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

A Good Read

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

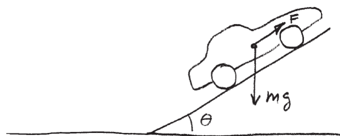


Figure: Schematic of a car on sloping road

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

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

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

PI Controller

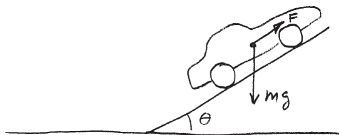


Figure: Schematic of a car on sloping road

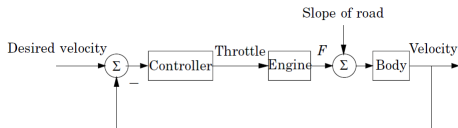


Figure: Block diagram of a car with cruise control

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

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

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

$$s^2 + (0.02 + k)s + k_i = 0$$

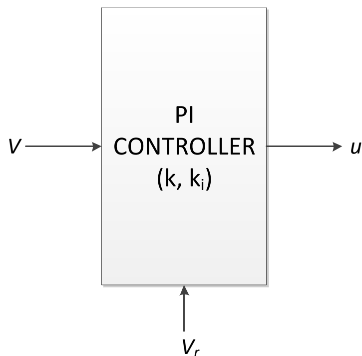
$$k = 2\zeta\omega_0 - 0.02$$

$$k_i = \omega_0^2$$

ζ is damping parameter

ω_0 is undamped natural frequency

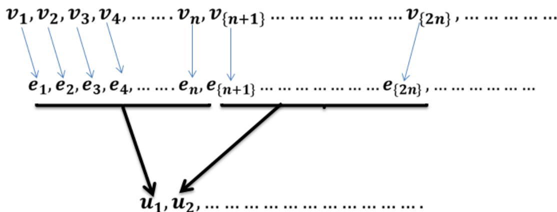
A PI Controller for Cruise Control



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

```
L: S = 0;
Timer = T;
while(Timer != 0){
     Sample v;
     S = S + (v_r - v)*k_i;
          Timer = Timer - τ
     wait for τ
}
Sample v;
     u = k * (v_r - v) + S;
GOTO L;
```

Signals as Flows



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,...

v=sampled velocity, e=instantaneous error, u=computed throttle input

How to Compute the Thrust u

```
process CruiseControl(?real v; !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:=  k_i*e when (count = 0) default  ((sum $ init 0) + k_i*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

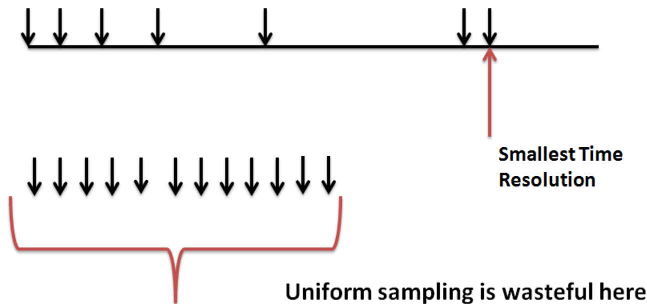
Handling Multiple Inputs

```
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*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;
```

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



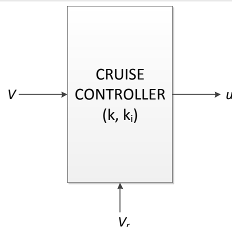
Outline of the talk

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

Concurrency

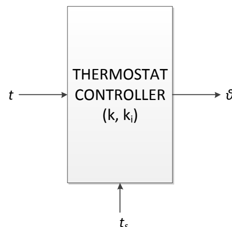
- While the car is sampling speed for cruise control
  - It is also sampling temperature for climate control
  - It is also sampling user input to C/D player for audio control
  - It is also sampling GPS signals for navigation
  - It is sampling many other things
    - not all require the same sampling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

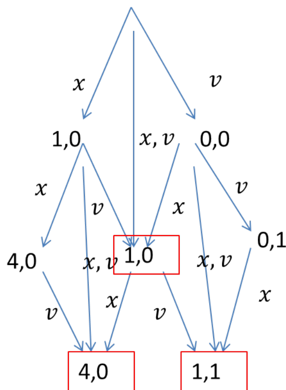
While (Timer != 0) {
Sample v
$S = S + (v_r - v) * k_i$
$Timer = Timer - \tau$
wait for τ
}
Sample v
$U = k * (v_r - v) + S * k_i$
Go to L;



L : S = 0 ;
Timer = T ;

While (Timer != 0) {
Sample t
$S = S + (t_s - t) * c_1$
$Timer = Timer - \tau$
wait for τ
}
Sample t
$\theta = c * (t_s - t) + S$
Go to L;

Consider a Simplified version of this



$$y = y\$init0 + x$$

$$|$$

$$u = u\$init0 + v$$

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*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

```
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 t_s,n,c,c_i, T}
(|e := t_s − 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) + c_i*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

```
process CCTC(?real v,real t;!boolean interrupt,real u,real θ)
    {parameter v_r,t_s n,m,k,k_i,c,c_i,T}
(|e1 :=  v_r − v
 |last_count1 := (count1 $ init 0)
 |count1:=(last_count1+1) when (last_count1 < n) default 0
 |sum1:=((sum1 $ init 0)+k_i*e1) when (last_count1 < n) default 0
 |u := (k*e1 + sum1) when (! interrupted when (count1 == n)
 |interrupted ^= (count1 == n)
 |count1 ^= v ^= sum1

 |e2 :=  t_s − 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)+c_i*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;
```

Modular Hierarchic CC+TC

```
process Modular_CCTC(?real v,real t;!boolean interrupt,real u,real θ)
     {parameter v_r,t_sn,m,k,k_i,c,c_i,T}
 (| u := Interruptible_CC{v_r, n, k,k_i}(v,interrupt)
  | θ, interrupt:= TempControl{t_s, m, c, c,c_i,T}(t)
  |)
```

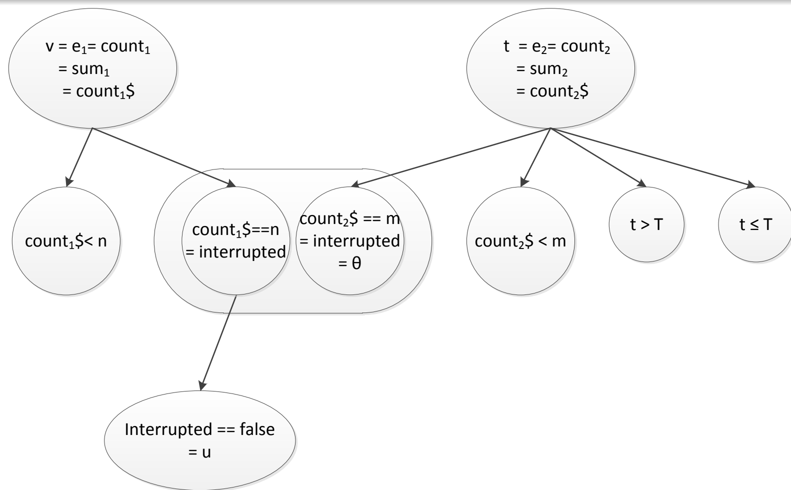
Modular Hierarchic CC+TC (2)

```
process TempControl(?real t;!real θ;!boolean interrupt)
    {parameter t_s,n,c,c_i, T}
 (|e := t_s - 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) + c_i*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
```

Modular Hierarchic CC+TC (3)

```
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)



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)

Mutual Timing Awareness

- Let us denote by $P_1 \| P_2$ – asynchronous composition of P_1 and P_2
- If we have proven $P_1 \mid P_2$ flow deterministic – safe to implement
  - Proving $P_1 \| P_2 \sim P_1 \mid P_2$ will accomplish our objective
  - $\sim$ – flow equivalence
- If $P_1 \| P_2 \nsim P_1 \mid P_2$ – then we have to find conditions or wrappers that would make it so.

Mutual Timing Awareness (2)

- if P_1 and P_2 share signals $x, y, ..$
  - if $P_1 \mid P_2$ is weakly endochronous – they have the same deterministic notion of timing of $x, y, ..$
  - Hence $P_1 \| P_2 \sim P_1 \mid P_2$
- If P_1 and P_2 is said to be isochronous if they have exact mutual timing awareness.

Making them isochronous

- Consider $P_1 = (|\ x \ := a \ default \ b\ |)$
- $P_2 = (|\ y \ := a \ default \ b\ |)$
- Since $(|\ x \ := a \ default \ b \mid y \ := a \ default \ b\ |) \sim (|\ x \ := a \ default \ b \mid y \ := x\ |)$
  - $P_1 \mid P_2$ (extended) flow deterministic.
  - But $P_1 \mid P_2 \nsim P_1 \| P_2$
  - 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 $P_1' = (|\ x\ := a\ default\ b \mid a\hat{\ }= when\, ca \mid b\hat{\ }= when\, cb \mid ca\hat{\ }= cb\ |)$
- Let $P_2' = (|\ a\hat{\ }= when\, ca \mid b\hat{\ }= when\, cb \mid ca\hat{\ }= cb \mid y\ := a\ default\ b\ |)$
  - $P_1' \mid P_2' \sim P_1' || P_2'$
  - 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.

Wrapper Synthesis (2)

- If the network can guarantee consistent delivery of a view of external signal synchronizations – e.g.,`present()` system call
- Let us define
  $PP_1 = (| P'_1 \mid ca := present(a) \mid cb := present(b) |) \setminus \{ca, cb\}$
- $PP_2 = (| P'_2 \mid ca := present(a) \mid cb := present(b) |) \setminus \{ca, cb\}$
  - $PP_1 \mid PP_2 \sim PP_1 \| PP_2$
  - 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()`

Wrapper Synthesis (3)

- If `present()` system call is not deterministically implemented, one can make one of the processes a master process as follows
- Let us define
  $PP_1 = (| P'_1 \mid ca := present(a) \mid cb := present(b) |)$
- $(PP_1 \mid P'_2) \setminus \{ca, cb\} \sim P_1 \| P_2$

Wrapper Synthesis (4)

- 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:
  $ADD_1(?a, b; !s_1) = s_1 := a + b$ and
  $ADD_2(?a, b; !s_2) = s_2 := a + b$
- In $ADD_1 \mid ADD_2$ 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

```
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 $ASYNDD_1 \mid ASYNDD_2$, 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) $ASYNDD1 \mid ASYNDD2 \sim ADD_1 \mid ADD_2$
- Thus provided that there is no overtaking of a or b in the network, $ASYNDD_1 \| ASYNDD_2 \sim ADD_1 \mid ADD_2$
- 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

1. "Embedding polychrony into synchrony" J. Brandt, M. Gemnde, K. Schneider, S. Shukla, and J.-P. Talpin. In Transactions on Software Engineering. IEEE, 2012.
2. "Representation of synchronous, asynchronous, and polychronous components by clocked guarded Actions" J. Brandt, M. Gemnde, K. Schneider, S. Shukla, and J.-P. Talpin. In Design Automation for Embedded Systems, Special Issue on Languages, Models and Model Based Design for Embedded Systems. Springer, 2012.
3. "Constructive polychronous systems". J.-P. Talpin, J. Brandt, M. Gemnde, K. Schneider, and S. Shukla. Logical Foundations in Computer Science (LFCS'12). Springer, January 2013

Further Reading (2)

1. Bijoy A. Jose, Jason Pribble and Sandeep K. Shukla, "Faster software synthesis using Actor Elimination Techniques for Polychronous formalism, in Proceedings of Applications of Concurrency in Synchronous (ACSD), Portugal, June 2010.
2. Bijoy A. Jose and Sandeep K. Shukla, MRICDF : A polychronous Model for Embedded Software Synthesis. Book Chapter in: "Synthesis of embedded software: frameworks and methodologies for correctness by construction software design", ISBN 978-1-4419-6399-4, Springer, 2010.
3. Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework M Nanjundappa, M Kracht, J Ouy, SK Shukla - System Level Synthesis Conference (ESLsyn), 2012
4. Bijoy A. Jose, Abdoulaye Gamati, Julien Ouy, Sandeep K. Shukla: SMT based false causal loop detection during code synthesis from Polychronous specifications. MEMOCODE 2011: 109-118

Further Reading (3)

1. Bijoy A. Jose, Sandeep K. Shukla: An alternative polychronous model and synthesis methodology for model-driven embedded software. ASP-DAC 2010: 13-18
2. Bijoy A. Jose, Jason Pribble, Sandeep K. Shukla: Faster Software Synthesis Using Actor Elimination Techniques for Polychronous Formalism. ACSD 2010: 147-156
3. M Nanjundappa, M Kracht, J Ouy, SK Shukla:A New Multi-threaded Code Synthesis Methodology and Tool for Correct-by-Construction Synthesis from Polychronous Specifications,ACSD 2013:21-30
4. SK Shukla, JR Ouy, M Nanjundappa, P Kumar, M Anderson, G Selvam, M Kracht: Techniques and Tools for Trustworthy Composition of Pre-Designed Embedded Software Components, AFRL Technical Report, 2012
5. Julien Ouy, Matthew Kracht, and Sandeep K. Shukla: Abstraction of Polychronous Dataflow Specifications into Mode-Automata, SAMOS XIII, 2013.

Any Questions??

Thank You!!