *aspect-oriented* programming?

Is this what Metropolis does with function/architecture models?

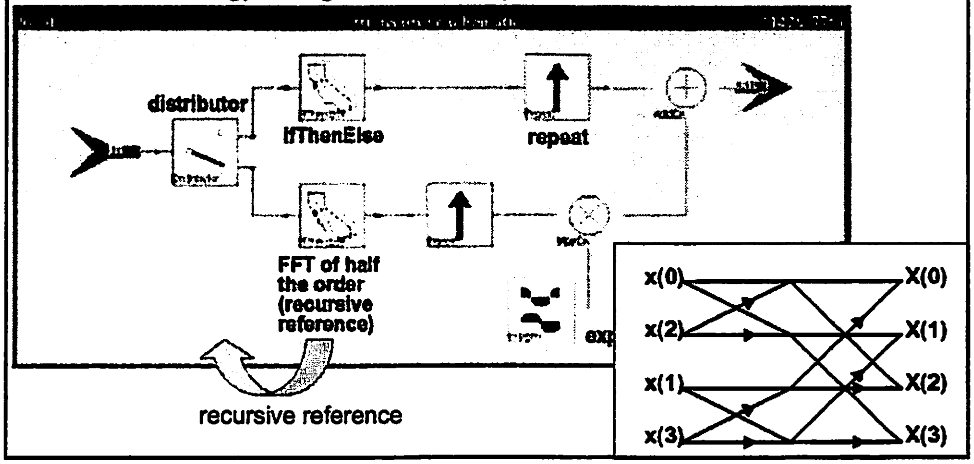
Recursive Containment
Can Hierarchical Classes Contain Instances of Themselves?



Note that in this case, unrolling cannot occur at "compile time".

Primitive Realization of this in Ptolemy Classic

FFT implementation in Ptolemy Classic (1995) used a partial evaluation strategy on higher-order components.



Conclusion

- Actor-oriented design remains a relatively immature area, but one that is progressing rapidly.
- It has huge potential.
- Many questions remain…