Compiling SCCharts to Hardware and Software

Reinhard von Hanxleden¹, Björn Duderstadt¹, Christian Motika¹, Steven Smyth¹, Michael Mendler², Joaquin Aguado², Stephen Mercer³, and Owen O’Brien³

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

SYNCHRON 2013, Schloss Dagstuhl
SCCharts Overview

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

SCCharts = Core SCCharts

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

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

![Compilation Tree Diagram]
High-Level Synthesis

Compilation Tree

Extended SCChart

H1) Expand Extensions

Core SCChart

H2) Normalize

Normalized Chart

H3) Analyze Dependencies

SC Graph (SCG)

1:

Circuit-Based Low-Level Synthesis

2:

Priority-Based Low-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 $\in \text{Prod}_f$
- $T_f$ can preserve features $\text{Handle}_f$
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

```
input output bool A
input output bool B
output bool O1
output bool O2

Init
/ O1 = false; O2 = false

WaitAB
```

```
WaitB
B / O1 = true
 DoneB
```

```
WaitA
A / B = true; O1 = true
 DoneA
```

```
GetAB
/ O1 = false; O2 = true
```
Example: Normalizing ABO
### High-Level Step 3: Map to SC Graph

<table>
<thead>
<tr>
<th>SCCharts</th>
<th>Region (Thread)</th>
<th>Superstate (Concurrency)</th>
<th>Trigger (Conditional)</th>
<th>Effect (Assignment)</th>
<th>State (Delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="Thread Diagram" /></td>
<td><img src="image2" alt="Superstate Diagram" /></td>
<td><img src="image3" alt="Trigger Diagram" /></td>
<td><img src="image4" alt="Effect Diagram" /></td>
<td><img src="image5" alt="State Diagram" /></td>
</tr>
</tbody>
</table>
High-Level Step 3: Map to SC Graph

<table>
<thead>
<tr>
<th>Region (Thread)</th>
<th>Superstate (Concurrency)</th>
<th>Trigger (Conditional)</th>
<th>Effect (Assignment)</th>
<th>State (Delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCCharts</td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>SCG</td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>SCL</td>
<td>$t$</td>
<td>fork $t_1$ par $t_2$ join</td>
<td>if ($c$) $s_1$ else $s_2$</td>
<td>$x = e$</td>
</tr>
</tbody>
</table>

[8/29]
Example: Mapping ABO to SCG
(Recall) The Big Picture

Compilation Tree

1:
Circuit-Based Low-Level Synthesis

2:
Priority-Based Low-Level Synthesis
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
<table>
<thead>
<tr>
<th>SCCharts</th>
<th>Region (Thread)</th>
<th>Superstate (Concurrency)</th>
<th>Trigger (Conditional)</th>
<th>Effect (Assignment)</th>
<th>State (Delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="Thread Diagram" /></td>
<td><img src="image2" alt="Concurrency Diagram" /></td>
<td><img src="image3" alt="Conditional Diagram" /></td>
<td><img src="image4" alt="Assignment Diagram" /></td>
<td><img src="image5" alt="Delay Diagram" /></td>
</tr>
<tr>
<td>Region (Thread)</td>
<td>Superstate (Concurrency)</td>
<td>Trigger (Conditional)</td>
<td>Effect (Assignment)</td>
<td>State (Delay)</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------</td>
<td>-----------------------</td>
<td>---------------------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>SCCharts</td>
<td>![SCCharts Diagram]</td>
<td>![Trigger Diagram]</td>
<td>![Effect Diagram]</td>
<td>![State Diagram]</td>
<td></td>
</tr>
<tr>
<td>SCG</td>
<td>![SCG Diagram]</td>
<td>![Trigger Diagram]</td>
<td>![Effect Diagram]</td>
<td>![State Diagram]</td>
<td></td>
</tr>
<tr>
<td>SCL</td>
<td>![SCL Diagram]</td>
<td>![Trigger Diagram]</td>
<td>![Effect Diagram]</td>
<td>![State Diagram]</td>
<td></td>
</tr>
</tbody>
</table>

- **SCCharts**
  - Region: Thread
  - Superstate: Concurrency
  - Trigger: Conditional
  - Effect: Assignment
  - State: Delay

- **SCG**
  - Region: Thread
  - Superstate: Concurrency
  - Trigger: Conditional
  - Effect: Assignment
  - State: Delay

- **SCL**
  - Region: Thread
  - Superstate: Concurrency
  - Trigger: Conditional
  - Effect: Assignment
  - State: Delay
<table>
<thead>
<tr>
<th>Region (Thread)</th>
<th>Superstate (Concurrency)</th>
<th>Trigger (Conditional)</th>
<th>Effect (Assignment)</th>
<th>State (Delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCCharts</td>
<td>![SCChart Diagram]</td>
<td>![Trigger Diagram]</td>
<td>![Effect Diagram]</td>
<td>![State Diagram]</td>
</tr>
<tr>
<td>SCG</td>
<td>![SCG Diagram]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCL</td>
<td>![SCL Diagram]</td>
<td>fork $t_1$ par $t_2$ join</td>
<td>if $(c)$ $s_1$ else $s_2$</td>
<td>$x = e$</td>
</tr>
</tbody>
</table>

**Data-Flow Code**

<table>
<thead>
<tr>
<th>$d = g_{exit}$</th>
<th>$g_{join} = (d_1 \lor m_1) \land (d_2 \lor m_2) \land (d_1 \lor d_2)$</th>
<th>$g = \bigvee g_{in}$</th>
<th>$g = \bigvee g_{in}$</th>
<th>$g_{depth} = \text{pre}(g_{surf})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m = \neg \bigvee_{surf \in t} g_{surf}$</td>
<td></td>
<td>$g_{true} = g \land c$</td>
<td>$x' = g ? e : x$</td>
<td></td>
</tr>
<tr>
<td>Circuits</td>
<td>![Circuit Diagram]</td>
<td>$g_{false} = g \land \neg c$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ABO SCG, With Dependencies and Scheduling Blocks
ABO SCG, With Dependencies and Scheduling Blocks
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;
}
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;
    }
    B = true;
    O1 = true;
    join;
    O1 = false;
    O2 = true;
}
module ABO
input output bool A, B;
output bool O1, O2;
bool GO, g0, g1, e2, e6, g2,
g3, g4, g5, g6, g7, g8;
bool g4.pre, g8.pre;
{
  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 || g6);
  if g1 {
    O1 = false;
    O2 = false;
  }
  g4.pre = g4;
  g8.pre = g8;
}
(Recall) Low-Level Synthesis I: The Circuit Approach

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

input output bool A, B;
output bool O1, O2;

bool GO, g0, g1, e2, e6, g2,
    g3, g4, g5, g6, g7, g8;

bool g4_pre, g8_pre;

{
    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);
```
module ABO−seq
input output bool A, B;
output bool O1, O2;
bool GO, g0, g1, e2, e6, g2,
g3, g4, g5, g6, g7, g8;
bool g4_pre, g8_pre;
{
gr0 = 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);
```vhdl
module ABO_seq
input output bool A, B;
output bool O1, O2;

bool GO, g0, g1, e2, e6, g2,
g3, g4, g5, g6, g7, g8;
bool g4_pre, g8_pre;

begin
  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);
end ABO_seq;
```

**Difference to software**

- All persistence (state, data) in external reg’s (".pre"-var’s)
- Permit only one value per wire per tick
  ⇒ SSA
```verilog
module ABO_seq

input output bool A, B;
output bool O1, O2;
bool GO, g0, g1, e2, e6, g2, g3, g4, g5, g6, g7, g8;

bool g4_pre, g8_pre;
{
  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);
}
```

**Difference to software**

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

```verilog
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);
```

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
- $\text{SCL}_P$: 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
- $\text{SCL}_p$: 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 ⇒ don’t need to keep track of thread hierarchies
- Fewer, more light-weight operators
- RISC instead of CISC
- More human-friendly syntax
// 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)
// 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_:
int tick ()
{
    tickstart(2);
    O1 = false;
    O2 = false;

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

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accepts instantaneous loops</td>
<td>+</td>
</tr>
<tr>
<td>Can synthesize hardware</td>
<td>+</td>
</tr>
<tr>
<td>Can synthesize software</td>
<td>+</td>
</tr>
<tr>
<td>Size scales well (linear in size of SCChart)</td>
<td>+</td>
</tr>
<tr>
<td>Speed scales well (execute only “active” parts)</td>
<td>–</td>
</tr>
<tr>
<td>Instruction-cache friendly (good locality)</td>
<td>+</td>
</tr>
<tr>
<td>Pipeline friendly (little/no branching)</td>
<td>+</td>
</tr>
<tr>
<td>WCRT predictable (simple control flow)</td>
<td>+/–</td>
</tr>
<tr>
<td>Low execution time jitter (simple/fixed flow)</td>
<td>–</td>
</tr>
</tbody>
</table>
Comparison of Low-Level Synthesis Approaches

<table>
<thead>
<tr>
<th></th>
<th>Circuit</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accepts instantaneous loops</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Can synthesize hardware</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Can synthesize software</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
## Comparison of Low-Level Synthesis Approaches

<table>
<thead>
<tr>
<th>Feature</th>
<th>Circuit</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accepts instantaneous loops</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Can synthesize hardware</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Can synthesize software</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Size scales well (linear in size of SCChart)</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Comparison of Low-Level Synthesis Approaches

<table>
<thead>
<tr>
<th></th>
<th>Circuit</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accepts instantaneous loops</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Can synthesize hardware</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Can synthesize software</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Size scales well (linear in size of SCChart)</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Speed scales well (execute only “active” parts)</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Instruction-cache friendly (good locality)</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Pipeline friendly (little/no branching)</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>WCRT predictable (simple control flow)</td>
<td>+</td>
<td>+/−</td>
</tr>
<tr>
<td>Low execution time jitter (simple/fixed flow)</td>
<td>+</td>
<td>−</td>
</tr>
</tbody>
</table>
... What About That Acyclicity? (Bonus Track)

```
module schizo
  output O;

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

⇒

module schizo—conc
  output bool O;
{
  while (true) {
    bool S, _Term;
    _Term = false;
    fork
      O = S;
      pause;
      S = S | true;
      _Term = true;
    par
      while (true) {
        S = false;
        if (_Term)
          break;
        pause;
      }
    join;
  }
}
```

Esterel
[Tardieu & de Simone ’04]

SCL (1st try)
... What About That Acyclicity? (Bonus Track)

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

Esterel
[Tardieu & de Simone ’04]

SCL (1st try)
... What About That Acyclicity? (Bonus Track)

```
module schizo
output O;

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

Esterel
[Tardieu & de Simone '04]

SC MoC ⇒ init (“S = false”) before update (“S = S or true”)

Q: The problem for code synthesis?

SCL (1st try)
What About That Acyclicity? (Bonus Track)

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

Esterel
[Tardieu & de Simone ’04]

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 bool O;
3 {
4   while (true) {
5     bool S, _Term;
6     _Term = false;
7     fork
8       O = S;
9       pause;
10      S = S | true;
11      _Term = true;
12     par
13       while (true) {
14          S = false;
15          if (_Term)
16            break;
17          pause;
18        }
19        join;
20   }
21 }

Esterel
[Tardieu & de Simone ’04]

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

SCL (1st try)
A Solution

```plaintext
module schizo
output O;
loop
signal S in present S then emit O end;
pause;
emit S;
signal S' in present S' then emit O end;
pause;
emit S';
end loop
end module
```

Esterel
A Solution

```estrel
module schizo
output O;

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

Esterel

```estrel
module schizo—cured
output O;

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

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

```plaintext
module schizo
output O;

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

- Duplicated loop body to separate signal instances
- Q: Complexity?
- A: Exponential 😞
A Better Solution

```plaintext
module schizo
output O;

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

Esterel

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

```plaintext
module schizo
output O;

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

Esterel

[Tardieu & de Simone '04]
(simplified)

- Duplicated loop body
- “Surface copy” transfers control immediately to “depth copy”
A Better Solution

```
module schizo
output O;

loop
  signal S in
  present S then
  emit O
  end;

  pause;
  emit S;
end loop
end module
```

Esterel

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

loop
  signal S in
  present S then
  emit O
  end;

  % Depth
  signal S’ in
  pause;
  emit S’;
end loop
end module
```

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

[Tardieu & de Simone ’04]
(simplified)
A Better Solution

```plaintext
module schizo
output O;

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

Esterel

[Tardieu & de Simone ’04]
(simplified)
```

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

```
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
d          do {
            pause;
           // Depth init
            S = false;
          } while (!_Term);
        join;
  }
end loop
end module
```

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

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

```
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;
  }
}
```

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

SCL ▶ “Surface initialization” at beginning of scope ▶ Delayed, concurrent “depth initialization” ▶ Q: Complexity?
The SCL Solution

```plaintext
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;
  }
}
```

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

Esterel

```plaintext
module schizo
output O;
loop
  signal S in
    present S then
      emit O end;
pause;
emit S end loop
end module
```
The SCL Solution

```plaintext
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;
    }
}
```

- “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

```plaintext
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
To avoid cycle at netlist level:
Parallel is non-instantaneous
⇒ Only "depth join"
```
SCG for schizo-cured

```plaintext
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;
  }
}

▶ Cycle now broken by delay
▶ To avoid cycle at netlist level:
  ▶ Parallel is non-instantaneous
  ⇒ Only “depth join”

SCL ▶ Cycle now broken by delay
▶ To avoid cycle at netlist level:
  ▶ Parallel is non-instantaneous
  ⇒ Only “depth join”

27/29
```
Take-Home Messages on SCCharts/SC

1. Sequential Constructiveness

2. Same semantic foundation from Extended SCCharts down to machine instructions/physical gates

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 downstream 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, ...
Take-Home Messages on SCCharts/SC

1. Sequential Constructiveness **natural** for synchrony
Take-Home Messages on SCCharts/SC

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 downstream 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, . . .

Thanks!
Take-Home Messages on SCCharts/SC

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 downstream 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, . . .

Thanks!
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 (PRETSY), multi-clock, visualization, ...
Take-Home Messages on SCCharts/SC

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, . . .

Thanks!
Take-Home Messages on SCCharts/SC

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, . . .

Thanks!
Effects of Expanding Expansions

- shifter3
- reincarnation
- cabin
- reactor control
- AVERAGE

Number of states

Extended
Core

Number of transitions

0 5 10 15 20 25 30

Priority Circuit

AVERAGE reactor control cabin reincarnation shifter3

Extended Core

0 2000 4000 6000 8000 10000 12000 14000 16000

AVERAGE reactor control cabin reincarnation shifter3

Priority Circuit

0 10 20 30 40 50 60 70 80 90
Effects of Expanding Expensions

<table>
<thead>
<tr>
<th>Component</th>
<th>Number of States</th>
</tr>
</thead>
<tbody>
<tr>
<td>shifter3</td>
<td>47434</td>
</tr>
<tr>
<td>reincarnation</td>
<td>46140</td>
</tr>
<tr>
<td>cabin</td>
<td>56188</td>
</tr>
<tr>
<td>reactor control</td>
<td>40093</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>48357</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model File Size (KBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Levels of Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority Circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extended Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AVERAGE</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>reactor</td>
<td>control</td>
<td>cabin</td>
<td>reincarnation</td>
<td>shifter3</td>
</tr>
</tbody>
</table>