

Is Truly Real-Time Computing Becoming Unachievable?

Edward A. Lee

*Robert S. Pepper Distinguished Professor and
Chair of EECS, UC Berkeley*

Keynote Talk

Real-Time and Embedded Technology and Applications Symposium (RTAS)

Bellevue, WA

April 3 - April 6, 2007

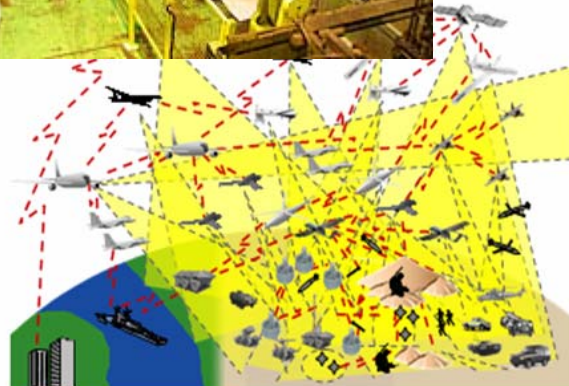


The Vision: Reliable and Evolvable Networked Time-Sensitive Systems, Integrated with Physical Processes

Orchestrated networked resources built with sound design principles on suitable abstractions.



<p>POWERTRAIN & DRIVETRAIN SYSTEMS MAGNA POWERTRAIN</p> <ul style="list-style-type: none"> • Drivetrain Systems & Components • Engine Systems & Components • Axles & Chassis Modules 	<p>MIRROR SYSTEMS MAGNA DONNELLY</p> <ul style="list-style-type: none"> • Interior Mirrors (Pneumatic & Electrochromic) • Exterior Mirrors • Camera Mirror Systems • Engineered Glass 	<p>CLOSURE SYSTEMS MAGNA CLOSURES</p> <ul style="list-style-type: none"> • Door Modules • Power Closure Systems • Locking Systems • Window Systems • Driver Controls • Handle Assemblies
<p>ELECTRONIC SYSTEMS MAGNA ELECTRONICS</p> <ul style="list-style-type: none"> • Power Systems • Driver Assistance & Safety • Body Electronics 	<p>SEATING SYSTEMS MAGNA AUTOMOTIVE SEATING</p> <ul style="list-style-type: none"> • Seating Systems • Seating Hardware Systems 	
<p>METAL BODY & CHASSIS SYSTEMS MAGNA INTERNATIONAL</p> <ul style="list-style-type: none"> • Body Systems • Chassis Systems 	<p>PLASTIC BODY LIGHTING & EXTERIOR MAGNA INTERNATIONAL</p> <ul style="list-style-type: none"> • Front and Rear End Modules • Exterior Trim • Plastic Body Panels • Lighting Systems • Concentration & Sealing Systems • Outside Entertainment Packages 	<p>COMPLETE VEHICLE ENGINEERING & ASSEMBLY MAGNA STEYR</p> <ul style="list-style-type: none"> • Complete Vehicles: Engineering & Assembly • Modules and Components • Space Technology

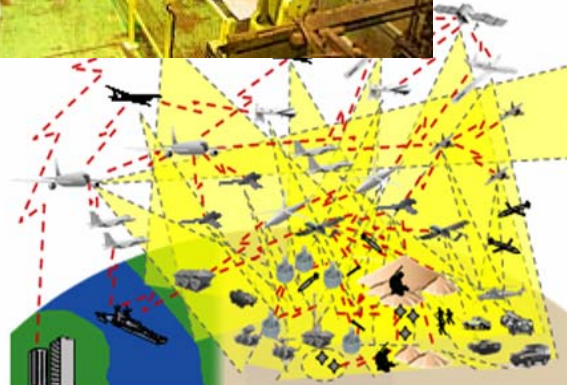


The Vision: Reliable and Evolvable Networked Time-Sensitive Systems, Integrated with Physical Processes

Orchestrated networked resources built with *sound design principles* on *suitable abstractions*.



<p>POWERTRAIN & DRIVETRAIN SYSTEMS MAGNA POWERTRAIN</p> <ul style="list-style-type: none"> • Drivetrain Systems & Components • Engine Systems & Components • Axles & Chassis Modules 	<p>MIRROR SYSTEMS MAGNA DONNELLY</p> <ul style="list-style-type: none"> • Interior Mirrors: Pneumatic & Electrochromic • Exterior Mirrors • Camera Mirror Systems • Engineered Glass 	<p>CLOSURE SYSTEMS MAGNA CLOSURES</p> <ul style="list-style-type: none"> • Door Modules • Power Closure Systems • Locking Systems • Window Systems • Driver Controls • Handle Assemblies
<p>ELECTRONIC SYSTEMS MAGNA ELECTRONICS</p> <ul style="list-style-type: none"> • Power Systems • Driver Assistance & Safety • Body Electronics 		
<p>SEATING SYSTEMS BOSCH AUTOMOTIVE SEATING</p> <ul style="list-style-type: none"> • Seating Systems • Seating Hardware Systems 		
<p>METAL BODY & CHASSIS SYSTEMS CUMMINS INTERNATIONAL</p> <ul style="list-style-type: none"> • Body Systems • Chassis Systems 	<p>PLASTIC BODY LIGHTING & EXTERIOR TRIM SYSTEMS CUMMINS INTERNATIONAL</p> <ul style="list-style-type: none"> • Front and Rear End Modules • Exterior Trim • Plastic Body Panels • Lighting Systems • Concentric & Sealing Systems • Outside Entertainment Packages 	<p>COMPLETE VEHICLE ENGINEERING & ASSEMBLY MAGNA STEYR</p> <ul style="list-style-type: none"> • Complete Vehicles: Engineering & Assembly • Modules and Components • Space Technology



A Fact About Programs

Correct execution of a program in C, C#, Java, Haskell, etc. has nothing to do with how long it takes to do anything. All our computation and networking abstractions are built on this premise.



Timing of programs is not repeatable, except at very coarse granularity.

Programmers have to step *outside* the programming abstractions to specify timing behavior.



Techniques that Exploit this Fact

- Programming languages
- Virtual memory
- Caches
- Dynamic dispatch
- Speculative execution
- Power management (voltage scaling)
- Memory management (garbage collection)
- Just-in-time (JIT) compilation
- Multitasking (threads and processes)
- Component technologies (OO design)
- Networking (TCP)
- ...

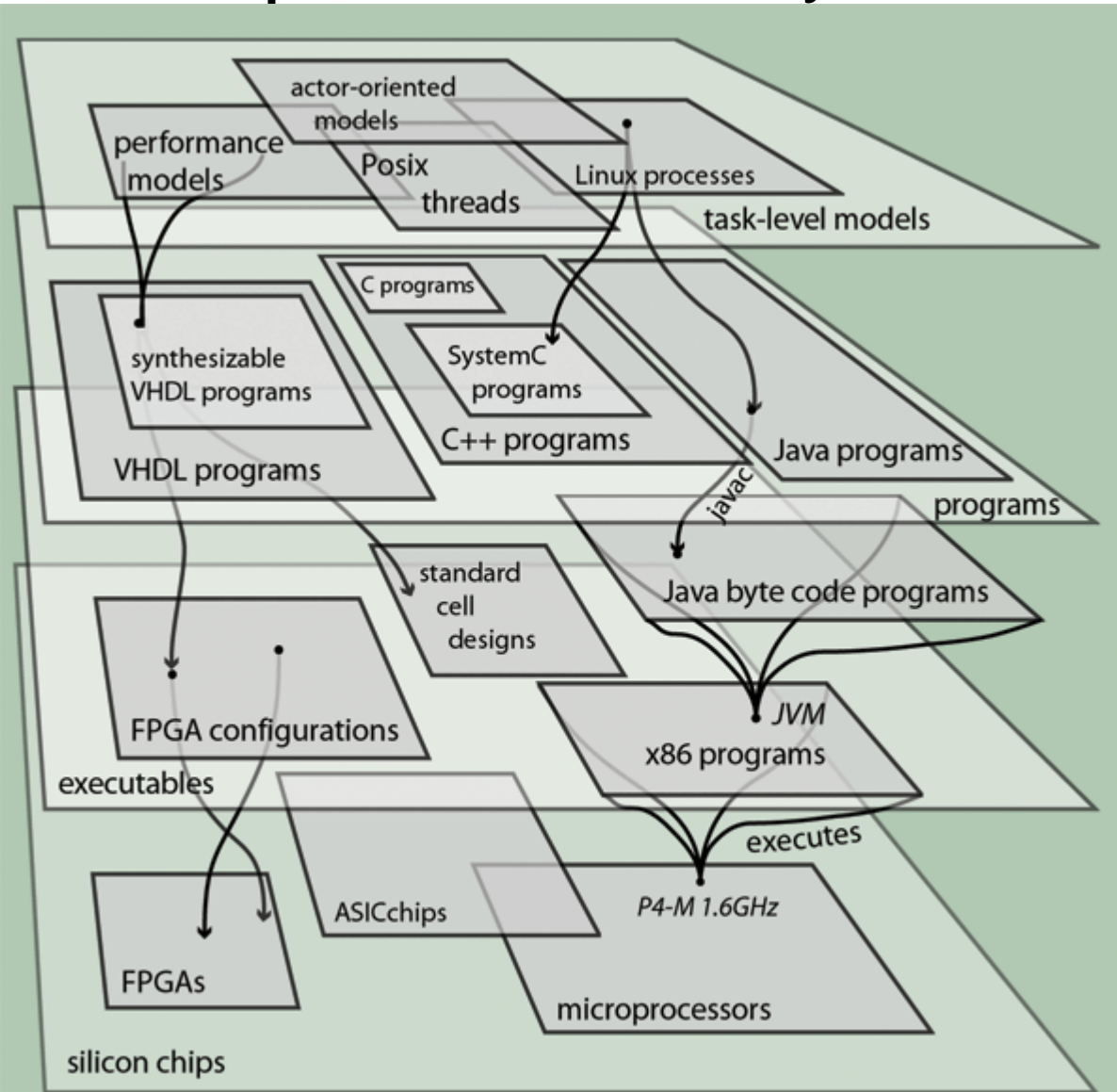


A Story

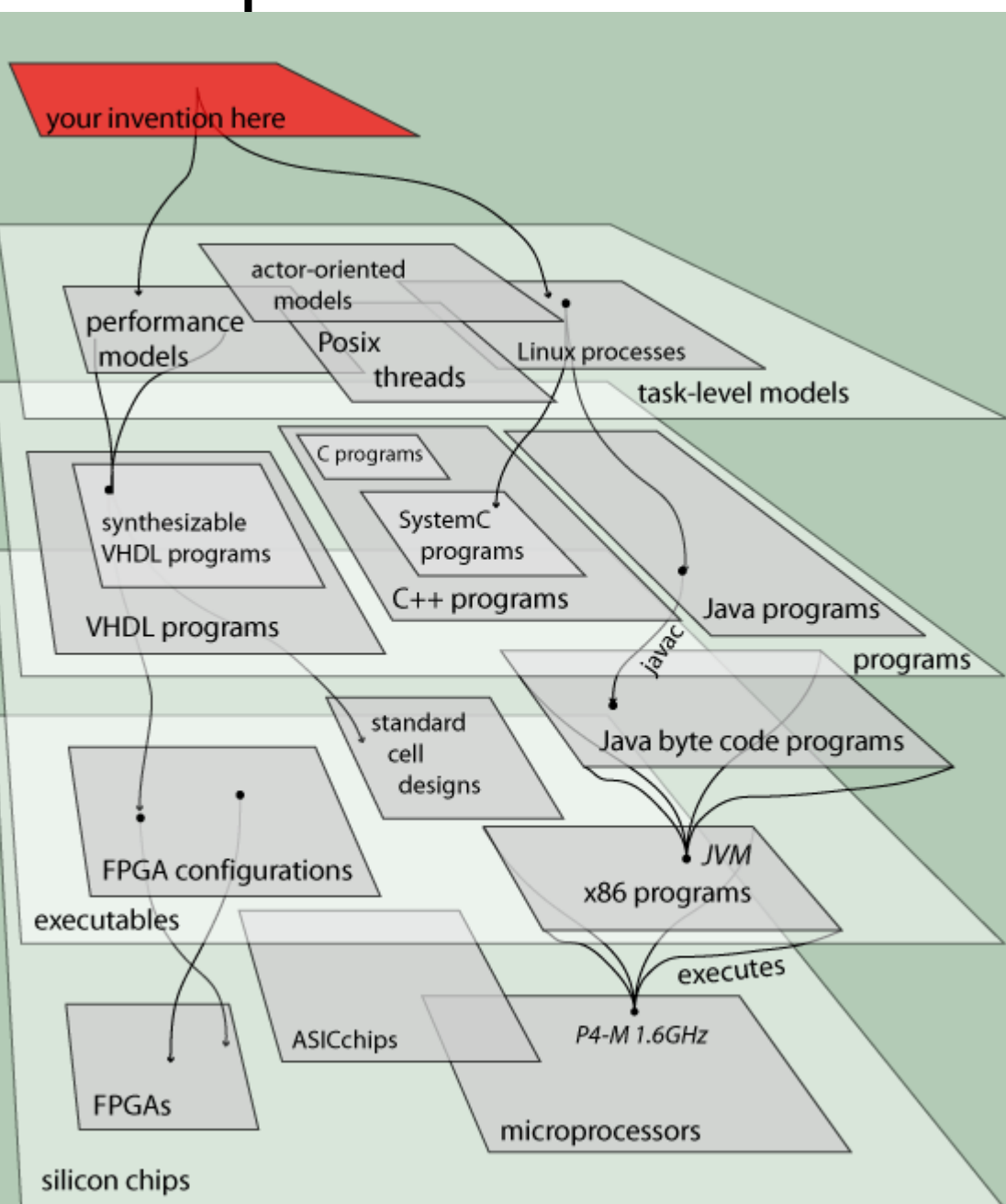


In “fly by wire” aircraft, certification of the software is extremely expensive. Regrettably, it is not the software that is certified but the entire system. If a manufacturer expects to produce a plane for 50 years, it needs a 50-year stockpile of fly-by-wire components that are all made from the same mask set on the same production line. Even a slight change or “improvement” might affect timing and require the software to be re-certified.

Abstraction Layers



The purpose for an abstraction is to hide details of the implementation below and provide a platform for design from above.



How about “raising the level of abstraction” to solve these problems?



But these higher abstractions rely on an increasingly problematic fiction: WCET

A war story:

Ferdinand et al. determine the WCET of astonishingly simple avionics code from Airbus running on a Motorola ColdFire 5307, a pipelined CPU with a unified code and data cache. Despite the software consisting of a fixed set of non-interacting tasks containing only simple control structures, their solution required detailed modeling of the seven-stage pipeline and its precise interaction with the cache, generating a large integer linear programming problem. The technique successfully computes WCET, but only with many caveats that are increasingly rare in software.

Fundamentally, the ISA of the processor has failed to provide an adequate abstraction.

C. Ferdinand et al., “Reliable and precise WCET determination for a real-life processor.” EMSOFT 2001.



The Key Problem

Electronics technology delivers highly and precise timing...

... and the overlaying software abstractions discard it.



Real-Time and Concurrency are Integrally Intertwined

Threads and objects dominate concurrent software.

- *Threads*: Sequential computation with shared memory.
- *Objects*: Collections of state variables with procedures for observing and manipulating that state.

Even distributed objects create the illusion of shared memory through proxies.

- The components (objects) are (typically) not active.
- Threads weave through objects in unstructured ways.
- This is the source of many software problems.



My Claim

*Nontrivial software written with threads,
and locks are incomprehensible to
humans.*

Is Concurrency Hard?



*It is not
concurrency that
is hard...*



...It is Threads that are Hard!

Threads are sequential processes that share memory. From the perspective of any thread, the entire state of the universe can change between any two atomic actions (itself an ill-defined concept).

Imagine if the physical world did that...



Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).



We Can Incrementally Improve Threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order...)
- Libraries (Stapl, Java 5.0, ...)
- Patterns (MapReduce, ...)
- Transactions (Databases, ...)
- Formal verification (Blast, thread checkers, ...)
- Enhanced languages (Split-C, Cilk, Guava, ...)
- Enhanced mechanisms (Promises, futures, ...)

But is it enough to refine a mechanism with flawed foundations?

Do Threads Provide a Sound Foundation?

If the foundation is bad, then we either tolerate *brittle designs* that are difficult to make work, or we have to rebuild from the foundations.

Note that this whole enterprise is held up by threads





What are Brittle Designs?

Small changes have big consequences...

Patrick Lardieri, *Lockheed Martin ATL*, about a vehicle management system in the JSF program:

“Changing the instruction memory layout of the Flight Control Systems Control Law process to optimize ‘Built in Test’ processing led to an unexpected performance change - System went from meeting real-time requirements to missing most deadlines due to a change that was expected to have no impact on system performance.”

National Workshop on High-Confidence Software Platforms for Cyber-Physical Systems (HCSP-CPS)
Arlington, VA November 30 –December 1, 2006

The Current State of Affairs

We build real-time software on abstractions where time is irrelevant using concurrency models that are incomprehensible.



Just think what we could do with the right abstractions!



My Proposed Solution

Reintroduce time into the core abstractions:

- *Bottom up*: **Make timing repeatable.**
- *Top down*: **Timed, concurrent components.**



Bottom Up: Make Timing Repeatable

We need a major historical event like this one:

*In 1980, Patterson and Ditzel **did not** invent reduced instruction set computers (RISC machines).*

See D. A. Patterson and D. R. Ditzel. “The case for the reduced instruction set computer.” *ACM SIGARCH Computer Architecture News*, 8(6):25–33, Oct. 1980.



It is Time for Another Major Historical Event

*In 2007, Edwards and Lee **did not** invent
precision-timed computers (PRET machines).*

See S. Edwards and E. A. Lee, "**The Case for the Precision
Timed (PRET) Machine**," to appear in the *Wild and Crazy
Ideas* Track of the *Design Automation Conference* (DAC), June
2007.

see: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.html>



Can Hardware Deliver on Timed Semantics?

PRET (Precision Timing) Machines

Make temporal behavior as important as logical function.

Timing precision is easy to achieve if you are willing to forgo performance. Let's not do that. Challenges:

- Memory hierarchy (scratchpads?)
- Deep pipelines (interleaving?)
- ISAs with timing (deadline instructions?)
- Predictable memory management (Metronome?)
- Languages with timing (discrete events? Giotto?)
- Predictable concurrency (synchronous languages?)
- Composable timed components (actor-oriented?)
- Precision networks (TTA? Time synchronization?)
- Dynamic adaptability (admission control?)

Making PRET Machines Practical

- Start with hardware designs on FPGAs
- Use soft cores
- Add precision-timing instructions
- Scale up from there



BEE 2, FPGA system, from BWRC



Ramp blue experimental platform, from the RAMP project.

e.g. Stephen Edwards (Columbia) has achieved software designs with ~40ns timing precision on simple soft cores. Source code is smaller and simpler than VHDL specification of comparable hardware.



Our Solution

Reintroduce time into the core abstractions:

- *Bottom up*: **Make timing repeatable.**

- *Top down*: **Timed, concurrent components.**

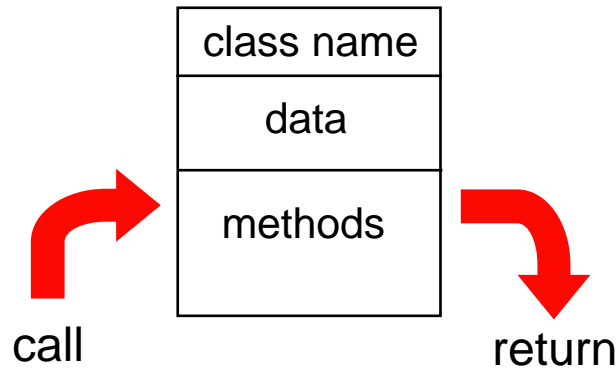


New Component Technology is more Palatable than New Languages

- It leverages:
 - Language familiarity
 - Component libraries
 - Legacy subsystems
 - Design tools
 - The simplicity of sequential reasoning
- It allows for innovation in
 - Distributed time-sensitive system design
 - Hybrid systems design
 - Service-oriented architectures
- Software is intrinsically concurrent
 - Better use of multicore machines
 - Better use of networked systems
 - Better potential for robust design

Object Oriented vs. Actor Oriented

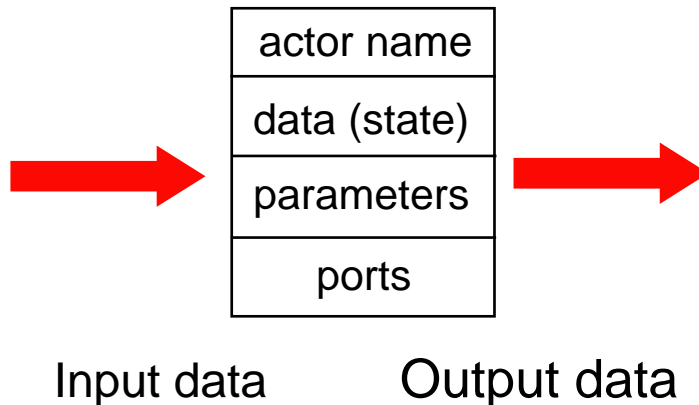
The established: Object-oriented:



What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:



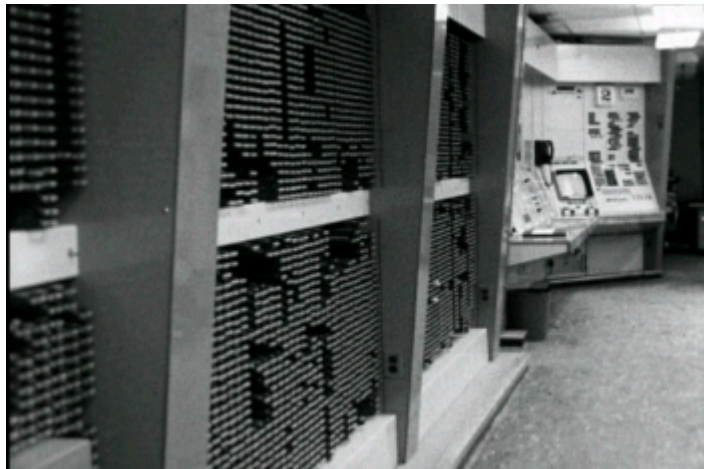
Actors make things happen

What flows through an object is evolving data

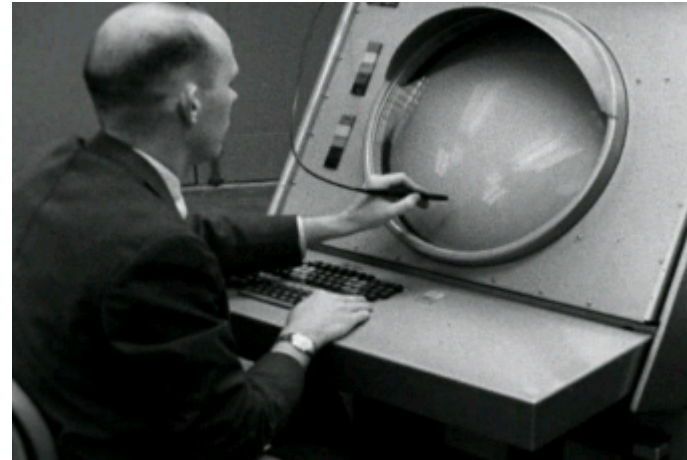
The First (?) Actor-Oriented Programming Language

The On-Line Graphical Specification of Computer Procedures

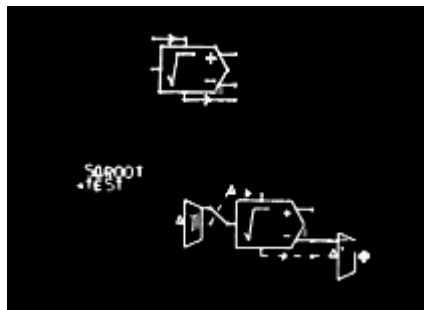
W. R. Sutherland, Ph.D. Thesis, MIT, 1966



MIT Lincoln Labs TX-2 Computer

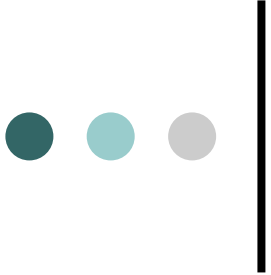


Bert Sutherland with a light pen



Bert Sutherland used the first acknowledged object-oriented framework (Sketchpad, created by his brother, Ivan Sutherland) to create the first actor-oriented programming language (which had a visual syntax).

Partially constructed actor-oriented model with a class definition (top) and instance (below).



Examples of Actor-Oriented Systems

- CORBA event service (distributed push-pull)
- ROOM and UML-2 (dataflow, Rational, IBM)
- VHDL, Verilog (discrete events, Cadence, Synopsys, ...)
- LabVIEW (structured dataflow, National Instruments)
- Modelica (continuous-time, constraint-based, Linkoping)
- OPNET (discrete events, Opnet Technologies)
- SDL (process networks)
- Occam (rendezvous)
- Simulink (Continuous-time, The MathWorks)
- SPW (synchronous dataflow, Cadence, CoWare)
- ...

Most of these are domain specific.

Many of these have visual syntaxes.

The semantics of these differ considerably, with significantly different approaches to concurrency.



Challenges

The technology is immature:

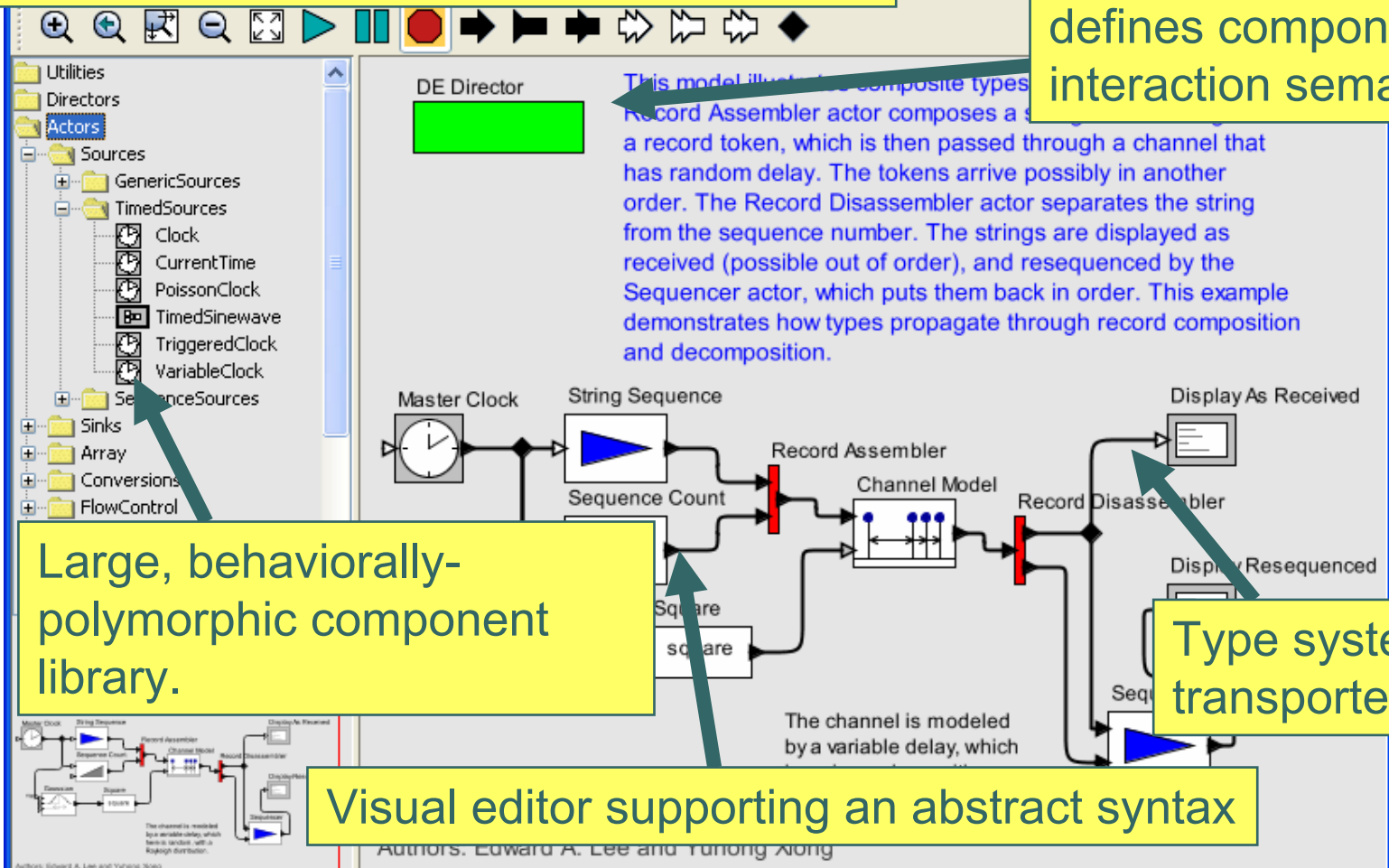
- Commercial actor-oriented systems are domain-specific
- Development tools are limited
- Little language support in C++, C#, Java
- Modularity mechanisms are underdeveloped
- Type systems are primitive
- Compilers (called “code generators”) are underdeveloped
- Formal methods are underdeveloped
- Libraries are underdeveloped

We are addressing these problems.

Enter Ptolemy II: Our Laboratory for Experiments with Models of Computation

Concurrency management supporting dynamic model structure.

Director from a library defines component interaction semantics

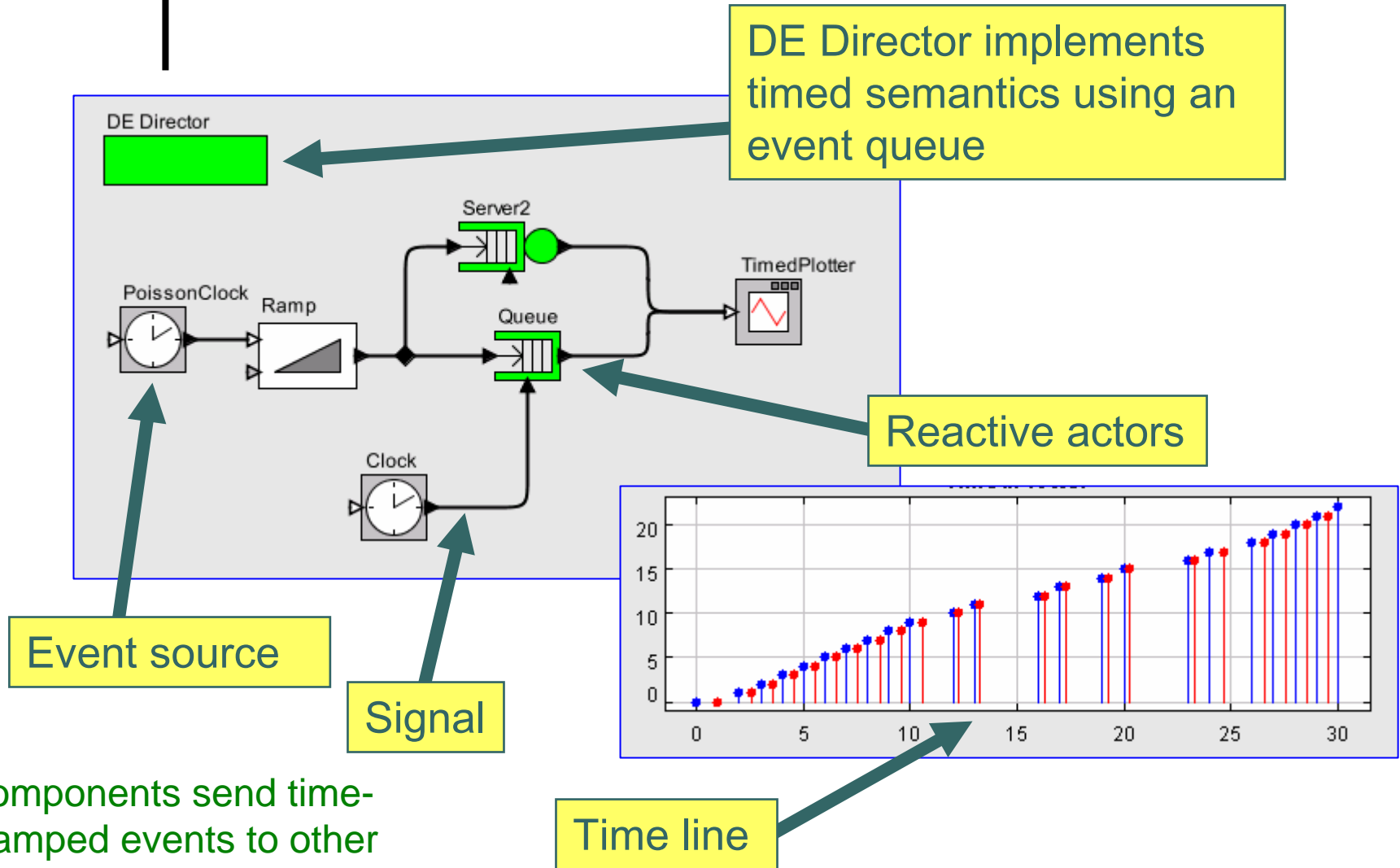


Large, behaviorally-polymorphic component library.

Type system for transported data

Visual editor supporting an abstract syntax

Example: Discrete Event Models





Using DE Semantics in Distributed Real-Time Systems

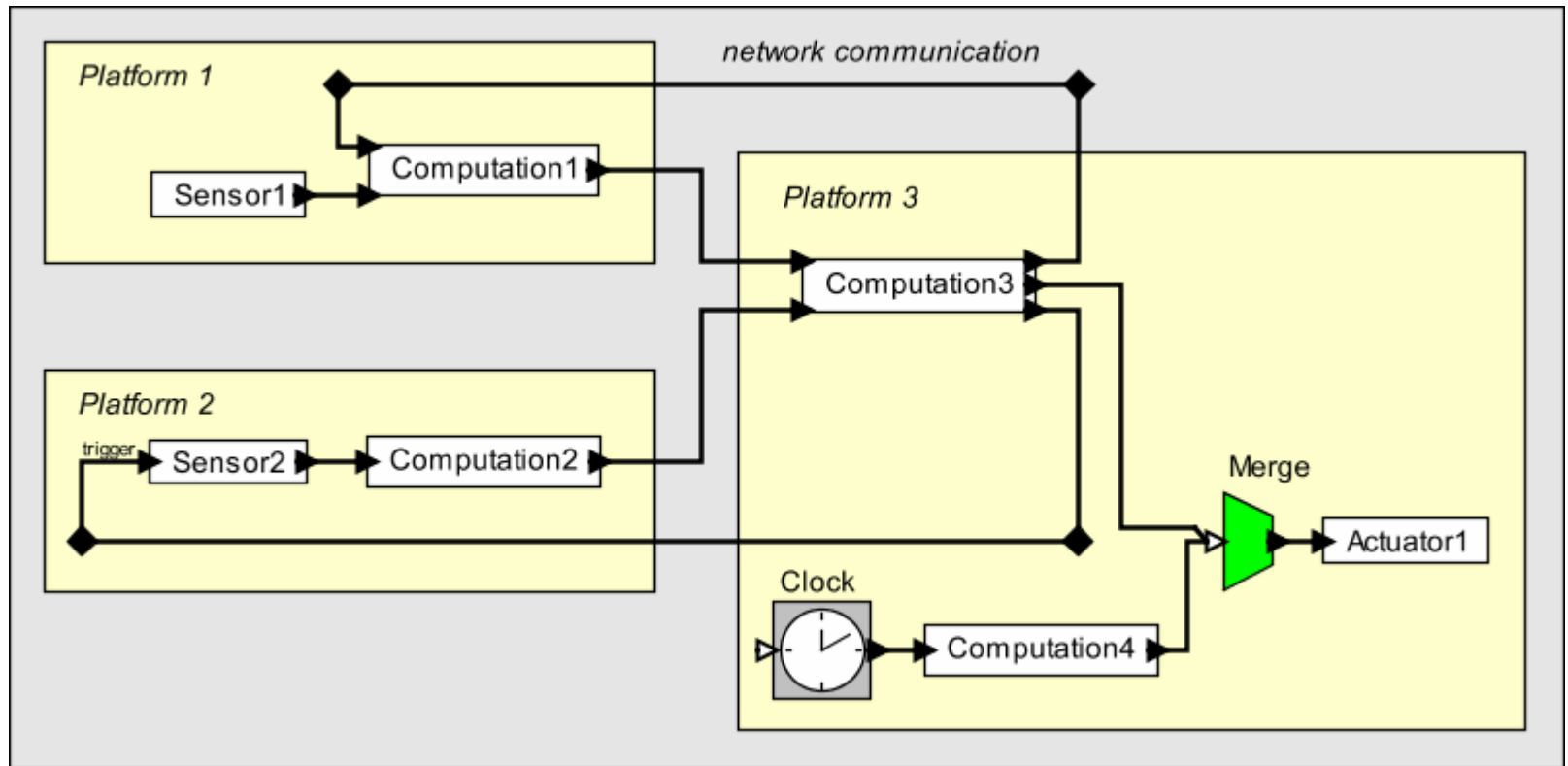
- DE is usually a simulation technology.
- Distributing DE is done for acceleration.
- Hardware design languages (e.g. VHDL) use DE where time stamps are literally interpreted as real time.

- 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 (work done with Yang Zhao and Jie Liu).*

PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

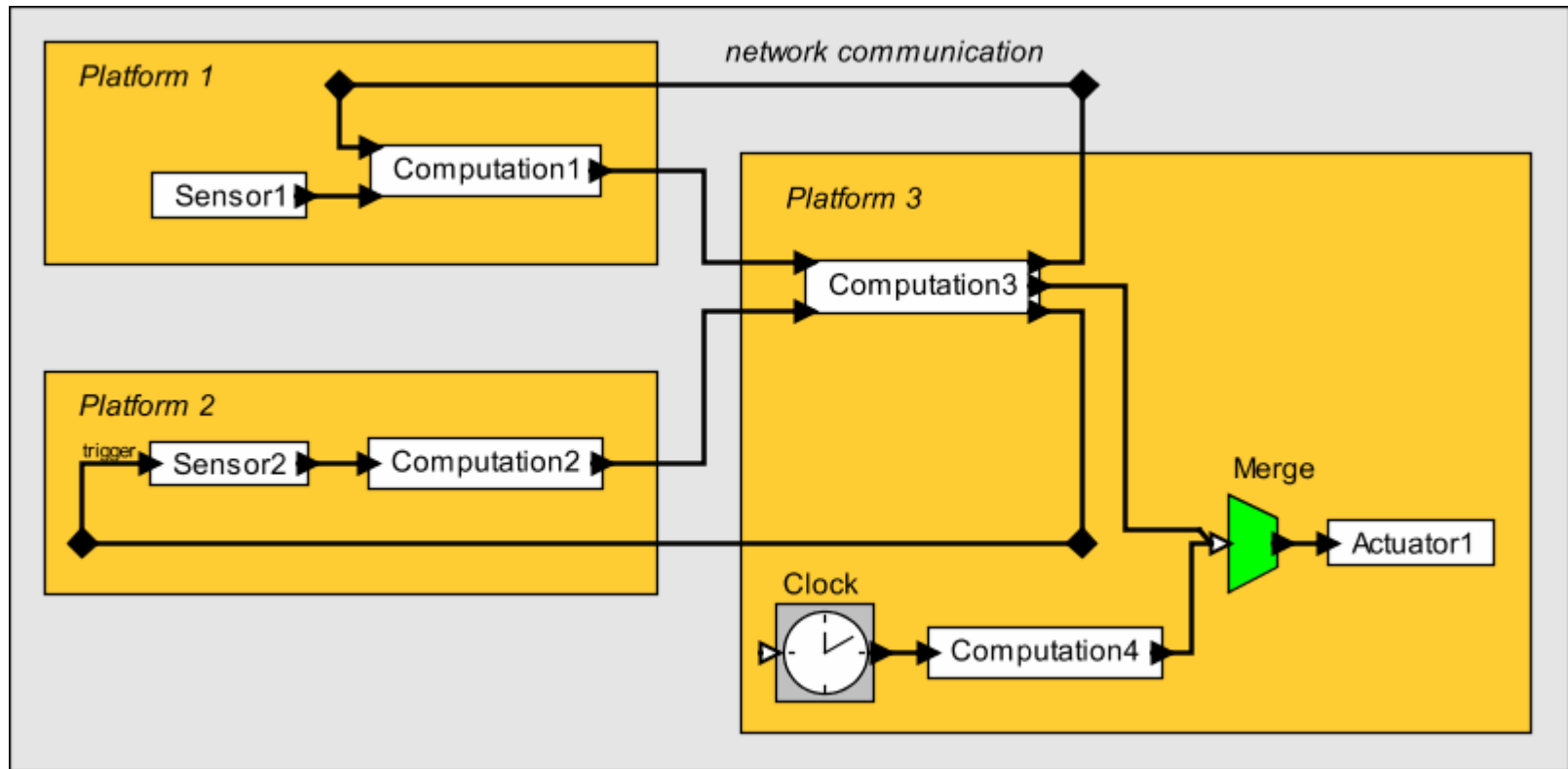
Consider a scenario:



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

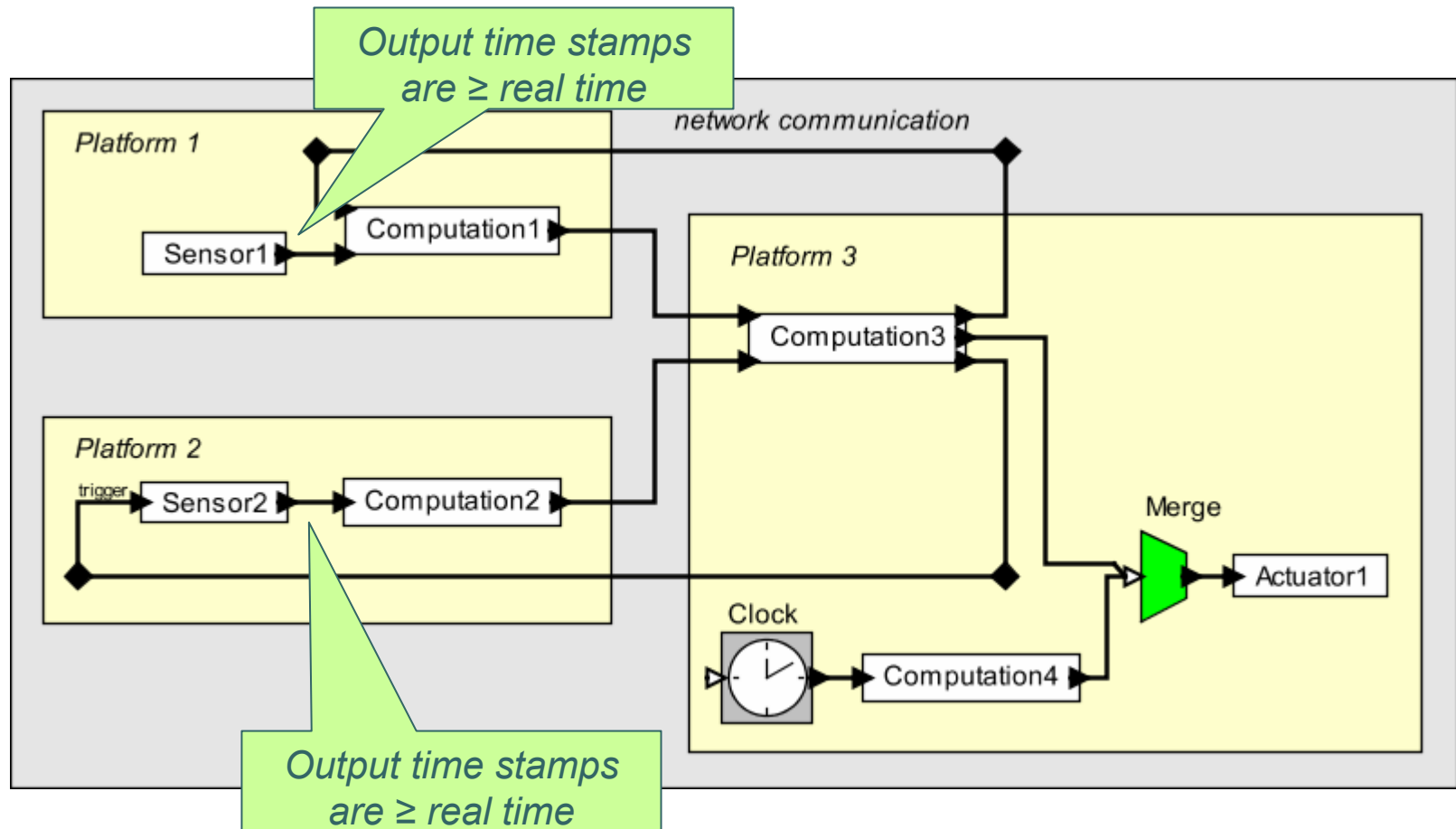
Assumption: Clocks on the distributed platforms are synchronized to some known precision (e.g. NTP, IEEE 1588)



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

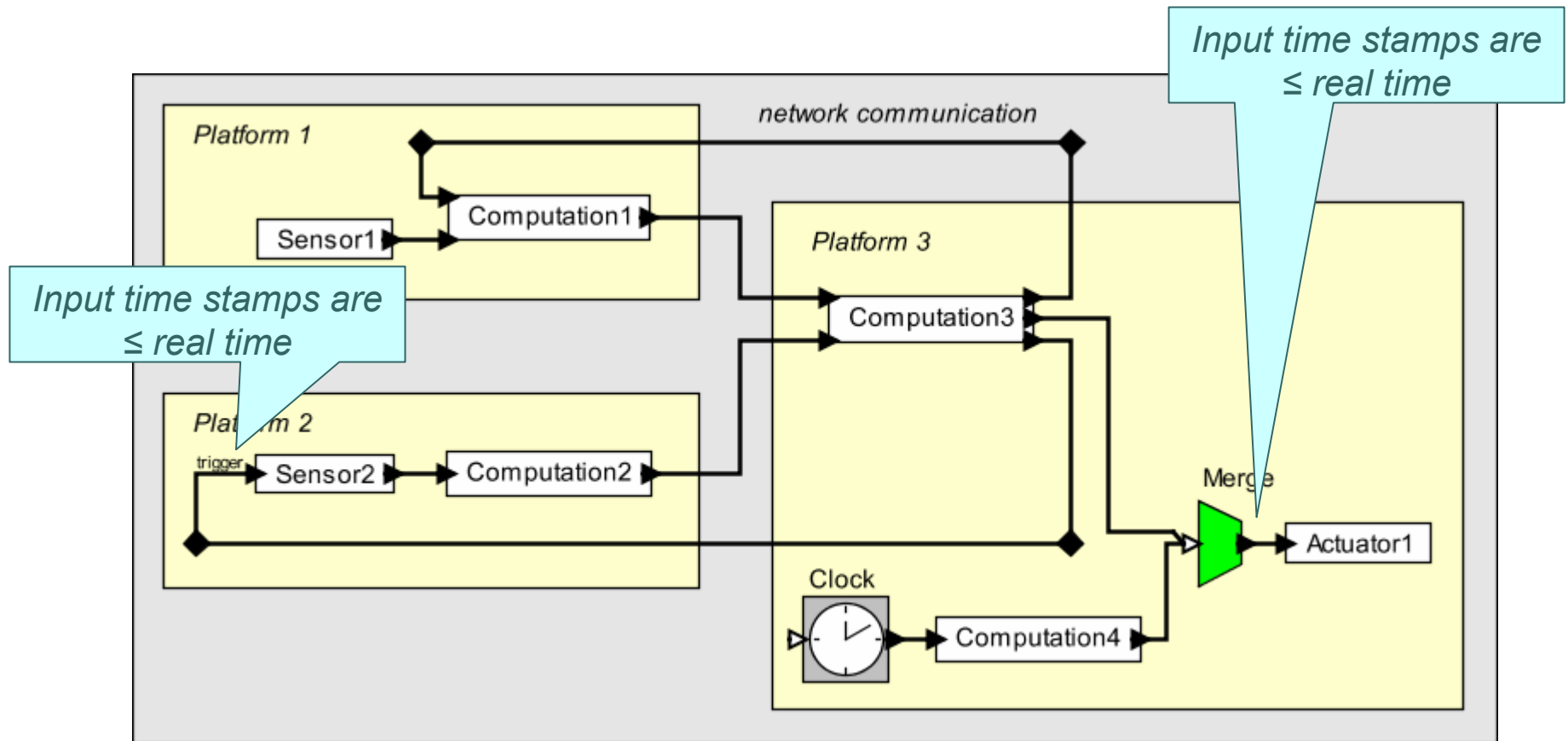
Bind model time to real time at the *sensors*:



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

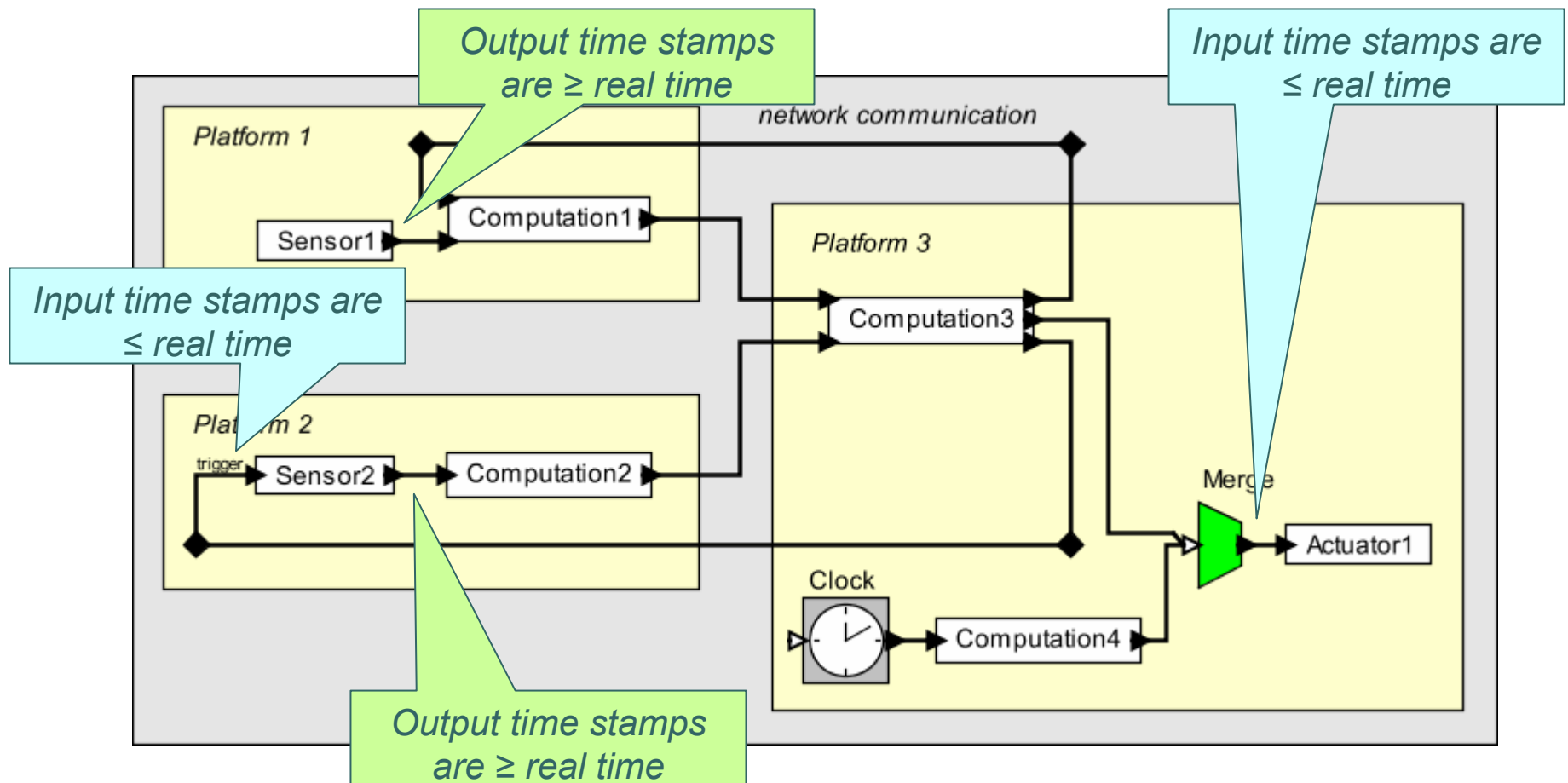
Bind model time to real time at the *actuators*:



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

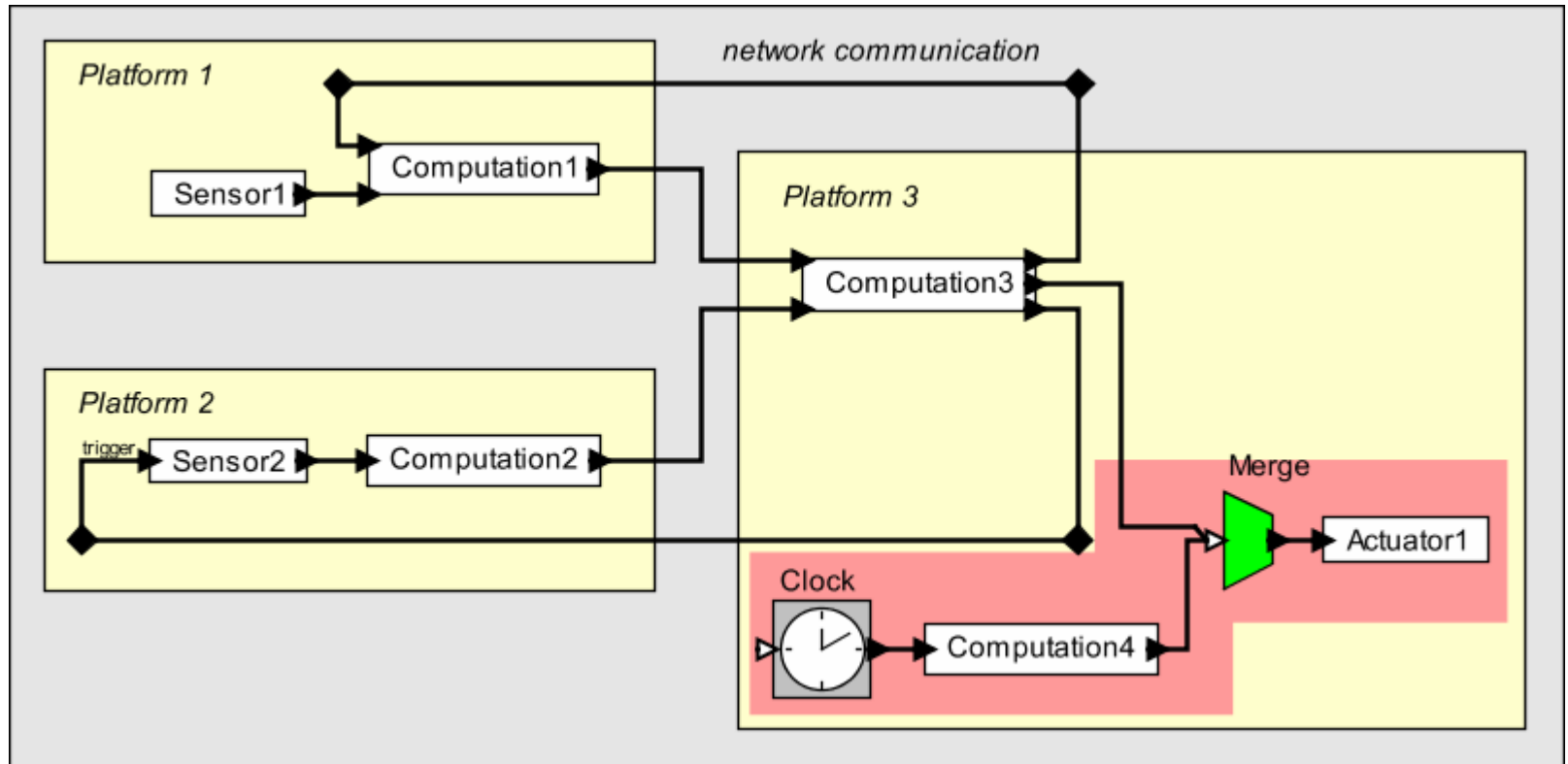
Schedulability is not violating these timing inequalities.



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

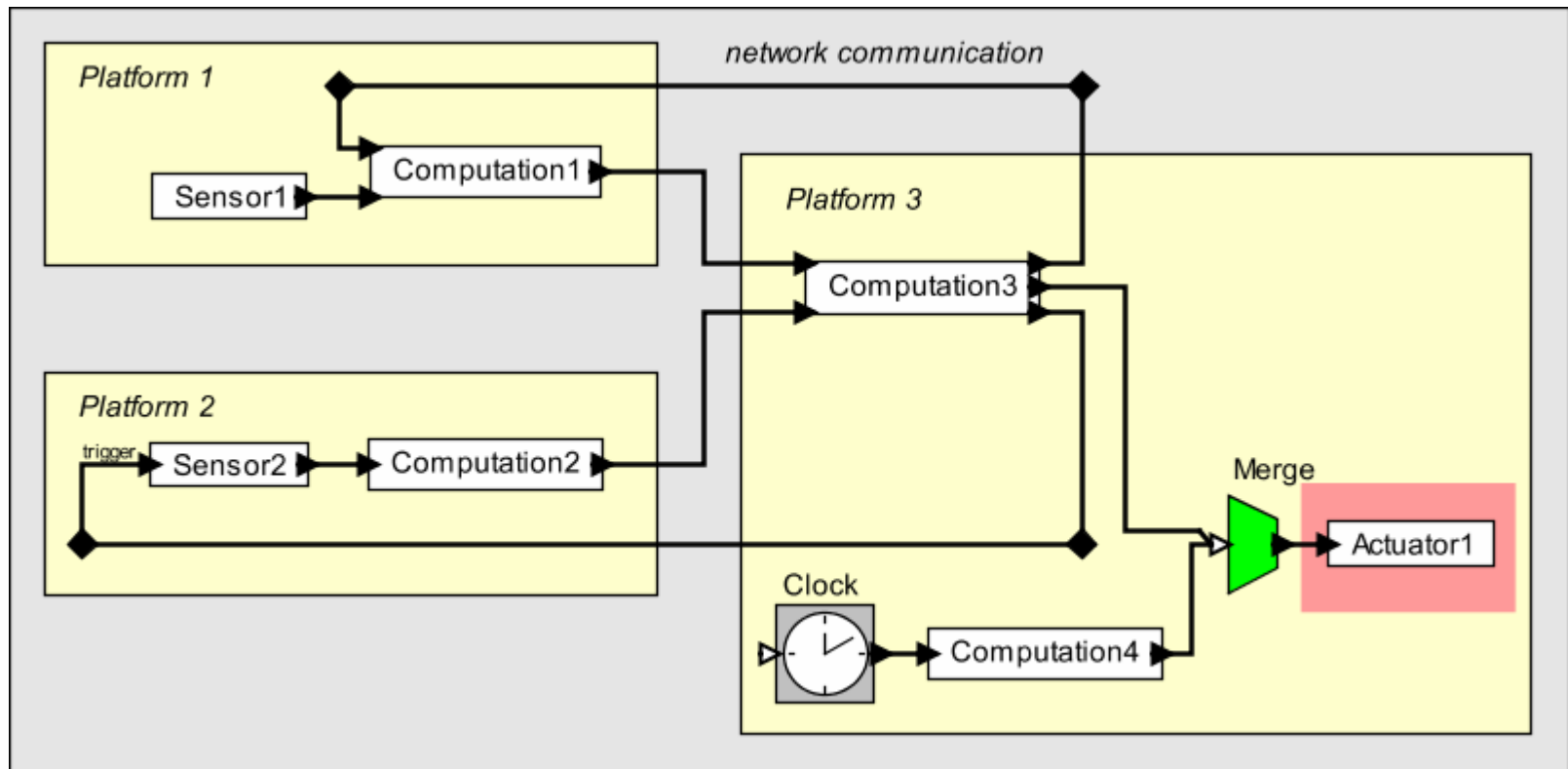
Conservative distributed DE (Chandy & Misra) would block actuation unnecessarily.



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

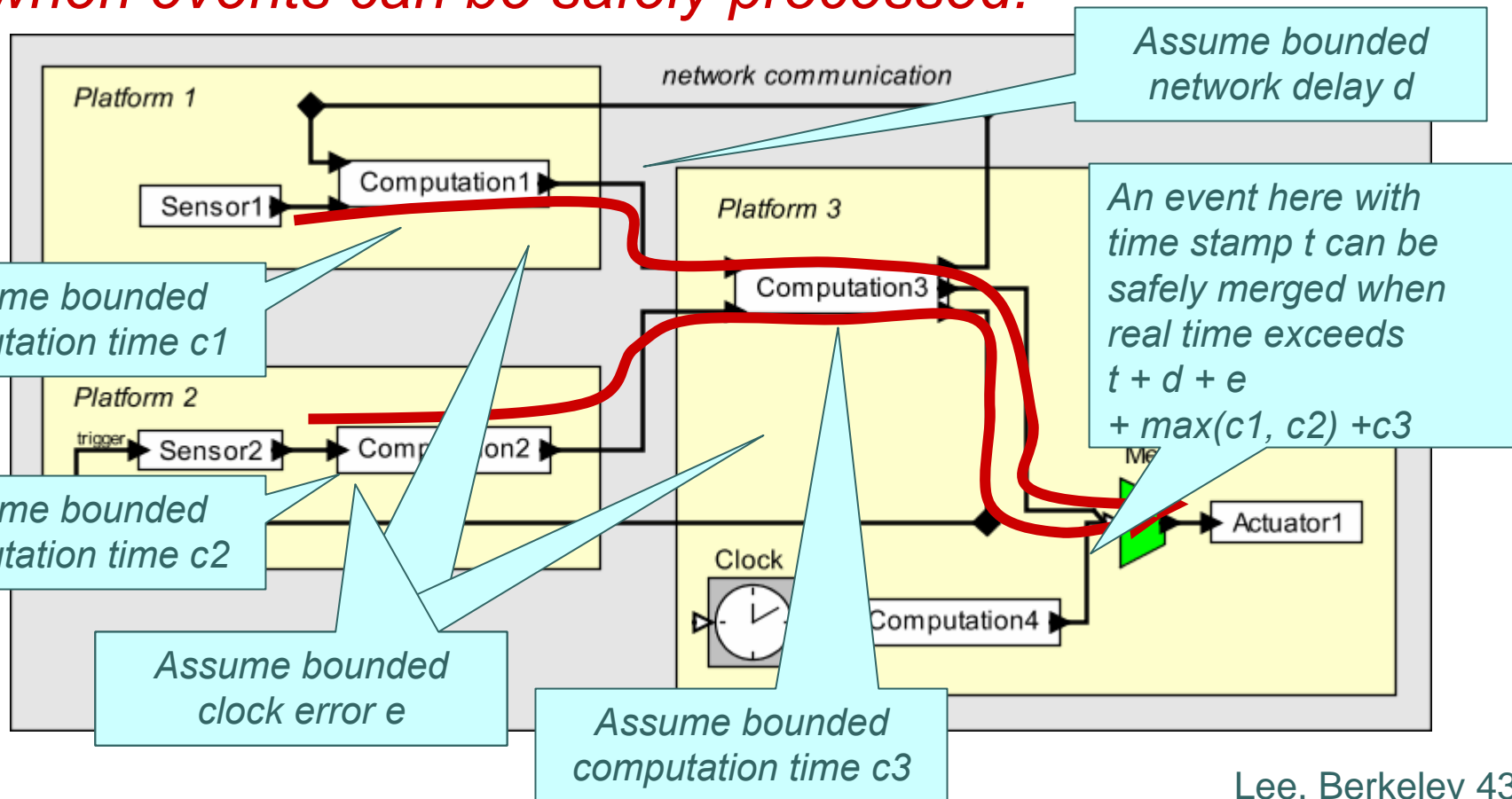
Optimistic distributed DE (Jefferson) would require being able to roll back the physical world.



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

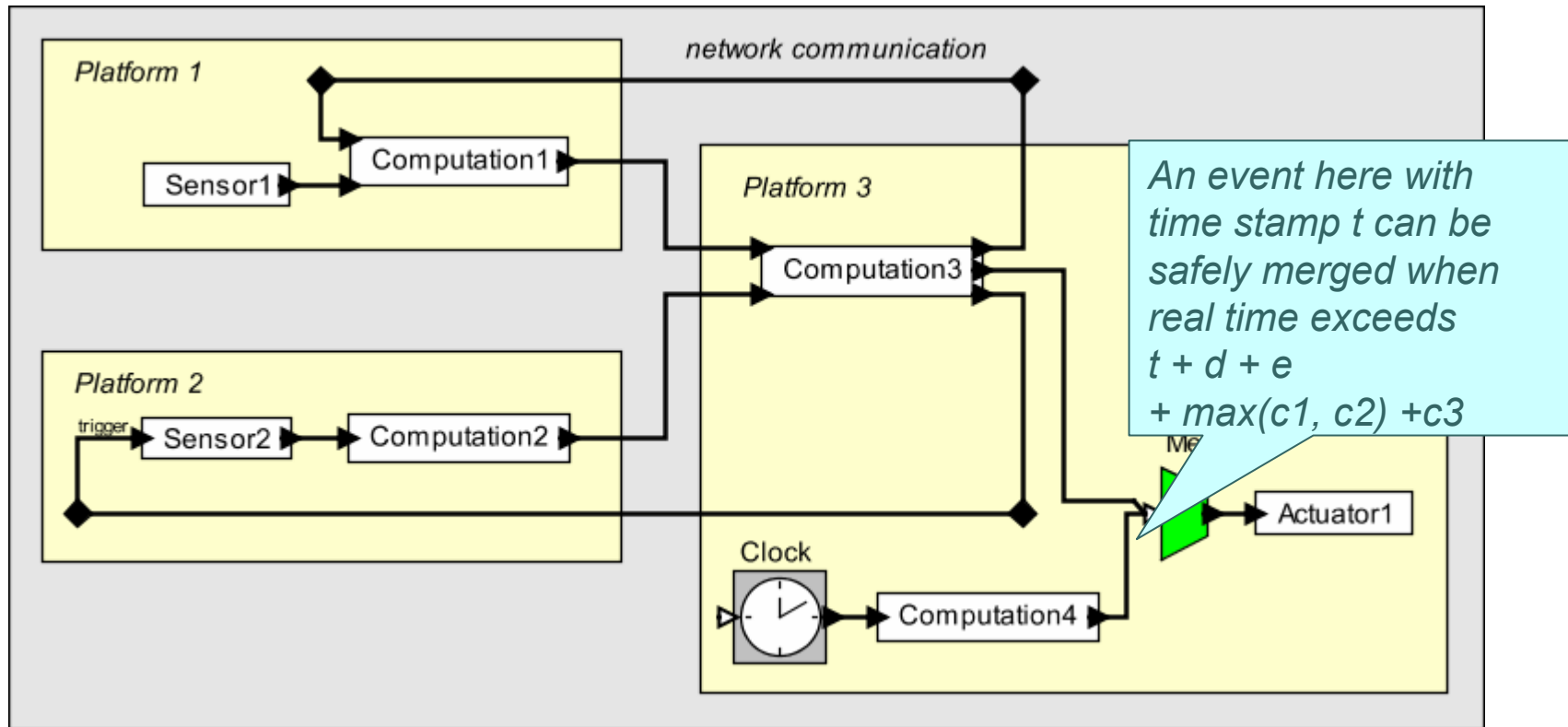
PTIDES uses static causality analysis to determine when events can be safely processed.



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

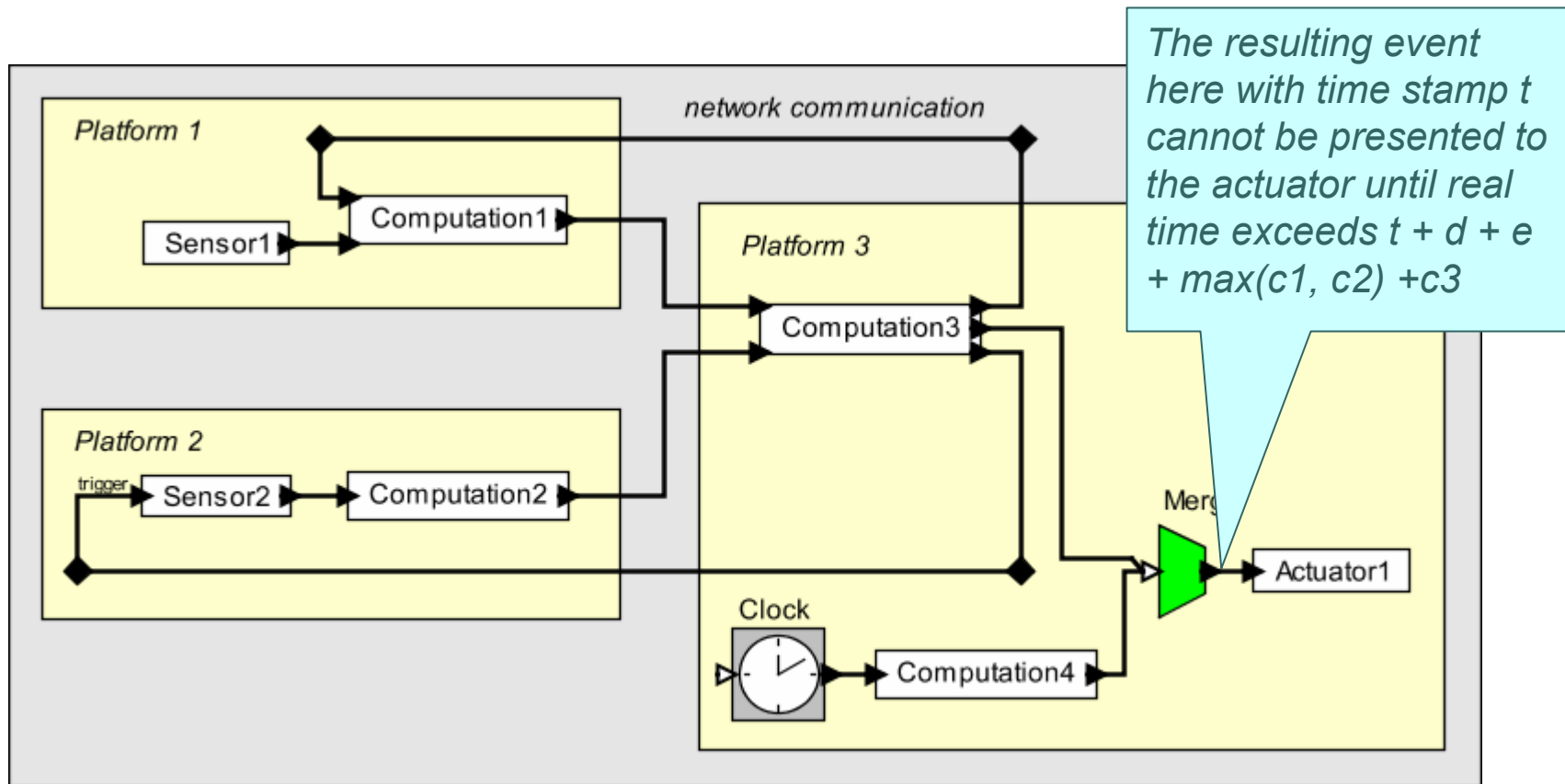
The execution model prevents remote processes from blocking local ones, and does not require backtracking.



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

However, this program is not schedulable!

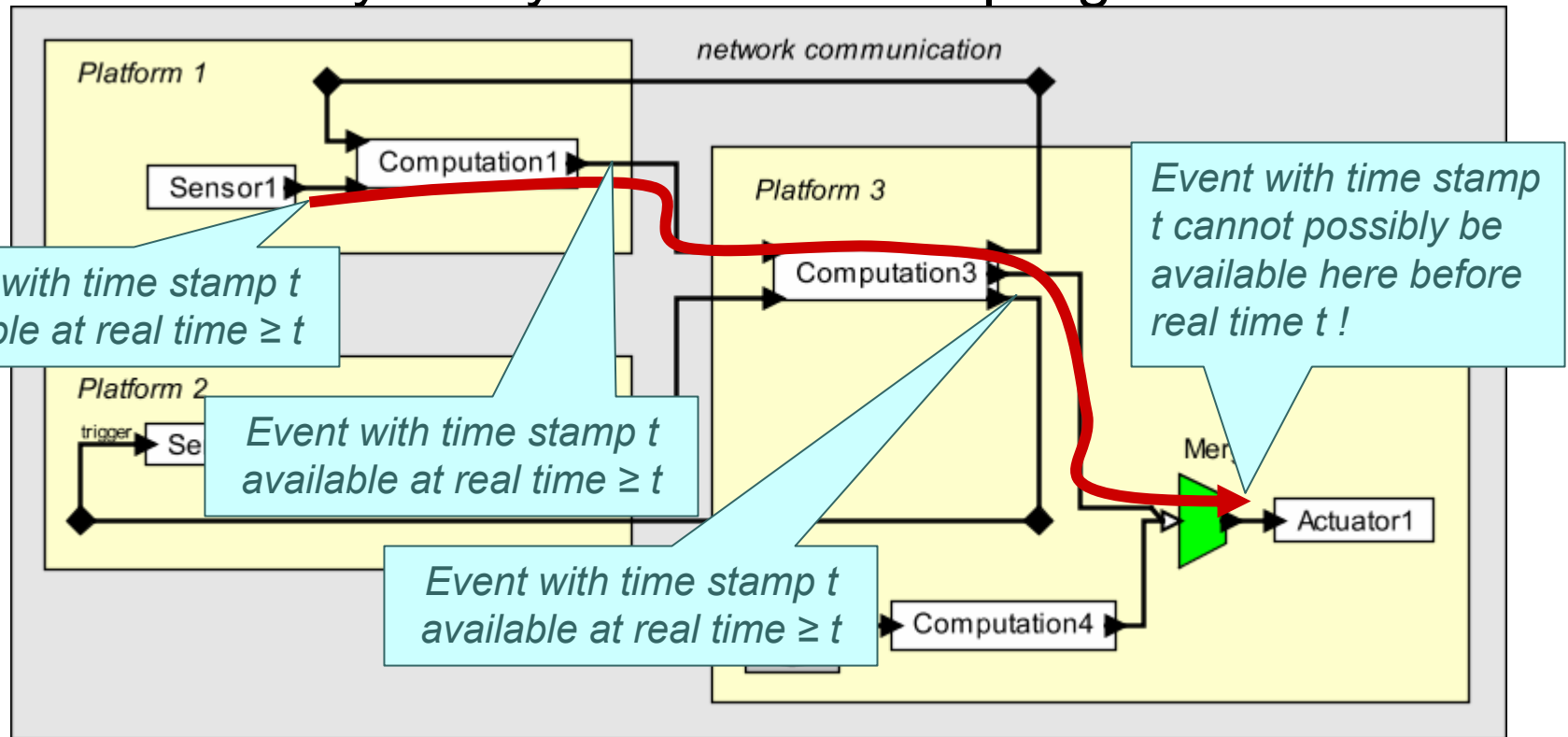


PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

Remote events also trigger real-time violations.

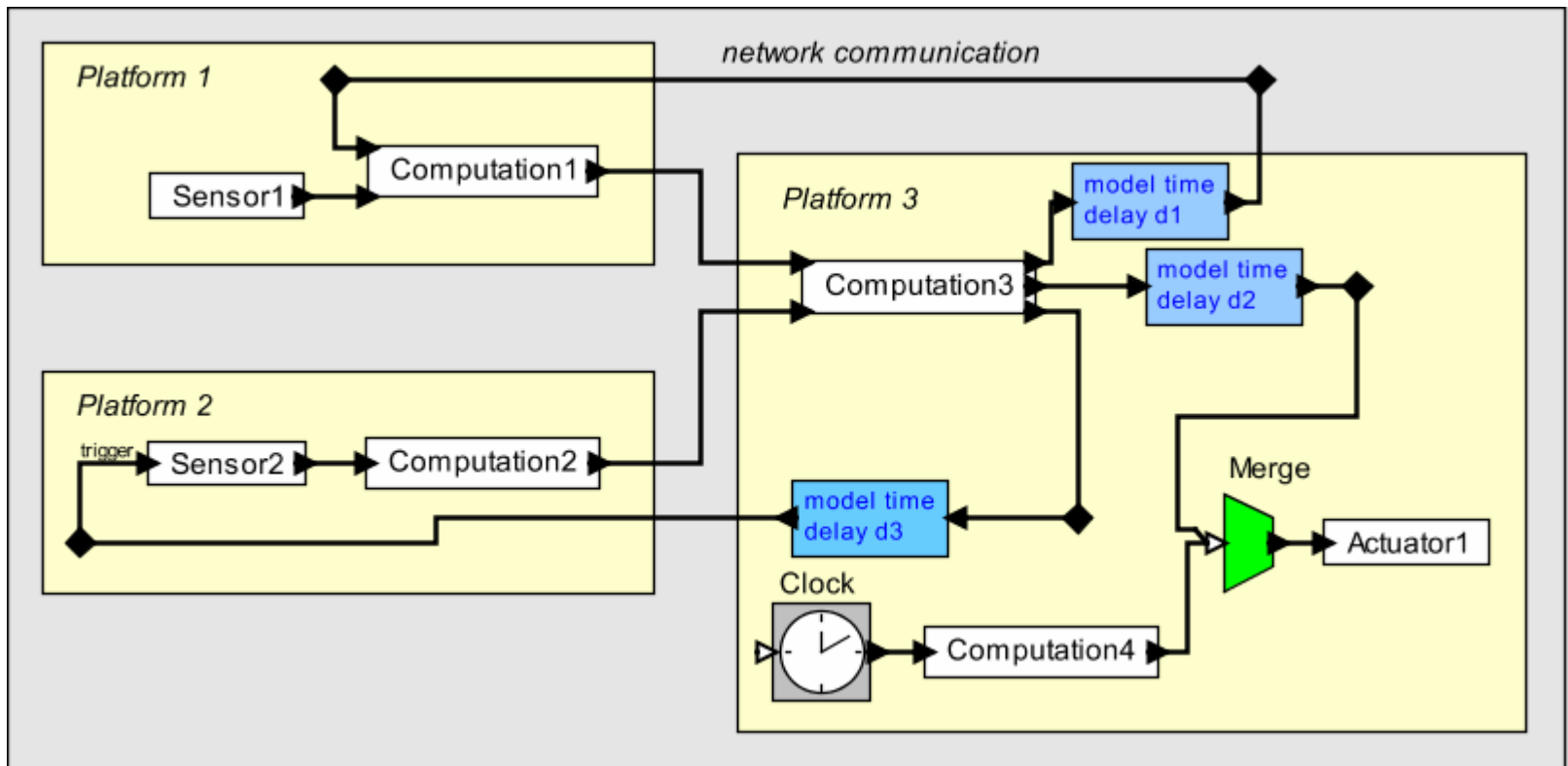
Schedulability analysis tells us the program is flawed.



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

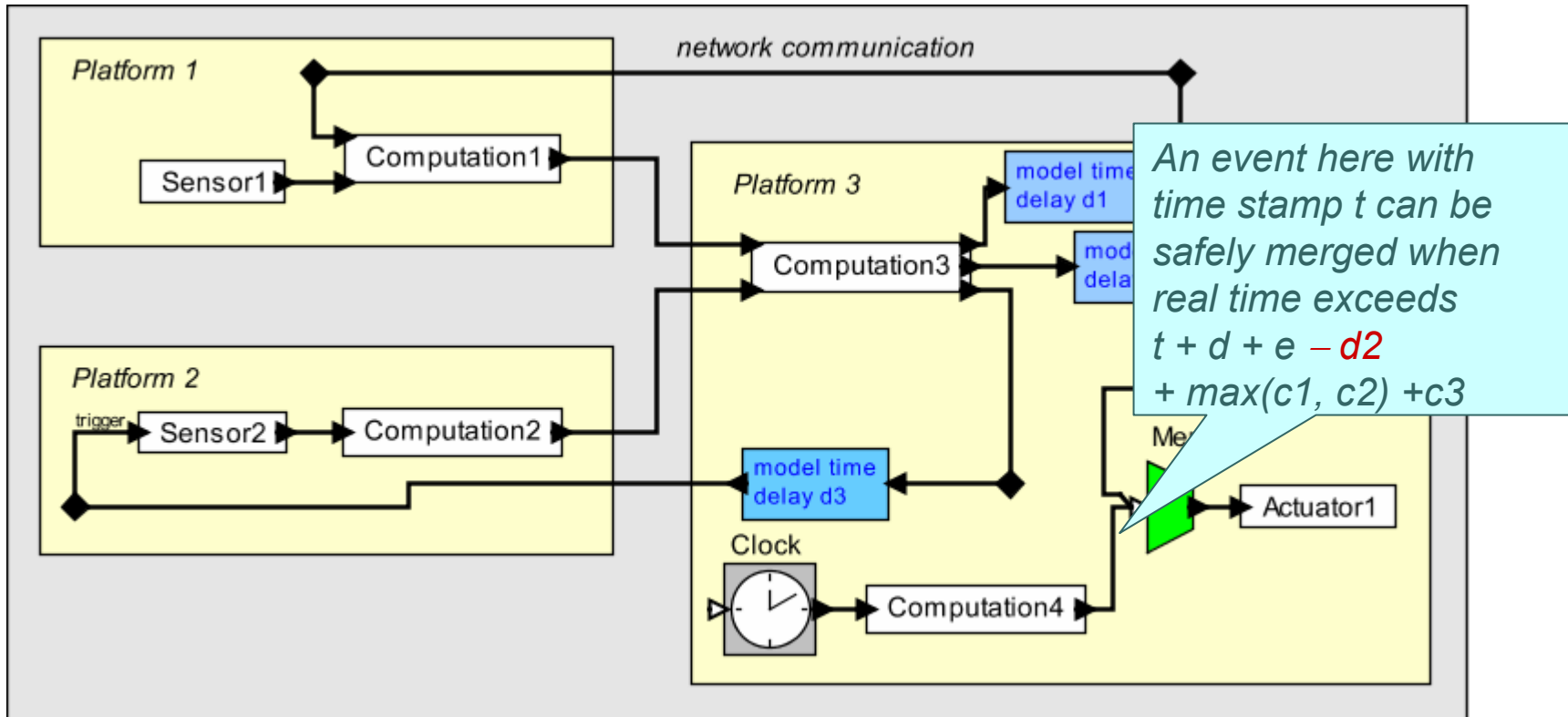
The program can be fixed with actors that increment the time stamps (model-time delays).



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

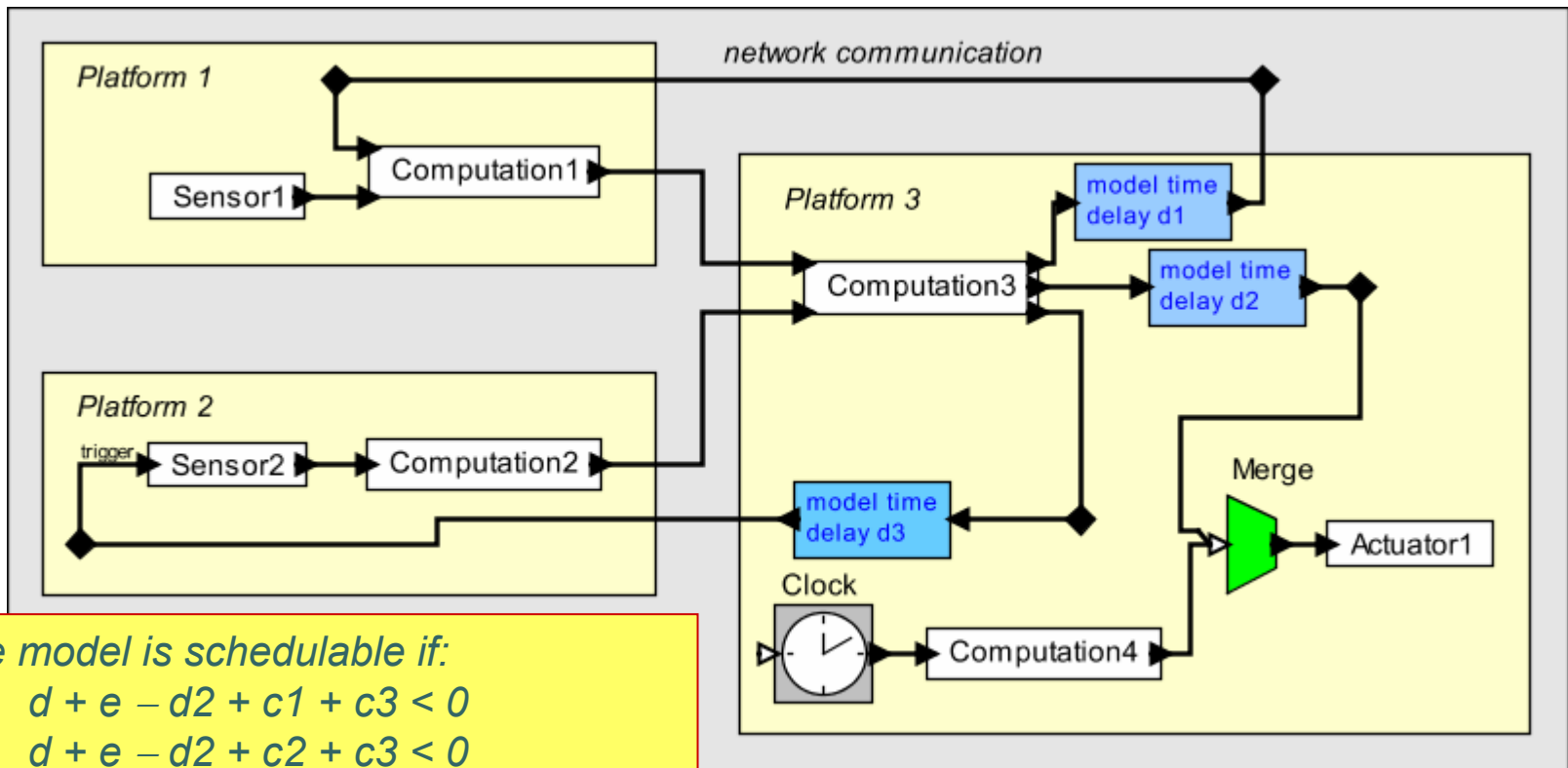
This relaxes scheduling constraints...



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

... and we can derive conditions for schedulability...



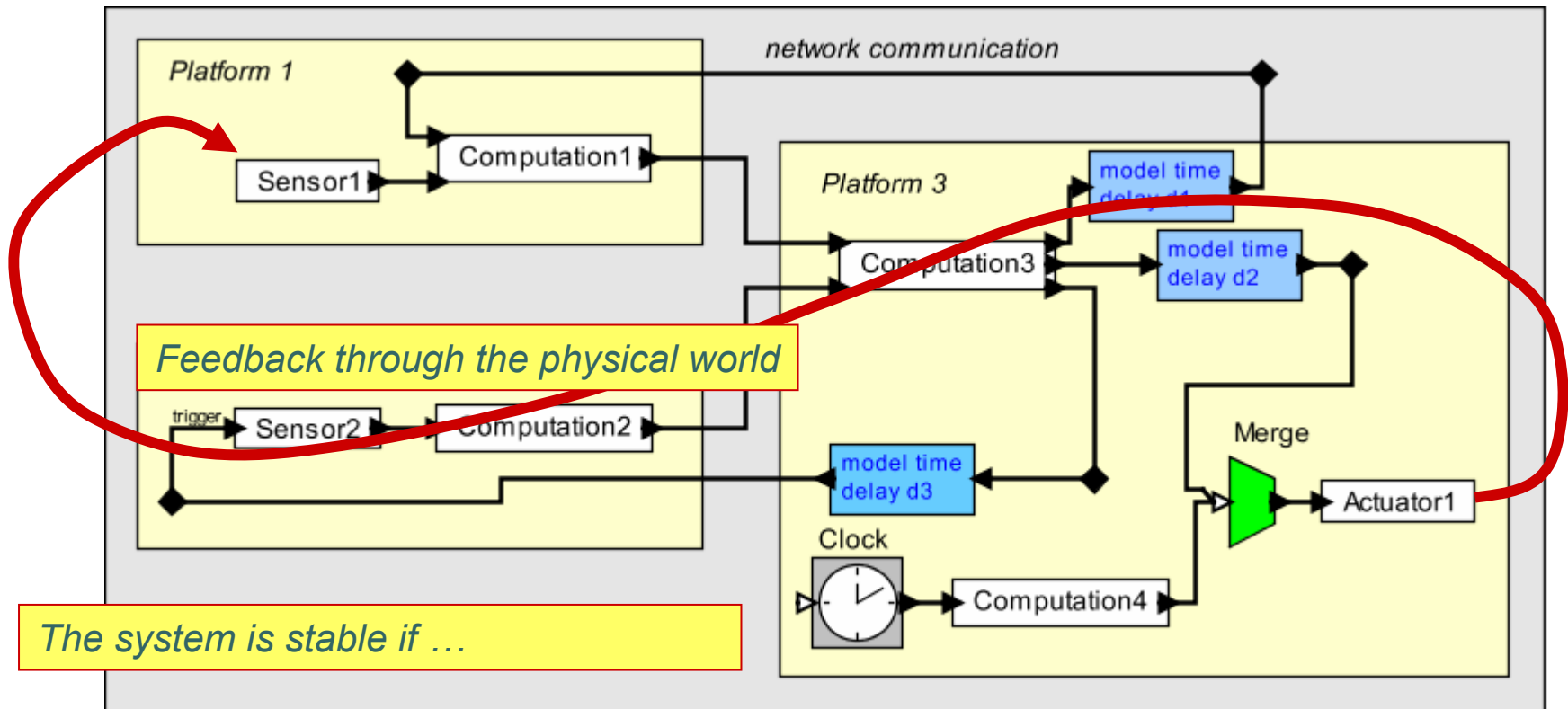
The model is schedulable if:

- 1) $d + e - d_2 + c_1 + c_3 < 0$
- 2) $d + e - d_2 + c_2 + c_3 < 0$
- 3) ...

PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

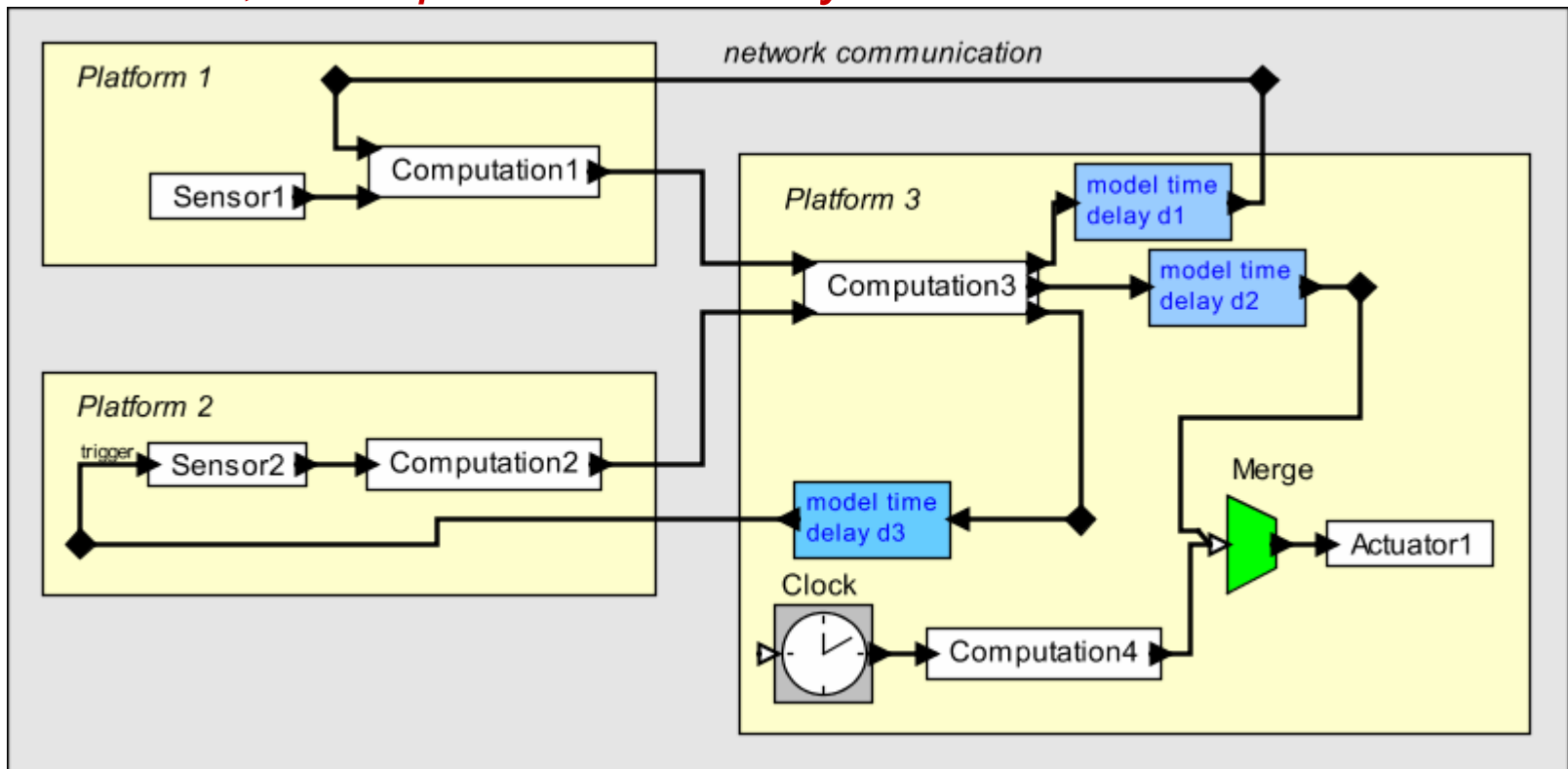
... and being explicit about time delays means that we can analyze control system dynamics...



PTIDES: Programming Temporally Integrated Distributed Embedded Systems

with Yang Zhao and Jie Liu

See “A Programming Model for Time-Synchronized Distributed Real-Time Systems”, Yang Zhao, Jie Liu, and Edward A. Lee, RTAS '07, to be presented Friday.





Is Truly Real-Time Computing Becoming Unachievable?

Yes!

But the problem is solvable:

Actor-oriented component architectures implemented in *coordination languages* that complement rather than replace existing languages (e.g. PTIDES).

and

PRET machines that deliver repeatable timing with efficient pipelining, memory hierarchy, and networking

See the Ptolemy Project for ongoing research addressing these problems: <http://ptolemy.org>