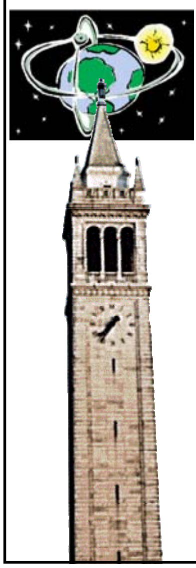


Using the Principles of Synchronous Languages in Discrete-event and Continuous-time Models

Edward A. Lee

Robert S. Pepper Distinguished Professor
Chair of EECS
UC Berkeley

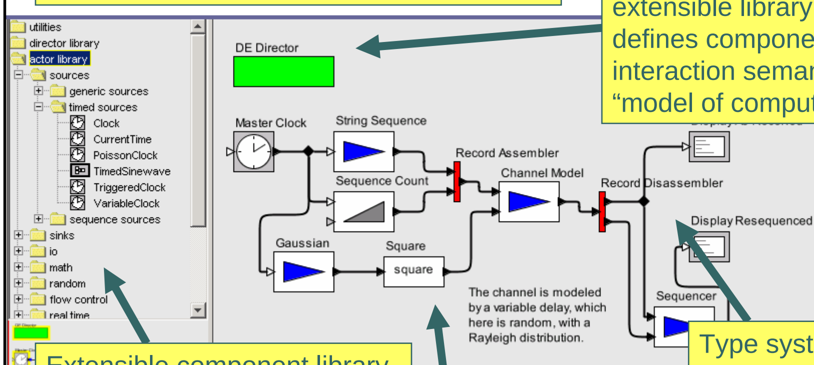
With special thanks to Stephen Edwards, Xioajun Liu, Eleftherios Matsikoudis, and Haiyang Zheng.

Chess Seminar
October 23, 2007
Berkeley, CA

Ptolemy II: Our Laboratory for Studying Concurrent Models of Computation

Concurrency management supporting dynamic model structure.

Director from an extensible library defines component interaction semantics or "model of computation."



Extensible component library.

Type system for communicated data

Visual editor for defining models

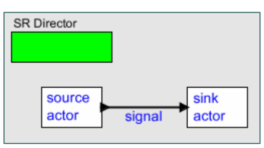
Some Models of Computation Implemented in Ptolemy II

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

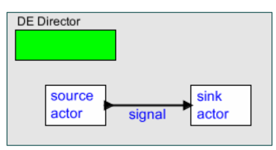
This talk will start with SR…

… and then show how CT and DE can be built on it.

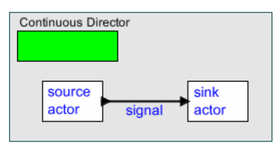
Signals in Ptolemy II under Three Directors



A signal has a value or is absent at each tick of a “clock.” By default, all ticks of the “clock” occur at model time 0.0, but they can optionally be spaced in time by setting the *period* parameter of the SR Director.

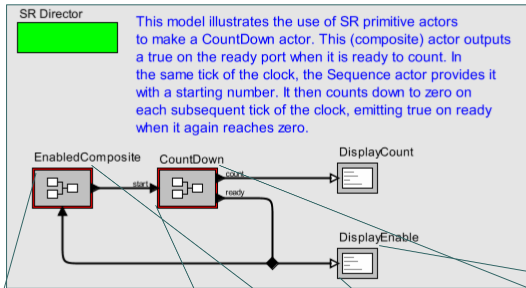


A signal is a set of events with time stamps (in model time) and the DE Director is responsible for presenting these events in time-stamp order to the destination actor.

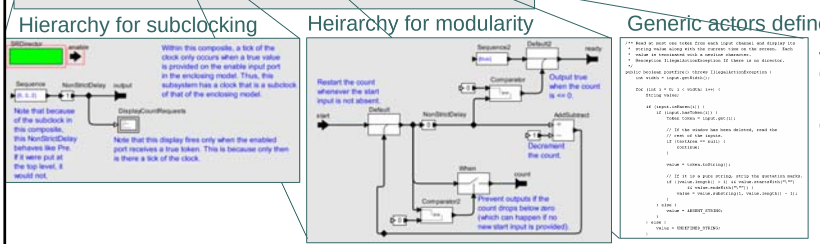


A signal is defined everywhere (in model time) and the Continuous Director chooses where it is evaluated. The value of the signal may be “absent,” allowing for signals that are discrete or have gaps.

Synchronous/Reactive (SR) Models in Ptolemy II

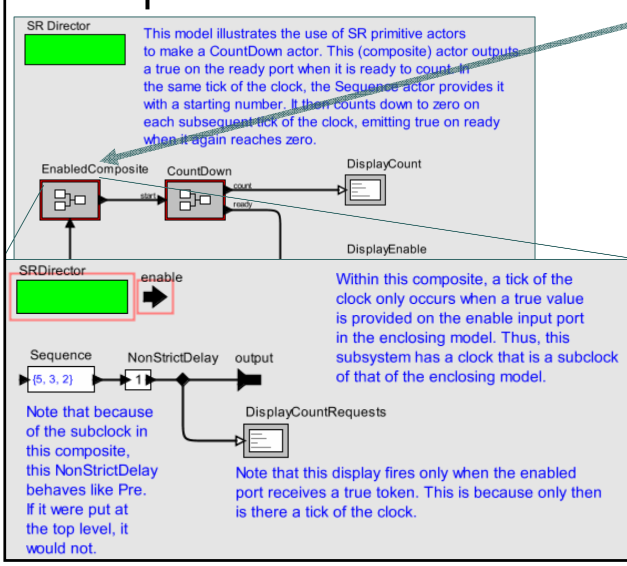


- Synchronous coordination language
- With a few specialized actors
  - When
  - Default
- Clocking
  - Global clock with subclocks
  - Structured multiclock
  - Structured nondeterminism



Java,
C,
MATLAB,
Cal,
Python, etc.

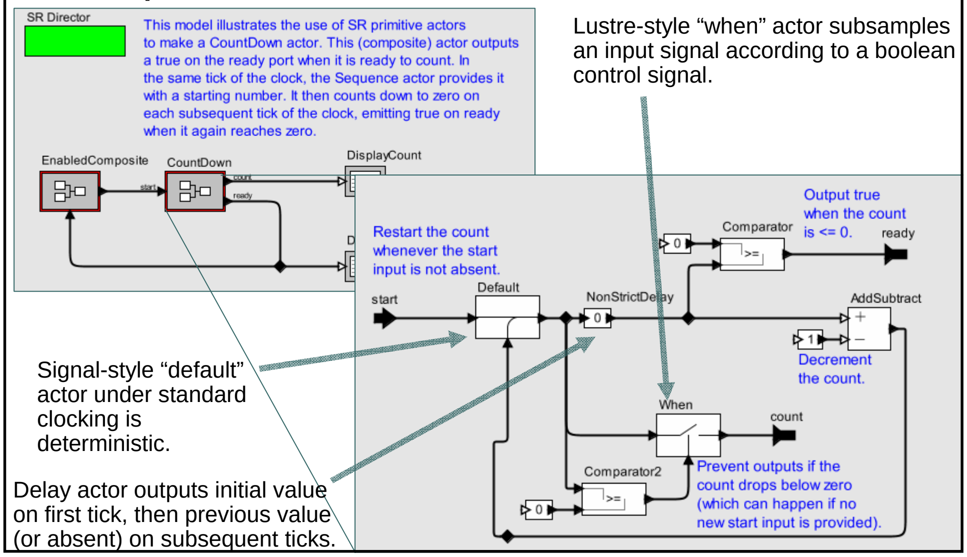
Synchronous/Reactive (SR) Models in Ptolemy II



Composite actor interfaces clock domains. In this case, the "enable" input port, when true, triggers a tick of the internal model, yielding "structured multiclock models."

A variant of this composite actor could run the internal system in a separate thread, yielding asynchronous, nondeterministic clock relationships. Combined with the Default actor, this yields a structured form of Signal's nondeterministic multiclock models.

Synchronous/Reactive (SR) Models in Ptolemy II

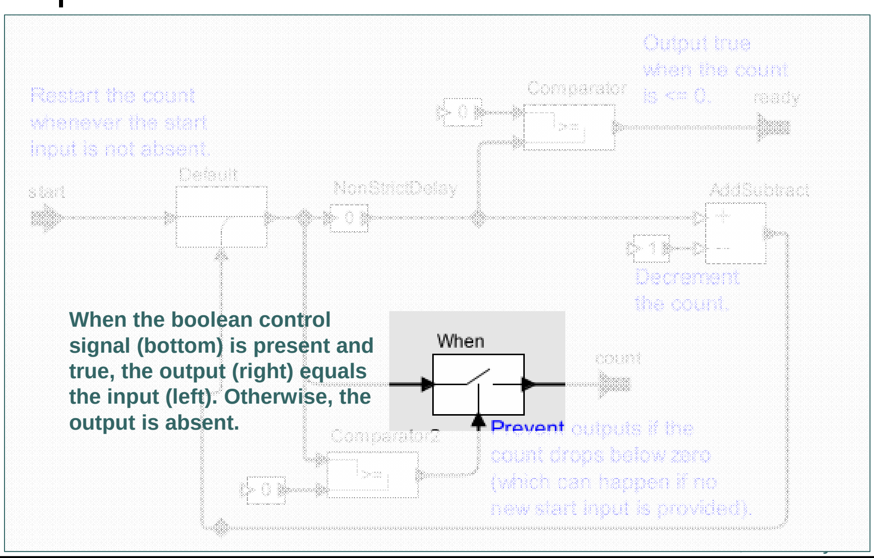


Lustre-style “when” actor subsamples an input signal according to a boolean control signal.

Signal-style “default” actor under standard clocking is deterministic.

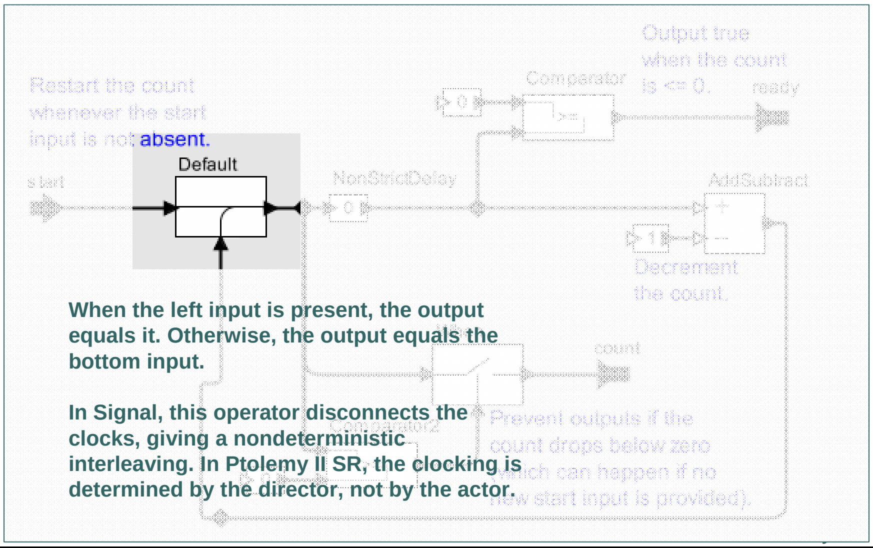
Delay actor outputs initial value on first tick, then previous value (or absent) on subsequent ticks.

Lustre-Style When Actor



When the boolean control signal (bottom) is present and true, the output (right) equals the input (left). Otherwise, the output is absent.

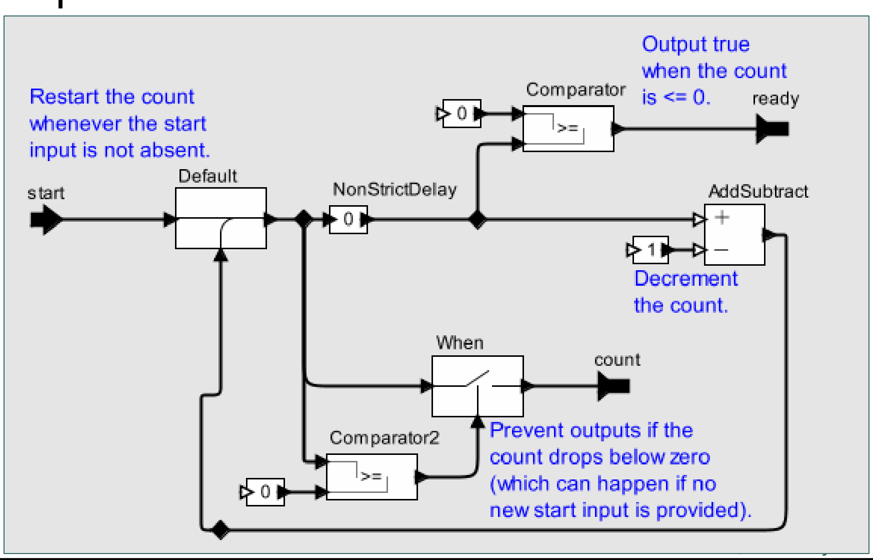
Signal-Style Default Actor



When the left input is present, the output equals it. Otherwise, the output equals the bottom input.

In Signal, this operator disconnects the clocks, giving a nondeterministic interleaving. In Ptolemy II SR, the clocking is determined by the director, not by the actor.

Guarded Count

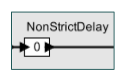


How Does This Work? Execution of Ptolemy II Actors

Flow of control:

- Initialization
- Execution
- Finalization

How Does This Work? Execution of Ptolemy II Actors



Flow of control:

- Initialization
- Execution
- Finalization

Initialize actors

How Does This Work?
Execution of Ptolemy II Actors

Flow of control:

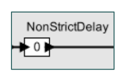
- Initialization
- Execution
- Finalization

Iterate

```
while (!fixpoint) {
    if (prefire()) {
        fire();
    }
}
postfire();
```

Only the postfire() method can change the state of the actor.

How Does This Work?
Execution of Ptolemy II Actors



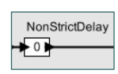
Flow of control:

- Initialization
- Execution
- Finalization

Definition of the NonStrictDelay Actor (Sketch)


```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        _previousToken = initialValue.getToken();
    }

    public boolean prefire() {
        return true;
    }

    public void fire() {
        if (_previousToken != null) {
            if (_previousToken == AbsentToken.ABSENT) {
                output.sendClear(0);
            } else {
                output.send(0, _previousToken);
            }
        } else {
            output.sendClear(0);
        }
    }

    public boolean postfire() {
        if (input.isKnown(0)) {
            if (input.hasToken(0)) {
                _previousToken = input.get(0);
            } else {
                _previousToken = AbsentToken.ABSENT;
            }
        }
        return true;
    }
}
```


Definition of the NonStrictDelay Actor (Sketch)



initialization

```
protected Token _previousToken;
public Parameter initialValue;

public void initialize() {
    _previousToken = initialValue.getToken();
}
```

```
            if (_previousToken == AbsentToken.ABSENT) {
                output.sendClear(0);
            } else {
                output.send(0, _previousToken);
            }
        } else {
            output.sendClear(0);
        }
    }

    public boolean postfire() {
        if (input.isKnown(0)) {
            if (input.hasToken(0)) {
                _previousToken = input.get(0);
            } else {
                _previousToken = AbsentToken.ABSENT;
            }
        }
        return true;
    }
}
```

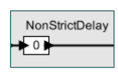
Definition of the NonStrictDelay Actor (Sketch)


```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        _previ
    }

    public boo
        return
    }

    public voi
        if (_p
            if

            }

            }
        } else
            output.sendClear(0);
        }
    }

    public boolean postfire() {
        if (input.isKnown(0)) {
            if (input.hasToken(0)) {
                _previousToken = input.get(0);
            } else {
                _previousToken = AbsentToken.ABSENT;
            }
        }
        return true;
    }
}
```

prefire: can the actor fire?

```
public boolean prefire() {
   return true;
}
```


Definition of the NonStrictDelay Actor (Sketch)




```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        _previousToken = initialValue.getToken();
    }

    public boolean prefire() {
        return
    }

    public voi
        if (_p
            if

            }

            }
        } else
            ou
        }
    }

    public boo
        if (in
            if

            }

            }
        }
        return true;
    }
}
```

fire: produce outputs (in this case, the output does not depend on the input).

```
public void fire() {
   if (_previousToken != null) {
       if (_previousToken == AbsentToken.ABSENT) {
           output.sendClear(0);
       } else {
           output.send(0, _previousToken);
       }
   } else {
       output.sendClear(0);
   }
}
```

Definition of the NonStrictDelay Actor (Sketch)


```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        _previousToken = initialValue.getToken();
    }

    public boolean prefire() {
        return true;
    }

    public void fire() {
        if (_p
            if
            } 
            }
        } else
            ou
        }
    }

    public boo
        if (in
            if
            }
            }
        }
        return
    }
}
```

```
public boolean postfire() {
    if (input.isKnown(0)) {
        if (input.hasToken(0)) {
            _previousToken = input.get(0);
        } else {
            _previousToken = AbsentToken.ABSENT;
        }
    }
    return true;
}
```

postfire: record state changes

Standard Synchronous Semantics

Let V be a family of values (a data type, or alphabet). Let

$$V_\perp = V \cup \{\varepsilon, \perp\}$$

be the set of values plus "absent" (ε) and "unknown" ($\perp$). Define a flat partial order $<$ where $\perp < \varepsilon$ and $\perp < v$ for all $v \in V$. At each tick, every actor realizes a monotonic firing function (in this order). The signal values at the tick are the least fixed point of the composition of these firing functions.

Ptolemy II SR Domain has a Constructive Version of the Synchronous Semantics

Let

$$V_\varepsilon = V \cup \{\varepsilon\}$$

(No "unknown" and no partial order). Let $\mathbb{N}$ be the non-negative integers. Let s be a signal, given as a partial function:

$$s : \mathbb{N} \rightharpoonup V_\varepsilon$$

defined on an initial segment of $\mathbb{N}$. An actor is a function mapping input signals into output signals. This function is required to be monotonic in a prefix order. The signals in a model are the least fixed point of the composition of these actor functions.

Xiaojun Liu and Edward A. Lee. " CPO Semantics of Timed Interactive Actor Networks," EECS Department, University of California, Berkeley UCB/EECS-2006-67, May 18, 2006.

Metric Time in SR

- By default, "time" does not advance when executing an SR model in Ptolemy II ("current time" remains at 0.0, a real number).
- Optionally, the SR Director can increment time by a fixed amount on each clock tick.

Our Model of Time: *Super-Dense Time*

Let $T = \mathbb{R}_+ \times \mathbb{N}$ be a set of "tags" where $\mathbb{N}$ is the natural numbers, and give a signal s as a partial function:

$$s : T \rightharpoonup V_\varepsilon$$

defined on an initial segment of T, assuming a lexical ordering on T:

$$(t_1, n_1) \leq (t_2, n_2) \iff t_1 < t_2, \text{ or } t_1 = t_2 \text{ and } n_1 \leq n_2 \,.$$

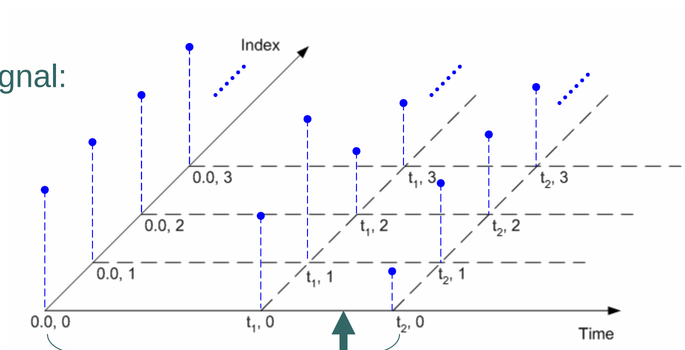
This allows signals to have a sequence of values at any real time t.

[Manna and Pnueli, 1992]

Super Dense Time

Example signal:

Values V

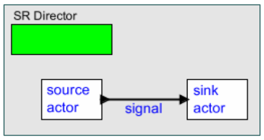


Initial segment $I \subseteq \mathbb{R}_+ \times \mathbb{N}$ where the signal is defined

Absent: $s(\tau) = \varepsilon$ for almost all $\tau \in I$.

Time in SR Models in Ptolemy II

$s : T \rightharpoonup V_{\varepsilon}$



A signal has a value or is absent at each tick of a "clock." By default, all ticks of the "clock" occur at model time 0.0, but they can optionally be spaced in time by setting the *period* parameter of the SR Director.

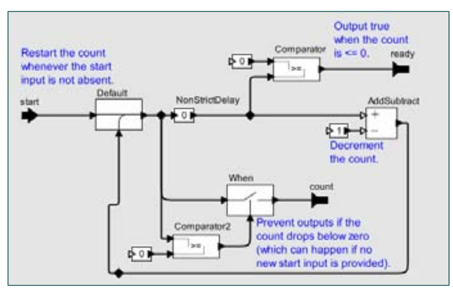
Assume the *period* parameter of the SR Director is given by p. The default value is $p = 0$.

- A *tag* is a time-index pair, $\tau = (t, n) \in T = \mathbb{R}_+ \times \mathbb{N}$.
- If $p = 0$, then by default, only the index advances, so actors are fired at model times $(0,0), (0,1), (0,2), \cdots$. Time never advances.
- If $p > 0.0$, then actors are fired at model times $(0,0), (p,0), (2p,0), \cdots$. Semantically, signals are "absent" at tags in between.

Execution of an SR Model (Conceptually)

- Start with all signals empty (i.e. defined on the empty initial segment).
- Initialize all actors.
- Invoke the following on all actors until either all signals are defined on the initial segment {(0,0)} or no progress can be made:

  if (prefire()) { fire(); }
- If not all signals are defined on {(0,0)}, declare a causality loop.
- Invoke postfire() for all actors.
- Choose the next tag *t* ((0,1) or (p, 0))
- Repeat to define signals on the initial segment [(0, 0), *t*].
- Etc.



The correctness of this is guaranteed by the fixed point semantics. Efficiency, of course, depends on being smart about the order in which actors are invoked.

Metric Time in SR

- By default, “time” does not advance when executing an SR model in Ptolemy II (“current time” remains at 0.0, a real number).
- Optionally, the SR Director can increment time by a fixed amount on each clock tick.

- *More interestingly, SR can be embedded within timed MoCs that model the environment and govern the passage of time.*

Some Models of Computation Implemented in Ptolemy II

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

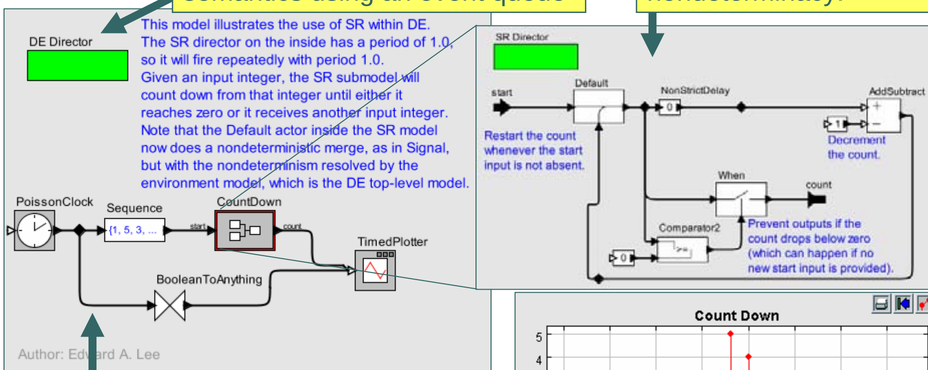
This talk started with SR…

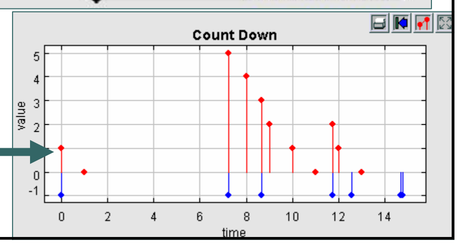
next show how DE can be built on it.

Discrete Events (DE): A Timed Concurrent Model of Computation

DE Director implements timed semantics using an event queue

SR subsystem implements structured nondeterminacy.





Actors communicate via “signals” that are marked point processes (discrete, valued, time-stamped events).

Advancing Time

- A signal is a partial function

$$s : T \rightharpoonup V_\varepsilon$$

  defined on an initial segment of

$$T = \mathbb{R}_+ \times \mathbb{N}$$

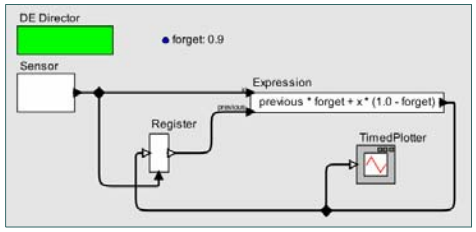
- But how to increment the initial segment on which the signal is defined? It won’t work to just proceed to the next one, as we did with SR.

Execution of a DE Model (Conceptually)

- Start with all signals empty.
- Initialize all actors (some will post tags on the event ueue).
- Take the smallest tag (t, n) from the event queue.
- Invoke the following on all actors that have input events until either all signals are defined on the initial segment S = [(0,0), (t,n)] or no progress can be made:

  if (prefire()) { fire(); }

- If not all signals are defined on S, declare a causality loop.
- Invoke postfire() for all actors (some will post tags on the event queue).
- Repeat with the next smallest tag on the event queue.



This is exactly the execution policy of SR, except that rather than just choosing the next tag in the tag set, we use a sorted event queue to choose an interval over which to increment the initial segment.

Lee, Berkeley 31

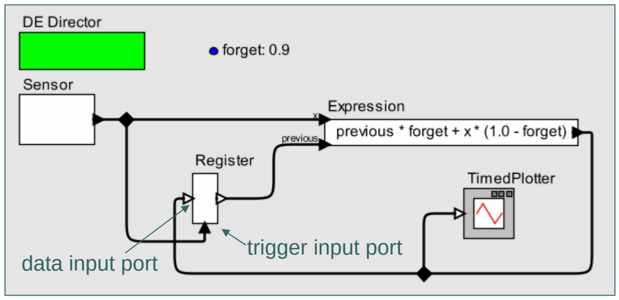
Subtle Difference Between SR and DE

- In SR, every actor is fired at every tick of its clock, as determined by a clock calculus and/or structured subclocks.
- In DE, an actor is fired at a tag only if it has input events at that tag or it has previously posted an event on the event queue with that tag.

In DE semantics, event counts may matter. If every actor were to be fired at every tick, then adding an actor in one part of a model could change the behavior in another part of the model in unexpected ways.

Lee, Berkeley 32

Feedback in DE Presents Semantic Challenges



In this model, a sensor produces measurements that are combined with previous measurements using an exponential forgetting function.

The feedback loop makes it impossible to present the Register actor with all its inputs at any tag before firing it.

Solving Feedback Loops

Solutions implemented by others:

- Find algebraic solution
- All actors have time delay
- Some actors have time delay, and every directed loop must have an actor with time delay.
- All actors have delta delay
- Some actors have delta delay and every directed loop must have an actor with delta delay.

Although each of these solutions is used, all are problematic.

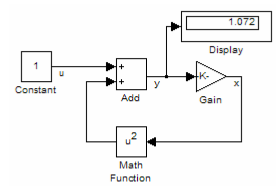
The root of the problem is simultaneous events.

Consider “Find Algebraic Solution”

This solution is used by Simulink, but is ill posed.

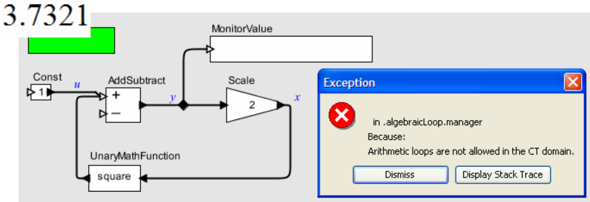
Consider:

$$y(t) = x^2(t) + u(t)$$
$$x(t) = K\, y(t)$$

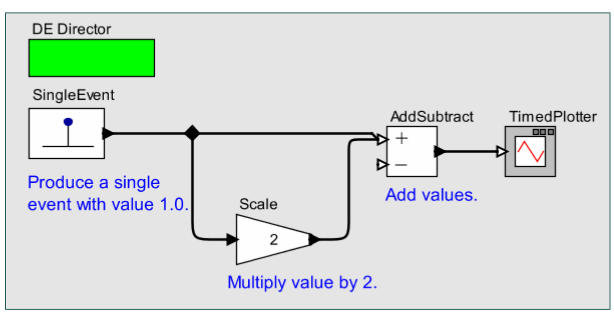


This has two solutions:

$$y(t) = 1.072, \quad x(t) = 0.268, \text{ or}$$
$$y(t) = 14.9282, \quad x(t) = 3.7321$$



Consider “All Actors Have Time Delay”



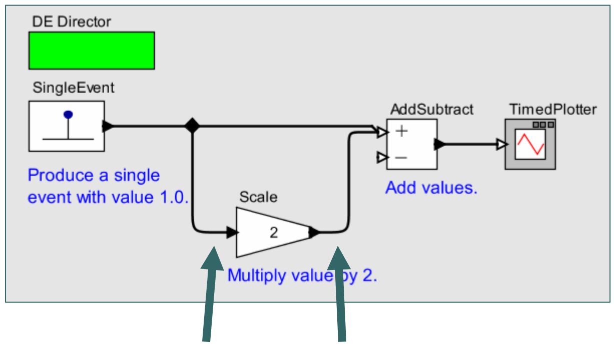
If all actors have time delay, this produces either:

- Event with value 1 followed by event with value 2, or
- Event with value 1 followed by event with value 3.

(the latter if signal values are persistent).

Neither of these is likely what we want.

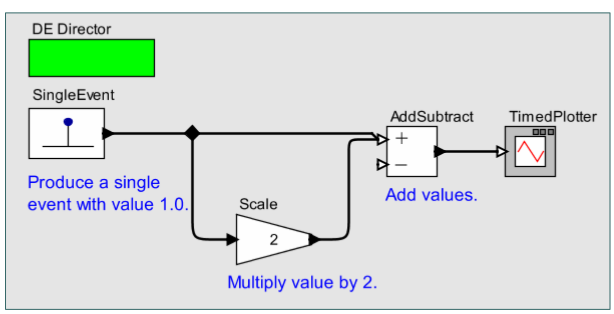
Consider “All Actors Have Delta Delay”



With delta delays, if an input event is ((t, n), v), the corresponding output event is ((t, n+1), v’). Every actor is assumed to give a delta delay.

This style of solution is used in VHDL.

Consider “All Actors Have Delta Delay”



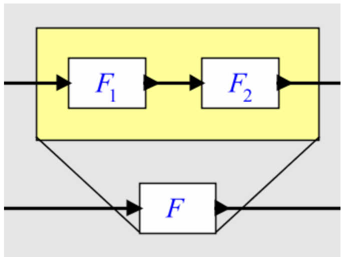
If all actors have a delta delay, this produces either:

- Event with value 1 followed by event with value 2, or
- Event with value 1 followed by event with value 3

(the latter if signal values are persistent, as in VHDL).

Again, neither of these is likely what we want.

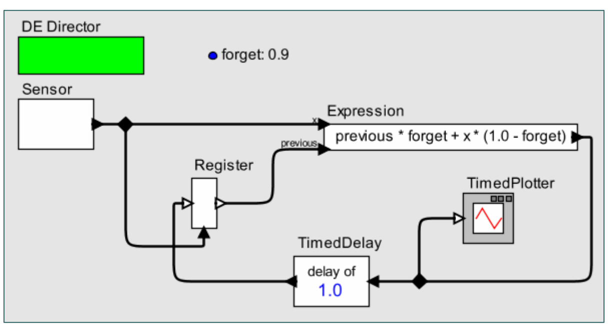
More Fundamental Problem: Delta Delay Semantics is Not Compositional



The top composition of two actors will have a two delta delays, whereas the bottom abstraction has only a single delta delay.

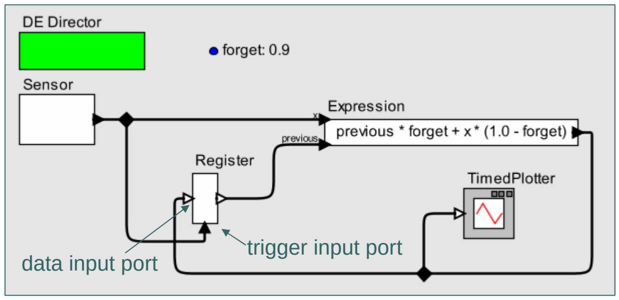
Under delta delay semantics, a composition of two actors cannot have the semantics of a single actor.

Consider "Some actors have time delay, and every directed loop must have an actor with time delay."



Any non-zero time delay imposes an upper bound on the rate at which sensor data can be accepted. Exceeding this rate will produce erroneous results.

Consider "Some actors have delta delay, and every directed loop must have an actor with delta delay."



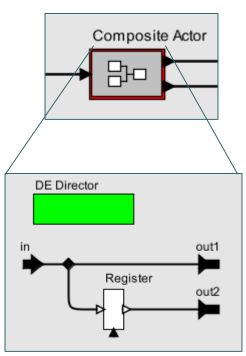
The output of the Register actor must be at least one index later than the data input, hence this actor has at least a delta delay.

To schedule this, could break the feedback loop at actors with delta delay, then do a topological sort.

Lee, Berkeley 41

Naïve Topological Sort is not Compositional

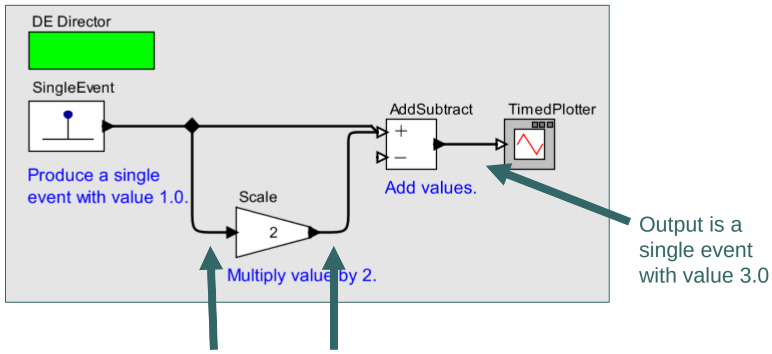
Breaking loops where an actor has a delta delay and performing a topological sort is not a compositional solution:



Does this composite actor have a delta delay or not?

Lee, Berkeley 42

Our Answer: No Required Delay, and Loops Have (Unique) Least Fixed Points Semantics

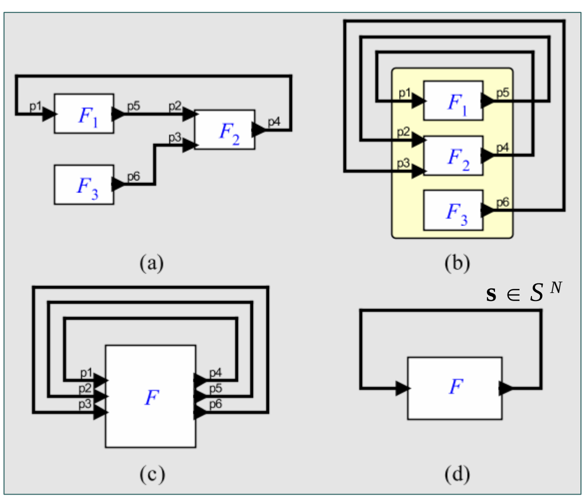


Given an input event ((t, n), v), the corresponding output event is ((t, n), v'). The actor has no delay.

The semantics is similar to SR, except that time may advance by a variable amount.

Lee, Berkeley 43

Existence and Uniqueness of the Least Fixed Point Solution (Summary)



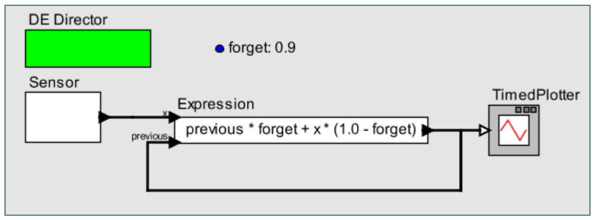
- Signal: $s\colon \mathbb{R}_+ \times \mathbb{N} \rightharpoonup V_\varepsilon$
- Set of signals: S
- Tuples of signals: $\mathbf{s} \in S^N$
- Actor: $F\colon S^N \to S^M$

A unique least fixed point, $\mathbf{s} \in S^N$ such that $F(\mathbf{s}) = \mathbf{s}$, exists and be constructively found if S^N is a CPO and F is (Scott) continuous.

Under our execution policy, actors are usually (Scott) continuous.

Lee, Berkeley 44

But: Need to Worry About Liveness: Deadlocked Systems



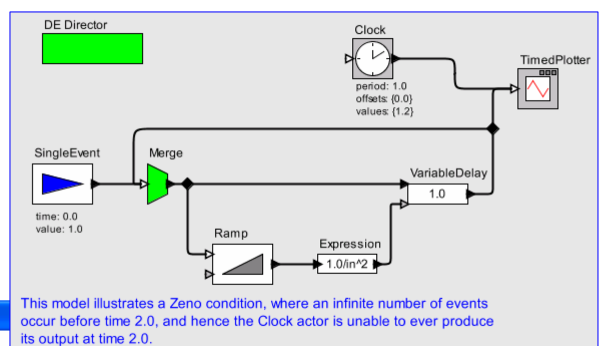
Existence and uniqueness of a solution is not enough.

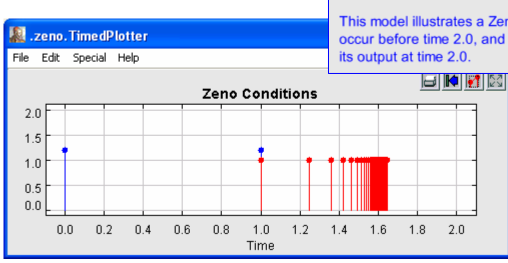
The least fixed point of this system consists of empty signals. It is deadlocked!

This is the same as SR causality loops.

Another Liveness Concern is *not* Present in SR: *Zeno Systems*

DE systems may have an infinite number of events in a finite amount of time. These “Zeno systems” can prevent time from advancing.





In this case, our execution policy fails to implement the Knaster-Tarski constructive procedure because some of the signals are not total.

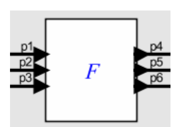
Liveness

- A signal is *total* if it is defined for all tags in T.
- A model with no inputs is *live* if all signals are total.
- A model with inputs is *live* if all input signals are total implies all signals are total.

Liveness ensures freedom from deadlock and Zeno.

- *Whether a model is live is, in general, undecidable.*
- We have developed a useful sufficient condition based on *causality* that ensures liveness.

Causality Ensures Liveness of an Actor



A monotonic actor F is *causal* if for all sets of input signals S_i, the corresponding set of output signals $S_o = F(S_i)$ satisfy

$$\bigcap_{s \in S_i} \mathrm{dom}(s) \subseteq \bigcap_{s \in S_o} \mathrm{dom}(s)\,.$$

An immediate consequence of this definition is that a causal actor is live. Thus, whether a composition of actors is causal will tell us whether it is live.

Causality does not imply continuity and continuity does not imply causality. Continuity ensures existence and uniqueness of a least fixed point, whereas causality ensures liveness.

Strict Causality Ensures Liveness of a Feedback Composition

A composition of causal actors without directed cycles is itself a causal actor. With cycles, we need:

- A monotonic actor F is *strictly causal* if for all sets of input signals S_i, the corresponding set of output signals $S_o = F(S_i)$ either consists only of total signals (defined over all T) or

$$\bigcap_{s \in S_i} \mathrm{dom}(s) \subset \bigcap_{s \in S_o} \mathrm{dom}(s) .$$

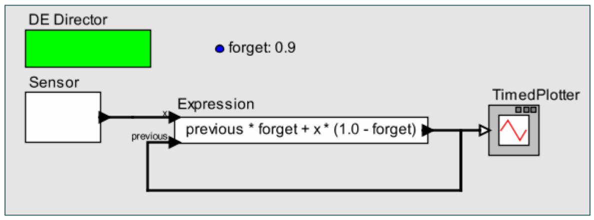
($\subset$ denotes strict subset). If F is a strictly causal actor with one input and one output, then $F(s_\perp) \neq s_\perp$. F must "come up with something from nothing."

Continuity, Liveness, and Causality

Theorem: Given a totally ordered tag set and a network of causal and continuous actors where in every dependency loop in the network there is at least one strictly causal actor, then the network is a causal and continuous actor.

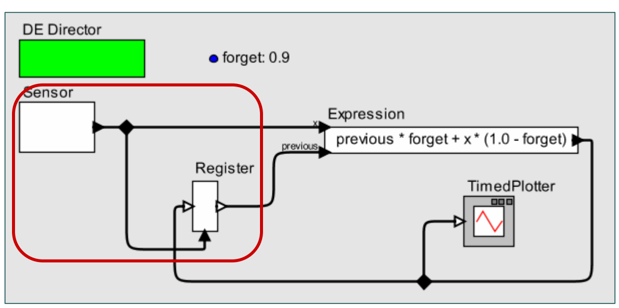
This gives us sufficient, but not necessary condition for freedom deadlock and Zeno.

Recall Deadlocked System



The feedback loop has no strictly causal actor.

Feedback Loop that is Not Deadlocked



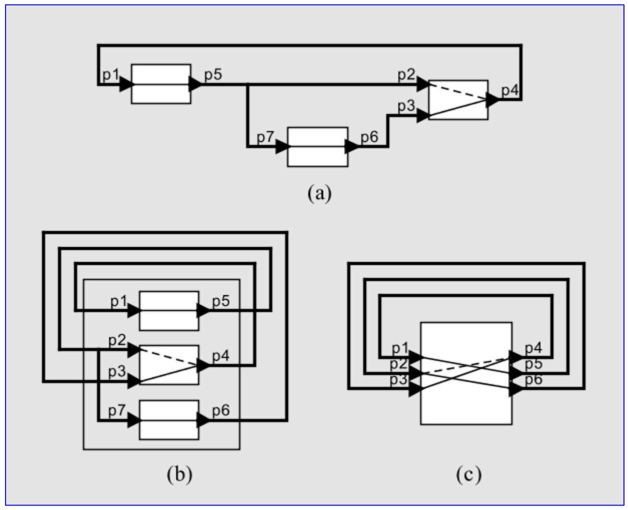
This feedback loop also has no strictly causal actor, unless…

We aggregate the two actors as shown into one.

Causality Interfaces Make Scheduling of Execution and Analysis for Liveness Efficient

A causality interface exposes just enough information about an actor to make scheduling and liveness analysis efficient.

An algebra of interfaces enables inference of the causality interface of a composition.



Models of Computation Implemented in Ptolemy II

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

Done

But will also establish connections with Continuous Time (CT).

Standard Model for Continuous-Time Signals

In ODEs, the usual formulation of the signals of interest is a function from the time line (a connected subset of the reals) to the reals:

$$p\colon \mathbb{R}_+ \to \mathbb{R}^n$$

$$\dot{p}\colon \mathbb{R}_+ \to \mathbb{R}^n$$

$$\ddot{p}\colon \mathbb{R}_+ \to \mathbb{R}^n$$

Such signals are continuous at $t \in \mathbb{R}_+$ if (e.g.):

$$\forall\, \epsilon > 0, \exists\, \delta > 0, \text{s.t.} \forall \tau \in (t-\delta, t+\delta), \quad ||\dot{p}(t) - \dot{p}(\tau)|| < \epsilon$$

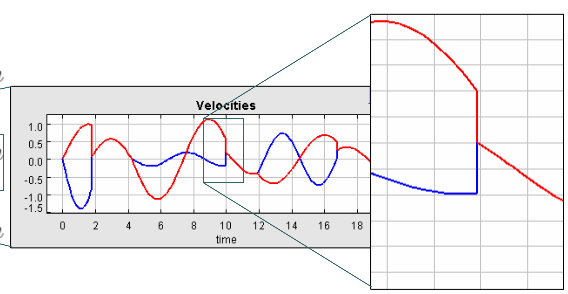
Piecewise Continuous Signals

In hybrid systems of interest, signals have discontinuities.

$$p\colon \mathbb{R}_+ \to \mathbb{R}^n$$

$$\dot{p}\colon \mathbb{R}_+ \to \mathbb{R}^n$$

$$\ddot{p}\colon \mathbb{R}_+ \to \mathbb{R}^n$$



Piecewise continuous signals are continuous at all $t \in \mathbb{R}_+ \setminus D$ where $D \subset \mathbb{R}_+$ is a *discrete set*.[1]

[1] A set D with an order relation is a *discrete set* if there exists an order embedding to the integers.

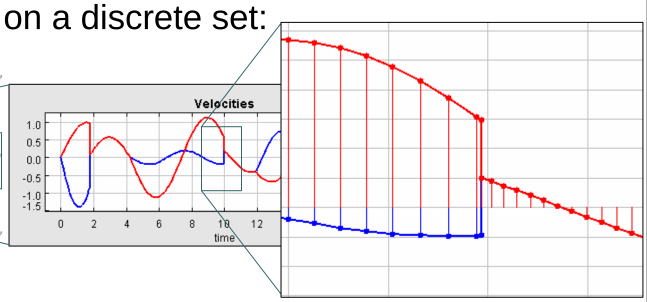
Piecewise Continuous Signals Evaluated in a Computer

A computer execution of a hybrid system is constrained to provide values on a discrete set:

$$p\colon \mathbb{R}_+ \to \mathbb{R}^n$$

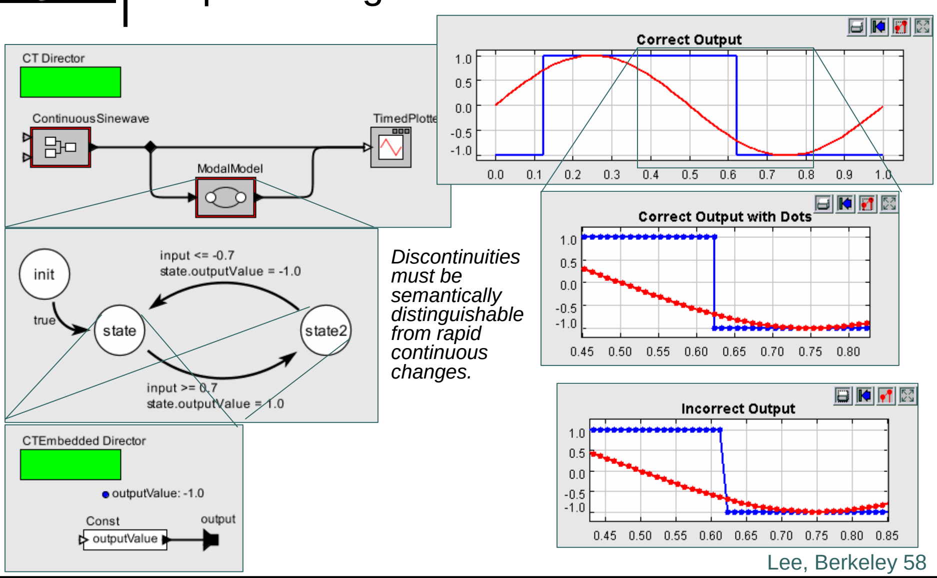
$$\dot{p}\colon \mathbb{R}_+ \to \mathbb{R}^n$$

$$\ddot{p}\colon \mathbb{R}_+ \to \mathbb{R}^n$$



Given this constraint, choosing $T \subset \mathbb{R}$ as the domain of these functions is an unfortunate choice. It makes it impossible to unambiguously represent discontinuities.

Discontinuities Are Not Just Rapid Changes



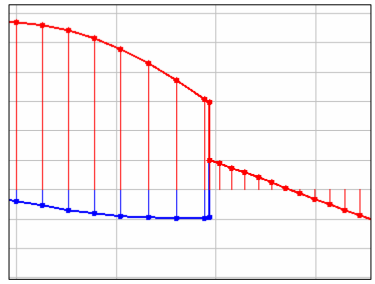
Discontinuities must be semantically distinguishable from rapid continuous changes.

Solution is the Same: Superdense Time

$$p\colon \mathbb{R}_+ \times \mathbb{N} \to \mathbb{R}^n$$

$$\dot{p}\colon \mathbb{R}_+ \times \mathbb{N} \to \mathbb{R}^n$$

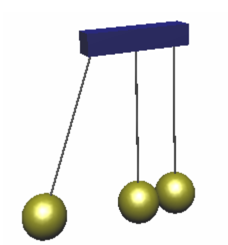
$$\ddot{p}\colon \mathbb{R}_+ \times \mathbb{N} \to \mathbb{R}^n$$



This makes it quite easy to construct models that combine continuous dynamics with discrete dynamics.

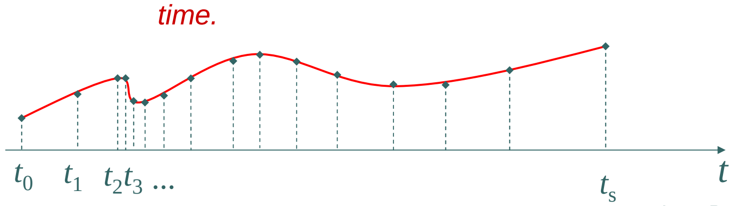
Ideal Solver Semantics for Continuous-Time Systems

[Liu and Lee, HSCC 2003]



In the *ideal solver semantics,* an ODE governing the hybrid system has a unique solution for intervals $[t_i, t_{i+1})$, the interval between discrete time points. A discrete trace loses nothing by not representing values within these intervals.

This elaborates our DE models only by requiring that an ODE solver be consulted when advancing time.



Recall Subtle Difference Between SR and DE. CT is more like SR.

- In SR, every actor is fired at every tick of its clock, as determined by a clock calculus and/or structured subclocks.
- In DE, an actor is fired at a tag only if it has input events at that tag or it has previously posted an event on the event queue with that tag.
- In CT, every actor is fired at every tick of the clock, as determined by an ODE solver.

In CT semantics, a signal has a value at every tag. But the solver to chooses to explicitly represent those values only at certain tags.

Key Contribution

- *With a common underlying fixed-point semantics, SR, DE, and CT can be (almost) arbitrarily combined hierarchically!*
- The only constraint is that it makes little sense to have SR at the top level if DE and CT are inside (because SR will either not advance time or will only advance it by fixed intervals).

Conclusions

A constructive fixed point semantics for synchronous/reactive models generalizes naturally to discrete-event and continuous-time models, enabling (almost) arbitrary combinations of the three modeling styles.

Further Reading

[1] E. A. Lee and H. Zheng, "Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems," in EMSOFT Salzburg, Austria: ACM, 2007.

[2] X. Liu and E. A. Lee, "CPO Semantics of Timed Interactive Actor Networks," UC Berkeley, Berkeley, CA, Technical Report EECS-2006-67, May 18 2006.

[3] X. Liu, E. Matsikoudis, and E. A. Lee, "Modeling Timed Concurrent Systems," in CONCUR 2006 - Concurrency Theory, Bonn, Germany, 2006.

[4] A. Cataldo, E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng, "A Constructive Fixed-Point Theorem and the Feedback Semantics of Timed Systems," in Workshop on Discrete Event Systems (WODES), Ann Arbor, Michigan, 2006.

[5] E. A. Lee, "Modeling Concurrent Real-time Processes Using Discrete Events," Annals of Software Engineering, vol. 7, pp. 25-45, March 4th 1998 1999.

[6] E. A. Lee, H. Zheng, and Y. Zhou, "Causality Interfaces and Compositional Causality Analysis," in *Foundations of Interface Technologies (FIT), Satellite to CONCUR*, San Francisco, CA, 2005.