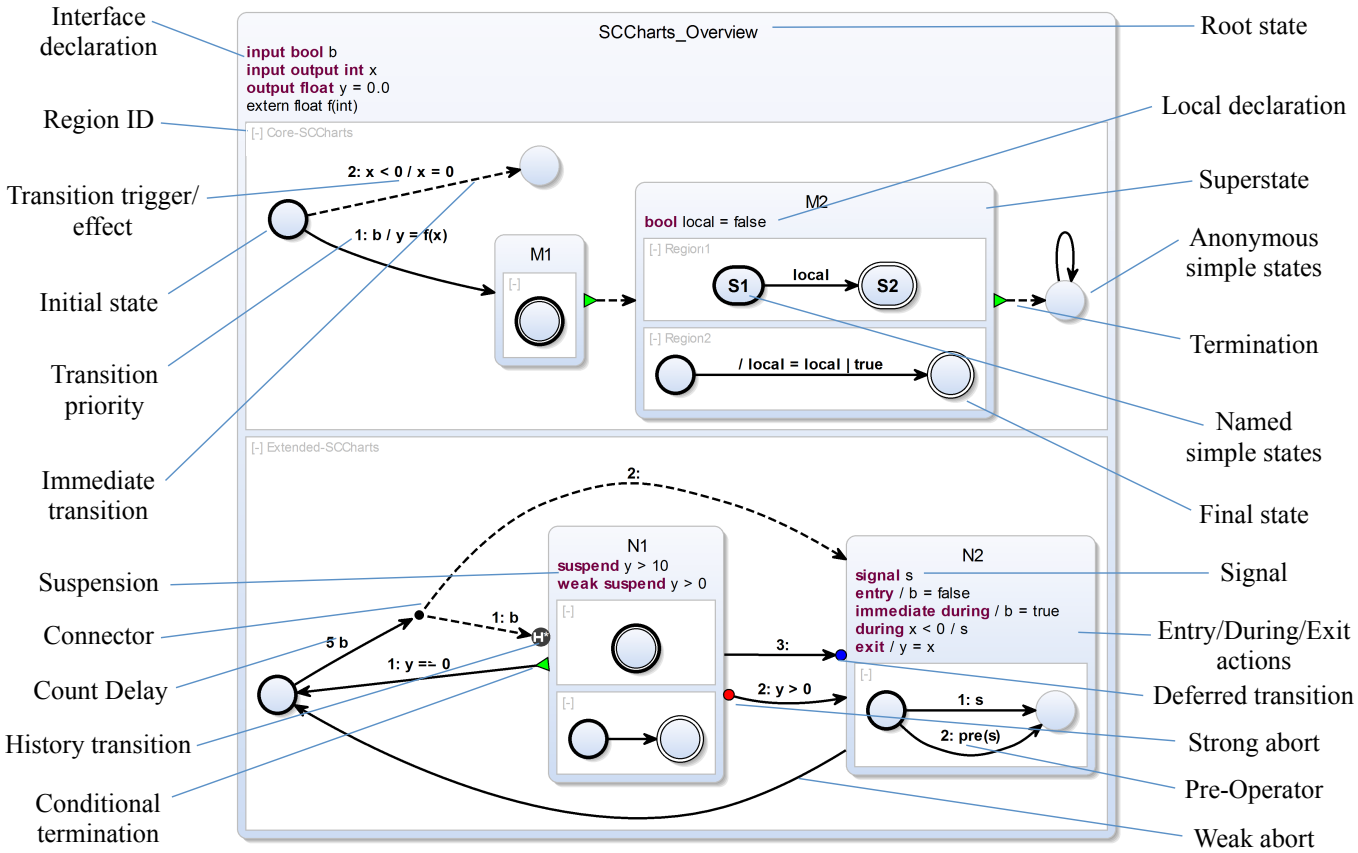
Compiling SCCharts to Hardware and Software

Reinhard von Hanxleden[1], Björn Duderstadt[1], Christian Motika[1], Steven Smyth[1], Michael Mendler[2], Joaquin Aguado[2], Stephen Mercer[3], and Owen O'Brien[3]

[1]Christian-Albrechts-Universität zu Kiel
[2]Bamberg University
[3]National Instruments

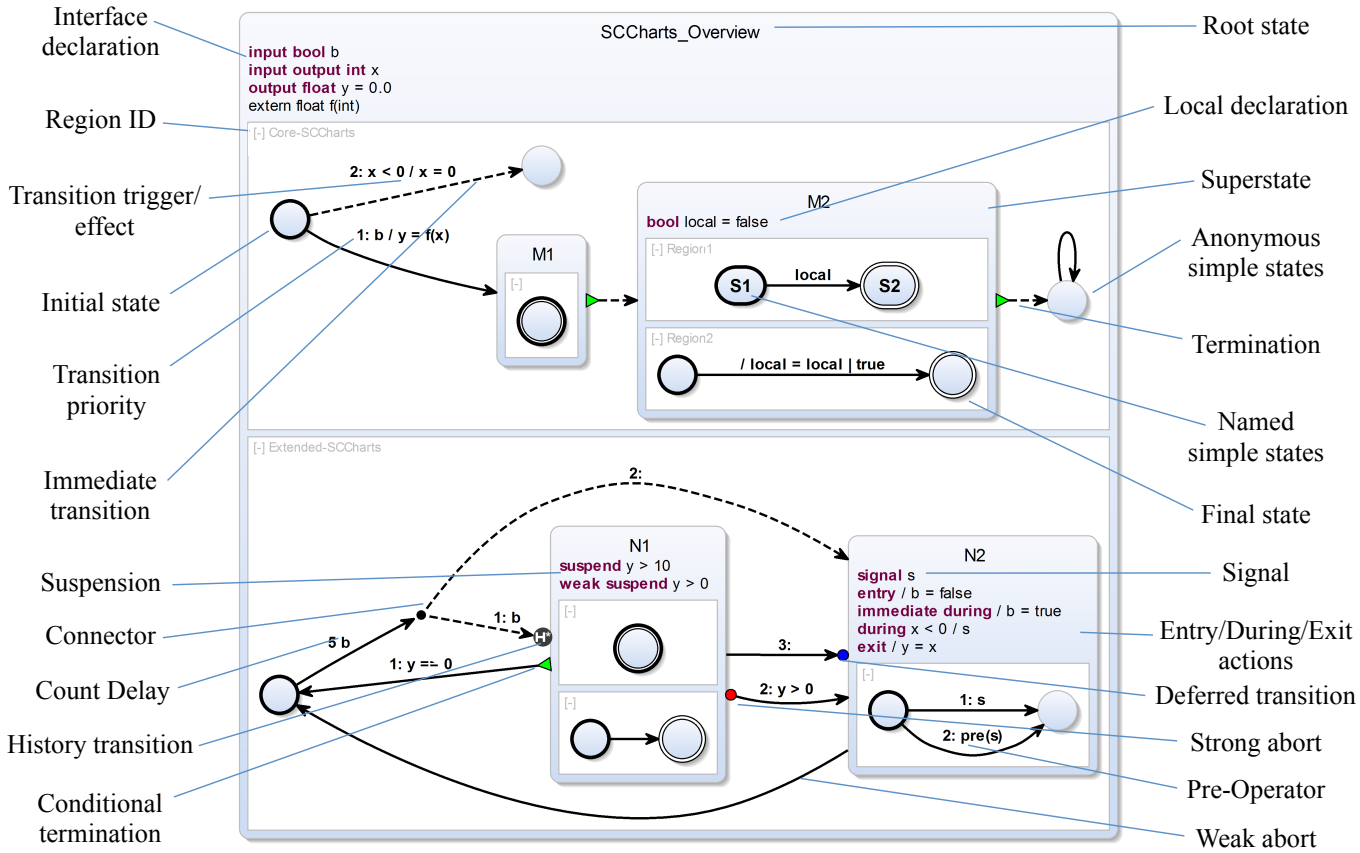
SYNCHRON 2013, Schloss Dagstuhl

SCCharts Overview



SCCharts SyncCharts + sequential constructiveness + extensions (e.g., from SCADE, Quartz)

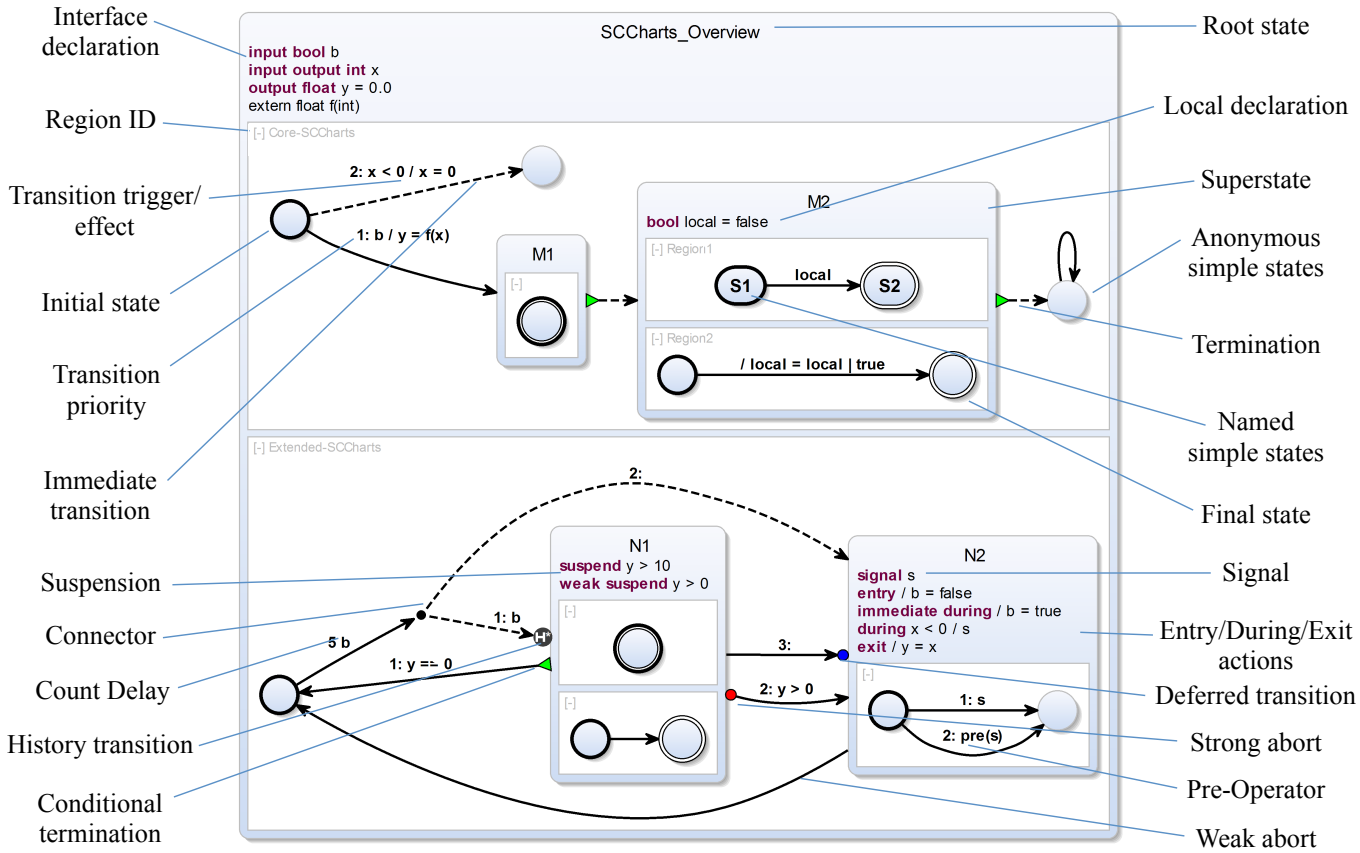
SCCharts Overview



SCCharts SyncCharts + sequential constructiveness + extensions (e.g., from SCADE, Quartz)

= Core SCCharts base language (*upper region*)

SCCharts Overview



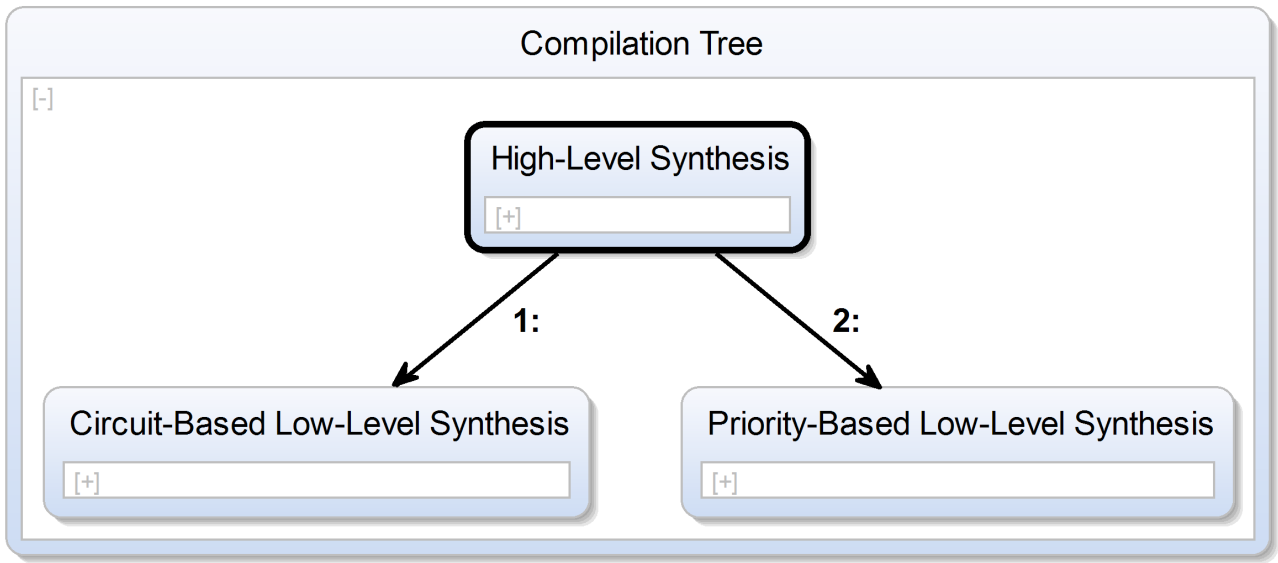
SCCharts SyncCharts + sequential constructiveness + extensions (e.g., from SCADE, Quartz)

= Core SCCharts base language (*upper region*)

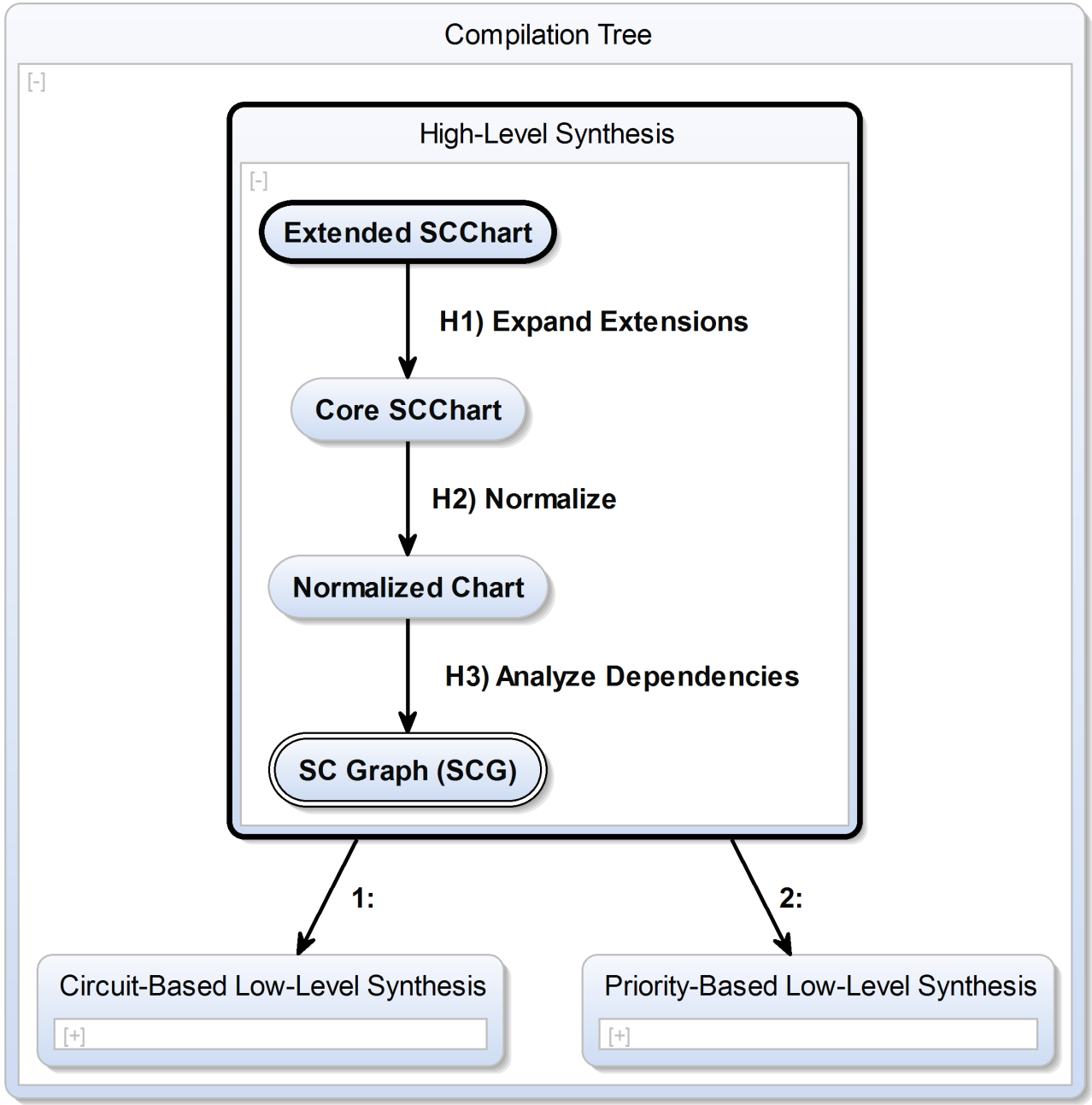
+ Extended SCCharts syntactic sugar (*lower region*)

Compilation — The Big Picture

Compilation — The Big Picture



High-Level Synthesis



High-Level Synthesis Step 1: Expand Extensions

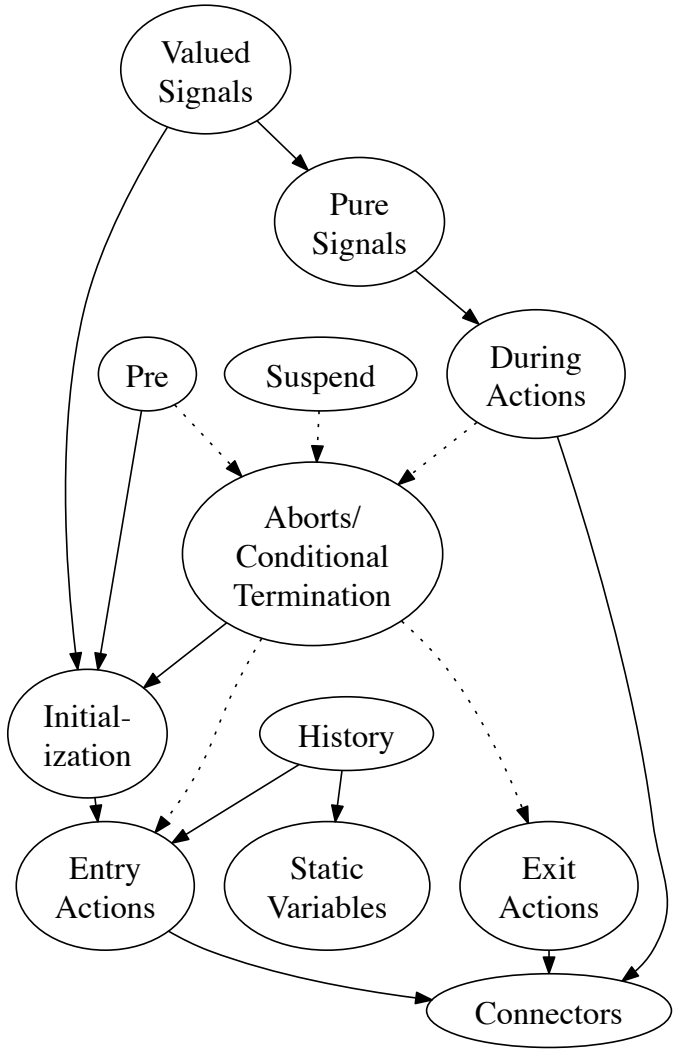
- Each extended SCChart feature f defined in terms of some M2M transformation T_f

High-Level Synthesis Step 1: Expand Extensions

- Each extended SCChart feature f defined in terms of some M2M transformation T_f
- T_f produces an SCChart not containing f but possibly containing features $\in Prod_f$
- T_f can preserve features $Handle_f$

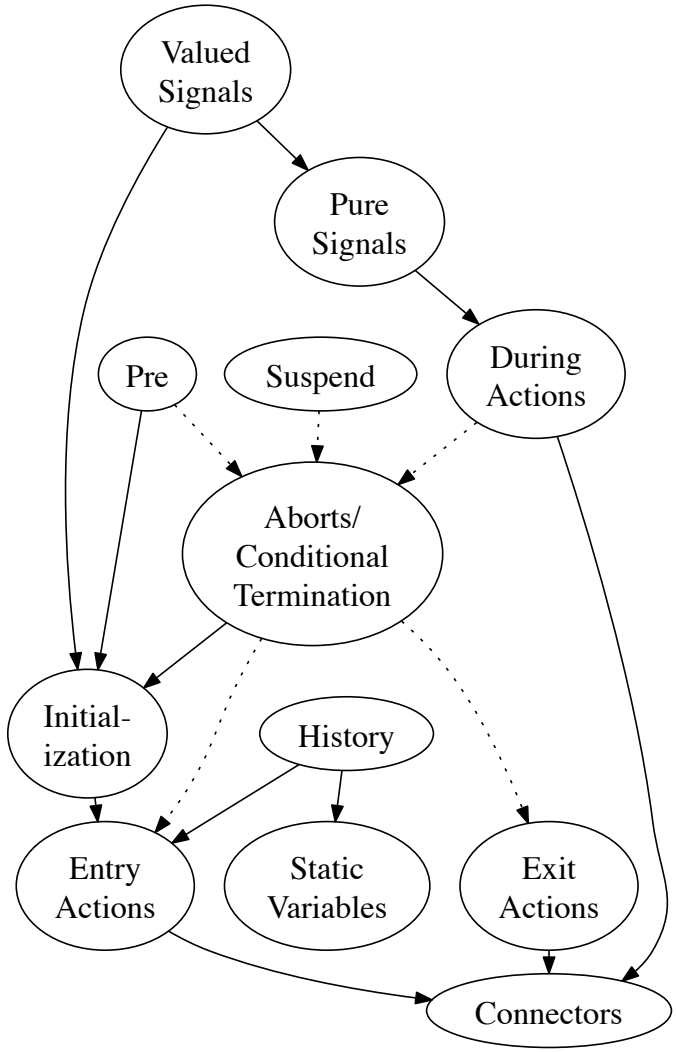
High-Level Synthesis Step 1: Expand Extensions

- Each extended SCChart feature f defined in terms of some M2M transformation T_f
- T_f produces an SCChart not containing f but possibly containing features $\in Prod_f$
- T_f can preserve features $Handle_f$
- Must perform transformation T_f before T_g if
  - $g \in Prod_f$ (solid lines), or
  - $f \notin Handle_g$ (dashed lines)



High-Level Synthesis Step 1: Expand Extensions

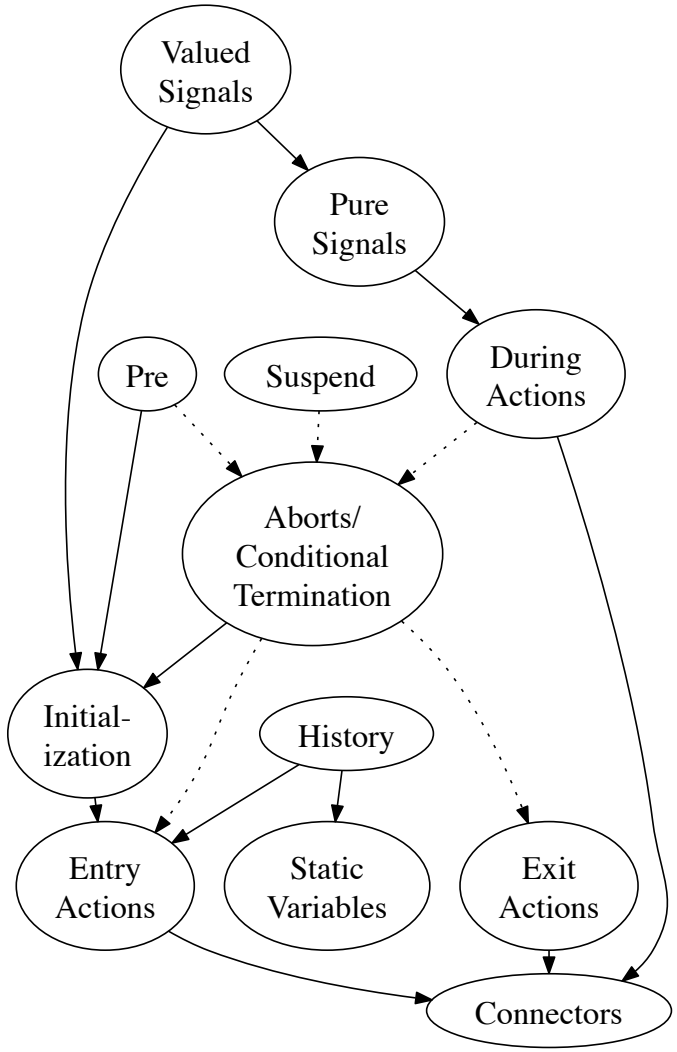
- Each extended SCChart feature f defined in terms of some M2M transformation T_f
- T_f produces an SCChart not containing f but possibly containing features $\in Prod_f$
- T_f can preserve features $Handle_f$
- Must perform transformation T_f before T_g if
  - $g \in Prod_f$ (solid lines), or
  - $f \notin Handle_g$ (dashed lines)



- Transformation interdependencies form partial order $\rightarrow$ single pass transformation sequence

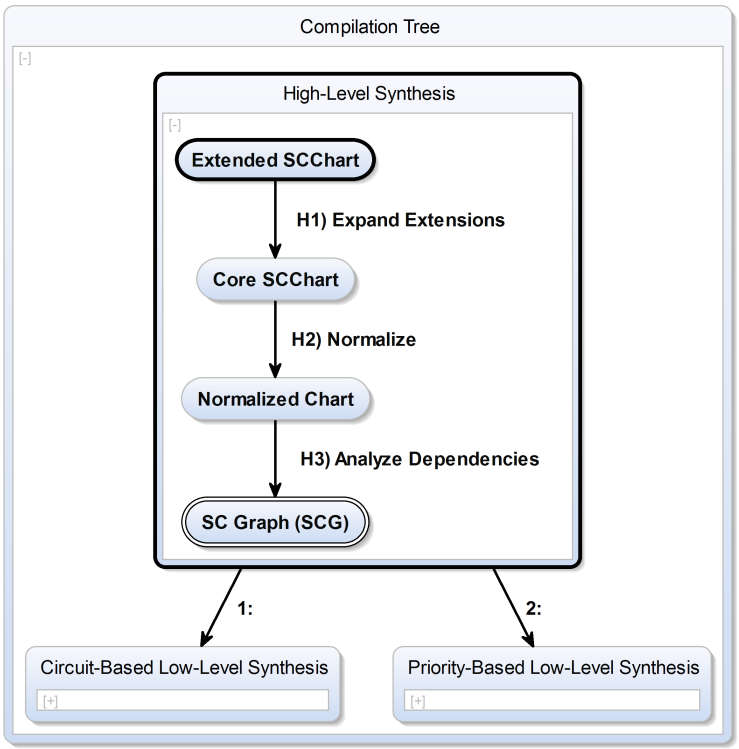
High-Level Synthesis Step 1: Expand Extensions

- Each extended SCChart feature f defined in terms of some M2M transformation T_f
- T_f produces an SCChart not containing f but possibly containing features $\in Prod_f$
- T_f can preserve features $Handle_f$
- Must perform transformation T_f before T_g if
  - $g \in Prod_f$ (solid lines), or
  - $f \notin Handle_g$ (dashed lines)



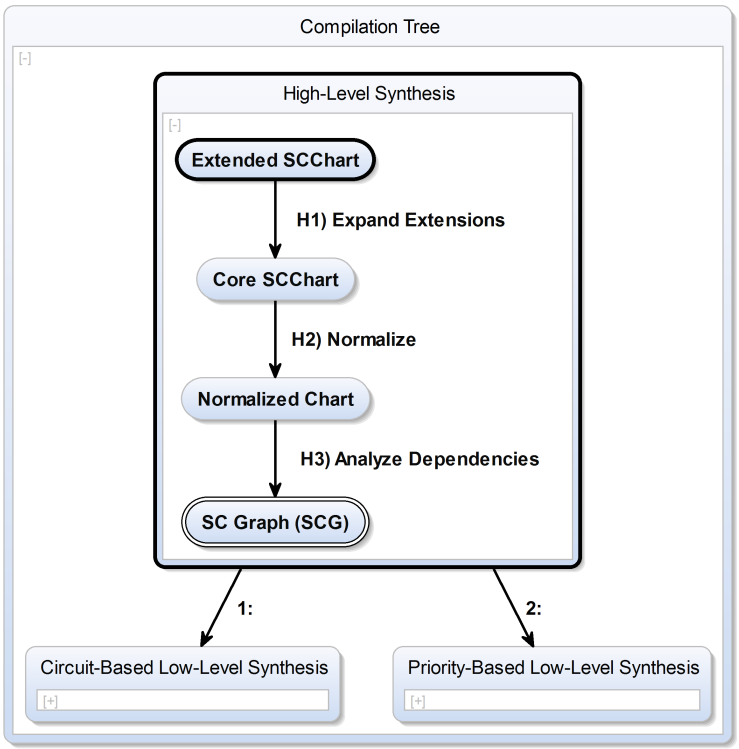
- Transformation interdependencies form partial order $\rightarrow$ single pass transformation sequence
- Can prune down language without breaking rest

High-Level Synthesis Step 2: Normalize SCCharts



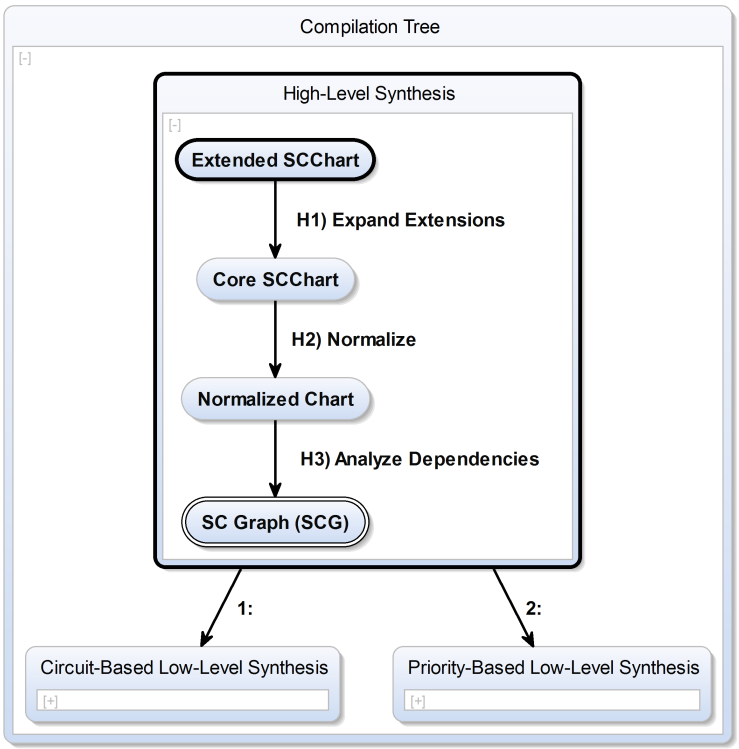
1. For superstates without termination transition: add termination transition to new, non-final state

High-Level Synthesis Step 2: Normalize SCCharts



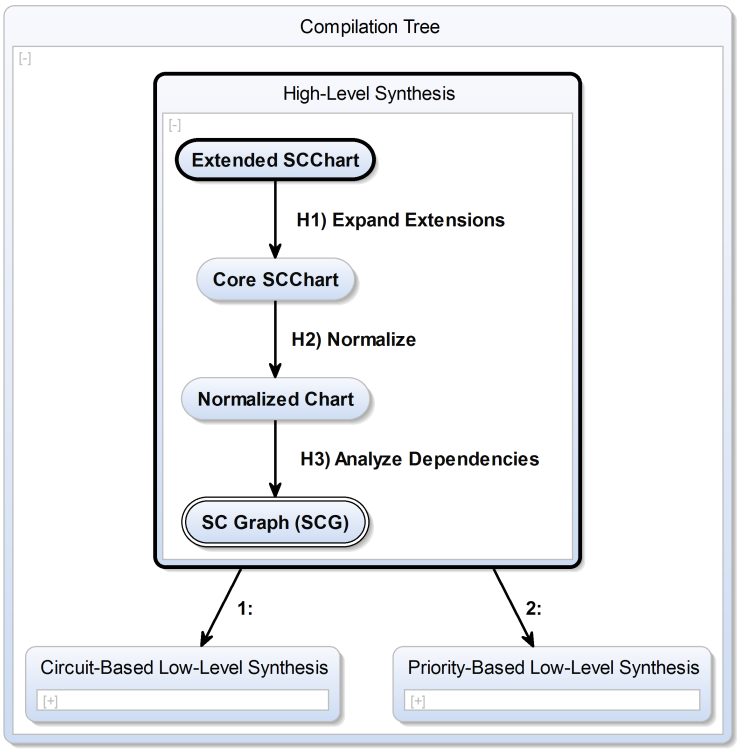
1. For superstates without termination transition:
   add termination transition to new, non-final state
2. For non-final states without outgoing transitions:
   add a delayed self transition (= halt)

High-Level Synthesis Step 2: Normalize SCCharts



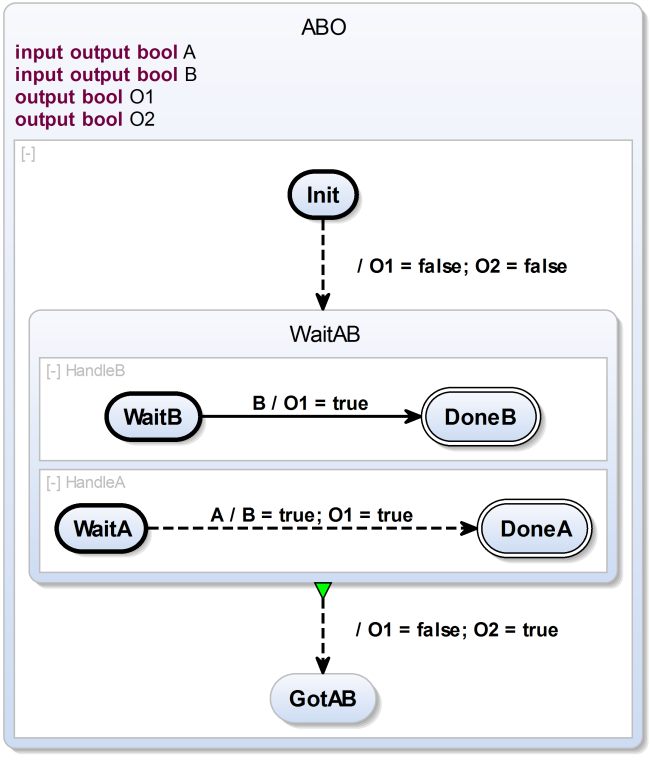
1. For superstates without termination transition:
   add termination transition to new, non-final state
2. For non-final states without outgoing transitions:
   add a delayed self transition (= halt)
3. Separate/split actions

High-Level Synthesis Step 2: Normalize SCCharts

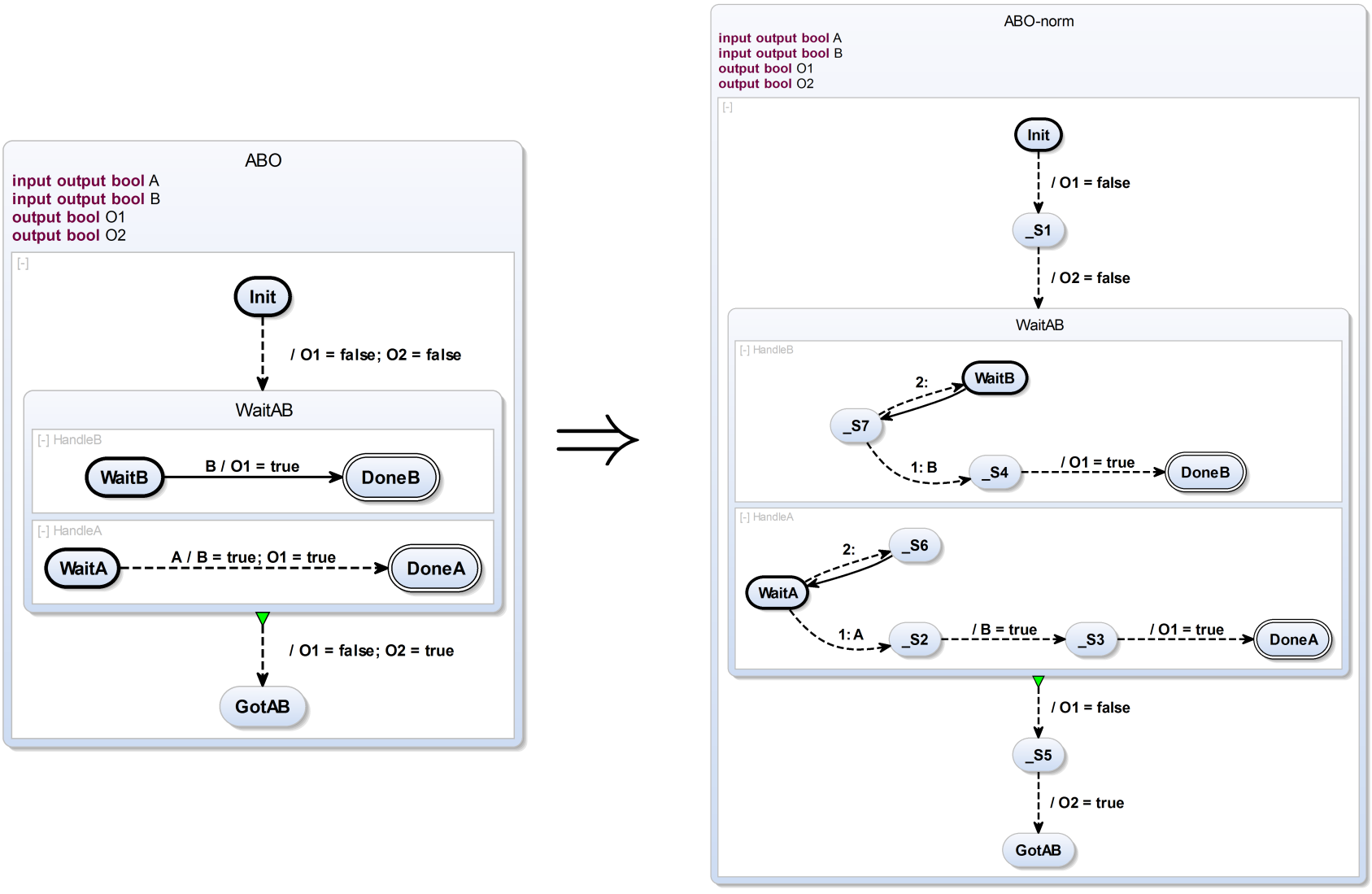


1. For superstates without termination transition:
   add termination transition to new, non-final state
2. For non-final states without outgoing transitions:
   add a delayed self transition (= halt)
3. Separate/split actions
4. Split triggers

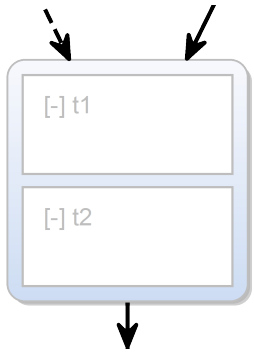
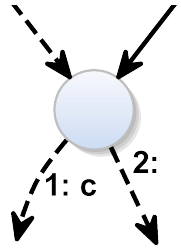
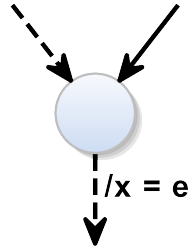
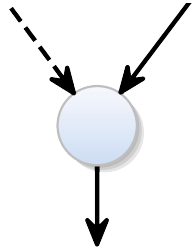
Example: Normalizing ABO



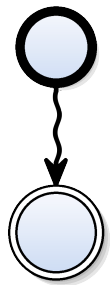
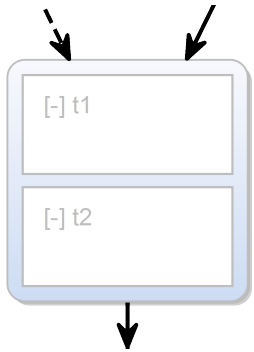
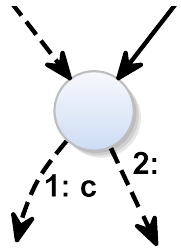
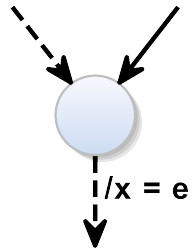
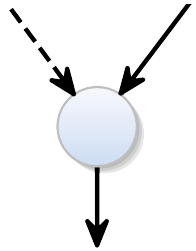
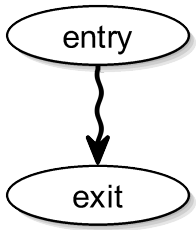
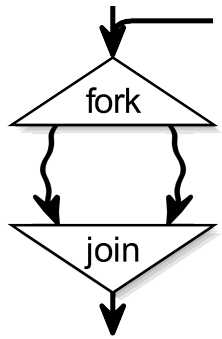
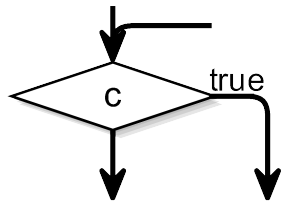
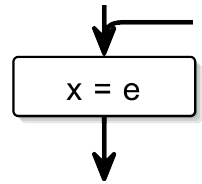
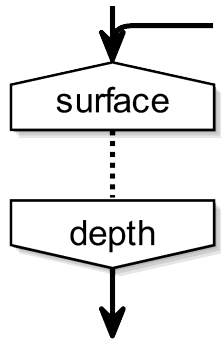
Example: Normalizing ABO



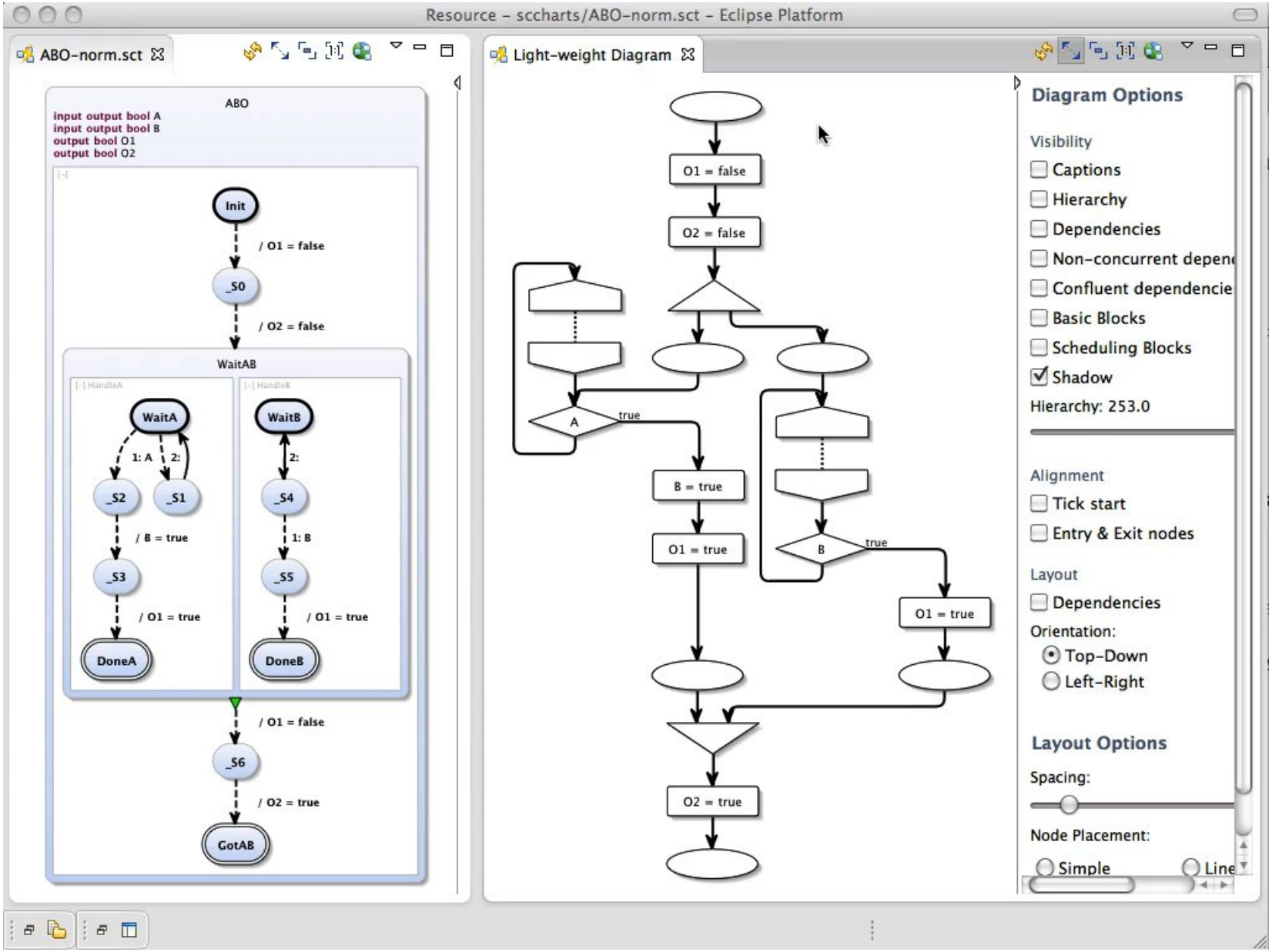
High-Level Step 3: Map to SC Graph

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

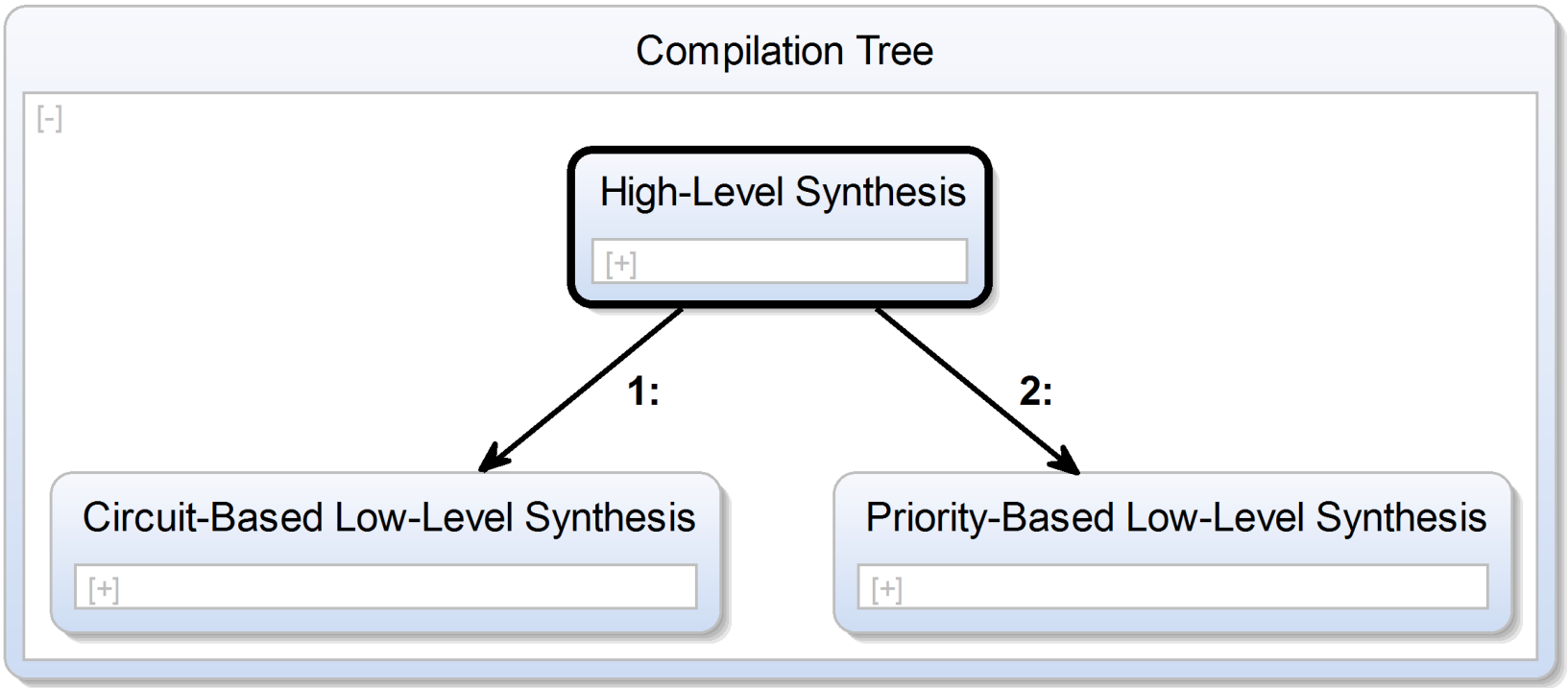
High-Level Step 3: Map to SC Graph

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

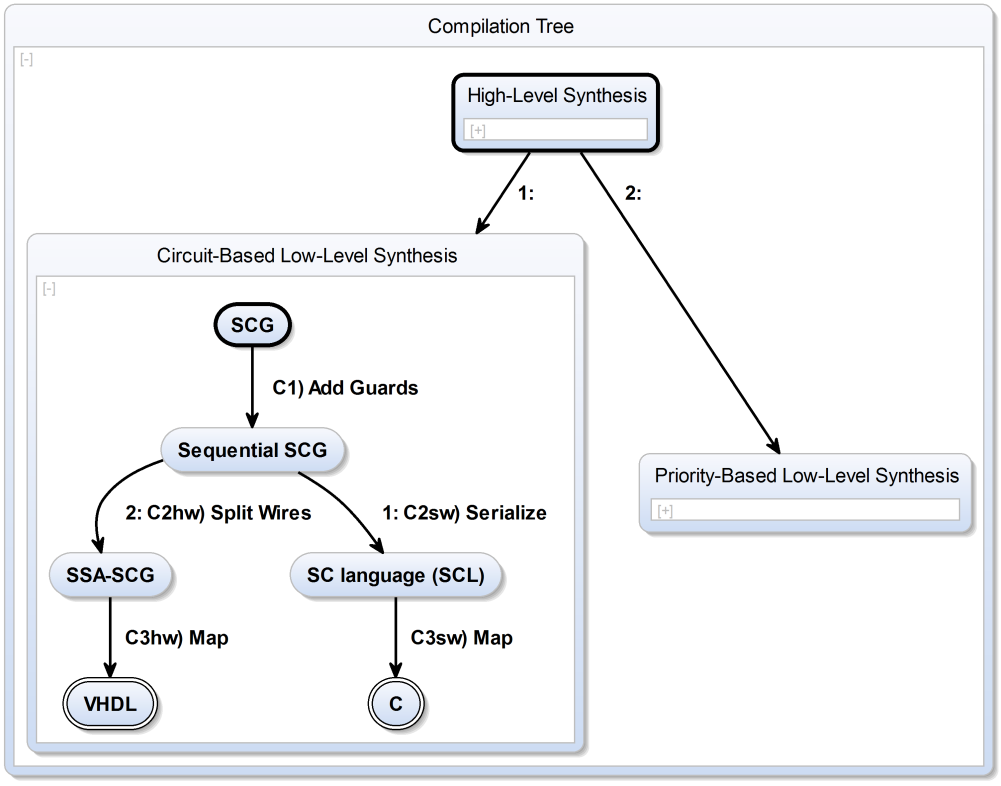
Example: Mapping ABO to SCG



(Recall) The Big Picture

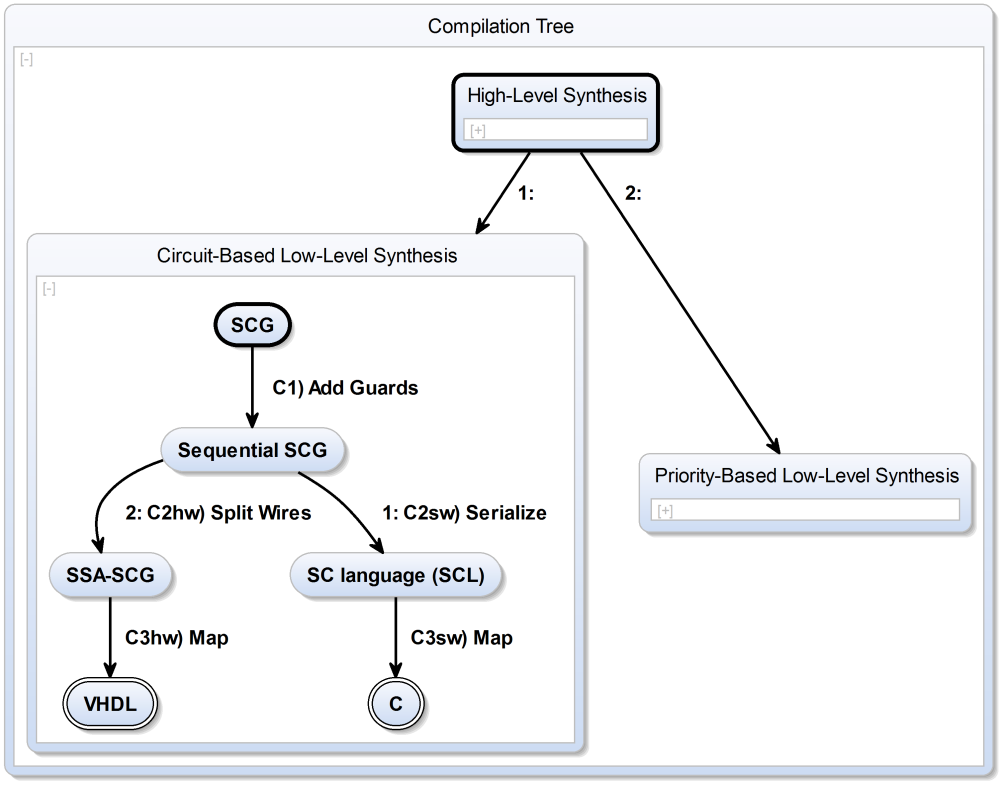


Low-Level Synthesis I: The Circuit Approach



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

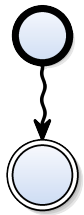
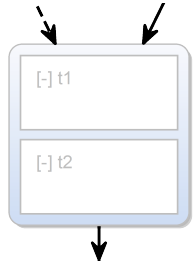
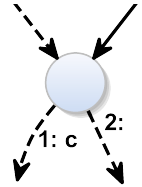
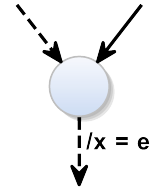
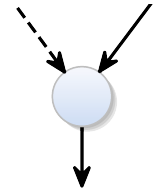
Low-Level Synthesis I: The Circuit Approach

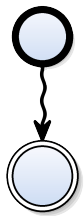
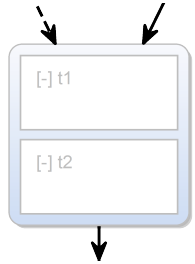
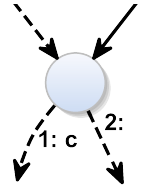
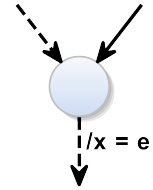
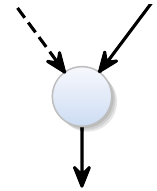
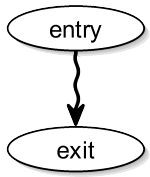
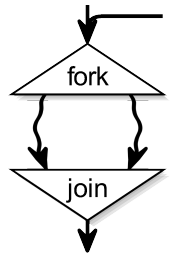
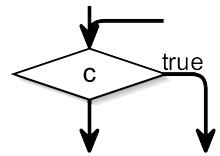
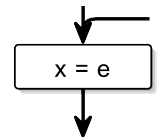
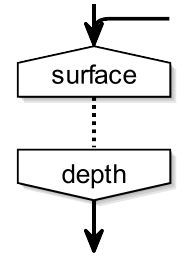


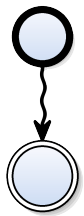
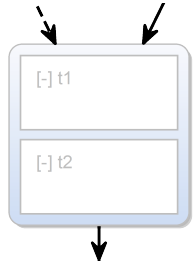
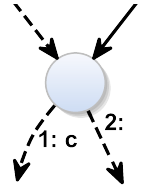
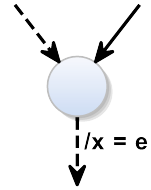
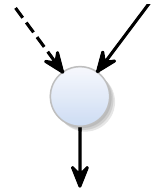
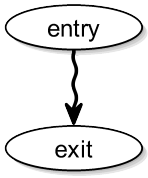
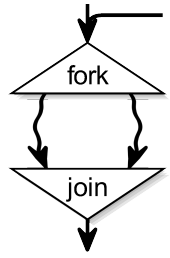
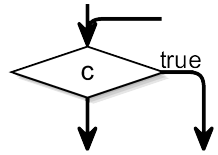
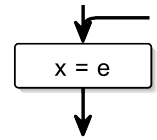
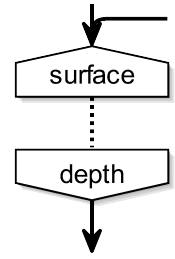
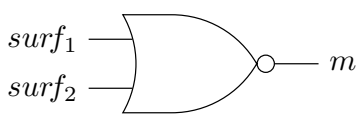
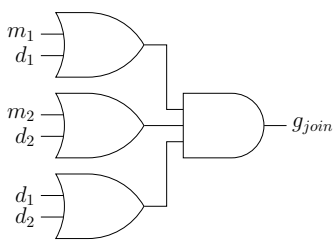
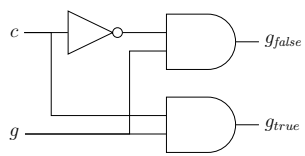
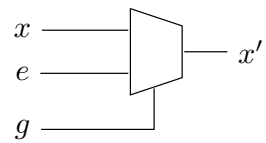
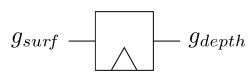
- Basic idea:
  Generate netlist
- Precondition:
  Acyclic SCG
  (with dependency edges, but without tick edges)
- Well-established approach for compiling SyncCharts/Esterel

Differences to Esterel circuit semantics [Berry '02]

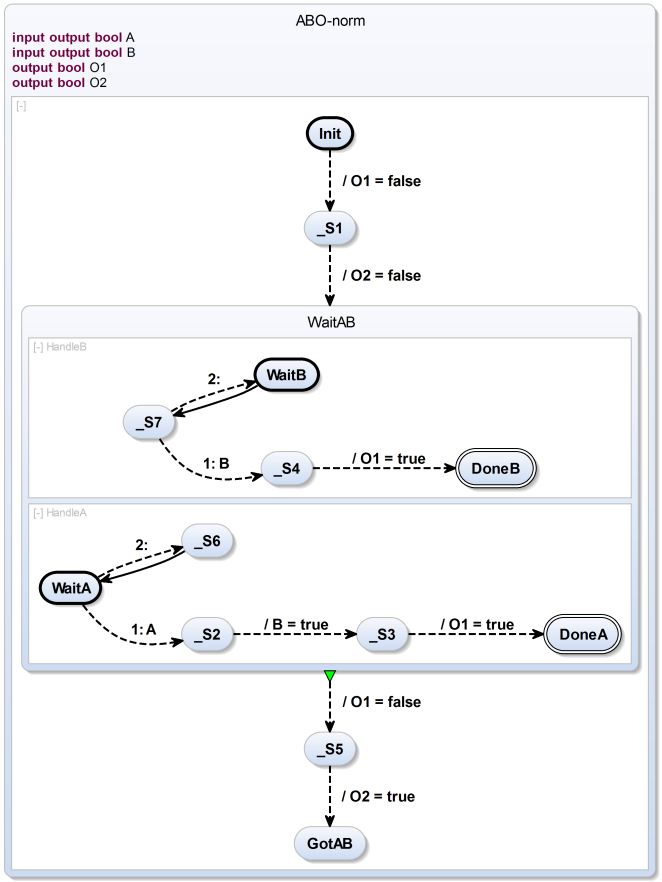
1. Simpler translation rules, as aborts/traps/suspensions already transformed away during high-level synthesis
2. SC MoC permits sequential assignments

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

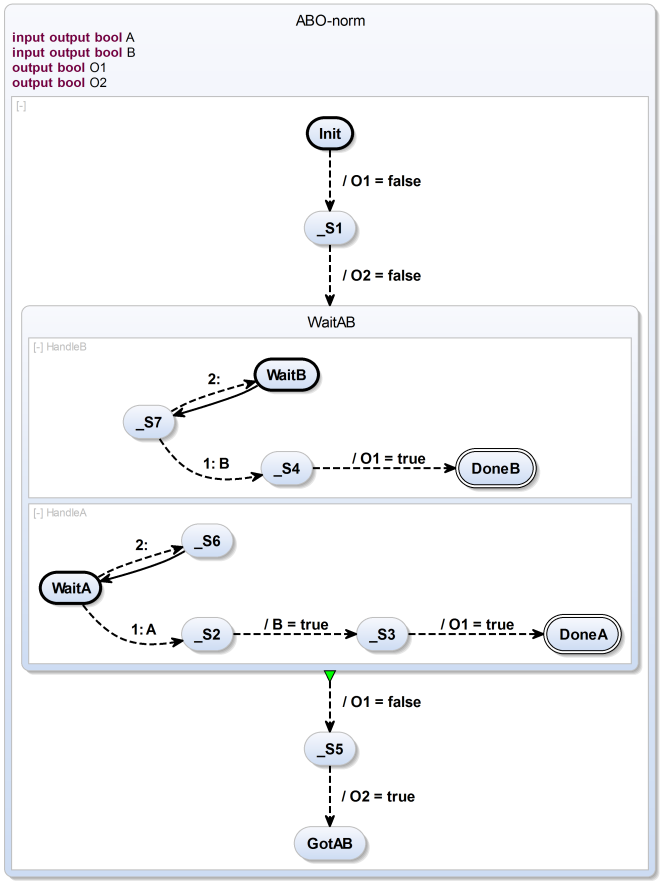
	Region (Thread)	Superstate (Concurrency)	Trigger (Conditional)	Effect (Assignment)	State (Delay)
SCCharts					
SCG					
SCL	t	fork t_1 par t_2 join	if (c) s_1 else s_2	$x = e$	pause

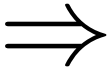
	Region (Thread)	Superstate (Concurrency)	Trigger (Conditional)	Effect (Assignment)	State (Delay)
SCCharts					
SCG					
SCL	t	fork t_1 par t_2 join	if (c) s_1 else s_2	$x = e$	pause
Data-Flow Code	$d = g_{exit}$ $m = \neg \bigvee_{surf \in t} g_{surf}$	$g_{join} = (d_1 \vee m_1) \wedge (d_2 \vee m_2) \wedge (d_1 \vee d_2)$	$g = \bigvee g_{in}$ $g_{true} = g \wedge c$ $g_{false} = g \wedge \neg c$	$g = \bigvee g_{in}$ $x' = g \; ? \; e : x$	$g_{depth} = \text{pre}(g_{surf})$
Circuits					

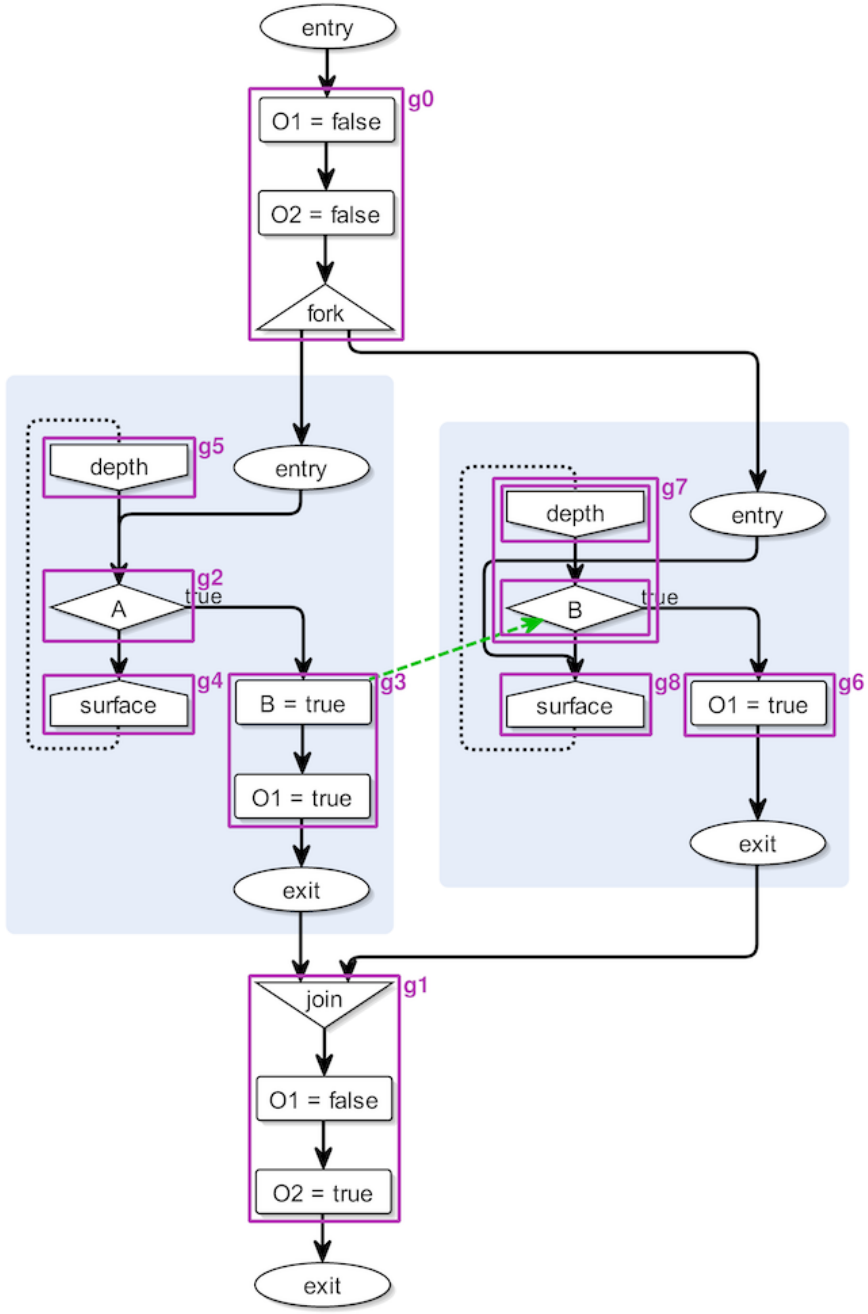
ABO SCG, With Dependencies and Scheduling Blocks



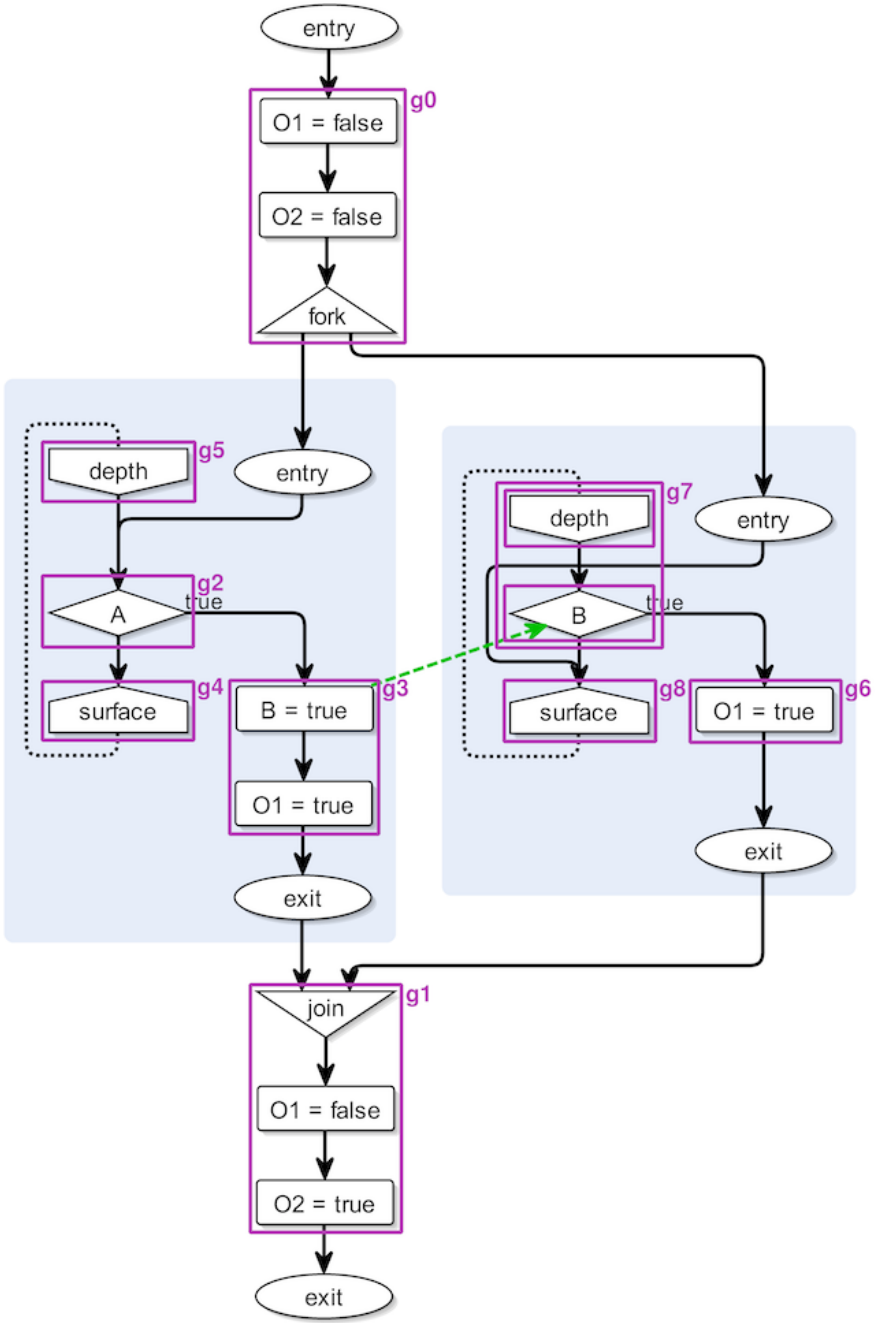
ABO SCG, With Dependencies and Scheduling Blocks



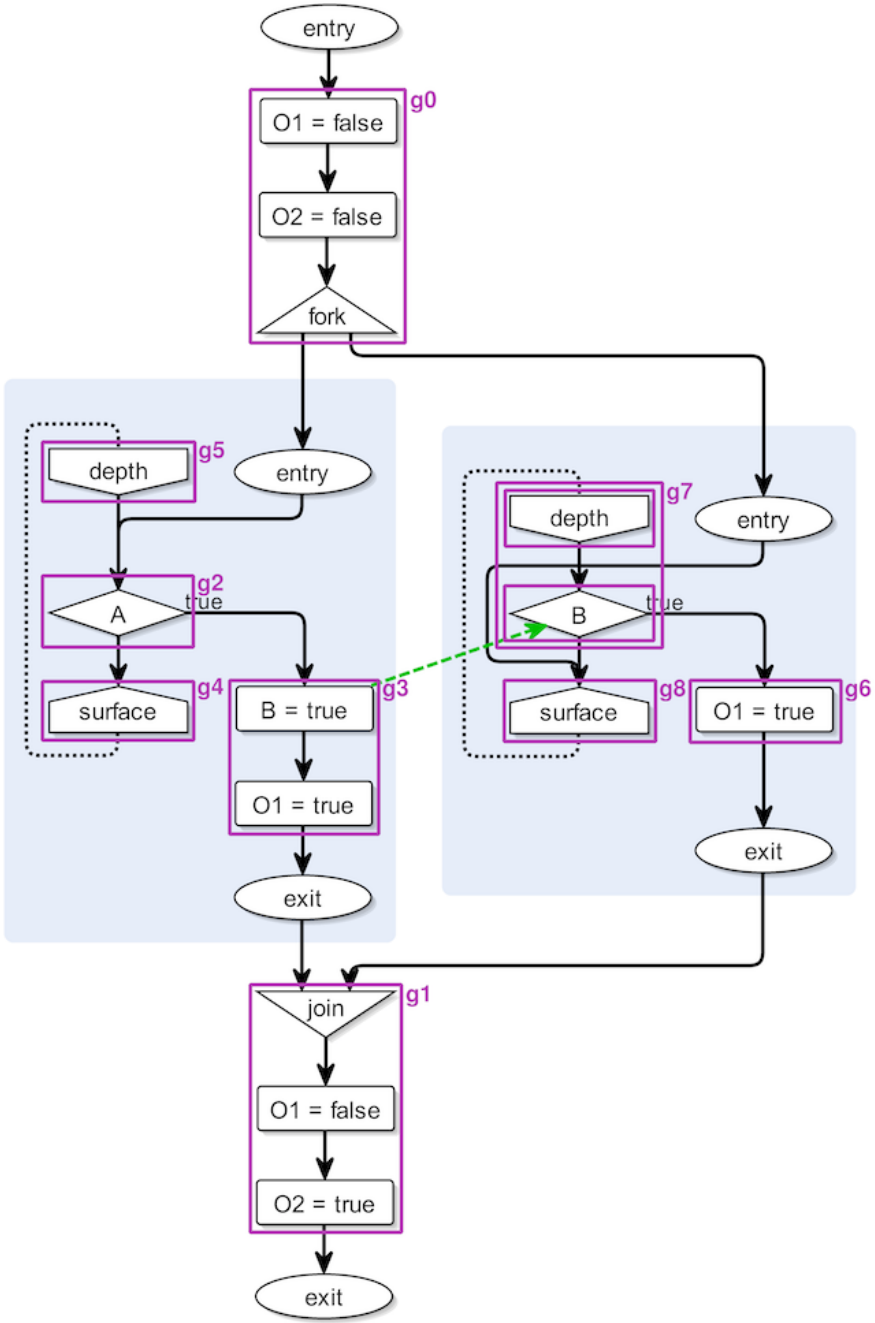




ABO SCL, Tick Function Synthesis (SW)

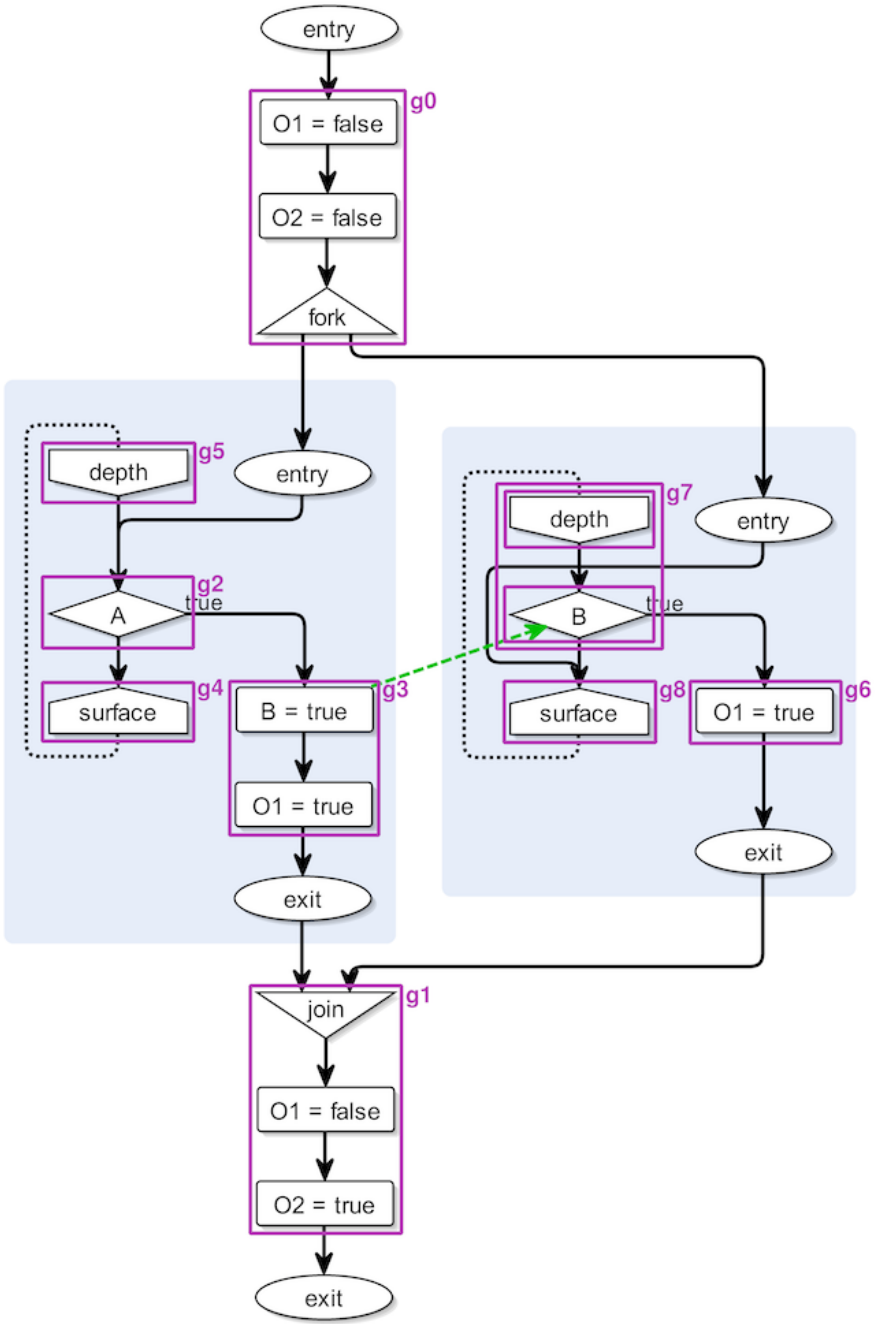


ABO SCL, Tick Function Synthesis (SW)




```
1  module ABO
2  input output bool A, B;
3  output bool O1, O2;
4  {
5    O1 = false;
6    O2 = false;
7
8    fork
9      HandleA:
10     if (!A) {
11       pause;
12       goto HandleA;
13     };
14     B = true;
15     O1 = true;
16
17   par
18     HandleB:
19     pause;
20     if (!B) {
21       goto HandleB;
22     };
23     O1 = true;
24
25   join;
26
27   O1 = false;
28   O2 = true;
29 }
```

ABO SCL, Tick Function Synthesis (SW)

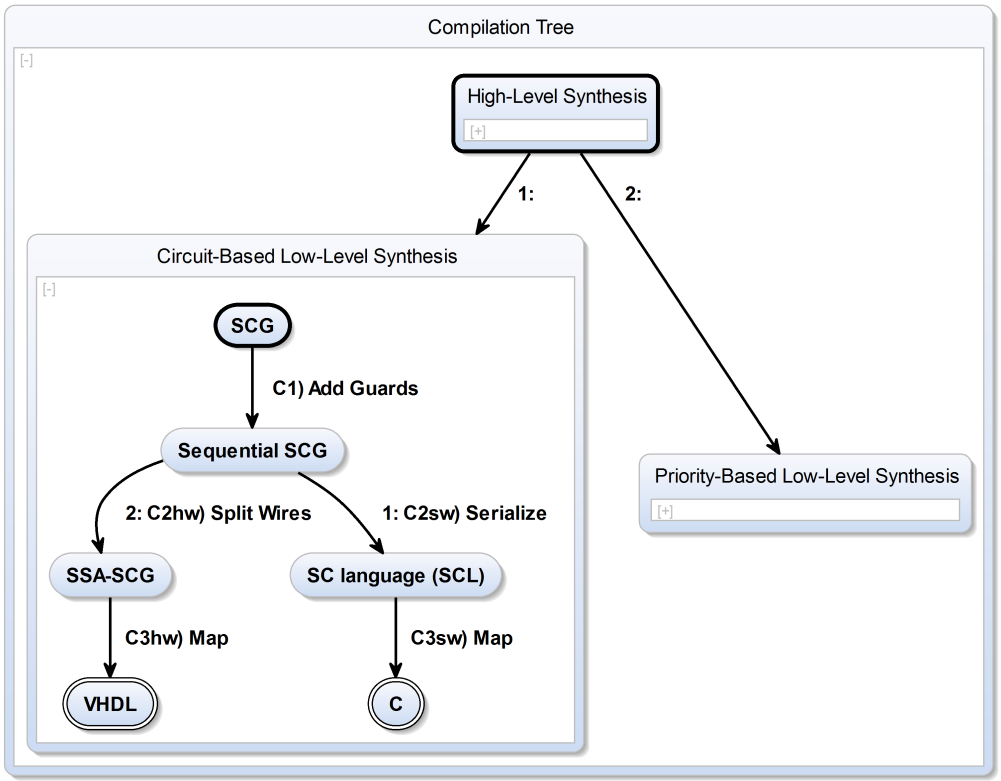



```
1  module ABO
2  input output bool A, B;
3  output bool O1, O2;
4  {
5    O1 = false;
6    O2 = false;
7
8    fork
9      HandleA:
10     if (!A) {
11       pause;
12       goto HandleA;
13     };
14     B = true;
15     O1 = true;
16
17   par
18     HandleB:
19     pause;
20     if (!B) {
21       goto HandleB;
22     };
23     O1 = true;
24
25   join;
26
27   O1 = false;
28   O2 = true;
29 }
```

⇒

```
1  module ABO—seq
2  input output bool A, B;
3  output bool O1, O2;
4  bool GO, g0, g1, e2, e6, g2,
5    g3, g4, g5, g6, g7, g8;
6  bool g4_pre, g8_pre;
7  {
8    g0 = GO;
9    if g0 {
10     O1 = false;
11     O2 = false;
12   };
13   g5 = g4_pre;
14   g7 = g8_pre;
15   g2 = g0 || g5;
16   g3 = g2 && A;
17   if g3 {
18     B = true;
19     O1 = true;
20   };
21   g4 = g2 && ! A;
22   g6 = g7 && B;
23   if g6 {
24     O1 = true;
25   };
26   g8 = g0 || (g7 && ! B);
27   e2 = ! g4;
28   e6 = ! g8;
29   g1 = (g3 || e2) &&
30     (g6 || e6) && (g3 || g6);
31   if g1 {
32     O1 = false;
33     O2 = true;
34   };
35   g4_pre = g4;
36   g8_pre = g8;
37 }
```

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



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

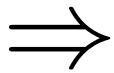
ABO SCL, Logic Synthesis (HW)

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

ABO SCL, Logic Synthesis (HW)

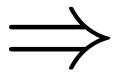
Difference to software

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



ABO SCL, Logic Synthesis (HW)

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



Difference to software

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

ABO SCL, Logic Synthesis (HW)

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

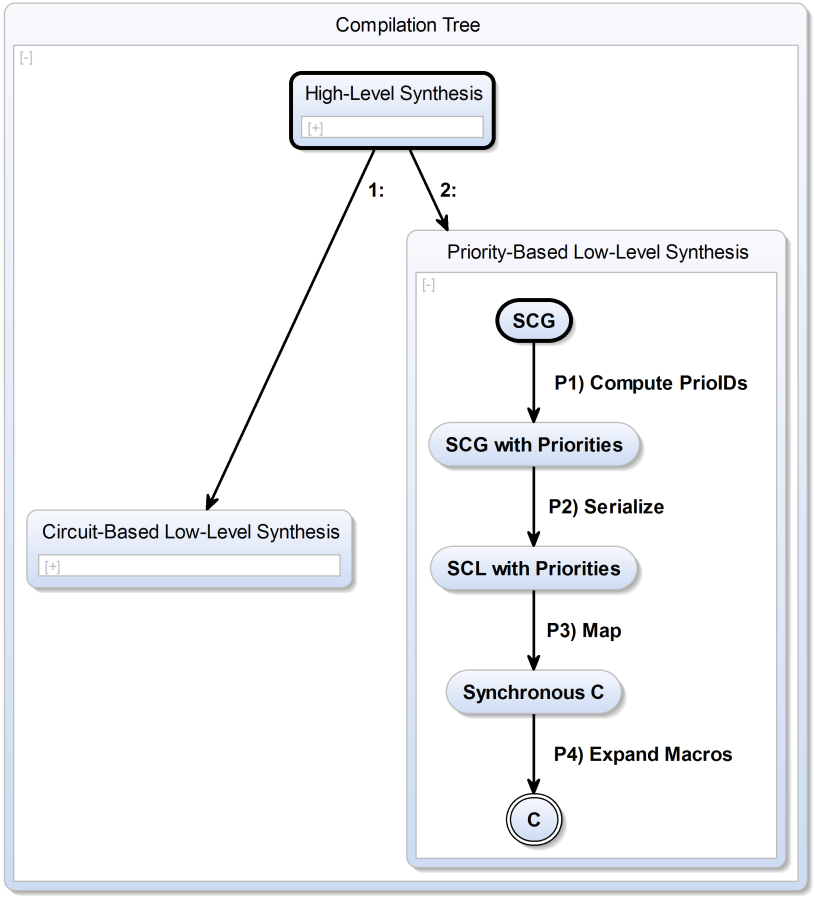
⇒

Difference to software

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

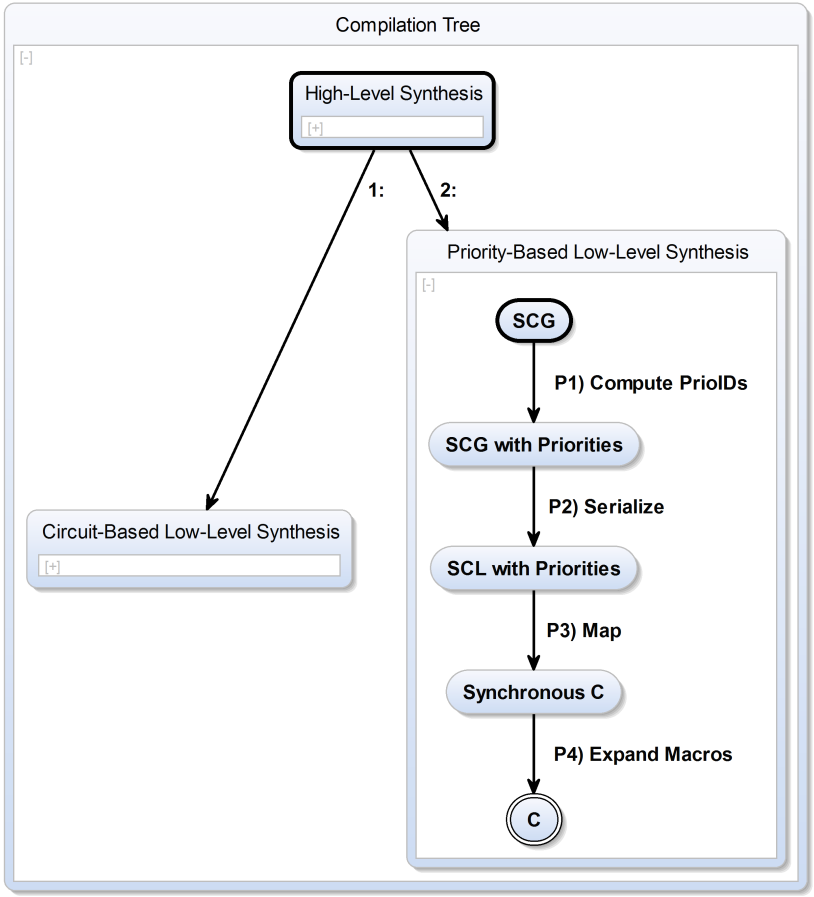
```
1  ARCHITECTURE behavior OF ABO IS
2  -- local signals definition, hidden
3  begin
4  -- main logic
5  g0 <= GO_local;
6  O1 <= false WHEN g0 ELSE O1_pre;
7  O2 <= false WHEN g0 ELSE O2_pre;
8  g5 <= g4_pre;
9  g7 <= g8_pre;
10 g2 <= g0 or g5;
11 g3 <= g2 and A_local;
12 B <= true WHEN g3 ELSE B_local;
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);
```

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

Low-Level Synthesis II: The Priority Approach

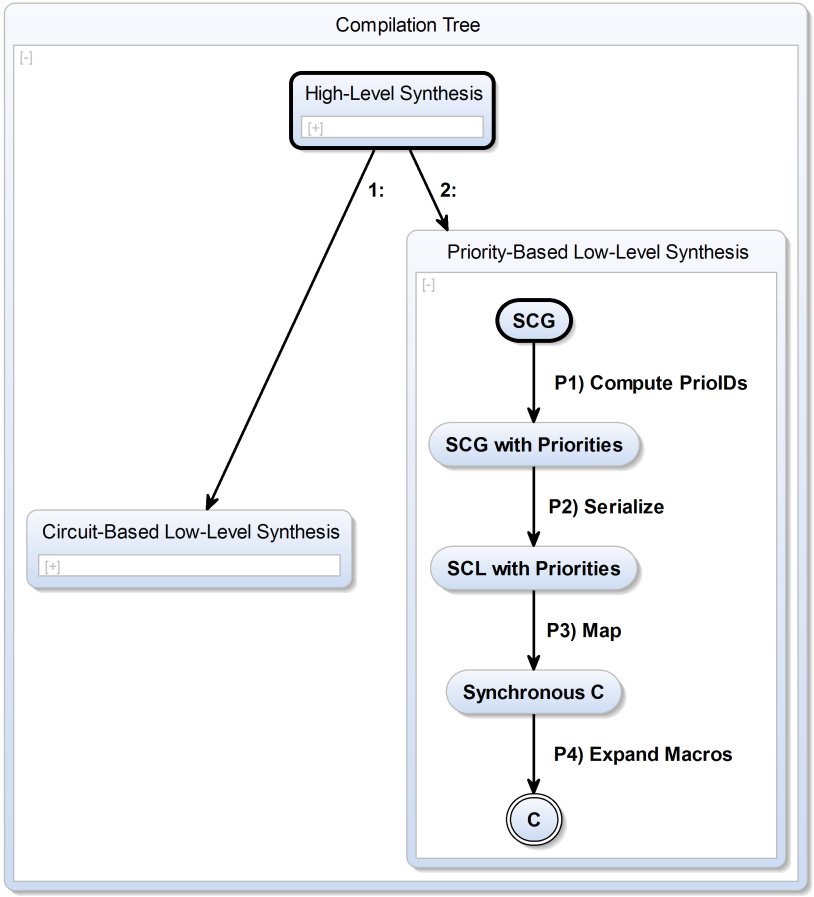


- More software-like
- Don't emulate control flow with guards/basic blocks, but with program counters/threads
- Priority-based thread dispatching
- SCL$_P$: SCL + PrioIDs
- Implemented as C macros

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

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

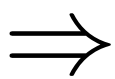
```
1  // Declare Boolean type
2  typedef int bool;
3  #define false 0
4  #define true  1
5
6  // Generate "_L<line-number>" label
7  #define _concat_helper(a, b)   a ## b
8  #define _concat(a, b)      _concat_helper(a, b)
9  #define _label_            _concat(_L, __LINE__)
10
11 // Enable/disable threads 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

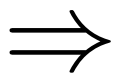
```
85  int  tick ()
86  {
87    tickstart(2);
88    O1 = false;
89    O2 = false;
90
91    fork1(HandleB, 1) {
92      HandleA:
93      if  (!A) {
94        pause;
95        goto HandleA;
96      }
97      B = true;
98      O1 = true;
99
100   } par {
```



```
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; {
```

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

Comparison of Low-Level Synthesis Approaches

	Circuit	**Priority**
Accepts instantaneous loops	−	+
Can synthesize hardware	+	−
Can synthesize software	+	+

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)	+	+

Comparison of Low-Level Synthesis Approaches

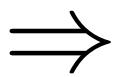
	Circuit	**Priority**
Accepts instantaneous loops	–	+
Can synthesize hardware	+	–
Can synthesize software	+	+
Size scales well (linear in size of SCChart)	+	+
Speed scales well (execute only “active” parts)	–	+
Instruction-cache friendly (good locality)	+	–
Pipeline friendly (little/no branching)	+	–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

. . . What About That Acyclicity? (Bonus Track)

```
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      _Term = false;
8      fork
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 }
```

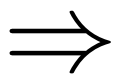
SCL (1st try)

. . . What About That Acyclicity? (Bonus Track)

```
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      _Term = false;
8      fork
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)

. . . What About That Acyclicity? (Bonus Track)

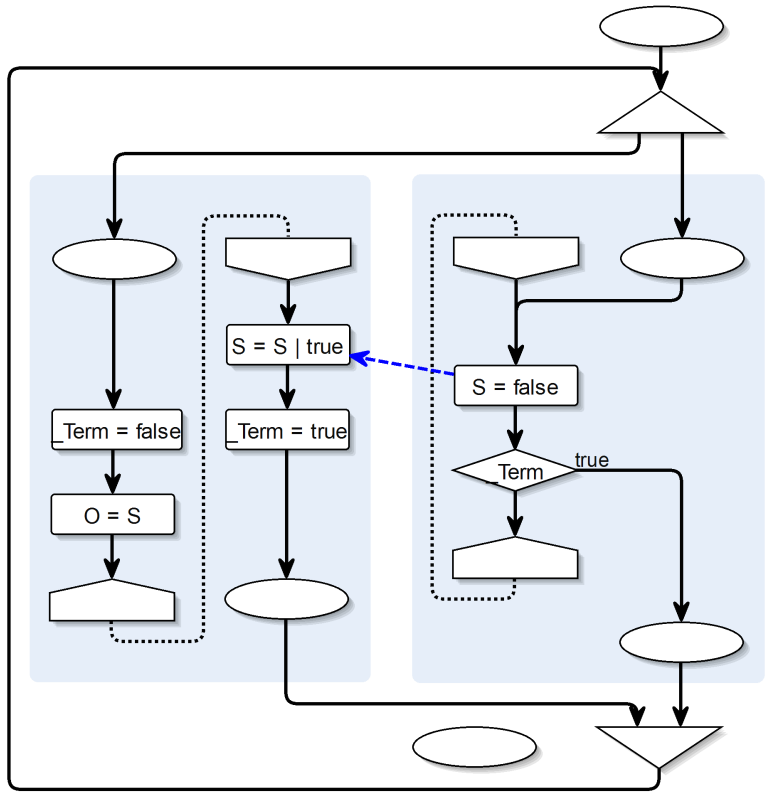
```
1  module schizo
2  output O;
3
4  loop
5    signal S in
6      present S
7        then
8        emit O
9      end;
10     pause;
11     emit S;
12   end;
13 end loop
14 end module
```

Esterel [Tardieu & de Simone '04]

$\Rightarrow$

```
1  module schizo−conc
2  output bool O;
3  {
4    while (true) {
5      bool S, _Term;
6
7      _Term = false;
8      fork
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)



- SC MoC $\Rightarrow$ init ("S = false") before update ("S = S or true")

Q: The problem for code synthesis?

. . . What About That Acyclicity? (Bonus Track)

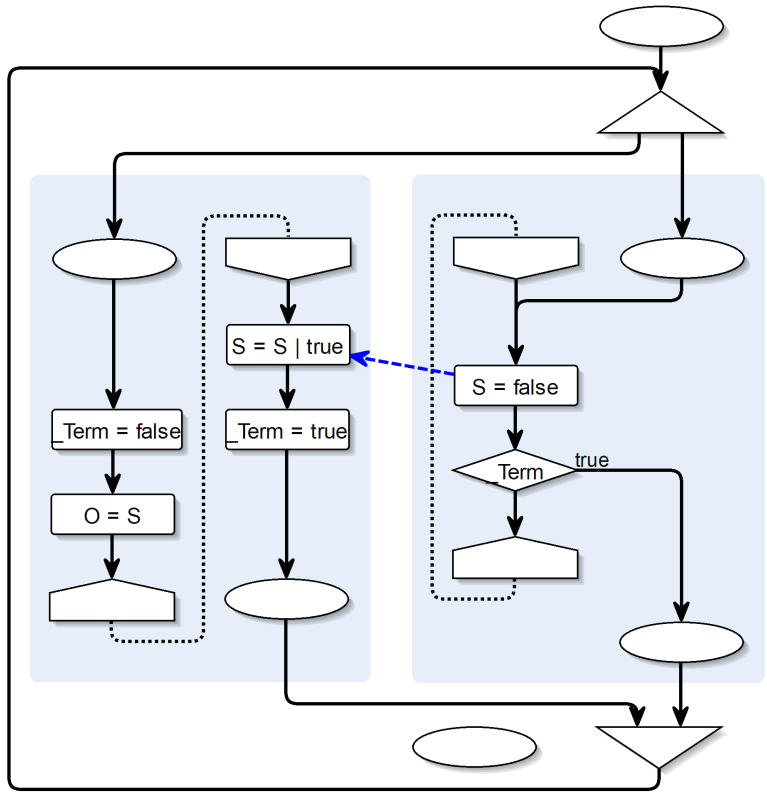
```
1  module schizo
2  output O;
3
4  loop
5    signal S in
6      present S
7        then
8        emit O
9      end;
10     pause;
11     emit S;
12   end;
13 end loop
14 end module
```

Esterel [Tardieu & de Simone '04]

⇒

```
1  module schizo_conc
2  output bool O;
3  {
4    while (true) {
5      bool S, _Term;
6
7      _Term = false;
8      fork
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)



- SC MoC ⇒ init ("S = false") before update ("S = S or true")

Q: The problem for code synthesis?

A: Cycle(s)!

. . . What About That Acyclicity? (Bonus Track)

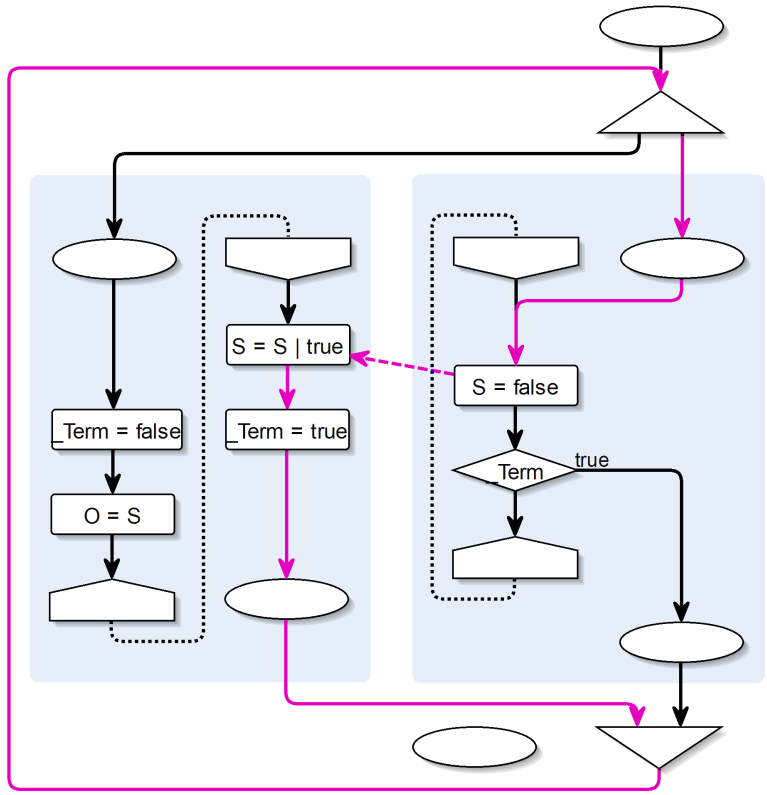
```
1  module schizo
2  output O;
3
4  loop
5    signal S in
6      present S
7        then
8        emit O
9      end;
10     pause;
11     emit S;
12   end;
13 end loop
14 end module
```

Esterel
[Tardieu & de Simone '04]

⇒

```
1  module schizo−conc
2  output bool O;
3  {
4    while (true) {
5      bool S, _Term;
6
7      _Term = false;
8      fork
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)



- SC MoC ⇒ init ("S = false") before update ("S = S or true")

Q: The problem for code synthesis?

A: Cycle(s)!
a.k.a. Signal reincarnation,
a.k.a. Schizophrenia

A Solution

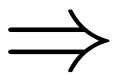
```
1  module schizo
2  output O;
3
4  loop
5     signal S in
6        present S
7           then
8           emit O
9        end;
10       pause;
11       emit S;
12    end;
13 end loop
14 end module
```

Esterel

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



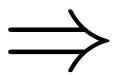
```
1  module schizo−cured
2  output O;
3
4  loop
5    signal S in
6      present S then
7        emit O
8      end;
9      pause;
10     emit S;
11   end;
12   signal S' in
13     present S' then
14       emit O
15     end;
16     pause;
17     emit S';
18   end;
19 end loop
```

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

A 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

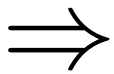
```
1  module schizo
2  output O;
3
4  loop
5     signal S in
6        present S
7           then
8           emit O
9        end;
10       pause;
11       emit S;
12    end;
13 end loop
14 end module
```

Esterel

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
```

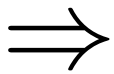
[Tardieu & de Simone '04] (simplified)

- Duplicated loop body
- "Surface copy" transfers control immediately to "depth copy"

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
```

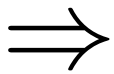
[Tardieu & de Simone '04] (simplified)

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

A 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

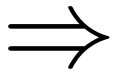
```
1  module schizo
2  output O;
3
4  loop
5     signal S in
6        present S
7           then
8           emit O
9        end;
10       pause;
11       emit S;
12    end;
13 end loop
14 end module
```

Esterel

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 }
```

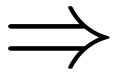
SCL

- “Surface initialization” at beginning of scope
- Delayed, concurrent “depth initialization”

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 }
```

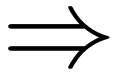
SCL

- “Surface initialization” at beginning of scope
- Delayed, concurrent “depth initialization”
- Q: Complexity?

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 }
```

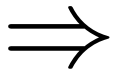
SCL

- "Surface initialization" at beginning of scope
- Delayed, concurrent "depth initialization"
- Q: Complexity?
- A: Linear ☺

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: This addresses **signal** reincarnation, i. e., instantaneously exiting and entering a signal scope
- Reincarnated **statements** still require duplication (quadratic worst case?)

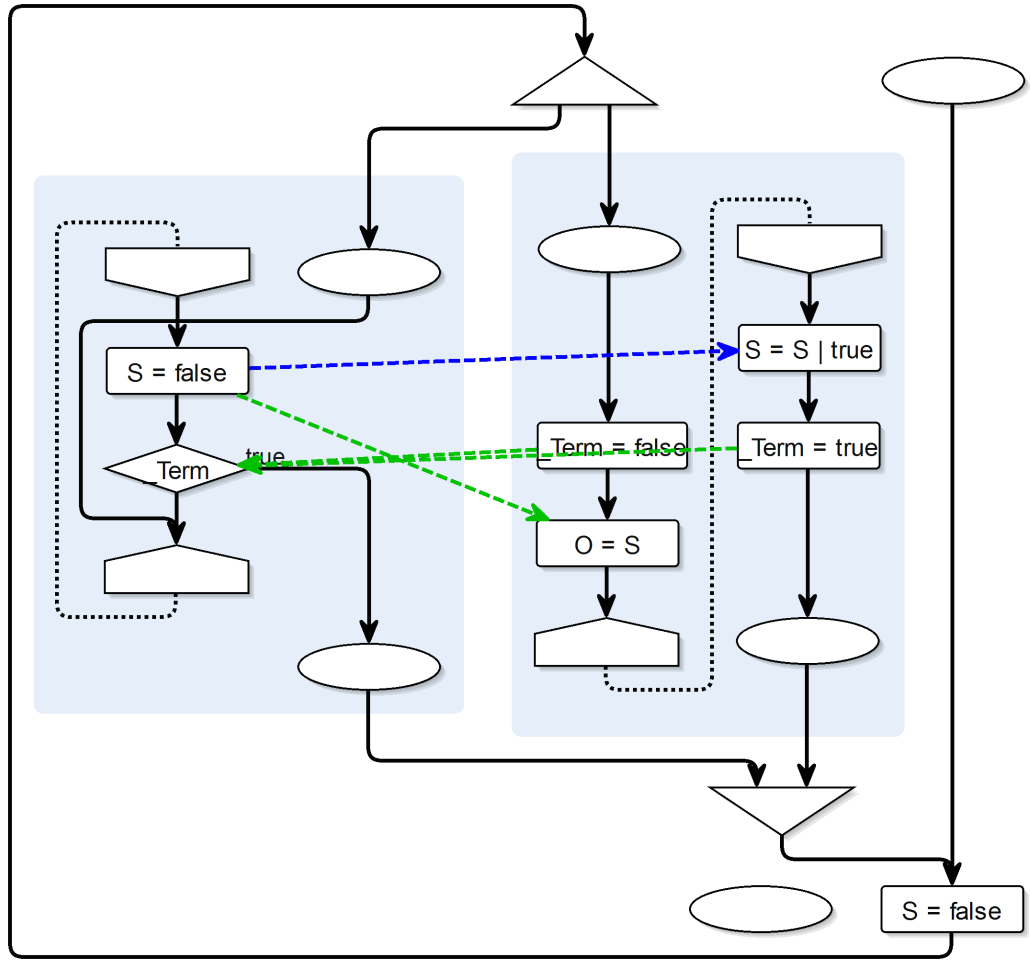
SCG for schizo-cured

```
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 }
```

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
- To avoid cycle at netlist level:
  - Parallel is non-instantaneous

  $\Rightarrow$ Only “depth join”

Take-Home Messages on SCCharts/SC

Take-Home Messages on SCCharts/SC

1. Sequential Constructiveness **natural** for synchrony
   - Already implicit in Synchronous C, PRET-C, SystemJ, Céu, ...

Take-Home Messages on SCCharts/SC

1. Sequential Constructiveness **natural** for synchrony
   - Already implicit in Synchronous C, PRET-C, SystemJ, Céu, ...
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

Take-Home Messages on SCCharts/SC

1. Sequential Constructiveness **natural** for synchrony
   - Already implicit in Synchronous C, PRET-C, SystemJ, Céu, ...
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, ...)

Take-Home Messages on SCCharts/SC

1. Sequential Constructiveness **natural** for synchrony
   - Already implicit in Synchronous C, PRET-C, SystemJ, Céu, ...
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)

Take-Home Messages on SCCharts/SC

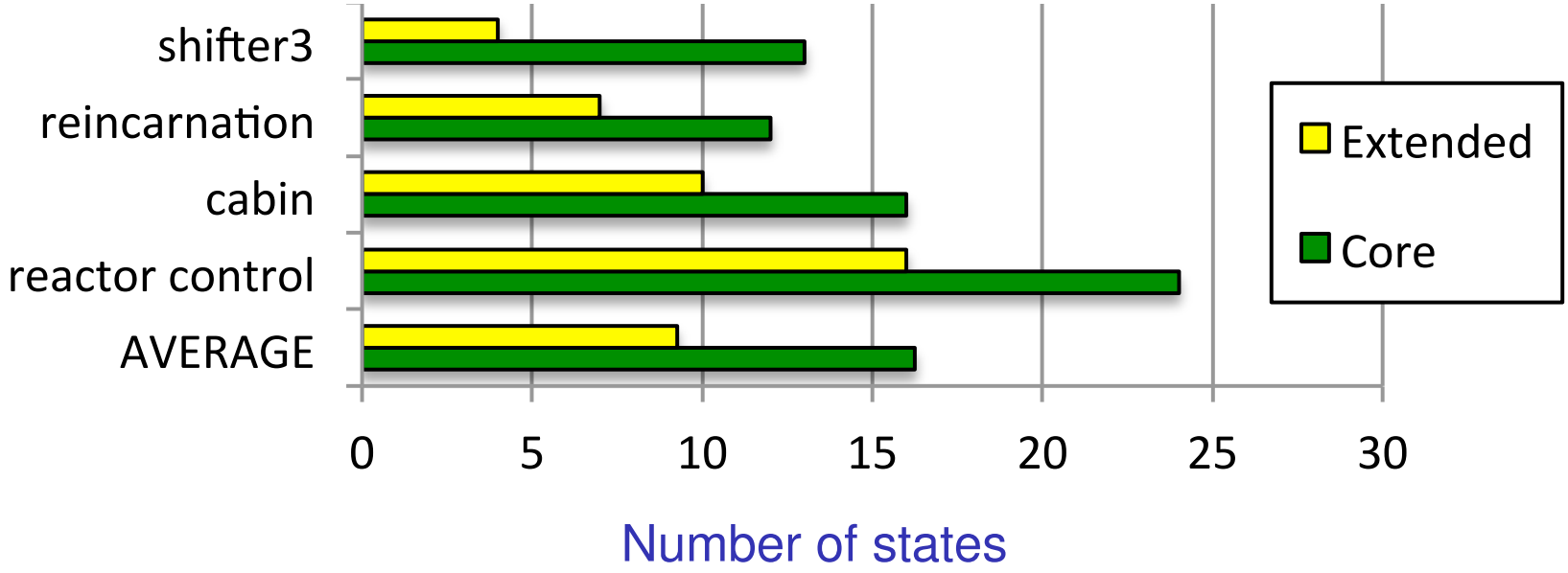
1. Sequential Constructiveness **natural** for synchrony
   - Already implicit in Synchronous C, PRET-C, SystemJ, Céu, ...
2. Same semantic foundation from Extended SCCharts down to machine instructions/physical gates
   - Modeler/programmer has direct access to target platform
   - No conceptual breaks, 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, ...

Take-Home Messages on SCCharts/SC

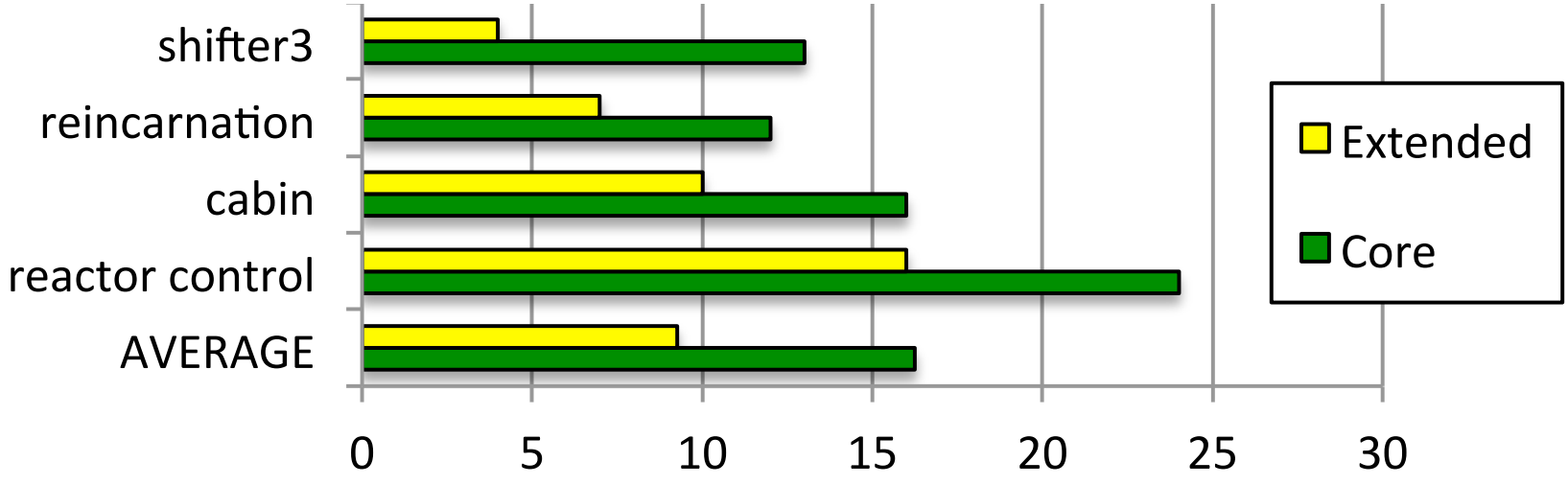
1. Sequential Constructiveness **natural** for synchrony
   - Already implicit in Synchronous C, PRET-C, SystemJ, Céu, ...
2. Same semantic foundation from Extended SCCharts down to machine instructions/physical gates
   - Modeler/programmer has direct access to target platform
   - No conceptual breaks, 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, ...

Thanks!

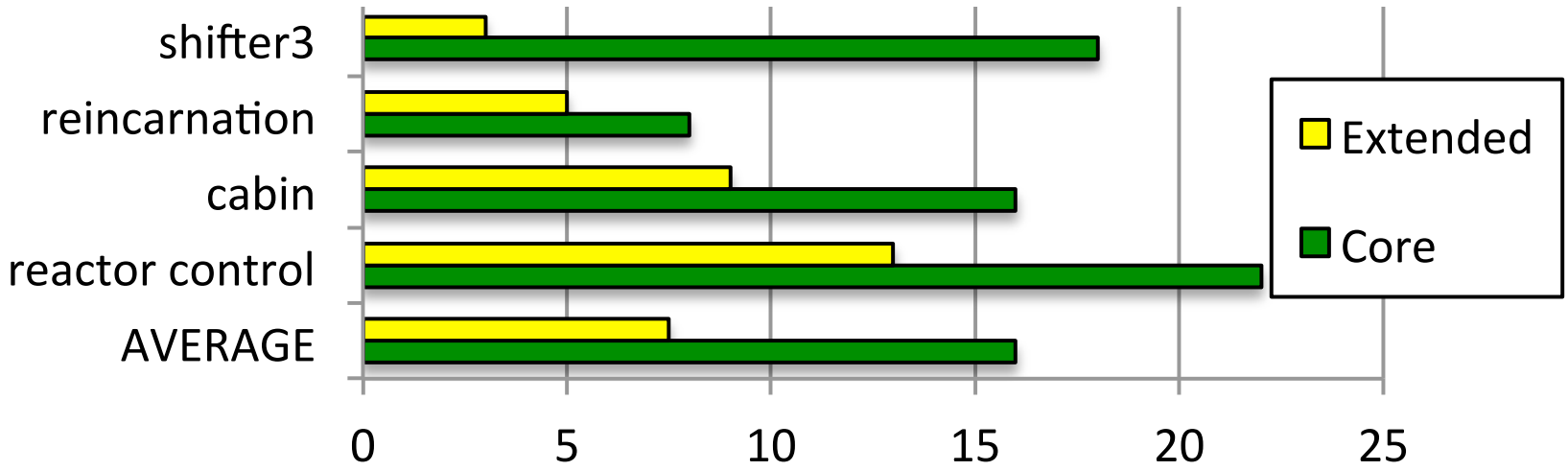
Effects of Expanding Expensions



Effects of Expanding Expensions



Number of states



Number of transitions