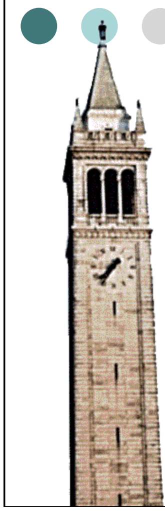


Concurrent Models of Computation for Embedded Software

Edward A. Lee
Professor, UC Berkeley

EECS 219D
Concurrent Models of Computation
Fall 2011

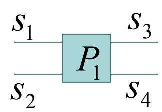
Lecture 15: Actor Abstract Semantics

Tags, Values, Events, and Signals

- 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. $s \subset T \times V$
- The set of all signals $S = P(T \times V)$
- A *functional signal* is a (partial) function $s\colon T \to V$
- A tuple of signals $\mathbf{s} \in S^n$
- The empty signal $\lambda = \varnothing \in S$
- The empty tuple of signals $\Lambda \in S^n$

Processes

A *process* is a subset of signals $P \subset S^n$



$$P_1 \subset S^4$$

The *sort* of a process is the identity of its signals. That is, two processes P_1 and P_2 are of the same sort if

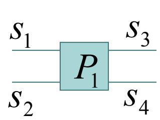
$$\forall i \in \{1,\ldots,n\}, \qquad \pi_i(P_1) = \pi_i(P_2)$$

projection

Alternative Notation

Instead of tuples of signals, let X be a set of variables. E.g.

$$X = \{s_1, s_2, s_3, s_4\}$$

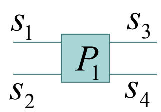


$$P_1 \subset [X \to S] = S^X$$

This is a better notation because it is explicit about the *sort*. This notation was introduced by [Benveniste, et al., 2003]. We will nonetheless stick to the original notation in [Lee, Sangiovanni 1998].

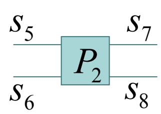
Process Composition

To compose processes, they may need to be augmented to be of the same sort:



$P_1 \subset S^4$

$P_1' = P_1 \times S^4 \subset S^8$

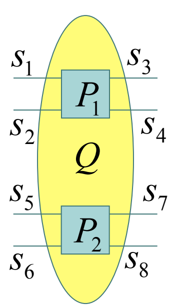


$P_2 \subset S^4$

$P_2' = S^4 \times P_2 \subset S^8$

Process Composition

To compose processes, they may need to be augmented to be of the same sort:



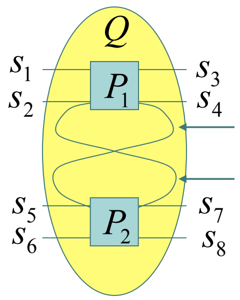
$P_1' = P_1 \times S^4 \subset S^8$

$P_2' = S^4 \times P_2 \subset S^8$

$Q = P_1' \cap P_2' = P_1 \times P_2$

Connections

Connections simply establish that signals are identical:



$$C_{4,5} = \{s \in S^8 \mid \pi_4(s) = \pi_5(s)\}$$

$$C_{2,7} = \{s \in S^8 \mid \pi_2(s) = \pi_7(s)\}$$

$$Q = P_1' \cap P_2' \cap C_{4,5} \cap C_{2,7}$$

Projections (Hiding and Renaming)

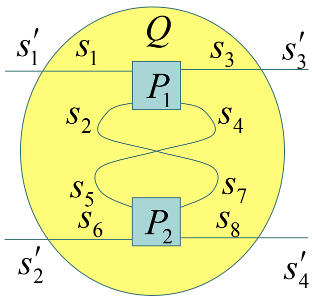
Given an m-tuple of indexes: $I \in \{1, \ldots, n\}^m$

the following projection accomplishes
hiding and/or renaming:

$$\pi_I(P) = (\pi_{\pi_1(I)}(P), \ldots, \pi_{\pi_m(I)}(P))$$

Example of Projections (Hiding)

Projections change the sort of a process:



$$I = (1,3,6,8)$$

$$Q = \pi_I(P_1' \cap P_2' \cap C_{4,5} \cap C_{2,7}) \subset S^4$$

Inputs

Given a process $P \subset S^n$, an *input* is a subset of the same sort, $A \subset S^n$, that constrains the behaviors of the process to

$$P' = P \cap A$$

An input could be a single event in a signal, an entire signal, or any combination of events and signals. A particular process may "accept" only certain inputs, in which case the process is defined by $P \subset S^n$ and $B \subset P(S^n)$, where any input A is required to be in B,

$$A \in B$$

Closed System (no Inputs)

A process $P \subset S^n$ with input set $B \subset P(S^n)$ is *closed* if

$$B = \{S^n\}$$

This means that the only possible input (constraint) is:

$$A = S^n$$

which imposes no constraints at all in

$$P' = P \cap A$$

Functional Processes

Model for a process $P \subset S^n$ that has m input signals and p output signals (exercise: what is the input set B?)

- Define two index sets for the input and output signals:

$$I \in \{1,\ldots,n\}^m, \quad O \in \{1,\ldots,n\}^p$$

- The process is *functional* w.r.t. (I, O) if

$$\forall s, s' \in P, \quad \pi_I(s) = \pi_I(s') \Rightarrow \pi_O(s) = \pi_O(s')$$

- In this case, there is a (possibly partial) function

$$F : S^m \to S^p \quad s.t. \quad \forall s \in P, \quad \pi_O(s) = F(\pi_I(s))$$

Determinacy

A process P with input set B is determinate if for any input $A \in B$,

$$|P \cap A| \in \{0,1\}$$

That is, given an input, there is no more than one behavior.

Note that by this definition, a functional process is assured of being determinate if all its signals are visible on the output.

Refinement Relations

A process (with input constraints) (P', B') is a *refinement* of the process (P, B) if

$$B \subseteq B'$$

and

$$\forall A \in B, \quad P' \cap A \subseteq P \cap A$$

That is, the refinement accepts any input that the process it refines accepts, and for any input it accepts, its behaviors are a subset of the behaviors of the process it refines with the same input.

Tags for Discrete-Event Systems

For DE, let $T = R \times N$ with a total order (the lexical order) and an ultrametric (the Cantor metric). Recall that we have used the structure of this tag set to get nontrivial results:

If processes are functional and causal and every feedback path has at least one delta-causal process, then compositions of processes are determinate and we have a procedure for identifying their behavior.

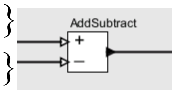
Synchrony

- Two events are synchronous if they have the same tag.
- Two signals are synchronous if all events in one a synchronous with an event in the other.
- A process is synchronous if for in every behavior in the process, every signal is synchronous with every other signal.

Tags for Process Networks

- The tag set T is a poset.
- The tags $T(s)$ on each signal s are totally ordered.
- A *sequential* process has a signal associated with it that imposes ordering constraints on the other signals. For example:

$$s_1 = \{(v_{1,1}, t_{1,1}), (v_{1,2}, t_{1,2}), \ldots\}$$
$$s_2 = \{(v_{2,1}, t_{2,1}), (v_{2,2}, t_{2,2}), \ldots\}$$



$$s_3 = \{(v_{3,1}, t_{3,1}), (v_{3,2}, t_{3,2}), \ldots\}$$

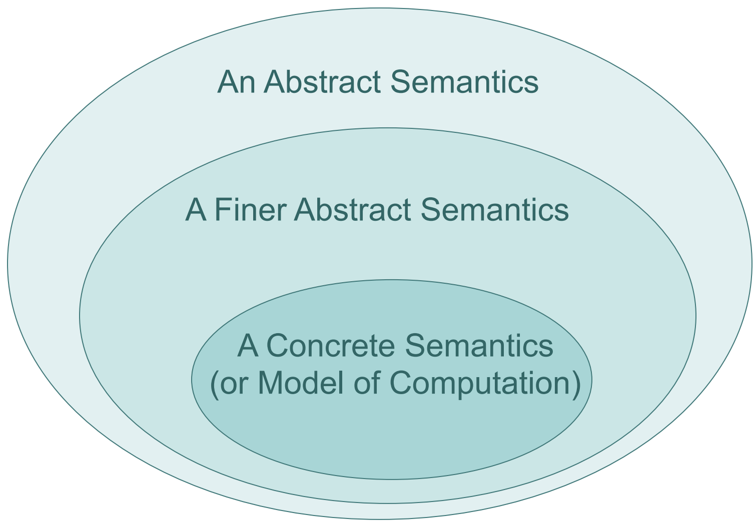
$$s_4 = \{(x, t_{4,1}), (x, t_{4,2}), \ldots\}$$

$$t_{i,j} < t_{i,j+1} \qquad t_{1,j} < t_{4,\,2j},\quad t_{2,j} < t_{4,\,2j+1},\quad t_{3,j} > t_{4,\,2j+1}$$

Tags Can Model …

- Dataflow firing
- Rendezvous in CSP
- Ordering constraints in Petri nets
- etc. (see paper)

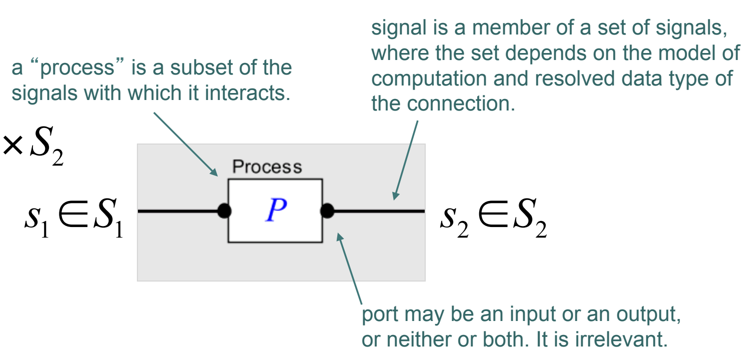
The Tagged Signal Model can be used to Define *Abstract Semantics*



Tagged Signal Abstract Semantics

Tagged Signal Abstract Semantics:

$$P \subset S_1 \times S_2$$



This outlines a general *abstract semantics* that gets specialized. When it becomes concrete you have a *model of computation.*

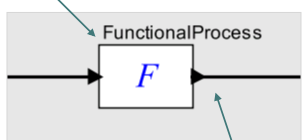
A Finer Abstraction Semantics

Functional Abstract Semantics:

a process is now a function from
input signals to output signals.

$$F : S_1 \to S_2$$

$s_1 \in S_1$



$s_2 \in S_2$

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

This outlines an *abstract semantics* for deterministic producer/consumer actors.

Uses for Such an Abstract Semantics

Give structure to the sets of signals

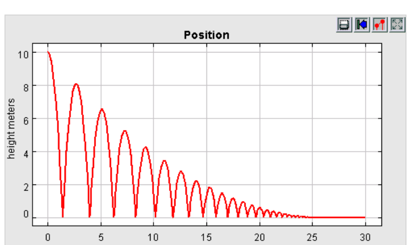
- e.g. Use the Cantor metric to get a metric space.

Give structure to the functional processes

- e.g. Contraction maps on the Cantor metric space.

Develop static analysis techniques

- e.g. Conditions under which a hybrid systems is provably non-Zeno.



Another Finer Abstract Semantics

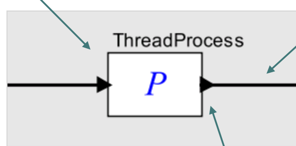
Process Networks Abstract Semantics:

A process 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 …

$$P \subset S_1 \times S_2$$

$s_1 \in S_1$



$s_2 \in S_2$

process 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.

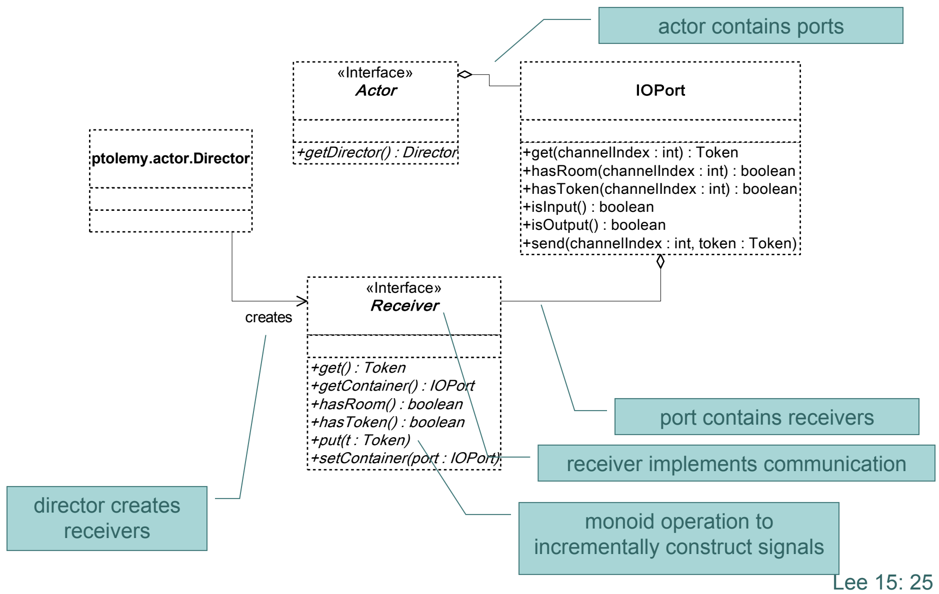
Concrete Semantics that Conform with the Process Networks Abstract Semantics

- 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:

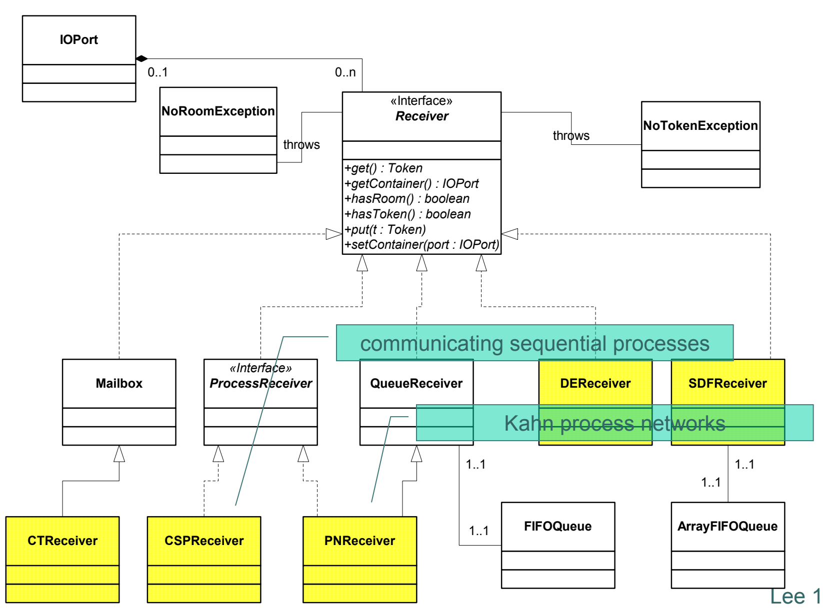
- Occam, Lucid, and Ada languages
- Ptolemy Classic and Ptolemy II (PN and CSP domains)
- System C
- Metropolis

Process Network Abstract Semantics in Ptolemy II



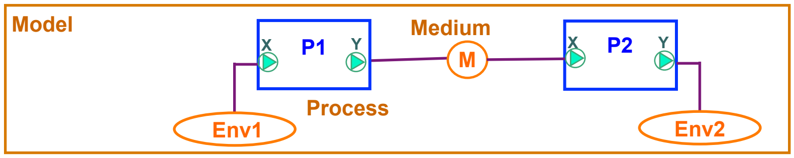
Lee 15: 25

Several Concrete Semantics Refine this Abstract Semantics



Lee 15: 26

Process Network Abstract Semantics in Metropolis




```
process P{
  port reader X;
  port writer Y;
  thread(){
   while(true){
    ...
    z = f(X.read());
    Y.write(z);
   }}}
```

```
interface reader extends Port{
    update int read();
    eval int n();
}
```

```
interface writer extends Port{
    update void write(int i);
    eval int space();
}
```

```
medium M implements reader, writer{
   int storage;
   int n, space;
   void write(int z){
       await(space>0; this.writer ; this.writer)
            n=1; space=0; storage=z;
    }
    word read(){ ... }
}
```

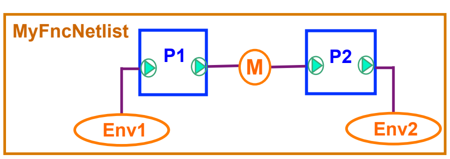
Thanks to
Doug Densmore

Lee 15: 27

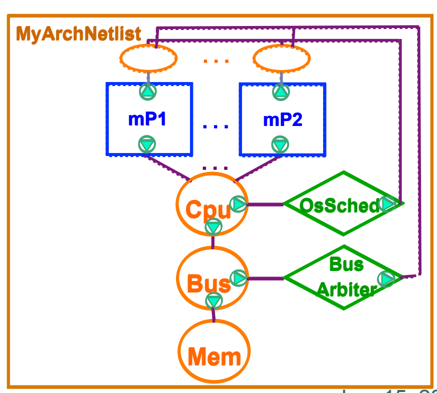
Leveraging Abstract Semantics for Joint Modeling of Architecture and Application

MyMapNetlist

B(P1, M.write) <=> B(mP1, mP1.writeCpu); E(P1, M.write) <=> E(mP1, mP1.writeCpu);
B(P1, P1.f) <=> B(mP1, mP1.mapf); E(P1, P1.f) <=> E(mP1, mP1.mapf);
B(P2, M.read) <=> B(P2, mP2.readCpu); E(P2, M.read) <=> E(mP2, mP2.readCpu);
B(P2, P2.f) <=> B(mP2, mP2.mapf); E(P2, P2.f) <=> E(mP2, mP2.mapf);



The abstract semantics provides natural points of the execution (where the monoid operations are invoked) that can be synchronized across models. Here, this is used to model operations of an application on a candidate implementation architecture.



Lee 15: 28

A Finer Abstract Semantics

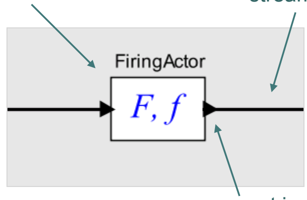
Firing Abstract Semantics:

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

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

$$F : S_1 \to S_2$$

$s_1 \in S_1$



$s_2 \in S_2$

port is still either an input or an output.

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

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*.

Actor Language for the Firing Abstract Semantics: Cal

Cal is an experimental 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:

$$U_1 = \{\langle(\text{true}),(v),\perp\rangle : v \in \mathbf{Z}\},\ f_1 : \langle(\text{true}),(v),\perp\rangle \mapsto (v)$$
$$U_2 = \{\langle(\text{false}),\perp,(v)\rangle : v \in \mathbf{Z}\},\ f_2 : \langle(\text{false}),\perp,(v)\rangle \mapsto (v)$$

Thanks to Jorn Janneck, Xilinx

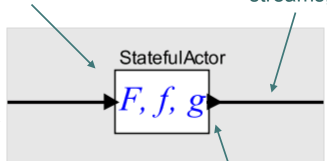
A Still Finer Abstract Semantics

Stateful Firing Abstract Semantics:

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

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

$$F : S_1 \to S_2$$

$s_1 \in S_1$  $s_2 \in S_2$

$$f : S_1 \times \Sigma \to S_2$$
$$g : S_1 \times \Sigma \to \Sigma$$

state space

port is still either an input or an output.

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

Models of Computation that Conform to the Stateful Firing Abstract Semantics

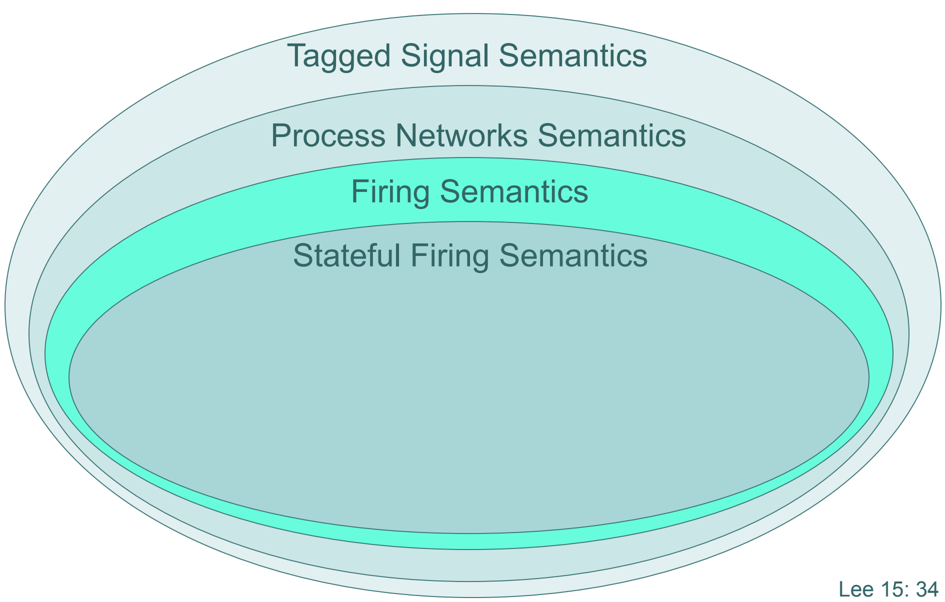
- Synchronous reactive
- Continuous time
- Hybrid systems

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

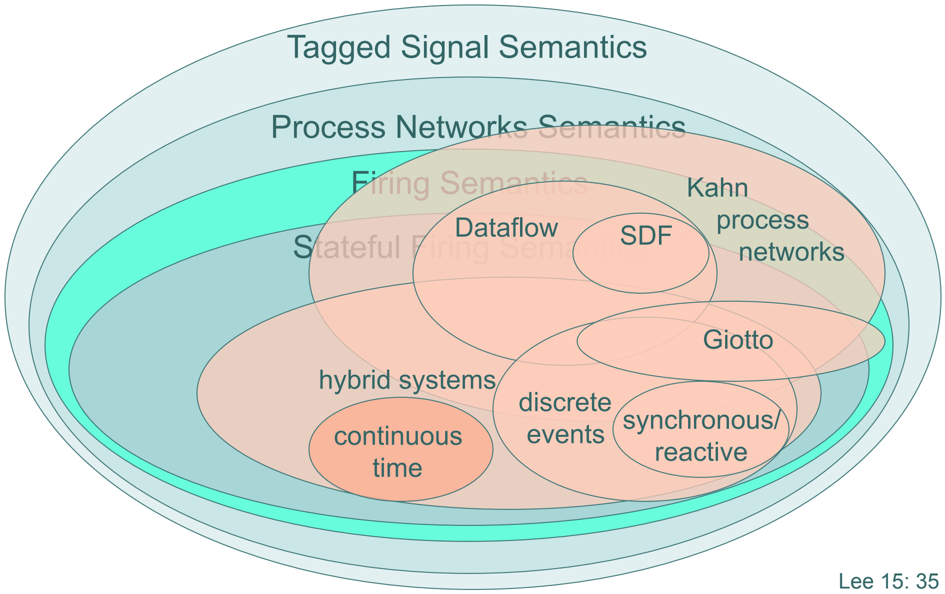
In Ptolemy II, actors written to the stateful firing abstract semantics can be used with directors that conform only to the firing abstract semantics or to the process network abstract semantics.

Such actors are said to be *behaviorally polymorphic*.

Where We Are



Where We Are



Many Ptolemy II Actors work in All these MoCs!
Execution of Ptolemy II Actors

Flow of control:
Preinitialization
Initialization
Execution
Finalization

How Does This Work?
Execution of Ptolemy II Actors

Flow of control:

- Preinitialization
- Initialization
- Execution
- Finalization

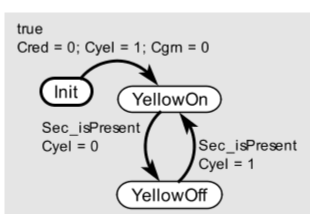
E.g., Partial evaluation (esp. higher-order components), set up type constraints, etc. Anything that needs to be done prior to static analysis (type inference, scheduling, …)

How Does This Work?
Execution of Ptolemy II Actors

Flow of control:

- Preinitialization
- Initialization
- Execution
- Finalization

E.g., Initialize actors, produce initial outputs, etc.



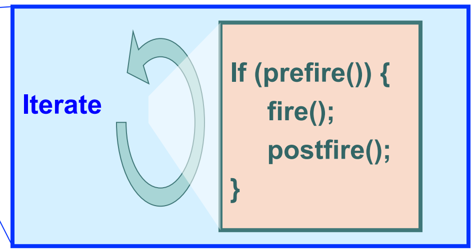
E.g., set the initial state of a state machine. Initialization may be repeated during the run (e.g. if the *reset* parameter of a transition is set and the destination state has a refinement).

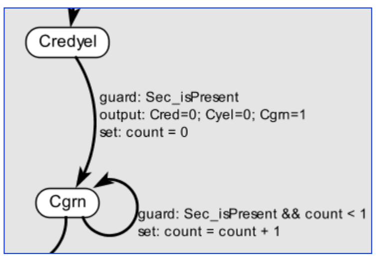
Lee 15: 38

How Does This Work? Execution of Ptolemy II Actors

Flow of control:

- Preinitialization
- Initialization
- Execution
- Finalization





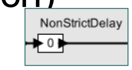
In fire(), an FSM first fires the refinement of the current state (if any), then evaluates guards, then produces outputs specified on an enabled transition. In postfire(), it postfires the current refinement (if any), executes set actions on an enabled transition, and takes the transition.

How Does This Work? Execution of Ptolemy II Actors

Flow of control:

- Preinitialization
- Initialization
- Execution
- Finalization

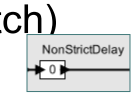
Definition of the NonStrictDelay Actor (Sketch)




```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        _previousToken = initialValue.getToken();
    }

    public boolean prefire() {
        return true;
    }

    public void fire() {
        if (_previousToken != null) {
            if (_previousToken == AbsentToken.ABSENT) {
                output.sendClear(0);
            } else {
                output.send(0, _previousToken);
            }
        } else {
            output.sendClear(0);
        }
    }

    public boolean postfire() {
        if (input.isKnown(0)) {
            if (input.hasToken(0)) {
                _previousToken = input.get(0);
            } else {
                _previousToken = AbsentToken.ABSENT;
            }
        }
        return true;
    }
}
```

Lee 15: 41

Definition of the NonStrictDelay Actor (Sketch)



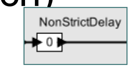
initialization

```
protected Token _previousToken;
public Parameter initialValue;

public void initialize() {
    _previousToken = initialValue.getToken();
}
```

```
            if (_previousToken == AbsentToken.ABSENT) {
                output.sendClear(0);
            } else {
                output.send(0, _previousToken);
            }
        } else {
            output.sendClear(0);
        }
    }

    public boolean postfire() {
        if (input.isKnown(0)) {
            if (input.hasToken(0)) {
                _previousToken = input.get(0);
            } else {
                _previousToken = AbsentToken.ABSENT;
            }
        }
        return true;
    }
}
```

Lee 15: 42

Definition of the NonStrictDelay Actor (Sketch)

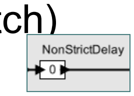



```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        previ
    }

    public boo
        return
    }

    public voi
        if (_p
            if
            }
            }
        } else
            output.sendClear(0);
        }
    }

    public boolean postfire() {
        if (input.isKnown(0)) {
            if (input.hasToken(0)) {
                _previousToken = input.get(0);
            } else {
                _previousToken = AbsentToken.ABSENT;
            }
        }
        return true;
    }
}
```

prefire: can the actor fire?

```
public boolean prefire() {
    return true;
}
```


Definition of the NonStrictDelay Actor (Sketch)

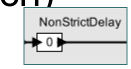



```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        _previousToken = initialValue.getToken();
    }

    public boolean prefire() {
        return
    }

    public voi
        if (_p
            if
            }
            }
        } else
            ou
        }
    }

    public boo
        if (in
            if
            }
            }
        }
        return true;
    }
}
```

fire: produce outputs (in this case, the output does not depend on the input).

```
public void fire() {
    if (_previousToken != null) {
        if (_previousToken == AbsentToken.ABSENT) {
            output.sendClear(0);
        } else {
            output.send(0, _previousToken);
        }
    } else {
        output.sendClear(0);
    }
}
```

Definition of the NonStrictDelay Actor (Sketch)

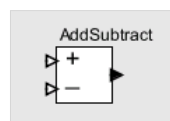



```
public class NonStrictDelay extends TypedAtomicActor {
    protected Token _previousToken;
    public Parameter initialValue;

    public void initialize() {
        _previousToken = initialValue.getToken();
    }

    public boolean prefire() {
        return true;
    }

    public void fire() {
```

postfire: record state changes

```
public boolean postfire() {
    if (input.isKnown(0)) {
        if (input.hasToken(0)) {
            _previousToken = input.get(0);
        } else {
            _previousToken = AbsentToken.ABSENT;
        }
    }
    return true;
}
```


A Consequence of Our Abstract Semantics: Behavioral Polymorphism

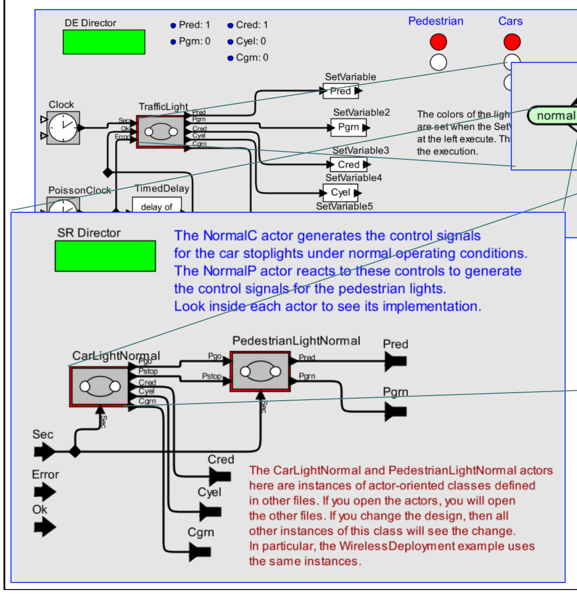


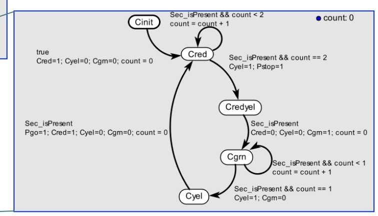
Data polymorphism:
- Add numbers (int, float, double, Complex)
- Add strings (concatenation)
- Add composite types (arrays, records, matrices)
- Add user-defined types

Behavioral polymorphism:
- In dataflow, add when all connected inputs have data
- In a synchronous/reactive model, add when the clock ticks
- In discrete-event, add when any connected input has data, and add in zero time
- In process networks, execute an infinite loop in a thread that blocks when reading empty inputs
- In rendezvous, execute an infinite loop that performs rendezvous on input or output
- In push/pull, ports are push or pull (declared or inferred) and behave accordingly

By not choosing among these when defining the component, we get a huge increment in component re-usability. Abstract semantics ensures that the component will work in all these circumstances.

More Interestingly, Hierarchical Models are Also Behaviorally Polymorphic





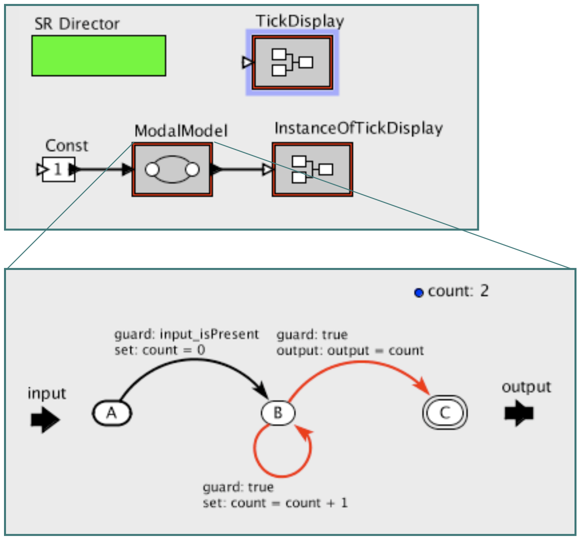
The same FSM infrastructure works in DE and SR! (and also continuous time, dataflow, etc.)

Modal Models

Modal models are actors that have multiple *modes* of operation, where the switching between modes is governed by a state machine.

In each mode, the *mode refinement* specifies (part of) the input output behavior.

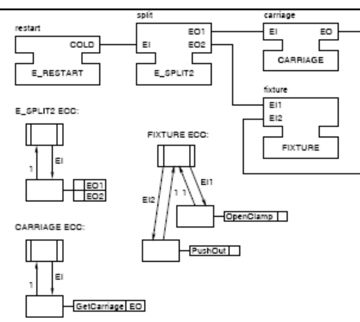
Using this in an SR model



Here, the behavior of an actor is given as a state machine that reads inputs, writes outputs, and updates both local variables and its state.

A very tricky part about executing this is that one of the two nondeterminate transitions produces an output. That output must be produced in fire(), and then postire() has to take that same transition.

Some efforts get confused: IEC 61499



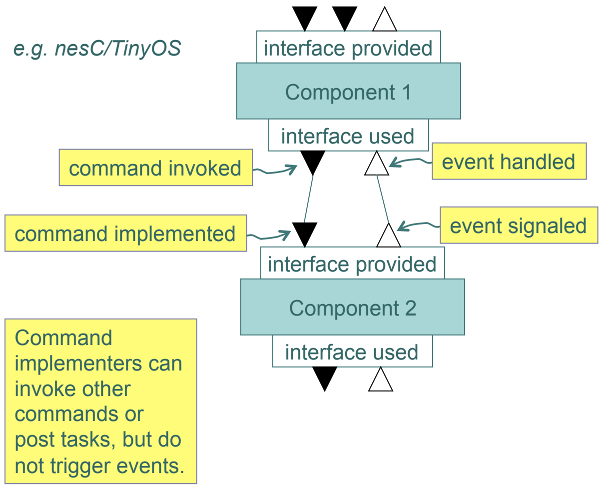
International Electrotechnical Commission (IEC) 61499 is a standard established in 2005 for distributed control systems software engineering for factory automation.

The standard is (apparently) inspired by formal composition of state machines, and is intended to facilitate formal verification.

Regrettably, the standard essentially fails to give a concurrency model, resulting in radically different behaviors of the same source code from on runtime environments from different vendors, and (worse) highly nondeterministic behaviors on runtimes from any given vendor.

See: Ĉengić, G., Ljungkrantz, O. and Åkesson, K., Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime. in *11th IEEE International Conference on Emerging Technologies and Factory Automation*, (Prague, Czech Republic 2006).

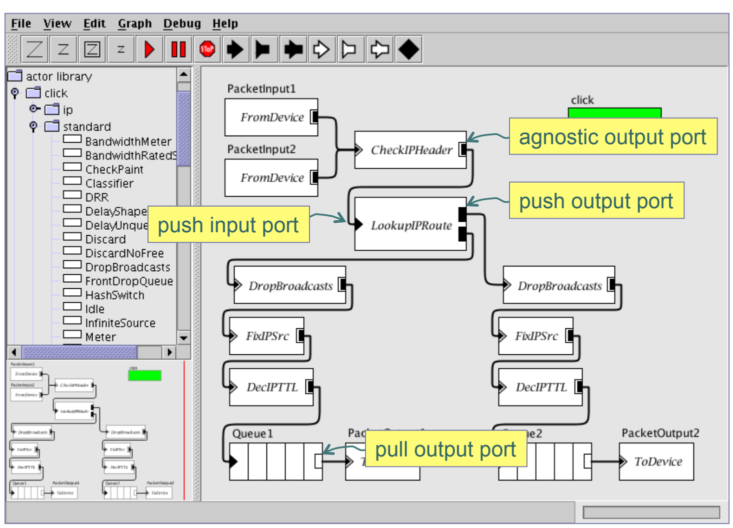
Other MoCs that may be suitable for TSM modeling: Sensor Network Languages



Typical usage pattern:

- hardware interrupt signals an event.
- event handler posts a task.
- tasks are executed when machine is idle.
- tasks execute atomically w.r.t. one another.
- tasks can invoke commands and signal events.
- hardware interrupts can interrupt tasks.
- exactly one mutex, implemented by disabling interrupts.

Other MoCs that may be suitable for TSM modeling: Network Languages



Click (Kohler) with a visual syntax in Mescal (Keutzer)

Typical usage:

- queues have push input, pull output.
- schedulers have pull input, push output.
- thin wrappers for hardware have push output or pull input only.

Related Work

- Abramsky, et al., Interaction Categories
- Agha, et al., Actors
- Hoare, CSP
- Mazurkiewicz, et al., Traces
- Milner, CCS and Pi Calculus
- Reed and Roscoe, Metric Space Semantics
- Scott and Strachey, Denotational Semantics
- Winskel, et al., Event Structures
- Yates, Networks of real-time processes

Conclusion and Open Issues

- The *tagged signal model* provides a very general conceptual framework for comparing and reasoning about models of computation,
- The tagged signal model provides a natural model of *design refinement*, which offers the possibility of type-system-like formal structures that deal with dynamic behavior, and not just static structure.
- The idea of *abstract semantics* offers ways to reason about multi-model frameworks like Ptolemy II and Metropolis, and offers clean definitions of behaviorally polymorphic components.