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\

: lively excitement : a feeling of enjoyment and enthusiasm

: small pieces of the skin of a lemon, orange, or lime that are used to flavor food

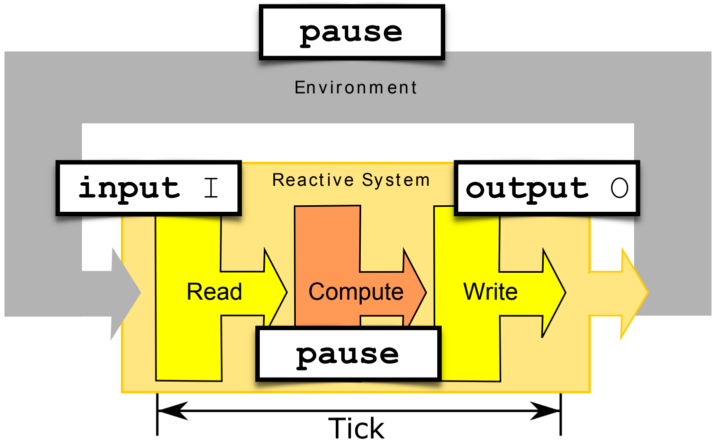
[http://www.merriam-webster.com/dictionary/zest]

scest *noun* \'zest\

: lively excitement : a feeling of enjoyment and enthusiasm

: small pieces of a model of computation that are used to flavor programming languages

R1: inputs <u>determine</u> outputs
R2: **`pause`** separates <u>reactions</u>



R1: inputs <u>determine</u> outputs
R2: **`pause`** separates <u>reactions</u>

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

R1: inputs <u>determine</u> outputs
R2: **`pause`** separates <u>reactions</u>

	Esterel	SCEst
`O = 1 || O = 2`	Rejected	Rejected
`present Done else … emit Done end`	Rejected	Accepted
`emit O(1); emit O(?O + 1)`	Rejected	Accepted
`emit O(1); pause; emit O(pre(?O)+1)`	Accepted	Accepted

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

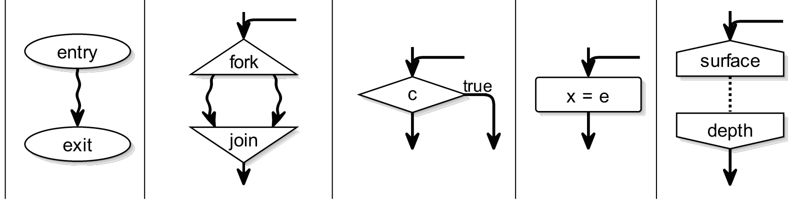
SCEst – Language

- 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 **C**onstructive **L**anguage/**G**raph

	Thread	Concurrency	Conditional	Assignment	Delay
SCL	t	fork t_1 par t_2 join	if (c) s_1 else s_2	$x = e$	pause
SCG					



In addtion, 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**

	Variables			Pure Signals		Signal Values	
	C	**Esterel**	**SCEst**	**Esterel**	**SCEst**	**Esterel**	**SCEst**

Syntax	`x = y`
	`if (x)`
		`x := y`
		`if x`
			`x = y`
			`if (x)`
				`emit x`
				`present x`
					`emit x`
					`unemit x`
					`present x`
					`if (x)`
						`emit x(v)`
						`?x`
							`emit x(v)`
							`?x`
							`set x(v)`
							`unemit x`

Type	arbitrary	arbitrary	arbitrary	present/ absent	present/ absent	arbitrary	arbitrary
Initialized each tick	no	no	no	yes (absent)	yes (absent)	no	no
Persistence across ticks	yes	yes	yes	no	no	yes	yes
Allow multiple values / tick	yes	yes	yes	no	yes	no	yes
Sequential scheduling constraints	none	none	none	first `emit` ➛ reads	none	emits ➛ reads	none
Concurrent scheduling constraints	none	read only	inits ➛ updates ➛ reads	first emit ➛ reads	unemits ➛ first emit ➛ reads	emits ➛ reads	unemits ➛ sets ➛ emits ➛ reads
I/O determinacy guaranteed	no	yes	yes	yes	yes	yes	yes

First Example

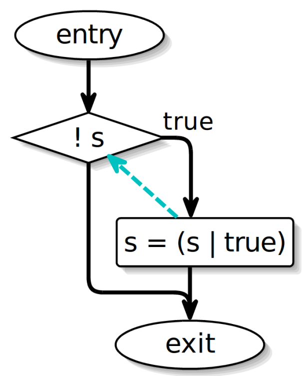
SCEst

```
present (not s) then
  emit s
end
```

SCL

```
if (!s) {
  s = s | true
}
```

SCG



First Rules

`p`, `q`: statement(s)
`s`: pure signal
`l`: fresh label
`c`: boolean exp.

<table>
<tr><th>SCEst</th><th>SCL</th></tr>
<tr><td>[
  p || q
]</td><td>fork
  p par q
join</td></tr>
<tr><td>loop
  p
end</td><td>l: p;
   goto l</td></tr>
<tr><td>do
  p
while (c)</td><td>l: p;
   if (c) goto l</td></tr>
<tr><td>while (c) {
  p }</td><td>l: if (c) {
   p; goto l }</td></tr>
</table>

Esterel Rules Still Hold

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

Pure Signals

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

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

Rule for output similar

<table>
<tr><th>SCEst</th><th>SCL</th></tr>
<tr><td>signal s in
  p
end</td><td>{ bool s;
  bool _f = false;
  fork
    p; _f = true
  par
l: s = false;
    if (!_f) {
      pause;
      goto l }
  join }</td></tr>
<tr><td>signal s in
  pnt
end</td><td>{ bool s;
  fork
    pnt
  par
l: s = false;
    pause; goto l
 join }</td></tr>
<tr><td>emit s</td><td>s = s | true</td></tr>
<tr><td>present s …</td><td>if (s) …</td></tr>
</table>

Pure Signals, avoiding schizophrenia

To be applied if

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

`f`: fresh flag
`pni`: non-instantaneous statement(s)

<table>
<tr><th>SCEst</th><th>SCL</th></tr>
<tr><td>signal s in
  pni
end</td><td>{
  bool f = false;
  // surface init
  bool s = false;
  fork
    p;
    f = true
  par
    do
      pause;
      // depth init
      s = false;
    while (!f)
  join
}</td></tr>
</table>

Schizophrenic Signal Example

```
loop
  signal S in
    present S then
      emit O
    end;
    pause;
    emit S
  end
end
```

signal →

```
loop
   bool f = false;
   bool S = false;
   fork
     if (S)
        O |= true;
     pause;
     S |= true;
     f = true;
   par
     do
        pause;
        S = false;
     while (!f);
   join
end
```

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

Trap Example

```
trap T in
  fork
    pause;
    A |= true;
    pause;
    exit T
  par
    l: pause;
    if (!B) goto l;
    C |= true
  join
end trap;
D |= true
```

trap →

```
  {
   bool T = false;
   fork
     if (T) goto l1;
     pause;
     A |= true;
     if (T) goto l1;
     pause;
     T |= true;
     goto l1;
l1:
   par
l:   if (T) goto l2;
     pause;
     if (!B) goto l;
     C |= true;
l2:
   join;
   if (T) goto l0
  };
l0:D |= true
```


Trap / Exit

SCEst	SCL
`trap t in` `p` `end`	`{ bool _t = false;` `p [ exit t ->` `{_t = true; gotoj _l}` `| pause ->` `if (_t) goto _exit; pause` `| join ->` `join; if (_t) gotoj _l];` `_l: }`

p: statement(s) without **trap**

gotoj _l: **goto** _l, if **goto** in same thread as _l
goto _exit, otherwise

_exit: label at end of thread

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

```
  {
   bool T = false;
   fork
     if (T) goto l1;
     pause;
     A |= true;
     if (T) goto l1;
     pause;
     T |= true;
     goto l1;
l1:
   par
l:   if (T) goto l2;
     pause;
     if (!B) goto l;
     C |= true;
l2:
   join;
   if (T) goto l0
  };
l0:D |= true
```

opt. →

```
  {
   bool T = false;
   fork
     pause;
     A |= true;
     pause;
     T |= true;
   par
l:   if (T) goto l2;
     pause;
     if (!B) goto l;
     C |= true;
l2:
   join;
  };
  D |= true
```

Nested Trap Example

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

trap

```
    {
      bool T1 = false;
      {
        bool T2 = false;
        fork
          T1 |= true;
          goto l1
l1:
        par
          T2 |= true;
          goto l2
l2:
        join;
        if (T1) goto l4;
        if (T2) goto l3;
      };
l3:   A |= true
    }
l4:B |= true
```


Deduction of Await Rule 1

```
await s
```

definition of await

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


Deduction of Await Rule 2

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

to SCL

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


Deduction of Await Rule 3

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

eliminate _t

```
pause;
_l: if (s)
  goto _l1
else
  pause;
goto _l;
_l1:
```

Deduction of Await Rule 4

```
pause;
_l: if (s)
  goto _l1
else
  pause;
goto _l;
_l1:
```

→ opt

```
_l: pause;
if (!s)
  goto _l
```


Resulting Await Rule

```
await S
```

→ await

```
_l: pause;
if (!s)
  goto _l
```

Esterel definitions of derived statements
+ SCEst-SCL translation rules for kernel statements
+ Reasoning at SCL-level
= Optimized rules for derived statements

No ad-hoc rules for derived statements!

Abort

SCEst	SCL

```
SCEst      | SCL
-----------+------------------------------------------------------
           | { bool _t = false;
abort      |   p [ pause -> pause; if (s)
  p        |                    {_t = true; gotoj _l}
when s     |     | join -> join; if (_t) gotoj _l];
           | _l: }
```

Further rules for weak and/or immediate abort, also WTO

Abort – Optimized

```
SCEst      | SCL
-----------+------------------------------------------------------
abort      | p [ pause -> pause; if (s) gotoj _l
  pni      |   | join -> join; if (s) gotoj _l];
when s     | _l:
```

`pni`: statements without instantaneously reachable join

ABRO

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


ABRO

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

parallel

```
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
      await A
    par
      await B
    join;
    emit O;
    halt
  when R
end
```

await

```
 loop
   abort
     fork
l1:    pause;
       if (!A)
         goto l1
     par
l2:    pause;
       if (!B)
         goto l2
     join;
     emit O;
     halt
   when R
 end
```

```
loop
   abort
      fork
l1:      pause;
         if (!A)
            goto l1
      par
l2:      pause;
         if (!B)
            goto l2
      join;
      emit O;
      halt
   when R
end
```

33

```
loop
   abort
      fork
l1:      pause;
         if (!A)
            goto l1
      par
l2:      pause;
         if (!B)
            goto l2
      join;
      emit O;
      halt
   when R
end
```

halt

```
loop
   abort
      fork
l1:      pause;
         if (!A)
            goto l1
      par
l2:      pause;
         if (!B)
            goto l2
      join;
      emit O;
l3:   pause;
      goto l3;
   when R
end
```

34

```
loop
   abort
      fork
l1:      pause;
         if (!A)
            goto l1
      par
l2:      pause;
         if (!B)
            goto l2
      join;
      emit O;
l3:   pause;
      goto l3;
   when R
end
```

35

```
loop
   abort
      fork
l1:      pause;
         if (!A)
            goto l1
      par
l2:      pause;
         if (!B)
            goto l2
      join;
      emit O;
l3:   pause;
      goto l3;
   when R
end
```

abort

```
   loop
     fork
l1:    pause;
       if (R) goto l4;
       if (!A) goto l1;
l4:
     par
l2:    pause;
       if (R) goto l5;
       if (!B) goto l2;
l5:
     join;
     if (R) goto l6;
     emit O;
l3:  pause;
     if (R) goto l6;
     goto l3;
l6:end
```

36

```
   loop
     fork
l1:    pause;
       if (R) goto l4;
       if (!A) goto l1;
l4:
     par
l2:    pause;
       if (R) goto l5;
       if (!B) goto l2;
l5:
     join;
     if (R) goto l6;
     emit O;
l3:  pause;
     if (R) goto l6;
     goto l3;
l6:end
```

37

```
   loop
     fork
l1:    pause;
       if (R) goto l4;
       if (!A) goto l1;
l4:
     par
l2:    pause;
       if (R) goto l5;
       if (!B) goto l2;
l5:
     join;
     if (R) goto l6;
     emit O;
l3:  pause;
     if (R) goto l6;
     goto l3;
l6:end
```

loop

```
l7:fork
l1:  pause;
     if (R) goto l4;
     if (!A) goto l1;
l4:
   par
l2:  pause;
     if (R) goto l5;
     if (!B) goto l2;
l5:
   join;
   if (R) goto l6;
   emit O;
l3:pause;
   if (R) goto l6;
   goto l3;
l6:goto l7
```

38

```
l7:fork
l1:  pause;
     if (R) goto l4;
     if (!A) goto l1;
l4:
   par
l2:  pause;
     if (R) goto l5;
     if (!B) goto l2;
l5:
   join;
   if (R) goto l6;
   emit O;
l3:pause;
   if (R) goto l6;
   goto l3;
l6:goto l7
```

39

```
l7:fork
l1:  pause;
     if (R) goto l4;
     if (!A) goto l1;
l4:
   par
l2:  pause;
     if (R) goto l5;
     if (!B) goto l2;
l5:
   join;
   if (R) goto l6;
   emit O;
l3:pause;
   if (R) goto l6;
   goto l3;
l6:goto l7;
```

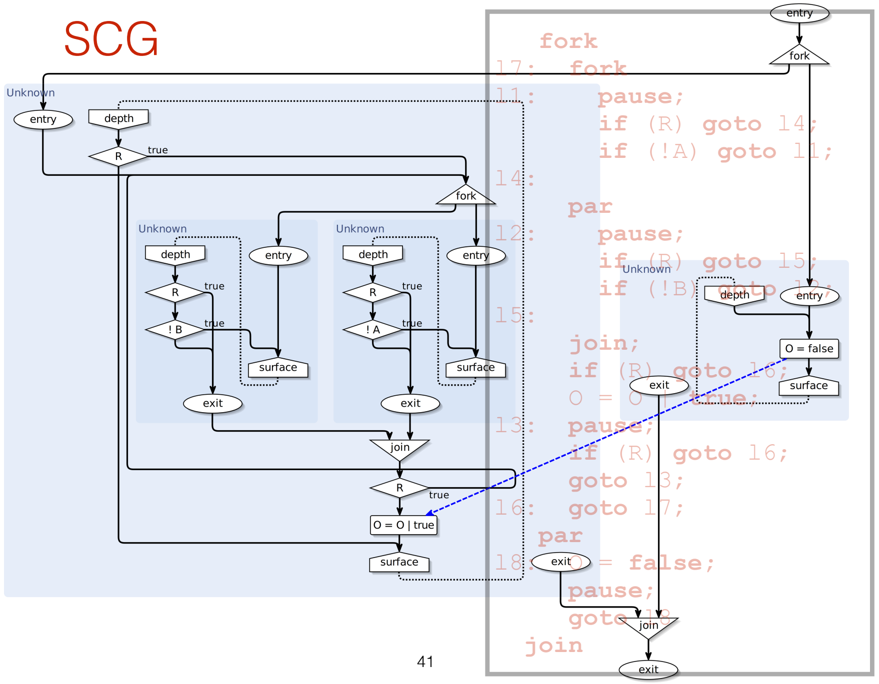
emit, output

```
  fork
l7:  fork
l1:    pause;
       if (R) goto l4;
       if (!A) goto l1;
l4:
     par
l2:    pause;
       if (R) goto l5;
       if (!B) goto l2;
l5:
     join;
     if (R) goto l6;
     O = O | true;
l3:  pause;
     if (R) goto l6;
     goto l3;
l6:  goto l7;
  par
l8:  O = false;
     pause;
     goto l8
  join
```

init → update

40

SCG



Downstream Compilation

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

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

von Hanxleden, Duderstadt, Motika, et al.
SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications
PLDI'14

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