Partial vs. Total Order a.k.a Polychrony vs. Synchrony

Models of Time for Safety Critical Systems

Sandeep K. Shukla
FERMAT Lab
Hume Center for National Security and Technology
Virginia Tech Arlington Research Center
Arlington, VA.

MBSE Colloq. at the University of Maryland

This work is partially supported by funds from AFRL and OSD
Ivan Sutherland,” The Tyranny of the Clock – Promoting a clock-free paradigm that fits everything learned about programming since Turing”, Communications of ACM, October 2012.
Motivating this Talk

- Describe a partial ordered model of logical time – Polychrony
- Show some essential distinctions between synchronous programming (totally ordered logical time) and Polychrony
- Show a calculus of logical time as a calculus for deterministic implementation, provable refinement, and more
- A Polychronous methodology for distributed deterministic implementation of model-driven Cyber Physical System design

L-3 and VT will produce a Robust Industrial Strength Implementation of the Model Driven Synthesis Tool Based on this.
Outline of the talk

1. Motivation
2. Introduction
3. Concurrency and Multi-Threading
4. Distribution over Asynchronous Network
5. Concluding Remarks
Outline of the talk

1. Motivation
2. Introduction
3. Concurrency and Multi-Threading
4. Distribution over Asynchronous Network
5. Concluding Remarks
Motivation

- Cyber
  - Sampling/sensing
  - Compute based on control laws
  - Actuating
Motivation

- **Cyber**
  - Sampling/sensing
  - Compute based on control laws
  - Actuating

- **Physical**
  - Dynamic
  - Continuous
  - Multiple Modes (piecewise continuous)
What we will not talk About

- Modeling the Physical Dynamics as Dynamical System
- Adaptive Zero-crossing Issues
- Real-Time Scheduling of Reactions
- Higher Level Data Types and Extended Type System
- Constructive Semantics for Polychrony
- Combining Synchrony and Polychrony into one Framework – Onyx
- Visual Polychrony – EmCodeSyn Environment
- Extending class of synthesizable Polychronous Programs beyond weak endochrony
Outline of the talk

1. Motivation
2. Introduction
3. Concurrency and Multi-Threading
4. Distribution over Asynchronous Network
5. Concluding Remarks
PI Controller

Figure: Schematic of a car on sloping road

\[
\begin{align*}
    m \frac{dv}{dt} + cv &= F - mg \theta \\
    \frac{dv}{dt} + 0.02v &= u - 10\theta \\
    u &= k(v_r - v) + \int_0^t k_i(v_r - v(\tau)) d\tau
\end{align*}
\]
**PI Controller**

\[ m \frac{dv}{dt} + cv = F - mg\theta \]

\[ \frac{dv}{dt} + 0.02v = u - 10\theta \]

\[ u = k(v_r - v) + \int_0^t k_i(v_r - v(\tau)) d\tau \]

**Figure:** Schematic of a car on sloping road

\[ s^2 + (0.02 + k)s + k_i = 0 \]

\[ k = 2\zeta \omega_0 - 0.02 \]

\[ k_i = \omega_0^2 \]

\( \zeta \) is damping parameter

\( \omega_0 \) is undamped natural frequency

**Figure:** Block diagram of a car with cruise control
A PI Controller for Cruise Control

\[ u = k(v_r - v) + \int_0^t k_i(v_r - v(\tau)) \, d\tau \]

\[
\begin{align*}
L: & \quad S = 0; \\
\quad & \quad \text{Timer} = T; \\
\quad & \quad \text{while}(\text{Timer} \neq 0)\{ \\
& \quad \quad \text{Sample } v; \\
& \quad \quad S = S + (v_r - v)k_i; \\
& \quad \quad \text{Timer} = \text{Timer} - \tau \\
& \quad \quad \text{wait for } \tau \\
\quad & \quad \}\} \\
\quad & \quad \text{Sample } v; \\
& \quad \quad u = k \times (v_r - v) + S; \\
\quad & \quad \text{GOTO } \text{L};
\end{align*}
\]
Signals as Flows

\[ v_1, v_2, v_3, v_4, \ldots, v_n, v_{n+1}, \ldots, v_{2n}, \ldots, \ldots, \ldots \]

\[ e_1, e_2, e_3, e_4, \ldots, e_n, e_{n+1}, \ldots, e_{2n}, \ldots, \ldots, \ldots \]

\[ u_1, u_2, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots \]

FLOW RELATIONS

\[ e_m = v_r - v_m, \quad m = 1, 2, \ldots \]

\[ u_j = k(v_r - v_{2j}) + \sum_{k=(j-1)n+1}^{jn} k_i(v_r - v_k) \]

\[ j = 1, 2, \ldots \]

\[ v = \text{sampled velocity}, \quad e = \text{instantaneous error}, \quad u = \text{computed throttle input} \]
How to Compute the Thrust $u$

```plaintext
process CruiseControl(?real vr; !real u) {parameter vr,n,k,k;}
(
  | e := vr - v
  | last_count := count $ init 0
  | count:=(last_count + 1) when (last_count < n) default 0;
  | sum:= k*e when (count = 0) default ((sum $ init 0) + k*e)
  | u := (k*e + (sum $ init 0)) when (count = 0)
  |)
where
  real sum, e;
  integer count, last_count;
```
Timing Issues

- Sampling of a new velocity $v$ drives the computation.
- Computation of $e$, $count$, $sum$ are synchronized to sampling of $v$.
- Computation of $u$ is only a sub-sampling of the flow of $v$.
  - Only when $count = 0$.
- This is *almost synchronous programming*. 
Differences with Synchronous Programming

- Usually in imperative synchronous program
  - A tick indicates a new cycle of computation
    - Sampling of all signals are done at the tick
  - Values are computed as necessary
    - Those not computed are absent (Esterel), or contain default values (Quartz)
  - Whatever happens at the instigation of a tick until the next tick is a ‘reaction’
    - The duration is abstracted to a point (logical instant)
    - Logical instants are totally ordered
Handling Multiple Inputs

```plaintext
process CruiseControl(?real v; integer rpm; !real u)
  {parameter v_r,n,k,k_i,rpm_th,u_d}
  (| e := v_r - v
  | last_count := count $ init 0
  | count:=(last_count + 1) when (last_count < n) default 0;
  | sum:= k_i*e when (count = 0) default ((sum $ init 0) + k_i*e)
  | u := (k_i*e + (sum $ init 0)) when ((count = 0) when (rpm < rpm_th))←
                   default (u$ init u_d)
  | rpm ^= (count = 0)
  |)
where
  real sum, e;
  integer count, last_count;
```

Sandeep K. Shukla  FERMAT Lab  Hume Center for National Security and Technology  Virginia Tech Arlington Research Center  Arlington, VA.
Timing Issues

- Sampling of a new velocity $v$ drives the computation
- Computation of $e$, $count$, $sum$ are synchronized to sampling of $v$
- Computation of $u$ is only a sub-sampling of the flow of $v$
  - only when $count = 0$ and the sampled $rpm$ is below a threshold $rpm_{th}$
- The sampling of $rpm$ is aligned with that of $v$ but every $n$ samples of $v$
- Logical time is totally ordered.
To Sample or not to Sample

Uniform sampling is wasteful here

Smallest Time Resolution
Outline of the talk

1. Motivation
2. Introduction
3. Concurrency and Multi-Threading
4. Distribution over Asynchronous Network
5. Concluding Remarks
Concurrency

- While the car is sampling speed for cruise control
  - It is also sampling temperature for climate control
  - It is also sampling user input to C/D player for audio control
  - It is also sampling GPS signals for navigation
  - It is sampling many other things
    - not all require the same sampling rate

- Further, in some cases, whether to sample depends on the values of already sampled ones.
  - e.g. Only if the sampled temperature too high, sample the coolant level
Multi-Attention Scenario

\[ L : S = 0 ; \]
\[ \text{Timer} = T ; \]

\[ \text{While} (\text{Timer} != 0) \{ \]
\[ \text{Sample} \nu \]
\[ S = S + (v_r - \nu) * k_i \]
\[ \text{Timer} = \text{Timer} - \tau \]
\[ \text{wait for} \ \tau \]
\[ \} \]
\[ \text{Sample} \nu \]
\[ U = k * (v_r - \nu) + S * k_i \]
\[ \text{Go to} \ L ; \]

\[ L : S = 0 ; \]
\[ \text{Timer} = T ; \]

\[ \text{While} (\text{Timer} != 0) \{ \]
\[ \text{Sample} t \]
\[ S = S + (t_s - t) * c_1 \]
\[ \text{Timer} = \text{Timer} - \tau \]
\[ \text{wait for} \ \tau \]
\[ \} \]
\[ \text{Sample} t \]
\[ \theta = c * (t_s - t) + S \]
\[ \text{Go to} \ L ; \]
Consider a Simplified version of this

\[ y = y^\text{init}0 + x \]

\[ u = u^\text{init}0 + v \]

where, \( x=1,3,4,5,7,9,10,-1,6,... \)

and \( v=0,1,3,4,5,6,... \)

There is

“quiescent determinism”
If we were to sample under global clock

- Read(x,v)?
- Read(x); Read(v); ?
- Read(v); Read(x); ?

None of them will be able to preserve all the possible flows shown.

Two distinct threads paced distinctly without any relationship between their paces – logical time is partially ordered.
What could have I done in Esterel/Lustre?

- Create Buffers?
  - What size?
  - Whatever size you choose, there are behaviors that get pruned out.
  - If you have any additional information between the paces of $x$ and $v$, then buffering may preserve all the behaviors
  - $\hat{x} = 3\hat{v} + 2$ (affine clocks)
When the threads interact!

- The previous example has two threads who never interact
- Two Esterel/Lustre processes could be written and run under two different clocks and avoid Polychrony
- But more often than not, these kinds of threads will interact
- A contrived example:
  - The temperature control thread might decide to disengage the cruise control when the temperature is too low
How to Handle Interrupt

```
process Interruptible_CC(? real v; ? boolean interrupt; ! real u)
  {parameter v_r, n, k, k_i}
  (| e := v_r - v
    | last_count := (count $ init 0)
    | count := (last_count + 1) when (last_count < n) default 0;
    | sum := ((sum $ init 0) + k_i * e) when (last_count < n) default 0;
    | u := (k_i * e + sum) when (!interrupt when (count = n))
    | interrupt ^= (count = n)
    | count ^= v ^= sum
  )
where
  real sum, e;
  integer count, last_count;
```

- 2 inputs with unrelated paces
  - interrupts happen once in a while
  - sampling of velocity happens regularly
- One solution: Check Interrupt only when outputting throttle
  - interrupt sampling is done at predetermined events – bring back total order
Another Solution

```plaintext
process Interruptible_CC(? real v; ? boolean interrupt; ! real u)
   {parameter v_r , n , k , k_i}
   (| e := v_r - v
    | last_count := (count $ init 0)
    | count := (last_count + 1) when (last_count < n) default 0;
    | sum := ((sum $ init 0) + k_i*e) when (last_count < n) default 0;
    | interrupted := interrupt default (interrupted $ init false)
    | u := (k*e + sum) when (! interrupted when (count == n))
    | interrupt ^= v
    | count ^= v ^= sum
   )
where
   real sum, e;
   integer count, last_count;
   boolean interrupted;
```

- Check for interrupt every time you sample v, and it has a value true iff there is an interrupt – total order.
Temperature Control Process (PI controller)

```
process TempControl( ?real t;!real θ;!event interrupt )
{parameter ts,n,c,ci,T}
  
  e := ts - t
  last_count := (count $ init 0)
  interrupt := true when (t < T)
  count := (last_count + 1) when (last_count < n) default 0;
  sum := ((sum $ init 0) + ci*e) when (last_count < n) default 0;
  θ := (c*e + sum) when (count == n)
  count ^= t ^= sum
 |
where
  real sum, e;
  integer count;
```

- Generate an interrupt as soon as temperature goes below a threshold T.
Combined CC + TC

```plaintext
process CCTC(?real v, real t;! boolean interrupt, real u, real θ) {
   {parameter vr, ts, n, m, k, ki, c, ci, T}

   (| e1 := vr − v
   | last_count1 := (count1 $ init 0)
   | count1:=(last_count1+1) when (last_count1 < n) default 0
   | sum1:=((sum1 $ init 0)+ki*e1) when (last_count1 < n) default 0
   | u := (k*e1 + sum1) when (! interrupted when (count1 == n)
   | interrupted ^ = (count1 == n)
   | count1 ^ = v ^ = sum1

   | e2 := ts − t
   | interrupt := true when (t>T) default interrupt $ init false
   | interrupted := interrupt when (count2 == m)
   | last_count2 := (count2 $ init 0)
   | count2:=(last_count2+1) when (last_count2 < n) default 0;
   | sum2:=((sum2 $ init 0)+ci*e) when (last_count2 < n) default 0;
   | θ := (c*e2 + sum2) when (count2 == m)
   | count2 ^ = t ^ = sum2
   |
   where
   real sum1, e1, sum2, e2;
   integer count1, count2;
   boolean interrupted;
}
```
process Modular_CCTC(?real v, real t;!boolean interrupt, real u, real θ)
{parameter vr, ts, n, m, k, ki, c, ci, T}
   (| u := Interruptible_CC{vr, n, k, ki}(v, interrupt)
   | θ, interrupt := TempControl{ts, m, c, c, ci, T}(t)
|)
process TempControl(? real t;! real θ;! boolean interrupt) {
  {parameter tₛ,n,c,cᵢ, T}
  (| e := tₛ − t
    | last_count := (count $ init 0)
    | in_interrupt := true when (t>T) default in_interrupt $ init false
    | interrupt := in_interrupt when (count == n)
    | count:=(last_count + 1) when (last_count < n) default 0;
    | sum:=((sum $ init 0) + cᵢ*e) when (last_count < n) default 0;
    | θ := (c*e + sum) when (count == n)
    | count ^= t ^= sum
  )
  where
    real sum, e;
    integer count;
    boolean in_interrupt
process Interruptible_CC(?real v;?boolean interrupt;!real u)
{parameter v_r,n,k,k_i}
(e := v_r − v
|last_count := (count $ init 0)
|count:=(last_count + 1) when (last_count < n) default 0;
|sum:=(sum $ init 0) + k_i*e) when (last_count < n) default 0;
|u := (k*e + sum) when (count == n) when !interrupt
|interrupt ^= (count == n)
|count ^= v ^= sum
}
where
real sum, e;
integer count, last_count;
Clock Hierarchy (Logical Time Hierarchy)

\[ v = e_1 = \text{count}_1 = \text{sum}_1 = \text{count}_1 \]
\[ t = e_2 = \text{count}_2 = \text{sum}_2 = \text{count}_2 \]

\[ \text{count}_1 < n \]
\[ \text{count}_1 = n = \text{interrupted} \]
\[ \text{count}_2 = m = \text{interrupted} = \theta \]
\[ \text{count}_2 < m \]
\[ t > T \]
\[ t \leq T \]

Interrupted == false = \text{u}
This process can be synthesized into two threads TC and CC

- **TC** in every cycle, samples temperature
  - At the same cycle when it issues temperature correction it checks if temperature exceeds threshold
  - if so, it generates interrupt and wait until CC’s has read it
  - then goes back to computing its control, and then starts the same cycle again.

- **CC** in every cycle samples speed,
  - computes the control, but checks for interrupted status which is by default false during every cycle, except when TC had raised the interrupt, and waiting.
  - interrupted status only changes at the same cycle as throttle computation
  - The CC’s throttle computation is synchronized with TC’s temperature correction

- The thread synchronization mechanism must ensure that TC can check when CC sets its interrupted status to true (via...
Outline of the talk

1. Motivation
2. Introduction
3. Concurrency and Multi-Threading
4. Distribution over Asynchronous Network
5. Concluding Remarks
Flow Determinism

What does it mean to design GALS implementation?

- Design a Concurrent System in Polychronous Framework
- Prove Correctness with respect to High Level Flow Equations
- Split the System into Concurrent Components
- Deploy over distributed nodes with no global clock
- Prove flow equivalence
Flow Determinism

- What does it mean to design GALS implementation?
  - Design a Concurrent System in Polychronous Framework
  - Prove Correctness with respect to High Level Flow Equations
  - Split the System into Concurrent Components
  - Deploy over distributed nodes with no global clock
  - Prove flow equivalence

- Let $P_1$ and $P_2$ be two Polychronous processes such that $P_1 \mid P_2$ is weakly endochronous
  - This means $P_1 \mid P_2$ has deterministic multi-threaded implementation with flow determinism
Flow Determinism (2)

What is flow determinism?

- Usually Polychronous operators define relations between flows.
- If endochronous – such relations turn out to be functions (endochrony).
- If weakly endochronous – such relations turn out to be functions modulo partial order trace equivalence (Mazurkiewicz trace theory).
Let us denote by $P_1 \parallel P_2$ – asynchronous composition of $P_1$ and $P_2$

If we have proven $P_1 \leftarrow P_2$ flow deterministic – safe to implement

- Proving $P_1 \parallel P_2 \sim P_1 \leftarrow P_2$ will accomplish our objective
- $\sim$ – flow equivalence

If $P_1 \parallel P_2 \sim P_1 \leftarrow P_2$ – then we have to find conditions or wrappers that would make it so.
if \( P_1 \) and \( P_2 \) share signals \( x, y, .. \)
- if \( P_1 \parallel P_2 \) is weakly endochronous – they have the same deterministic notion of timing of \( x, y, .. \)
- Hence \( P_1 \parallel P_2 \sim P_1 \mid P_2 \)

If \( P_1 \) and \( P_2 \) is said to be isochronous if they have exact mutual timing awareness.
Making them isochronous

- Consider $P_1 = (| x := a \text{ default } b |)$
- $P_2 = (| y := a \text{ default } b |)$
- Since $(| x := a \text{ default } b | y := a \text{ default } b |) \sim (| x := a \text{ default } b | y := x |)$
  - $P_1 | P_2$ (extended) flow deterministic.
  - But $P_1 | P_2 \not\sim P_1 || P_2$
  - Because relative delays of $a$ and $b$ are not guaranteed.

- Therefore, in order to deploy these two processes in a GALS environment, we need wrappers.
Wrapper Synthesis

- Let us define $P'_1 = (| x := a \text{ default } b | a^\wedge = \text{ when } ca | b^\wedge = \text{ when } cb | ca^\wedge = cb | )$
- Let $P'_2 = (| a^\wedge = \text{ when } ca | b^\wedge = \text{ when } cb | ca^\wedge = cb | y := a \text{ default } b | )$
- $P'_1 | P'_2 \sim P'_1 || P'_2$
- Now $P'_1$ is a wrapped version of $P_1$, and $P'_2$ is a wrapped version of $P_2$
- $P'_1$ and $P'_2$ has two extra inputs $ca$ and $cb$ which encode presence and absence of $a$, $b$, and thus both processes have mutual awareness of presence/absence of $a$ and $b$.
- If the network can guarantee synchronized signals are synchronously visible at both nodes ($ca$ and $cb$) – that is sufficient for this to work.
If the network can guarantee consistent delivery of a view of external signal synchronizations – e.g., `present()` system call

Let us define

\[ PP_1 = (| P'_1 | ca := present(a) | cb := present(b) |) \setminus \{ca, cb\} \]

\[ PP_2 = (| P'_2 | ca := present(a) | cb := present(b) |) \setminus \{ca, cb\} \]

\[ PP_1 \parallel PP_2 \]

Now \( PP_1 \) is a wrapped version of \( P_1 \), and \( PP_2 \) is a wrapped version of \( P_2 \).

\( PP_1 \) and \( PP_2 \) do not even need any change to their interface as the distributed O/S delivers a consistent information to both.

The Question is how does the O/S implement a deterministic system call such as `present()`
If `present()` system call is not deterministically implemented, one can make one of the processes a master process as follows:

Let us define

\[ PP_1 = (\| P'_1 \| ca := present(a) \| cb := present(b) \|) \]

\[ (PP_1 \| P'_2) \setminus \{ca, cb\} \sim P_1 \parallel P_2 \]
In these solutions the logical timing is not changed, thus the logical synchronizations are preserved. This is not required to preserve flow equivalence.

Consider the following example:

\[
ADD_1(\langle a, b; !s_1 \rangle) = s_1 := a + b \quad \text{and} \\
ADD_2(\langle a, b; !s_2 \rangle) = s_2 := a + b
\]

In \( ADD_1 \parallel ADD_2 \) we have \( s_1 \) and \( s_2 \) as synchronous flows – as so are \( a \) and \( b \).

Now let us create synchronous/asynchronous interfaces for these processes which can be wrapped on the synchronous \( ADD_i \) to be used in GALS.
Asynchronous Interface

```plaintext
process ASYNIF(? real a, b; ! real aa, ab)
  (| ma := a cell ^b
    | mb := b cell ^a
    | do_add = a^*b default (a ^+ b) when (number-arrived = 1)
    | number-arriving = (0 when do_add) default ((number-arrived + 1) ←
      when (a ^+ b))
    | number-arrived = number-arriving $ init 0
    | number_arriving ^= a ^+ b
    | aa := ma when do_add
    | bb := mb when do_add |)

where
  real ma, mb;
  integer number-arriving, number-arrived;
  event do_add;
end;
process ASYNADD1 (? real a, b; ! real s1)
  (| aa, bb := ASYNINF(a, b)
    | s1 := aa + bb
    | ) where
  real aa, ab;
end;
```
Asynchronous Interface

- In $\text{ASYNDD}_1 \mid \text{ASYNDD}_2$, $s_1$ and $s_2$ still are synchronous flows, but $a$ and $b$ are asynchronous.
- If there are no overtaking of $a$ or $b$ (there is never more than one occurrence of each flow in advance)
  
  $\text{ASYNDD}_1 \mid \text{ASYNDD}_2 \sim \text{ADD}_1 \mid \text{ADD}_2$
- Thus provided that there is no overtaking of $a$ or $b$ in the network, $\text{ASYNDD}_1 \parallel \text{ASYNDD}_2 \sim \text{ADD}_1 \mid \text{ADD}_2$
- Synchronization is not preserved, thus we do not have process equality.
Outline of the talk

1. Motivation
2. Introduction
3. Concurrency and Multi-Threading
4. Distribution over Asynchronous Network
5. Concluding Remarks
Final Remarks

- We talked about the basics of Polychrony and Calculus of Partially ordered Logical Instants
- How to use the Calculus to refine spec to implementation
- We did not talk about our most recent work.
Further Reading


Further Reading (2)


Further Reading (3)
