EE 144/244: Fundamental Algorithms for System Modeling, Analysis, and Optimization
Fall 2016

Discrete Event Simulation

Stavros Tripakis
University of California, Berkeley

Timed Systems

In circuits, as well as in embedded / cyber-physical systems, timing is key:

- proper timing = an issue of **correctness**
- the right values, at the right time (not too late, not too early)
- c.f. **real-time control**.

Contrast this to personal computers: "best-effort" systems – timing an issue of performance.

(Timed) Discrete-Event (DE) Systems vs. Continuous Control Systems

Continuous control systems:

- Coming from continuous system theory.
- Typically implemented by periodic sampling controllers (but sometimes also event-driven controllers): these are discrete, but try to approximate the continuous ones.

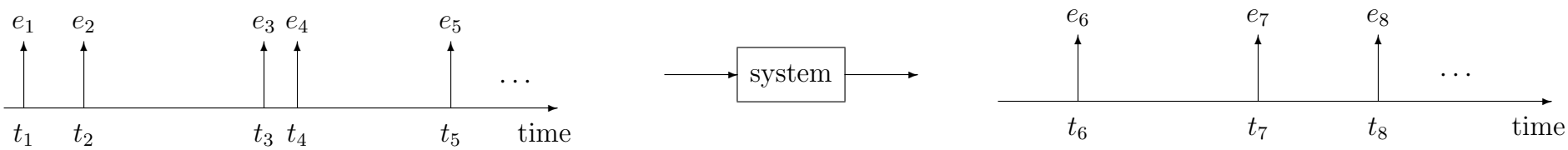
Discrete-event systems:

- More "sparse" events (typically).
- Discrete control: e.g., mode switches.
- Typically higher level: e.g., supervisory control (DE) vs. cruise control (continuous).
- Other application domains:
  - Queueing theory.
  - Circuits (VHDL, Verilog, SystemC).

(Timed) Discrete Events

Event:

- something occurring at some point in time
- may also carry a value
- event = (timestamp, value)
- discrete-event systems: consumers/producers of event streams



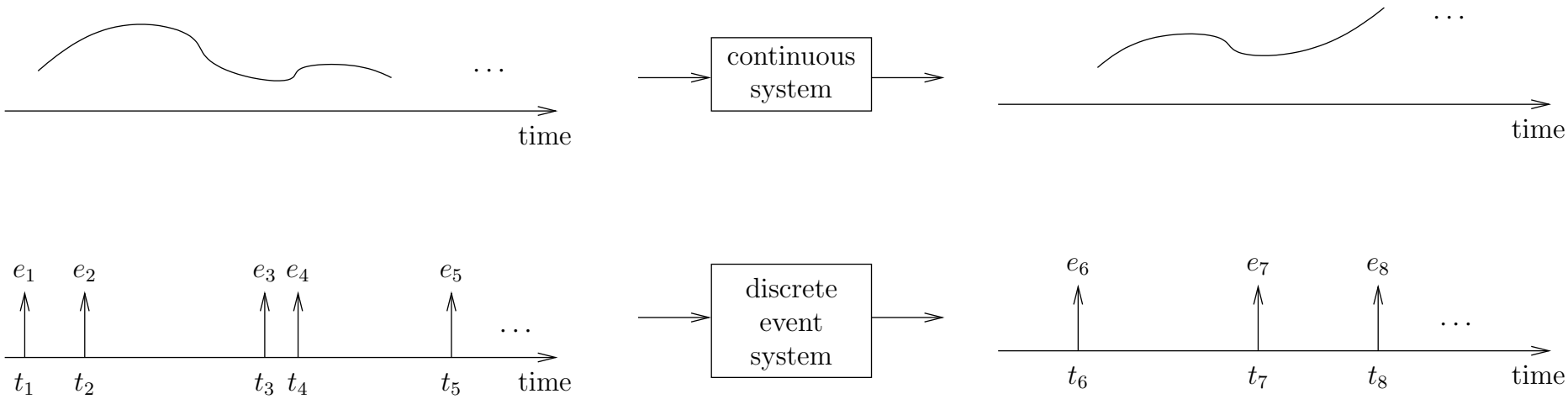
Continuous vs. Discrete Event Systems

- Continuous systems: functions on **continuous signals**.
  Continuous signal x = continuous function of dense time ($\mathbb{R}_+$)

$$x : \mathbb{R}_+ \to V$$

  $x(t)$: value of x at time t; belongs to some set of values V (e.g., $\mathbb{R}$)

- Timed Discrete Event Systems: deal with **timed discrete-event signals**.
  Timed discrete-event signal: sequence of timed events.



DIGRESSION:

CONTINUOUS vs. DISCRETE SIGNALS

Continuous vs. discrete signals

Intuition:

Discrete: only finite number of events can happen in a finite amount of time.

Continuous: not discrete.

Continuous vs. discrete signals

Intuition:

Discrete: only finite number of events can happen in a finite amount of time.

Continuous: not discrete.

Let's try to formalize this intuition.

Continuous vs. discrete signals

Let $\mathbb{R}_+$ be the set of non-negative reals, modeling time.

Let V be a set of values.

A **signal** can be defined as a set of events:

$$s \subseteq \mathbb{R}_+ \times V$$

(this is the *tagged signal model* [Lee and Sangiovanni-Vincentelli(1998)]).

Continuous vs. discrete signals

Let $\mathbb{R}_+$ be the set of non-negative reals, modeling time.

Let V be a set of values.

A **signal** can be defined as a set of events:

$$s \subseteq \mathbb{R}_+ \times V$$

(this is the *tagged signal model* [Lee and Sangiovanni-Vincentelli(1998)]).

Let $T(s)$ be the set of all timestamps in signal s:

$$T(s) = \{t \mid \exists (t, v) \in s\}$$

Continuous vs. discrete signals

Let $\mathbb{R}_+$ be the set of non-negative reals, modeling time.

Let V be a set of values.

A **signal** can be defined as a set of events:

$$s \subseteq \mathbb{R}_+ \times V$$

(this is the *tagged signal model* [Lee and Sangiovanni-Vincentelli(1998)]).

Let $T(s)$ be the set of all timestamps in signal s:

$$T(s) = \{t \mid \exists (t, v) \in s\}$$

Then we can define:

- s is **discrete** if $T(s)$ is **order-isomorphic to a subset of** $\mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, ...\}$ is the set of natural numbers.
- Otherwise, s is **continuous**.

Order-isomorphisms

A order-isomorphic B means there is an order-preserving bijection $f : A \to B$.

In our case the order is the usual $<$ on $\mathbb{R}_+$ and $\mathbb{N}$. So:

- f must be a **bijection**: (1) $f(a)$ must be defined for all $a \in A$, (2) for all $b \in B$ there must exist $a \in A$ such that $f(a) = b$, and (3) $a \neq a' \Rightarrow f(a) \neq f(a')$.
- f must be **order-preserving**: $a < a' \Rightarrow f(a) < f(a')$.

Order-isomorphisms

A order-isomorphic B means there is an order-preserving bijection $f : A \to B$.

In our case the order is the usual $<$ on $\mathbb{R}_+$ and $\mathbb{N}$. So:

- f must be a **bijection**: (1) $f(a)$ must be defined for all $a \in A$, (2) for all $b \in B$ there must exist $a \in A$ such that $f(a) = b$, and (3) $a \neq a' \Rightarrow f(a) \neq f(a')$.
- f must be **order-preserving**: $a < a' \Rightarrow f(a) < f(a')$.

Is $\mathbb{N} \times \mathbb{N}$ (with lexicographic order) order-isomorphic to $\mathbb{N}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$? Discrete.

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$? Discrete.

$s_5 = \{\}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$? Discrete.

$s_5 = \{\}$? Discrete.

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$? Discrete.

$s_5 = \{\}$? Discrete.

$s_6 = \{(t, t) \mid t \in [0, 1]\}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$? Discrete.

$s_5 = \{\}$? Discrete.

$s_6 = \{(t, t) \mid t \in [0, 1]\}$? Continuous.

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$? Discrete.

$s_5 = \{\}$? Discrete.

$s_6 = \{(t, t) \mid t \in [0, 1]\}$? Continuous.

$s_7 = \{(t, \lfloor t \rfloor) \mid t \in \mathbb{R}_+\}$?

Continous and discrete signals

Are the signals below discrete or continuous?

$s_1 = \{(t, t) \mid t \in \mathbb{R}_+\}$? Continuous.

$s_2 = \{(n, v) \mid n \in \mathbb{N}, v \in V\}$? Discrete.

$s_3 = \{(0, 0), (1, 1,), (2, 2)\}$? Discrete.

$s_4 = \{(0.3, 0), (1.27, 1,), (2\pi, 2)\}$? Discrete.

$s_5 = \{\}$? Discrete.

$s_6 = \{(t, t) \mid t \in [0, 1]\}$? Continuous.

$s_7 = \{(t, \lfloor t \rfloor) \mid t \in \mathbb{R}_+\}$? Continuous, according to our definition. But it is also **piecewise constant**, and could therefore be considered as essentially discrete. This is how discrete signals are modeled in Simulink.

Continuous vs. discrete signals

Our definition of discrete vs. continuous signals:

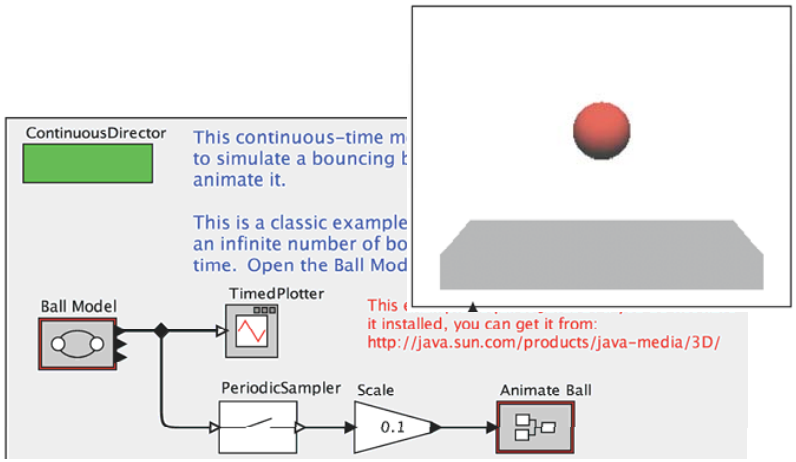
- s is **discrete** if $T(s)$ is *order-isomorphic* to a subset of $\mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, ...\}$ is the set of natural numbers.
- Otherwise, s is **continuous**.

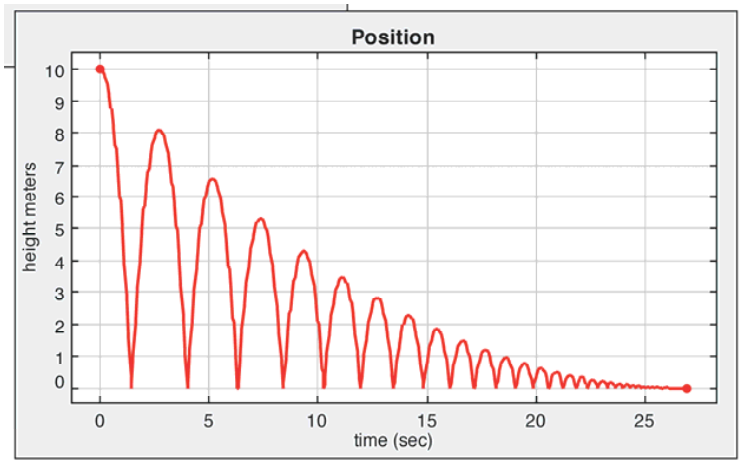
Recall our intuition:

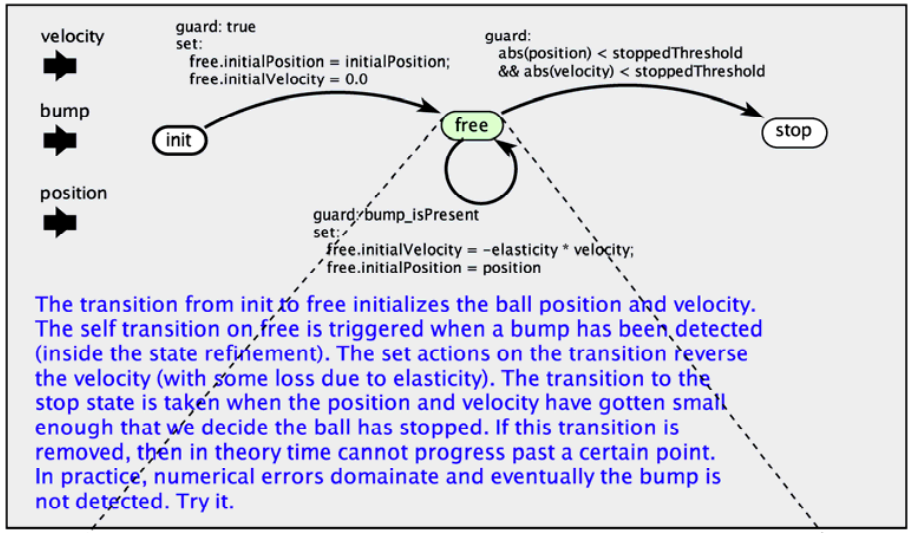
- Discrete: only finite number of events can happen in a finite amount of time.
- Continuous: not discrete.

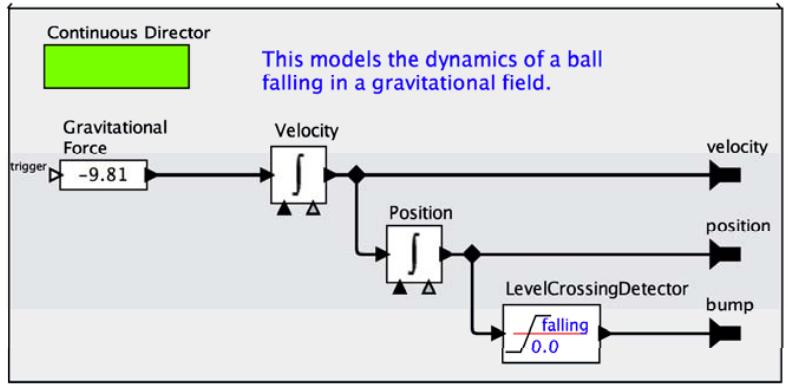
Does the definition capture our original intuition?

Example: bouncing ball





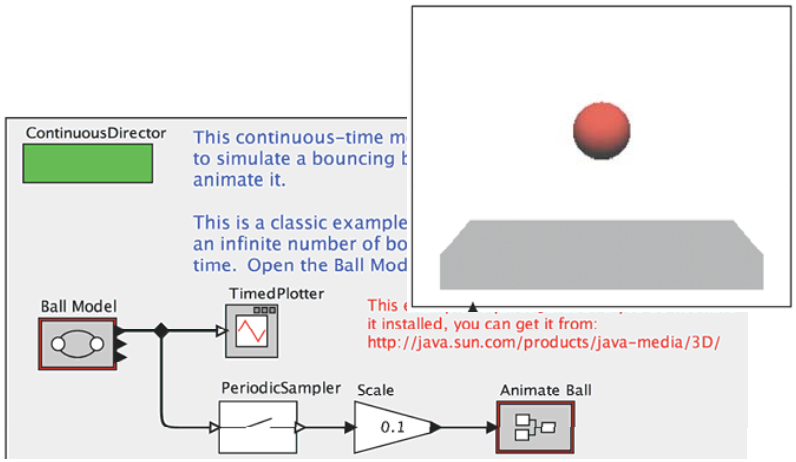


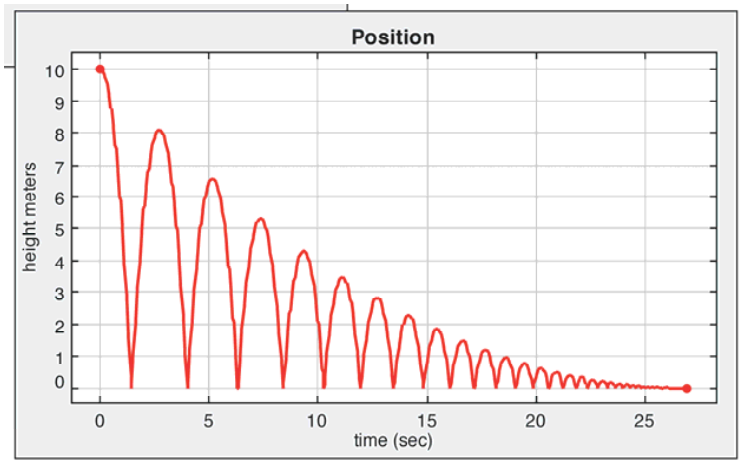


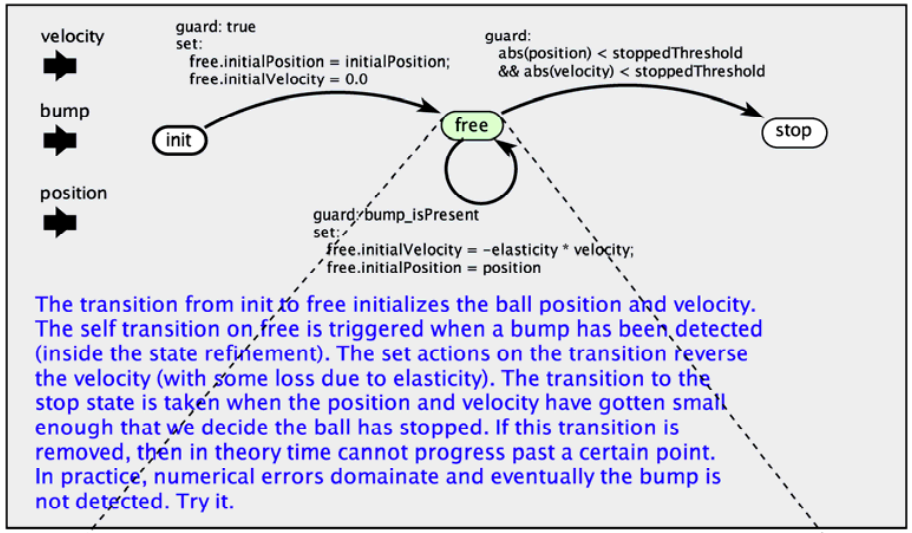
Jie Liu and Edward A. Lee

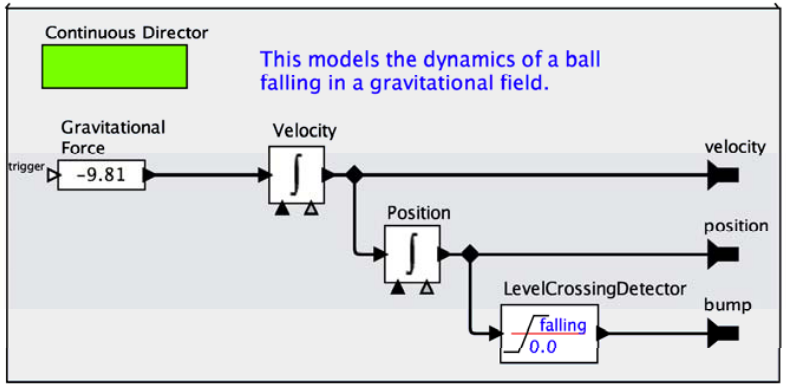
Example: bouncing ball

Zeno system: infinite # of discrete events in a finite amount of time => time blocked.









Jie Liu and Edward A. Lee

Zeno

Zeno's "Achilles and the tortoise" paradox:

- Achilles and the tortoise enter a race. Achilles runs of course much faster. He graciously allows the tortoise a head start of 1 meter. Who will win?

Zeno

Zeno's "Achilles and the tortoise" paradox:

- Achilles and the tortoise enter a race. Achilles runs of course much faster. He graciously allows the tortoise a head start of 1 meter. Who will win?

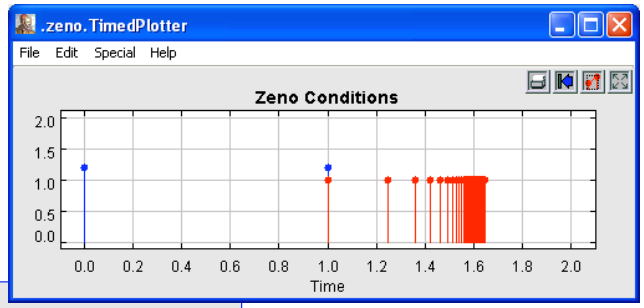
  "In a race, the quickest runner can never overtake the slowest, since the pursuer must first reach the point whence the pursued started, so that the slower must always hold a lead."

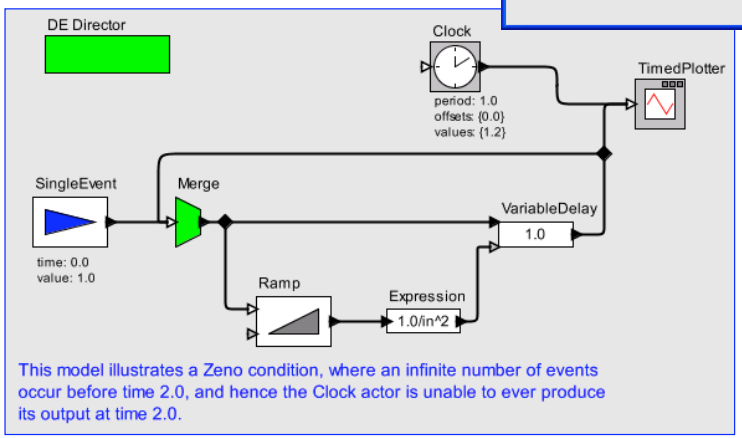
(from wikipedia)

Zeno DE systems

A DE system is zeno if it generates an infinite number of events in a finite amount of time:

Zeno Signals





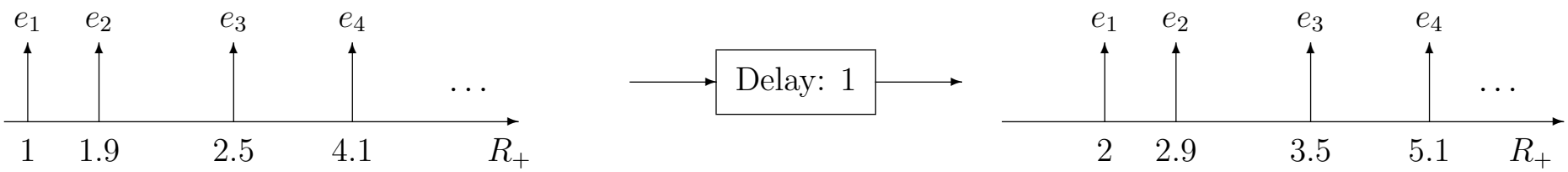
Eventually, execution stops advancing time. Why?

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.

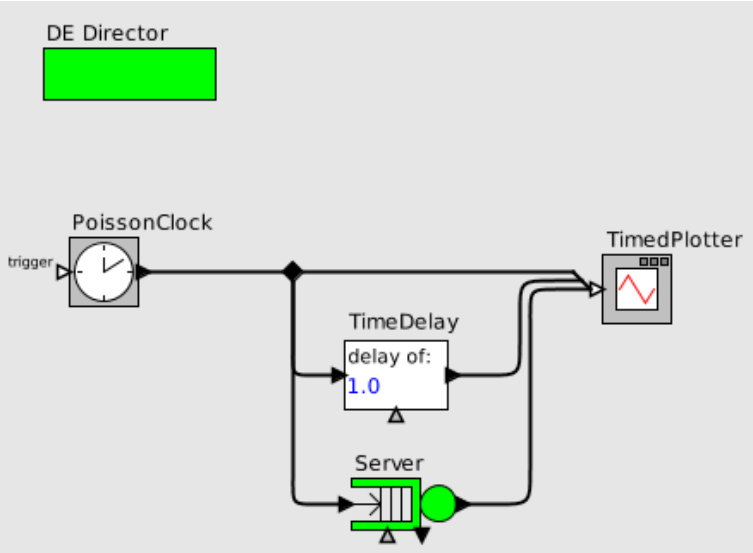
Zeno systems are sometimes useful (c.f., bouncing ball) but often an error of modeling. We will see how to avoid it in DE simulation, to avoid blocking time **globally**.

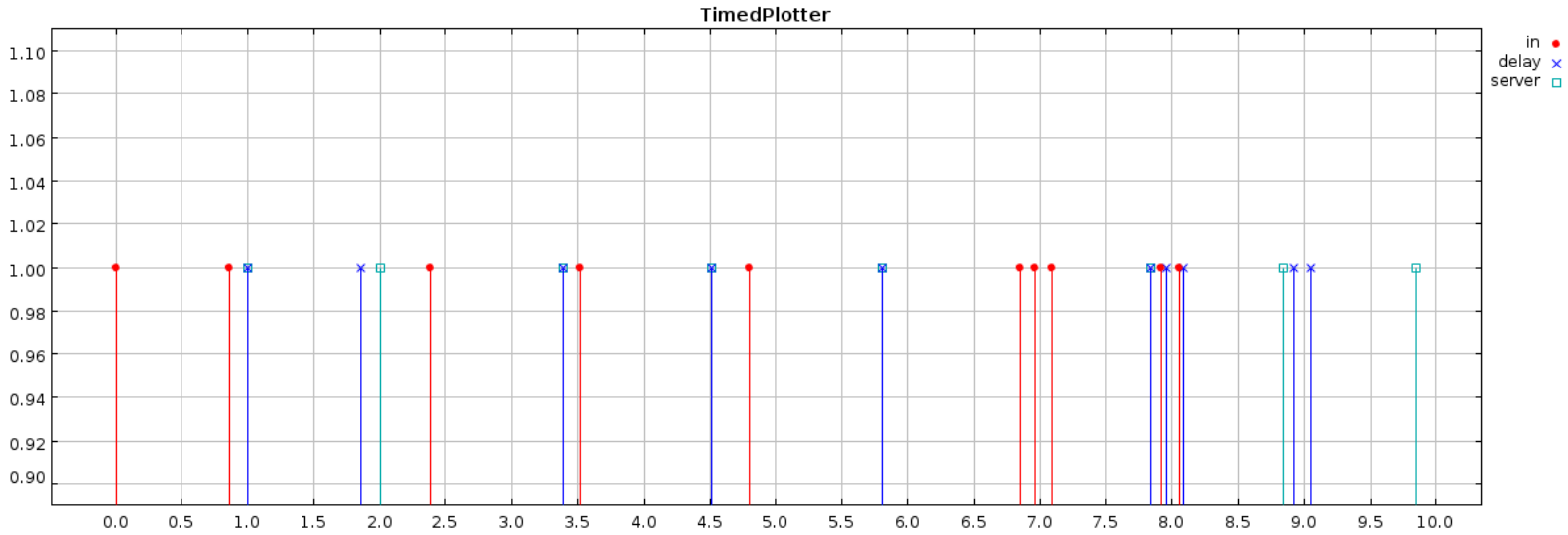
END DIGRESSION

Example Discrete-Event System: Dense-Time Delay



Delay vs. Server





DIGRESSION:

EVENTS vs. STATES

From States to Events

If my formalism only has the notion of state, can I define events?

From States to Events

If my formalism only has the notion of state, can I define events?

Event = change of state

From States to Events

If my formalism only has the notion of state, can I define events?

Event = change of state

Example: Lustre program

```
node UpwardEdge (X : bool) returns (E : bool);
let
  E = false -> X and not pre X ;
tel
```

From Events to States

If my formalism only has events as primitives, can I define state?

From Events to States

If my formalism only has events as primitives, can I define state?

State = history of events observed so far

From Events to States

If my formalism only has events as primitives, can I define state?

State = history of events observed so far

Formally:

- Σ: set of events
- Σ^*: set of finite event sequences = histories
- Every $s \in \Sigma^*$ can be seen as a state

From Events to States

If my formalism only has events as primitives, can I define state?

State = history of events observed so far

Formally:

- Σ: set of events
- Σ^*: set of finite event sequences = histories
- Every $s \in \Sigma^*$ can be seen as a state

C.f. a famous theorem:

Theorem (Myhill-Nerode theorem)

A language $L \subseteq \Sigma^*$ is regular iff the equivalence relation over words

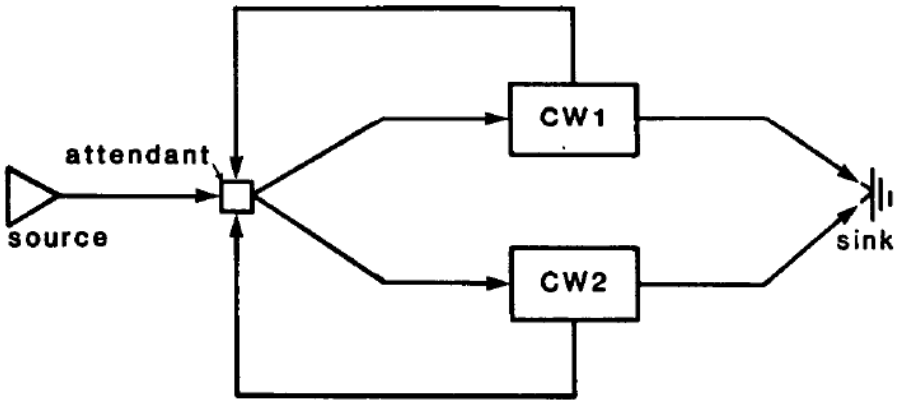
$$s \sim_L s' \quad \widehat{=} \quad \forall s'' \in \Sigma^* : s \cdot s'' \in L \Leftrightarrow s' \cdot s'' \in L$$

has a finite set of equivalence classes. The number of equivalence classes of $\sim_L$ is the number of states in the smallest DFA recognizing L.

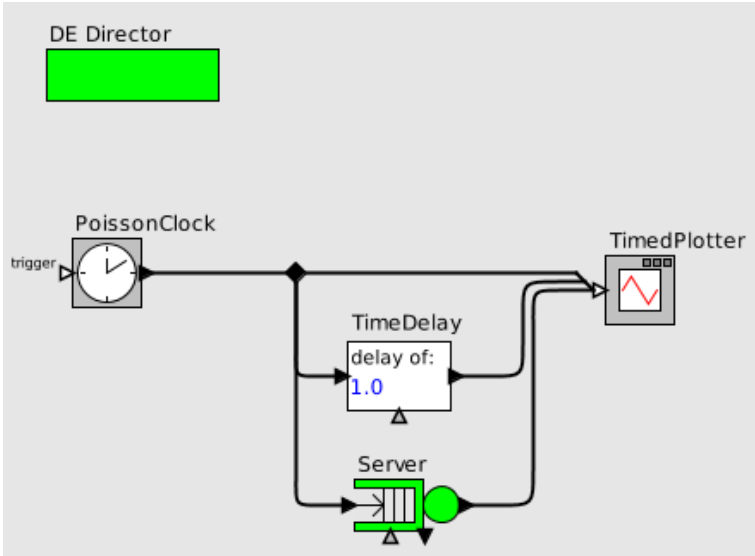
END DIGRESSION

Discrete-Event Models (DE)

Networks of *actors* such as Delay, Server, sources, sinks, ...



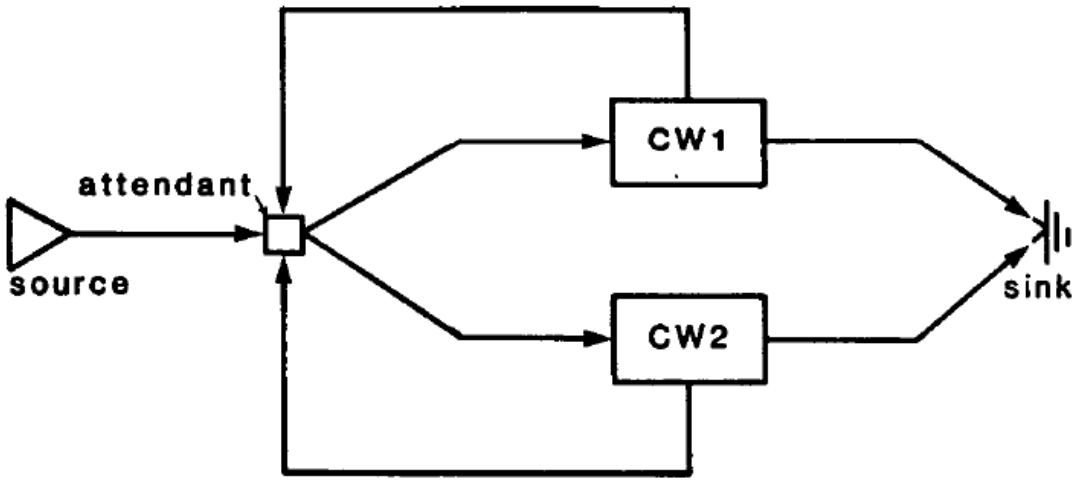
model 1 (drawing)



model 2 (ptolemy)

Example: car wash

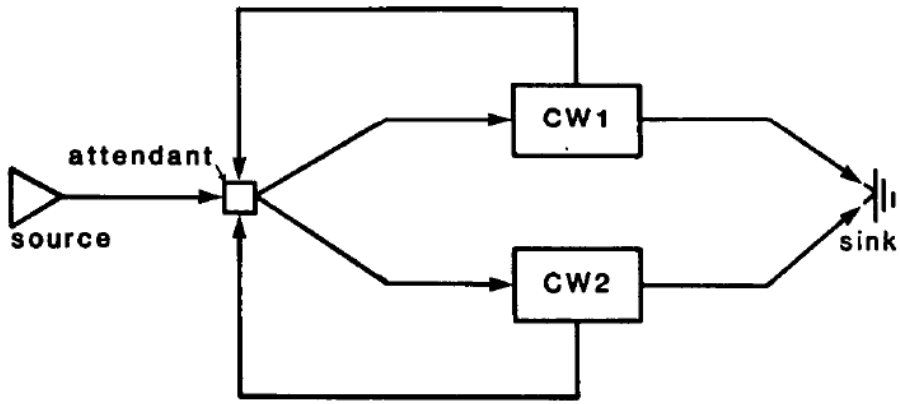
Taken from [Misra(1986)]:



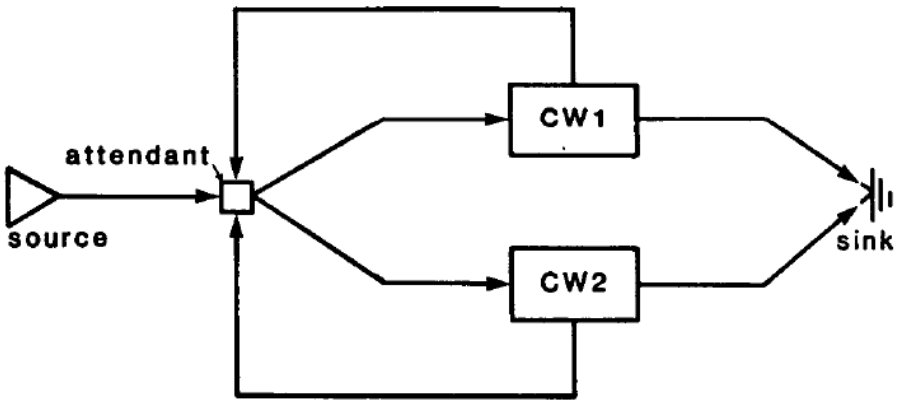
- Source generates car arrivals at some arbitrary times.
- Attendant directs cars to car wash stations CW1 or CW2:
  - if both CW1 and CW2 are free, then to CW1;
  - if only one is free, then to this free one;
  - otherwise car waits until a station becomes free.
  - Cars are served by attendant in FIFO order.
- CW1 spends 8 mins to wash a car.
- CW2 spends 10 mins to wash a car.

Delay vs. Server

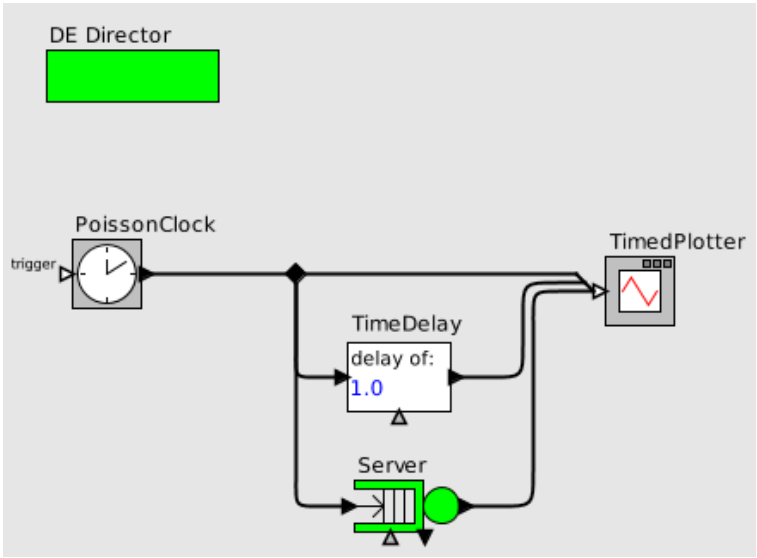
Are CW1, CW2 delays or servers?



Analysis of DE Models



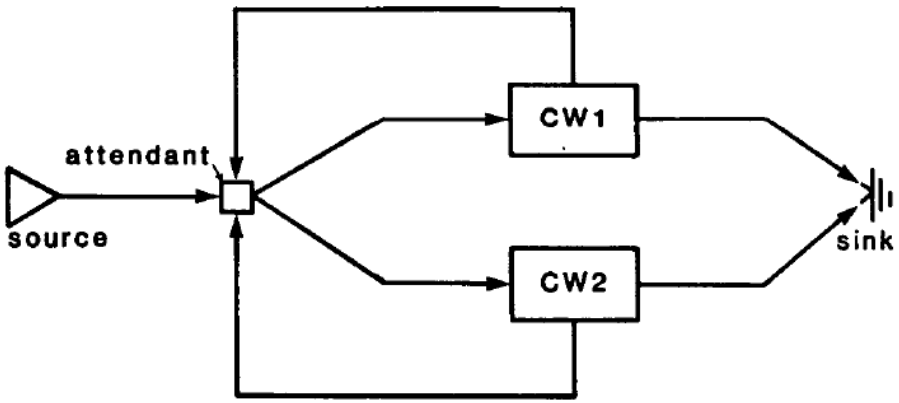
model 1 (drawing)



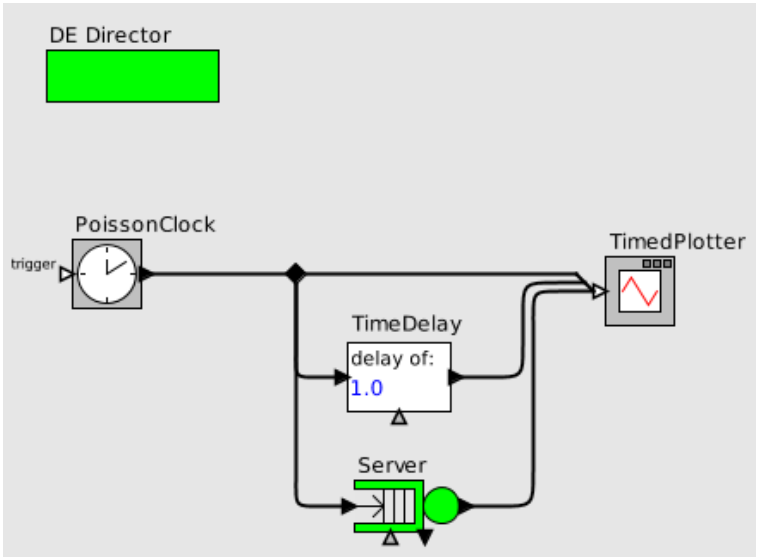
model 2 (ptolemy)

- We will look at **simulation** of DE models.
- Exhaustive verification (model-checking) of DE models: not well studied (see [Stergiou et al. 2013])

Analysis of DE Models



model 1 (drawing)



model 2 (ptolemy)

- We will look at **simulation** of DE models.
- Exhaustive verification (model-checking) of DE models: not well studied (see [Stergiou et al. 2013])
- Well studied problem: exhaustive verification of another type of DE system: **timed automata** [Alur and Dill(1994)]

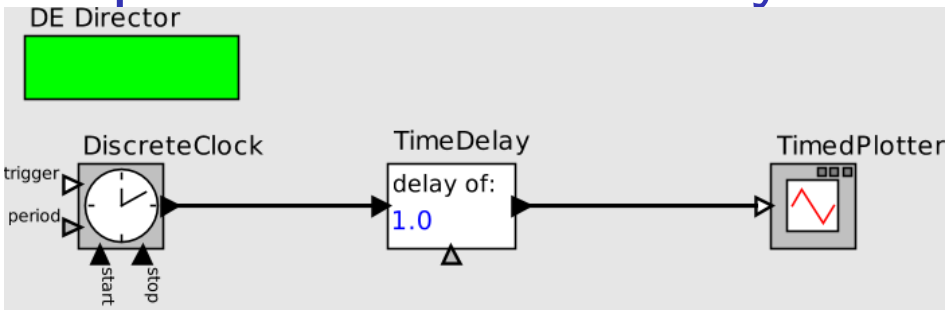
DISCRETE-EVENT SIMULATION

Discrete-Event Simulation: Basic Idea

Standard DE simulation scheme:

1: $t := 0$; *// initialize simulation time to 0*
2: initialize global event queue Q with a set of initial events;
// events in Q ordered by timestamp
3: **while** Q is not empty **do**
4: remove earliest event $e = (v_e, t_e)$ from Q;
5: $t := t_e$; *// advance global time*
6: "execute" event e: update system state, generate possible future events, and add them to Q, ordered by timestamps;
7: **end while**

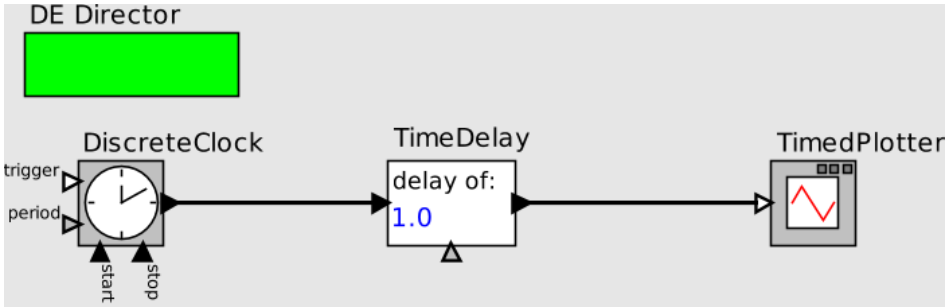
Example: Clock and Delay



Clock period: 0.6
c_i: events generated by Clock
d_i: events generated by Delay

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
3: **while** Q is not empty **do**
4: remove earliest event $e = (v_e, t_e)$ from Q;
5: $t := t_e$;
6: "execute" event e;
7: **end while**

Example: Clock and Delay



Clock period: 0.6
c_i: events generated by Clock
d_i: events generated by Delay

point in algo	t	Q	current event e
after initialization (step 2)	0	$[(c_0, 0), (c_1, 0.6), (c_2, 1.2), ...]$	
after step 4		$[(c_1, 0.6), (c_2, 1.2), ...]$	$(c_0, 0)$
after step 5		$[(c_1, 0.6), (d_0, 1.0), (c_2, 1.2), ...]$	
after step 6	0		
after step 4		$[(d_0, 1.0), (c_2, 1.2), ...]$	$(c_1, 0.6)$
after step 5		$[(d_0, 1.0), (c_2, 1.2), (d_1, 1.6), ...]$	
after step 6	0.6		
after step 4		$[(c_2, 1.2), (d_1, 1.6), ...]$	$(d_0, 1.0)$
after step 5		Q does not change, but something gets printed	
after step 6	1.0		
...			

Discrete-Event Simulation: Issues

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
3: **while** Q is not empty **do**
4: remove earliest event $e = (v_e, t_e)$ from Q;
5: $t := t_e$;
6: "execute" event e: update system state, generate possible future events, and add them to Q, ordered by timestamps;
7: **end while**

- Appears intuitive, but details are left unspecified: steps 2, 6.
- *Not modular* : step 6 appears to work on the entire system state, not on individual *actors*.
- How to make such a scheme completely modular is an active topic of research (we will come back to this).

Discrete-Event Simulation: Issues

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
3: **while** Q is not empty **do**
4: remove earliest event $e = (v_e, t_e)$ from Q;
5: $t := t_e$;
6: "execute" event e: update system state, generate possible future events, and add them to Q, ordered by timestamps;
7: **end while**

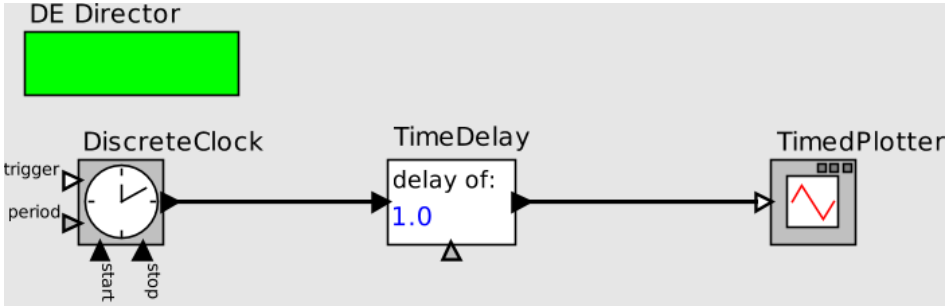
- Appears intuitive, but details are left unspecified: steps 2, 6.
- *Not modular* : step 6 appears to work on the entire system state, not on individual *actors*.
- How to make such a scheme completely modular is an active topic of research (we will come back to this).

Let's try to flesh out the details of steps 2 and 6.

Modeling Source Actors

Source actor = an actor with no inputs.

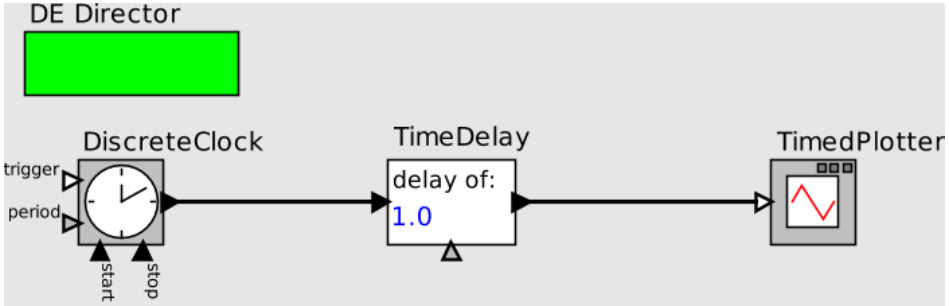
Clock is a source:



Modeling Source Actors

Source actor = an actor with no inputs.

Clock is a source:

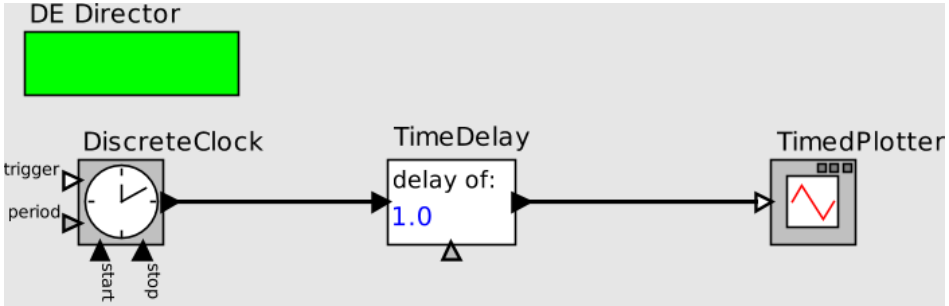


- Option 1 – sources generate all their events at initialization.
  - Simulation time is finite, so presumably only finite number of events.
  - But it may be very large.

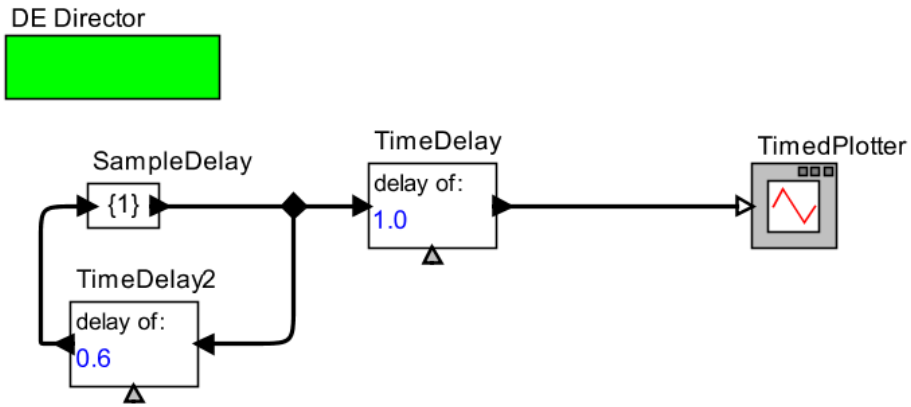
Modeling Source Actors

Source actor = an actor with no inputs.

Clock is a source:

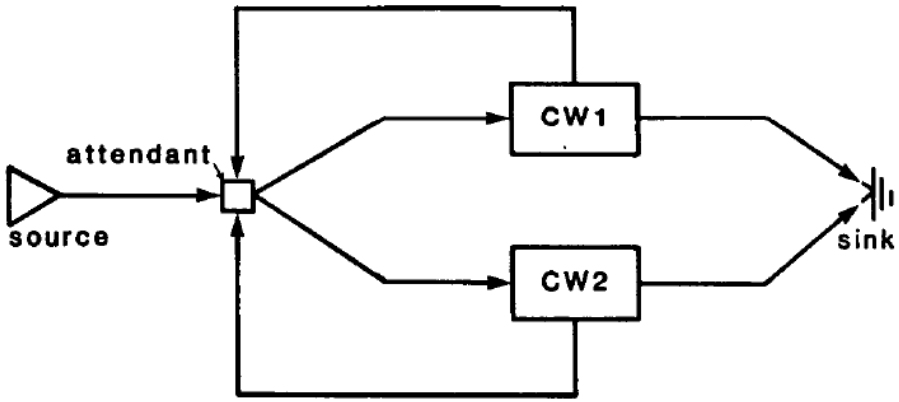


- Option 1 – sources generate all their events at initialization.
  - Simulation time is finite, so presumably only finite number of events.
  - But it may be very large.
- Option 2 – model sources using feedback loops with initial events:



Feedback loops are necessary in general

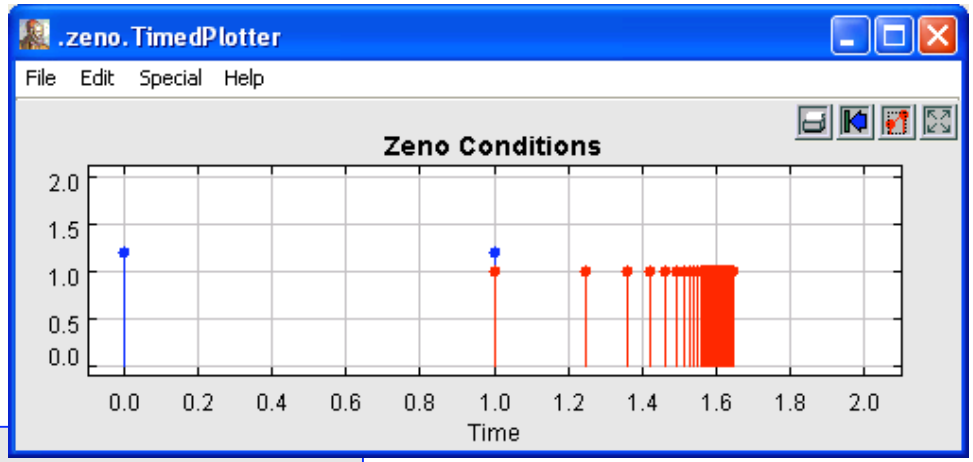
Example: car wash (taken from [Misra(1986)]):

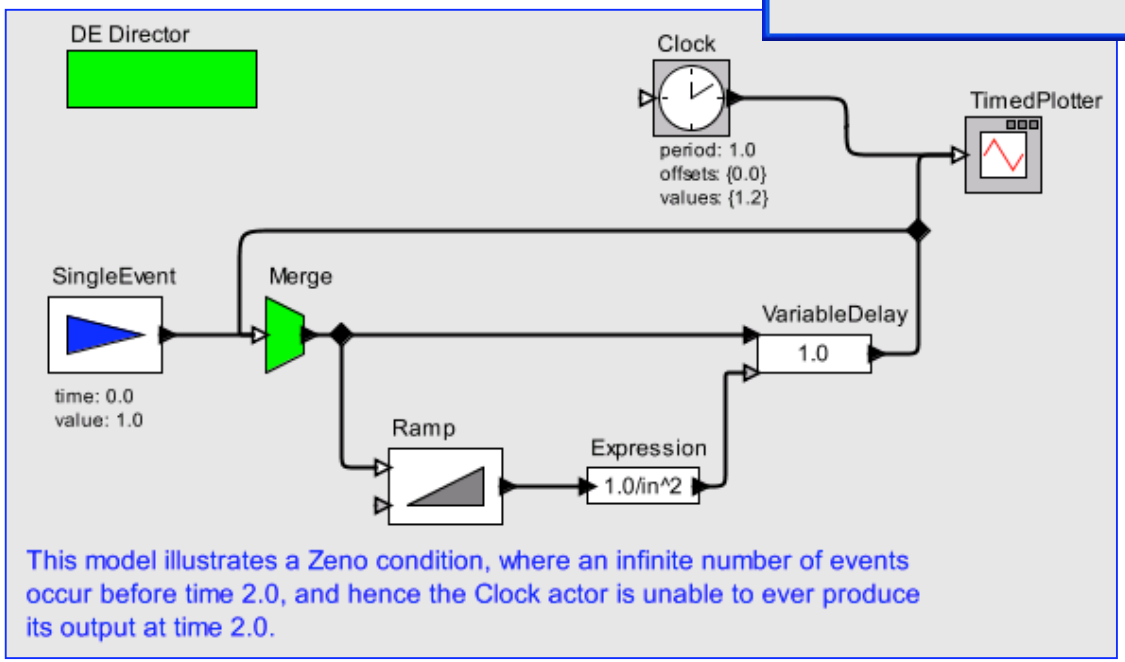


- Source generates car arrivals at some arbitrary times (e.g., at times 3, 8, 9, 14, 16, 22)
- Attendant directs cars to car wash stations CW1 or CW2:
  - if both CW1 and CW2 are free, then to CW1
  - if only one is free, then to this free one
  - otherwise car waits until a station becomes free
  - cars are served by attendant in FIFO order
- CW1 (a server actor) spends 8 mins to wash a car
- CW2 (a server actor) spends 10 mins to wash a car

But feedback loops can also be dangerous ...

Zeno Signals



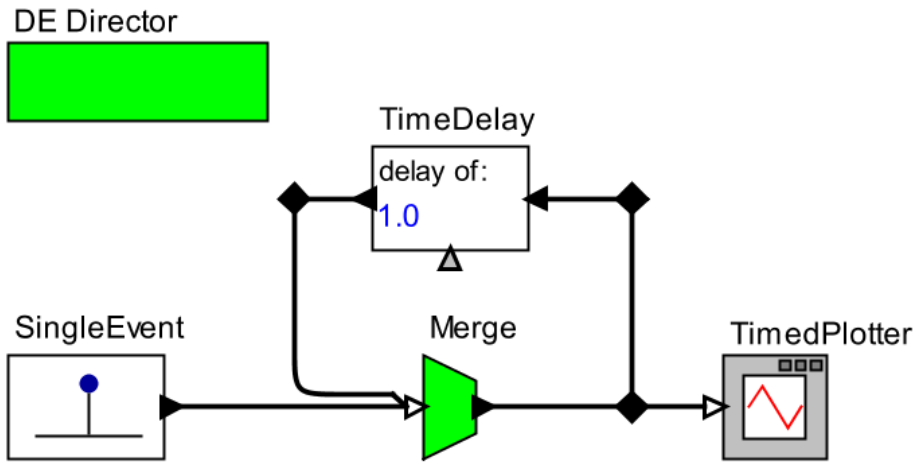


Eventually, execution stops advancing time. Why?

Note that if the Ramp is set to produce integer outputs, then eventually the output will overflow and become

Avoiding Zeno Systems

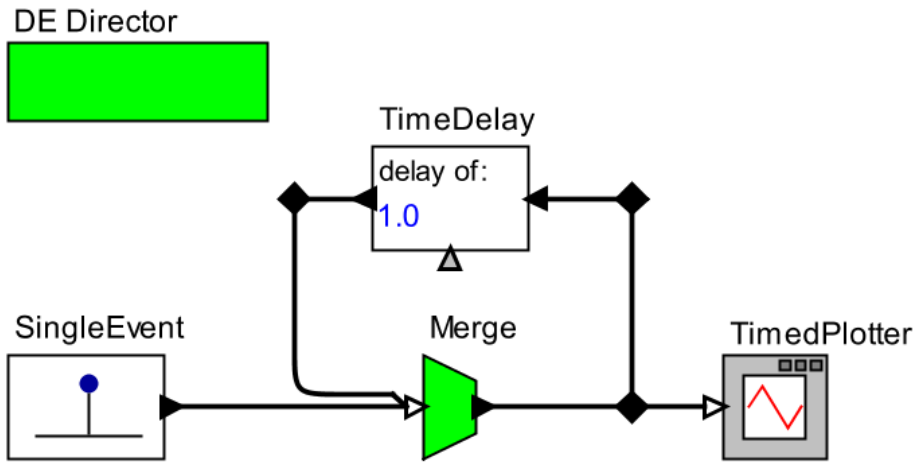
Suppose we add a constant (or at least bounded from below) non-zero delay in every feedback loop:



Is it sufficient to avoid zenoness?

Avoiding Zeno Systems

Suppose we add a constant (or at least bounded from below) non-zero delay in every feedback loop:

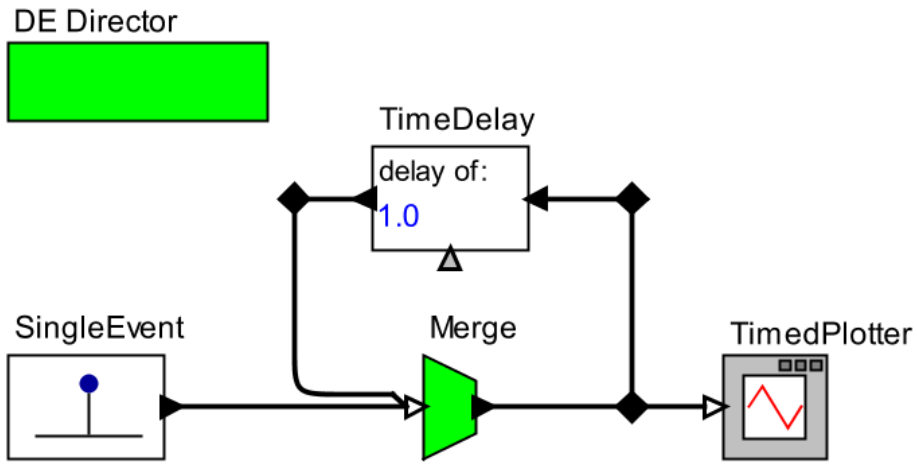


Is it sufficient to avoid zenoness?

Yes. **Exercise:** prove it.

Avoiding Zeno Systems

Suppose we add a constant (or at least bounded from below) non-zero delay in every feedback loop:



Is it sufficient to avoid zenoness?

Yes. **Exercise:** prove it.

From now on we assume a constant non-zero delay in every feedback loop.

Another Example: Alarm

- Alarm actor: produces event at given time t, unless it receives input at time $t' \leq t$.

Another Example: Alarm

- Alarm actor: produces event at given time t, unless it receives input at time $t' \leq t$.
- How does the DE simulation algorithm handle this example?

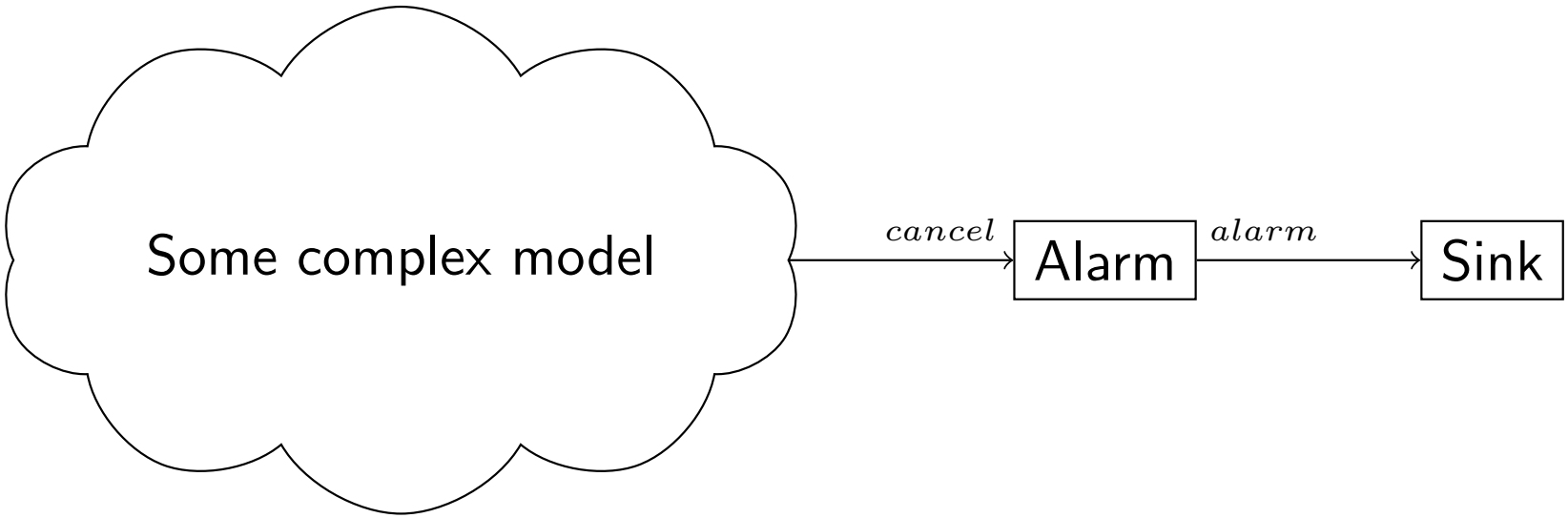
Another Example: Alarm

- Alarm actor: produces event at given time t, unless it receives input at time $t' \leq t$.
- How does the DE simulation algorithm handle this example?
- It appears that Alarm should post an initial event with time t

  ... but this event may then have to be **canceled** during simulation if something arrives at the input before t.
- Canceling events = removing them from the event queue.

Why insert events only to cancel them later?

Whether an event will be generated before t may not always be easy to determine:

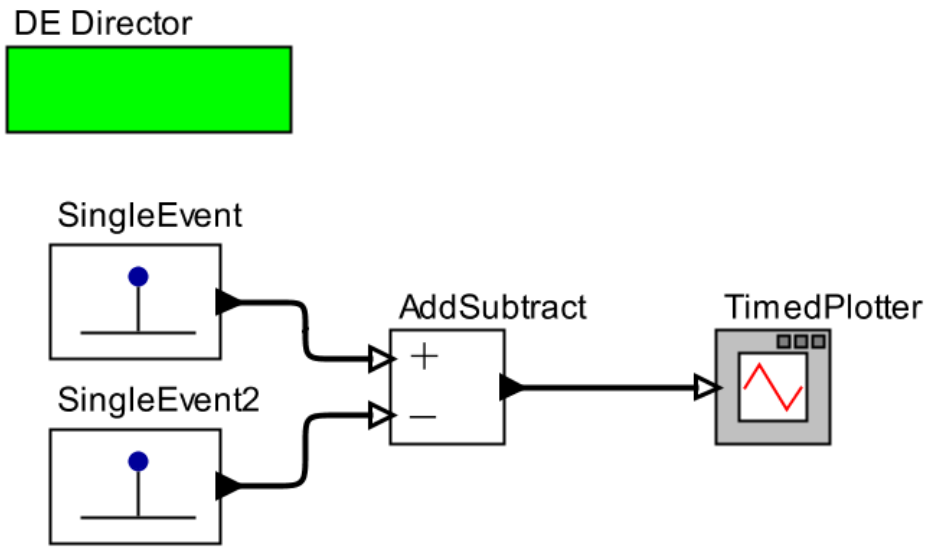


- DE algorithm must work independently of how Alarm is connected.
- That's what modular means.

Discrete-Event Simulation – version 2

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
3: **while** Q is not empty **do**
4: remove earliest event $e = (v_e, t_e)$ from Q;
5: $t := t_e$;
6: execute event e: update system state, generate possible future events, and add them to Q, ordered by timestamps; possibly remove events from Q;
7: **end while**

Another Issue: Simultaneous Events



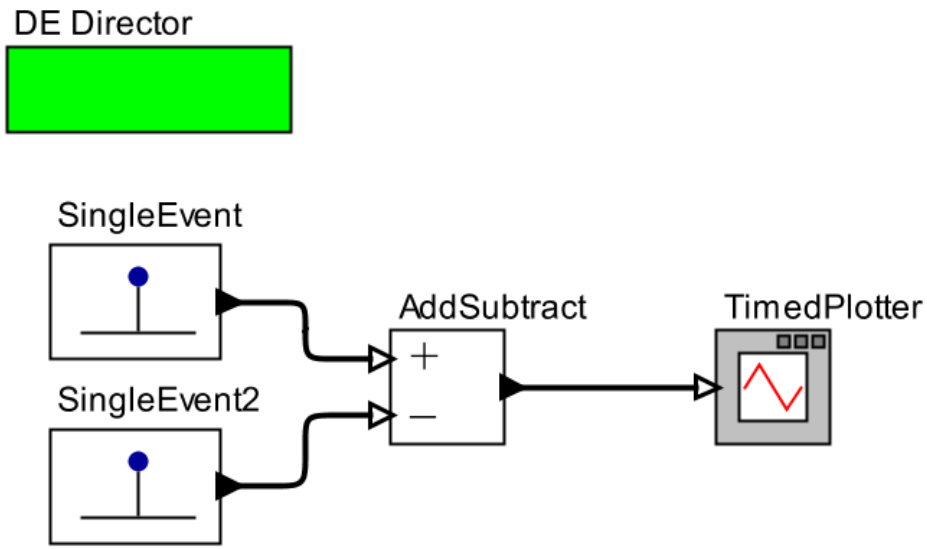
The *AddSubtract* actor is supposed to behave as follows:

- If it receives two simultaneous events, it adds/subtracts their values and produces a *single* event at its output with the resulting value.
- If it receives an event in just one of the two inputs, it simply forwards it.

Suppose the two *SingleEvent* actors produce two simultaneous events with the same value x.

What should the output be?

Another Issue: Simultaneous Events



The *AddSubtract* actor is supposed to behave as follows:

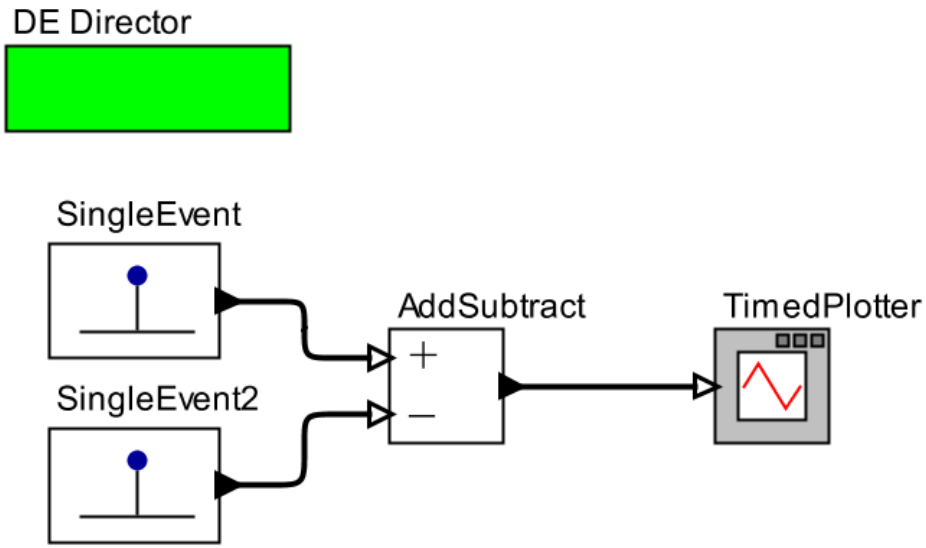
- If it receives two simultaneous events, it adds/subtracts their values and produces a *single* event at its output with the resulting value.
- If it receives an event in just one of the two inputs, it simply forwards it.

Suppose the two *SingleEvent* actors produce two simultaneous events with the same value x.

What should the output be? A **single** event with value $x - x = 0$.

Another Issue: Simultaneous Events

How to achieve the desired behavior with the DE simulation algorithm?



1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
3: **while** Q is not empty **do**
4: remove earliest event $e = (v_e, t_e)$ from Q;
5: $t := t_e$;
6: execute event e: update system state, generate possible future events, and add them to Q, ordered by timestamps; possibly remove events from Q;
7: **end while**

Discrete-Event Simulation – version 3

It appears that the DE simulation algorithm must execute **sets of simultaneous events**, instead of one event at a time:

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
3: **while** Q is not empty **do**
4: remove ~~earliest event $e = (v_e, t_e)$~~ set E of all (?) simultaneous earliest events from Q;
5: $t := t_e$;
6: execute ~~event e~~ set of events E: update system state, generate possible future events, and add them to Q, ordered by timestamps; possibly remove events from Q;
7: **end while**

Discrete-Event Simulation – version 3

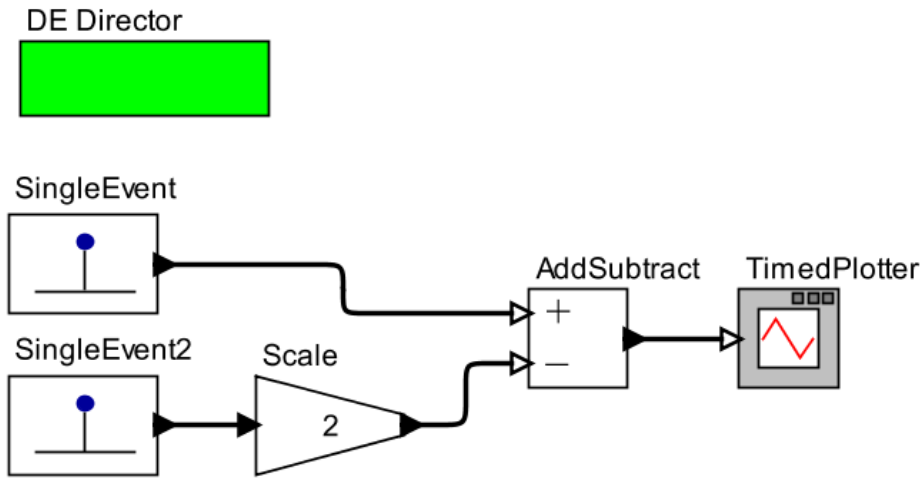
It appears that the DE simulation algorithm must execute **sets of simultaneous events**, instead of one event at a time:

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
3: **while** Q is not empty **do**
4: remove ~~earliest event $e = (v_e, t_e)$~~ set E of all (?) simultaneous earliest events from Q;
5: $t := t_e$;
6: execute ~~event e~~ set of events E: update system state, generate possible future events, and add them to Q, ordered by timestamps; possibly remove events from Q;
7: **end while**

Not as simple ...

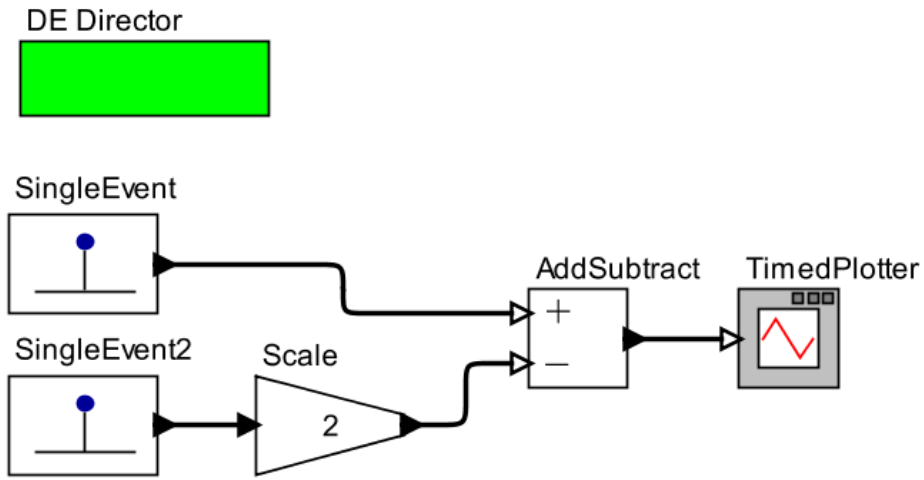
Issues with Simultaneous Events

Suppose the two SingleEvent sources produce two simultaneous events. Should these be processed together by the algorithm?



Issues with Simultaneous Events

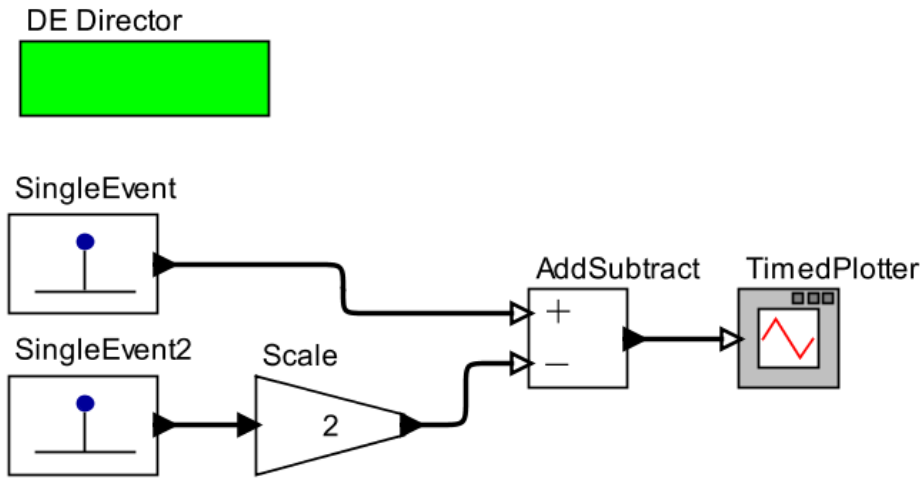
Suppose the two SingleEvent sources produce two simultaneous events. Should these be processed together by the algorithm?



No: AddSubtract needs to wait for the output of Scale.

Issues with Simultaneous Events

Suppose the two SingleEvent sources produce two simultaneous events. Should these be processed together by the algorithm?



No: AddSubtract needs to wait for the output of Scale.

Processing a set of simultaneous events E may result in new simultaneous events not in E ...

We need some systematic way to do this ...

Back to the Alarm Example

- Alarm actor: produces event at given time t, unless it receives input at time $t' \leq t$

Back to the Alarm Example

- Alarm actor: produces event at given time t, unless it receives input at time $t' \leq t$
- What if Source produces an event also at time t?

Back to the Alarm Example

- Alarm actor: produces event at given time t, unless it receives input at time $t' \leq t$
- What if Source produces an event also at time t?
- According to the semantics of Alarm, it should not raise an alarm event.
- Does the DE simulation algorithm guarantee this?

Back to the Alarm Example

1: $t := 0$;
2: initialize global event queue Q with $\{(alarm, t), (cancel, t)\}$;
3: **while** Q is not empty **do**
4: remove earliest event $e = (v_e, t_e)$ from Q;
5: $t := t_e$;
6: execute event e: update system state, generate possible future events, and add them to Q, ordered by timestamps; possibly remove events from Q;
7: **end while**

- Non-determinism!
  - Different results depending on which of the two instantaneous events $(alarm, t)$ and $(cancel, t)$ is first removed from Q.

Dealing with Simultaneous Events

- Chronological ordering (= ordering by timestamps) of events in the queue is not enough.
- Must also respect dependencies between simultaneous events
  - Alarm's output event at time t depends on Source's output event at time t
- How to define event dependencies?

Dealing with Simultaneous Events

- Chronological ordering (= ordering by timestamps) of events in the queue is not enough.
- Must also respect dependencies between simultaneous events
  - Alarm's output event at time t depends on Source's output event at time t
- How to define event dependencies?
- First let's formalize actor dependencies.

Dependency Relation among Actors

Let A_1, A_2 be two actors in the DE model.

Define the dependency relation $A_1 \to A_2$ (A_2 depends on A_1) as follows:

$A_1 \to A_2 \ \widehat{=} \ A_1$ is **zero-delay** (i.e., its output may have the same timestamp as the input that produced it) and there is a connection from an output of A_1 to an input of A_2.

Dependency Relation among Actors

Let A_1, A_2 be two actors in the DE model.

Define the dependency relation $A_1 \to A_2$ (A_2 depends on A_1) as follows:

$A_1 \to A_2 \ \widehat{=} \ A_1$ is **zero-delay** (i.e., its output may have the same timestamp as the input that produced it) and there is a connection from an output of A_1 to an input of A_2.

Claim: $\to$ is acyclic.

Why?

Dependency Relation among Actors

Let A_1, A_2 be two actors in the DE model.

Define the dependency relation $A_1 \to A_2$ (A_2 depends on A_1) as follows:

$A_1 \to A_2 \ \widehat{=}\ A_1$ is **zero-delay** (i.e., its output may have the same timestamp as the input that produced it) and there is a connection from an output of A_1 to an input of A_2.

Claim: $\to$ is acyclic.

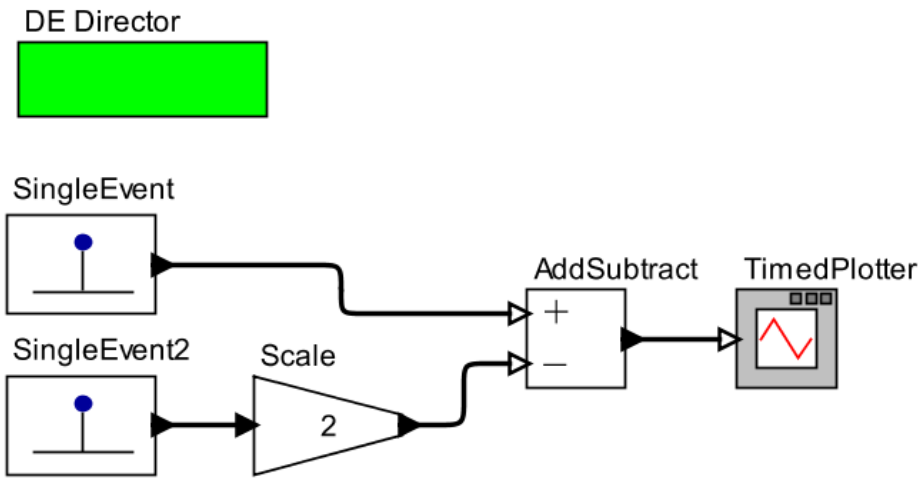
Why? Because every loop is assumed to have a non-zero-delay actor.

Actor Dependencies – Examples

Alarm $\rightarrow$ Sink

Actor Dependencies – Examples

Alarm $\rightarrow$ Sink



Scale $\rightarrow$ AddSubtract $\rightarrow$ TimedPlotter

Precedence Relation on Events

Let $e_1 = (v_1, t_1)$ and $e_2 = (v_2, t_2)$ be two events generated during DE simulation.

Let A_1 and A_2 be the recipient actors of e_1 and e_2:

- This information can be encoded in v_1, v_2.
- We assume a unique recipient per event.
  - No loss of generality: can view fan-out junctions as zero-delay actors which copy every input event to all their outputs.

Precedence Relation on Events

Let $e_1 = (v_1, t_1)$ and $e_2 = (v_2, t_2)$ be two events generated during DE simulation.

Let A_1 and A_2 be the recipient actors of e_1 and e_2:

- This information can be encoded in v_1, v_2.
- We assume a unique recipient per event.
  - No loss of generality: can view fan-out junctions as zero-delay actors which copy every input event to all their outputs.

We define precedence of events e_1, e_2:

$$e_1 \prec e_2 \quad \widehat{=} \quad t_1 < t_2 \text{ or } \Big(t_1 = t_2 \text{ and } A_1 \rightarrow^* A_2 \text{ and } A_1 \neq A_2 \Big)$$

where $\rightarrow^*$ is the transitive closure of $\rightarrow$.

Precedence Relation on Events

Let $e_1 = (v_1, t_1)$ and $e_2 = (v_2, t_2)$ be two events generated during DE simulation.

Let A_1 and A_2 be the recipient actors of e_1 and e_2:

- This information can be encoded in v_1, v_2.
- We assume a unique recipient per event.
  - No loss of generality: can view fan-out junctions as zero-delay actors which copy every input event to all their outputs.

We define precedence of events e_1, e_2:

$$e_1 \prec e_2 \quad \widehat{=} \quad t_1 < t_2 \text{ or } \Big(t_1 = t_2 \text{ and } A_1 \rightarrow^* A_2 \text{ and } A_1 \neq A_2 \Big)$$

where $\rightarrow^*$ is the transitive closure of $\rightarrow$.

Claim: $\prec$ is acyclic.

Event Precedences – Examples

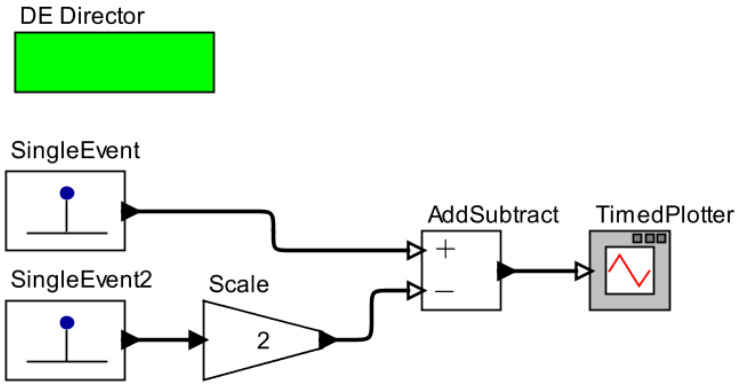
Assuming simultaneous events in the examples below:

Alarm $\rightarrow$ Sink, therefore $cancel \prec alarm$.

Event Precedences – Examples

Assuming simultaneous events in the examples below:

Alarm $\rightarrow$ Sink, therefore $cancel \prec alarm$.



Suppose there are 3 events, e_1, e_2, e_3, pending at the input port of Scale and the two input ports of AddSubtract, respectively. Then:

$$e_1 \prec e_2 \text{ and } e_1 \prec e_3.$$

e_2 and e_3 are independent.

Discrete-Event Simulation – final version

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
// Q is always implicitly ordered w.r.t. timestamps
// and among events with same timestamp
// w.r.t. event dependencies
3: **while** Q is not empty **do**
4: remove set E of all minimal events w.r.t. $\prec$ from Q;
// these are earliest and simultaneous events,
// which depend on no other events
5: $t := t_e$;
6: execute set of events E: update system state, generate possible future events, and add them to Q, ordered by timestamps; possibly remove events from Q;
7: **end while**

Discrete-Event Simulation – final version

1: $t := 0$;
2: initialize global event queue Q with a set of initial events;
// Q is always implicitly ordered w.r.t. timestamps
// and among events with same timestamp
// w.r.t. event dependencies
3: **while** Q is not empty **do**
4: remove set E of all minimal events w.r.t. $\prec$ from Q;
// these are earliest and simultaneous events,
// which depend on no other events
5: $t := t_e$;
6: execute set of events E: update system state, generate possible future events, and add them to Q, ordered by timestamps; possibly remove events from Q;
7: **end while**

Claim: any new event e produced in step 6 is guaranteed to be greater than all events in set E w.r.t. $\prec$. That is, either e has greater timestamp than all events in E, or it depends on some event in E.

DE Simulation and HDLs

- HDLs: Hardware Description Languages
- Verilog, VHDL, SystemC, ...
- Real-world languages
- EDA (Electronics Design Automation) industry: billions of $$$
- Simulation tools: based on DE simulation
- But note: many variants, details, ...
  - E.g., SystemC specification[1] is > 600 pages long.
  - Description of the simulation algorithm (in English) is 16 pages long.

[1] IEEE Standard 1666 - 2011, freely available online

SystemC

Remarks:

- *Co-operative multitasking*: processes must release control back to the kernel/scheduler
  - Process executes forever $\Rightarrow$ zeno system!
- Processes may generate instantaneous events and the same process may become runnable multiple times without time advancing – *immediate* and *delta* steps
- *"The order in which process instances are selected from the set of runnable processes is implementation-defined."*

SystemC

Remarks:

- *Co-operative multitasking*: processes must release control back to the kernel/scheduler
  - Process executes forever $\Rightarrow$ zeno system!
- Processes may generate instantaneous events and the same process may become runnable multiple times without time advancing – *immediate* and *delta* steps
- *"The order in which process instances are selected from the set of runnable processes is implementation-defined."*
- Execution apparently not ordered w.r.t. dependencies.
  $\Rightarrow$ non-deterministic simulation results!

Bibliography

R. Alur and D. Dill.

A theory of timed automata.
Theoretical Computer Science, 126:183–235, 1994.

M. Broy and K. Stølen.

Specification and development of interactive systems: focus on streams, interfaces, and refinement.
Springer, 2001.

E. Lee and A. Sangiovanni-Vincentelli.

A unified framework for comparing models of computation.
IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, 17(12):1217–1229, December 1998.

E. A. Lee.

Modeling concurrent real-time processes using discrete events.
Annals of Software Engineering, 7:25–45, 1999.

J. Misra.

Distributed discrete-event simulation.
ACM Comput. Surv., 18(1):39–65, March 1986.

C. Stergiou, S. Tripakis, E. Matsikoudis, and E. A. Lee.

On the Verification of Timed Discrete-Event Models.
In *11th International Conference on Formal Modeling and Analysis of Timed Systems – FORMATS 2013*. Springer, 2013.