Controller Design Using Multiple Models of Computation

Edward Lee

Johan Eker

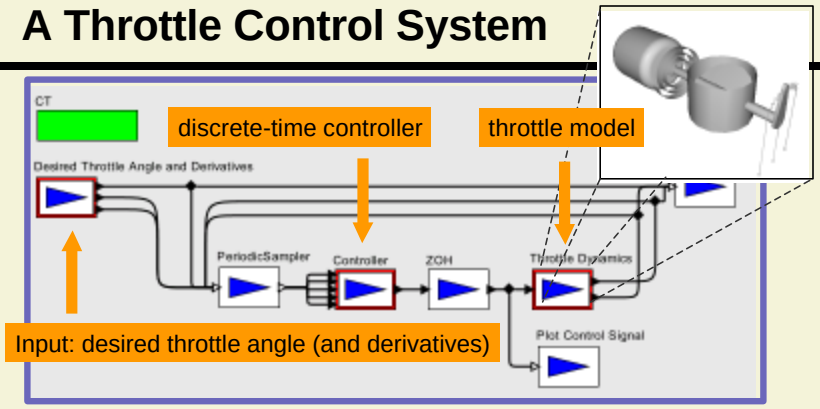
with thanks to Paul Griffiths, Jie Liu, Xiaojun Liu, Steve Neuendorffer, and Yuhong Xiong

MoBIES PI-Meeting, 16-18 July 2001, Jackson Hole

Overview

- Mixed signal control systems
  - The throttle control system by Paul Griffith
- Modal controllers
  - Extended throttle controller by Johan Eker
- Design patterns
  - The use of domains and hierarchy
  - Mixed signal models
  - Modal models
- Code generation
  - co-compilation

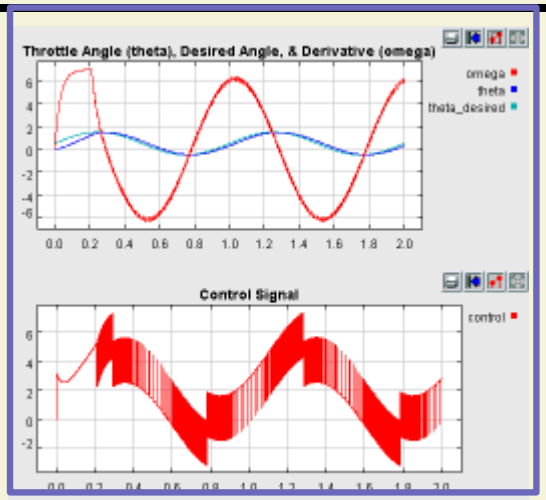
A Throttle Control System



Throttle controller by Paul Griffiths
(Mobies Phase II, UC Berkeley)

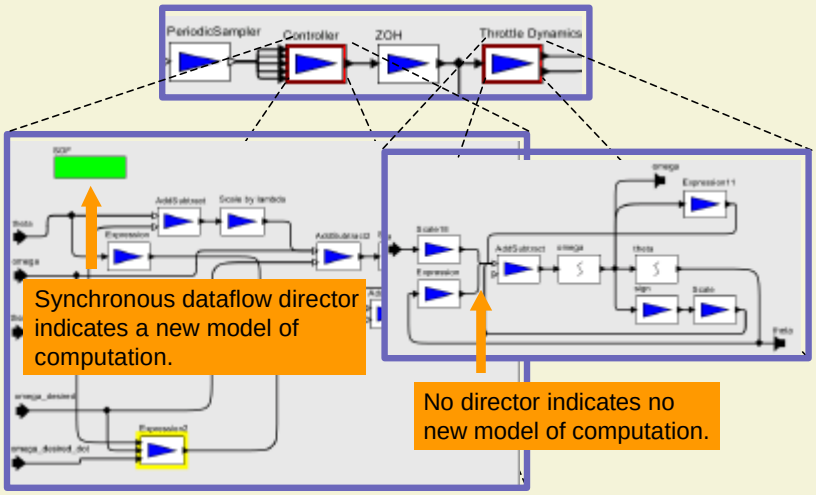
Top-level model
Continuous-time (CT) domain in Ptolemy II

A Throttle Control System



Result of executing the model shows intial convergence phase followed by tracking with a slight phase lag. The control signal chatters to overcome friction.

Design Pattern: Discrete Time in Continuous Time



Control Engineer View

- plant dynamics in continuous time
- controller in discrete time
- focus on stability, phase margins, rise time...
- assume ideal sampling with no or little latency

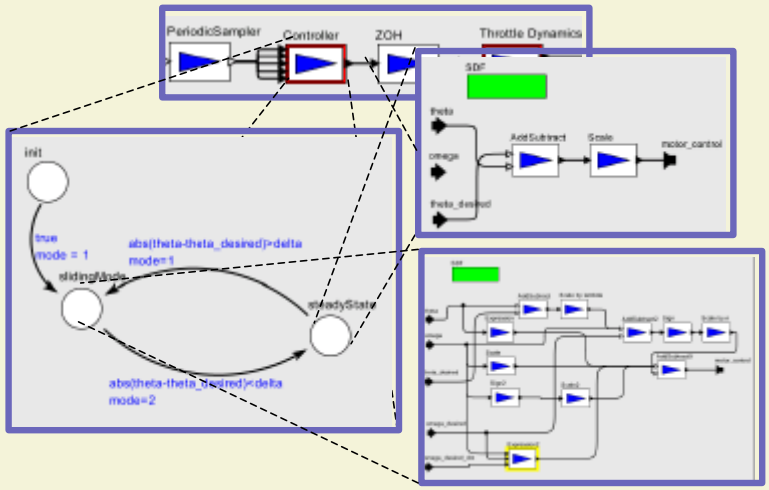
Embedded System Engineer View

- dynamics modeled with RK-4, variable-step solver
- controller modeled in synthesizable SDF, FRP, …
- focus on scheduling, memory, communication...
- assume fixed controller design

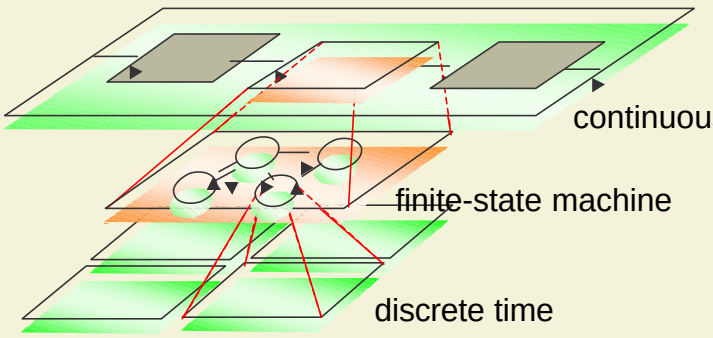
A More Integrated Approach

- controller design informed by software issues
  - domain-oriented modeling language
  - modeling = implementation
  - latency and jitter are part of the model
- software design informed by controller issues
  - expressing timing constraints
  - correct-by-construction synthesis
  - heterogeneous modeling

Elaborated Throttle Control System

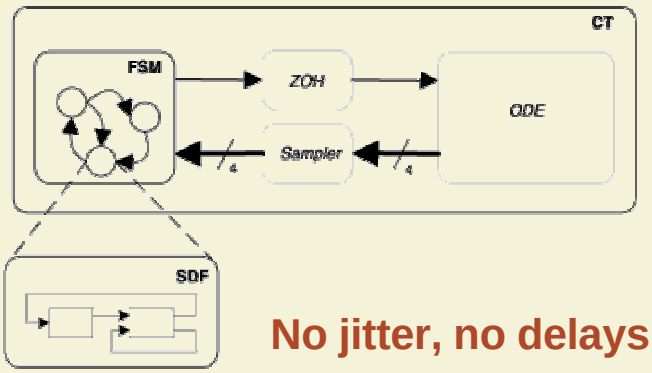


Design Pattern: Modal Discrete-Time Controller in Continuous Time



Hierarchical, heterogeneous model

This is Still An Idealized Model

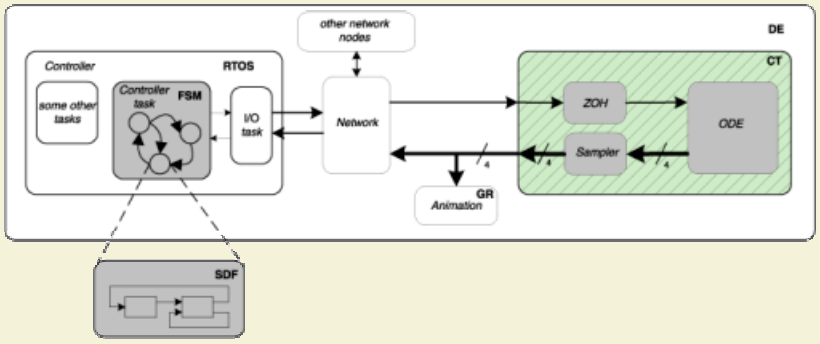


No jitter, no delays

Extend the ideal model

- Influence from implementation:
  - Jitter
  - Control delay
- Execution:
  - Multitasking environment
  - Incorporate the behavior of the RTOS
- Communication:
  - Shared communication links
  - Behavior of the network

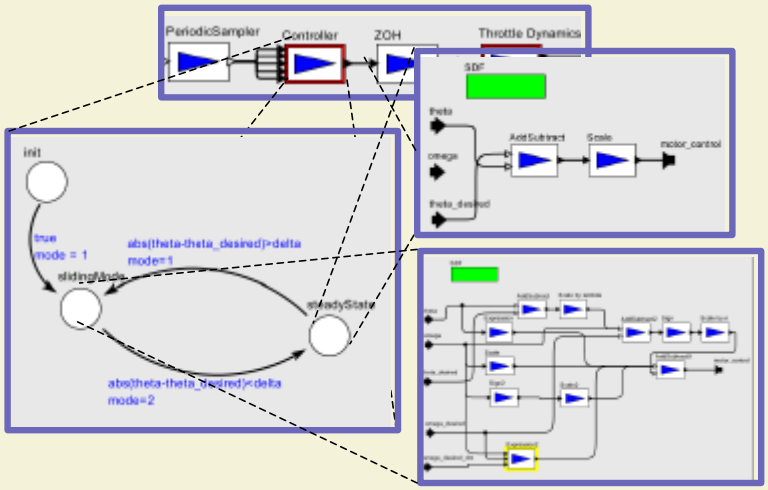
A more accurate model



Software Realism

- Express timing constraints
  - sample rates
  - latency
  - jitter tolerances
- Build models in appropriate abstractions
  - Giotto: time-triggered
  - HPM: hierarchical preemptive multitasking
  - FRP: functional-reactive programming
  - …
  - These facilitate correct-by-construction implementation

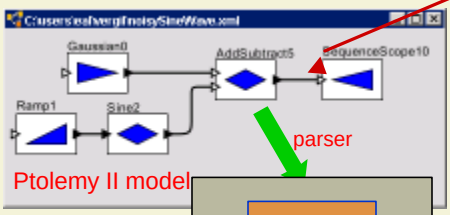
The Next Problem: Synthesizing an Implementation



Outline of our Approach

Jeff Tsay,
Christopher Hylands,
Steve Neuendorffer

Model of Computation semantics defines communication, flow of control



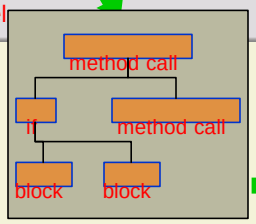
Ptolemy II model

parser

scheduler

Schedule:
- fire Gaussian0
- fire Ramp1
- fire Sine2
- fire AddSubtract5
- fire SequenceScope10

All actors are given in Java, then translated to embedded Java, C, VHDL, etc.



abstract syntax tree

```
for (int i = 0; i < plus.getWidth(); i++) {
  if (plus.hasToken(i)) {
    if (sum == null) {
      sum = plus.get(i);
    } else {
      sum = sum.add(plus.get(i));
    }
  }
}
```

target code

Division of Responsibility

- MoC semantics defines
  - flow of control across actors
  - communication protocols between actors
- Actors define:
  - functionality of components
- Actors are compiled by a MoC-aware compiler
  - generate specialized code for actors in context
- Hierarchy:
  - Code generation at a level of the hierarchy produces a new actor definition

We call this *co-compilation.*

Multiple domains may be used in the same model

Software Progress

Build on:

- First version on Titanium (UC Berkeley)
- Second version on Soot (McGill)

Targeting:

- Simulation acceleration
- Embedded software synthesis
  - Maryland subcontract
- Configurable hardware synthesis
  - delegated to Brigham Young

Our Generator Approach

- Actor libraries are built and maintained in Java
  - more maintainable, easier to write
  - polymorphic libraries are rich and small
- Java + MoC translates to target language
  - concurrent and imperative semantics
- Efficiency gotten through code transformations
  - specialization of polymorphic types
  - code substitution using MoC semantics
  - removal of unnecessary code

Code transformations (data types)

```
// Original actor source
Token t1 = in.get(0);
Token t2 = in.get(1);
out.send(0, t1.multiply(t2));
```

specialization of Token declarations

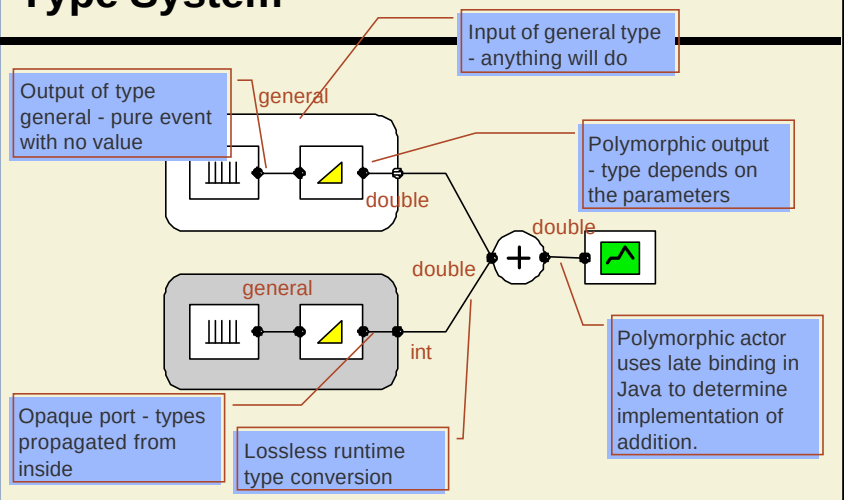
```
// With specialized types
IntMatrixToken t1 = in.get(0);
IntMatrixToken t2 = in.get(1);
out.send(0, t1.multiply(t2));
```

The Ptolemy II type system supports polymorphic actors with propagating type constraints and static type resolution. The resolved types can be used in optimized generated code.

See Jeff Tsay, *A Code Generation Framework for Ptolemy II*

Type system by Yuhong Xiong

Type System



Type System

- Extensible type lattice
  - Knowledgeable users can add full-featured types
- Unification infrastructure
  - Finds a least fixed point
- Composite types
  - records, arrays, matrices
- Higher-order types planned
  - model = data
- Experiments with dependent types
  - encoding MoC constraints

Code transformations (MoC-informed)

```
// With specialized types
IntMatrixToken t1 = in.get(0);
IntMatrixToken t2 = in.get(1);
out.send(0, t1.multiply(t2));
```

MoC-polymorphic code is replaced with specialized code.

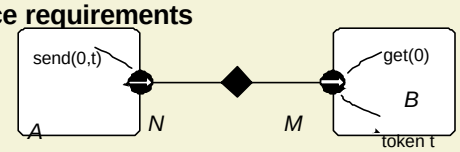
transformation using MoC semantics

```
// Extended Java with specialized communication
int[][] t1 = _inbuf[0][_inOffset = (_inOffset+1)%5];
int[][] t2 = _inbuf[1][_inOffset = (_inOffset+1)%5];
_outbuf[_outOffset = (_outOffset+1)%8] = t1 * t2;
```

See Jeff Tsay, *A Code Generation Framework for Ptolemy II*

Synchronous Dataflow (SDF) Domain

- **Balance equations (one for each channel):**
  $F_A N = F_B M$
- **Scheduled statically**
- **Decidable resource requirements**



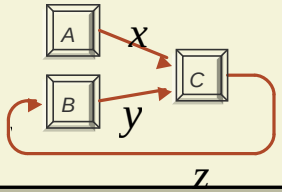
Available optimizations:

- **eliminate checks for input data**
- **statically allocate communication buffers**
- **statically sequence actor invocations (and inline?)**

Synchronous/Reactive Domain

- A discrete model of time progresses as a sequence of "ticks." At a tick, the signals are defined by a fixed point equation:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_t = \begin{pmatrix} f_{A,t}(1) \\ f_{B,t}(z) \\ f_{C,t}(x,y) \end{pmatrix}$$



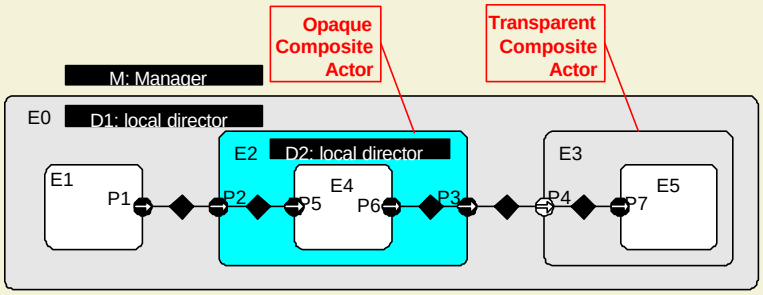
Available optimizations:
- **Statically sequence fixed-point iteration**
- **Communication via registers**

Other Domains with Useful Properties for Code Generation

- **Strong static analyzability**
  - Giotto (time triggered)
  - HPM (hierarchical preemptive multitasking)
  - FRP (functional reactive programming – Yale)
  - Finite state machines
  - Discrete time
- **Good for hardware descriptions**
  - Discrete events
  - Process networks
  - Continuous time (analog hardware)

Hierarchical Heterogeneity

Ptolemy II composes domains hierarchically, where components in a model can be refined into subcomponents where the component interactions follow distinct semantics.



Conclusions

- Hierarchically heterogenous modeling matches the applications well…
- Hierarchically heterogenous modeling appears to be suited to high-quality sythesis