Synchronous Languages—Lecture 12

Prof. Dr. Reinhard von Hanxleden

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

4 June 2020

Last compiled: June 2, 2020, 11:51 hrs

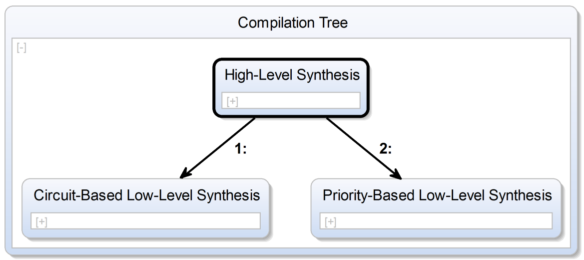
Code Generation for Sequential Constructiveness

The 5-Minute Review Session

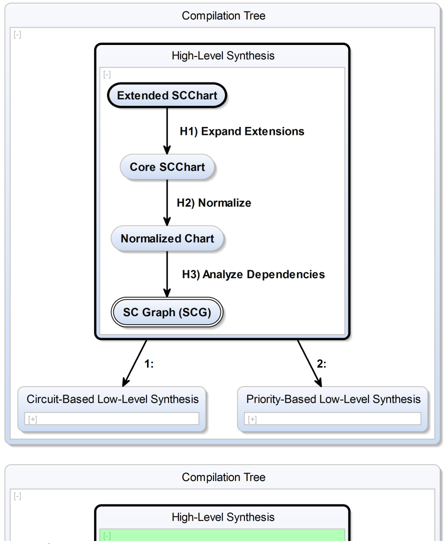
1. What are *Statecharts*? Who invented them?
2. What is the difference between *SyncCharts* and Statecharts?
3. How can we transform Esterel to SyncCharts? How about the other direction?
4. What are *SCCharts*? What is their motivation?
5. What are *Core SCCharts*?

Overview

Compilation — Overview

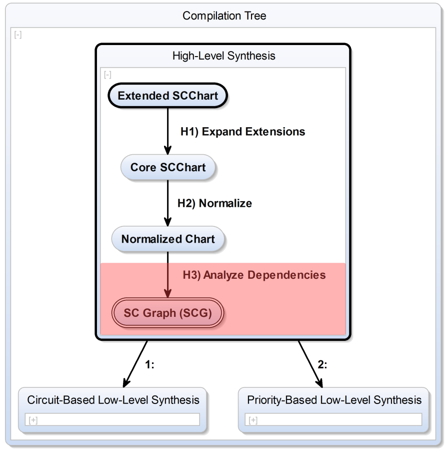


Compilation — High-Level Synthesis



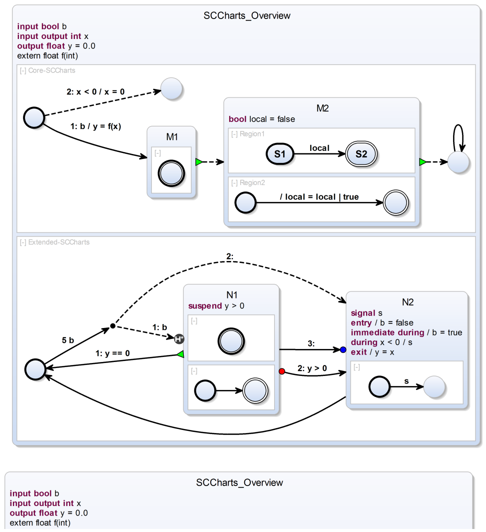
- Green:
  covered in previous lecture

Compilation — High-Level Synthesis



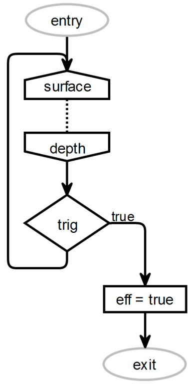
- Red:
  coming up now

(Recall) SCCharts - Core & Extended Features



Overview

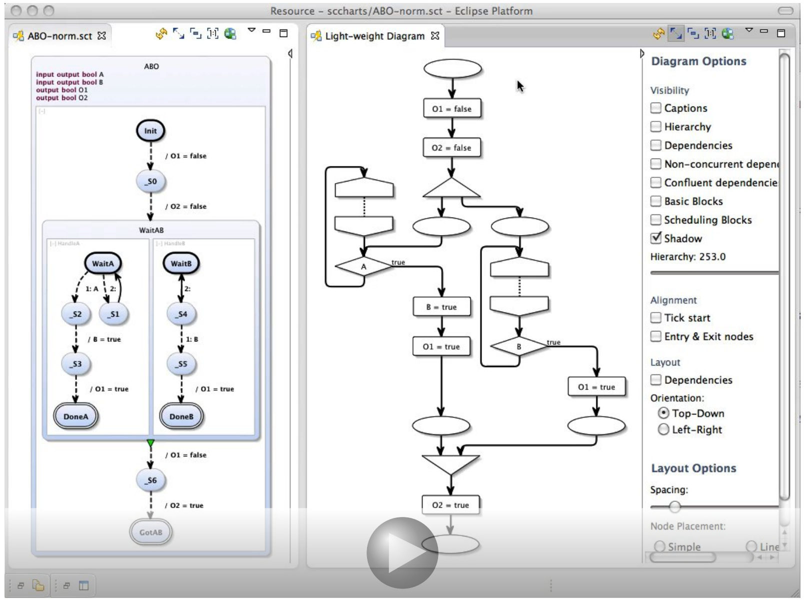
The SC Graph



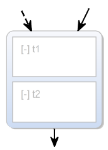
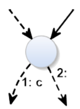
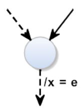
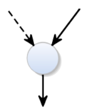
SC Graph:
Labeled graph $G = (S, E)$

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

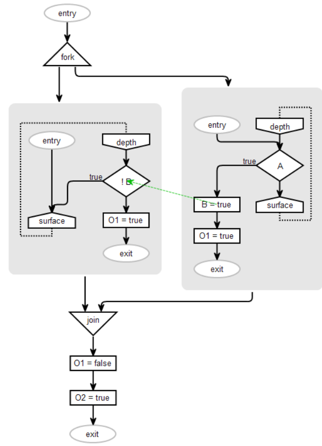
Example: Mapping ABO to SCG



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

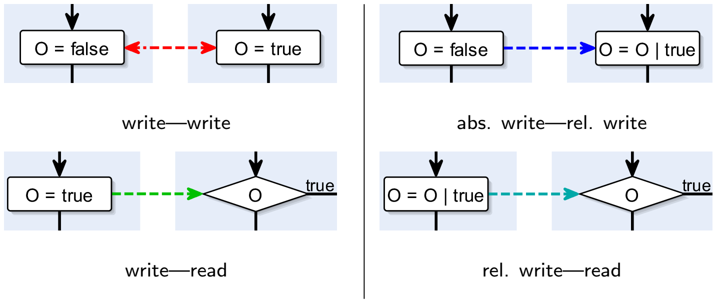
- they share a *least common ancestor fork* node.

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.)

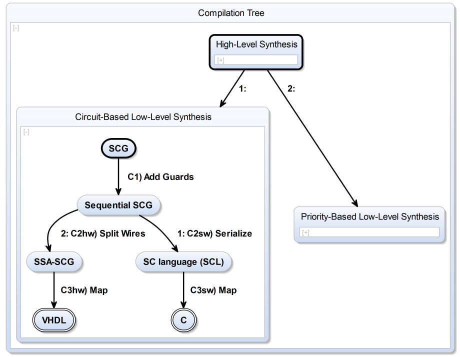
Overview

SCG Mapping & Dependency Analysis

Code Generation Approaches
- Circuit-based Approach
- Priority-based Approach
- Approach Comparison

Schizophrenia Revisited

Low-Level Synthesis I: The Circuit Approach

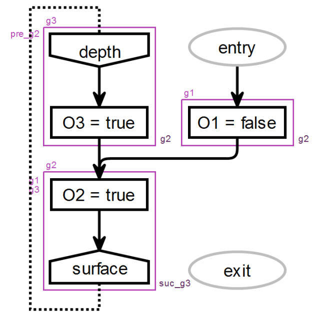


- Basic idea:
  Generate netlist
- Precondition:
  Acyclic SCG
  (with dependency edges, but without tick edges)
- Well-established approach for compiling SyncCharts/Esterel

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



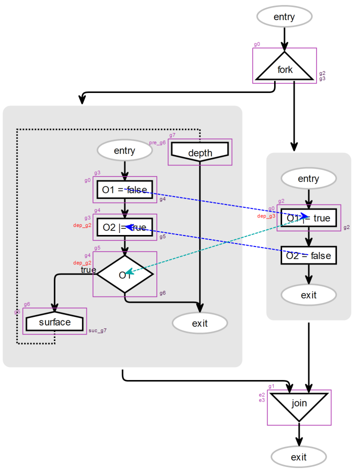
Basic Block:
A collection of SCG nodes / SCL statements

- that can be executed monolithically

Rules:

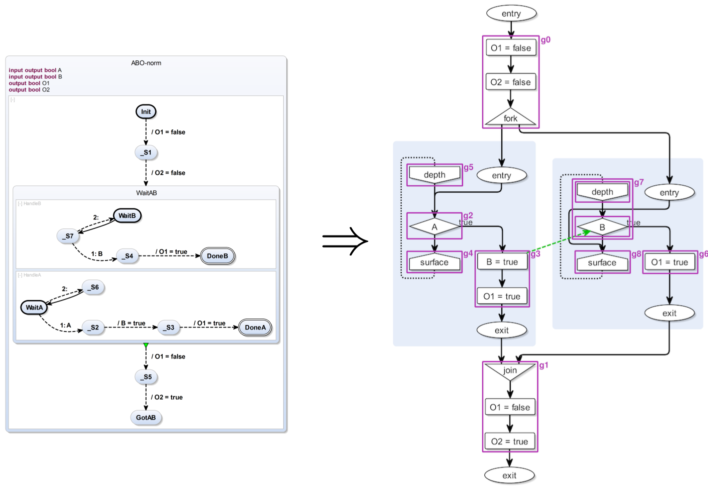
- Split at nodes with more than one incoming control flow edge
- Split at nodes with more than one outgoing control flow edge
- Split at tick edges
- Split after fork nodes and before join nodes
- **Each node can only be included in one basic block at any time**

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!

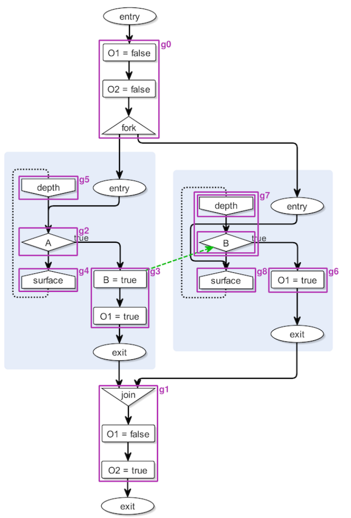
ABO SCG, With Dependencies and Scheduling Blocks



	Trigger (Conditional)	Effect (Assignment)	State (Delay)	Region (Thread)	Superstate (Concurrency)
Normalized Core SCCharts	1: c 2:	/x = e			[-] t1 [-] t2

	Trigger (Conditional)	Effect (Assignment)	State (Delay)	Region (Thread)	Superstate (Concurrency)
Normalized Core SCCharts	1: c 2:	/x = e			[-] t1 [-] t2
SCL	if (c) s_1 else s_2	$x = e$	pause	t	fork t_1 par t_2 join
SCG	c true	x = e	surface depth	entry exit	fork join

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

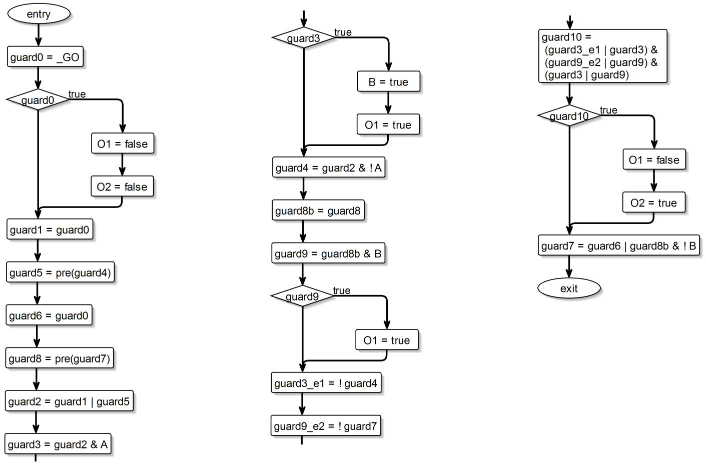


```
module ABO
input output bool
    A, B;
output bool O1,
    O2;
{
  O1 = false;
  O2 = false;
  fork
  HandleA:
    if (!A) {
      pause;
      goto HandleA
        ;
    };
    B = true;
    O1 = true;
  par
  HandleB:
    pause;
    if (!B) {
      goto HandleB
        ;
    };
    O1 = true;
  join;
  O1 = false;
  O2 = true;
}
```

⇒

```
input output bool A, B;
output bool O1, O2;
bool GO, g0, g1, ...
{
  g0 = GO;
  if g0 {
    O1 = false;
    O2 = false;
  };
  g5 = g4_pre;
  g7 = g8_pre;
  g2 = g0 || g5;
  g3 = g2 && A;
  if g3 {
    B = true;
    O1 = true;
  };
  g4 = g2 && ! A;
  g6 = g7 && B;
  if g6 {
    O1 = true;
  };
  g8 = g0 || (g7 && ! B);
  e2 = ! g4;
  e6 = ! g8;
  g1 = (g3 || e2) &&
     (g6 || e6) && (g3 ||
  if g1 {
    O1 = false;
    O2 = true;
  };
  g4_pre = g4;
  g8_pre = g8;
}
```

Sequential SCG — ABO



ABO SCL, Logic Synthesis (HW)

Difference to software

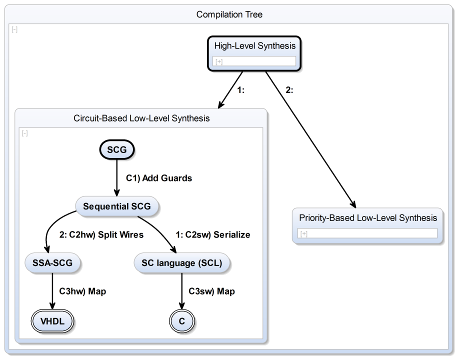
```
module ABO-seq
input output bool A, B;
output bool O1, O2;
bool GO, g0, g1, ...
{
  g0 = GO;
  if g0 {
    O1 = false;
    O2 = false;
  };
  g5 = g4_pre;
  g7 = g8_pre;
  g2 = g0 || g5;
  g3 = g2 && A;
  if g3 {
    B = true;
    O1 = true;
  };
  g4 = g2 && ! A;
  g6 = g7 && B;
  if g6 {
    O1 = true;
  };
  g8 = g0 || (g7 && ! B);
  e2 = ! g4;
  e6 = ! g8;
  g1 = (g3 || e2) &&
```

$\Longrightarrow$

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

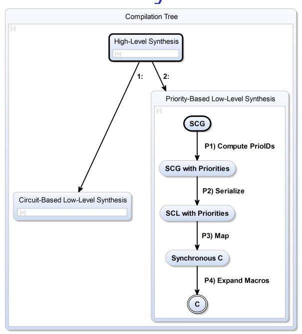
```
ARCHITECTURE behavior OF ABO IS
-- local signals definition, hidden
begin
-- main logic
g0 <= GO_local;
O1 <= false WHEN g0 ELSE O1_pre;
O2 <= false WHEN g0 ELSE O2_pre;
g5 <= g4_pre;
g7 <= g8_pre;
g2 <= g0 or g5;
g3 <= g2 and A_local;
B <= true WHEN g3 ELSE B_local;
O1_2 <= true WHEN g3 ELSE O1;
g4 <= g2 and not A_local;
g6 <= g7 and B;
O1_3 <= true WHEN g6 ELSE O1_2;
g8 <= g0 or (g7 and not B);
e2 <= not (g4);
e6 <= not (g8);
g1 <= (g3 or e2) and
```


(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

SCL$_P$ Macros I

```
// Declare Boolean type
typedef int bool;
#define false 0
#define true 1

// Generate "_L<line-number>" label
#define _concat_helper(a, b) a ## b
#define _concat(a, b) _concat_helper(a, b)
#define _label_    _concat(_L, __LINE__)

// Enable/disable threads with prioID p
#define _u2b(u)      (1 << u)
#define _enable(p) _enabled |= _u2b(p); _active |= _u2b(p)
#define _isEnabled(p) ((_enabled & _u2b(p)) != 0)
#define _disable(p) _enabled &= ~_u2b(p)
```

ABO SCL$_P$ I

```
int tick()
{
  tickstart(2);
  O1 = false;
  O2 = false;

  fork1(HandleB,
      1) {
   HandleA:
   if (!A) {
    pause;
    goto HandleA
         ;
   }
   B = true;
   O1 = true;

  } par {
```

⇒

```
int tick()
{
  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;
      } ;
  O1 = 0;
  O2 = 0;

  _pc[1] = &&HandleB; _enabled |= (1 <<
      1); _active |= (1 << 1); {
   HandleA:
   if (!A) {
    _pc[_cid] = &&_L94; goto _L_PAUSE;
        _L94:;
    goto HandleA;
   }
   B = 1;
   O1 = 1;

  } goto _L_TERM; {
```

SCL$_P$ Macros II

```
// Set current thread continuation
#define _setPC(p, label) _pc[p] = &&label

// Pause, resume at <label> or at pause
#define _pause(label) _setPC(_cid, label); goto _L_PAUSE
#define pause      _pause(_label_); _label_:

// Fork/join sibling thread with prioID p
#define fork1(label, p) _setPC(p, label); _enable(p);
#define join1(p)   _label_: if (_isEnabled(p)) { _pause(_label_); }

// Terminate thread at "par"
#define par        goto _L_TERM;

// Context switch (change prioID)
#define _prio(p)  _deactivate(_cid); _disable(_cid); _cid = p; \
 _enable(_cid); _setPC(_cid, _label_); goto _L_DISPATCH; _label_:
```

ABO SCL$_P$ II

```
   HandleB:
   pause;
   if (!B) {
    goto HandleB
         ;
   }
   O1 = true;
  } join1(2);

  O1 = false;
  O2 = true;
  tickreturn;
}
```

⇒

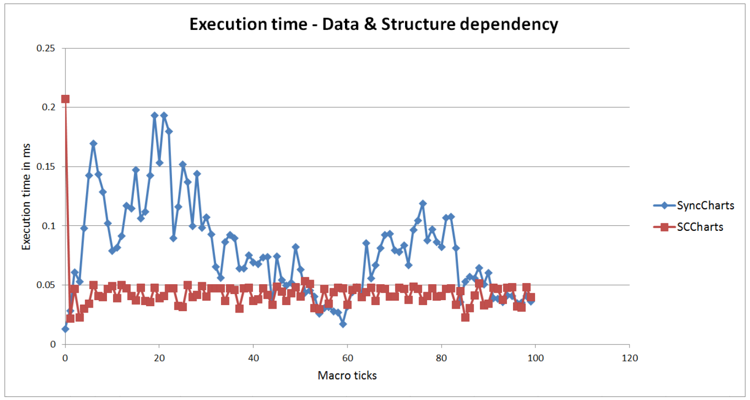
```
   HandleB:
   _pc[_cid] = &&_L103; goto _L_PAUSE;
       _L103:;
   if (!B) {
    goto HandleB;
   }
   O1 = 1;
  } _L108: if (((_enabled & (1 << 2)) !=
       0)) { _pc[_cid] = &&_L108; goto
      _L_PAUSE; };

  O1 = 0;
  O2 = 1;
  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];
}
```

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)	+	−

Overview

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?

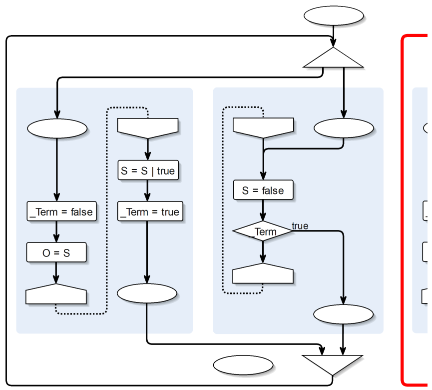
```
module schizo
output O;

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

Esterel [Tardieu & de Simone '04]

⇒

```
module schizo-conc
output bool O;
{
  while (true) {
    bool S, _Term;

    fork
      _Term = false;
      O = S;
      pause;
      S = S || true;
      _Term = true;
    par
      while (true) {
        S = false;
        if (_Term)
          break;
        pause;
      }
    join;
  }
}
```

SCL (1st try)



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

A Solution

```
module schizo
output O;

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

Esterel

⟹

```
module schizo-cured
output O;

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

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

A Better Solution

```
module schizo
output O;

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

Esterel

⟹

```
module
schizo-cured2-strl
output O;

loop
  % Surface
  signal S in
    present S then
      emit O
    end;
  end;

  % Depth
  signal S' in
    pause;
    emit S';
  end;
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

```
module schizo
output O;

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

Esterel

⟹

```
module schizo-cured
output bool O;
{
  while (true) {
    bool S, _Term;

    // Surf init
    S = false;
    _Term = false;
    fork
      O = S;
      pause;
      S = S || true;
      _Term = true;
    par
      do {
        pause;
        // Depth init
        S = false;
      } while (!_Term);
    join;
  }
}
```

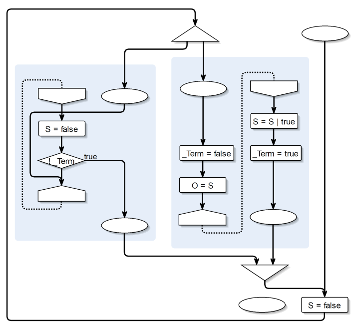
SCL

- "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

```
module schizo-cured
output bool O;
{
  while (true) {
    bool S, _Term;

    // Surf init
    S = false;
    _Term = false;
    fork
      O = S;
      pause;
      S = S || true;
      _Term = true;
    par
      do {
        pause;
        // Depth init
        S = false;
      } while (!_Term);
    join;
  }
}
```

SCL

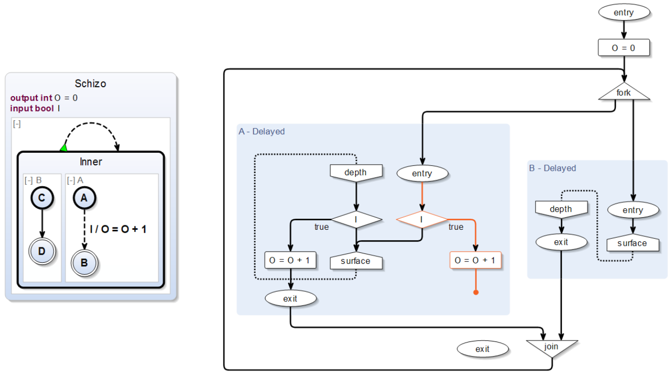


- Cycle now broken by delay
- Only the "depth initialization" of S creates a concurrent "initialize before update" scheduling dependence

Schizophrenic Parallel

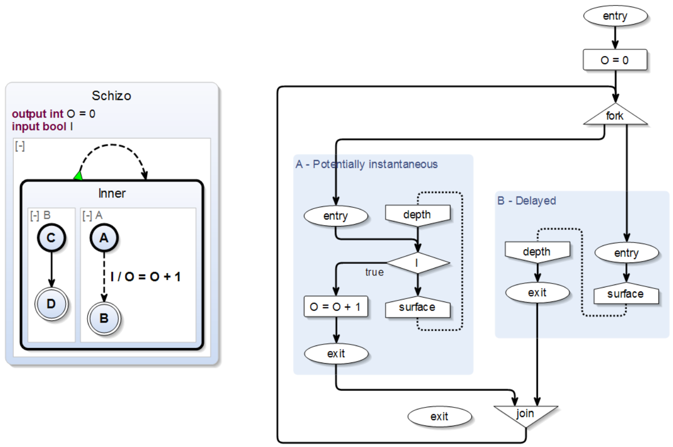
- Recall the equations for joining (two) threads: $g_{join} = (d_1 \vee m_1) \wedge (d_2 \vee m_2) \wedge (d_1 \vee d_2)$, where for each thread t_i it is "done" $d_i = g_{exit}$ and "empty" $m_i = \neg(g_{fork} \vee \bigvee_{depth \in t_i} g_{depth})$
- The join guard g_{join} corresponds to the K0 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 (→ 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`