Synchronous Languages—Lecture 12

Prof. Dr. Reinhard von Hanxleden

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

11 Dec. 2018

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

Code Generation for
Sequential Constructiveness
The 5-Minute Review Session

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

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

- Green: covered in previous lecture
(Recall) SCCharts - Core & Extended Features
Compilation — High-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 \( G = (S, E) \)

- **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><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>
</tbody>
</table>

1: $c$  
2: $\text{[l]}$  
$\text{[l]}$  
$\text{i/x} = e$  
$\text{[c]}$
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.

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.)
Overview

SCG Mapping & Dependency Analysis

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

Schizophrenia Revisited
Low-Level Synthesis I: The Circuit Approach

- **Basic idea:**
  Generate netlist

- **Precondition:**
  Acyclic SCG (with dependency edges, but without tick edges)

- **Well-established approach for compiling SyncCharts/Esterel**

**Differences to Esterel circuit semantics [Berry '02]**

1. Simpler translation rules, as aborts/traps/suspensions already transformed away during high-level synthesis

2. SC MoC permits sequential assignments
Basic Blocks

Basic Block:
A collection of SCG nodes / SCL statements
- that can be executed monolithically

Rules:
- Split at nodes with more than one incoming control flow edge
- Split at nodes with more than one outgoing control flow edge
- Split at tick edges
- Split after fork nodes and before join nodes
- Each node can only be included in one basic block at any time
Basic blocks may be interrupted when a data dependency interferes.

Structure basic blocks further:

- **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></td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<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="image6" alt="Diagram" /></td>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
<td><img src="image9" alt="Diagram" /></td>
<td><img src="image10" alt="Diagram" /></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SCG</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="image11" alt="Diagram" /></td>
<td><img src="image12" alt="Diagram" /></td>
<td><img src="image13" alt="Diagram" /></td>
<td><img src="image14" alt="Diagram" /></td>
<td><img src="image15" alt="Diagram" /></td>
</tr>
</tbody>
</table>
ABO SCG, With Dependencies and Scheduling Blocks
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) &&
28       (g6 || e6) && (g3 || g6);
29   if g1 {
30       O1 = false;
31       O2 = true;
32   };
33   g4_pre = g4;
34   g8_pre = g8;
35 }
Sequential SCG — ABO
(Recall) Low-Level Synthesis I: The Circuit Approach

- Can use sequential SCL directly for SW synthesis
- Synthesizing HW needs a little further work...
ABO SCL, Logic Synthesis (HW)

Difference to software

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

- More software-like
- Don’t emulate control flow with guards/basic blocks, but with program counters/threads
- Priority-based thread dispatching
- SCL\(_P\): SCL + ProIDs
- 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)
SCLₚ 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_:

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_); }

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

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_:
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);
    };

    O1 = 0;
    O2 = 0;

    _pc[1] = &&HandleB; _enabled |= (1 << 1); _active |= (1 << 1);
    _notInitial = 1;

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

100 } goto _L_TERM; {
ABO SCL_p II

102  HandleB:
103       _pc[_cid] = &&_L103; goto _L_PAUSE;
104  _L103:
105       if (!B) {
106          goto HandleB;
107       }
108       O1 = true;
109       _L108:
110       if ((_enabled & (1 << 2)) != 0) { _pc[_cid] = &&_L108; goto
111          _L_PAUSE; }
112       O1 = false;
113       O2 = true;
114       tickreturn;
115   }
### 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>
<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

- 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?

```
1 module schizo
2 output O;

3 loop
4 signal S in
5 present S
6 then
7 emit O
8 end;
9 emit S;
10 end loop
11 end module
```

Esterel
[Tardieu & de Simone ’04]

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

SCL (1st try)

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

Esterel

```plaintext
module schizo
output O;

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

```plaintext
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**

```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
```

---

[Tardieu & de Simone '04]

(adjusted)

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

```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
- Only the “depth initialization” of S creates a concurrent “initialize before update” scheduling dependence
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 \( g_{d-\text{join}} \).

- In SCG, if thread terminates instantaneously in non-instantaneous parallel, we end in unjoined exit, visualized with small solid disk.
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 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
