EE 144/244: Fundamental Algorithms for System Modeling, Analysis, and Optimization
Fall 2014

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 controllers (automotive, robotic surgeon, ...).

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).
Continuous vs. Discrete Event Systems

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

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

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

Main notion: **event**
- something occurring at some point in time
- may also carry a value
- **event** = (timestamp, value)
- systems are viewed as consumers/producers of event streams
Example System: Dense-Time Delay

\[ e_1 \quad e_2 \quad e_3 \quad e_4 \quad \ldots \]

\[ 1 \quad 1.9 \quad 2.5 \quad 4.1 \quad R_+ \]

\[ e_1 \quad e_2 \quad e_3 \quad e_4 \quad \ldots \]

\[ 2 \quad 2.9 \quad 3.5 \quad 5.1 \quad R_+ \]

\[ \text{Delay: 1} \]
Delay vs. Server

**Diagram:**
- **DE Director**
- **PoissonClock**
- **TimeDelay** (delay of 1.0)
- **Server**

**Graph:**
- TimedPlotter
- X-axis: 0.0 to 10.0
- Y-axis: 0.9 to 1.1

**Plot Data:**
- Points indicating time delays and server activities.
Discrete-Event Models (DE)

Networks of *timed 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 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

We will look at simulation of DE models.

Exhaustive verification (model-checking): open research problem! Recent work: [Stergiou et al. 2013].
DISCRETE-EVENT SIMULATION
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 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

<table>
<thead>
<tr>
<th>point in algo</th>
<th>( t )</th>
<th>( Q )</th>
<th>current event ( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>after initialization</td>
<td>0</td>
<td>([c_0, 0), (c_1, 0.6), (c_2, 1.2), \ldots)</td>
<td></td>
</tr>
<tr>
<td>after step 4</td>
<td>0</td>
<td>([c_1, 0.6), (c_2, 1.2), \ldots)</td>
<td>((c_0, 0))</td>
</tr>
<tr>
<td>after step 5</td>
<td>0</td>
<td>([c_1, 0.6), (d_0, 1.0), (c_2, 1.2), \ldots)</td>
<td></td>
</tr>
<tr>
<td>after step 6</td>
<td>0.6</td>
<td>([d_0, 1.0), (c_2, 1.2), (d_1, 1.6), \ldots)</td>
<td></td>
</tr>
<tr>
<td>after step 4</td>
<td>0</td>
<td>([d_0, 1.0), (c_2, 1.2), \ldots)</td>
<td>((c_1, 0.6))</td>
</tr>
<tr>
<td>after step 5</td>
<td>0.6</td>
<td>([d_0, 1.0), (c_2, 1.2), (d_1, 1.6), \ldots)</td>
<td></td>
</tr>
<tr>
<td>after step 6</td>
<td>1.0</td>
<td>([c_2, 1.2), (d_1, 1.6), \ldots)</td>
<td>((d_0, 1.0))</td>
</tr>
</tbody>
</table>

\( Q \) does not change, but something gets printed

\( \ldots \)
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).
- Scheme is 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: \textbf{while} \( Q \) is not empty \textbf{do}
4: \hspace{1em} remove earliest event \( e = (v_e, t_e) \) from \( Q \);
5: \hspace{1em} \( t := t_e; \)
6: \hspace{1em} execute event \( e \): update system state, generate possible future events, and add them to \( Q \), ordered by timestamps;
7: \textbf{end while}

- Appears intuitive, but details are left unspecified (steps 2, 6).
- Scheme is not \textit{modular}: step 6 appears to work on the entire system state, not on individual \textit{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.
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 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 negative, which will cause an exception.
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’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)
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 zeroness?
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 zeroness?

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 zenolessness?

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 \( t' \leq t \).

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 \( t \).
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?

![Diagram](image-url)
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.
Another Example: Alarm

Note:

- Whether an event will be generated before $t$ is not always 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: \textbf{while} \( Q \) is not empty \textbf{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; \textbf{possibly remove}
events from \( Q \);
7: \textbf{end while}
Another Issue: Simultaneous Events

The \textit{AddSubtract} actor is supposed to behave as follows:

- If it receives two simultaneous events, it adds/subtracts their values and produces a \textit{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 \textit{SingleEvent} actors produce two simultaneous events with the same value $x$.

What should the output be?
Another Issue: Simultaneous Events

The \textit{AddSubtract} actor is supposed to behave as follows:

- If it receives two simultaneous events, it adds/subtracts their values and produces a \textit{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 \textit{SingleEvent} actors produce two simultaneous events with the same value $x$.

What should the output be? A \textbf{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
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?

No: AddSubtract needs to wait for the output of Scale. Processing a set of simultaneous events may result in new simultaneous events not in ...
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 $t' \leq t$

![Diagram of Alarm Example](image-url)
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 \( \{(\text{alarm}, t), (\text{cancel}, t)\} \);
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 \( (\text{alarm}, t) \) and \( (\text{cancel}, t) \) 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.
Let $A_1, A_2$ be two actors.

Define the precedence relation $A_1 \rightarrow A_2$ ($A_2$ depends on $A_1$) as follows:

$A_1 \rightarrow A_2 \equiv A_1$ is zero-delay and there is a connection from an output of $A_1$ to an input of $A_2$ in the model.
Let $A_1, A_2$ be two actors.

Define the precedence relation $A_1 \rightarrow A_2$ ($A_2$ depends on $A_1$) as follows:

$A_1 \rightarrow A_2 \triangleq A_1$ is zero-delay and there is a connection from an output of $A_1$ to an input of $A_2$ in the model.

**Claim:** $\rightarrow$ is acyclic.

Why?
Let \( A_1, A_2 \) be two actors.

Define the precedence relation \( A_1 \rightarrow A_2 \) (\( A_2 \) depends on \( A_1 \)) as follows:

\[
A_1 \rightarrow A_2 \equiv A_1 \text{ is zero-delay and there is a connection from an output of } A_1 \text{ to an input of } A_2 \text{ in the model.}
\]

**Claim:** \( \rightarrow \) is acyclic.

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

Alarm $\rightarrow$ Sink.

We will not need Source $\rightarrow$ Alarm (sources don't participate in cycles anyway).
Actor Dependencies – Examples

Source $\rightarrow$ cancel $\rightarrow$ Alarm $\rightarrow$ alarm $\rightarrow$ Sink

Alarm $\rightarrow$ Sink.

We will not need Source $\rightarrow$ Alarm (sources don't participate in cycles anyway).

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.

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.

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 \( e_1, e_2 \):

\[
e_1 \prec e_2 \iff t_1 < t_2 \text{ or } \left( t_1 = t_2 \text{ and } A_1 \rightarrow^* A_2 \text{ and } A_1 \neq A_2 \right)
\]

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.

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 \( e_1, e_2 \):

\[
e_1 \prec e_2 \iff t_1 < t_2 \text{ or } \left( t_1 = t_2 \text{ and } A_1 \rightarrow^* A_2 \text{ and } A_1 \neq A_2 \right)
\]

where \( \rightarrow^* \) is the transitive closure of \( \rightarrow \).

Claim: \( \prec \) is acyclic.
Event Dependencies – Examples

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:

- $e_1 \prec e_2$ and $e_1 \prec e_3$.
- $e_2$ and $e_3$ are independent.
Event Dependencies – Examples

Alarm → Sink, therefore \( \text{cancel} < \text{alarm} \).

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:

\[ e_1 < e_2 \text{ and } e_1 < e_3. \]

\( e_2 \) and \( e_3 \) are independent.
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: \textbf{while} \( Q \) is not empty \textbf{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: \textbf{end while}
Discrete-Event Simulation – final version

1: \( t := 0; \)
2: initialize global event queue \( Q \) with a set of initial events;
   \[
   \text{// } Q \text{ is always implicitly ordered w.r.t. timestamps} \\
   \text{// and among events with same timestamp w.r.t. event dependencies} \\
   \text{// which depend on no other events}
   \]
3: \textbf{while } \( Q \) is not empty \textbf{ do}
4: \hspace{1em} remove set \( E \) of all minimal events w.r.t. \( \prec \) from \( Q; \)
   \[
   \text{// these are earliest and simultaneous events,} \\
   \text{// which depend on no other events}
   \]
5: \hspace{1em} \( t := t_e; \)
6: \hspace{1em} 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: \textbf{end while}

\textbf{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 \( E \).
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\textsuperscript{1} is > 600 pages long.
   ▶ Description of the simulation algorithm (in English) is 16 pages long.

\textsuperscript{1}IEEE Standard 1666 - 2011, freely available online
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.”
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-determinism!
M. Broy and K. Stølen.
 Specification and development of interactive systems: focus on streams, interfaces, and refinement.

E. Lee and A. Sangiovanni-Vincentelli.
 A unified framework for comparing models of computation.

E. A. Lee.
 Modeling concurrent real-time processes using discrete events.

J. Misra.
 Distributed discrete-event simulation.

 On the Verification of Timed Discrete-Event Models.