The Sparse Synchronous Model

Stephen A. Edwards

Chalmers, February 2, 2021

See also Edwards and Hui, FDL 2020
Time modeled arithmetically
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”)

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

0ms 50ms 100ms 150ms

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

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

\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(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

\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

\begin{itemize}
    \item \textcolor{red}{0ms}
    \item \textcolor{red}{\textit{led} ← 1}
    \item 50ms
    \item 100ms
    \item 150ms
\end{itemize}
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(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 _________
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>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0ms</td>
<td>led ← 1</td>
</tr>
<tr>
<td>50ms</td>
<td></td>
</tr>
<tr>
<td>100ms</td>
<td></td>
</tr>
<tr>
<td>150ms</td>
<td></td>
</tr>
</tbody>
</table>

led __________________
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
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)}) =
\begin{align*}
\text{loop} & \\
50 \text{ ms later } \text{led} & < 1 \\
\text{wait } \text{led} & \\
50 \text{ ms later } \text{led} & < 0 \\
\text{wait } \text{led} & 
\end{align*}

\textit{led} \ is \ a \ pass-by-reference \ integer \ that \ can \ be \ scheduled

\textit{Infinite loop}

\textit{Schedule a future update}

\textit{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

---

0ms

50ms

100ms

150ms

---

led

---
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)) =
  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
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

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)) =
   pipe slow led
   fast led
Pure events like “void” or “unit”

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

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

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

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

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

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

\[
\text{toggle}(\text{led} : \text{Ref} (\text{Sched Int})) =
\text{led} \leftarrow 1 - \text{led}
\]

\[
\text{slow}(\text{led} : \text{Ref} (\text{Sched Int})) =
\text{let } e1 = \text{Occur} : \text{Sched Event}
\text{let } e1 = \text{Occur} : \text{Sched Event}
\text{loop}
\text{toggle led}
30 \text{ ms later } e1 \leftarrow \text{Occur}
\text{wait } e1
\]

\[
\text{fast}(\text{led} : \text{Ref} (\text{Sched Int})) =
\text{let } e2 = \text{Occur} : \text{Sched Event}
\text{loop}
\text{toggle led}
20 \text{ ms later } e2 \leftarrow \text{Occur}
\text{wait } e2
\]

\[
\text{main}(\text{led} : \text{Ref} (\text{Sched Int})) =
\text{pipe}
\text{slow } \text{led}
\text{fast } \text{led}
\]
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)) =
  pipe slow led
  fast led
Concurrent Routines Execute in Syntactic Order for Determinism

```ocaml
main()
  let a = 1 : Int
  pipe foo a
  bar a
```

// foo runs first: a = 12 = (1 + 2) * 4
// bar runs first: a = 50 = (12 * 4) + 2
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
pipe foo a

pipe bar a
Concurrent Routines Execute in Syntactic Order for Determinism

```plaintext
foo(a : Ref Int) =
    a ← a + 2

bar(a : Ref Int) =
    a ← a * 4

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

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

**foo(a : Ref Int) =**
\[ a \leftarrow a + 2 \]

**bar(a : Ref Int) =**
\[ a \leftarrow a \times 4 \]

**main()**
\[
\begin{align*}
&\text{let } a = 1 : \text{Int} \\
&\text{pipe } \text{foo } a \\
&\text{bar } a \\
&\text{pipe } \text{bar } a \\
&\text{// foo runs first: } a = 12 = (1 + 2) \times 4 \\
&\text{pipe } \text{foo } a
\end{align*}
\]
Concurrent Routines Execute in Syntactic Order for Determinism

\[
\begin{align*}
\text{foo}(a : \text{Ref Int}) &= \quad \text{main() } \\
&= a \leftarrow a + 2 \\
\text{bar}(a : \text{Ref Int}) &= \\
&= a \leftarrow a \times 4
\end{align*}
\]

\[
\begin{align*}
\text{let } a = 1 : \text{Int} & \\
\text{pipe } \text{foo } a & \\
\text{pipe } \text{bar } a & \\
\text{pipe } \text{bar } a & \\
\text{foo } a & \\
\text{foo } a & \\
\end{align*}
\]

// foo runs first: \( a = 12 = (1 + 2) \times 4 \)

// bar runs first: \( a = 50 = (12 \times 4) + 2 \)
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>

[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></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>
</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>Compilation</td>
<td>Separate</td>
<td>Whole-program</td>
</tr>
<tr>
<td>Feature</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>
<tr>
<td>Runtime</td>
<td>Dynamic Event Queues</td>
<td>Statically Scheduled</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>
<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>SSM</th>
<th>Ptides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between instants</td>
<td>Discrete-event</td>
</tr>
<tr>
<td></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>
</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. Linga Franca]
SSM vs. Ptides

<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>

[Zhao, Liu, and Lee, RTAS 2007]

[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