Lecture 10: Synchronous Reactive and Timed MoCs

Seyed-Hosein Attarzadeh-Niaki

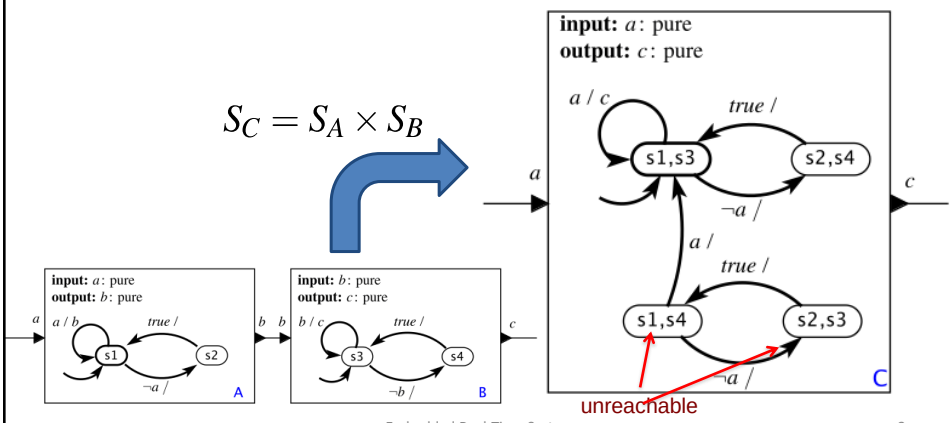
Some slides from Edward Lee and Tom Henzinger

Review

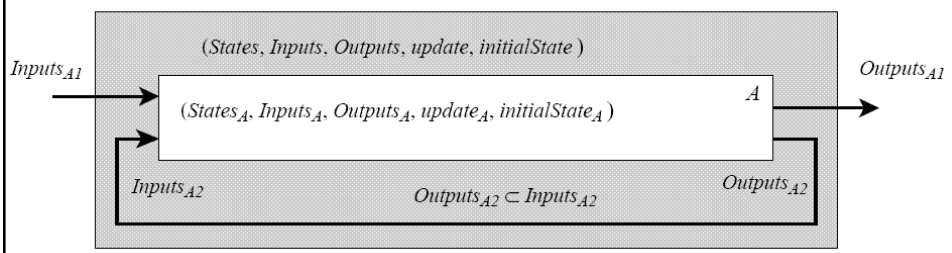
- Dataflow MoCs
- Analyzability of dataflow models
  - Scheduling
  - Buffer size
- Synchronous dataflow
- Other variants of Dataflow MoCs
  - HSDF
  - BDF
- Comparison of dataflow MoCs

Recap:
Synchronous Reactive Model of Computation-
Reactions are *Simultaneous* and *Instantaneous*

In this model, you must not think of machine A as reacting before machine B in physical time. If it did, the unreachable states would not be unreachable.



Feedback Composition

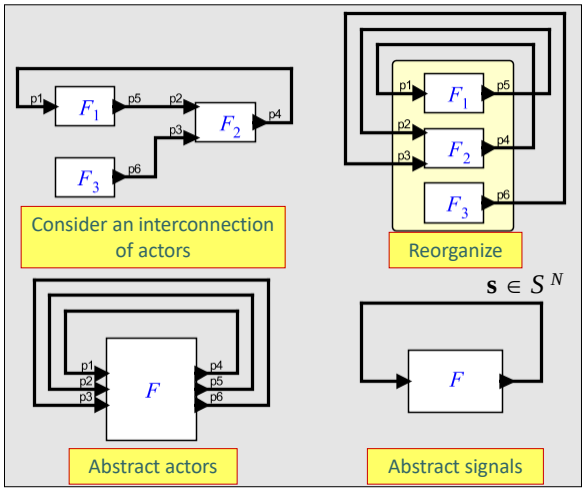


Turns out everything can be viewed as feedback composition…

Synchronous Composition of FSMs: Solutions

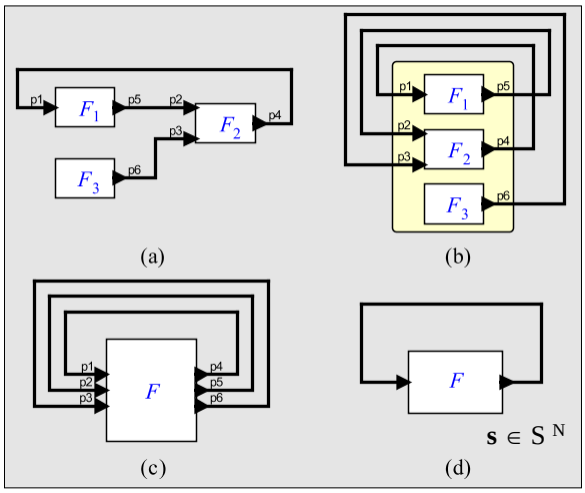
1. **Micro-steps**: a reaction is a sequence of elementary micro-steps
   - Adopted by VHDL, Verilog, StateCharts
2. **Acyclic**: zero-delay feedback loops are forbidden
   - Synchronous hardware design, Lustre
3. **Unique fixed-point**: reactions are solutions to fixed-point equations
   - More difficult to compile, Esterel
4. **Relation or constraint**: reactions as constraints
   - Multiple solutions as non-deterministic behavior, Signal

Observation: Any Composition is a Feedback Composition



The behavior of the system is generally a "fixed point."

Fixed Point Semantics



We seek an **s** ∈ S N that satisfies *F*(**s**) = **s**. Such an **s** is called a *fixed point*.

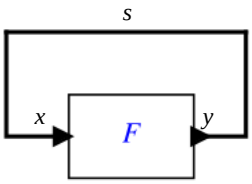
We would like the fixed point to exist and be unique and we would like a constructive procedure to find it.

It is the *behavior* of the system.

Well-Formed Feedback

At the n-th reaction, we seek $s(n) \in V_y \cup \{absent\}$ such that

$$s(n) = (f(n))(s(n))$$



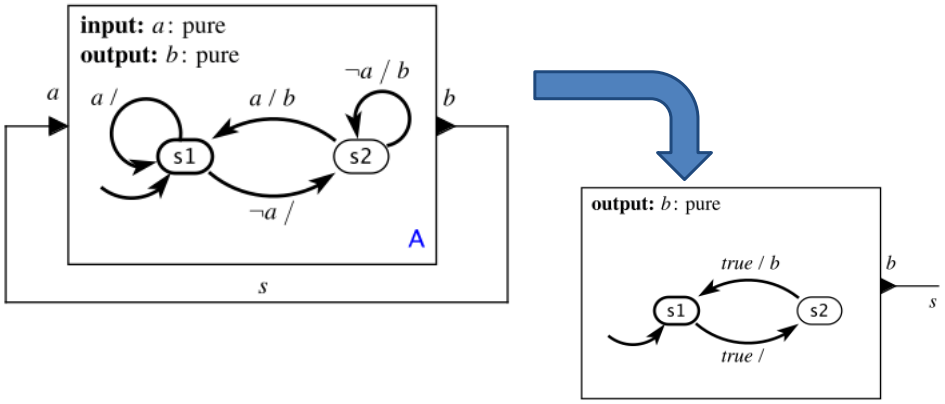
There are two potential problems:

1. It does not exist.
2. It is not unique.

In either case, we call the system **ill formed**. Otherwise, it is **well formed**.

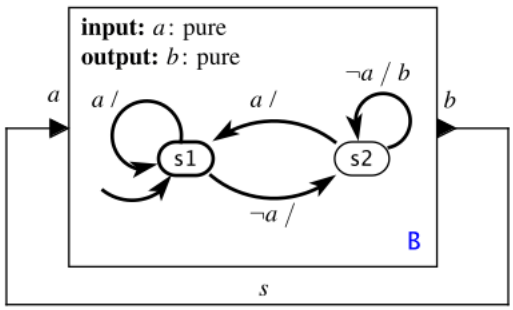
Note that if a state is not reachable, then it is irrelevant to determining whether the machine is well formed.

Well-Formed Feedback Example



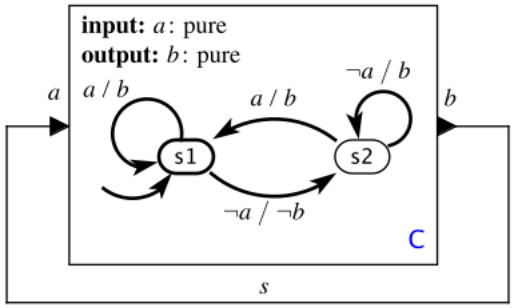
In state s1, we get the unique $s(n) = absent$.
In state s2, we get the unique $s(n) = present$.
Therefore, s alternates between *absent* and *present*.

Ill-Formed Example 1 (Existence)



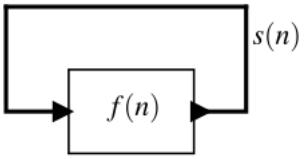
In state s1, we get the unique $s(n) = absent$.
In state s2, there is no fixed point.
Since state s2 is reachable, this composition is ill formed.

Ill-Formed Example 2 (Uniqueness)



In s1, both $s(n) = absent$ and $s(n) = present$ are fixed points.
In state s2, we get the unique $s(n) = present$.
Since state s1 is reachable, this composition is ill formed.

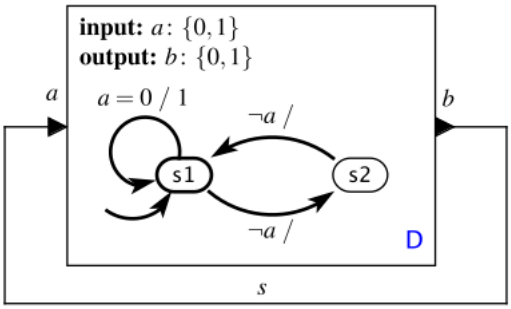
Constructive Semantics: Single Signal



1. Start with $s(n)$ *unknown*.
2. Determine as much as you can about $(f(n))(s(n))$.
3. If $s(n)$ becomes known (whether it is present, and if it is not pure, what its value is), then we have a unique fixed point.

A state machine for which this procedure works is said to be **constructive**.

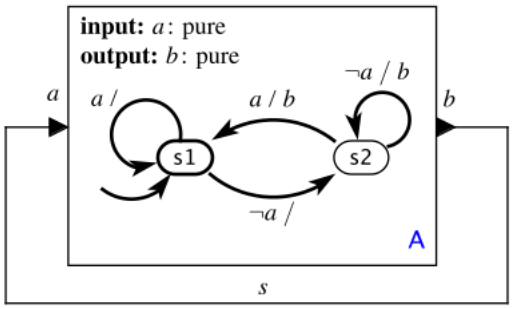
Non-Constructive Well-Formed State Machine



In state s1, if the input is unknown, we cannot immediately tell what the output will be. We have to try all the possible values for the input to determine that in fact $s(n) = absent$ for all n.

For non-constructive machines, we are forced to do **exhaustive search**. This is only possible if the data types are finite, and is only practical if the data types are small.

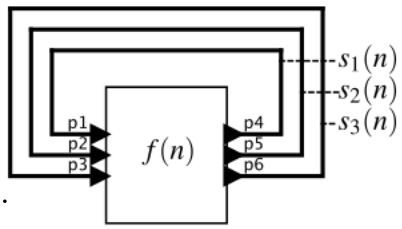
Must / May Analysis



For the above constructive machine, in state s1, we can immediately determine that the machine *may not* produce an output. Therefore, we can immediately conclude that the output is *absent*, even though the input is unknown.

In state s2, we can immediately determine that the machine *must* produce an output, so we can immediately conclude that the output is *present*.

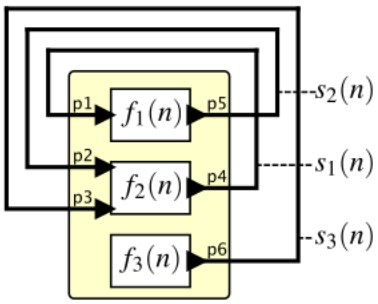
Constructive Semantics: Multiple Signals



1. Start with $s_1(n), \cdots, s_N(n)$ *unknown*.
2. Determine as much as you can about $(f(n))(s_1(n), \cdots, s_N(n))$.
3. Using new information about $s_1(n), \cdots, s_N(n)$, repeat step (2) until no information is obtained.
4. If $s_1(n), \cdots, s_N(n)$ all become known, then we have a unique fixed point and a constructive machine.

A state machine for which this procedure works is said to be **constructive**.

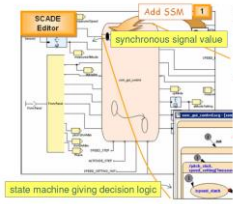
Constructive Semantics: Multiple Actors



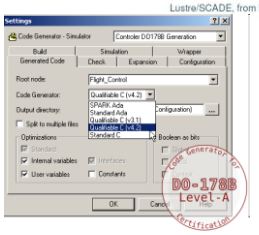
- Procedure is the same.

Application

- Tools based on the SR MoC has been very successful in the domain of safety-critical embedded systems
- *SCADE Suite, including the SCADE KCG Qualified Code Generator, is used by AIRBUS and many of its main suppliers for the development of most of the A380 and A400M critical on board software, and for the A340-500/600 Secondary Flying Command System, aircraft in operational use since August 2002.*
- François Pilarski, Systems Engineering Framework - Senior Manager Engineering,Systems & Integration Tests; Airbus France.



Lustre/SCADE, from http://www.esterel-technologies.



Timed Models of Computation

Time-Triggered Models

Discrete-Event (DE) Models

Continuous-Time Models

Why Do We Need A Notion of Time?

Event identification and generation

- State before vs. after the event

Event ordering

- Causal order (e.g., *a* may only have caused *b* if *a* happened before *b*)
- Temporal order (e.g., flight booking: who was first, *A* in VIE or *B* in LA?)

Coordination–coordinated action at specified time

- Possibly among multiple (a/)synchronous agents

Duration–measurement/control

- (e.g., X-ray: exposure time, video: gap between frames)

Modeling of **physical time**

- Comply to laws/dynamics of physics (*second*, physical time, real time)
- Read input, produce output "at the right time" (e.g., control loops)

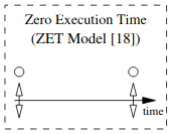
Time-Triggered Models: Time-Triggered Architecture

- Periodically trigger distributed computations according to a distributed clock
- Practical use in the design of safety-critical avionics and automotive systems
- Unlike the SR MoC, **computations take time**
  - Logical execution time

Real-Time Programming Abstractions

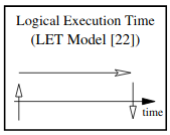
Zero Execution Time (ZET)

Foundation of synchronous reactive programming
Execution time of a program including reading input and writing output is assumed to be zero
Execution is correct if the program writes output before new input becomes available



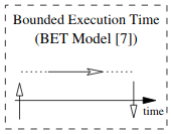
Logical Execution Time (LET)

Determines the (logical) time it takes from reading program input to writing program output regardless of the time it takes to execute the program
Assumes zero time for reading input and writing output
Deadline is not only an upper bound, but also a (logical) lower bound on execution time



Bounded Execution Time (BET)

Foundation of real-time scheduling theory
Bounds the execution time using deadline instead of abstracting execution times
Execution is incorrect if the program does not complete within the deadline

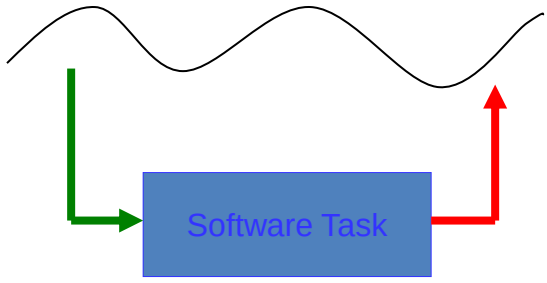


Kirsch, Christoph M., and Ana Sokolova. "The logical execution time paradigm." In *Advances in Real-Time Systems*, pp. 103-120. Springer, Berlin, Heidelberg, 2012.

Time-Triggered Models

- Inputs to the computation are provided at ticks of the global clock
- Outputs are not visible to other computations until the *next* tick of the global clock
  - *No interaction between the computations*
  - Concurrency difficulties such as race conditions do not exist
  - Computations are not (logically) instantaneous -> there are no difficulties with feedback
  - All models are constructive

The LET (Logical Execution Time) Programming Model

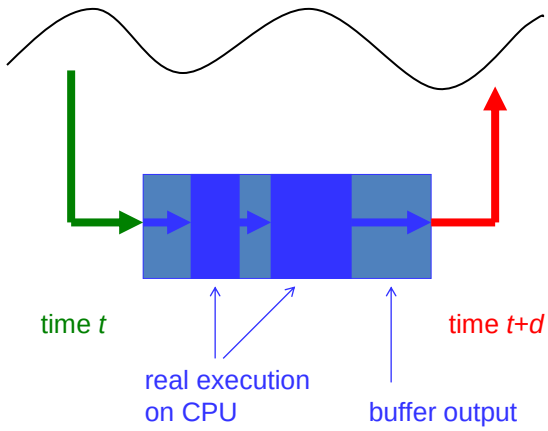


read sensor
input at time *t*

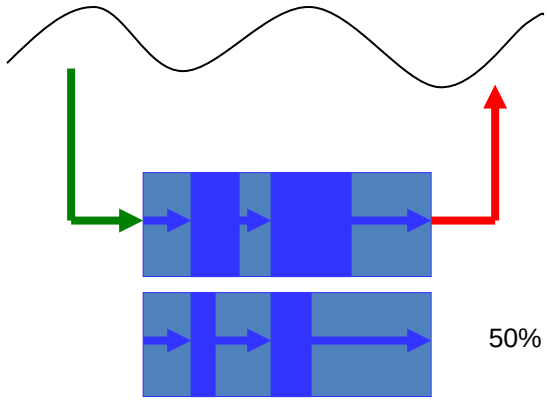
write actuator
output at time *t+d*,
for specified *d*

Examples: Giotto, TDL, …

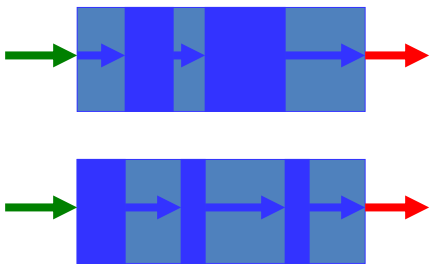
The LET (Logical Execution Time) Programming Model



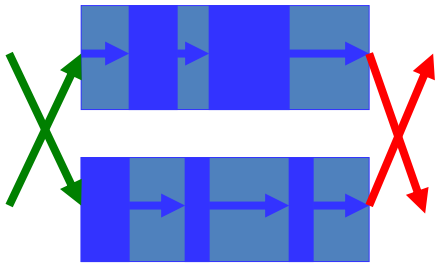
Portability



Composability

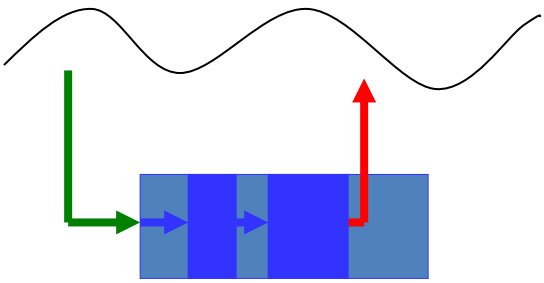


Determinism



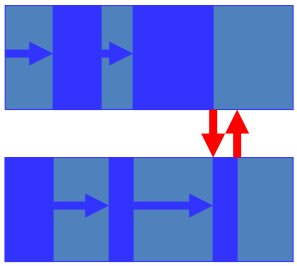
Timing predictability: minimal jitter
Function predictability: no race conditions

Contrast LET with Standard Practice

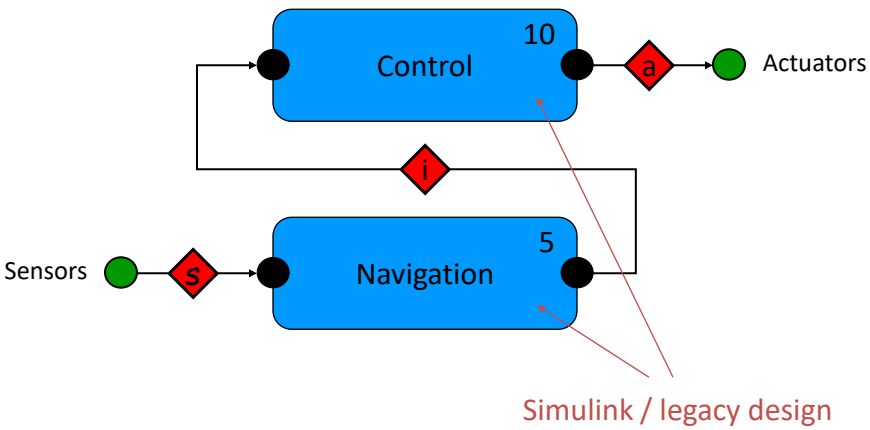


make output available
as soon as ready

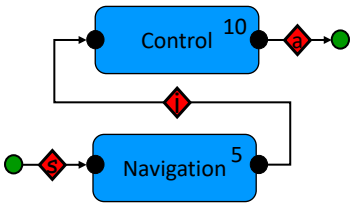
Contrast LET with Standard Practice



Simplified Helicopter Software



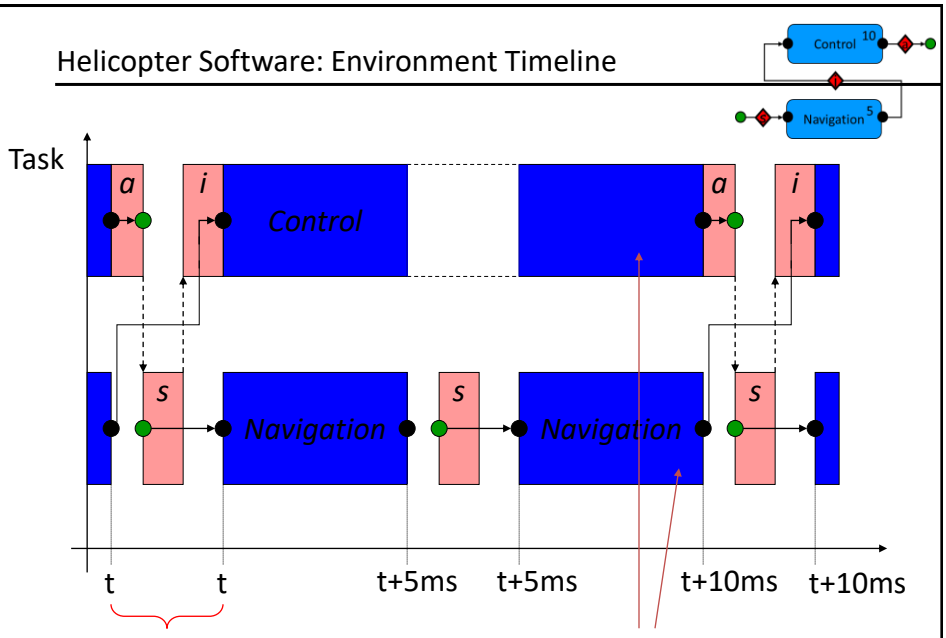
Helicopter Software: Giotto Syntax




```
...
mode Flight ( ) period 10ms
  {
     actfreq 1 do Actuator ( actuating ) ;

     taskfreq 1 do Control ( input ) ;
     taskfreq 2 do Navigation ( sensing ) ;
  }
...
```


Helicopter Software: Environment Timeline



Discrete-Event MoC

- Have been used for decades to build simulations for an enormous variety of applications
- DEVS is an extension of Moore machines that associates a non-zero lifespan with each state
- In DE Events are endowed with a time stamp

Discrete-Event MoC

- A DE model is a network of actors where actors
  - react to input events in time-stamp order
  - produce output events in time-stamp order
- To execute, an *event queue*, which is a list of events sorted by time stamp, is used
  - Select earliest event in the event queue and determining which actor should receive that event
  - The actor reacts ("fires") and can produce output events (placed in the queue)

Next Lecture

- Embedded Processors

SPARE SLIDES

Extending the SR MoC with Time: The Ptides MoC

- Define ticks of the synchronous clock to occur at a *time* in some model of time.
- Signals between actors now consist of *time-stamped events.*

[see Lee & Zheng, "Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems," EMSOFT, September 30, October 3, 2007, Salzburg, Austria]

Roots of the Idea

Using Time Instead of Timeout for Fault-Tolerant Distributed Systems

LESLIE LAMPORT
SRI International

A general method is described for implementing a distributed system with any desired degree of fault-tolerance. Instead of relying upon explicit timeouts, processes execute a simple clock-driven algorithm. Reliable clock synchronization and a solution to the Byzantine Generals Problem are assumed.

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*network operating systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*synchronization*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; *real-time systems*

General Terms: Design, Reliability

Additional Key Words and Phrases: Clocks, transaction commit, timestamps, interactive consistency, Byzantine Generals Problem

ACM Transactions on Programming Languages and Systems, 1984.

Ptides – A Robust Distributed DE MoC for IoIT Applications

in Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07), Bellevue, WA, United States.

A Programming Model for Time-Synchronized Distributed Real-Time Systems

Yang Zhao
EECS Department
UC Berkeley

Jie Liu
Microsoft Research
One Microsoft Way

Edward A. Lee
EECS Department
UC Berkeley

Abstract: Discrete-event (DE) models are formal system specifications that have analyzable deterministic behaviors. Using a global, consistent notion of time, DE components communicate via time-stamped events. DE models have primarily been used in performance modeling and simulation, where time stamps are a modeling property bearing no relationship to real time during execution of the model. In this paper, we extend DE models with the capability of relating certain events to physical time…

Google Spanner – A Reinvention

Google independently developed a very similar technique and applied it to distributed databases.

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google

Proceedings of OSDI 2012

PTIDES: Discrete-Event Semantics + Synchronized Clocks + Sensors and Actuators

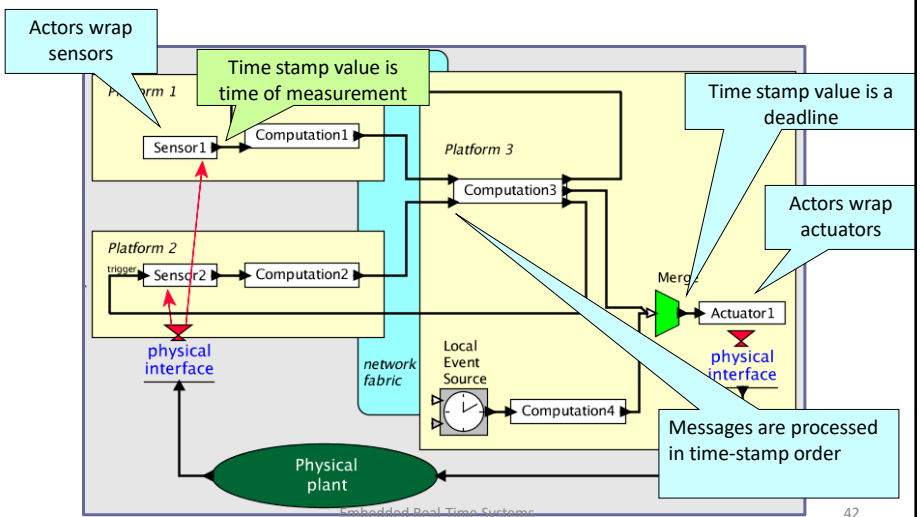
Time-stamped events that are processed in time-stamp order.

This MoC is widely used in simulation and HDLs.

Given time-stamped inputs, it is a deterministic concurrent MoC.

A few texts that use the DE MoC

Ptides: Time stamps bind to real time at sensors and actuators



Deterministic Distributed Real-Time

Assume bounds on:

- *clock synchronization error*
- *network latency*

then ***events are processed in time-stamp order*** at every component. If in addition we assume

- *bounds on execution time*

then events are delivered to actuators on time.

See http://chess.eecs.berkeley.edu/ptides

So Many Assumptions?

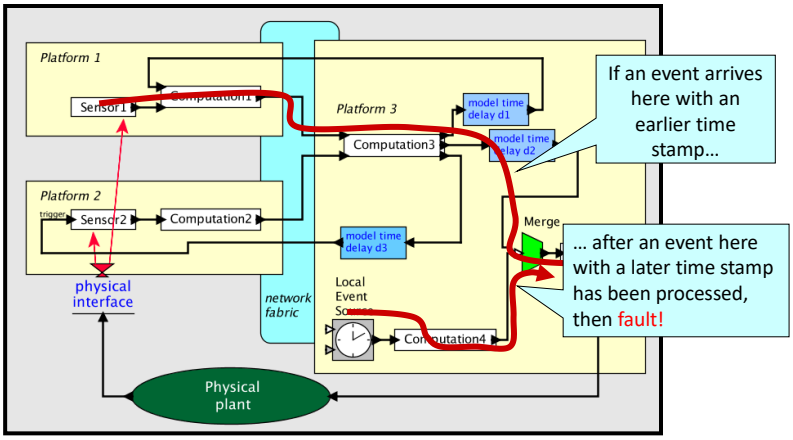
- All of the assumptions are achievable with today's technology, and are requirements anyway for hard-real-time systems.
- A Ptides model makes the requirements **explicit**.
- Violations of the requirements are detectable as out-of-order events *and can be treated as faults*.

Non-Synchronized Clocks

Handling Faults

A fault manifests as out-of-order events.

Occurrence of a fault implies one or more of the assumptions was violated.



Ptides

- High-precision clock synchronization will become ubiquitous.
- Networks will become able to “guarantee” bounded latencies.
- The time is right.