

Synchronous Languages—Lecture 21

Prof. Dr. Reinhard von Hanxleden
Steven Smyth

Christian-Albrechts Universität Kiel
Department of Computer Science
Real-Time Systems and Embedded Systems Group

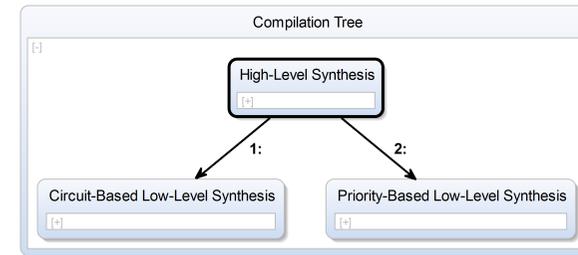
16 January 2013

Last compiled: January 31, 2014, 8:57 hrs



Code Generation for Sequential Constructiveness

Compilation — Overview



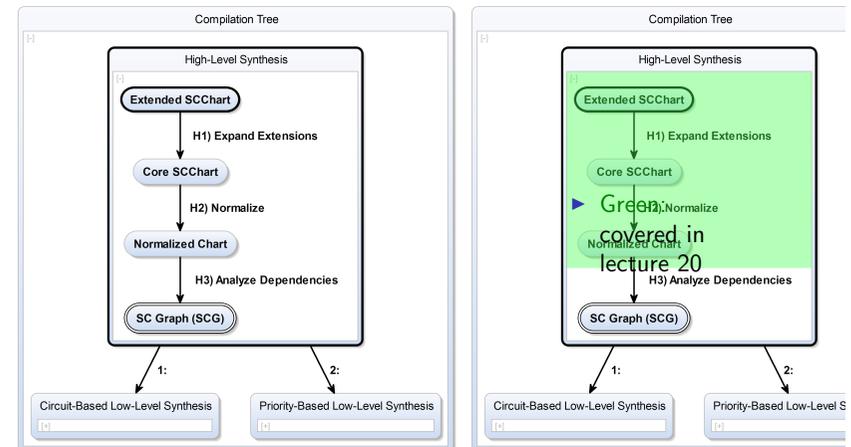
Overview

SCG Mapping & Dependency Analysis

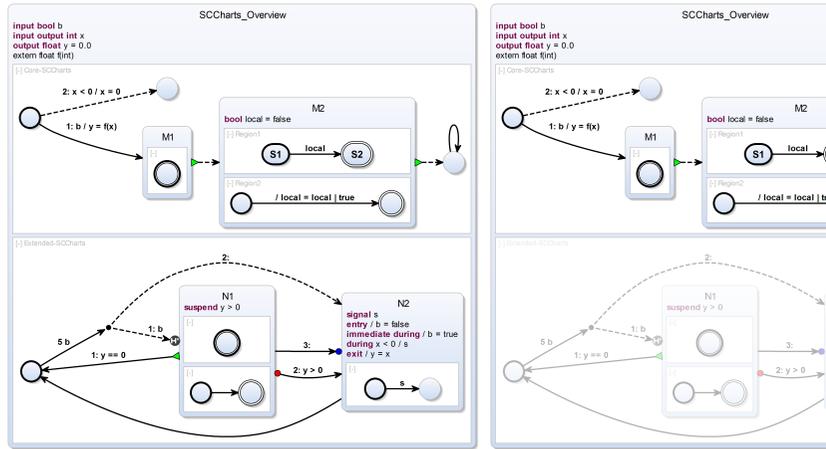
Code Generation Approaches

SCCharts Benefits

Compilation — High-Level Synthesis



(Recall) SCCharts - Core & Extended Features



Overview

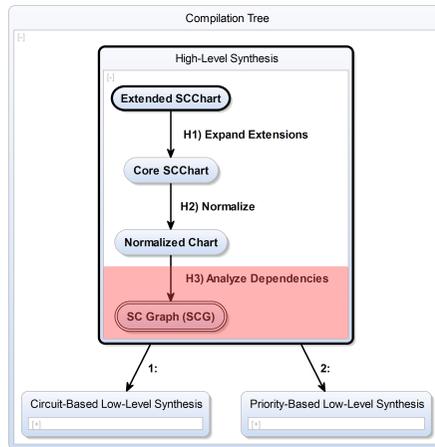
SCG Mapping & Dependency Analysis

- Compilation Overview
- The SC Graph
- Map to SC Graph
- Dependency Analysis

Code Generation Approaches

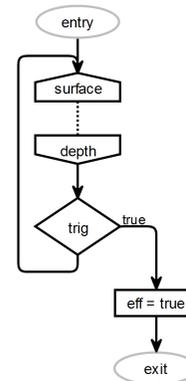
SCCharts Benefits

Compilation — High-Level Synthesis



► Red:
upcoming now

The SC Graph

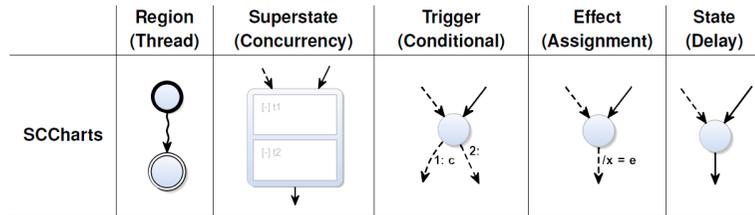


SC Graph:

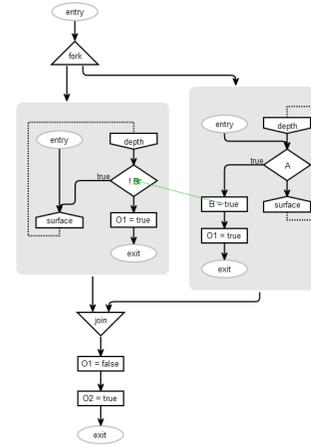
Labeled graph $G = (S, E)$

- Nodes S correspond to statements of sequential program
- Edges E reflect sequential execution control flow

High-Level Step 3: Map to SC Graph



The SC Graph — Dependencies



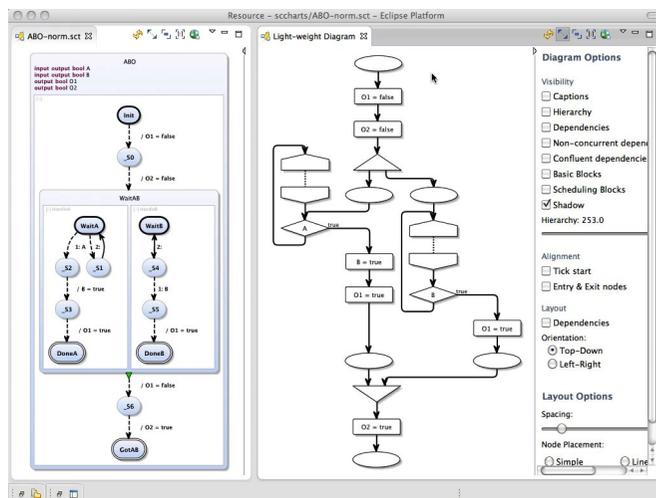
Two assignments within the SC Graph are **concurrent** iff

- ▶ they share a *least common ancestor fork* node.

Two assignments are **confluent** iff

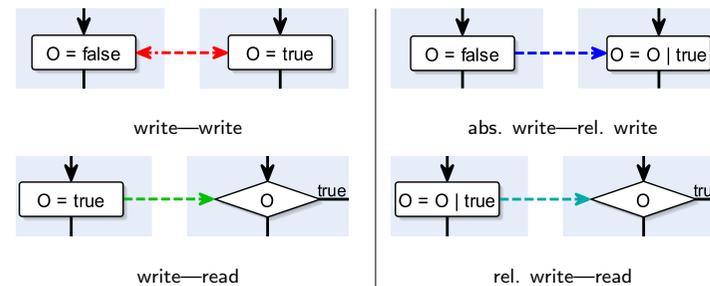
- ▶ the order of their assignments does not matter.

Example: Mapping ABO to SCG



Dependency Types

Dependencies are further categorized in



The SC MoC employs a strict “initialize - update - read” protocol.

(More on the SC MoC will follow in Lecture 22.)

Overview

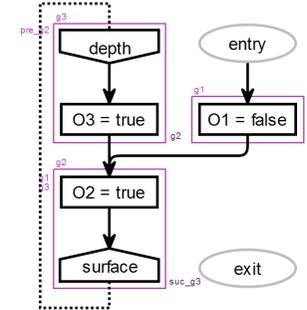
SCG Mapping & Dependency Analysis

Code Generation Approaches

Circuit-based Approach
Priority-based Approach
Approach Comparison

SCCharts Benefits

Basic Blocks



Basic Block:

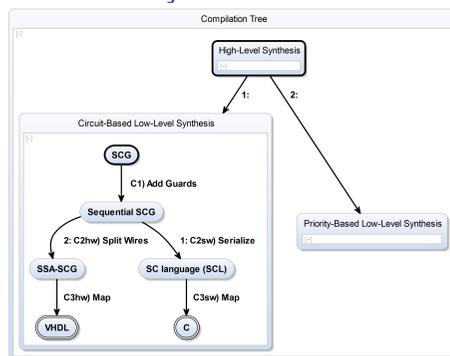
A collection of SCG nodes / SCL statements

- ▶ that can be executed monolithically

Rules:

- ▶ Split at nodes with more than one incoming control flow edge
- ▶ Split at nodes with more than one outgoing control flow edge
- ▶ Split at tick edges
- ▶ Split after fork nodes and before join nodes
- ▶ **Each node can only be included in one basic block at any time**

Low-Level Synthesis I: The Circuit Approach

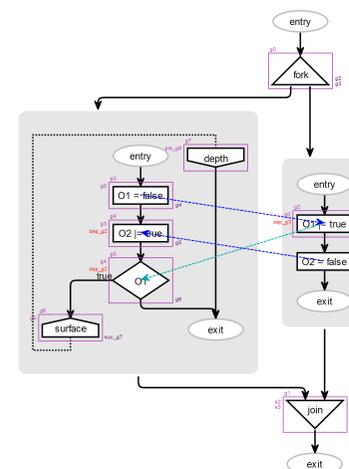


- ▶ Basic idea: Generate netlist
- ▶ Precondition: Acyclic SCG (with dependency edges, but without tick edges)
- ▶ Well-established approach for compiling SyncCharts/Esterel

Differences to Esterel circuit semantics [Berry '02]

1. Simpler translation rules, as aborts/traps/suspensions already transformed away during high-level synthesis
2. SC MoC permits sequential assignments

Scheduling Blocks



- ▶ Basic blocks may be interrupted when a data dependency interferes.

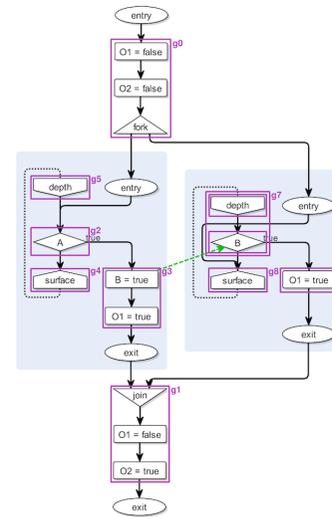
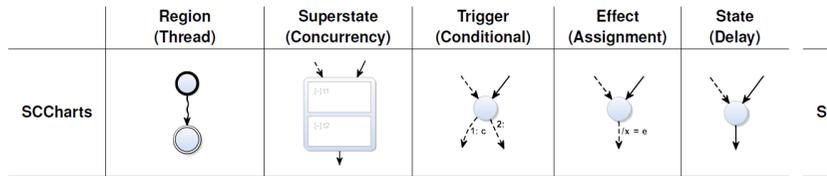
- ▶ Structure basic blocks further: **Scheduling Blocks**

Rules:

- ▶ Split a basic block at incoming dependency edge

But...

- ▶ want to minimize the number of context switches
- ▶ ⇒ Room for optimization!

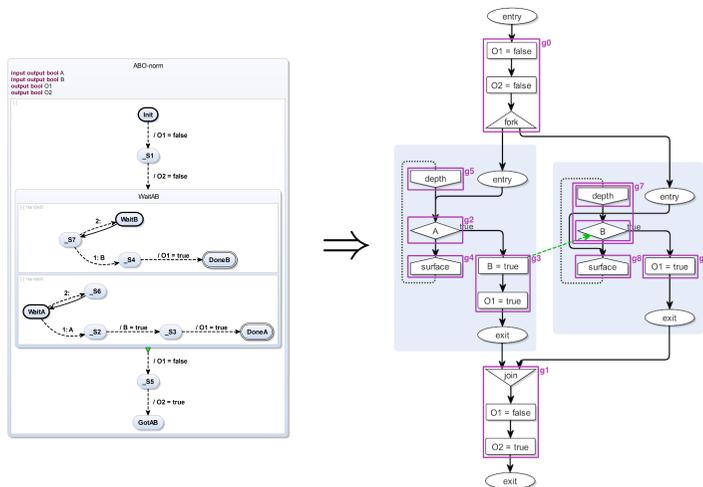


```

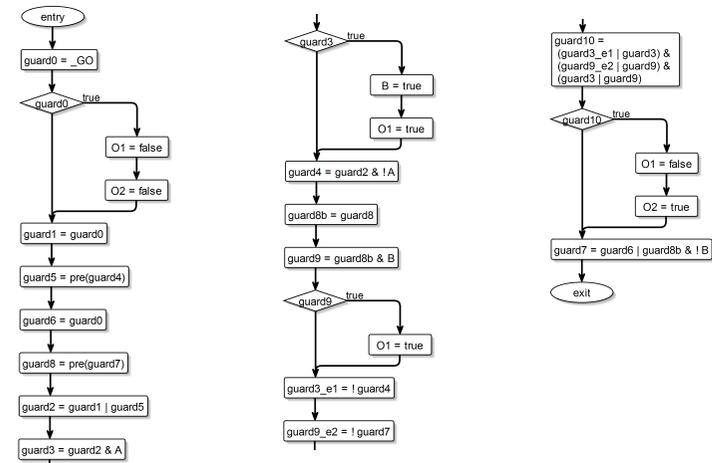
1 module ABO
2 input output bool
  A, B;
3 output bool O1,
  O2;
4 {
5   O1 = false;
6   O2 = false;
7   fork
8     HandleA:
9     if (!A) {
10      pause;
11      goto HandleA
12    };
13   B = true;
14   O1 = true;
15   par
16     HandleB:
17     pause;
18     if (!B) {
19      goto HandleB
20    };
21   O1 = true;
22   join;
23   O1 = false;
24   O2 = true;
25 }

```

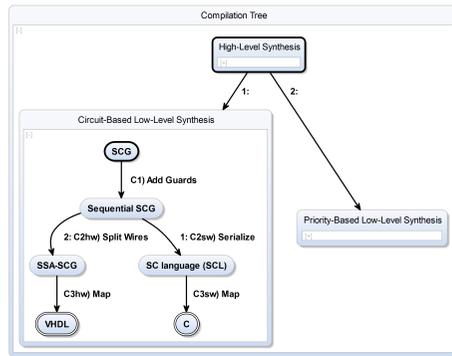
ABO SCG, With Dependencies and Scheduling Blocks



Sequential SCG — ABO

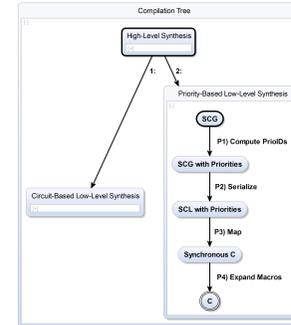


(Recall) Low-Level Synthesis I: The Circuit Approach



- ▶ Can use sequential SCL directly for SW synthesis
- ▶ Synthesizing HW needs a little further work ...

Low-Level Synthesis II: The Priority Approach



- ▶ More software-like
- ▶ Don't emulate control flow with guards/basic blocks, but with program counters/threads
- ▶ Priority-based thread dispatching
- ▶ SCL_P : SCL + PrioIDs
- ▶ Implemented as C macros

Differences to Synchronous C [von Hanxleden '09]

- ▶ No preemption \Rightarrow don't need to keep track of thread hierarchies
- ▶ Fewer, more light-weight operators
- ▶ RISC instead of CISC
- ▶ More human-friendly syntax

ABO SCL, Logic Synthesis (HW)

Difference to software

```

1 module ABO-seq
2 input output bool A, B;
3 output bool O1, O2;
4 bool GO, g0, g1, ...
5 {
6   g0 = GO;
7   if g0 {
8     O1 = false;
9     O2 = false;
10  };
11  g5 = g4_pre;
12  g7 = g8_pre;
13  g2 = g0 || g5;
14  g3 = g2 && A;
15  if g3 {
16    B = true;
17    O1 = true;
18  };
19  g4 = g2 && ! A;
20  g6 = g7 && B;
21  if g6 {
22    O1 = true;
23  };
24  g8 = g0 || (g7 && ! B);
25  e2 = ! g4;
26  e6 = ! g8;
27  ...

```



```

1 ARCHITECTURE behavior OF ABO IS
2 -- local signals definition, hidden
3 begin
4 -- main logic
5 g0 <= GO_local;
6 O1 <= false WHEN g0 ELSE O1_pre;
7 O2 <= false WHEN g0 ELSE O2_pre;
8 g5 <= g4_pre;
9 g7 <= g8_pre;
10 g2 <= g0 or g5;
11 g3 <= g2 and A_local;
12 B <= true WHEN g3 ELSE B_local;
13 O1_2 <= true WHEN g3 ELSE O1;
14 g4 <= g2 and not A_local;
15 g6 <= g7 and B;
16 O1_3 <= true WHEN g6 ELSE O1_2;
17 g8 <= g0 or (g7 and not B);
18 e2 <= not (g4);
19 e6 <= not (g8);
20 ...

```

- ▶ All persistence (state, data) in external reg's ("pre"-var's)
- ▶ Permit only one value per wire per tick \Rightarrow SSA

SCL_P Macros I

```

1 // Declare Boolean type
2 typedef int bool;
3 #define false 0
4 #define true 1
5
6 // Generate "_L<line-number>" label
7 #define _concat_helper(a, b) a ## b
8 #define _concat(a, b) _concat_helper(a, b)
9 #define _label_ _concat(_L, _LINE_)
10
11 // Enable/disable threads with prioID p
12 #define _u2b(u) (1 << u)
13 #define _enable(p) _enabled |= _u2b(p); _active |= _u2b(p)
14 #define _isEnabled(p) ((_enabled & _u2b(p)) != 0)
15 #define _disable(p) _enabled &= ~_u2b(p)

```

SCL_P Macros II

```

17 // Set current thread continuation
18 #define _setPC(p, label) _pc[p] = &label
19
20 // Pause, resume at <label> or at pause
21 #define _pause(label) _setPC(_cid, label); goto _L_PAUSE
22 #define pause      _pause(_label_); _label_:
23
24 // Fork/join sibling thread with prioID p
25 #define fork1(label, p) _setPC(p, label); _enable(p);
26 #define join1(p)      _label_: if (_isEnabled(p)) { _pause(_label_); }
27
28 // Terminate thread at "par"
29 #define par          goto _L_TERM;
30
31 // Context switch (change prioID)
32 #define _prio(p)     _deactivate(_cid); _disable(_cid); _cid = p; \
33 _enable(_cid); _setPC(_cid, _label_); goto _L_DISPATCH; _label_:

```

ABO SCL_P I

```

85 int tick()
86 {
87     tickstart(2);
88     O1 = false;
89     O2 = false;
90
91     fork1(HandleB,
92           1) {
93         HandleA:
94         if (!A) {
95             pause;
96             goto HandleA
97         }
98         B = true;
99         O1 = true;
100     } par {

```

⇒

```

85 int tick()
86 {
87     if (_notInitial) { _active = _enabled;
88         goto _L_DISPATCH; } else { _pc[0]
89         = &&_L_TICKEND; _enabled = (1 <<
90         0); _active = _enabled; _cid = 2;
91         ; _enabled |= (1 << _cid); _active
92         |= (1 << _cid); _notInitial = 1;
93         } ;
94     O1 = 0;
95     O2 = 0;
96
97     _pc[1] = &&HandleB; _enabled |= (1 <<
98         1); _active |= (1 << 1); {
99     HandleA:
100     if (!A) {
101         _pc[_cid] = &&_L94; goto _L_PAUSE;
102         _L94:;
103         goto HandleA;
104     }
105     B = 1;
106     O1 = 1;
107
108     } goto _L_TERM; {

```

ABO SCL_P II

```

102 HandleB:
103 _pc[_cid] = &&_L103; goto _L_PAUSE;
104 _L103:;
105 if (!B) {
106     goto HandleB;
107 }
108 O1 = 1;
109 } _L108: if ((_enabled & (1 << 2)) !=
110 0) { _pc[_cid] = &&_L108; goto
111 _L_PAUSE; };
112
113 }

```

⇒

```

102 HandleB:
103 pause;
104 if (!B) {
105     goto HandleB;
106 }
107 O1 = 1;
108 } _L108: if ((_enabled & (1 << 2)) !=
109 0) { _pc[_cid] = &&_L108; goto
110 _L_PAUSE; };
111 O1 = 0;
112 O2 = 1;
113 goto _L_TERM; _L_TICKEND: return (
114 _enabled != (1 << 0)); _L_TERM:
115 _enabled &= ~(1 << _cid); _L_PAUSE
116 : _active &= ~(1 << _cid);
117 _L_DISPATCH: __asm volatile("bsrl
118 %1,%0\n" : "=r" (_cid) : "r" (
119 _active) ); goto *_pc[_cid];
120
121 }

```

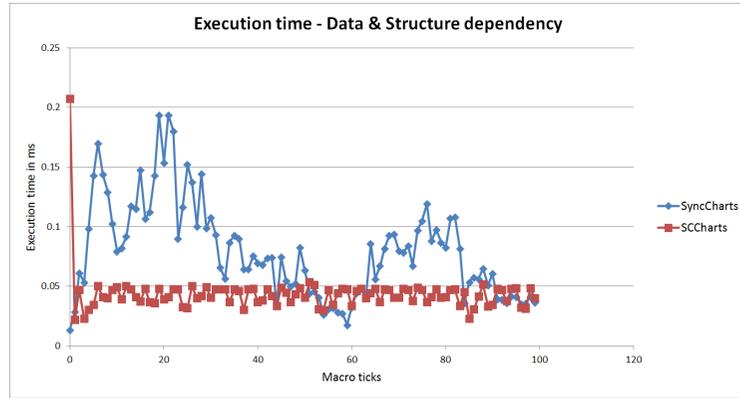
SCG Mapping & Dependency Analysis
Code Generation Approaches
SCCharts Benefits

Circuit-based Approach
Priority-based Approach
Approach Comparison

Comparison of Low-Level Synthesis Approaches

	Circuit	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)	+	-

Comparison — Jitter



Execution time comparison of statecharts with multiple hierarchies depicts

- ▶ low jitter in circuit-based approach
- ▶ execution time in priority-based approach more dependant to structure and input data of the statechart

... What About That Acyclicity?

```

1 module schizo
2 output O;
3
4 loop
5   signal S in
6   present S
7   then
8     emit O
9   end;
10  pause;
11  emit S;
12 end;
13 end loop
14 end module
    
```

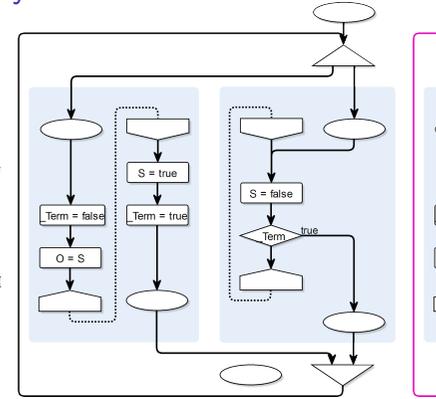
⇒

```

1 module schizo-conc
2 output bool O;
3 {
4   while (true) {
5     bool S, _Term;
6
7     fork
8       _Term = false;
9       O = S;
10      pause;
11      S = true;
12      _Term = true;
13    par
14      while (true) {
15        S = false;
16        if (_Term)
17          break;
18        pause;
19      }
20    join;
21  }
22 }
    
```

Esterel
[Tardieu & de Simone '04]

SCL (1st try)



Q: The problem?
A: Instantaneous loop!
a.k.a. Signal reincarnation
a.k.a. Schizophrenia

Overview

SCG Mapping & Dependency Analysis

Code Generation Approaches

SCCharts Benefits
Schizophrenia
Summary

A Solution

```

1 module schizo-cured
2 output O;
3
4 loop
5   signal S in
6   present S then
7     emit O
8   end;
9   pause;
10  emit S;
11 end;
12 signal S' in
13 present S' then
14   emit O
15 end;
16 pause;
17 emit S';
18 end;
19 end loop
    
```

Esterel

- ▶ Duplicated loop body to separate signal instances
- ▶ Q: Complexity?
- ▶ A: Exponential ☹️

A Better Solution

```

1 module schizo
2 output O;
3
4 loop
5 signal S in
6 present S
7 then
8 emit O
9 end;
10 pause;
11 emit S;
12 end;
13 end loop
14 end module

1 module
2 schizo-cured2-str1
3 output O;
4
5 loop
6 % Surface
7 signal S in
8 present S then
9 emit O
10 end;
11 end;
12
13 % Depth
14 signal S' in
15 pause;
16 emit S';
17 end;
18 end loop
    
```

Esterel

[Tardieu & de Simone '04]
(simplified)

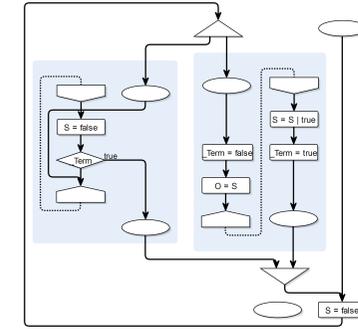
- ▶ Duplicated loop body
- ▶ “Surface copy” transfers control immediately to “depth copy”
- ▶ Q: Complexity?
- ▶ A: Quadratic ☹

SCG for schizo-cured

```

1 module schizo-cured
2 output bool O;
3 {
4 while (true) {
5 bool S, _Term;
6
7 // Surf init
8 S = false;
9 fork
10 _Term = false;
11 O = S;
12 pause;
13 S = true;
14 _Term = true;
15 par
16 do {
17 pause;
18 // Depth init
19 S = false;
20 } while (!_Term);
21 join;
22 }
23 }
    
```

SCL



- ▶ Cycle now broken by delay
- ▶ Only the “depth initialization” of S creates a concurrent “initialize before update” scheduling dependence

The SCL Solution

```

1 module schizo
2 output O;
3
4 loop
5 signal S in
6 present S
7 then
8 emit O
9 end;
10 pause;
11 emit S;
12 end;
13 end loop
14 end module

1 module schizo-cured
2 output bool O;
3 {
4 while (true) {
5 bool S, _Term;
6
7 // Surf init
8 S = false;
9 fork
10 _Term = false;
11 O = S;
12 pause;
13 S = true;
14 _Term = true;
15 par
16 do {
17 pause;
18 // Depth init
19 S = false;
20 } while (!_Term);
21 join;
22 }
23 }
    
```

Esterel

SCL

- ▶ “Surface initialization” at beginning of scope
- ▶ Delayed, concurrent “depth initialization”
- ▶ Q: Complexity?
- ▶ A: Linear ☺
- ▶ Caveat: We only talk about **signal** reincarnation, *i. e.*, instantaneously exiting and entering a signal scope
- ▶ Reincarnated **statements** still require duplication (quadratic worst case?)

Summary

1. Sequential Constructiveness **natural** for synchrony
 2. Same semantic foundation from Extended SCCharts down to machine instructions/physical gates
 - ▶ Modeler/programmer has direct access to target platform
 - ▶ No conceptual breaks, *e. g.*, when mapping signals to variables
 3. Efficient synthesis paths for hw and sw, building on established techniques (circuit semantics, guarded actions, SSA, ...)
 4. Treating advanced constructs as syntactic sugar simplifies down-stream synthesis (CISC vs. RISC)
 5. **Plenty of future work:** compilation of Esterel-like languages, trade-off RISC vs. CISC, WCRT analysis, timing-predictable design flows (→ PRETSY), multi-clock, visualization, ...
- ⇒ Visit us if you are interested in the subject!

To Go Further

-  KIELER website: <http://rtsys.informatik.uni-kiel.de/kieler>
-  C. André. *Semantics of SyncCharts*. Technical Report ISRN I3S/RR-2003-24-FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
-  G. Berry. *The foundations of Esterel*. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425-454, Cambridge, MA, USA, 2000.
-  R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications*. Technical Report 1311, Christian-Albrechts-Universitaet zu Kiel, Department of Computer Science, Dec 2013.
-  R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. O'Brien. *Sequentially Constructive Concurrency – A conservative extension of the synchronous model of computation*. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, Grenoble, France, March 2013.