Actors Revisited for Time-Critical Systems

Edward A. Lee
Professor of the Graduate School
UC Berkeley

Invited Talk

Technical University of Eindhoven
Eindhoven, The Netherlands, March 21, 2019

With:
Marten Lohstroh
Martin Schoeberl
Andrés Goens
Armin Wasicek
Christopher Gill
Marjan Sirjani
Predictability requires determinacy and depends on timing, including execution times and network delays.
Motivation:
Some Questions of Interest

- What combinations of periodic, sporadic, arrival curve behaviors are manageable?
- How do execution times affect feasibility? How can we know execution times?
- How do we get repeatable and testable behavior even when communication is across networks?
- How do we specify, ensure, and enforce deadlines?
Actors, Loosely

Actors are concurrent objects that communicate by sending each other messages.
Data + Message Handlers

Some Realizations of Hewitt/Agha Actors

• Erlang [Armstrong, et al. 1996]
• Rebeca [Sirjani and Jaghoori, 2011]
• Akka [Roestenburg, et al. 2017]
• Ray [Moritz, et al. 2017]
• ...
An actor with simple operations on its state:

Actor Foo {
    int state = 1;
    handler double() {
        state *= 2;
    }
    handler increment(arg) {
        state += arg;
        print state;
    }
}
An actor that uses actor Foo:

```java
Actor Bar {
  handler main()
  {
    Foo x = new Foo();
    x.double();
    x.increment(1);
  }
}
```

Semantics is “send and forget.”
Actor Bar {
    handler main() {
        Foo x = new Foo();
        x.double();
        x.increment(1);
    }
}

What is printed?

Actor Foo {
    int state = 1;
    handler double() {
        state *= 2;
    }
    handler increment(arg) {
        state += arg;
        print state;
    }
}
Aside: Innovation in Ray

Messages can return “futures”:

```java
Actor Bar {
    handler main() {
        Foo x = new Foo();
        Future a = x.double();
        Future b = x.increment(1);
        print a.get() + b.get();
    }
}
```

Semantics is still “send and forget,” but later remember.

[Moritz, et al. 2017]
Pass-Through Actor

Baz: Given an actor of type Foo, send it “double”:

```scala
Actor Baz {
    handler pass(Foo x) {
        x.double();
    }
}
```
New Composition

What is printed?

Actor Bar {
    handler main()
    {
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.increment(1);
    }
}

Actor Baz {
    handler pass(Foo x)
    {
        x.double();
    }
}

Actor Foo {
    int state = 1;
    handler double()
    {
        state *= 2;
    }
    handler increment(arg)
    {
        state += arg;
        print state;
    }
}
Hewitt/Agha Actors are Not Predictable

Messages are handled in nondeterministic order.
One Solution: Analyze and Use Dependencies

Actor Bar {
    handler main() {
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.increment(1);
    }
}

Actor Baz {
    handler pass(Foo x) {
        x.double();
    }
}

Actor Foo {
    int state = 1;
    handler double() {
        state *= 2;
    }
    handler increment(arg) {
        state += arg;
        print state;
    }
}

But how? Where is the dependence graph?
One Solution: Analyze and Use Dependencies

And what if the dependence graph is data dependent?

Actor Bar {
    handler main() {
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.increment(1);
    }
}

Actor Baz {
    handler pass(Foo x) {
        if (something) {
            x.double();
        }
    }
}

Actor Foo {
    int state = 1;
    handler double() {
        state *= 2;
    }
    handler increment(arg) {
        state += arg;
        print state;
    }
}
Part 1 of our Solution: Ports

Instead of referring to other actors, an actor refers to its own ports.

reactor Bar {
  output double, increment;
  reaction main() {
    double.send();
    increment.send(1);
  }
}

reactor Baz {
  input in;
  output out;
  reaction (in) {
    send(out);
  }
}
Input ports do not look much different from ordinary message handlers.

```java
reactor Foo {
  input double, increment;
  int state = 1;
  reaction(double) {
    state *= 2;
  }
  reaction(increment) {
    state += increment;
    print state;
  }
}
```
Part 2 of our Solution: Hierarchy

```java
composite Top {
    reaction main() {
        Foo x = new Foo();
        Bar y = new Bar();
        Baz z = new Baz();
        connect(y.double, z.in);
        connect(y.increment, x.increment);
        connect(z.out, x.double);
    }
}
```
Part 3 of our Solution: Scheduling

```java
composite Top {
    reaction main() {
        Foo x = new Foo();
        Bar y = new Bar();
        Baz z = new Baz();
        connect(y.double, z.in);
        connect(y.increment, x.increment);
        connect(z.out, x.double);
    }
}
```

Scheduling becomes especially interesting when production or consumption of messages is data dependent.

Ensure that Baz completes before Foo’s handlers are invoked.
Some Strategies

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)
Dataflow

- Computation Graphs [Karp, 1966]
- Dataflow [Dennis, 1974]
- Dynamic dataflow [Arvind, 1981]
- Structured dataflow [Matwin & Pietrzykowski 1985]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow and LabVIEW [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Dataflow synchronous languages [Lustre, Signal, 1980’s]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- ...

Jack Dennis
An actor with no inputs can fire at any time.
An actor with inputs has to specify at all times how many tokens it needs on each input in order to fire.
An actor inputs has to specify at all times how many tokens it needs on each input in order to fire.

When it fires, each reaction is invoked in a deterministic order.

[Lee & Matsikoudis, 2009]
When the firing rules and production patterns are static integer constants, then a lot of analysis and optimization is possible.

[Lee & Messerschmitt, 1986]
If execution times are also known, then throughput and latency bounds are derivable and optimal scheduling is possible (albeit intractable).

[Lee & Messerschmitt, 1986]
Dataflow Scheduling with Dynamic Firing Rules

reactor Baz {
  input in;
  output out;
  reaction (in) {
    if (something) {
      send (out);
    }
  }
}

What should be the firing rule for Foo?
Buck [1993] showed that scheduling problems in general are undecidable in this framework.

Associate a symbolic variable with production and consumption parameters. Solve the scheduling problem symbolically.
[]Buck and Lee, 1993]
Various Dataflow Variants that Remain Decidable

- Cyclostatic dataflow [Lauwereins 1994]
- Parameterized dataflow [Bhattacharya & Bhattacharyya, 2001]
- Structured dataflow [Thies, 2002]
- Scenario-aware dataflow [Theelen, Geilen, Basten, et al. 2006]
- Reconfigurable dataflow [Fradet, Girault, et al., 2019]
A state machine governs the switching between production/consumption patterns and also execution times.

[Theelen, Geilen, Basten, et al. 2006]
Some Strategies

• Dataflow (DF)
• Process Networks (PN)
• Synchronous/Reactive (SR)
• Discrete Events (DE)
A Different Solution: Blocking Reads

In Kahn Process Networks (KPN), every actor is a process that blocks on reading inputs until data is available.

KPNActor Foo {
    input double, increment;
    int state = 1;
    while (true) {
       read(double);
       state *= 2;
       x = read(increment);
       state += x;
       print state;
    }
}

[Kahn, 1974] [Kahn and MacQueen, 1977]
Blocking reads have trouble with data-dependent flow patterns

```
KPNActor Baz {
  input in;
  output out;
  while(true) {
    read(in);
    if (something) {
      send(out);
    }
  }
}

KPNActor Foo {
  input double, increment;
  int state = 1;
  while(true) {
    read(double);
    state *= 2;
    x = read(increment);
    state += x;
    print state;
  }
}
```
Blocking reads have trouble with data-dependent flow patterns

KPNActor Baz {
    input in;
    output out;
    while (true) {
        read (in);
        if (something) {
            send (out);
        }
    }
}

KPNActor Foo {
    input double, increment;
    int state = 1;
    while (true) {
        if (something) {
            read (double);
            state *= 2;
        }
        x = read (increment);
        state += x;
        print state;
    }
}
Actor Baz {
    input in;
    output out;
    handler in() {
        if (something) {
            out.send();
        } 
    }
}

Actor Foo {
    input double, increment;
    int state = 1;
    while(true) {
        if (something) {
            read(double);
            state *= 2;
        }
        x = read(increment);
        state += x;
        print state;
    }
}
Some Strategies

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)
An Alternative Approach to Coordination

Make the notion of the “absence” of a message as meaningful as its presence.
In the synchronous/reactive approach, there is a conceptual global “clock,” and on each “tick” of this clock, a connection either has a well-defined value or is “absent.” Each actor realizes a time-varying function mapping inputs to outputs.

[Benveniste & Berry, 1991]
At each tick of the clock, the job of the execution engine is to find a valuation $s$ for all signals such that $F(s) = s$.

This is called a fixed point of the function $F$. A theory of partial orders guarantees existence and uniqueness.

[Edwards and Lee, 2003]
Physically asynchronous, logically synchronous (PALS)

[Sha et al., 2009]
Some Strategies

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)
Discrete-Event Languages

DE is a generalization of SR, where there is a notion of “time between ticks.”

WARNING: immediately have (at least) two time lines: logical time and physical time(s).

[Lee & Zheng, 2007]
Finally! We can talk about the motivating example.

Sporadic events are assigned a time stamp based on the local physical-time clock.

Computations have logically zero delay.

Every reactor handles events in time-stamp order. If time-stamps are equal, events are “simultaneous”.

Actuators can have a deadline $D$. An input with time stamp $t$ is required to be delivered to the actuator before the local clock hits $t + D$. 
Simple, Single-Machine Realization

- Sort reactors topologically based on precedences.
- Global notion of “current time” $t$.
- Event queue containing future events.
- Choose earliest time stamp $t'$ on the queue.
- Wait for the real-time clock to match $t'$.
- Execute reactors in topological sort order.

When a sporadic sensor triggers (or an asynchronous event like a network message arrives), assign a time stamp based on the local physical-time clock.
This example has a pre-defined latency from physical sensing to physical actuation, thereby delivering a closed-loop deterministic cyber-physical model.
Classical real-time systems scheduling and execution-time analysis determines whether the specification can be met.

[Buttazzo, 2005]  [Wilhelm et al., 2008]
Precision-timed (PRET) machines deliver deterministic clock-cycle-level repeatable timing with no loss of performance on sporadic workloads.

[Edwards & Lee, 2007] [Lee et al., 2017]
If the PeriodicSource does not depend on physical inputs, then pre-computing (logical time ahead of physical time) becomes possible, based on dependence analysis.
Models of Time: Superdense Time

At each tag, the signal has exactly one value.

At each time point, the signal has a sequence of values.

\[ \mathbf{v}: \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}^3 \]

Initial value: \( \mathbf{v}(t_i, 0) = 0 \)

Intermediate value: \( \mathbf{v}(t_i, 1) = K \)

Final value: \( \mathbf{v}(t_i, n) = 0, \quad n \geq 2 \)

[Lee & Zheng, 2005]  [Maler, Manna, Pnuelli, 92]
The red arrows indicate value changes between tags, which correspond to discontinuities. Signals are continuous from the left and continuous from the right at points of discontinuity.
When is this “safe to process”?

When \( \tau \geq T + W_1 + E + N \), where

- \( \tau \) is the local physical clock time
- \( W_1 \) is worst-case execution time
- \( E \) is the bound on the clock synchronization error
- \( N \) the bound on the network delay

[Zhao et al., 2007]
[Edison et al., 2012]
[Corbett et al., 2012]
Networked Scheduling: PTides

Will the deadline at ActuatorA be met? Yes if $D + d_1 \geq T + W_1 + E + N + W_2$

[Zhao et al., 2007]
[Edison et al., 2012]
[Corbett et al., 2012]
Imposing deadlines on network interfaces decouples the real-time analysis problem. Each execution platform can be individually verified for meeting deadlines. E.g., $I_2 \geq W_2$, $D_2 \geq W_2$, $D_3 \geq D_2 + W_3$, ...

[Zhao et al., 2007]
Other Issues: Feedback

- Fixed-point semantics
- Causality loops
- Superdense time
- ...
Conclusion

- Hewitt/Agha actors are nondeterministic
- Some solutions:
  - Dataflow
  - Process networks
  - Synchronous/Reactive models
  - Discrete-Event
- Reactors are actors revisited with DE semantics

Pseudo code shown is based on Lingua-Franca.
Invited: Actors Revisited for Time-Critical Systems

Marten Lohstroh  
UC Berkeley, USA

Martin Schoebert  
TU Denmark, Denmark

Andrés Goens  
TU Dresden, Germany

Armin Wasicek  
Avast, USA

Christopher Gill  
Washington Univ., St. Louis, USA

Marjan Sirjani  
Mälardalen Univ., Sweden

Edward A. Lee  
UC Berkeley, USA

ABSTRACT
Programming time-critical systems is notoriously difficult. In this paper we propose an actor-oriented programming model with a semantic notion of time and a deterministic coordination semantics based on discrete events to exercise precise control over both the computational and timing aspects of the system behavior.

2 ACTORS
The actor model was introduced by Hewitt [6] in the early 70s. Since then, the use of actors has proliferated in programming languages [1, 2], coordination languages [14, 15], distributed systems [7, 11], and simulation and verification engines [13, 17]. Actors have much in common with objects—a paradigm focused
Many dataflow papers: [https://ptolemy.berkeley.edu/publications/dataflow.htm](https://ptolemy.berkeley.edu/publications/dataflow.htm)

References

References


