Synchronous Languages—Lecture 12

Prof. Dr. Reinhard von Hanxleden

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

11 Dec. 2018

Last compiled: January 29, 2019, 10:55 hrs

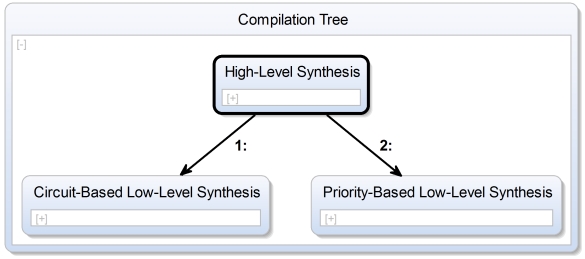
Code Generation for Sequential Constructiveness

The 5-Minute Review Session

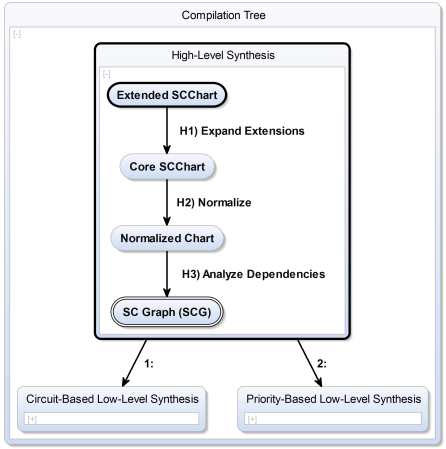
1. What are *Statecharts*? Who invented them?
2. What is the difference between *SyncCharts* and Statecharts?
3. How can we transform Esterel to SyncCharts? How about the other direction?
4. What are *SCCharts*? What is their motivation?
5. What are *Core SCCharts*?

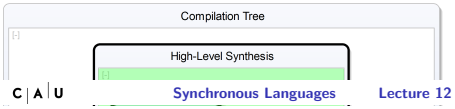
Overview

Compilation — Overview



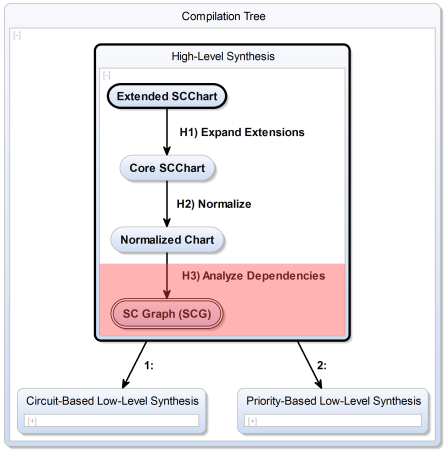
Compilation — High-Level Synthesis





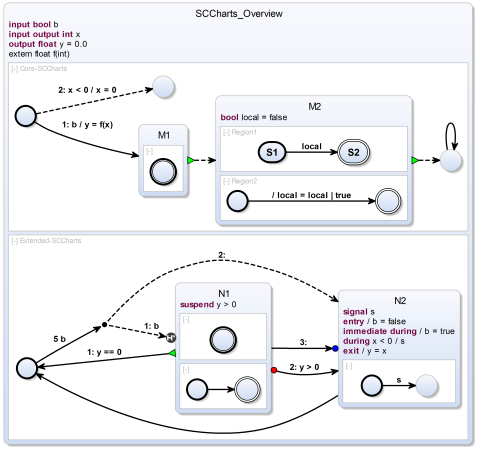
- Green:
  covered in
  previous lecture

Compilation — High-Level Synthesis



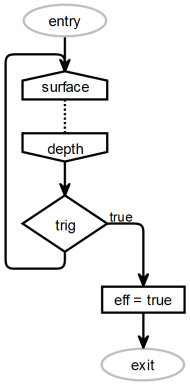
- Red:
  coming up now

(Recall) SCCharts - Core & Extended Features



Overview

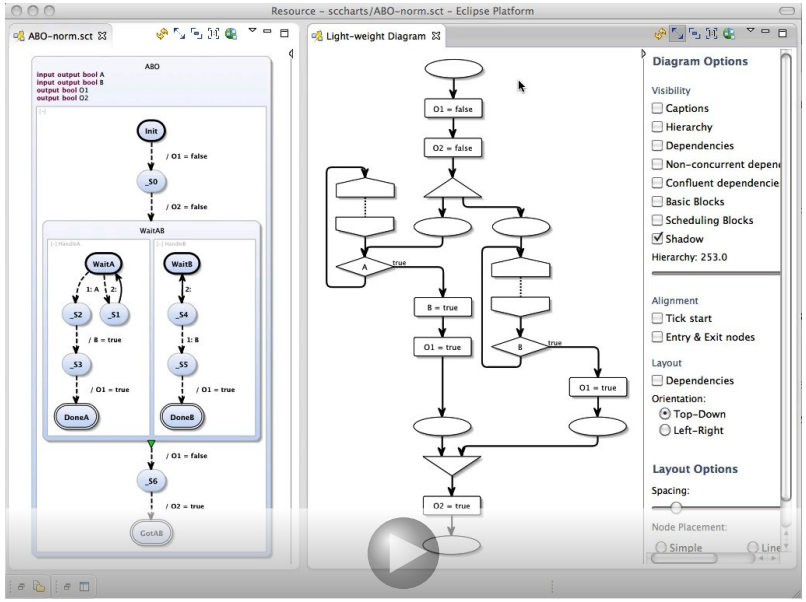
The SC Graph



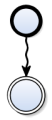
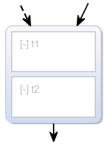
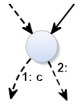
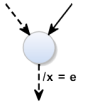
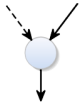
SC Graph:
Labeled graph $G = (S, E)$

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

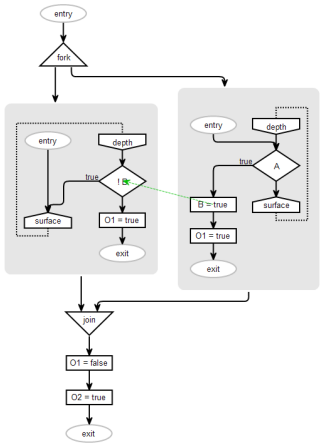
Example: Mapping ABO to SCG



High-Level Step 3: Map to SC Graph

	Region (Thread)	Superstate (Concurrency)	Trigger (Conditional)	Effect (Assignment)	State (Delay)
SCCharts					

The SC Graph — Dependencies



Two assignments within the SC Graph are concurrent iff

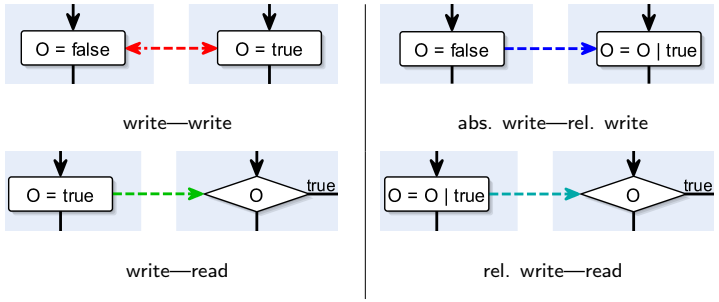
- they share a *least common ancestor fork* node.

Two assignments are confluent iff

- the order of their assignments does not matter.

Dependency Types

Dependencies are further categorized in

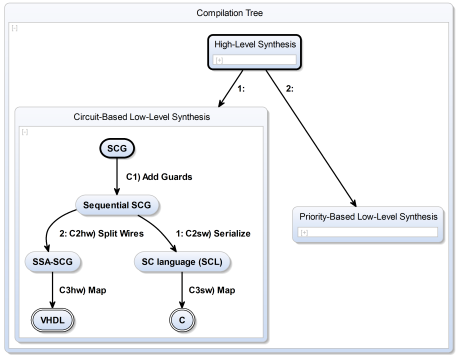


The SC MoC employs a strict "initialize - update - read" protocol.

(More on the SC MoC will follow in next lecture.)

Overview

Low-Level Synthesis I: The Circuit Approach

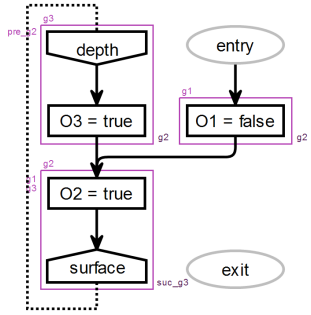


- Basic idea: Generate netlist
- Precondition: Acyclic SCG (with dependency edges, but without tick edges)
- Well-established approach for compiling SyncCharts/Esterel

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

Basic Blocks



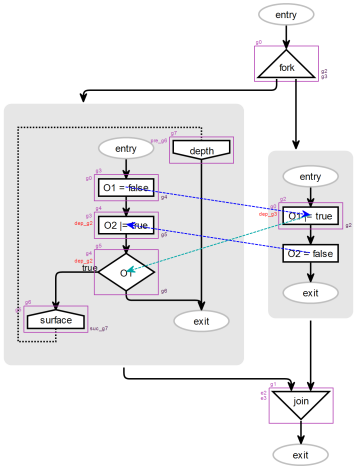
Basic Block:
A collection of SCG nodes / SCL statements

- that can be executed monolithically

Rules:

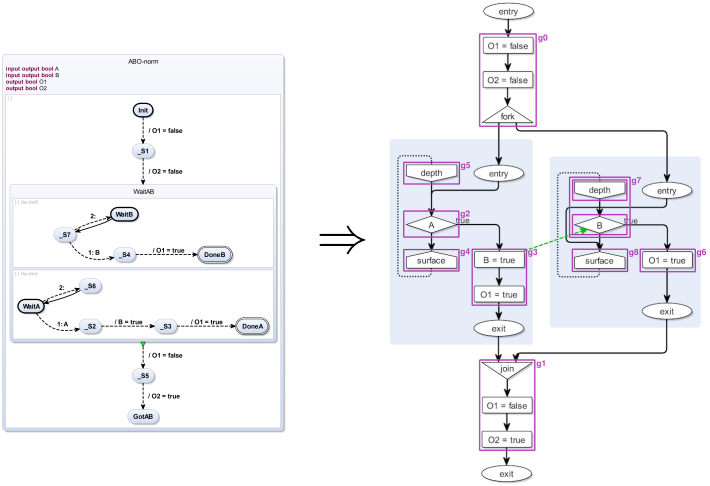
- Split at nodes with more than one incoming control flow edge
- Split at nodes with more than one outgoing control flow edge
- Split at tick edges
- Split after fork nodes and before join nodes
- **Each node can only be included in one basic block at any time**

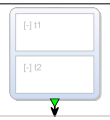
Scheduling Blocks

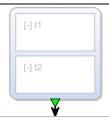
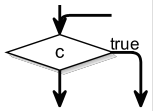
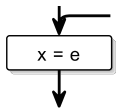
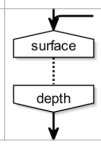
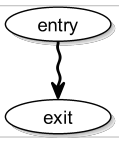
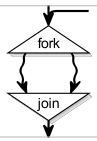


- Basic blocks may be interrupted when a data dependency interferes.
- Structure basic blocks further: Scheduling Blocks
- Rules:
  - Split a basic block at incoming dependency edge
- But...
  - want to minimize the number of context switches
  - ⇒ Room for optimization!

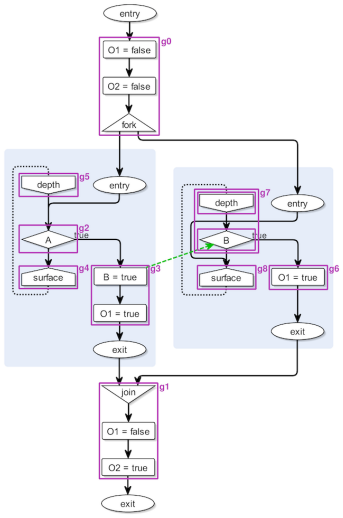
ABO SCG, With Dependencies and Scheduling Blocks



	Trigger (Conditional)	Effect (Assignment)	State (Delay)	Region (Thread)	Superstate (Concurrency)
Normalized Core SCCharts					

	Trigger (Conditional)	Effect (Assignment)	State (Delay)	Region (Thread)	Superstate (Concurrency)
Normalized Core SCCharts					
SCL	if (c) s_1 else s_2	$x = e$	pause	t	fork t_1 par t_2 join
SCG					

	Trigger (Conditional)	Effect (Assignment)	State (Delay)	Region (Thread)	Superstate (Concurrency)

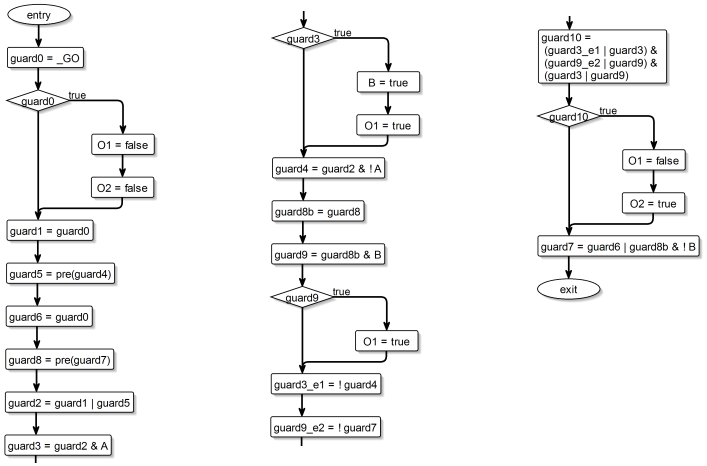



```
1 module ABO
2 input output bool
        A, B;
3 output bool O1,
        O2;
4 {
5   O1 = false;
6   O2 = false;
7   fork
8     HandleA:
9     if (!A) {
10      pause;
11      goto HandleA
              ;
12    };
13    B = true;
14    O1 = true;
15  par
16    HandleB:
17    pause;
18    if (!B)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

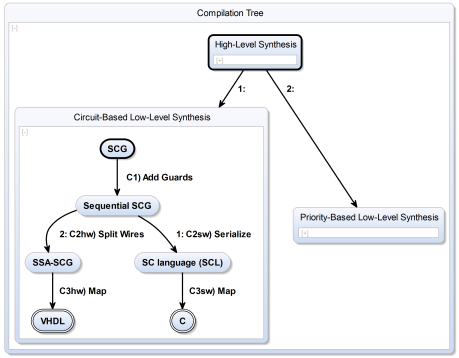
⇒

```
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  g1 = (g3 || e2) &&
28    (g6 || e6) && (g3 ||
29  if g1 {
30    O1 = false;
31    O2 = true;
32  };
33  g4_pre = g4;
34  g8_pre = g8;
35 }
```

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 . . .

ABO SCL, Logic Synthesis (HW)

Difference to software

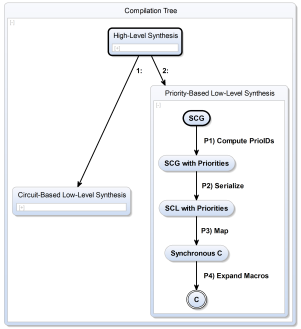
```
1 module ABO-seq
2 input output bool A, B;
3 output bool O1, O2;
4 bool GO, g0, g1, ...
5 {
6  g0 = GO;
7  if g0 {
8    O1 = false;
9    O2 = false;
10 };
11 g5 = g4_pre;
12 g7 = g8_pre;
13 g2 = g0 || g5;
14 g3 = g2 && A;
15 if g3 {
16   B = true;
17   O1 = true;
18 };
19 g4 = g2 && ! A;
20 g6 =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

⇒

```
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 g1 <= (g3 or e2) and
```

- All persistence (state, data) in external reg's ("_pre"-var's)
- Permit only one value per wire per tick ⇒ SSA

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 ⇒ don't need to keep track of thread hierarchies
- Fewer, more light-weight operators
- RISC instead of CISC
- More human-friendly syntax

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)
```

ABO SCL$_P$ I

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

⇒

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

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$_P$ II

```
102    HandleB:
103    pause;
104    if (!B) {
105      goto HandleB
             ;
106    }
107    O1 = true;
108  } join1(2);
109
110  O1 = false;
111  O2 = true;
112  tickreturn;
113 }
```

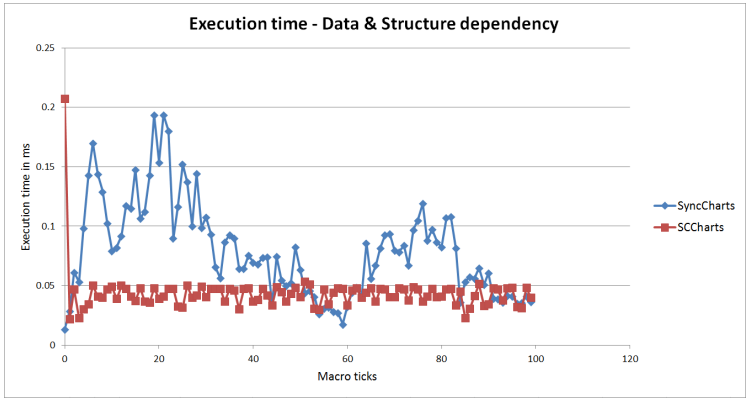
⇒

```
102    HandleB:
103    _pc[_cid] = &&_L103; goto _L_PAUSE;
           _L103:;
104    if (!B) {
105      goto HandleB;
106    }
107    O1 = 1;
108  } _L108: if (((_enabled & (1 << 2)) !=
         0)) { _pc[_cid] = &&_L108; goto
        _L_PAUSE; };
109
110  O1 = 0;
111  O2 = 1;
112  goto _L_TERM; _L_TICKEND: return (
        _enabled != (1 << 0)); _L_TERM:
        _enabled &= ~(1 << _cid); _L_PAUSE
        : _active &= ~(1 << _cid);
        _L_DISPATCH: __asm volatile("bsrl
        %1,%0\n" : "=r" (_cid) : "r" (
        _active) ); goto *_pc[_cid];
113 }
```

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

Overview

. . . What About That Acyclicity?

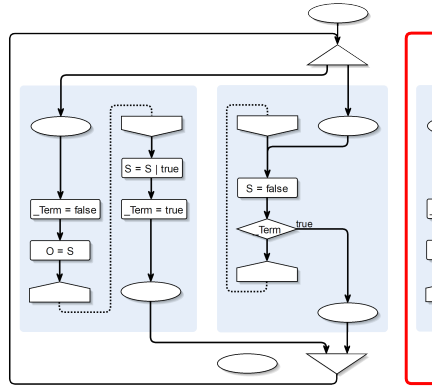
```
1 module schizo
2 output O;
3
4 loop
5   signal S in
6     present S
7       then
8       emit O
9     end;
10    pause;
11    emit S;
12  end;
13 end loop
14 end module
```

Esterel
[Tardieu & de Simone '04]

⇒

```
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 = 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 }
```

SCL (1st try)



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

A 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
```

Esterel

⇒

```
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
```

- 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
```

Esterel

⇒

```
1 module
2 schizo-cured2-strl
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
```

[Tardieu & de Simone '04]
(simplified)

- Duplicated loop body
- "Surface copy" transfers control immediately to "depth copy"
- Q: Complexity?
- A: Quadratic 😐

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
```

Esterel

⇒

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

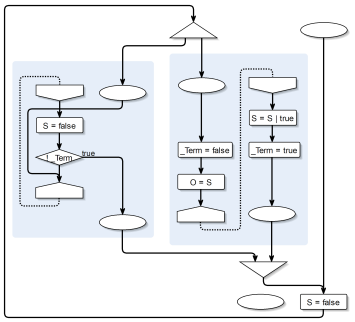
SCL

- "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?)

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    _Term = false;
10   fork
11     O = S;
12     pause;
13     S = 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

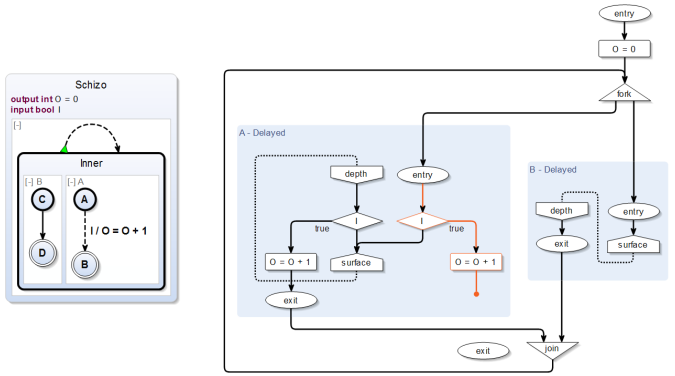


- Cycle now broken by delay
- Only the "depth initialization" of S creates a concurrent "initialize before update" scheduling dependence

Schizophrenic Parallel

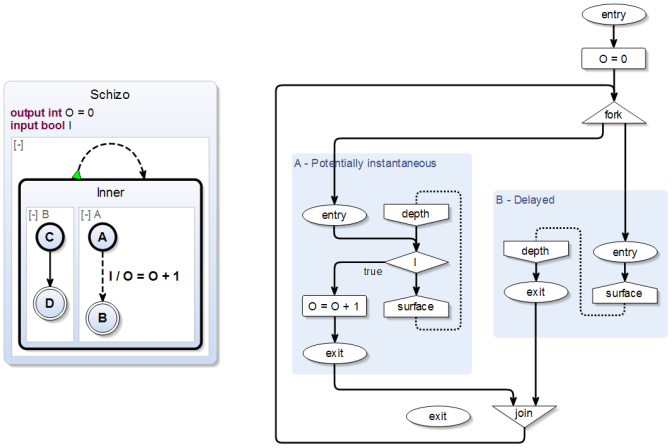
- Recall the equations for joining (two) threads:
  $g_{join} = (d_1 \vee m_1) \wedge (d_2 \vee m_2) \wedge (d_1 \vee d_2)$, where for each thread t_i it is "done" $d_i = g_{exit}$ and "empty" $m_i = \neg(g_{fork} \vee \bigvee_{depth \in t_i} g_{depth})$
- The join guard g_{join} corresponds to the K0 output of the Esterel circuit synthesis
- Since g_{join} depends on g_{fork}, the reincarnation of parallel leads to non-constructive circuits, just as with Esterel circuit synthesis
- We may apply same solution: divide join into surface join g_{s-join} and depth join g_{d-join}
- The logic for suface join and depth is the same, except that in depth join, we replace g_{fork} by false
- One can construct examples where both g_{s-join} and g_{d-join} are needed.
- If parallel is not instantaneous, only need g_{d-join}.
- In SCG, if thread terminates instantaneously in non-instantaneous parallel, we end in unjoined exit, visualized with small solid disk

Statement Reincarnation



- To remove cycle, must duplicate the part of the surface of the thread that might instantaneously terminate, i.e., nodes on instantaneous path from entry to exit
- The second increment of O leads to unjoined exit

Statement Reincarnation



- Consider I absent in initial tick, present in next tick
- Must then increment O twice

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 ($\rightarrow$ PRETSY), multi-clock, visualization, . . .

To Go Further

- J. Aguado, M. Mendler, R. von Hanxleden, I. Fuhrmann. *Grounding Synchronous Deterministic Concurrency in Sequential Programming*. In Proceedings of the 23rd European Symposium on Programming (ESOP'14), Grenoble, France, April 2014. `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/esop14.pdf`
- R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop. *Sequentially Constructive Concurrency – A Conservative Extension of the Synchronous Model of Computation*. ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design, July 2014, 13(4s). `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/tecs14.pdf`