Component Architectures for Time-Sensitive Systems
Part 2

Edward A. Lee
Robert S. Pepper Distinguished Professor and
The Onassis Foundation Science Lecture Series
The 2008 Lectures in Computer Science
Embedded Networked Systems: Theory and Applications

With thanks to Thomas Huning Feng, Yang Zhao, and Ye (Rachel) Zhou
Heraklion, Crete
July 24-28, 2008

Our Solution

Reintroduce time into the core abstractions:

- **Foundations**: Timed computational semantics.
- **Bottom up**: Make timing repeatable.
- **Top down**: Timed, concurrent components.
- **Holistic**: Model engineering.

this talk is posted at http://chess.eecs.berkeley.edu/pubs/472.html
### Object Oriented vs. Actor Oriented

#### The established: Object-oriented:

<table>
<thead>
<tr>
<th>class name</th>
<th>data</th>
<th>methods</th>
</tr>
</thead>
</table>

What flows through an object is sequential control

![Diagram showing object-oriented programming](image)

#### The alternative: Actor oriented:

<table>
<thead>
<tr>
<th>actor name</th>
<th>data (state)</th>
<th>parameters</th>
<th>ports</th>
</tr>
</thead>
</table>

What flows through an object is evolving data

![Diagram showing actor-oriented programming](image)

### Our Agenda

I will show a particular approach to the design of concurrent and distributed time-sensitive systems that is an actor-oriented component technology.

The approach is called PTIDES (pronounced “tides”), for Programming Temporally Integrated Distributed Embedded Systems.


Our Approach is based on Discrete Events (DE)

- Concurrent actors
- Exchange time-stamped messages

A correct execution is one where every actor reacts to input events in time-stamp order.

Time stamps are in “model time,” which typically bears no relationship to “real time” (wall-clock time).

Example

DE Director specifies that this will be a DE model
Example

Model of regularly spaced events (e.g., a clock signal).

Example

Model of irregularly spaced events (e.g., a failure event).
Example

Model of a subsystem that goes down on error events

Example

Model of an observer subsystem
Example

Events on the two input streams must be seen in time stamp order.

Note that DE MoCs have considerable subtleties when it comes to simultaneous events and events that prevent time from progressing (Zeno conditions).

This is a Component Technology

Model of a subsystem given as a state machine.
This is a Component Technology

Other types of components:
- Functional expressions.
- Submodels in DE
- Submodels in other MoCs

Using DE Semantics in Distributed Real-Time Systems

- DE is usually a simulation technology.
- Distributing DE is done for acceleration.
- Hardware design languages (e.g. VHDL) use DE where time stamps are literally interpreted as real time, or abstractly as ticks of a physical clock.

- We are using DE for distributed real-time software, binding time stamps to real time only where necessary.
- PTIDES: Programming Temporally Integrated Distributed Embedded Systems
Consider a simpler scenario:

Assumption: Wall clocks on the distributed platforms are synchronized to some known precision (e.g. NTP, IEEE 1588)
PTIDES: Programming Temporally Integrated Distributed Embedded Systems

Bind model time to real time at the sensors:

Output time stamps are ≤ real time

Bind model time to real time at the actuators:

Input time stamps are ≥ real time
**PTIDES: Programming Temporally Integrated Distributed Embedded Systems**

*Schedulability is not violating these timing inequalities.*

Assume bounded sensor delay $s$

Assume bounded computation time $c_1$

Assume bounded computation time $c_2$

Assume bounded clock error $e$

Assume bounded computation time $c_3$

An event here with time stamp $t$ can be safely merged when real time exceeds $t + s + d + e + \max(c_1, c_2) + c_3$

**PTIDES uses static causality analysis to determine when events can be safely processed.**
The execution model prevents remote processes from blocking local ones, and does not require backtracking.

An event here with time stamp $t$ can be safely merged when real time exceeds $t + s + d + e + \max(c_1, c_2) + c_3$.

However, this program is not schedulable!

The resulting event here with time stamp $t$ cannot be presented to the actuator until real time exceeds $t + s + d + e + \max(c_1, c_2) + c_3$. 
Remote events also trigger real-time violations. Schedulability analysis tells us the program is flawed.

The program can be fixed with actors that increment the time stamps (model-time delays).
This relaxes scheduling constraints...

Through static analysis we can derive sufficient conditions for schedulability...

The model is schedulable if:
1) \( s + d + e - d_2 + c_1 + c_3 < 0 \)
2) \( s + d + e - d_2 + c_2 + c_3 < 0 \)
3) ...
PTIDES: Programming Temporally Integrated Distributed Embedded Systems

… and being explicit about time delays means that we can analyze control system dynamics...

The system is stable if...

Compare with Classical Distributed DE Simulation Technologies

Conservative distributed DE (Chandy & Misra) would block actuation unnecessarily.
Compare with Classical Distributed DE Simulation Technologies

Optimistic distributed DE (Jefferson) would require being able to roll back the physical world.

But this schedulability analysis is not quite as easy as it might look

**Bounding computation time requires careful analysis of execution time and scheduling policies.**
But this schedulability analysis is not quite as easy as it might look.

Bounding clock error requires network time synchronization (IEEE 1588, NTP)

Assume bounded sensor delay $s$
Assume bounded computation time $c_1$
Assume bounded computation time $c_2$
Assume bounded clock error $e$

An event here with time stamp $t$ can be safely merged when real time exceeds $t + s + d + e + \max(c_1, c_2) + c_3$.

Bounding network delay requires a real-time network (FlexRay, TTP, …)

Assume bounded sensor delay $s$
Assume bounded computation time $c_1$
Assume bounded computation time $c_2$
Assume bounded clock error $e$

An event here with time stamp $t$ can be safely merged when real time exceeds $t + s + d + e + \max(c_1, c_2) + c_3$. 
But this schedulability analysis is not quite as easy as it might look

*Bounding sensor delay requires bounding interrupt latency and thread context switching.*

- Assume bounded sensor delay $s$
- Assume bounded computation time $c_1$
- Assume bounded computation time $c_2$
- Assume bounded computation time $c_3$
- Assume bounded clock error $e$

An event here with time stamp $t$ can be safely merged when real time exceeds

$$t + s + d + e + \max(c_1, c_2) + c_3$$

Making this Systematic

Levels of analysis:

1. Assume zero execution time for actors (exposes modeling errors)
2. Assume known worst-case execution time (WCET) for actors, unbounded compute resources (exposes complexity problems).
3. Assume WCET and a scheduling policy over finite resources (exposes resources limitations).
Exposing Modeling Errors

Levels of analysis:
1. Assume zero execution time for actors (exposes modeling errors)
2. Assume known worst-case execution time (WCET) for actors, unbounded compute resources (exposes inadequate compute speed).
3. Assume WCET and a scheduling policy over finite resources (exposes resources limitations).

Do this using causality interfaces.


Causality Interfaces

δ : P × P → R+ U {∞} yields the minimum model-time delay between any two ports (a causality interface).
(P – set of ports; R+ – set of non-negative real numbers)

Infer causality from causality interfaces using a min-plus algebra.
Example: δ(i₅, o₁) = min(δ₅ + δ₁, δ₅ + δ₄ + δ₂), where δ₁, …, δ₆ ∈ R+ are pre-defined.
When is it safe to process $e = (v, t)$ at $i_1$?

1. future events at $i_1$, $i_2$ and $i_3$ have time stamps $\geq t$ (conventional), or
2. future events at $i_1$ and $i_2$ have time stamps $\geq t$, or
3. future events at $i_1$ have time stamps $\geq t$, and future events at $i_2$ depend on events at $i_4$ with time stamps $\geq t - \delta_4$, or
4. future events at $i_1$ and $i_2$ depend on events at $i_5$ and $i_6$ with time stamps $\geq t - \min\{\delta_5, \delta_6, \delta_5 + \delta_4, \delta_6 + \delta_4\}$.

### Relevant Dependency [Ye Zhou]

$i \sim i'$ iff they are input of the same actor and affect a common output. An equivalence class is a transitive closure of $\sim$.

Construct a collapsed graph, and compute relevant dependency between equivalence classes.

$$d(e', e) = \min_{i', e' \in e, i \in e} \{\delta(i', i)\}$$
A dependency cut for \( \varepsilon \) is a minimal but complete set of equivalence classes that needs to be considered to process an event at \( \varepsilon \).

Example: \( C_1 \) and \( C_2 \) are both dependency cuts for \( \varepsilon_1 \).

Choosing a Dependency Cut

Determine earliest event \( e = (v, t) \) at \( \varepsilon_1 \) safe to process

- If we choose \( C_1 \): all unprocessed events at \( \varepsilon_1 \) will have time stamps \( \geq t \).
- If we choose \( C_2 \): for any \( \varepsilon \in C_2 \), all unprocessed events at \( \varepsilon \) depend on events at \( \varepsilon_1 \) with time stamps \( \geq t - d(\varepsilon, \varepsilon_1) \).
- We can freely choose a dependency cut.
Reference Application: Distributed Cameras

- $n$ cameras located around a football field, all connected to a central computer.
- Events at **blue** ports satisfy $t \leq \tau$
  ($t$ – time stamp of any event; $\tau$ – real time)
- Events at **red** ports satisfy $t \geq \tau$

Problems to solve

- Make event-processing decisions locally
- Guarantee timely command delivery to the Devices
- Guarantee real-time update at the Display
- Tolerate images loss or corruption at Image Processor
Choose Dependency Cuts at Platform Boundaries

- $n + 1$ platforms with synchronized clocks (IEEE 1588).
- Choose dependency cuts at platform boundary.
- A queue stores events local to the platform.
- At real time $r$, future events have time stamps $\geq r - d_n$.  

Time-sensitive computation is significantly different from other computation

Some misleading statements:

- “Computing takes time”
- “Time is a resource”
- “Time is a non-functional property”
- “Real time is a quality of service problem”