Synchronous Languages—Lecture 12

Prof. Dr. Reinhard von Hanxleden

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

11 Dec. 2018

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

Code Generation for Sequential Constructiveness
The 5-Minute Review Session
The 5-Minute Review Session

1. What are Statecharts?
The 5-Minute Review Session

1. What are Statecharts? Who invented them?
The 5-Minute Review Session

1. What are Statecharts? Who invented them?
2. What is the difference between SyncCharts and Statecharts?
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?
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?
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?
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

SCG Mapping & Dependency Analysis

Code Generation Approaches

Schizophrenia Revisited
Compilation — Overview

Compilation Tree

1:
Circuit-Based Low-Level Synthesis

2:
Priority-Based Low-Level Synthesis

High-Level Synthesis
Compilation — High-Level Synthesis

Compilation Tree

1: Circuit-Based Low-Level Synthesis
2: Priority-Based Low-Level Synthesis

Extended SCChart

H1) Expand Extensions
Core SCChart

H2) Normalize
Normalized Chart

H3) Analyze Dependencies
SC Graph (SCG)
Compilation — 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

Green: covered in previous lecture
(Recall) SCCharts - Core & Extended Features
(Recall) SCCharts - Core & Extended Features
(Recall) SCCharts - Core & Extended Features
(Recall) SCCharts - Core & Extended Features
Compilation — 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

▶ Red: coming up now
Overview

SCG Mapping & Dependency Analysis
  Compilation Overview
  The SC Graph
  Dependency Analysis

Code Generation Approaches

Schizophrenia Revisited
The SC Graph

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

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

- Nodes $S$ correspond to statements of sequential program
The SC Graph

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

- **Nodes** $S$ correspond to statements of sequential program
- **Edges** $E$ reflect sequential execution control flow
High-Level Step 3: Map to SC Graph

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- SCCharts
- Region (Thread)
- Superstate (Concurrency)
- Trigger (Conditional)
- Effect (Assignment)
- State (Delay)
### 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><strong>SCCharts</strong></td>
<td>![SCCharts Diagram]</td>
<td><img src="trigger_diagram1.png" alt="" /></td>
<td><img src="effect_diagram1.png" alt="" /></td>
<td><img src="state_diagram1.png" alt="" /></td>
</tr>
<tr>
<td><strong>SCG</strong></td>
<td>![SCG Diagram]</td>
<td>![SCG Trigger Diagram]</td>
<td>![SCG Effect Diagram]</td>
<td>![SCG State Diagram]</td>
</tr>
<tr>
<td><strong>SCL</strong></td>
<td>![SCL Diagram]</td>
<td>![SCL Trigger Diagram]</td>
<td>![SCL Effect Diagram]</td>
<td>![SCL State Diagram]</td>
</tr>
</tbody>
</table>
Example: Mapping ABO to SCG
The SC Graph — Dependencies

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

- They share a least common ancestor fork node.
- The order of their assignments does not matter.
Two assignments within the SC Graph are **concurrent** iff

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

write—write
abs. write—rel. write
write—read
rel. write—read

The SC MoC employs a strict "initialize - update - read" protocol.
(More on the SC MoC will follow in next lecture.)
Dependency Types

Dependencies are further categorized in:

- **write—write**

![Diagram showing O = false and O = true with a dashed line between them.](image-url)
Dependency Types

Dependencies are further categorized in

- write—write
- abs. write—rel. write
Dependency Types

Dependencies are further categorized in

- write—write

- write—read

- abs. write—rel. write

The SC MoC employs a strict "initialize - update - read" protocol. (More on the SC MoC will follow in next lecture.)
Dependency Types

Dependencies are further categorized in:

- **write—write**
  - O = false → O = true
  - O = true → O = true

- **abs. write—rel. write**
  - O = false → O = O | true
  - O = O | true → O

- **write—read**
  - O = false → O = true
  - O = true → O

- **rel. write—read**
  - O = false → O = O | true
  - O = O | true → O

The SC MoC employs a strict "initialize - update - read" protocol. (More on the SC MoC will follow in next lecture.)
Dependency Types

Dependencies are further categorized in:

- **write—write**
  - O = false
  - O = true

- **abs. write—rel. write**
  - O = false
  - O = O | true

- **write—read**
  - O = true
  - O

- **rel. write—read**
  - O = O | true
  - O

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
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
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
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
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
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

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
Basic Blocks

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.
Scheduling Blocks

- Basic blocks may be interrupted when a data dependency interferes.
- Structure basic blocks further: 
  **Scheduling Blocks**
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
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!
<table>
<thead>
<tr>
<th>Normalized Core SCCharts</th>
<th>Trigger (Conditional)</th>
<th>Effect (Assignment)</th>
<th>State (Delay)</th>
<th>Region (Thread)</th>
<th>Superstate (Concurrency)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

*SCCharts Data-Flow Code*

- \( g \land c \lor g \land \neg c \)
- \( g \land x' = e \)
- \( g \land \text{surf} \lor g \land \text{depth} \in t \lor g \land \text{fork} \)
- \( m = \neg (g \land \text{fork} \lor g \land \text{depth} \in t) \)
- \( g \land \text{join} = (d_1 \lor m_1) \land (d_2 \lor m_2) \land (d_1 \lor d_2) \)
<table>
<thead>
<tr>
<th>Normalized Core SCCharts</th>
<th>Trigger (Conditional)</th>
<th>Effect (Assignment)</th>
<th>State (Delay)</th>
<th>Region (Thread)</th>
<th>Superstate (Concurrency)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="Diagram1.png" alt="Diagram" /></td>
<td><img src="Diagram2.png" alt="Diagram" /></td>
<td><img src="Diagram3.png" alt="Diagram" /></td>
<td><img src="Diagram4.png" alt="Diagram" /></td>
<td><img src="Diagram5.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCL</th>
<th>if ( c ) ( s_1 ) else ( s_2 )</th>
<th>( x = e )</th>
<th>pause</th>
<th>( t )</th>
<th>fork ( t_1 ) par ( t_2 ) join</th>
</tr>
</thead>
</table>

| SCG                     | ![Diagram](Diagram6.png) | ![Diagram](Diagram7.png) | ![Diagram](Diagram8.png) | ![Diagram](Diagram9.png) | ![Diagram](Diagram10.png) |

Circuits
<table>
<thead>
<tr>
<th>Normalized Core SCCharts</th>
<th>Trigger (Conditional)</th>
<th>Effect (Assignment)</th>
<th>State (Delay)</th>
<th>Region (Thread)</th>
<th>Superstate (Concurrency)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fork $t_1$ par $t_2$ join</td>
</tr>
</tbody>
</table>

| SCL                      |                       | $x = e$             | pause        | $t$             |                          |

| SCG                      |                       |                     |              |                 |                          |

| Data-Flow Code           | $g = \lor g_{in}$      | $g = \lor g_{in}$   | $d = g_{exit}$| $g_{fork} = \lor g_{in}$|
|                          | $g_{true} = g \land c$ | $x' = g ? e : x$    | $m = \neg (g_{fork} \lor$| $g_{join} = (d_1 \lor m_1) \land (d_2 \lor m_2)$|
|                          | $g_{false} = g \land \neg c$ |                 | $\lor depth \in t g_{depth}$|                          |

| Circuits                 |                       |                     |              |                 |                          |

| Circuits                 |                       |                     |              |                 |                          |
ABO SCG, With Dependencies and Scheduling Blocks
ABO SCG, With Dependencies and Scheduling Blocks

The diagram illustrates the flow of ABO SCG with dependencies and scheduling blocks, showing the interconnections between different states and conditions like `O1`, `O2`, and blocks labeled `DoneA`, `DoneB`, `WaitA`, and `WaitB`. The flowchart emphasizes the synchronization and dependency aspects in the SCG mapping process.
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;
goto HandleB
;
B = true;
O1 = true;
O2 = true;
union
handle B:
\text{...}
}
1 module ABO
2 input output bool A, B;
3 output bool O1, O2;
4 bool GO, g0, g1, ...
5 {
6   g0 = GO;
7   if g0 {
8     O1 = false;
9     O2 = false;
10   };
11   g5 = g4_pre;
12   g7 = g8_pre;
13   g2 = g0 || g5;
14   g3 = g2 && A;
15   if g3 {
16     B = true;
17     O1 = true;
18   };
19   g4 = g2 && !A;
20   g6 = g7 && B;
21   if g6 {
22     O1 = true;
23   };
24   g8 = g0 || (g7 && !B);
25   e2 = !g4;
26   e6 = !g8;
27   g1 = (g3 || e2) && (g6 || e6) && (g3 || g6);  
28   if g1 {
29     O1 = false;
30     O2 = true;
31   };
32   g4_pre = g4;
33   g8_pre = g8;
34 }
Sequential SCG — ABO

entry

guard0 = _GO

guard0 true

O1 = false

O2 = false

guard1 = guard0

guard5 = pre(guard4)

guard6 = guard0

guard8 = pre(guard7)

guard2 = guard1 | guard5

guard3 = guard2 & A

guard3 true

B = true

O1 = true

guard4 = guard2 & !A

guard8b = guard8

guard9 = guard8b & B

guard9 true

O1 = true

guard3_e1 = !guard4

guard9_e2 = !guard7

guard10 = (guard3_e1 | guard3) & (guard9_e2 | guard9) & (guard3 | guard9)

guard10 true

O1 = false

O2 = true

guard7 = guard6 | guard8b & !B

exit
(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)

```
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) &&
```
ABO SCL, Logic Synthesis (HW)

Difference to software

```vhdl
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) &&
```
A BO SCL, Logic Synthesis (HW)

Difference to software

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

```vhdl
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) &&
```
ABO SCL, Logic Synthesis (HW)

Difference to software

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

```vhdl
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) &&
```

```vhdl
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;
  if g3 {
    B = true;
    O1 = true;
  }
  g4 <= g2 and not A_local;
  g6 <= g7 and B;
  if g6 {
    O1 = true;
  }
  g8 <= g0 or (g7 and not B);
  e2 <= not (g4);
  e6 <= not (g8);
  g1 <= (g3 or e2) and
```

⇒ All persistence (state, data) in external reg’s (“_pre”-var’s)
⇒ Permit only one value per wire per tick ⇒ SSA
Low-Level Synthesis II: The Priority Approach

- More software-like
- Don’t emulate control flow with guards/basic blocks, but with program counters/threads
- Priority-based thread dispatching
- $\text{SCL}_P$: $\text{SCL} + \text{PrioIDs}$
- 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 + PriolIDs
- 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
- $SCL_P$: $SCL + \text{PrioIDs}$
- 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_: 

SCLₚ Macros II
int tick()
{
    tickstart(2);
    O1 = false;
    O2 = false;

    fork1(HandleB, 1) {
        HandleA:
            if (!A) {
                pause;
                goto HandleA
            }
        B = true;
        O1 = true;
    }
    par {
        _pc[1] = &&HandleB; _enabled |= (1 << 1); _active |= (1 << 1);
    }
}

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;

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;

}
## 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>
## 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></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>
Comparison — Jitter

Execution time comparison of statecharts with multiple hierarchies depicts

Execution time comparison of statecharts with multiple hierarchies depicts

Execution time — Data & Structure dependency

Execution time in ms

Macro ticks

SyncCharts

SCCharts
Comparison — Jitter

Execution time comparison of statecharts with multiple hierarchies depicts

- low jitter in circuit-based approach
Comparison — Jitter

Execution time comparison of statecharts with multiple hierarchies depicts

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

SCG Mapping & Dependency Analysis

Code Generation Approaches

Schizophrenia Revisited
  Classic Approaches
  The SCL Solution
  Summary
... What About That Acyclicity?

```Esterel
1 module schizo
2 output O;
3
4 loop
5 signal S in
6 present S
7 then
8 emit 0
9 end;
10 pause;
11 emit S;
12 end;
13 end loop
14 end module
```

Esterel
[Tardieu & de Simone '04]
What About That Acyclicility?

```
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]

```
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)
... What About That Acyclicity?

1 module schizo-conc
2 output bool O;
3 {
4  while (true) {
5    bool S, _Term;
6    fork
7      _Term = false;
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    }
20  }
21}

Esterel  
[Tardieu & de Simone '04]

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

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

Esterel
[Tardieu & de Simone ’04]

Q: The problem?

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

```plaintext
module schizo
  output O;

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

Q: The problem?
A: Instantaneous loop!

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

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

Esterel
[Tardieu & de Simone '04]

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

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

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

Esterel
```
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

```plaintext
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 Solution

Esterel

```plaintext
module schizo
output O;

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

Esterel
A Better Solution

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

Esterel

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

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

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

Esterel

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

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

Esterel

```plaintext
[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

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

▶ “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
    end;
pause;
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?
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;
      } while (!_Term);
      join;
  }
}
```

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

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

```c
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
```

SCL ▶ Cycle now broken by delay ▶ Only the "depth initialization" of S creates a concurrent "initialize before update" scheduling dependence
SCG for schizo-cured

```c
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
- Only the “depth initialization” of S creates a concurrent “initialize before update” scheduling dependence

SCL
Recall the equations for joining (two) threads:
\[ g_{\text{join}} = (d_1 \lor m_1) \land (d_2 \lor m_2) \land (d_1 \lor d_2), \]
where for each thread \( t_i \) it is “done” \( d_i = g_{\text{exit}} \) and “empty” \( m_i = \neg (g_{\text{fork}} \lor \bigvee_{\text{depth} \in t_i} g_{\text{depth}}) \).
Recall the equations for joining (two) threads:
\[ g_{\text{join}} = (d_1 \lor m_1) \land (d_2 \lor m_2) \land (d_1 \lor d_2), \]
where for each thread \( t_i \) it is “done” \( d_i = g_{\text{exit}} \) and “empty” \( m_i = \neg (g_{\text{fork}} \lor \bigvee_{\text{depth} \in t_i} g_{\text{depth}}) \).

The join guard \( g_{\text{join}} \) corresponds to the K0 output of the Esterel circuit synthesis.

Since \( g_{\text{join}} \) depends on \( g_{\text{fork}} \), the reincarnation of parallel leads to non-constructive circuits, just as with Esterel circuit synthesis.
Recall the equations for joining (two) threads:
\[ g_{\text{join}} = (d_1 \lor m_1) \land (d_2 \lor m_2) \land (d_1 \lor d_2), \]
where for each thread \( t_i \) it is “done” \( d_i = g_{\text{exit}} \) and “empty” \( m_i = \neg (g_{\text{fork}} \lor \lor_{\text{depth} \in t_i} g_{\text{depth}}) \).

The join guard \( g_{\text{join}} \) corresponds to the K0 output of the Esterel circuit synthesis.

Since \( g_{\text{join}} \) depends on \( g_{\text{fork}} \), 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 \( g_{s-\text{join}} \) and depth join \( g_{d-\text{join}} \).

The logic for surface join and depth is the same, except that in depth join, we replace \( g_{\text{fork}} \) by false.
Schizophrenic Parallel

- Recall the equations for joining (two) threads:
  \[ g_{\text{join}} = (d_1 \lor m_1) \land (d_2 \lor m_2) \land (d_1 \lor d_2), \]
  where for each thread \( t_i \) it is “done” \( d_i = g_{\text{exit}} \) and “empty” \( m_i = \neg(g_{\text{fork}} \lor \bigvee_{\text{depth} \in t_i} g_{\text{depth}}) \)

- The join guard \( g_{\text{join}} \) corresponds to the K0 output of the Esterel circuit synthesis

- Since \( g_{\text{join}} \) depends on \( g_{\text{fork}} \), 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 \( g_{s-\text{join}} \) and depth join \( g_{d-\text{join}} \)

- The logic for surface join and depth is the same, except that in depth join, we replace \( g_{\text{fork}} \) by false

- One can construct examples where both \( g_{s-\text{join}} \) and \( g_{d-\text{join}} \) are needed.

- If parallel is not instantaneous, only need \( g_{d-\text{join}} \).
Schizophrenic Parallel

Recall the equations for joining (two) threads:

\[ g_{join} = (d_1 \lor m_1) \land (d_2 \lor m_2) \land (d_1 \lor d_2), \]

where for each thread \( t_i \) it is “done” \( d_i = g_{exit} \) and “empty” \( m_i = \neg (g_{fork} \lor \bigvee_{depth \in t_i} g_{depth}) \).

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 surface 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.
Consider I absent in initial tick, present in next tick
Statement Reincarnation

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

1. Sequential Constructiveness
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, ...
Summary

1. Sequential Constructiveness natural for synchrony
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 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, . . .
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, . . . )
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, . . .
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
