Lecture 15: SCEst
Sequentially Constructive Esterel

Reinhard von Hanxleden (Kiel U)

Thanks for discussions with Michael Mendler, Gérard Berry, Joaquin Aguado, Insa Fuhrmann, Christian Motika, Steven Smyth, Alain Girault, Marc Pouzet, Karsten Rathlev, Partha Roop, Frank Steffahn …

zest noun \'zest\n: lively excitement : a feeling of enjoyment and enthusiasm
: small pieces of a model of computation that are used to flavor programming languages

[http://www.merriam-webster.com/dictionary/zest]

Tick
Reactive System
Environment

R1: inputs determine outputs
R2: pause separates reactions

input I Reactive System output O
Read Compute Write

pause

Tick
**On R1:**
Unique values throughout tick (Esterel) not needed

**On R2:**
Avoid `pause` statements that split reaction

**Sequential Constructiveness:**
Permit sequential evolution of values **within** reaction
⇒ Programmer freedom
⇒ Avoid timing issues within reaction

---

### SCEst – MoC

- Based on Sequentially Constructive MoC
- A **conservative** extension of Esterel
- Valid Esterel programs are valid SCEst programs, with same semantics
- Transformation rules for Esterel also hold for SCEst

_Aguado, Mendler, von Hanxleden, Fuhrmann_
*Grounding Synchronous Deterministic Concurrency in Sequential Programming*
*ESOP '14*

---

### SCEst – Language

- Esterel + SCL
- So far, consider Esterel v5 as base
- Might also adopt Esterel v7

_Smyth, Motika, Rathlev, von Hanxleden, Mendler_
*SCEst: Sequentially Constructive Esterel*
*ACM TECS '17*
**Sequentially Constructive Language/Graph**

In addition, SCL contains sequence ; and **goto**

von Hanxleden, Mendler, Aguado, et al.
Sequentially Constructive Concurrency –
A Conservative Extension of the Synchronous Model of Computation
ACM TECS ’14

**SCEst – Definition**

- Defined (here) by mapping to SCL
- Can be viewed as syntactic sugar on top of SCL
- Can view SCL as (SC)Est kernel statements
  
  ✓ Simple definition of semantics
  
  ✓ Simple, incremental, certifiable (?) compiler

**First Example**

---

**Variables**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>C</th>
<th>Esterel</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = y)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>(x := y)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>(if (x))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Pure Signals**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Esterel</th>
<th>SCEst</th>
<th>Esterel</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>present (x)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>emit (x)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>emit (x(v))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Signal Values**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Esterel</th>
<th>SCEst</th>
<th>Esterel</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>emit (x(v))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>emit (x(v))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>emit (x(v))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>emit (x(v))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Concurrency**

- thread \(t\), fork \(t_1\) par \(t_2\) join
- \(c\) Boolean expression

**Assignment**

- \(x = e\)

**Delay**

- pause

**SCL**

- entry
- \(s_1\)
- \(s_2\)
- exit

**SCG**

- entry
- fork
- join
- \(c\)
- true
- \(s = e\)
- exit
First Rules

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>fork</td>
</tr>
<tr>
<td>p</td>
<td></td>
</tr>
<tr>
<td>]</td>
<td>join</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>l: p;</td>
</tr>
<tr>
<td>p</td>
<td>goto l</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>abort</td>
</tr>
<tr>
<td>p;</td>
<td>p</td>
</tr>
<tr>
<td>halt</td>
<td>when s</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>if (c) goto l</td>
</tr>
<tr>
<td>while (c)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (c) { l: if (c) {</td>
<td></td>
</tr>
</tbody>
</table>
| p }     | p; goto l }

Esterel Rules Still Hold

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal s in</td>
<td></td>
</tr>
<tr>
<td>pnt: non-terminating statement(s)</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal s in</td>
<td></td>
</tr>
<tr>
<td>pnt: non-instantaneous statement(s)</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

Pure Signals

f: fresh flag
pnt: non-terminating statement(s)

Recall: SC MoC orders
s = false (init)
before concurrent
s = s | true (update)

Pure Signals, avoiding schizophrenia

To be applied if
1. downstream-synthesis requires acyclic SCG, and
2. signal scopes are possibly instantaneously re-entered

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>emit s</td>
<td></td>
</tr>
<tr>
<td>s = s</td>
<td>true</td>
</tr>
<tr>
<td>present s ... if (s) ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool f - false;</td>
<td></td>
</tr>
<tr>
<td>bool _f = false;</td>
<td></td>
</tr>
<tr>
<td>fork p; _f = true par</td>
<td></td>
</tr>
<tr>
<td>p; f - true par</td>
<td></td>
</tr>
<tr>
<td>if (!f) {</td>
<td></td>
</tr>
<tr>
<td>pause; goto l</td>
<td></td>
</tr>
<tr>
<td>join</td>
<td></td>
</tr>
<tr>
<td>emit s</td>
<td></td>
</tr>
<tr>
<td>s = s</td>
<td>true</td>
</tr>
<tr>
<td>present s ... if (s) ...</td>
<td></td>
</tr>
</tbody>
</table>
Schizophrenic Signal Example

To avoid cycle in dataflow SCG, also need "depth join".

Trap / Exit

<table>
<thead>
<tr>
<th>SCEst</th>
<th>SCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>trap t in</td>
<td>p end</td>
</tr>
<tr>
<td>p: statement(s) without trap</td>
<td>gotoj _l: goto _l, if goto in same thread as _l</td>
</tr>
</tbody>
</table>
| gotoj _exit, otherwise | _l: }

Note: the jump at pause can only be triggered by a concurrent exit; the corresponding fork/join then must be nested within trap scope; thus, if we have to jump at pause, we must jump to _exit, never to _l

Trap Example

To avoid cycle in dataflow SCG, also need "depth join".
Nested Trap Example

```
trap T1 in
  trap T2 in
    fork
      exit T1
    par
      exit T2
    join
end;
A |= true
B |= true
```

Deduction of Await Rule 1

```
await s
```

Deduction of Await Rule 2

```
pause;
trap t in
  loop
    present s
      then exit t
    else pause
  end present
end loop
end trap
```

Deduction of Await Rule 3

```
pause;
{bool _t = false;
  _l: if (s) {
    _t |= true;
    goto _l1
  }
  else {
    if (_t)
      goto _l1;
    else pause
  }
  goto _l;
}
```

To SCL

```
pause;
{bool _t = false;
  _l: if (s) {
    _t |= true;
    goto _l1
  }
  else {
    if (_t)
      goto _l1;
    pause
  }
  goto _l1;
}
```
Deduction of Await Rule 4

```plaintext
pause;
_l: if (s)
go to _l1
else
  pause;
go to _l1;
_l1:
```

opt

Resulting Await Rule

```plaintext
await S
```
ABRO

loop abort
    [ await A
      || await B
    ];
    emit O;
    halt
when R
end

ABRO

parallel loop abort
    fork
      await A
    par
      await B
    join;
    emit O;
    halt
    when R
end

ABRO

loop abort
    fork
      await A
    par
      await B
    join; 
    emit O;
    halt
    when R
end

loop abort
    fork
      await A
    par
      await B
    join;
    emit O;
    halt
    when R
end
loop
  abort
  fork
11:    pause;
      if (!A)
        goto 11
    par
12:    pause;
      if (!B)
        goto 12
    join;
    emit 0;
    halt
 when R
end

loop
  abort
  fork
11:    pause;
      if (!A)
        goto 11
    par
12:    pause;
      if (!B)
        goto 12
    join;
    emit 0;
    goto 13;
 when R
end

loop
  abort
  fork
11:    pause;
      if (!A)
        goto 11
    par
12:    pause;
      if (!B)
        goto 12
    join;
    emit 0;
when R
end
Wrap-Up

- SCEst conservatively extends Esterel
- SC MoC reduces likelihood of causality cycles
- Easy to adapt (hopefully) for C/Java programmers
- Defined by simple mapping to SCL
- Experience from SCCharts promising

Downstream Compilation
So far, two alternative compilation strategies from SCL/SCG to C/VHDL

<table>
<thead>
<tr>
<th></th>
<th>Dataflow</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accepts instantaneous loops</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Can synthesize hardware</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Can synthesize software</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Size scales well (linear in size of SCChart)</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Speed scales well (execute only active parts)</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Instruction-cache friendly (good locality)</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Pipeline friendly (little/no branching)</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>WCRT predictable (simple control flow)</td>
<td>+</td>
<td>+/-</td>
</tr>
<tr>
<td>Low execution time jitter (simple/fixed flow)</td>
<td>+</td>
<td>–</td>
</tr>
</tbody>
</table>