SCEst

Sequentially Constructive Esterel

Reinhard von Hanxleden, Karsten Rathlev (Kiel U)

Thanks for discussions with Michael Mendler, Gérard Berry, Joaquin Aguado, Insa Fuhrmann, Christian Motika, Steven Smyth, Alain Girault, Marc Pouzet, Partha Roop ...

A work in progress report …

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]

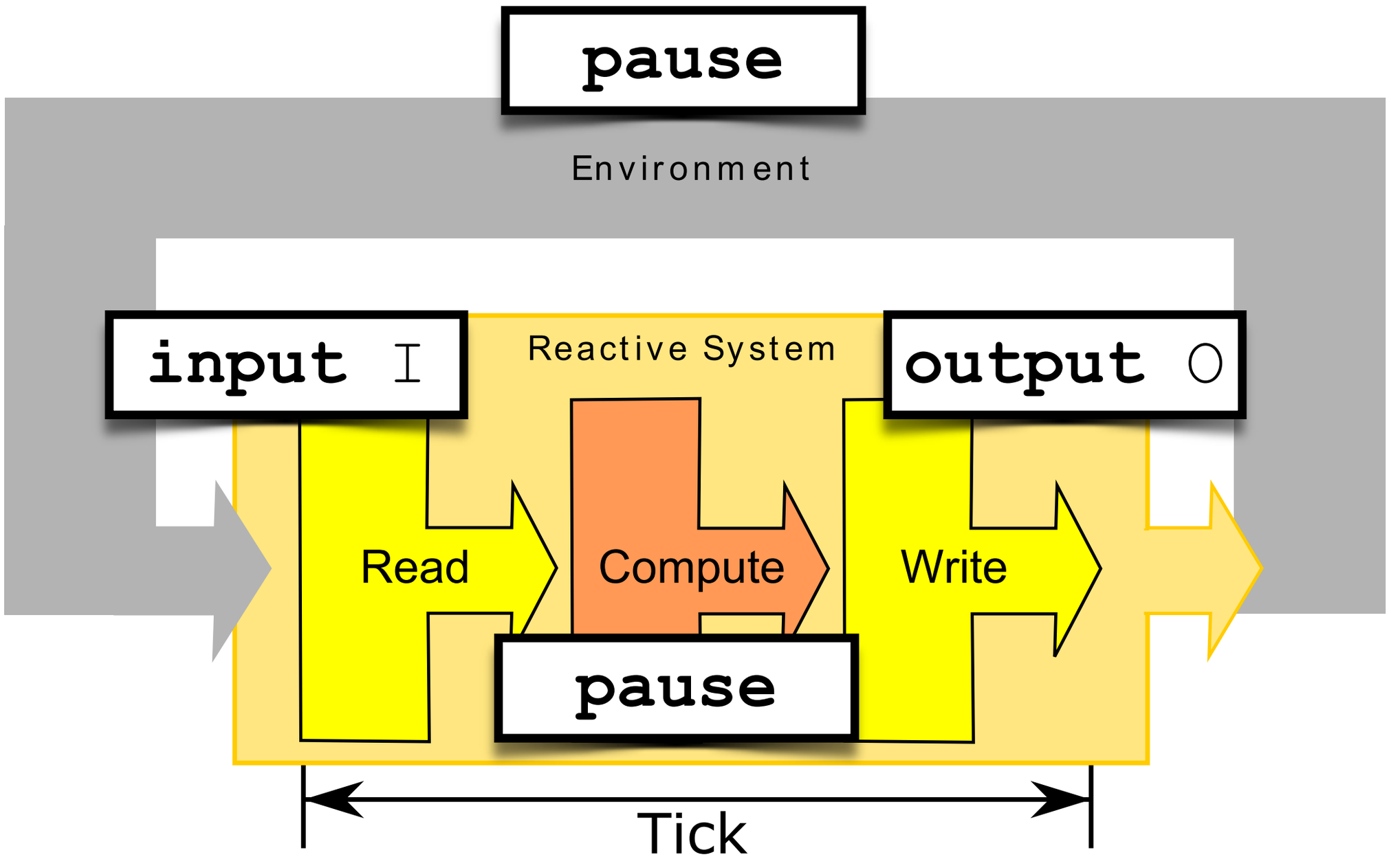
A work in progress report …

sc__**est**__ *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 determine outputs
R2: **`pause`** separates reactions



R1: inputs determine outputs
R2: **`pause`** separates reactions

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

$\Longrightarrow$ Programmer freedom

$\Longrightarrow$ Avoid timing issues within reaction

R1: inputs determine outputs
R2: **`pause`** separates reactions

	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

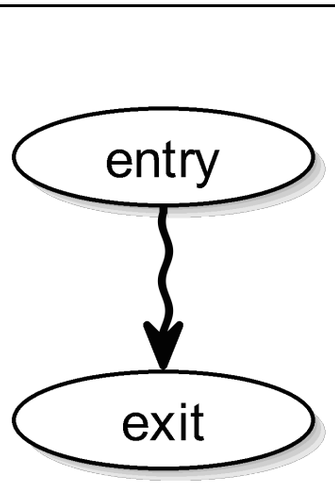
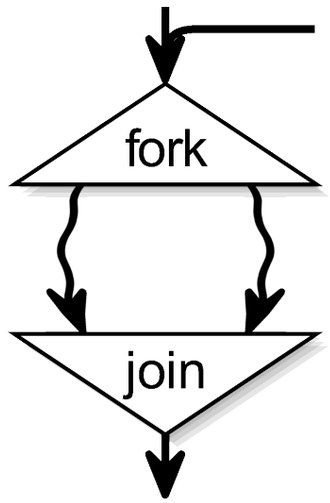
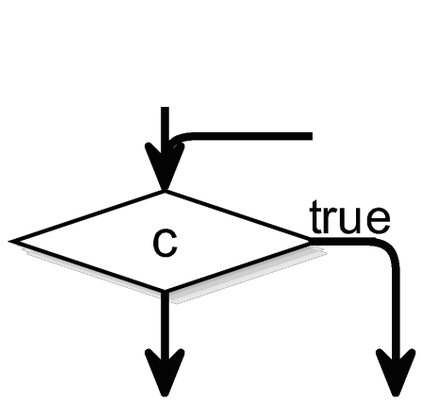
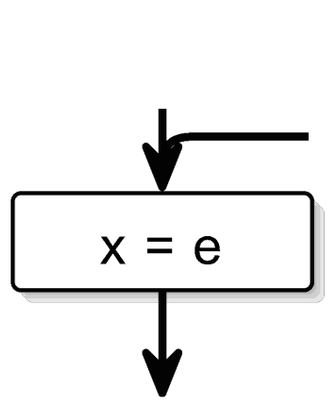
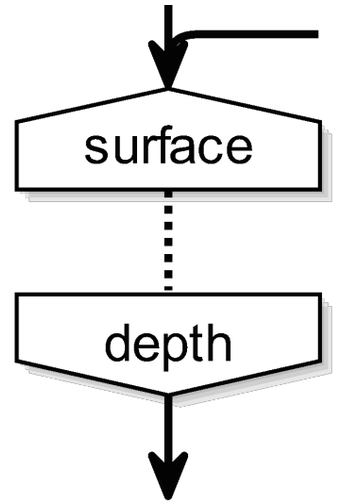
- Based on Sequentially Constructive MoC
- A **conservative** extension of Esterel
- Valid Esterel programs are valid SCEst programs, with same semantics
- Transformation rules for Esterel also hold for SCEst

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

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 **`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**

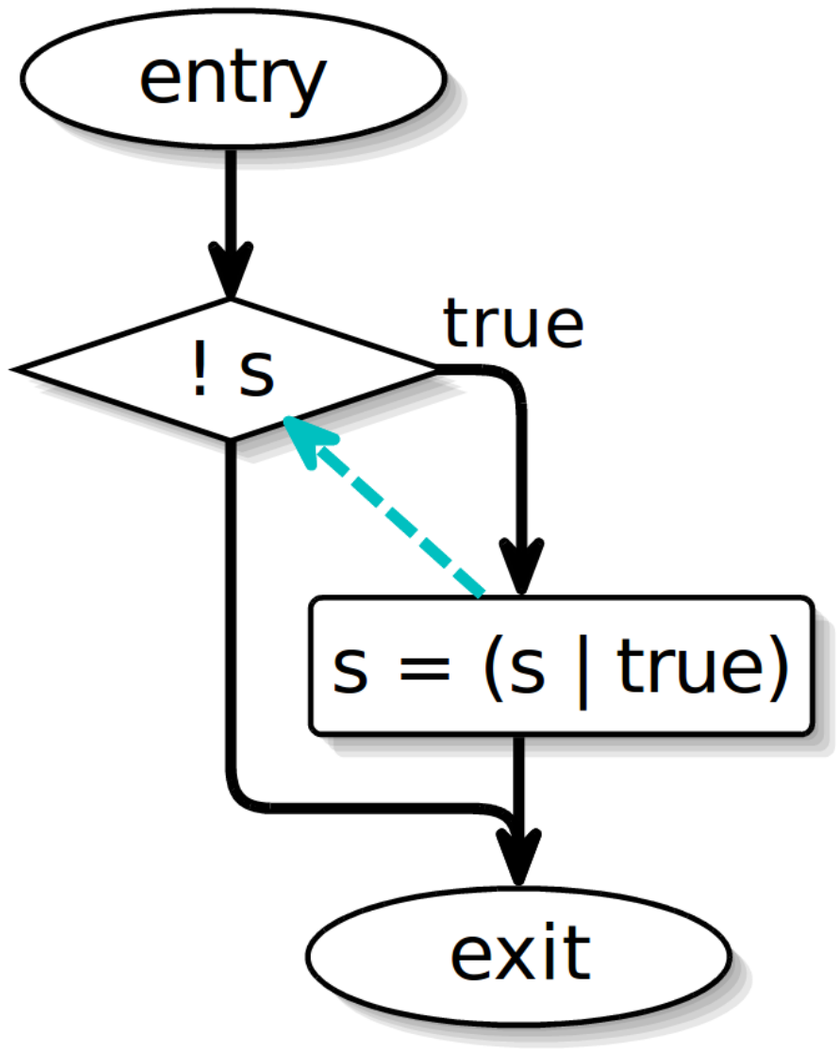
	C variables	SCL variables	Esterel variables	Esterel pure signals	Esterel signal values
Syntax	`x = y`	`x = y`	`x := y`	`emit x`	`emit x(?y)`
Type	arbitrary	arbitrary	arbitrary	present/ absent	arbitrary
Initialized each tick	no	no	no	yes (absent)	no
Persistence across ticks	yes	yes	yes	no	yes
Allow multiple values / tick	yes	yes	yes	no	no
Sequential scheduling constraints	**none**	**none**	**none**	first write ➔ reads	writes ➔ reads
Concurrent scheduling constraints	none	init ➔ updates ➔ reads	**read only**	first write ➔ reads	writes ➔ reads
I/O determinacy guaranteed	**no**	**yes**	**yes**	**yes**	**yes**

First Example

SCEst

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

SCG



SCL

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

First Rules

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

SCEst	SCL
`[` `  p || q` `]`	`fork` `  p par q` `join`
`loop` `  p` `end`	`l: p;` `  goto l`
`do` `  p` `while (c)`	`l: p;` `  if (c) goto l`
`await s`	`do` `  pause;` `while (!s)`
`while (c) {` `  p }`	`l: if (c) {` `  p; goto l }`
`await` `immediate s`	`while (!s) {` `  pause }`

Esterel Rules Still Hold

SCEst	SCEst
`halt`	`loop` `  pause` `end`
`loop` `  p` `each s`	`loop` `  abort` `  p;` `  halt` `  when s` `end`

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

SCEst	SCL
signal s **in** p **end**	{ **bool** s; **bool** f = **false**; **fork** p; f = **true** **par** l: s = false; **if** (!f) { **pause;** **goto** l } **join** }
signal s **in** pnt **end**	{ **bool** s; **fork** pnt **par** l: s = false; **pause; goto** l **join** }
emit s	s = s	**true**
present s …	**if** (s) …

Abort

SCEst	SCL
abort p **when** s	p [**pause** -> **pause; if** (s) **gotoj** l | **join** -> **join; if** (s) **gotoj** l]; l:

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

gotoj l: **goto** l, if **goto** in same thread as l
goto l_exit, otherwise

l_exit: label at end of thread

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

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
11:       pause;
          if (!A)
             goto 11
       par
12:       pause;
          if (!B)
             goto 12
       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
```

```
 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
```

```
 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
```

```
 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
```

```
   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
      fork
11:     pause;
        if (R) goto 14;
        if (!A) goto 11;
14:
      par
12:     pause;
        if (R) goto 15;
        if (!B) goto 12;
15:
      join;
      if (R) goto 16;
      emit O;
13:   pause;
      if (R) goto 16;
      goto 13;
16:end
```

loop

```
17:fork
11:   pause;
      if (R) goto 14;
      if (!A) goto 11;
14:
    par
12:   pause;
      if (R) goto 15;
      if (!B) goto 12;
15:
    join;
    if (R) goto 16;
    emit O;
13:pause;
    if (R) goto 16;
    goto 13;
16:goto 17
```

```
17:fork
11:  pause;
     if (R) goto 14;
     if (!A) goto 11;
14:
   par
12:  pause;
     if (R) goto 15;
     if (!B) goto 12;
15:
   join;
   if (R) goto 16;
   emit O;
13:pause;
   if (R) goto 16;
   goto 13;
16:goto 17
```

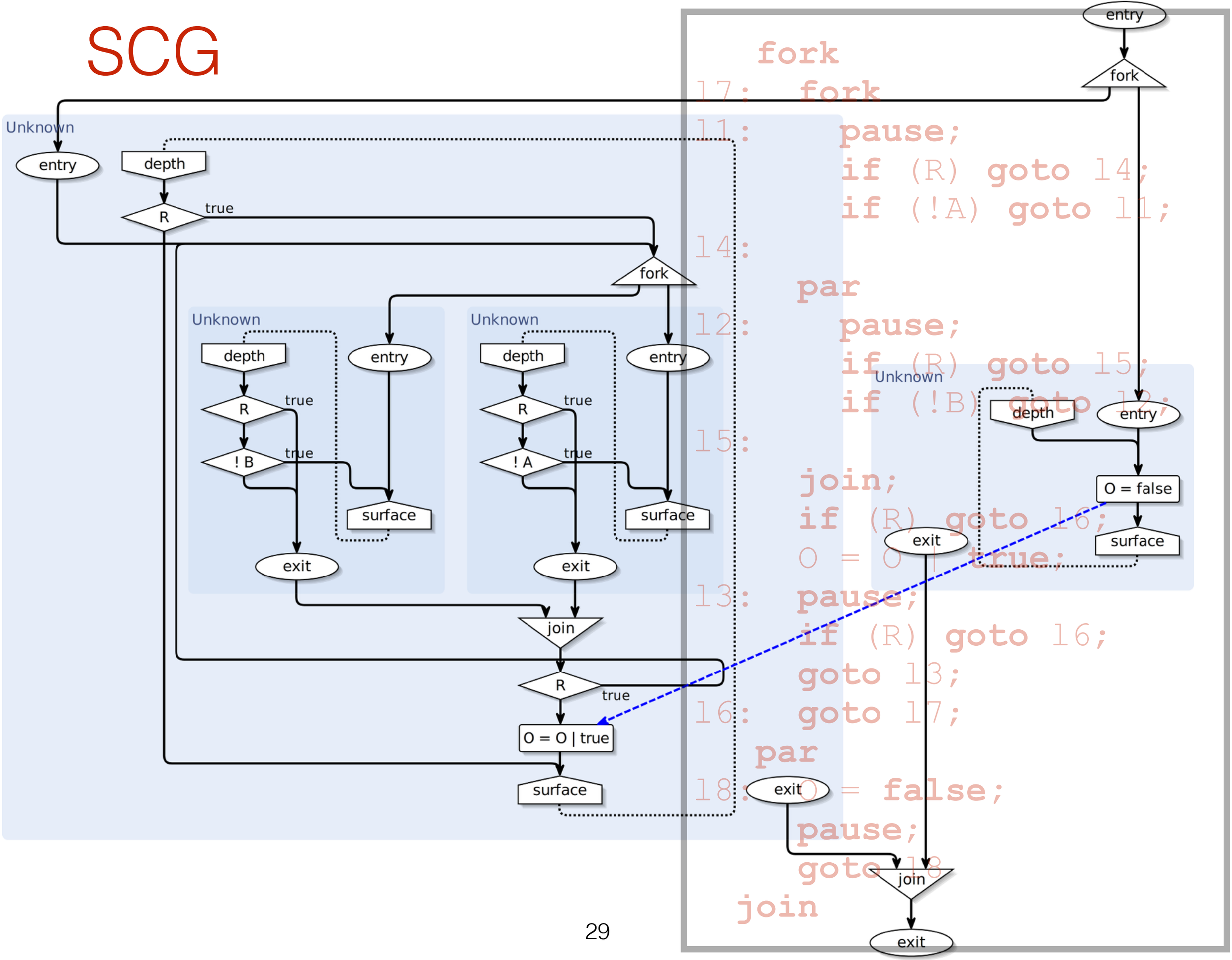
```
17:fork
11:   pause;
      if (R) goto 14;
      if (!A) goto 11;
14:
    par
12:   pause;
      if (R) goto 15;
      if (!B) goto 12;
15:
    join;
    if (R) goto 16;
    emit O;
13:pause;
    if (R) goto 16;
    goto 13;
16:goto 17;
```

emit, out-put

```
    fork
17:   fork
11:     pause;
        if (R) goto 14;
        if (!A) goto 11;
14:
      par
12:     pause;
        if (R) goto 15;
        if (!B) goto 12;
15:
      join;
      if (R) goto 16;
      O = O | true;
13:   pause;
      if (R) goto 16;
      goto 13;
16:   goto 17;
    par
18:   O = false;
      pause;
      goto 18
  join
```

init → update

SCG


Unknown
entry
depth
R
true
fork
Unknown
depth
R
true
! B
true
surface
exit
entry
Unknown
depth
R
true
! A
true
surface
exit
entry
join
R
true
O = O | true
surface
entry
fork
17: fork
11: pause;
if (R) goto 14;
if (!A) goto 11;
14:
par
12: pause;
if (R) goto 15;
if (!B) goto 12;
15:
join;
if (R) goto 16;
O = O | true;
13: pause;
if (R) goto 16;
goto 13;
16: goto 17;
par
18: O = false;
pause;
goto 18
join
Unknown
depth
entry
O = false
surface
exit
join
exit

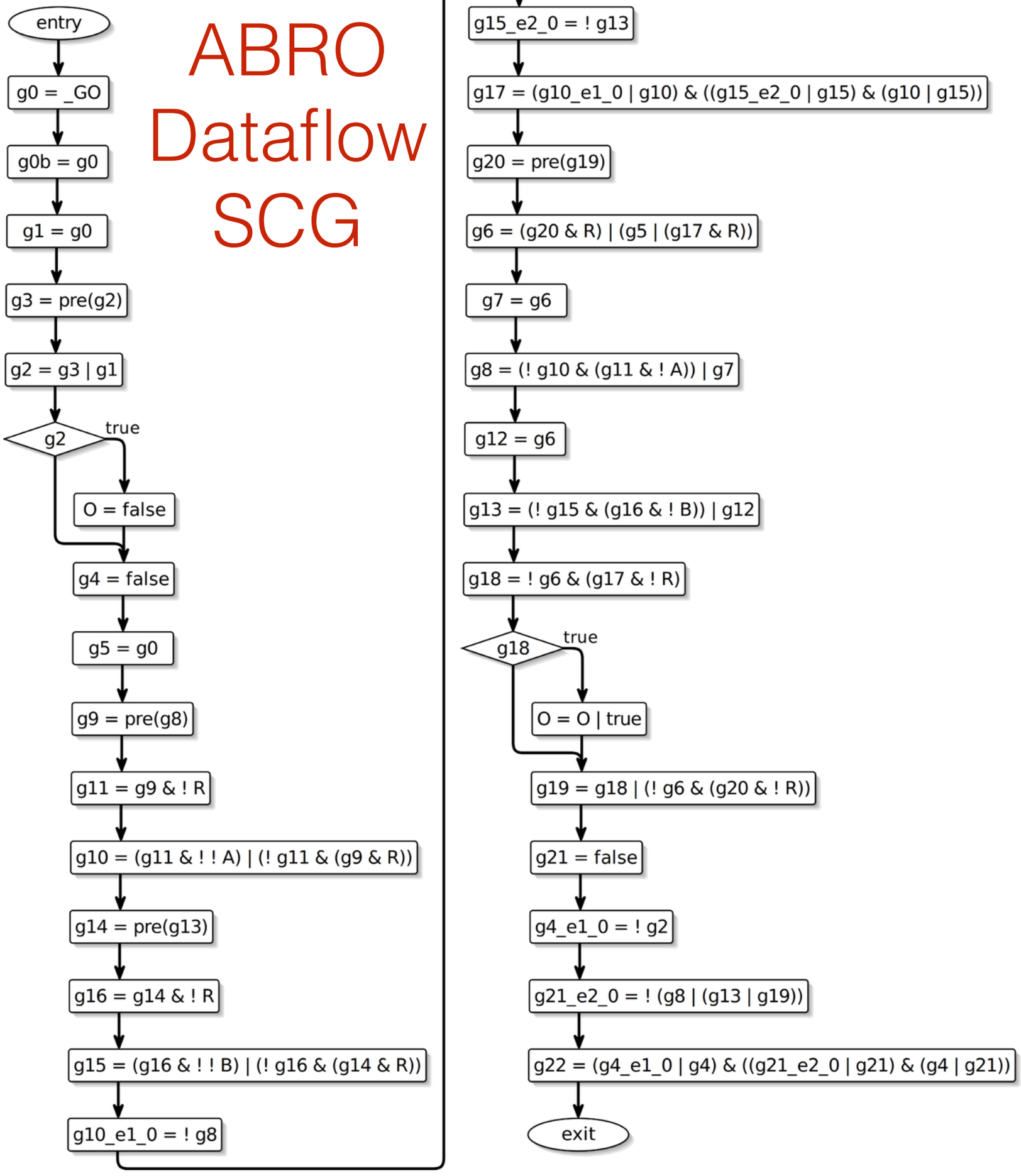
Downstream Compilation

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

ABRO Dataflow SCG




```
    fork
17:   fork
11:     pause;
        if (R) goto 14;
        if (!A)
          goto 11;
14:
      par
12:     pause;
        if (R) goto 15;
        if (!B)
          goto 12;
15:
      join;
      if (R) goto 16;
      O = O | true;
13:   pause;
      if (R) goto 16;
      goto 13;
16:   goto 17;
    par
18:   O = false;
      pause;
      goto 18
  join
```

SCCharts Modeling - SYNCHRON14/abro.strl - KIELER

File Edit Navigate Search Project Run Window Help

abro.strl

```
module ABRO:
input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O
each R

end module
```

Writable | Insert | 1 : 8

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

Outlook

- Complete compiler
- Get numbers
- Optimize
- From SCL to Esterel (SSA + ???)
- Schizophrenia (add "depth join")

SCEst

Simple Compilation of 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, ...

Trap / Exit

SCEst	SCL
trap s **in** p **end**	{ **bool** s = **false;** p [**exit** s -> s |= **true; gotoj** l | **pause** -> **if** (s) **gotoj** l; **pause** | **join** -> **join; if** (s) **gotoj** l] }; l:

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

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
```

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
```

Pure Signals, avoiding schizophrenia

To be applied if

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

`f`: fresh flag

`pni`: non-instantaneous statement(s)

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

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“