Overview of Ptolemy II, with focus on Discrete Event and QSS

Michael Wetter and
Thierry S. Nouidui

Simulation Research Group

June 19, 2015

BERKELEY LAB
Lawrence Berkeley National Laboratory

Overview

The purpose is to understand

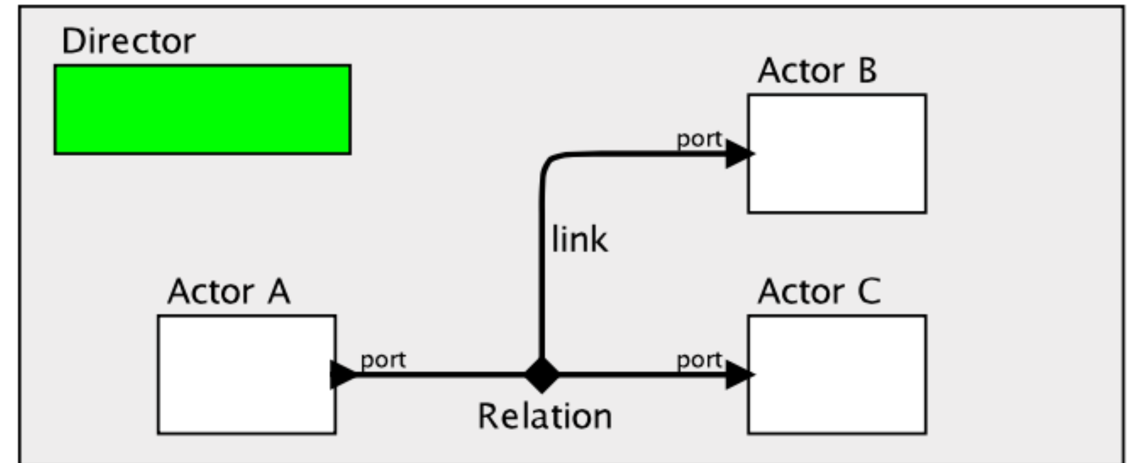
1. the structure of Ptolemy II
2. discrete event simulation
3. QSS methods

Introduction to Ptolemy II

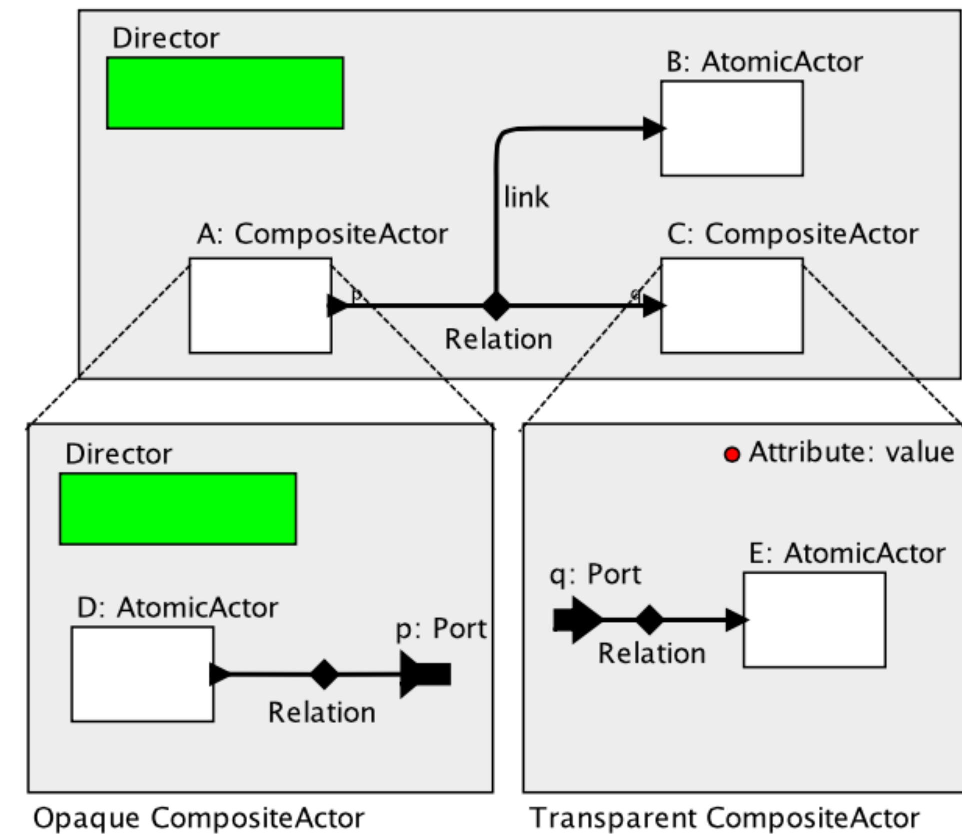
Acknowledgement: Much of the content is based on lecture notes of Prof. Edward Lee

Visual rendering of actor model

Icons represent software components.
The Director orchestrates the interaction of actors.
Actors implement the model equations.

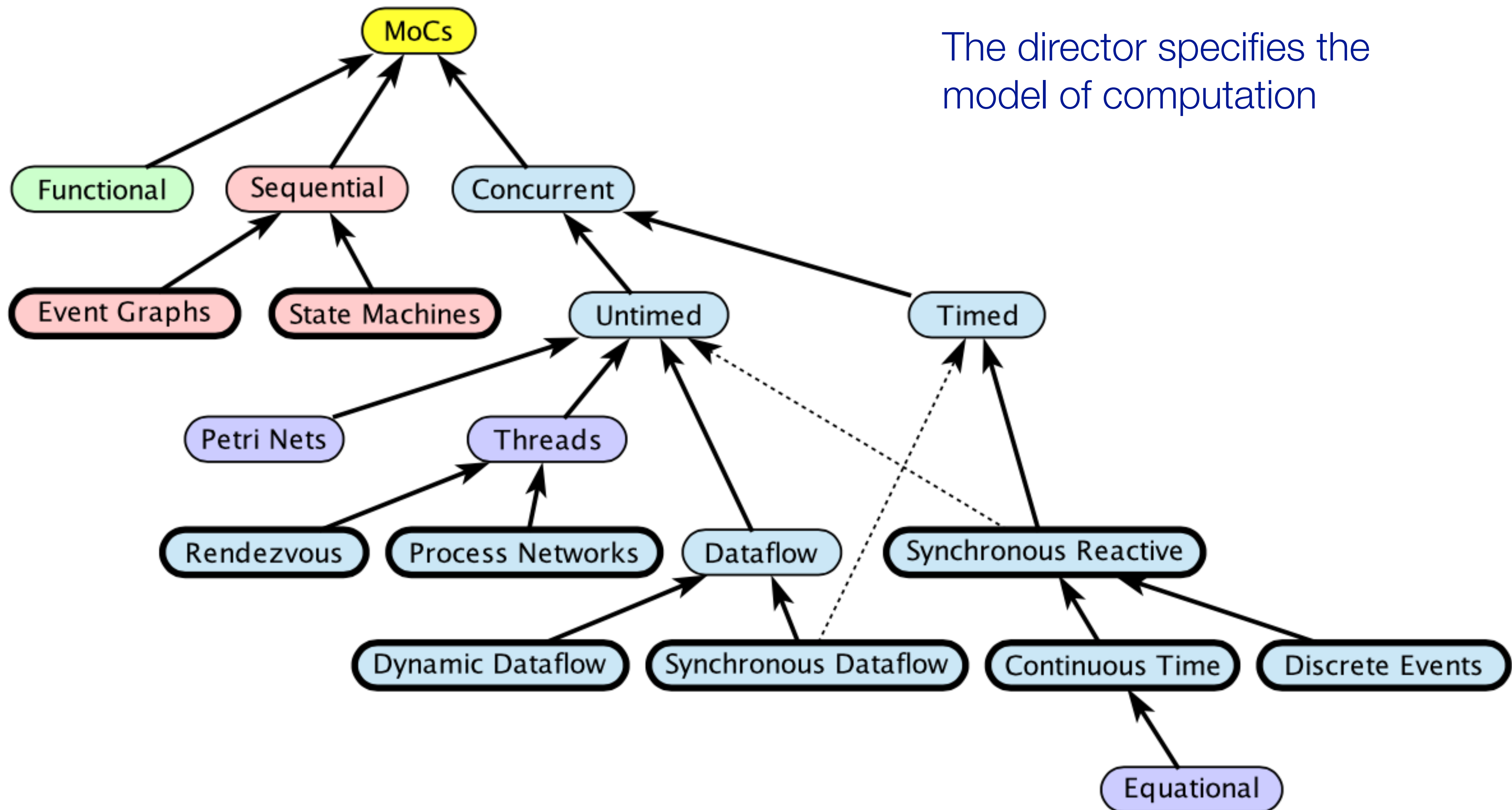


Actors can contain other actors
with or without a director



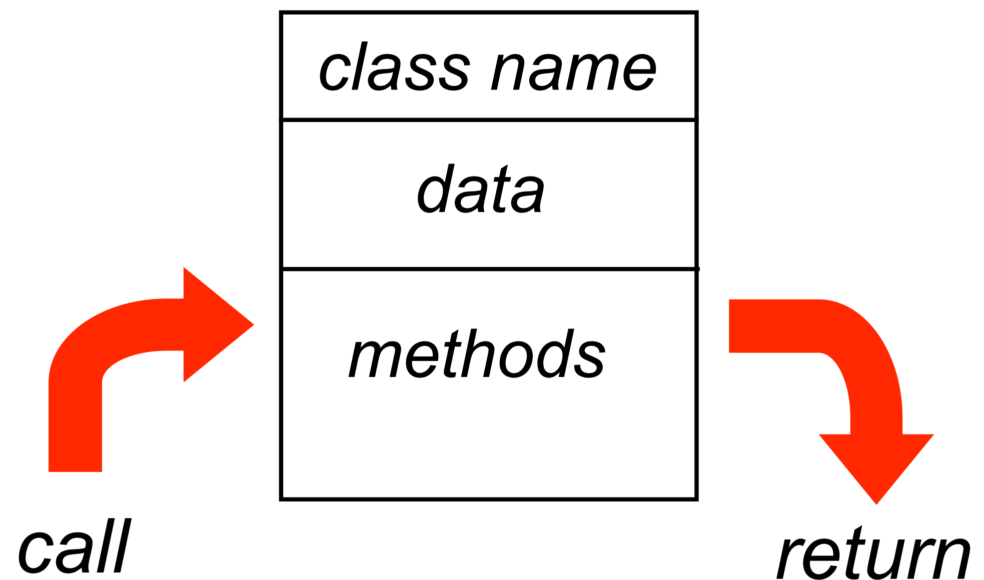
Models of Computations (MoCs) determine how system evolves

The director specifies the model of computation



Ptolemy components are actors and objects

The established: Object-oriented:

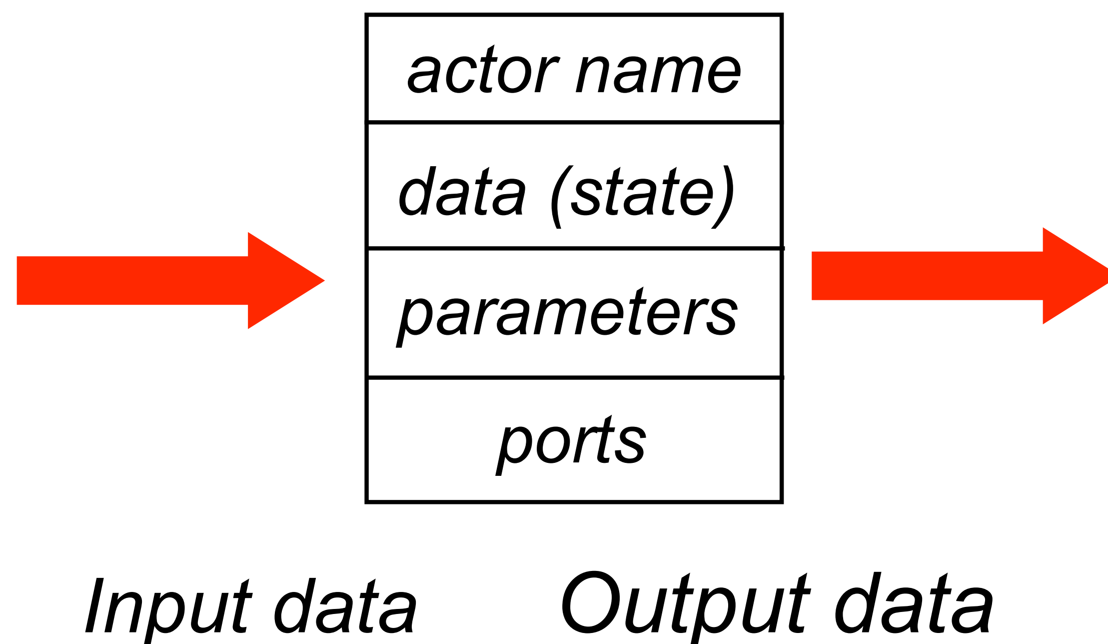


What flows through an object is sequential control

Things happen to objects

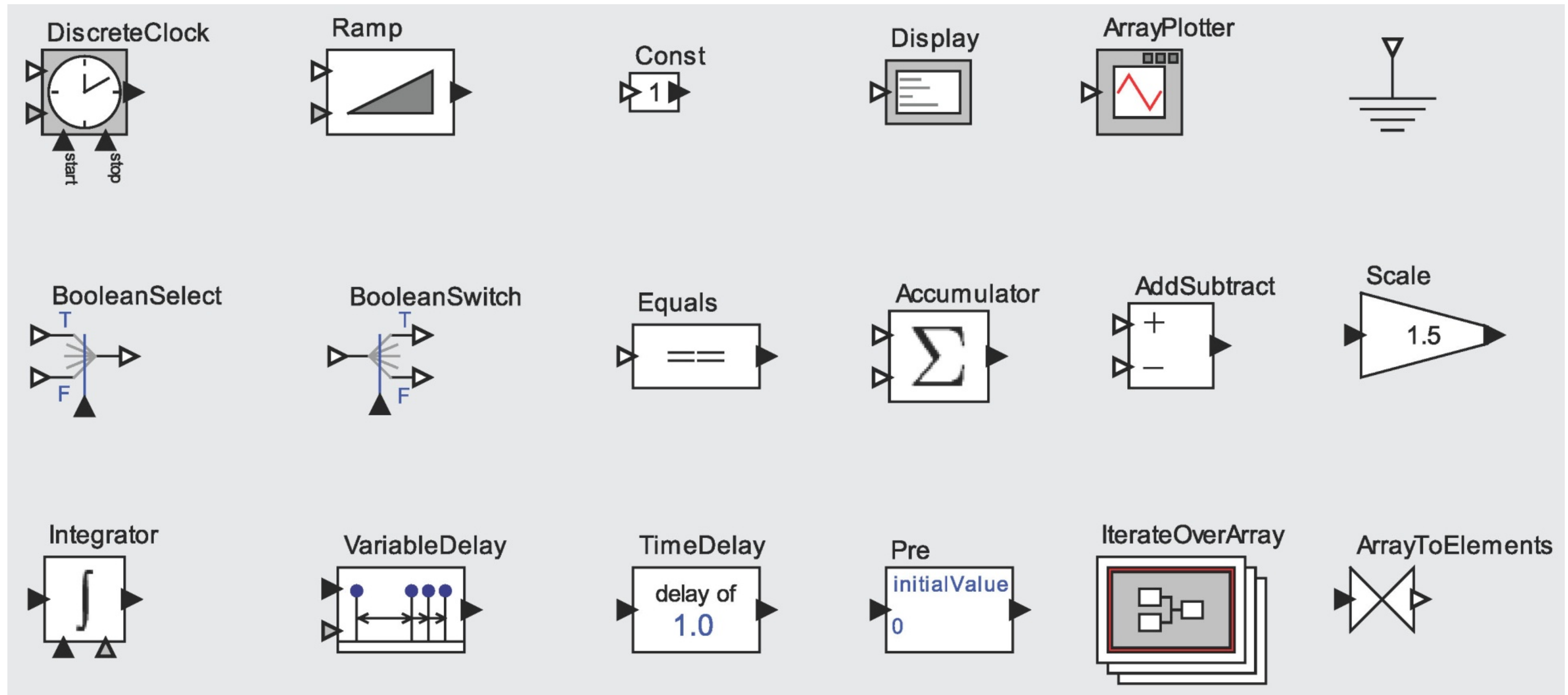
The alternative: Actor oriented:

Actors make things happen



What flows through an object is evolving data

Ptolemy has a library with pre-defined actors, mostly in Java, but can be in C, Python, Cal and MATLAB



Most actors are written in Java

file:/C:/ptII/ptolemy/data/type/demo/Router/Router.xml

File View Edit Graph Debug Help

Utilities
Directors
Actors
Sources
GenericSources
TimedSources
Clock
CurrentTime
PoissonClock
TimedSinewave
TriggeredClock
VariableClock
SequenceSources
Sinks
Array
Conversions
FlowControl
HigherOrderActors
IO
Logic

DE Director

Master Clock

String Sequence

Sequence Count

Gaussian

Customize
Documentation
Appearance
Save Actor In Library
Listen to Actor
Set Breakpoints
Convert to Class
Open Actor Ctrl+L
Open Instance

file:/C:/ptII/ptolemy/actor/lib/Gaussian.java

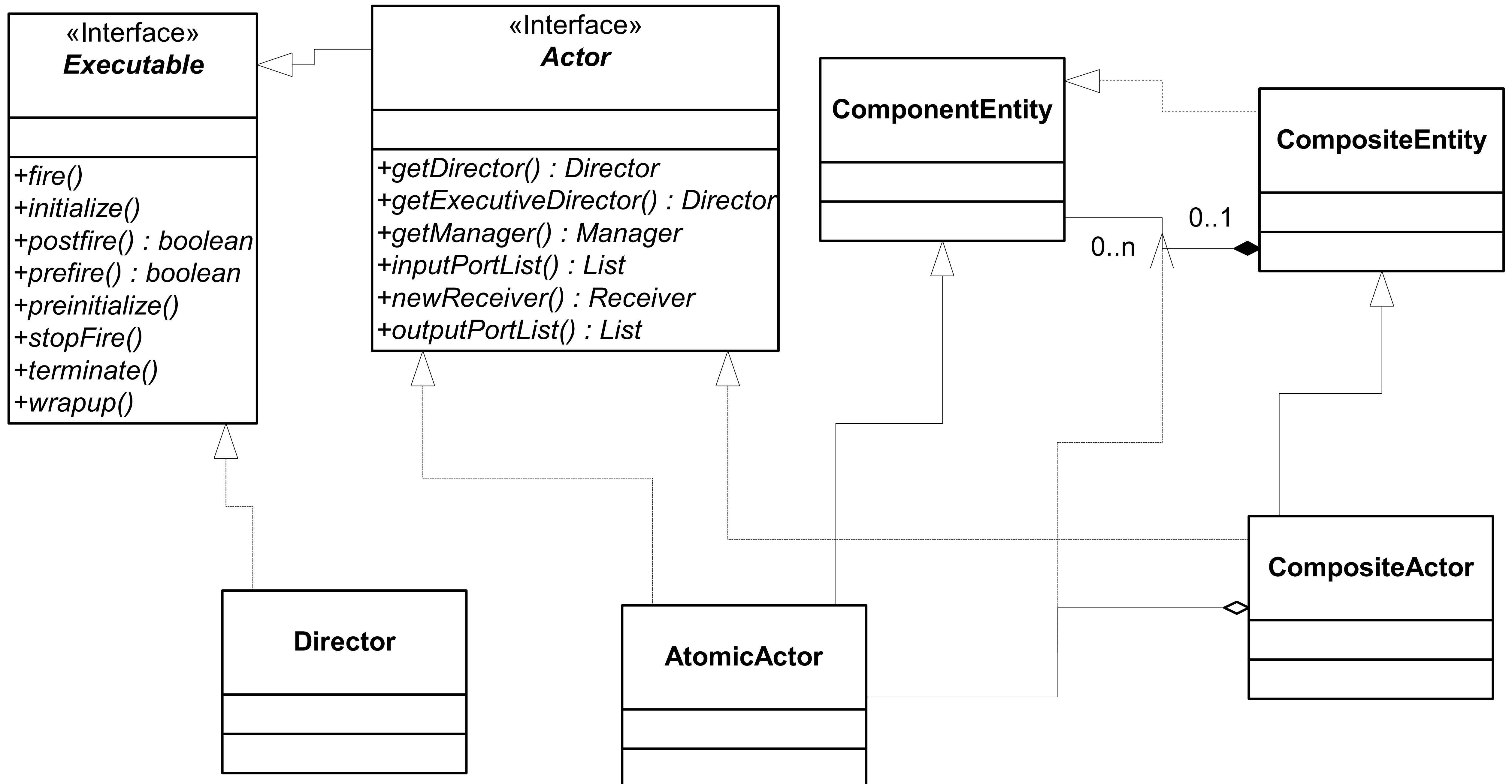
File Help

```
public class Gaussian extends RandomSource {
    /** Construct an actor with the given container and name.
     *  @param container The container.
     *  @param name The name of this actor.
     *  @exception IllegalActionException If the actor cannot be contained
     *   by the proposed container.
     *  @exception NameDuplicationException If the container already has an
     *   actor with this name.
     */
    public Gaussian(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException {
        super(container, name);

        output.setTypeEquals(BaseType.DOUBLE);

        mean = new PortParameter(this, "mean", new DoubleToken(0.0));
        mean.setTypeEquals(BaseType.DOUBLE);

        standardDeviation = new PortParameter(this, "standardDeviation");
        standardDeviation.setExpression("1.0");
        standardDeviation.setTypeEquals(BaseType.DOUBLE);
    }

    ///////////////////////////////////////////////////////////////////
    ////                     ports and parameters                  ////

    /** The mean of the random number.
```

Simple String manipulation actor

```
public class Ptolemnizer extends TypedAtomicActor {

    public Ptolemnizer(CompositeEntity container, String name)
            throws IllegalActionException, NameDuplicationException {
        super(container, name);

        input = new TypedIOPort(this, "input");
        input.setTypeEquals(BaseType.STRING);
        input.setInput(true);

        output = new TypedIOPort(this, "output");
        output.setTypeEquals(BaseType.STRING);
        output.setOutput(true);
    }

    public TypedIOPort input;
    public TypedIOPort output;

    public void fire() throws IllegalActionException {
        if (input.hasToken(0)) {
            Token token = input.get(0);
            String result = ((StringToken)token).stringValue();
            result = result.replaceAll("t", "pt");
            output.send(0, new StringToken(result));
} } }
```

Object model for executable components



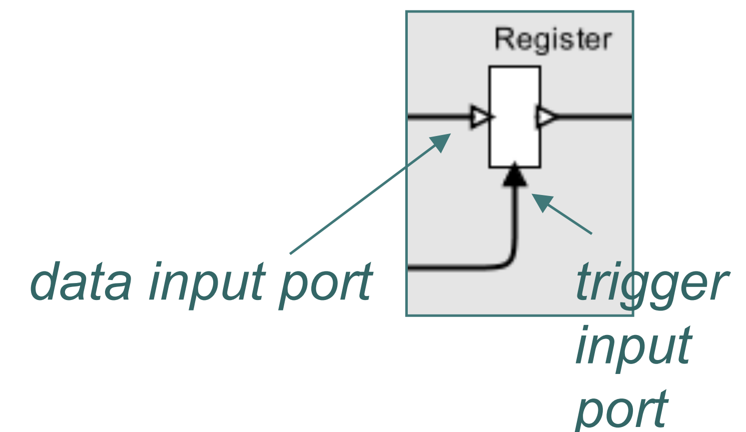
The main methods are prefire, fire and poster

Can the actor fire?

React to trigger input.

Read the data input and update the state.

```
class Register extends TypedAtomicActor {
  private Object state;
  boolean prefire() {
    if (trigger is known) { return true; }
  }
  void fire() {
    if (trigger is present) {
      send state to output;
    } else {
      assert output is absent;
    }
  }
  void postfire() {
    if (trigger is present) {
      state = value read from data input;
    }
  }
```



Abstract semantics

Views every actor as an Abstract State Machine:

- Actor = Inputs + Outputs + States + Initial state + Fire + Postfire

Fire = output function: produces outputs given current inputs + state $F : S \times I \rightarrow O$

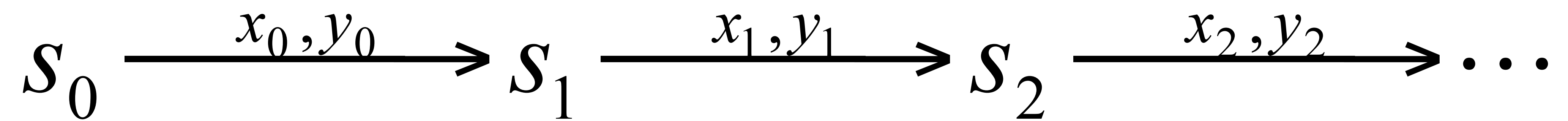
Postfire = transition function: updates state given current inputs + state $P : S \times I \rightarrow S$

Why separate fire and postfire?

Source: http://chess.eecs.berkeley.edu/pubs/712.html

Behaviors

Set of traces:



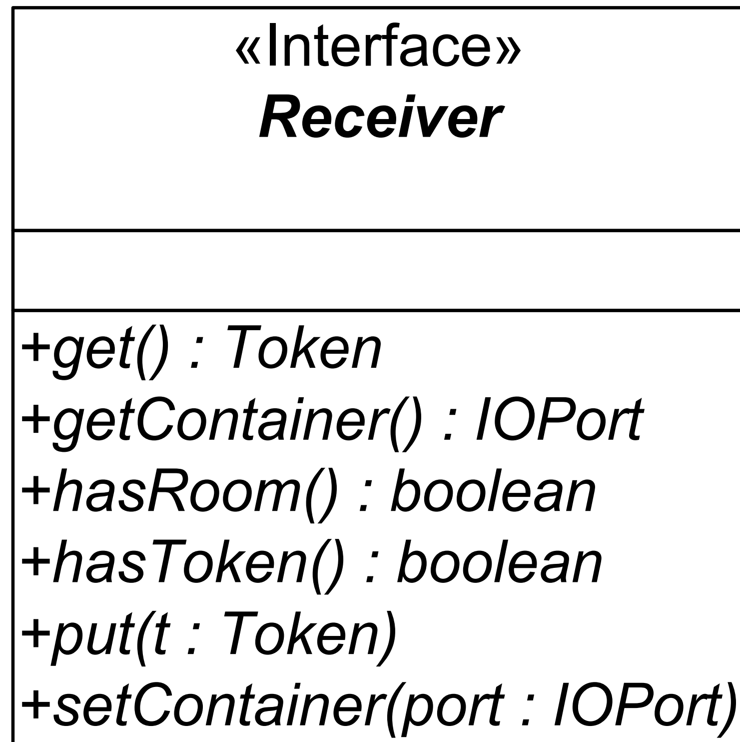
such that for all i :

$$y_i = F(s_i, x_i)$$

$$s_{i+1} = P(s_i, x_i)$$

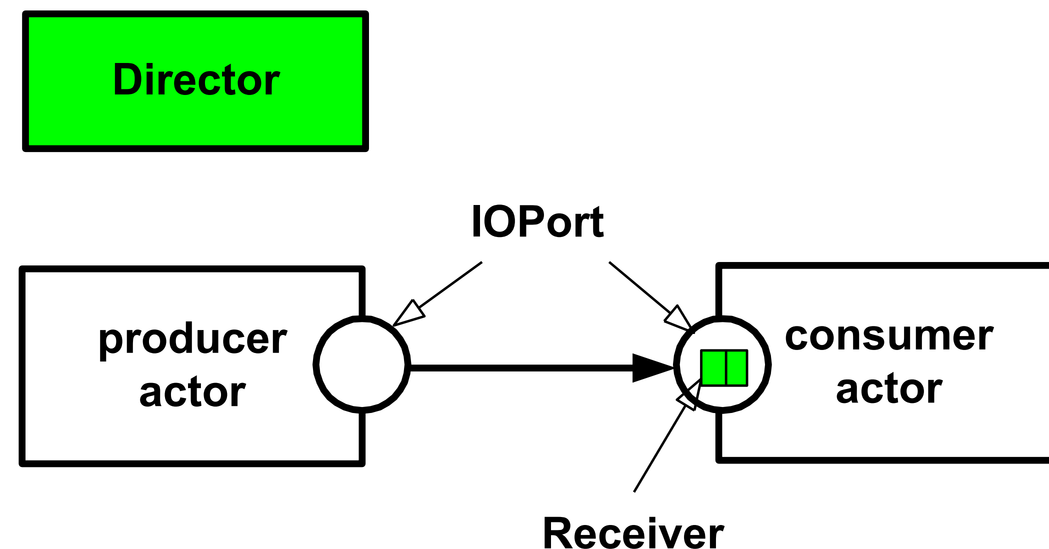
Source: http://chess.eecs.berkeley.edu/pubs/712.html

Object-oriented approach to behavioral polymorphism



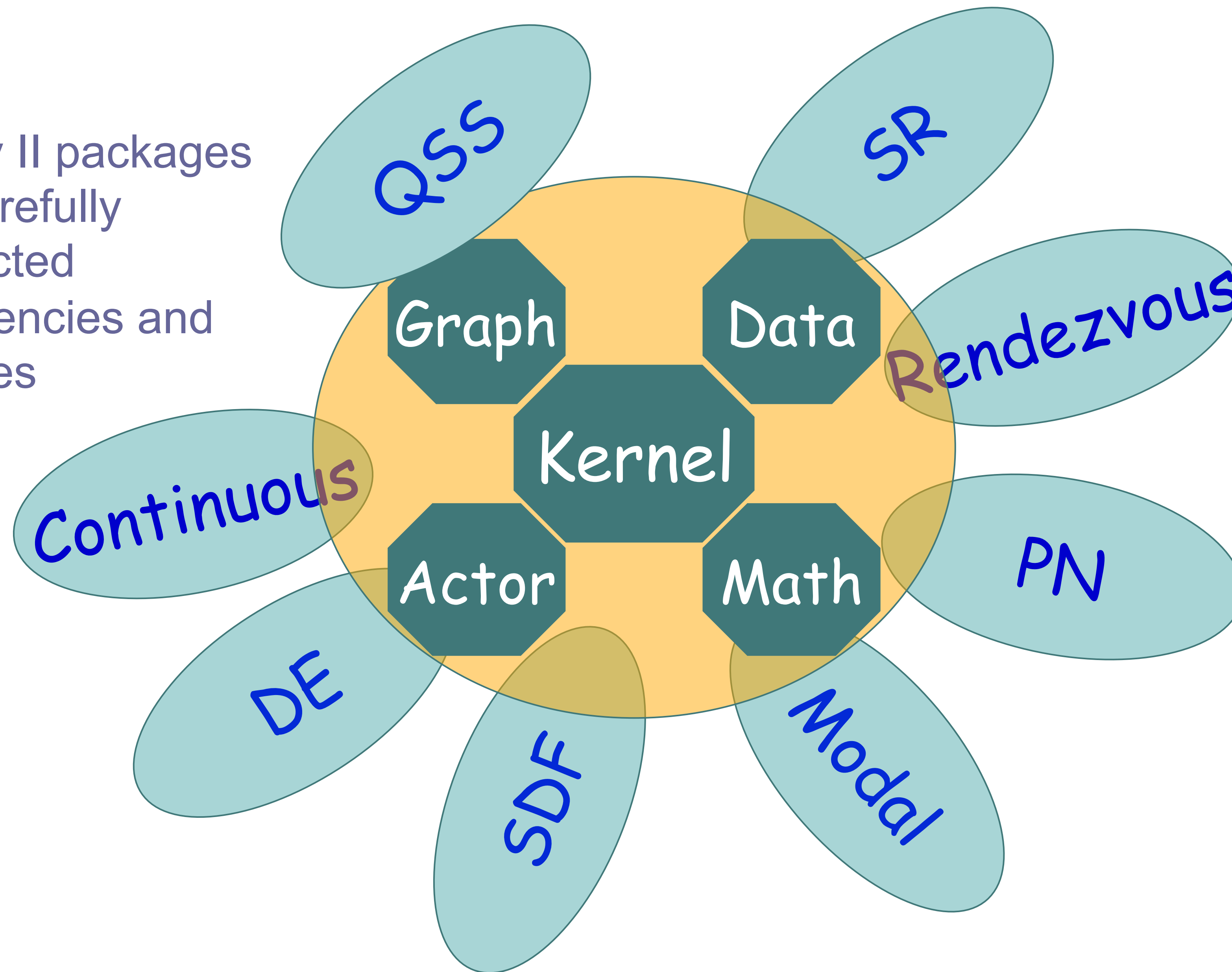
These polymorphic methods implement the communication semantics of a domain in Ptolemy II. The receiver instance used in communication is supplied by the director, not by the component.

Behavioral polymorphism *is the idea that components can be defined to operate with multiple MoCs.*



Extensible software architecture

Ptolemy II packages have carefully constructed dependencies and interfaces



In SOEP, we will use Discrete Event and Synchronous Data Flow

Discrete event for QSS solver.

Synchronous Data Flow is for models that allow static scheduling.

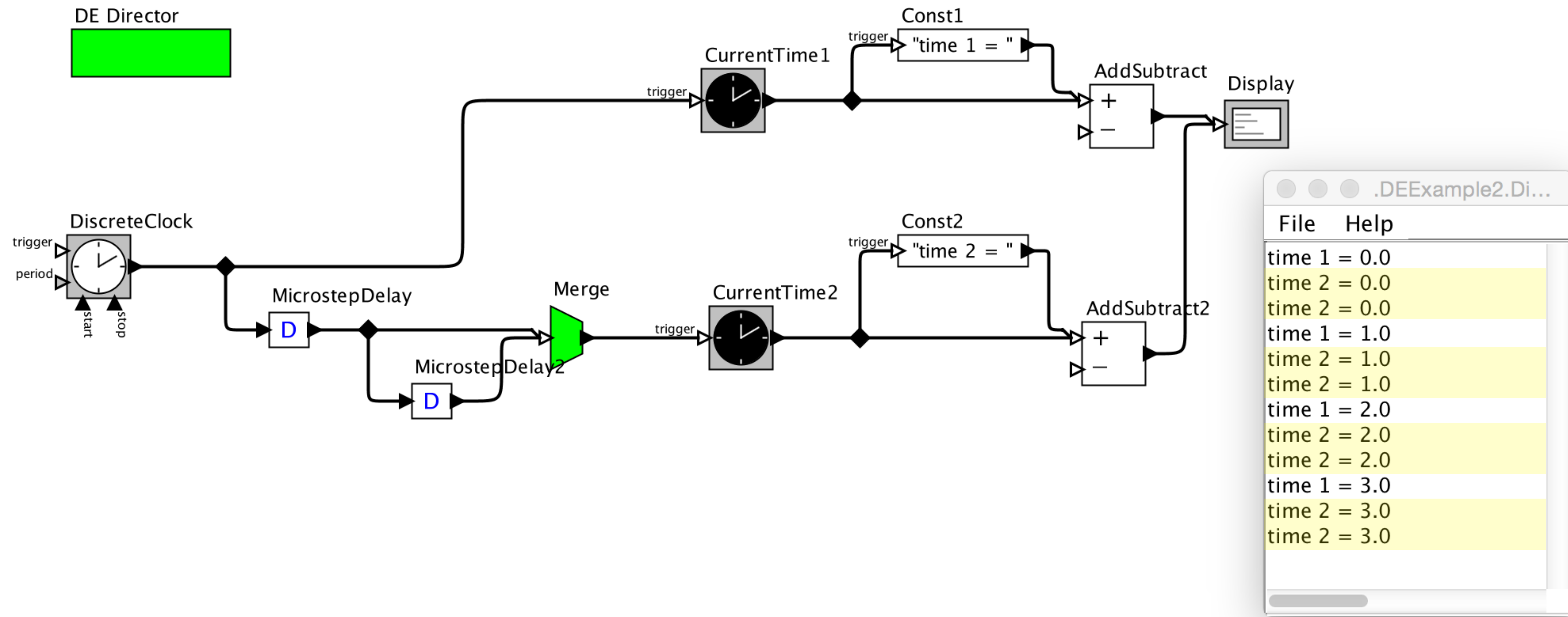
Discrete event simulation

Further details: Chapter 7 in the System Design, Modeling, and Simulation using Ptolemy II of Claudius Ptolemaeus.

Continuous domain simulation

	Continuous domain	Discrete event
Time	Continuum, in tools like EnergyPlus or TRNSYS, typically one state at each instant.	In general, need not be timed. For our applications, a signal is defined at certain time instants.
Time data type	In most continuous time simulator, a real number	Superdense time: Real number and integer
Signal	Exists for all time instant	Exists only at discrete time instants, and is absent otherwise

In DE, actors send time-stamped events to each other, and events are processed in chronological order



Time in Ptolemy II

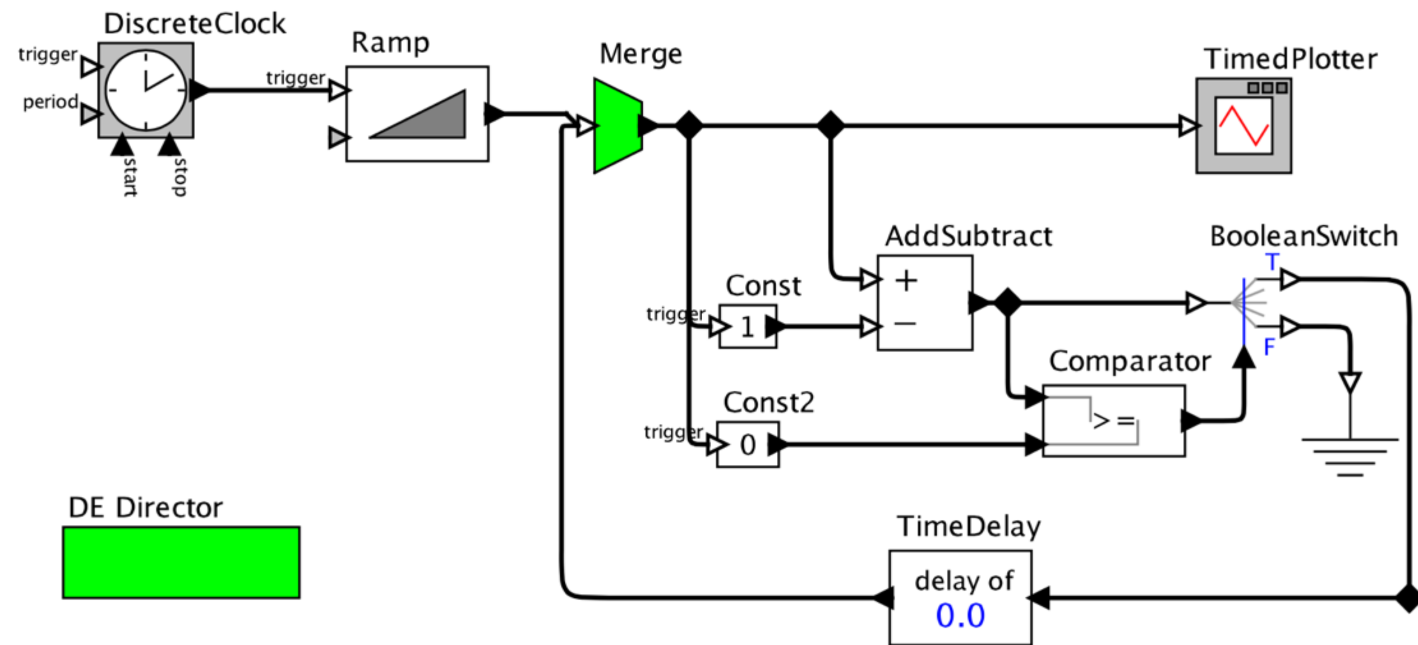
Time is a tuple: $(t, n) \in (\mathbb{R}, \mathbb{N})$

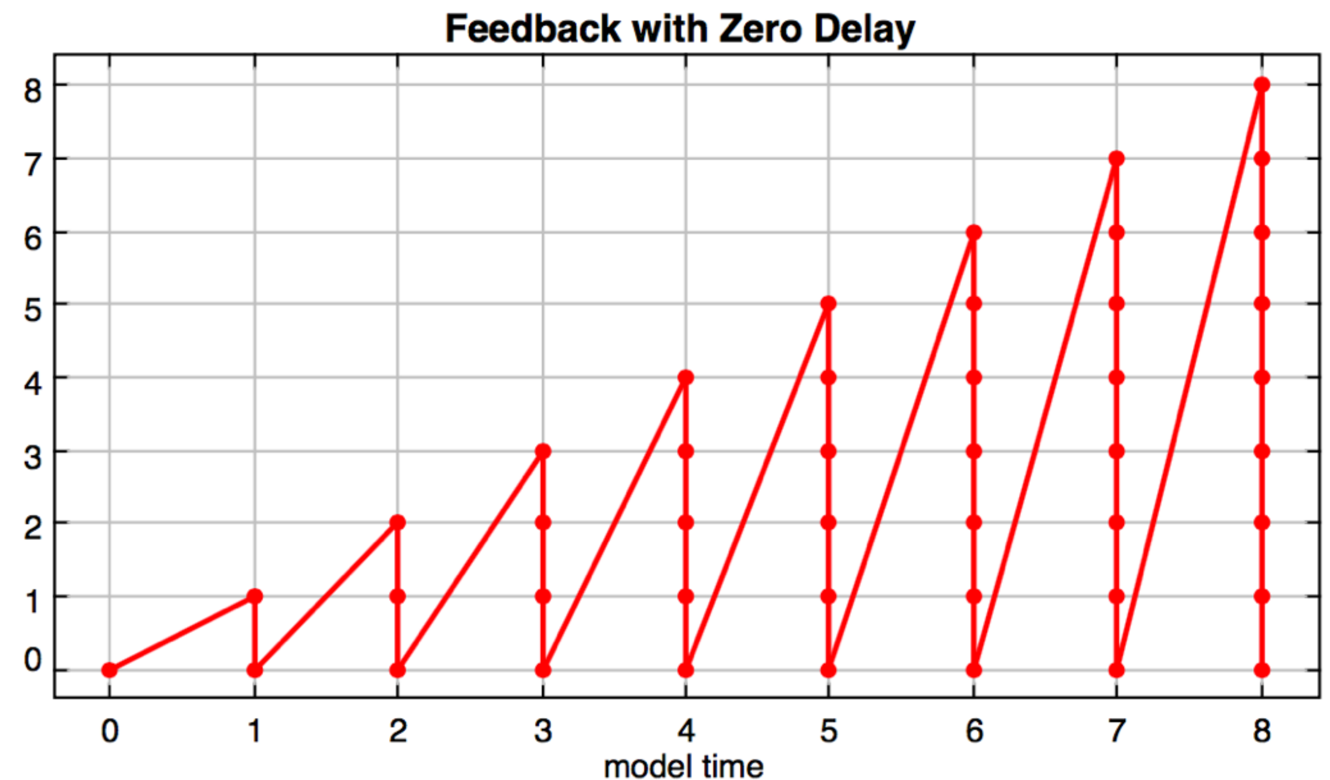
Two events *(t_1, n_1)* and *(t_2, n_2)* are *weakly* simultaneous if $t_1 = t_2$, and *strongly* simultaneous if in addition $n_1 = n_2$.

A signal can have two distinct values at *(t, n_1)* and *(t, n_2)*.

Every feedback loop must have at least one actor that introduces a time delay.

The order in which actors are fired is governed by a topological sort.





But how can time be compared?

In Ptolemy, model time t is $t = m\, r$.

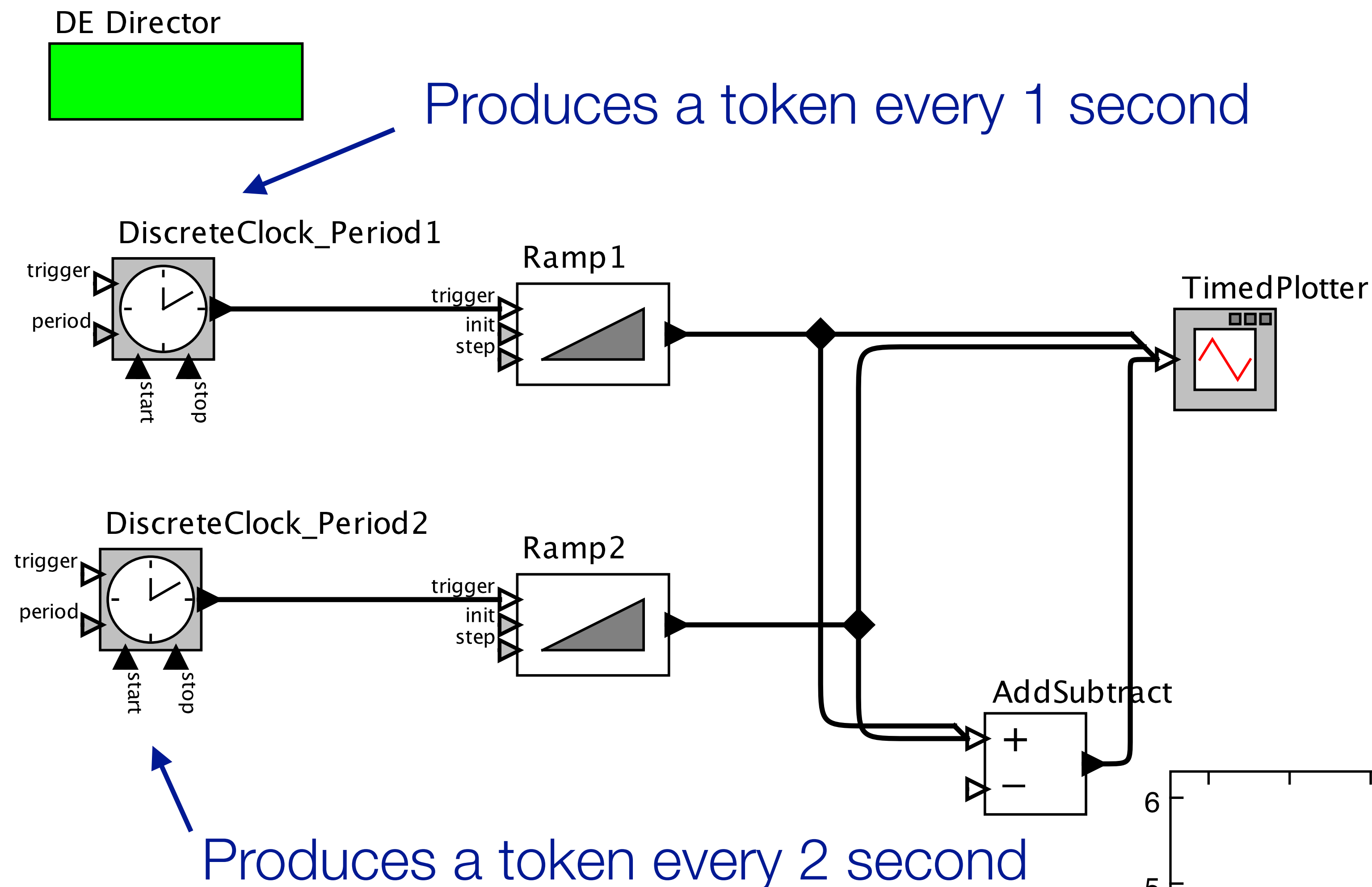
- m arbitrarily large integer
- r time resolution (by default, 10^{-10} seconds).

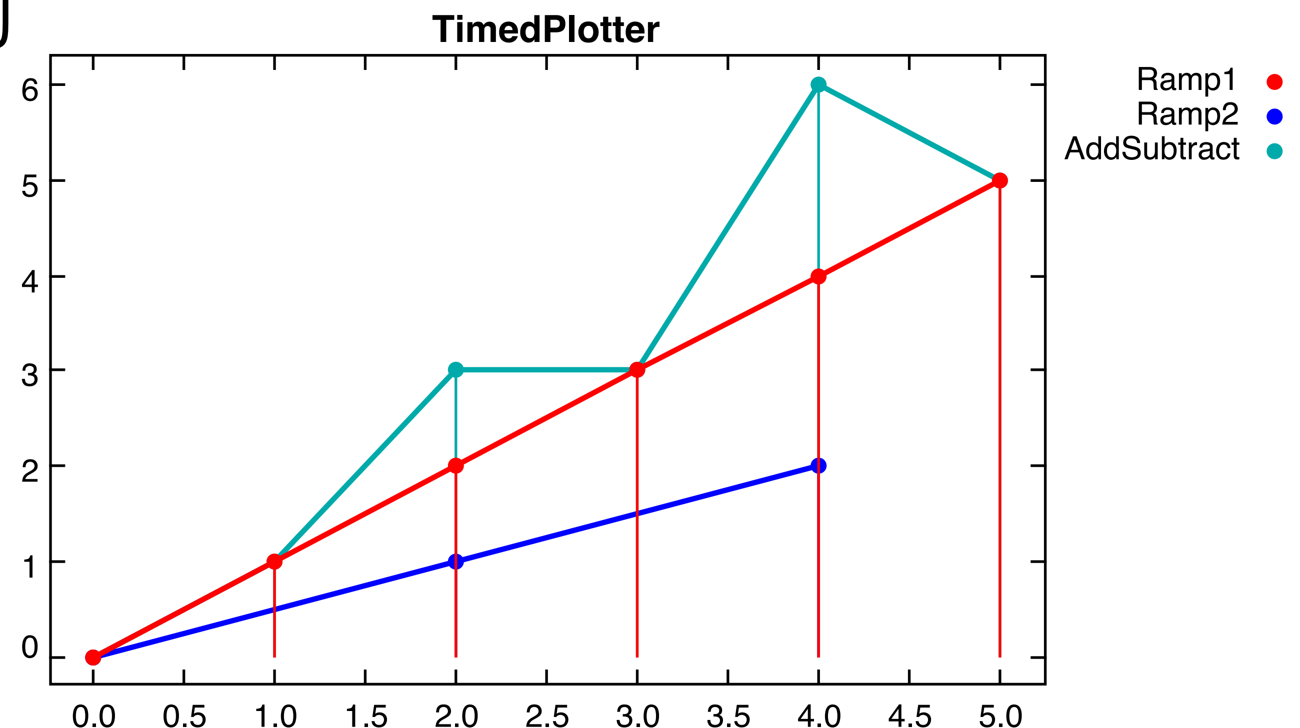
Example: $m=10^{11}$ represent 10 seconds.

Addition and subtraction is implemented so it does not suffer quantization errors.

Hence, $t_1 + t_2 + t_3 = t_1 + (t_2 + t_3)$

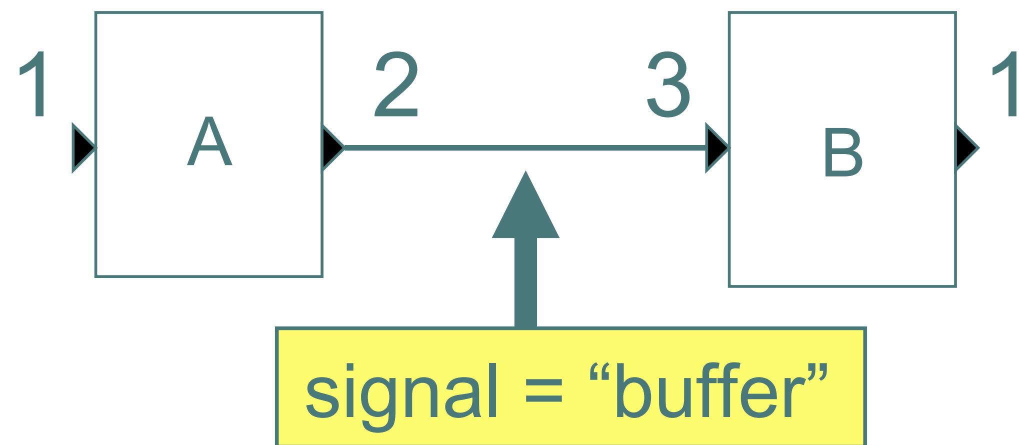
A simple DE example





What output of AddSubtract would you have expected in a similar model in E+?

In SOEP, we may also use the Synchronous Data Flow (SDF) director

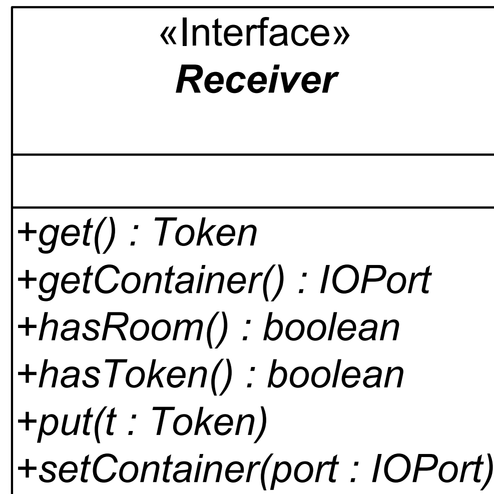


In each firing, actors consume a fixed number of tokens from the input streams, and produce a fixed number of tokens on the output streams.

SDF allows *static* scheduling for when actors will be fired.

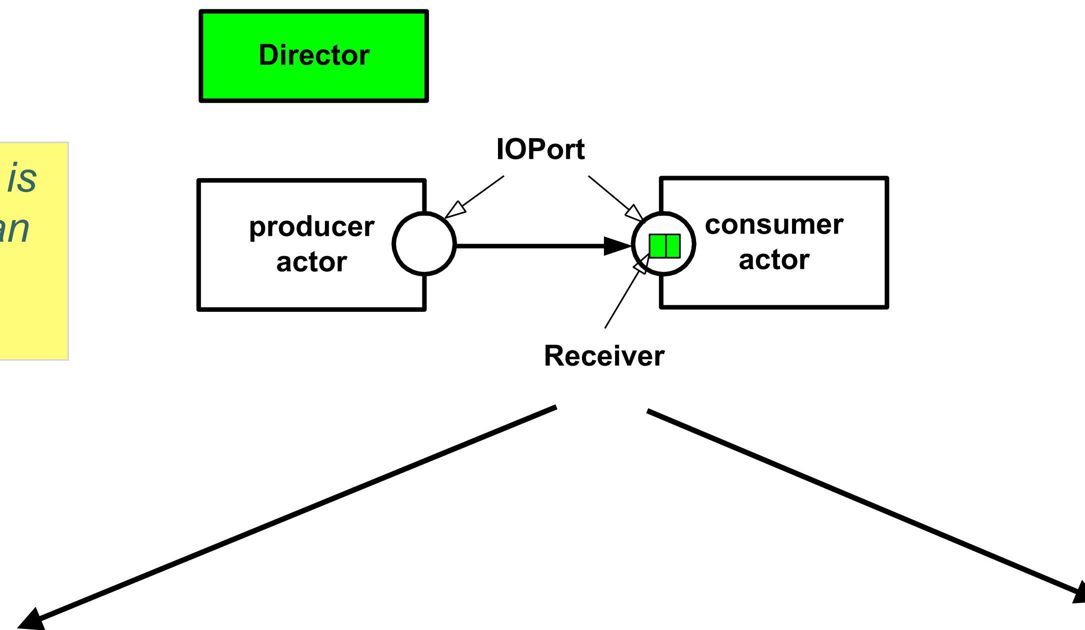
This is what most models in the BCVTB use.

Behavioral polymorphism



These polymorphic methods implement the communication semantics of a domain in Ptolemy II. The receiver instance used in communication is supplied by the director, not by the component.

Behavioral polymorphism *is the idea that components can be defined to operate with multiple MoCs.*



In discrete event domain

```
public Token get(){
  return (Token)_tokens.removeFirst();
}
```

In continuous time domain

```
public Token get(){
  return _token;
}
```

The token is no longer there after it is received.

Note: Exception handling removed for clarity.

QSS

Background Applications

QSS history

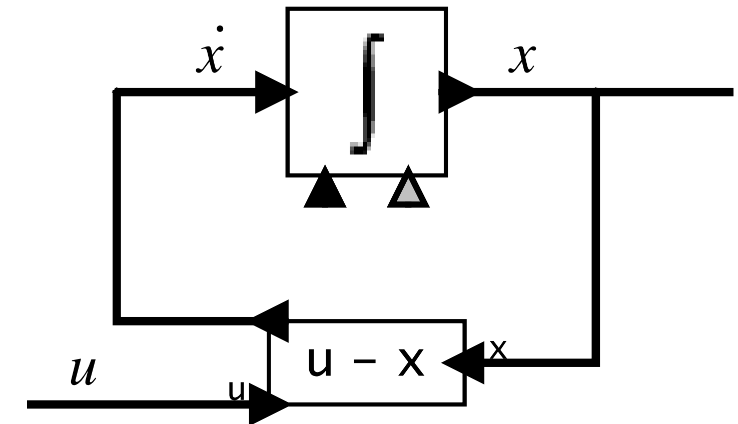
- Basic idea: Zeigler and Lee (1998)
- QSS1: Kofman and Junco (2001)
- QSS2: Kofman (2002)
- QSS3: Kofman (2006)
  PowerDEVS implementation: Floros et al. (2010)
- Extended to handle stiff systems: Migoni et al.
- (2013)
- Hybrid systems: Bliudze and Furic (2014)

The main goal is to get a discrete-event style of execution, avoiding iterative solving and backtracking, and good interfaces to discrete systems.

Simple first order system

Consider a first-order system with state x and input u given by

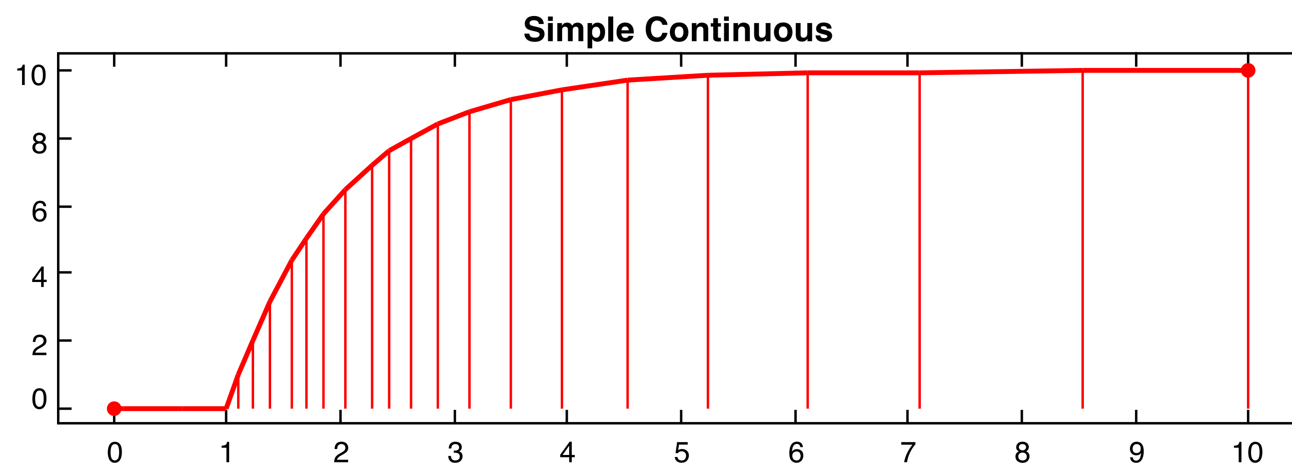
$$\dot{x}(t) = u(t) - x(t).$$



Let

$$u(t) = \begin{cases} 0 & t < 1 \\ 10 & t \geq 1 \end{cases}$$

A variable-step-size RK 2-3 solver quantizes time according to an error estimate and produces:



Simple first order system

Quantize the state as follows

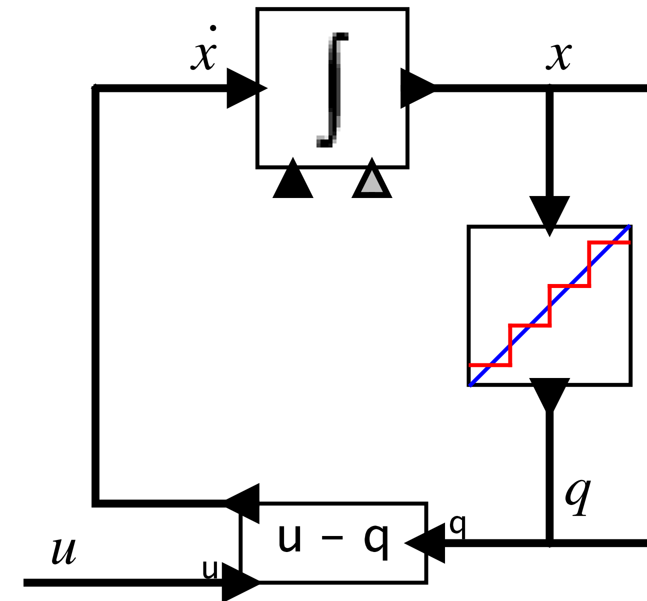
$$\dot{x}(t) = u(t) - q(t)$$

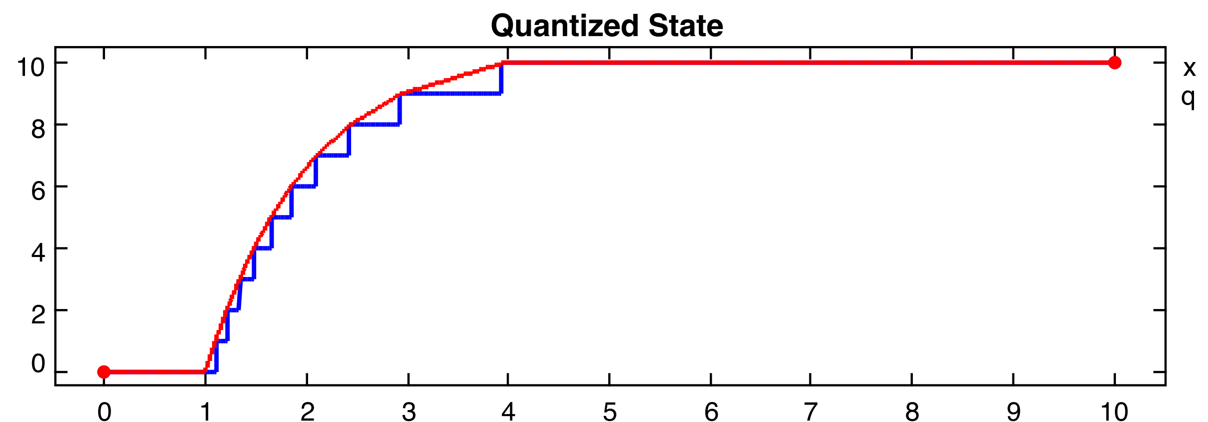
where

$$q(t) = \lfloor x(t) \rfloor.$$

Let

$$u(t) = \begin{cases} 0 & t < 1 \\ 10 & t \geq 1 \end{cases}$$





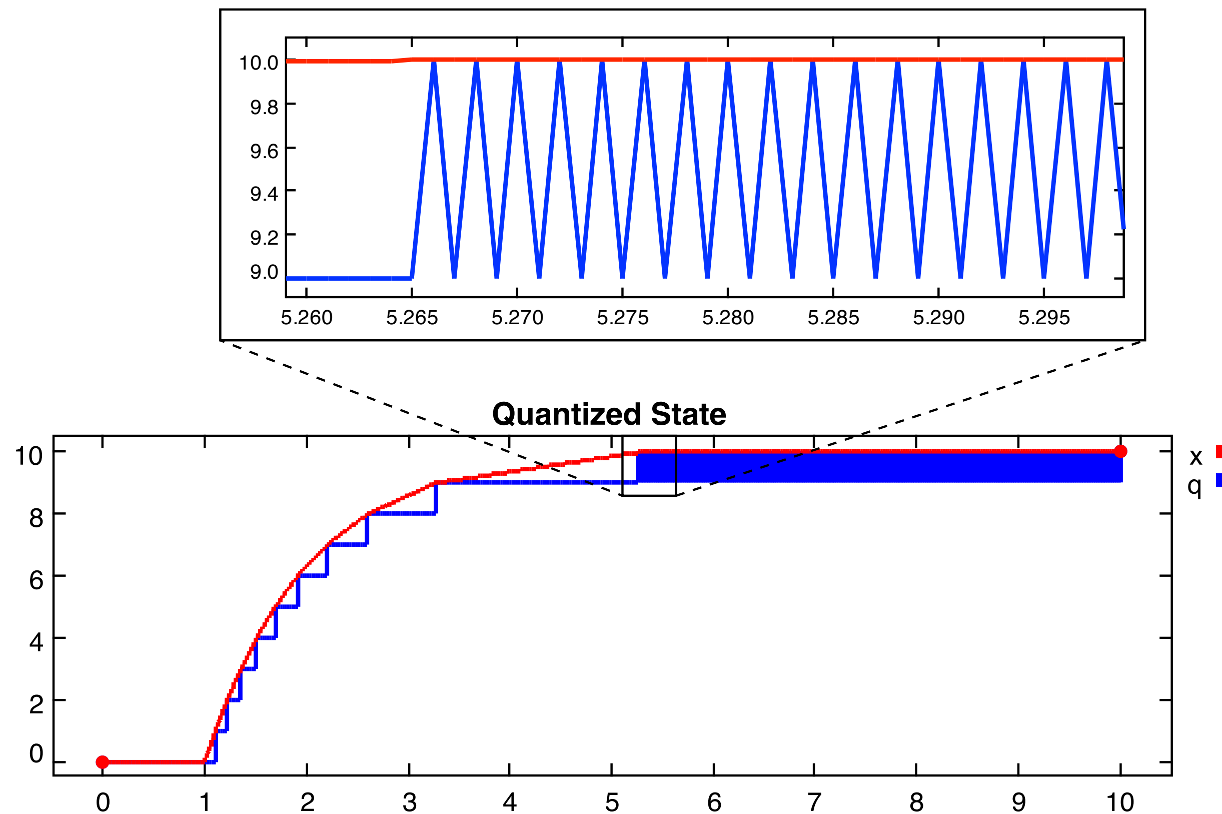
How does this behave if u is piece-wise constant?

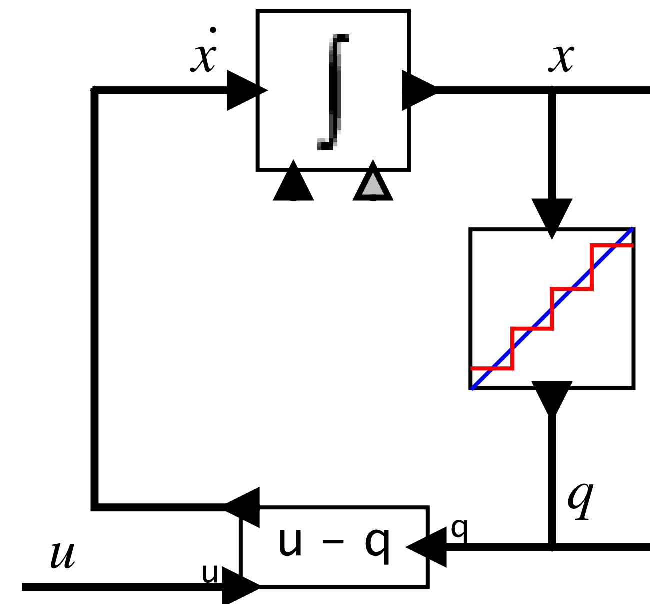
But this implementation can cause chattering

But it doesn't quite work. Suppose the input is instead

$$u(t) = \begin{cases} 0 & t < 1 \\ 9.5 & t \geq 1 \end{cases}$$

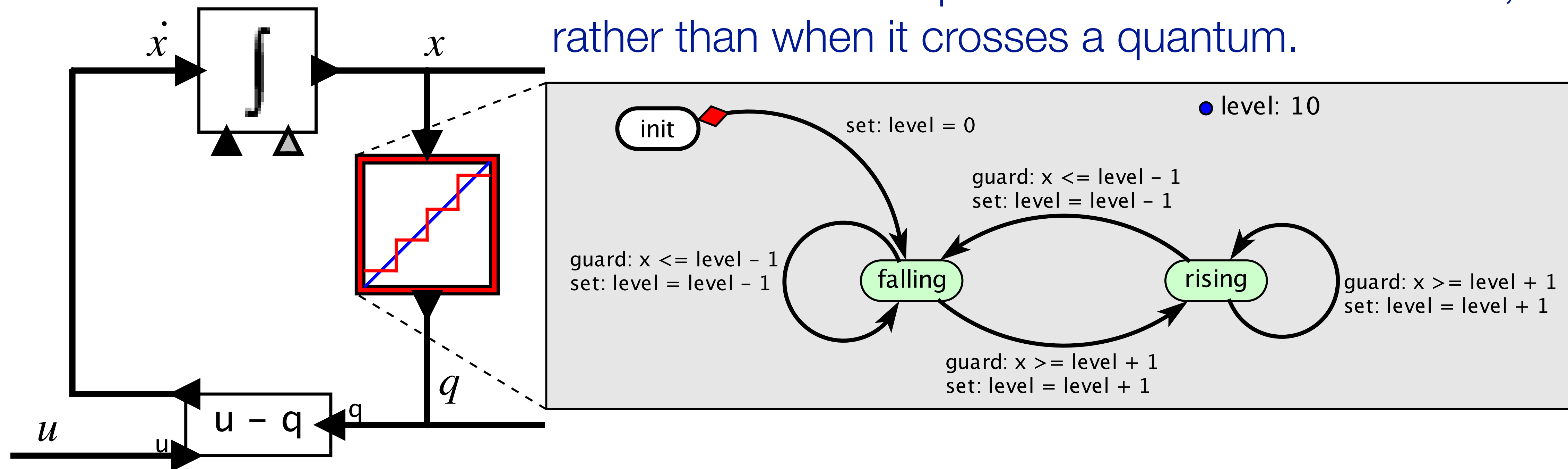
Then we get this:

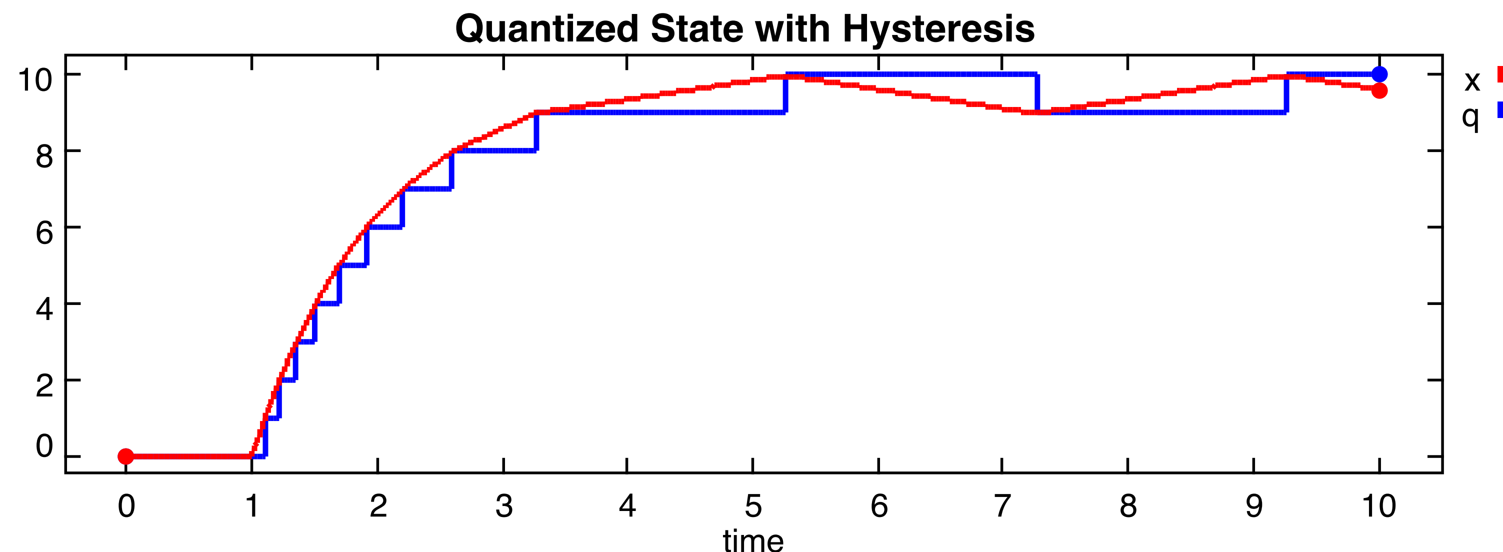




Need to have hysteresis to avoid chattering

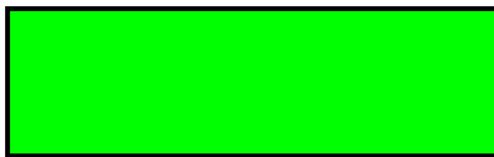
Change the output of the quantizer only if its input differs more than a quantum from the current level, rather than when it crosses a quantum.

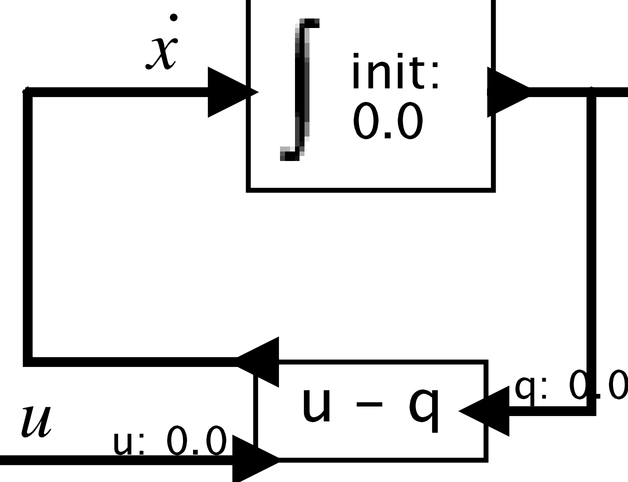


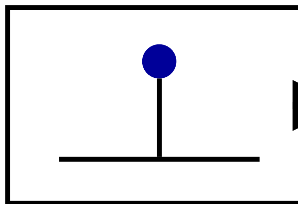


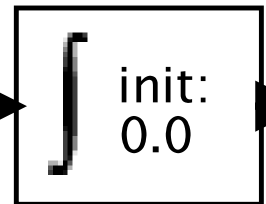
We just constructed a QSS1 method.

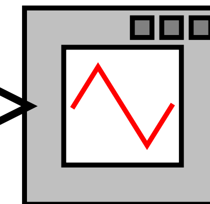
It is more convenient to package the quantizer in the integrator

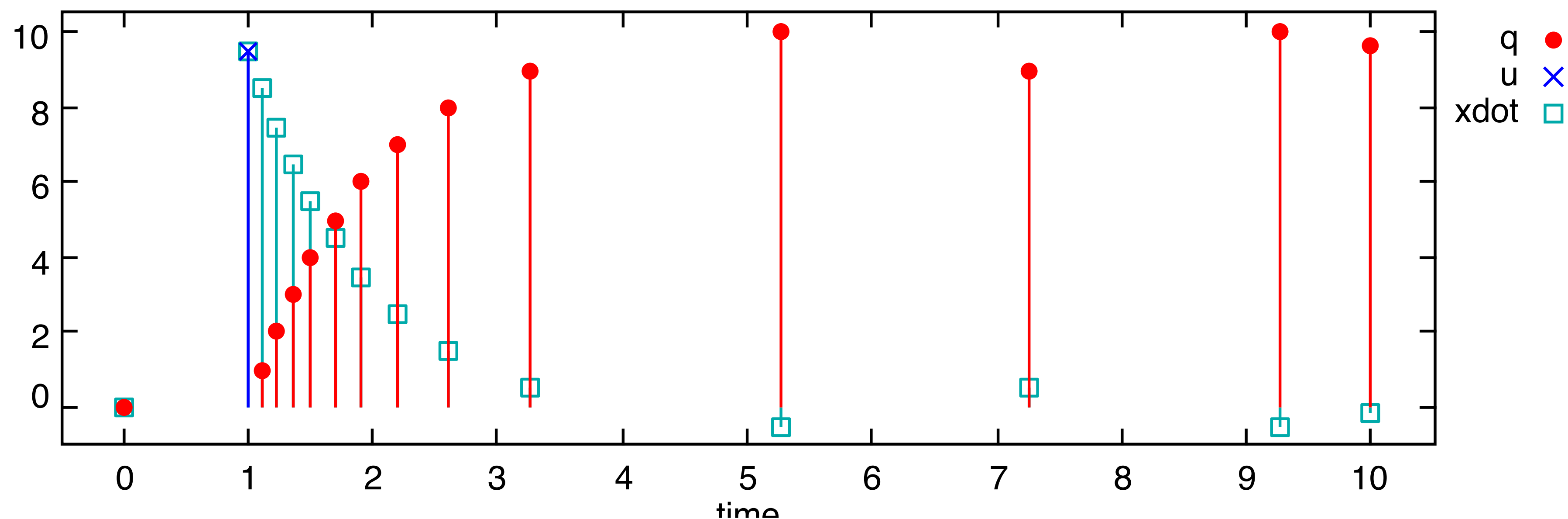












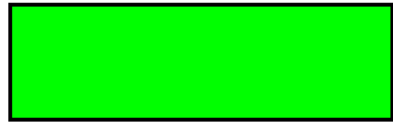
Various QSS variants

	QSS1	QSS2	QSS3	LIQSS{1, 2, 3}
Signal	Piece-wise constant	Piece-wise linear	Piece-wise quadratic	As for QSSx
Stiff systems	not suited	not suited	not suited	suited if stiffness appears on diagonal of Jacobian

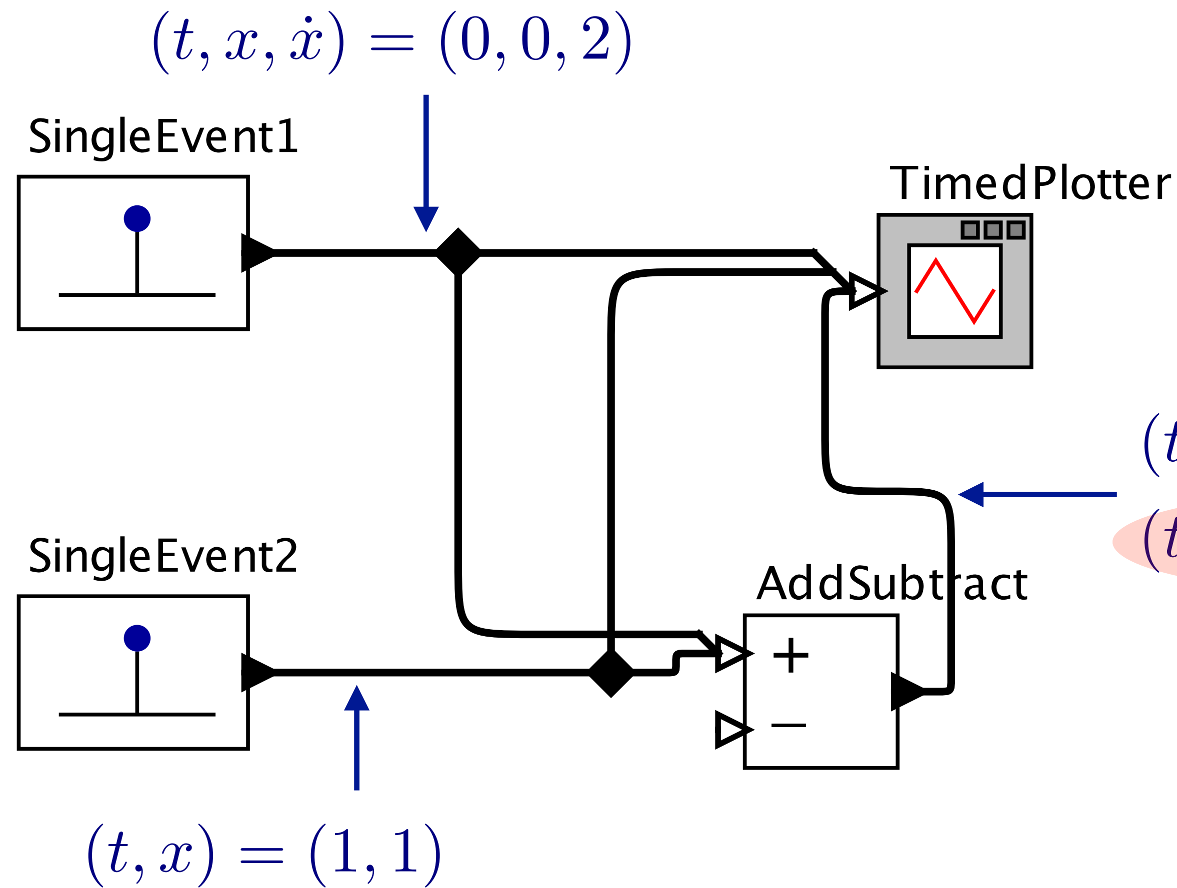
How do you represent these signals?

Backward QSS is another variant.

Need a “smooth token” that carries time stamp, value and derivative(s)



$(t, x, \dot{x}) = (0, 0, 2)$

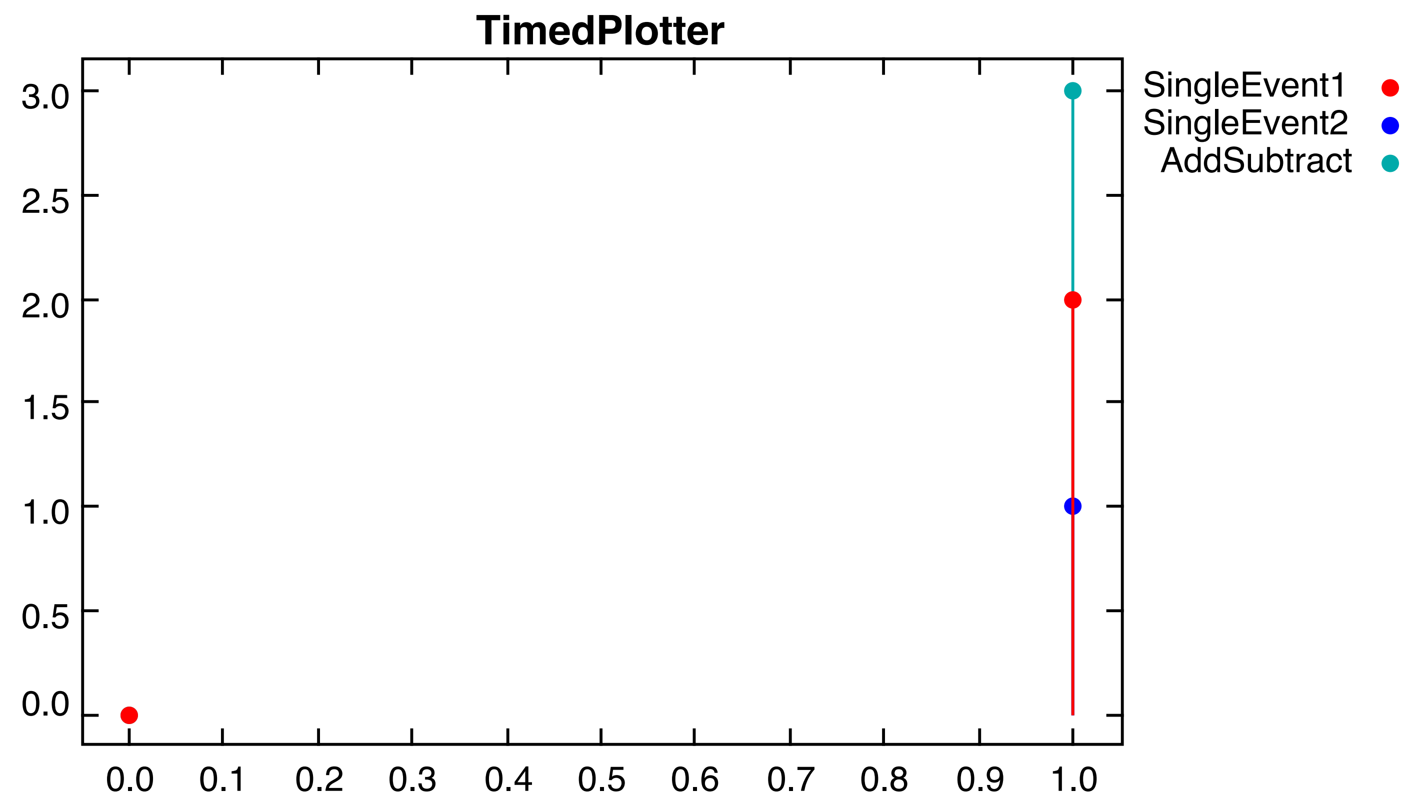


$(t, x) = (1, 1)$

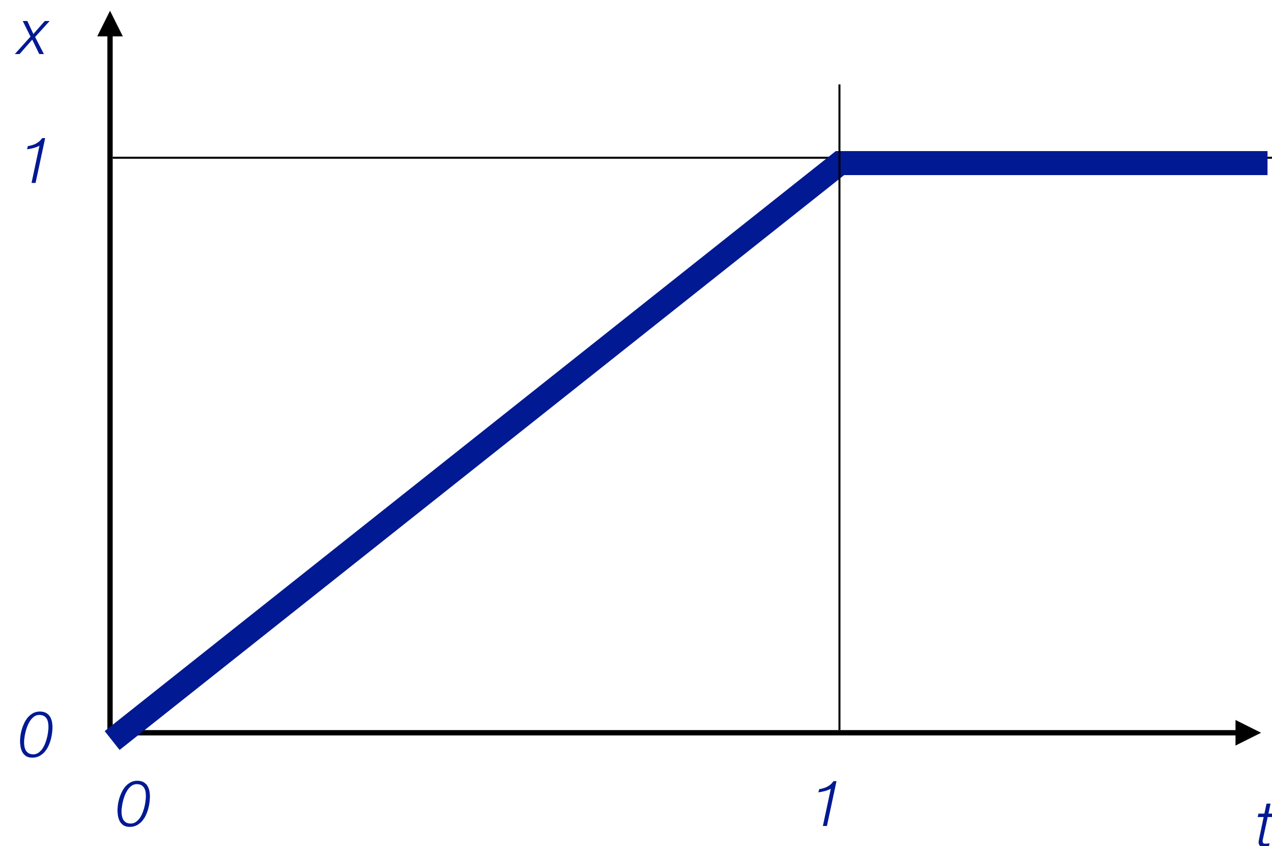
$(t, x, \dot{x})_1 = (0, 0, 2)$

$(t, x, \dot{x})_2 = (1, 3, 2)$

How is this produced?



State events do not require iterations



Suppose

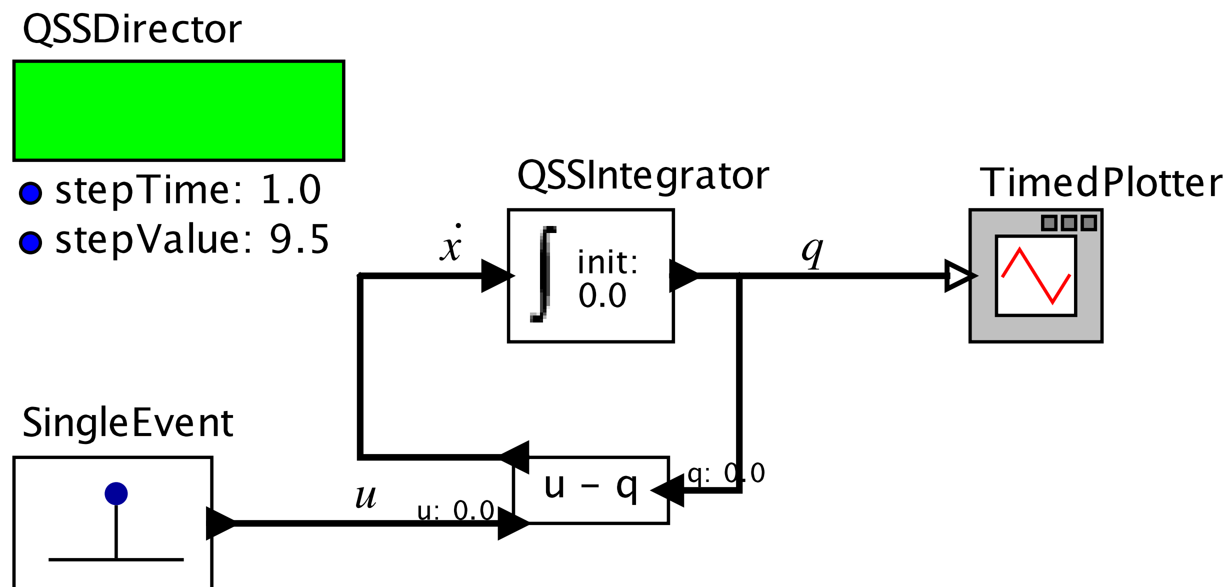
$$x(0) = 0$$

$$\dot{x}(t) = \begin{cases} 1, & \text{if } x(t) < 1, \\ 0, & \text{otherwise} \end{cases}$$

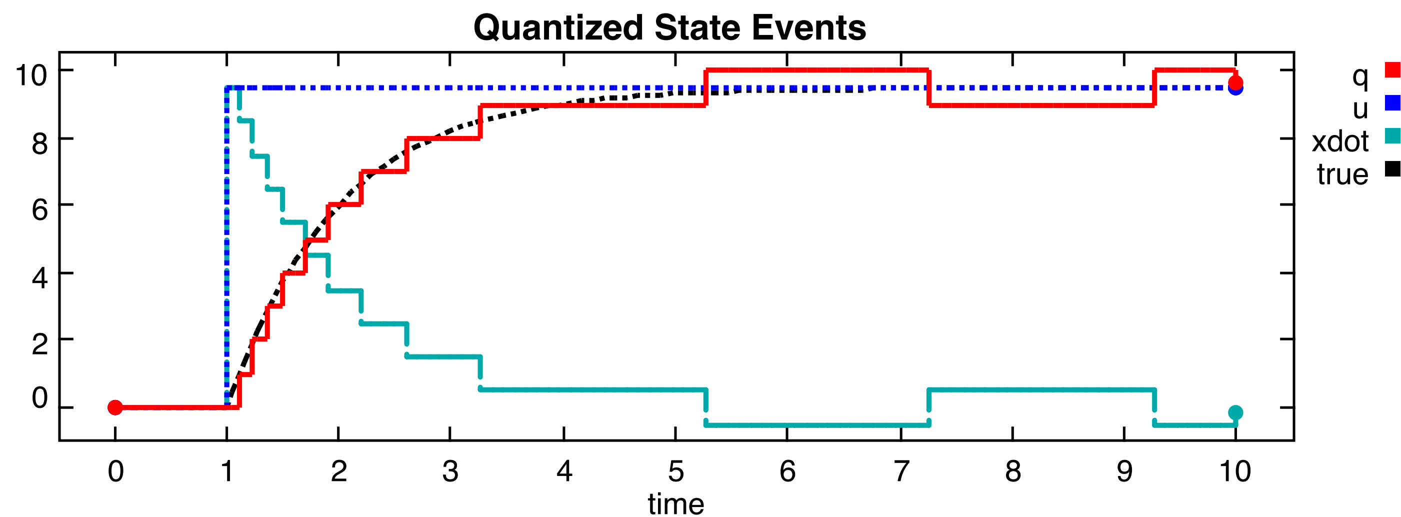
1. Compute *x(t) = 0 + 1 t*
2. Solve for *t*>0* that satisfies *x(t*) = 1*
3. Schedule an event at *t**
4. Reevaluate the differential equation at *t**
   to yield *x(t) = 1 + 0 t*
5. Schedule an event at t=infinity

In comparison, standard differential equation solvers would integrate until they achieve *x(t) > 1*, then iterate to find *t**, and then restart the integration from *t*.*

Oscillations around steady-state



If the input is piecewise constant, but doesn't match the quantum, the system may oscillate around the steady state.



Optimizing QSS for such situations may be an important research topic.

A simple heat transfer problem with feedback control

$$C\,I\,\frac{dT(t)}{dt} = \begin{pmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & -2 & -2 \end{pmatrix} UA\,T(t) + \begin{pmatrix} UA\,T_{amb} + \dot{Q}\,y(t) \\ 0 \\ 0 \end{pmatrix}$$

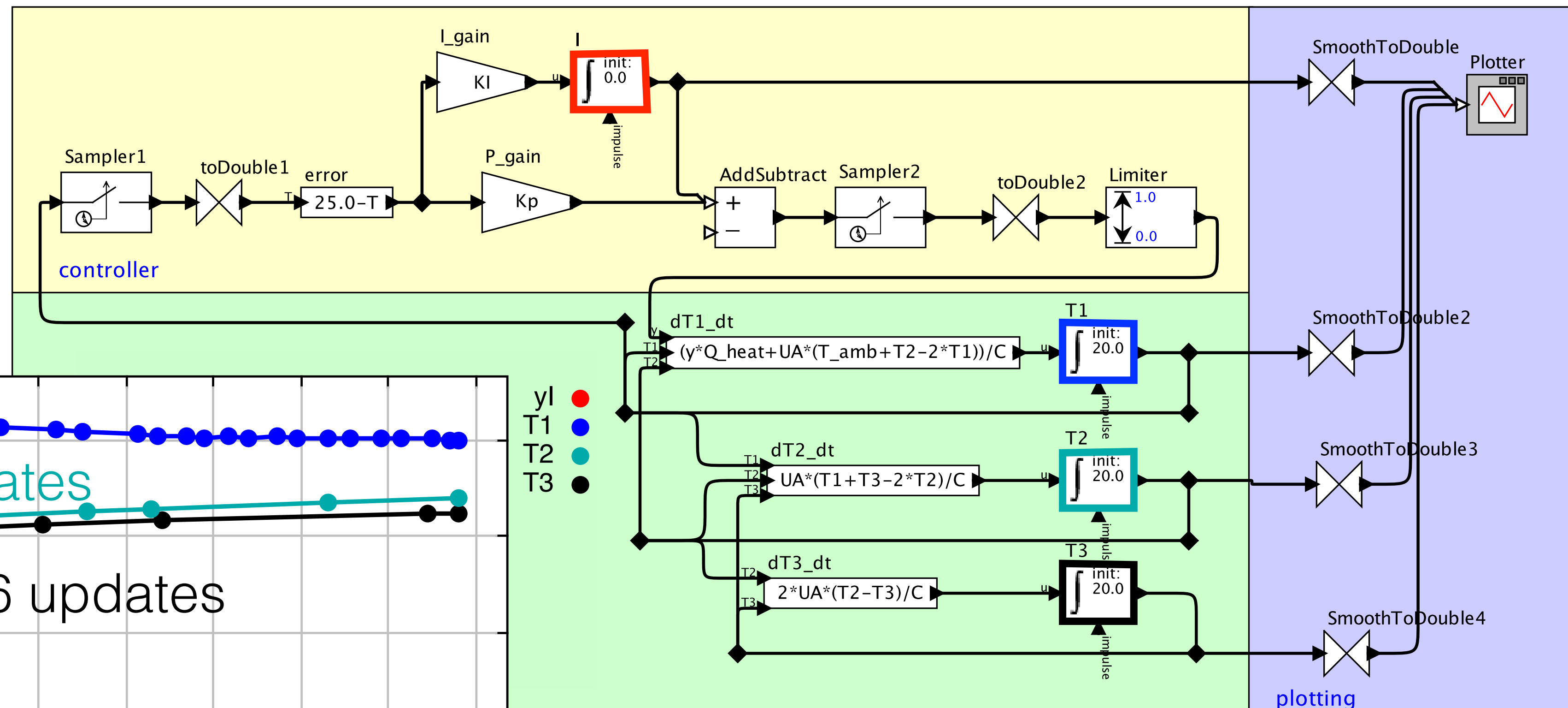
CyPhy Director

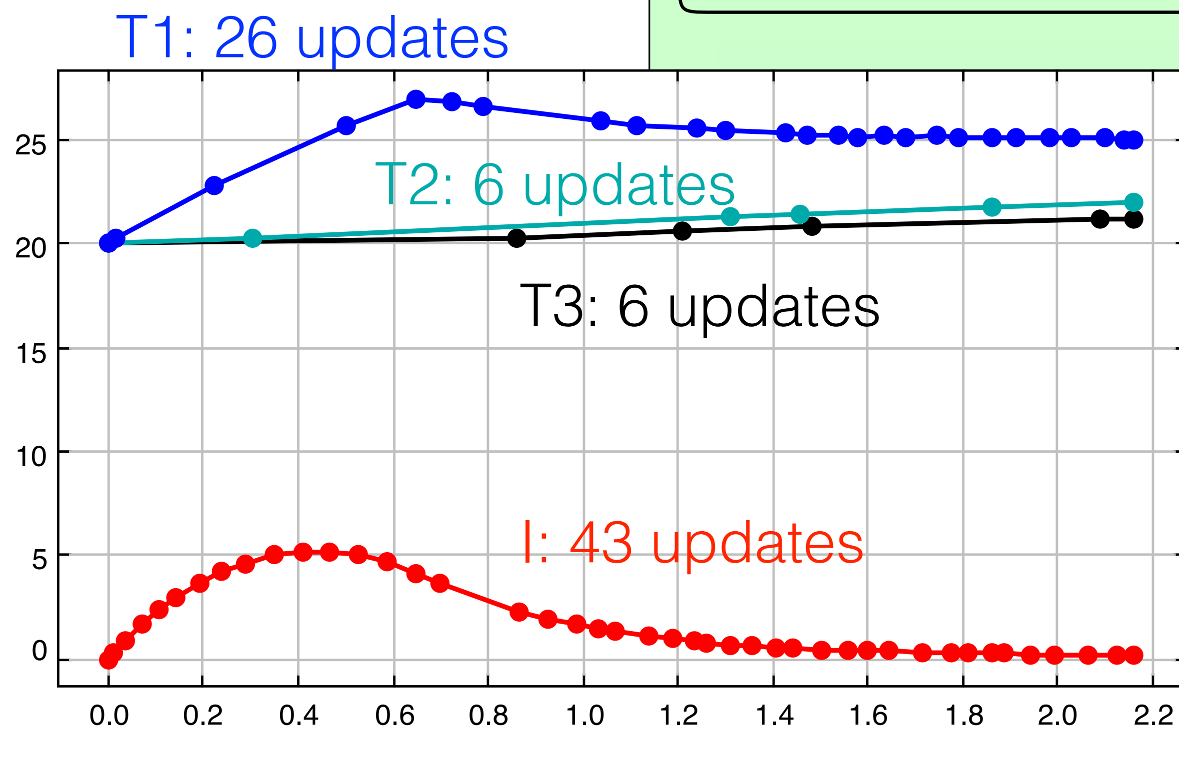
- stopTime: 6*3600
- samplePeriod: 120.0
- Kp: 2.0
- KI: 0.0005
- UA: 5.0
- C: 150000.0
- T_amb: 20.0
- Q_heat: 200.0

Author: Michael Wetter

Plant model with PI controller and discrete time sampling of its input and output.
The plant has a thermal mass, approximated using finite differences
with three states. It has heat loss to T_amb and a heat gain.
The heat gain is controlled by the PI controller.
After the sampler, the derivatives of the tokens are discharged by converting the
token to a double, as samplers only output values and not derivatives.

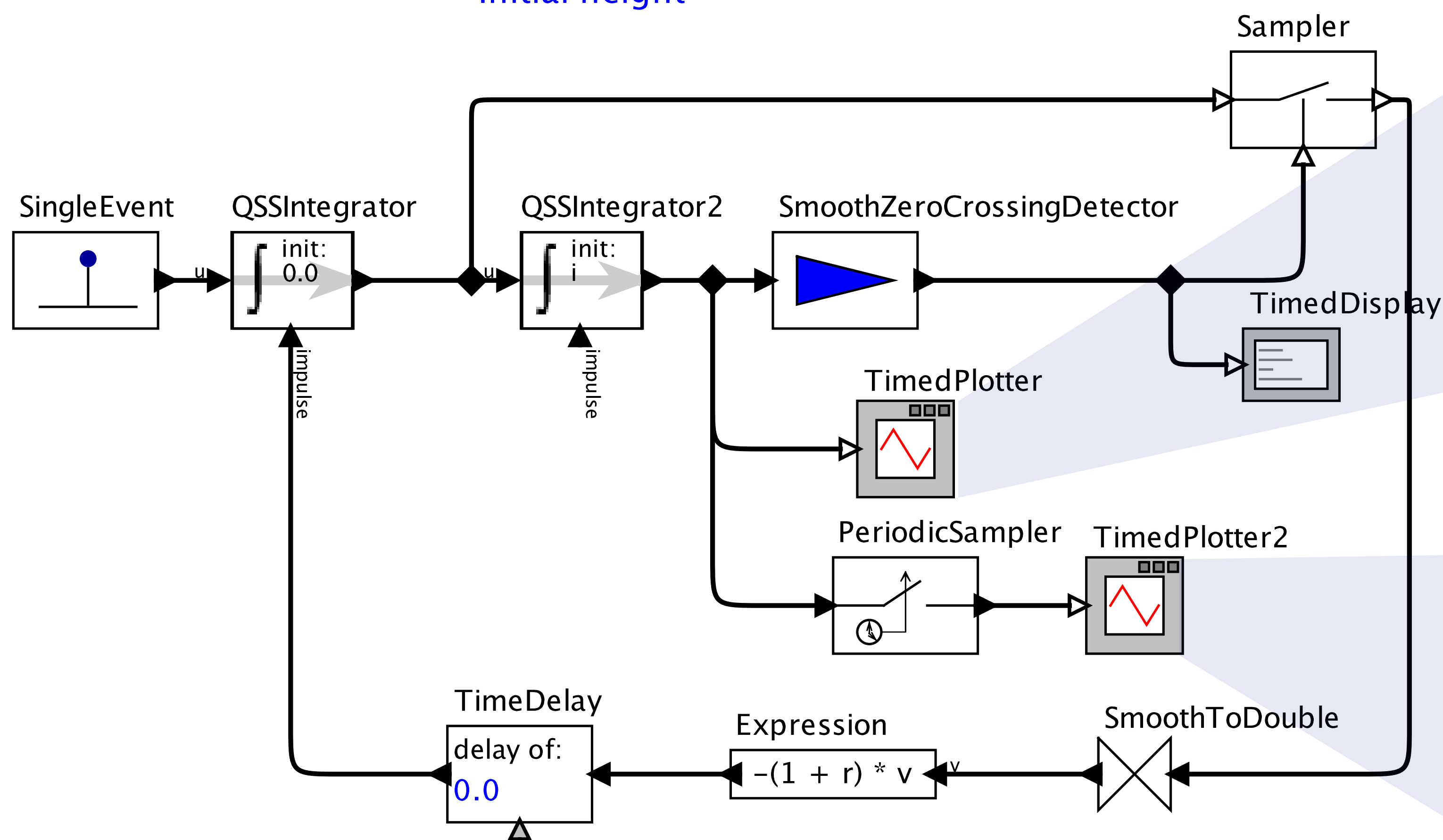
This model illustrates how QSS methods decouple state variables. The dots on the plot
show that the slow varying states are updated much less frequent than the control signal.

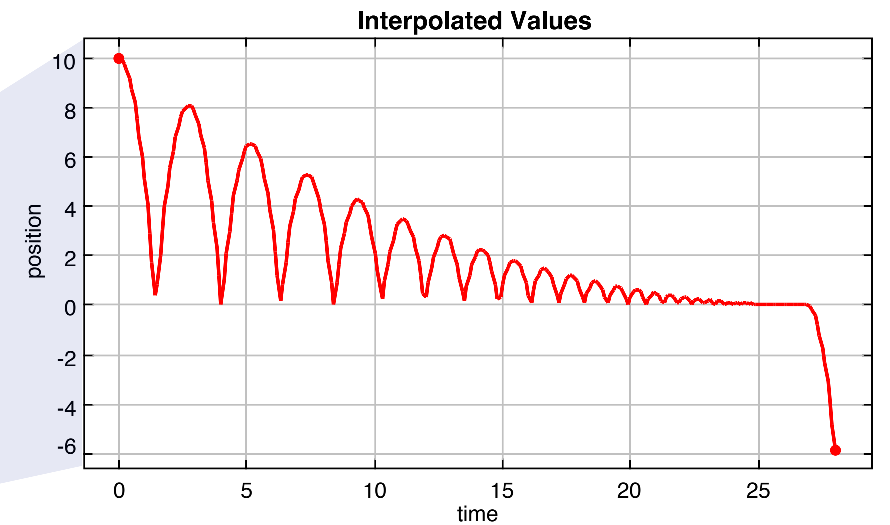


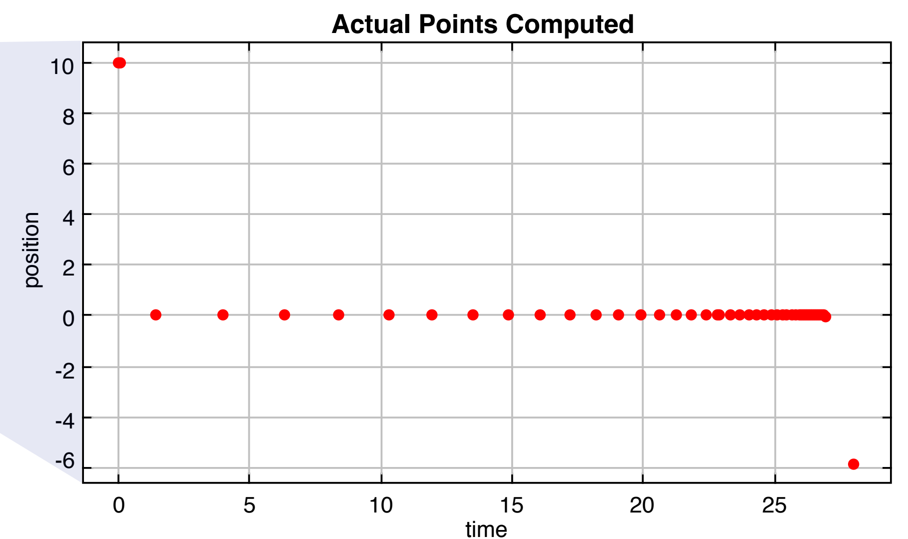


Controller samples: 181.
In comparison, RK requires 362 state updates.

Bouncing ball simulated with one integration step per impact, and exact computation of impacts without any iteration







QSS: 46 points
RK23: 14,072 points (not counting rejected steps)
QSS was about 38 times faster.

Modeling and Simulating Cyber-Physical Systems using CyPhySim
Lee, Ninami, Nouidui, Wetter, in review.

Numerical benchmarks versus EnergyPlus 8.2 and Dymola 2015 FD01

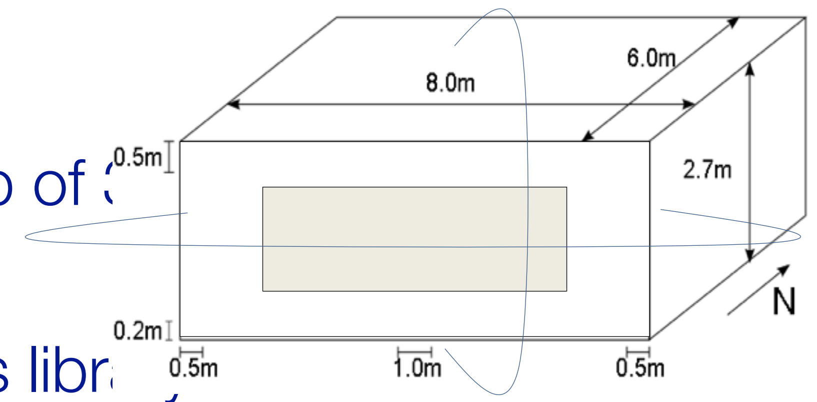
A cell contains a room, the slab with pipes and a controller.

Climate: Chicago, IL

Simulation time: One year

Simulation environments

- EnergyPlus with Conduction Finite Difference and time step of
  which is the largest time step that gives stable results.
- Dymola with dassl and models from the Modelica Buildings libr
- Ptolemy II with QSS2 and tolerance of 1E-3.
  FMUs generated from models from the Buildings library.

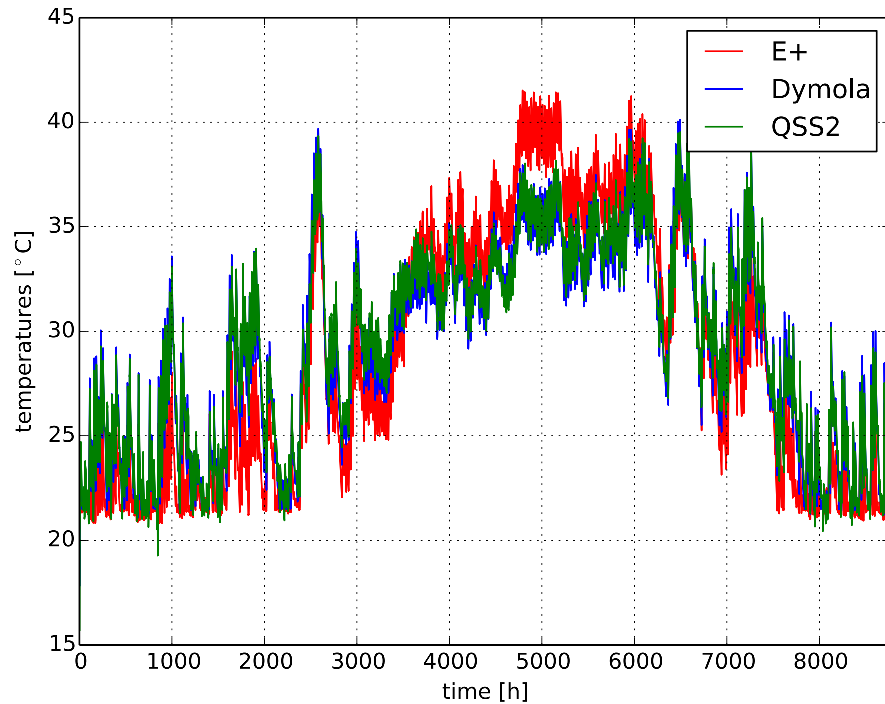


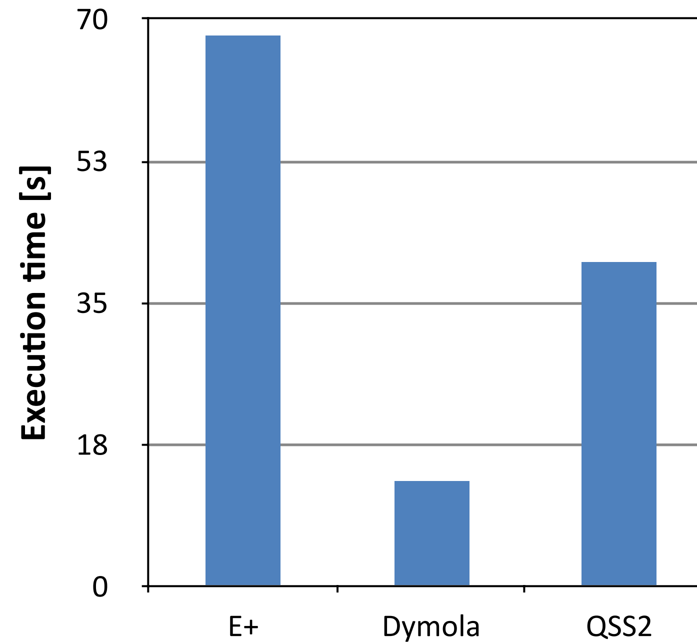
1 cell experiments
using 5 FMUs,

9 cell experiments
using 45 FMUs

Results and execution times

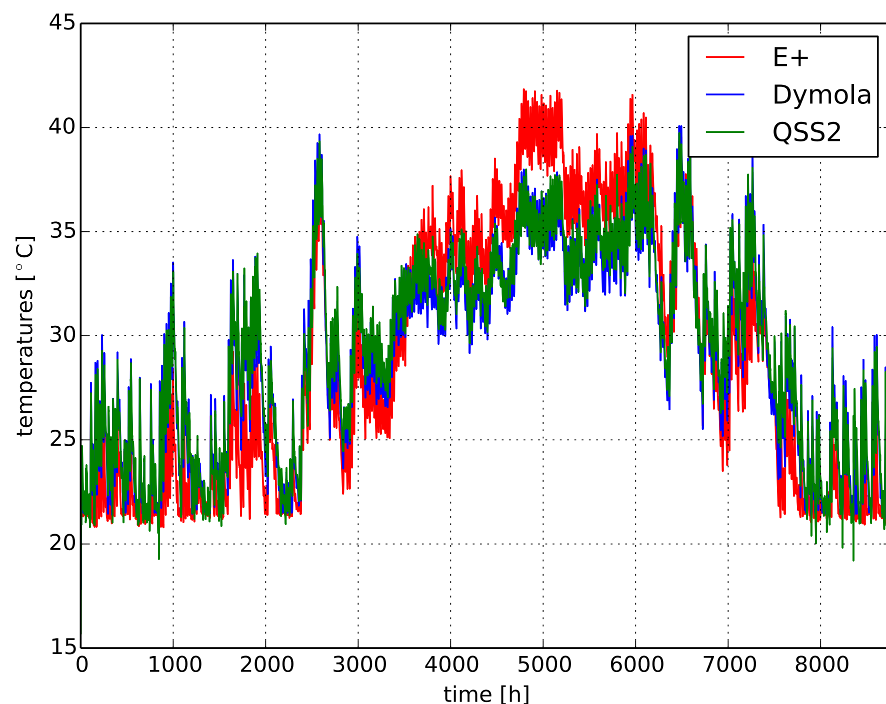
1 cell (5 FMUs)

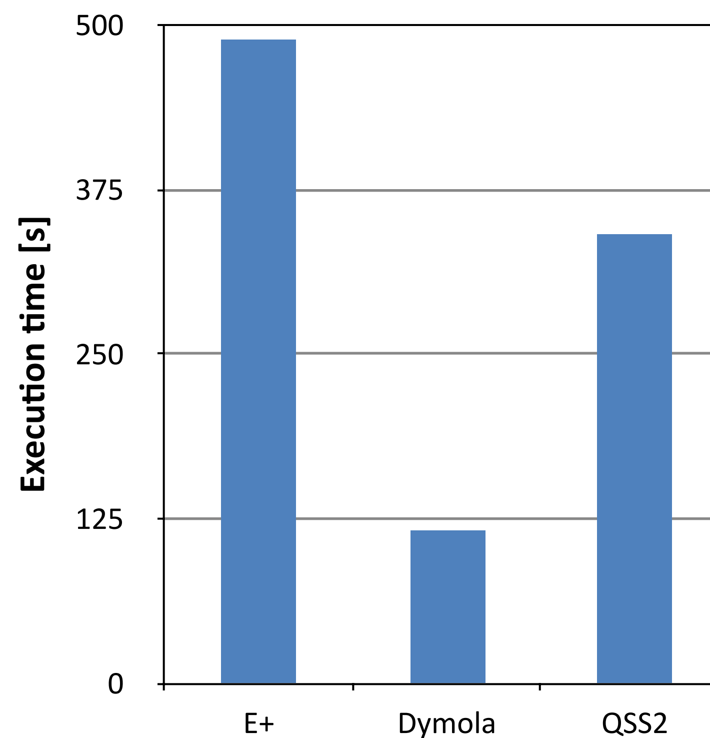


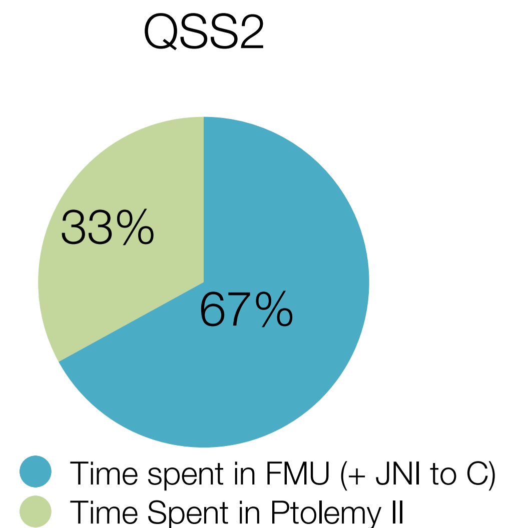


For high performance, need QSS implemented in Modelica to C compiler.

9 cells (45 FMUs)







Properties of QSS

Favoring QSS:

- State events are predictable. No iteration is required to find them.
- Step sizes are predictable. No need to reject step sizes and backtrack.
- For some models, QSS is computationally exact.
  There is no numerical approximation due to integration.
- States are integrated asynchronously.
- Computing time grows linear in the number of state variables.
- Global error bound can be computed explicitly for asymptotically stable linear time-invariant systems.
- LIQSS accounts for stiffness that appears on the diagonal of the Jacobian.
- Algebraic loops are solved locally, only when they are triggered by a state transition or an input function (Cellier & Kofman, p. 582)

Favoring classical ODE solvers:

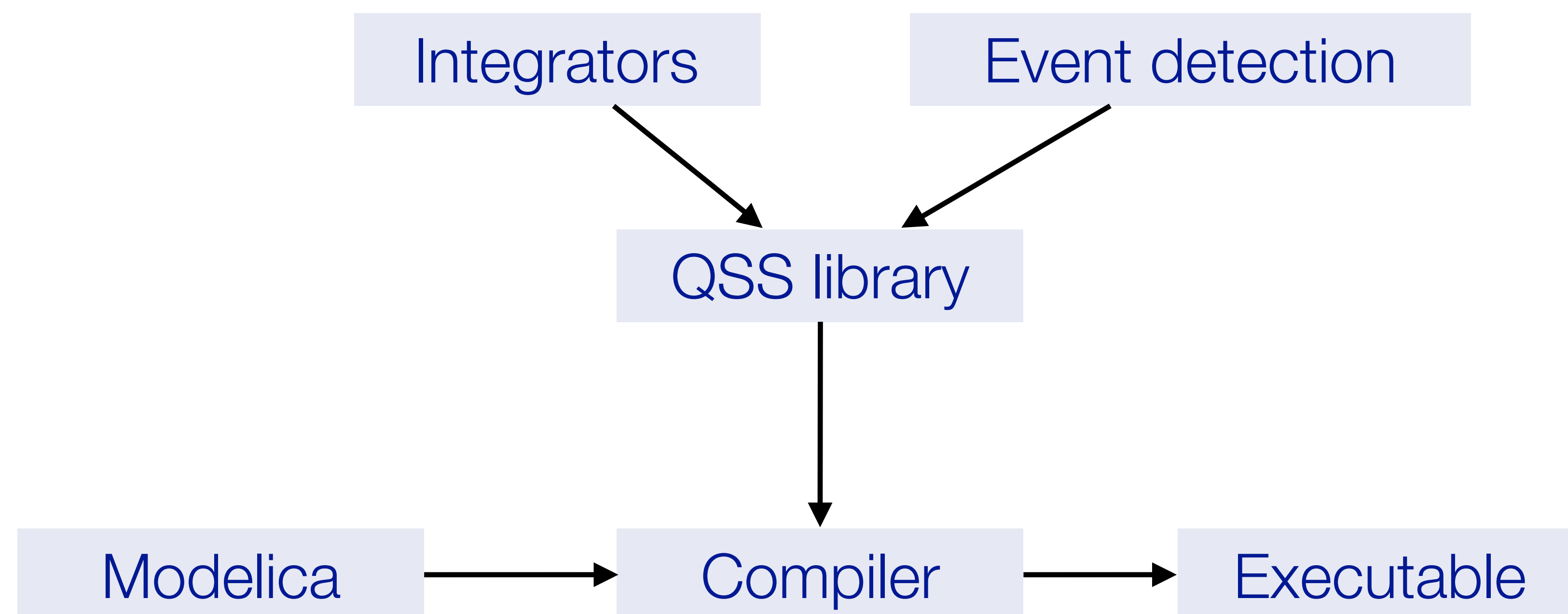
- Inputs may not be quantized.
- Feedback systems may oscillate around a steady state.
- Certain stiff systems.

QSS are rather new methods and we need to have a larger set of test cases to evaluate them rigorously.

Implementing QSS into the Modelica compiler is likely leading to higher performance

Fernandez and Kofman (2014) report an order of magnitude faster simulation by using special QSS simulator rather than PowerDEVS.

(And two orders faster than OpenModelica.)

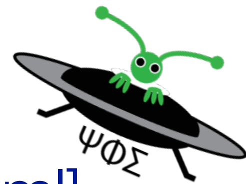


This should be a (longer term) activity.

To get started

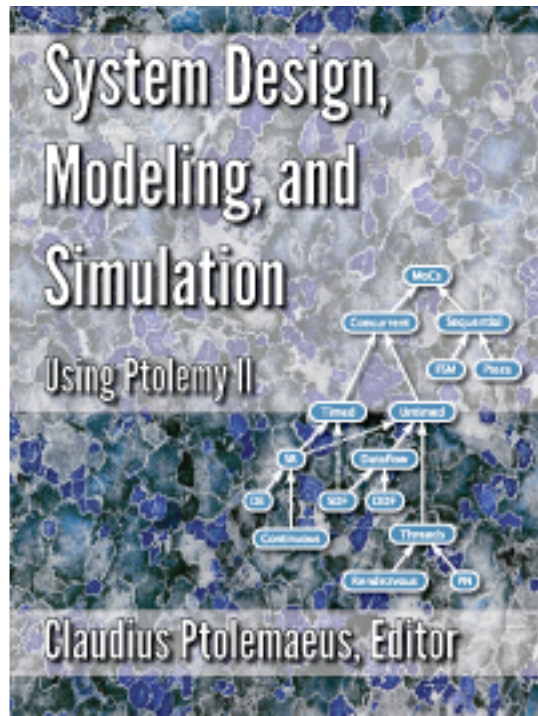
Download Ptolemy

Start



- cyphysim [model.xml] (Subset of Ptolemy II which we use for SOEP)
- vergil [model.xml], or (Ptolemy II with GUI)
- ptexecute model.xml (Ptolemy II as a console application)

References



Free Ptolemy II book: http://ptolemy.eecs.berkeley.edu/books/Systems/

David Broman, Lev Greenberg, Edward A. Lee, Michael Massin, Stavros Tripakis and Michael Wetter.
Requirements for Hybrid Cosimulation Standards.
18th International Conference on Hybrid Systems: Computation and Control (HSCC 2015), Seattle, WA, April 2015.

Christopher Brooks, Edward A. Lee, David Lorenzetti, Thierry S. Nouidui and Michael Wetter.
Demo: CyPhySim - A Cyber-Physical Systems Simulator.
18th International Conference on Hybrid Systems: Computation and Control (HSCC 2015), Seattle, WA, April 2015.

David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter.
Determinate Composition of FMUs for Co-Simulation.
Proc. of the International Conference on Embedded Software (EMSOFT 2013), p. 1--12, Montreal, Canada, 2013.

Summary

Ptolemy II

- Actors communicate with each other by sending tokens to ports.
- A director synchronizes the execution.

DE

- Signals are only defined at certain (superdense) time instants

QSS

- The state is discretized, not time.
- Leads to a discrete event simulation.
- Linear scaling in problem size and explicit event handling.

?
ΨΦΣ