The Sparse Synchronous Model

Stephen A. Edwards

Synchon 2020, November 26, 2020

See also Edwards and Hui, FDL 2020
Time modeled arithmetically

main(led : Ref (Sched Int)) =
  loop
    50 ms later
    led < -1
    wait led
    50 ms later
    led < -0
    wait led
  led 0
Time modeled arithmetically
Quantized; quantum
not user-visible
Time modeled arithmetically
Quantized; quantum not user-visible

Infinitely fast processor model:
Program execution a series of zero-time instants
(hence “synchronous”)

0ms  50ms  100ms  150ms
Time modeled arithmetically
Quantized; quantum
not user-visible

Infinitely fast processor model:
Program execution a series of
zero-time instants
(hence “synchronous”)

Nothing happens in
most instants (hence “sparse”)

main(led : Ref (Sched Int)) =
loop
50 ms
later
led <
−1
wait
led
50 ms
later
led <
−0
wait
led

0ms  50ms  100ms  150ms
main(led : Ref (Sched Int)) =

loop

  50 ms later led ← 1
  wait led

  50 ms later led ← 0
  wait led

led is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
main(led : Ref (Sched Int)) =
loop
  50 ms later led <- 1
  wait led
  50 ms later led <- 0
  wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
main(led : Ref (Sched Int)) =

loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

 led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
main(led : Ref (Sched Int)) =
  loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

*led* is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable

---

<table>
<thead>
<tr>
<th>0ms</th>
<th>50ms</th>
<th>100ms</th>
<th>150ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>❤️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

led 0
main(led : Ref (Sched Int)) =

loop
50 ms later led ← 1
wait led
50 ms later led ← 0
wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable

0ms 50ms 100ms 150ms

led ← 1
led 0
main(led : Ref (Sched Int)) =

loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable

led 0
main(\text{led} : \text{Ref (Sched Int)}) = 
\begin{align*}
&\text{loop} \\
&\text{50 ms later } \text{led} \leftarrow 1 \\
&\text{wait } \text{led} \\
&\text{50 ms later } \text{led} \leftarrow 0 \\
&\text{wait } \text{led}
\end{align*}

\textit{led} is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
main(\text{led} : \text{Ref (Sched Int)}) =
\begin{align*}
\text{loop} \\
50 \text{ ms later} & \text{ led} \leftarrow -1 \\
\text{wait} & \text{ led} \\
50 \text{ ms later} & \text{ led} \leftarrow 0 \\
\text{wait} & \text{ led}
\end{align*}

\text{led is a pass-by-reference integer that can be scheduled} \\
\text{Infinite loop} \\
\text{Schedule a future update} \\
\text{Wait for a write on a variable} 

\text{led} 

0ms \quad \downarrow \quad 50\text{ms} \quad 100\text{ms} \quad 150\text{ms}
main(led : Ref (Sched Int)) =

loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

*led* is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable

0ms  led ← 1  50ms  100ms  150ms

led ———
main(\text{led} : \text{Ref (Sched Int)}) =
\text{loop}
\quad 50 \text{ ms later } \text{led} \leftarrow -1
\quad \text{wait led}
\quad 50 \text{ ms later } \text{led} \leftarrow 0
\quad \text{wait led}

\text{led} \leftarrow 1

\text{led is a pass-by-reference integer that can be scheduled}
\text{Infinite loop}
\text{Schedule a future update}
\text{Wait for a write on a variable}
main(led : Ref (Sched Int)) =
  loop
    50 ms later led ← 1
    wait led
    50 ms later led ← 0
    wait led

*led* is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
main(led : Ref (Sched Int)) =

loop

50 ms later led ← 1
wait led

50 ms later led ← 0
wait led

*led* is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
main(led : Ref (Sched Int)) =
  loop
    50 ms later led ← 1
    wait led
    50 ms later led ← 0
    wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
main(led : Ref (Sched Int)) =

loop

50 ms later led ← 1

wait led

50 ms later led ← 0

wait led

led is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
main(led : Ref (Sched Int)) =

  loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

*led* is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
main(\text{led} : \text{Ref (Sched Int)}) = 
\text{loop} 
\text{50 ms later } \text{led} \leftarrow -1 
\text{wait } \text{led} 
\text{50 ms later } \text{led} \leftarrow 0 
\text{wait } \text{led} 

\text{led is a pass-by-reference integer that can be scheduled} 
\text{Infinite loop} 
\text{Schedule a future update} 
\text{Wait for a write on a variable}
main(led : Ref (Sched Int)) =

loop
50 ms later led ← 1
wait led
50 ms later led ← 0
wait led

*led* is a pass-by-reference integer that can be scheduled

Infinite loop
Schedule a future update
Wait for a write on a variable
main(led : Ref (Sched Int)) =
loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

_ led is a pass-by-reference integer that can be scheduled_

_Infinite loop_

_Schedule a future update_

_Wait for a write on a variable_
main(led : Ref (Sched Int)) =
loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
main(led : Ref (Sched Int)) =

loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
main(led : Ref (Sched Int)) =
  loop
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led

*led* is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
main(led : Ref (Sched Int)) =
loop
50 ms later led ← 1
wait led
50 ms later led ← 0
wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
main(led : Ref (Sched Int)) =

\[
\text{loop} \\
\text{50 ms later } \text{led} \leftarrow 1 \\
\text{wait led} \\
\text{50 ms later } \text{led} \leftarrow 0 \\
\text{wait led}
\]

\(\text{led}\) is a pass-by-reference integer that can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
main(led : Ref (Sched Int)) =

loop

50 ms later led ← 1
wait led
50 ms later led ← 0
wait led

led is a pass-by-reference integer that can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
Missing Deadlines Doesn’t Affect Period

```plaintext
main(led : Ref (Sched Int)) =

loop

50 ms later led <- 1
wait led
50 ms later led <- 0
wait led
```
main(led : Ref (Sched Int)) =

loop
  fib 19 r
  50 ms later led ← 1
  wait led
  50 ms later led ← 0
  wait led
main(led : Ref (Sched Int)) =
    loop
        fib 23 r
        50 ms later led ← 1
        wait led
        50 ms later led ← 0
        wait led
Recursive subroutines

toggle(led : Ref (Sched Int)) =
   led ← 1 − led
Pure events like “void” or “unit”

```plaintext
toggle(led : Ref (Sched Int)) =
    led ← 1 − led

slow(led : Ref (Sched Int)) =
    let e1 = Occur : Sched Event

fast(led : Ref (Sched Int)) =
    let e2 = Occur : Sched Event

main(led : Ref (Sched Int)) =
    par slow led fast led
```
Function call

toggle(led : Ref (Sched Int)) =
    led ← 1 − led

slow(led : Ref (Sched Int)) =
    let e1 = Occur : Sched Event
    loop
        toggle led
    wait e1

fast(led : Ref (Sched Int)) =
    let e2 = Occur : Sched Event
    loop
        toggle led
    wait e2

main(led : Ref (Sched Int)) =
    par
        slow led
        fast led
"Occur": only value of a pure event

toggle(led : Ref (Sched Int)) =
   led ← 1 − led

slow(led : Ref (Sched Int)) =
   let e1 = Occur : Sched Event
   loop
      toggle led
      30 ms later e1 ← Occur
   wait e1
Concurrent function calls

toggle(led : Ref (Sched Int)) =
    led ← 1 – led

slow(led : Ref (Sched Int)) =
    let e1 = Occur : Sched Event
    loop
    toggle led
    30 ms later e1 ← Occur
    wait e1

fast(led : Ref (Sched Int)) =
    let e2 = Occur : Sched Event
    loop
    toggle led
    20 ms later e2 ← Occur
    wait e2

main(led : Ref (Sched Int)) =
    par slow led
    fast led
Concurrent Routines Execute in Syntactic Order for Determinism

```plaintext
main()
let a = 1 : Int
par foo a
   bar a
```

Concurrent Routines Execute in Syntactic Order for Determinism

```
foo(a : Ref Int) =
a ← a + 2
bar(a : Ref Int) =
a ← a * 4
```

```
main()
let a = 1 : Int
par foo a
par bar a
```
Concurrent Routines Execute in Syntactic Order for Determinism

foo(a : Ref Int) =
  a ← a + 2

bar(a : Ref Int) =
  a ← a * 4

main()
  let a = 1 : Int
  par foo a
     bar a

// foo runs first: a = 12 = (1 + 2) * 4
Concurrent Routines Execute in Syntactic Order for Determinism

\[
\text{foo}(a : \text{Ref Int}) = \begin{cases} 
    a & \leftarrow a + 2 \\
\end{cases}
\]

\[
\text{bar}(a : \text{Ref Int}) = \begin{cases} 
    a & \leftarrow a \times 4 \\
\end{cases}
\]

\[
\text{main}() \\
\hspace{1cm} \text{let } a = 1 : \text{Int} \\
\hspace{1cm} \text{par } \text{foo } a \\
\hspace{1cm} \text{bar } a \\
\hspace{1cm} \text{par } \text{bar } a \\
\hspace{1cm} \text{foo } a \\
\hspace{1cm} \texttt{// foo runs first: } a = 12 = (1 + 2) \times 4
\]
Concurrent Routines Execute in Syntactic Order for Determinism

foo(a : Ref Int) =
  a ← a + 2

bar(a : Ref Int) =
  a ← a * 4

main()
let a = 1 : Int
par foo a
par bar a

// foo runs first: a = 12 = (1 + 2) * 4
// bar runs first: a = 50 = (12 * 4) + 2
SSM vs. Esterel

[Berry and Gonthier, SCP 1992]
### SSM vs. Esterel

[Berry and Gonthier, SCP 1992]

<table>
<thead>
<tr>
<th></th>
<th>SSM</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
# SSM vs. Esterel

[Berry and Gonthier, SCP 1992]

<table>
<thead>
<tr>
<th>Feature</th>
<th>SSM</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>Sparse</td>
<td>Dense</td>
</tr>
</tbody>
</table>
### SSM vs. Esterel

[Berry and Gonthier, SCP 1992]

<table>
<thead>
<tr>
<th>Feature</th>
<th>SSM</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>Sparse</td>
<td>Dense</td>
</tr>
<tr>
<td>Within instants</td>
<td>Totally-ordered</td>
<td>Constructive</td>
</tr>
<tr>
<td></td>
<td>SSM</td>
<td>Esterel</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>Sparse</td>
<td>Dense</td>
</tr>
<tr>
<td>Within instants</td>
<td>Totally-ordered</td>
<td>Constructive</td>
</tr>
<tr>
<td>Compilation</td>
<td>Separate</td>
<td>Whole-program</td>
</tr>
</tbody>
</table>

[Berry and Gonthier, SCP 1992]
### SSM vs. Esterel

[Berry and Gonthier, SCP 1992]

<table>
<thead>
<tr>
<th></th>
<th>SSM</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>Sparse</td>
<td>Dense</td>
</tr>
<tr>
<td>Within instants</td>
<td>Totally-ordered</td>
<td>Constructive</td>
</tr>
<tr>
<td>Compilation</td>
<td>Separate</td>
<td>Whole-program</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dynamic Event Queues</td>
<td>Statically Scheduled</td>
</tr>
</tbody>
</table>
# SSM vs. Esterel

[Berry and Gonthier, SCP 1992]

<table>
<thead>
<tr>
<th></th>
<th>SSM</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>Sparse</td>
<td>Dense</td>
</tr>
<tr>
<td>Within instants</td>
<td>Totally-ordered</td>
<td>Constructive</td>
</tr>
<tr>
<td>Compilation</td>
<td>Separate</td>
<td>Whole-program</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dynamic Event Queues</td>
<td>Statically Scheduled</td>
</tr>
<tr>
<td>Topology</td>
<td>Dynamic, recursive</td>
<td>Static</td>
</tr>
</tbody>
</table>

---
SSM vs. Ptides

[Zhao, Liu, and Lee, RTAS 2007]
SSM vs. Ptides

[Zhao, Liu, and Lee, RTAS 2007]

<table>
<thead>
<tr>
<th></th>
<th>SSM</th>
<th>Ptides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between instants</td>
<td>Discrete-event</td>
<td>Discrete-Event</td>
</tr>
<tr>
<td>Dynamic, recursive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>Single-threaded</td>
<td>Distributed</td>
</tr>
<tr>
<td>Topology</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SSM vs. Ptides

[Zhao, Liu, and Lee, RTAS 2007]

<table>
<thead>
<tr>
<th></th>
<th>SSM</th>
<th>Ptides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between instants</td>
<td>Discrete-event</td>
<td>Discrete-Event</td>
</tr>
<tr>
<td>Within instants</td>
<td>Totally-ordered</td>
<td>Discrete-Event</td>
</tr>
</tbody>
</table>
### SSM vs. Ptides

[Zhao, Liu, and Lee, RTAS 2007]

<table>
<thead>
<tr>
<th></th>
<th>SSM</th>
<th>Ptides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between instants</td>
<td>Discrete-event</td>
<td>Discrete-Event</td>
</tr>
<tr>
<td>Within instants</td>
<td>Totally-ordered</td>
<td>Discrete-Event</td>
</tr>
<tr>
<td>Topology</td>
<td>Dynamic, recursive</td>
<td>Static</td>
</tr>
</tbody>
</table>

[See also Lee, Lohstroh et al.]
**SSM vs. Ptides**

[Zhao, Liu, and Lee, RTAS 2007]

<table>
<thead>
<tr>
<th></th>
<th>SSM</th>
<th>Ptides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between instants</td>
<td>Discrete-event</td>
<td>Discrete-Event</td>
</tr>
<tr>
<td>Within instants</td>
<td>Totally-ordered</td>
<td>Discrete-Event</td>
</tr>
<tr>
<td>Topology</td>
<td>Dynamic, recursive</td>
<td>Static</td>
</tr>
<tr>
<td>Implementation</td>
<td>Single-threaded</td>
<td>Distributed</td>
</tr>
</tbody>
</table>

[Zou Ph.D 2011] See also Lee, Lohstroh et al. *Linga Franca*
Compared to Dynamic Ticks

Haxlenden, Bourke, Girault, FDL 2017

Dynamic ticks uses repeated “min” to decide “how long to wait”

SSM uses an event (priority) queue to decide this

Dynamic Ticks uses the richer, but harder-to-compile Esterel semantics
Compared to Boussinot’s Work

Boussinot’s schedule-based-on-syntactic-order inspired the SSM policy

Boussinot: Round-robin cooperative scheduler; SSM: totally-ordered-within-an-instant

Less concern for real-time behavior; more an operational replacement for Esterel-style semantics