Actor Networks

Edward A. Lee
Robert S. Pepper Distinguished Professor
Chair of EECS
UC Berkeley

Invited Talk
Workshop Foundations and Applications of
Component-based Design
Seoul, Korea, Oct. 26, 2006
Key Concepts in Model-Based Design

- Specifications are executable models.
- Models are composed to form designs.
- Models evolve during design.
- Deployed code is generated from models.
- Modeling languages have formal semantics.
- Modeling languages themselves are modeled.

For general-purpose software, this is about
- Object-oriented design

For embedded systems, this is about
- Time
- Concurrency
What We Have Learned

Embedded systems demand a different approach to computation.
Instead of a Program Specifying…

\[ f : \{0,1\}^* \rightarrow \{0,1\}^* \]

… a (partial) function from bit sequences to bit sequences …
A Program Should Specify

\[ f : \left[ T \rightarrow \{0,1\}^* \right]^P \rightarrow \left[ T \rightarrow \{0,1\}^* \right]^P \]

...where \( T \) is a (partially) ordered set representing time, precedence ordering, causality, synchronization, etc.
This Leads to What We Call Actor-Oriented Component Composition

If actors are functions on signals, then the nontrivial part of this is feedback.

Some of the Possible Models of Computation:

- Time-Triggered
- Discrete Events
- Dataflow
- Rendezvous
- Synchronous/Reactive
- Continuous Time
- Mixtures of the above
- …
Examples of Actor-Oriented “Languages”

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

The semantics of these differ considerably, but all can be modeled as

\[ f: [T \rightarrow \{0,1\}^*]^P \rightarrow [T \rightarrow \{0,1\}^*]^P \]

with appropriate choices of the set \( T \).

Many of these are domain specific.

Many of these have visual syntaxes.
The Catch…

\[ f : [T \rightarrow \{0,1\}^*]^P \rightarrow [T \rightarrow \{0,1\}^*]^P \]

- This is not what (mainstream) programming languages do.
- This is not what (mainstream) software component technologies do.
- This is not what (most) semantic theories do.

Let’s deal with this one first…
How much Theory is Based on 

\( f: \{0,1\}^* \rightarrow \{0,1\}^* \) ?

- Effectively computable functions [Turing, Church]
- Operational semantics as sequences of transformations of state [Various]
- Denotational semantics as functions mapping a syntax into a function that maps state into state [Winskel]
- Equivalence as bisimulation [Milner]
- Verification as model checking [Various]
- …

See [Lee, FORMATS 2006] for further discussion of this.
Our Approach to a More Suitable Theory: The Tagged Signal Model

[Lee & Sangiovanni-Vincentelli, 1998]

- A set of values $V$ and a set of tags $T$
- An event is $e \in T \times V$
- A signal $s$ is a set of events. I.e. $s \subset T \times V$
- A functional signal is a (partial) function $s: T \rightarrow V$
- The set of all signals $S = 2^{T \times V}$

Related models:
- Interaction Categories [Abramsky, 1995]
- Interaction Semantics [Talcott, 1996]
- Abstract Behavioral Types [Arbab, 2005]
Actors, Ports, and Behaviors

An actor has $N$ ports $P$

A behavior is a tuple of signals $\sigma = S^N$

An actor is a set of behaviors $A \subset S^N$
Actor Composition

Composition is simple intersection

\[ A = A_1 \cap A_2 \]

\[ A_1 \subseteq S^4 \]

\[ A_2 \subseteq S^4 \]
Connectors

Connectors are (typically) trivial actors.

\[ p_1 A_1 p_2 p_3 A_2 p_4 \]

\[ c \subset S^4, \; s \in c \Rightarrow s_2 = s_3 \]

\[ A = A_1 \cap A_2 \cap c \]
Functional Actors

- Ports become inputs or outputs.
- Actors become functions from inputs to outputs.

\[ p_1 \rightarrow A \rightarrow p_2 \quad A \subseteq S^4 \]

\[ \forall s, s' \in A, \quad s_1 = s'_1 \implies s_2 = s'_2 \]
For Functional Actors, Arbitrary Composition has a Fixed-Point Semantics
Structure of the Tag Set

The algebraic properties of the tag set $T$ are determined by the concurrency model, e.g.:

- Process Networks
- Synchronous/Reactive
- Time-Triggered
- Discrete Events
- Dataflow
- Rendezvous
- Continuous Time
- Hybrid Systems
- ...

Associated with these may be a richer model of the connectors between actors.
Example of a Partially Ordered Tag Set $T$ for Kahn Process Networks

Each signal maps a totally ordered subset of $T$ into values.

Example: Tag Set $T$ for Kahn Process Networks

Ordering constraints on tags imposed by computation:

Totally Ordered Tag Sets

- Example: $T = \mathbb{N}$ (synchronous languages)
- Example: $T = \mathbb{R}_0 \times \mathbb{N}$, with lexicographic order ("super dense time").
  - Used to model
    - hardware,
    - continuous dynamics,
    - hybrid systems,
    - embedded software

See [Liu, Matsikoudis, Lee, CONCUR 2006].
Recall The Catch…

\[ f: [T \rightarrow \{0,1\}^*]^P \rightarrow [T \rightarrow \{0,1\}^*]^P \]

- This is not what (mainstream) programming languages do.
- This is not what (mainstream) software component technologies do.
- This is not what (most) semantic theories do.

Let’s look at the second problem next…
Actor-Oriented Design

Established component interactions:

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

What flows through an object is sequential control

Things happen to objects

The alternative: “Actor oriented:"

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

Actors make things happen

What flows through an object is evolving data

Input data    Output data
The Key To Success: Separation of Concerns

- Abstract Syntax
  - Concrete Syntax
  - Syntax-Based Static Analysis: e.g. Type Systems
- Abstract Semantics
- Concrete Semantics
- Semantics-Based Static Analysis: e.g. Verification
Abstract syntaxes can be formalized.
See [Jackson and Sztipanovits, EMSOFT 2006]
Using GME (from Vanderbilt) an abstract syntax is specified as an object model (in UML) with constraints (in OCL), or alternatively, with MOF.

Such a spec can be used to synthesize visual editors and models transformers.

Meta-model of Ptolemy II abstract syntax, constructed in GME by H. Y. Zheng.
The Key To Success: Separation of Concerns

- Abstract Syntax
- **Concrete Syntax**
- Syntax-Based Static Analysis: e.g. Type Systems
- Abstract Semantics
- Concrete Semantics
- Semantics-Based Static Analysis: e.g. Verification
Concrete Syntax

Example concrete syntax in XML:

```xml
...<entity name="FFT" class="ptolemy.domains.sdf.lib.FFT">
   <property name="order" class="ptolemy.data.expr.Parameter" value="order">
   </property>
   <port name="input" class="ptolemy.domains.sdf.kernel.SDFIOPort">
      ...
   </port>
   ...
</entity>
...
<link port="FFT.input" relation="relation"/>
<link port="AbsoluteValue2.output" relation="relation"/>
...
```

XML and XSLT have made concrete syntax even less important than it used to be. Going a step further, GReAT (from Vanderbilt) works with GME to synthesize model transformers from meta models.
The Key To Success: Separation of Concerns

- Abstract Syntax
- Concrete Syntax
- Syntax-Based Static Analysis: e.g. Type Systems
- Abstract Semantics
- Concrete Semantics
- Semantics-Based Static Analysis: e.g. Verification

The Key To Success: Separation of Concerns

- Abstract Syntax
- Concrete Syntax
- Syntax-Based Static Analysis: e.g. Type Systems
- Abstract Semantics
- Concrete Semantics
- Semantics-Based Static Analysis: e.g. Verification
Where We Are Headed

- An Abstract Semantics
- A Finer Abstract Semantics
- A Concrete Semantics (or Model of Computation)
Tagged Signal Abstract Semantics:

an actor is a subset of the signals with which it interacts. signal is a set of events.

\[ P \subseteq S_1 \times S_2 \]

\[ s_1 \in S_1 \]

\[ s_2 \in S_2 \]

Port may be an input or an output, or neither or both. It is irrelevant.

This outlines a general abstract semantics that gets specialized. When it becomes concrete you have a model of computation.
Functional Abstract Semantics:

An actor is now a function from input signals to output signals.

\[ F : S_1 \rightarrow S_2 \]

\[ s_1 \in S_1 \quad \text{FunctionalProcess} \quad s_2 \in S_2 \]

port is now either an input or an output (or both).

This outlines an abstract semantics for deterministic producer/consumer actors.
Another Finer Abstract Semantics

Process Networks Abstract Semantics:

An actor is a sequence of operations on its signals where the operations are the associative operation of a monoid.

sets of signals are monoids, which allows us to incrementally construct them. E.g.
- stream
- event sequence
- rendezvous points …

\[ P \subseteq S_1 \times S_2 \]

Actor is not necessarily functional (can be nondeterministic).

port is either an input or an output or both.

This outlines an abstract semantics for actors constructed as processes that incrementally read and write port data.
Concrete Semantics that Conform with the Process Networks Abstract Semantics

- Communicating Sequential Processes (CSP) [Hoare]
- Calculus of Concurrent Systems (CCS) [Milner]
- Kahn Process Networks (KPN) [Kahn]
- Nondeterministic extensions of KPN [Various]
- Actors [Hewitt]

Some Implementations:
- Occam, Lucid, and Ada languages
- Ptolemy Classic and Ptolemy II (PN and CSP domains)
- System C
- Metropolis
Process Network Abstract Semantics has a Natural Software Implementation
Process Network Abstract Semantics in Ptolemy II

actor contains ports

IOPort

+get(channelIndex : int) : Token
+hasRoom(channelIndex : int) : boolean
+hasToken(channelIndex : int) : boolean
+isInput() : boolean
+isOutput() : boolean
+send(channelIndex : int, token : Token)

receiver implements communication

receivers

director creates receivers

ptolemy.actor.Actor

+getDirector() : Director

ptolemy.actor.Director

port contains receivers

creates

Actor

+get() : Token
+getContainer() : IOPort
+hasRoom() : boolean
+hasToken() : boolean
+put(t : Token)
+setContainer(port : IOPort)

receiver implements communication to incrementally construct signals

Token

Lee, Berkeley 35
Several Concrete Semantics
Refine this Abstract Semantics

IOPort

```
<table>
<thead>
<tr>
<th>NoRoomException</th>
</tr>
</thead>
<tbody>
<tr>
<td>+get() : Token</td>
</tr>
<tr>
<td>+getContainer() : IOPort</td>
</tr>
<tr>
<td>+hasRoom() : boolean</td>
</tr>
<tr>
<td>+hasToken() : boolean</td>
</tr>
<tr>
<td>+put(t : Token)</td>
</tr>
<tr>
<td>+setContainer(port : IOPort)</td>
</tr>
<tr>
<td>throws</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>NoTokenException</th>
</tr>
</thead>
<tbody>
<tr>
<td>throws</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>QueueReceiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>+get() : Token</td>
</tr>
<tr>
<td>+getContainer() : IOPort</td>
</tr>
<tr>
<td>+hasRoom() : boolean</td>
</tr>
<tr>
<td>+hasToken() : boolean</td>
</tr>
<tr>
<td>+put(t : Token)</td>
</tr>
<tr>
<td>+setContainer(port : IOPort)</td>
</tr>
<tr>
<td>throws</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>FIFOQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
</tr>
<tr>
<td>1..n</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ArrayFIFOQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Mailbox</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>ProcessReceiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
</tr>
<tr>
<td>0..n</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>QueueReceiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>DEREceiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>SDFReceiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>CTReceiver</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>CSPReceiver</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>PNReceiver</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>FIFOQueue</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>ArrayFIFOQueue</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>communicating sequential processes</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>Kahn process networks</th>
</tr>
</thead>
</table>
```

Lee, Berkeley 36
A Still Finer Abstract Semantics

Firing Abstract Semantics:

An actor is still a function from input signals to output signals, but that function now is defined in terms of a firing function.

\[ F : S_1 \rightarrow S_2 \]

\[ s_1 \in S_1 \]

\[ s_2 \in S_2 \]

The process function \( F \) is the least fixed point of a functional defined in terms of \( f \).

signals are in monoids (can be incrementally constructed) (e.g. streams, discrete-event signals).

port is still either an input or an output.
Models of Computation that Conform to the Firing Abstract Semantics

- Dataflow models (all variations)
- Discrete-event models
- Time-driven models (Giotto)

In Ptolemy II, actors written to the firing abstract semantics can be used with directors that conform only to the process network abstract semantics.

Such actors are said to be behaviorally polymorphic.
Actors Language for the Firing Abstract Semantics: Cal

Cal is an actor language designed to provide statically inferable actor properties w.r.t. the firing abstract semantics. E.g.:

```
actor Select () S, A, B ==> Output:
    action S: [sel], A: [v] ==> [v]
    guard sel end

    action S: [sel], B: [v] ==> [v]
    guard not sel end
end
```

Inferable firing rules and firing functions:

\[
U_1 = \left\{ \langle (\text{true}), (v), \bot \rangle : v \in \mathbb{Z} \right\}, f_1 : \langle (\text{true}), (v), \bot \rangle \mapsto (v)
\]

\[
U_2 = \left\{ \langle (\text{false}), \bot, (v) \rangle : v \in \mathbb{Z} \right\}, f_2 : \langle (\text{false}), \bot, (v) \rangle \mapsto (v)
\]

Thanks to Jorn Janneck, Xilinx
A Still Finer Abstract Semantics

Stateful Firing Abstract Semantics:

An actor is still a function from input signals to output signals, but that function now is defined in terms of two functions.

\[ F : S_1 \rightarrow S_2 \]
\[ s_1 \in S_1 \]
\[ F(s_1, f, g) \rightarrow s_2 \in S_2 \]

\[ f : S_1 \times \Sigma \rightarrow S_2 \]
\[ g : S_1 \times \Sigma \rightarrow \Sigma \]

The function \( f \) gives outputs in terms of inputs and the current state. The function \( g \) updates the state.

Signals are monoids (can be incrementally constructed) (e.g. streams, discrete-event signals).

Port is still either an input or an output.
Models of Computation that Conform to the Stateful Firing Abstract Semantics

- Synchronous reactive
- Continuous time
- Hybrid systems

Stateful firing supports iteration to a fixed point, which is required for hybrid systems modeling.

In Ptolemy II, actors written to the stateful firing abstract semantics can be used with directors that conform only to the firing abstract semantics or to the process network abstract semantics.

Such actors are said to be *behaviorally polymorphic*. 

Lee, Berkeley 41
Where We Are

- Tagged Signal Semantics
- Process Networks Semantics
- Firing Semantics
- Stateful Firing Semantics
Where We Are

Tagged Signal Semantics

Process Networks Semantics

Firing Semantics

Kahn process networks

Giotto

synchronous/reactive

discrete events

hybrid systems

continuous time

Lee, Berkeley 43
Ptolemy II emphasizes construction of “behaviorally polymorphic” actors with stateful firing semantics (the “Ptolemy II actor semantics”), but also provides support for broader abstract semantic models via its abstract syntax and type system.
A Consequence: Heterogeneous Composition Semantics

Models of computation can be systematically composed.
The Key To Success: Separation of Concerns

- Abstract Syntax
- Concrete Syntax
- Syntax-Based Static Analysis: e.g. Type Systems
- Abstract Semantics
- Concrete Semantics
- Semantics-Based Static Analysis: e.g. Verification
An algebra of interfaces provides operators for cascade and parallel composition and necessary and sufficient conditions for causality loops, zero-delay loops, and deadlock.

See [Zhou and Lee, EMSOFT 2006]
Recall The Catch…

\[ f : [T \rightarrow \{0,1\}^*]^P \rightarrow [T \rightarrow \{0,1\}^*]^P \]

- This is not what (mainstream) programming languages do.
- This is not what (mainstream) software component technologies do.
- This is not what (most) semantic theories do.

Let’s look at the first problem last…
Programming Languages

- Imperative reasoning is simple and useful
- Keep it!

- The problem is that timing is unpredictable.

- Fix this at the architecture level:
  - Replace cache memories with scratchpads
  - Replace dynamic dispatch with pipeline interleaving
  - Define decidable subsets of standard language
  - Deliver rigorous, precise, and tight WCET bounds.
Conclusion

*The time is right to create the 21-st century theory of (embedded) computing.*