The 5-Minute Review Session

1. How do *SCCharts* and *SyncCharts* differ?
2. What does the *initialize-update-read protocol* refer to?
3. What is the *SCG*?
4. What are *basic blocks*? What are *scheduling blocks*?
5. When compiling from the SCG, what types of *low-level synthesis* do we distinguish? How do they compare?
Safety-Critical Embedded Systems

- Embedded systems often safety-critical
- Safety-critical systems must react deterministically
- Computations often exploit concurrency
- Key challenge: Concurrency must be deterministic!

Thanks to Michael Mendler (U Bamberg) for support with these slides
Implementing (Deterministic) Concurrency

- **C, Java, etc.:**
  - Familiar
  - Expressive sequential paradigm
  - Concurrent threads *unpredictable* in functionality and timing

- **Synchronous Programming:**
  - *predictable* by construction
  - Constructiveness
  - Unfamiliar to most programmers
  - Restrictive in practice

**Aim:** Deterministic concurrency with synchronous foundations, but without synchronous restrictions.
Comparing Both Worlds

**Sequential Languages**
- C, Java, ...
- Asynchronous schedule
  - By default: Multiple concurrent readers/writers
  - On demand: Single assignment synchronization (locks, semaphores)
- Imperative
  - All sequential control flow prescriptive
  - Resolved by programmer

**Synchronous Languages**
- Esterel, Lustre, Signal, SCADE, SyncCharts ...
- Clocked, cyclic schedule
  - By default: Single writer per cycle, all reads initialized
  - On demand: Separate multiple assignments by clock barrier (pause, wait)
- Declarative
  - All micro-steps sequential control flow descriptive
  - Resolved by scheduler
Comparing Both Worlds (Cont’d)

Sequential Languages
- Asynchronous schedule
  - 😞 No guarantees of determinism or deadlock freedom
  - 😊 Intuitive programming paradigm

Synchronous Languages
- Clocked, cyclic schedule
  - 😊 Deterministic concurrency and deadlock freedom
  - 😞 Heavy restrictions by constructiveness analysis

Sequentially Constructive Model of Computation (SC MoC)
- 😊 Deterministic concurrency and deadlock freedom
- 😊 Intuitive programming paradigm
Implementing **Deterministic Concurrency**: SC MoC

- **Concurrent** micro-step control flow:
  - Descriptive
  - Resolved by scheduler
  - \(\implies\) Deterministic concurrency and deadlock freedom

- **Sequential** micro-step control flow:
  - Prescriptive
  - Resolved by the programmer
  - \(\implies\) Intuitive programming paradigm
A Sequentially Constructive Program

Motivation
Formalizing Sequential Constructiveness (SC)
Wrap-Up

A Sequentially Constructive Program

Request

Dispatch

Control

checkReq

grant

req

pend

grant

free
A Sequentially Constructive Program (Cont’d)

Control

Threads

Request

Dispatch

- request to a resource
- resource is pending
- resource has been granted
- free

- checkReq
- grant

The resource may be free or not
A Sequentially Constructive Program (Cont’d)

```plaintext
Req_entry:
p kend = false;
if req then
    pend = true;
checkReq = req;
if pend && grant then
    pend = false;
pause;
goto Req_entry;
```

Control

```plaintext
checkReq
```

```plaintext
Dis_entry:
gr ant = false;
if checkReq && free then
    grant = true;
pause;
goto Dis_entry;
```

```plaintext
req
pend
grant
free
```
A Sequentially Constructive Program (Cont’d)

Imperative program order (sequential access to shared variables)

- “write-after-write” can change value sequentially
- Prescribed by programmer
  - Accepted in SC MoC
  - Not permitted in standard synchronous MoC
A Sequentially Constructive Program (Cont’d)

```plaintext
Req_entry:
pend = false;
if req then
  pend = true;
checkReq = req;
if pend && grant then
  pause;
goto Req_entry;

Dis_entry:
grant = false;
if checkReq && free then
  grant = true;
pause;
goto Dis_entry;
```

**Concurrency** scheduling constraints (access to shared variables):
- “write-before-read” for concurrent write/reads
- “write-before-write” (*i.e.*, conflicts!) for concurrent & non-confluent writes
- Micro-tick thread scheduling prohibits race conditions
- Implemented by the SC compiler
A Constructive Game of Schedulability

logically reactive program

Programmer

Compiler

Programmer

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler

Programmer

Compiler
Sequential Admissibility – Basic Idea

- **Sequentially ordered** variable accesses
  - Are enforced by the programmer
  - Cannot be reordered by compiler or run-time platform
  - Exhibit no races

- Only **concurrent writes/reads** to the same variable
  - Generate potential data races
  - Must be resolved by the compiler
  - Can be ordered under multi-threading and run-time

The following applies to **concurrent** variable accesses only ...
Organizing Concurrent Variable Accesses

SC Concurrent Memory Access Protocol (per macro tick)

- Concurrent, multi-writer, multi-reader variables

Tick → Initialise

Concurrent, multi-writer, multi-reader variables
Goals and Challenges

The idea behind SC is simple – but getting it “right” not so!

What we are up to:

1. Want to be conservative wrt “Berry constructiveness”
   - An Esterel program should also be SC
2. Want maximal freedom without compromising determinacy
   - A determinate program should also be SC
   - An SC program must be determinate
3. Want to exploit sequentiality as much as possible
   - But what exactly *is* sequentiality?
4. Want to define not only the exact concept of SC, but also a practical strategy to implement it
   - In practice, this requires conservative approximations
   - Compiler must not accept Non-SC programs
   - Compiler may reject SC programs
References

Most of the material here draws from this reference [TECS]:

Sequentially Constructive Concurrency – A Conservative Extension of the Synchronous Model of Computation.

Unless otherwise noted, the numberings of definitions, sections etc. refer to this.

There is also an extended version [TR]:

Sequentially Constructive Concurrency – A Conservative Extension of the Synchronous Model of Computation.
Overview

Motivation

Formalizing Sequential Constructiveness (SC)
  The SC Language (SCL) and the SC Graph (SCG) [Sec. 2]
  Free Scheduling of SCGs [Sec. 3]
  The SC Model of Computation [Sec. 4]

Wrap-Up
The Sequentially Constructive Language (SCL) [Sec. 2.1]

- Foundation for the SC MoC
- Minimal Language
- Adopted from C/Java and Esterel

```
s ::= x = e | s;s | if (e) s else s | l : s | goto l |
    fork s par s join | pause
```

- $s$ Statement
- $x$ Variable
- $e$ Expression
- $l$ Program label
The SC Graph (SCG) [Sec. 2.3]

The concurrent and sequential control flow of an SCL program is given by an SC Graph (SCG).

Internal representation for:
- Semantic foundation
- Analysis
- Code generation

**SC Graph:**

Labeled graph \( G = (N, E) \)

- **Nodes** \( N \) correspond to statements of sequential program
- **Edges** \( E \) reflect sequential execution control flow
Node Types in the SCG

Node $n \in N$ has statement type $n.st$

- $n.st \in \{\text{entry, exit, goto, } x = ex, \text{ if (ex), fork, join, surf, depth}\}$
- $x$: variable, $ex$: expression.
Define edge types:

- **iur-edges**  
  $$\alpha_{iur} = \{ww, iu, ur, ir\}$$

- **instantaneous edges**  
  $$\alpha_{ins} = \{seq\} \cup \alpha_{iur}$$

- **arbitrary edges**  
  $$\alpha_a = \{tick\} \cup \alpha_{ins}$$

- **flow edges**  
  $$\alpha_{flow} = \{seq, tick\}$$
Edge Types in the SCG [Def. 2.1]

Edge $e \in E$ has edge type $e.type \in \alpha_a$

- Specifies the nature of the particular ordering constraint expressed by $e$
- For $e.type = \alpha$, write $e.src \rightarrow_{\alpha} e.tgt$, pronounced “$e.src \alpha$-precedes $e.tgt$”

- $n_1 \rightarrow_{seq} n_2$: sequential successors
- $n_1 \rightarrow_{tick} n_2$: tick successors
- $n_1 \rightarrow_{seq} n_2, n_1 \rightarrow_{tick} n_2$: flow successors, induced directly from source program

- $\rightarrow_{seq}$: reflexive and transitive closure of $\rightarrow_{seq}$

- **Note:** $n_1 \rightarrow_{seq} n_2$ does not imply fixed run-time ordering between $n_1$ and $n_2$ (consider loops)
### Mapping SCL & SCG

<table>
<thead>
<tr>
<th>Thread (Region)</th>
<th>Concurrency (Superstate)</th>
<th>Conditional (Trigger)</th>
<th>Assignment (Effect)</th>
<th>Delay (State)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCG</td>
<td>entry → fork ↓ join</td>
<td>c true → x = e</td>
<td>surface</td>
<td></td>
</tr>
<tr>
<td>SCL</td>
<td>t fork t₁ par t₂ join</td>
<td>if (c) s₁ else s₂</td>
<td>depth</td>
<td></td>
</tr>
</tbody>
</table>

Plus “;” (Sequence) and “goto” to specify sequential successors (solid edges)
module Control
input bool free, req;
output bool grant, pend;
{
  bool checkReq;

  fork {
    // Thread Request
    Request entry:
    pend = false;
    if (req)
      pend = true;
    checkReq = req;
    if (pend && grant)
      pend = false;
    pause;
    goto Request entry;
  }

  par {
    // Thread Dispatch
    Dispatch entry:
    grant = false;
    if (checkReq && free)
      grant = true;
    pause;
    goto Dispatch entry;
  }

  join;
}
Sequentiality vs. Concurrency
Static vs. Dynamic Threads

Recall: We want to distinguish between *sequential* and *concurrent* control flow.
But what do “sequential” / “concurrent” mean?
This distinction is not as easy to formalize as it may seem . . .

To get started, distinguish

- **Static** threads: Structure of a program (based on SCG)
- **Dynamic** thread instance: thread in execution
Static Threads [Sec. 2.4]

- **Given**: SCG $G = (N, E)$
- Let $T$ denote the set of threads of $G$
- $T$ includes a top-level Root thread
- With each thread $t \in T$, associate unique
  - entry node $t_{en} \in N$
  - exit node $t_{ex} \in N$
- Each $n \in N$ belongs to a thread $th(n)$ defined as
  - Immediately enclosing thread $t \in T$
  - such that there is a flow path to $n$ that originates in $t_{en}$, does not traverse $t_{ex}$, and does not traverse any other entry node $t'_{en}$, unless that flow path subsequently traverses $t'_{ex}$ also
- For each thread $t$, define $sts(t)$ as the set of statement nodes $n \in N$ such that $th(n) = t$

---

1Added to definition in paper!
Threads in Control Example

module Control
input bool free, req;
output bool grant, pend;
{
    bool checkReq;
    fork {
        // Thread Request
        Request entry:
        pend = false;
        if (req)
            pend = true;
        checkReq = req;
        if (pend && grant)
            pend = false;
        pause;
        goto Request entry;
    }
    par {
        // Thread Dispatch
        Dispatch entry:
        grant = false;
        if (checkReq && free)
            grant = true;
        pause;
        goto Dispatch entry;
    }
    join;
}

Threads $T = \{ \text{Root}, \text{Request}, \text{Dispatch} \}$

$\text{Root}$ thread consists of the statement nodes

$sts(\text{Root}) = \{ L0, L7, L28, L29 \}$

The remaining statement nodes of $N$ are partitioned into

$sts(\text{Dispatch})$ and $sts(\text{Request})$
Static Thread Concurrency and Subordination [Def. 2.2]

Let $t, t_1, t_2$ be threads in $T$

- $\text{fork}(t) = \text{def} \text{ fork node immediately preceding } t_{en}$
- For every thread $t \neq \text{Root}$:
  
  $p(t) = \text{def} \text{ th(fork}(t)))$, the parent thread

- $p^*(t) = \text{def} \{t, p(t), p(p(t)), \ldots, \text{Root}\}$, the recursively defined set of ancestor threads of $t$

- $t_1$ is subordinate to $t_2$, written $t_1 \prec t_2$, if $t_1 \neq t_2 \land t_1 \in p^*(t_2)$

- $t_1$ and $t_2$ are (statically) concurrent, denoted $t_1 \parallel t_2$, iff $t_1$ and $t_2$ are descendants of distinct threads sharing a common fork node, i.e.:
  
  $\exists t'_1 \in p^*(t_1), t'_2 \in p^*(t_2) : t'_1 \neq t'_2 \land \text{fork}(t'_1) = \text{fork}(t'_2)$

- Denote this common fork node as $\text{lcafork}(t_1, t_2)$, the least common ancestor fork

- Lift (static) concurrency notion to nodes: $n_1 \parallel n_2 \iff \text{th}(n_1) \parallel \text{th}(n_2) \iff \text{lcafork}(n_1, n_2) = \text{lcafork}(\text{th}(n_1), \text{th}(n_2))$
Concurrency and Subordination in Control-Program

module Control
input bool free, req;
output bool grant, pend;
{
  bool checkReq;
  fork {
    // Thread Request
    Request entry:
    pend = false;
    if (req)
      pend = true;
      checkReq = req;
    if (pend && grant)
      pend = false;
    pause;
    goto Request entry;
  }

  par {
    // Thread Dispatch
    Dispatch entry:
    grant = false;
    if (checkReq && free)
      grant = true;
      pause;
    goto Dispatch entry;
  }
}

Note: Concurrency on threads, in contrast to concurrency on node instances, is purely static and can be checked with a simple, syntactic analysis of the program structure.
Thread Trees [TR, Sec. 3.7]

A **Thread Tree** illustrates the static thread relationships.

- Contains subset of SCG nodes:
  1. Entry nodes, labeled with names of their threads
  2. Fork nodes, attached to the entry nodes of their threads

- Similar to the AND/OR tree of Statecharts

Thread tree for Control example:

![Thread Tree Diagram]

- Root node
- Node L7
- Subtrees for Request and Dispatch
Thread Trees – The Reinc2 Example

```plaintext
module Reinc2
output int x, y;
{
  loop:
    fork { // Thread T1
      x = 1;
    }
    par { // Thread T2
      fork { // Thread T21
        y = 1;
      }
      par { // Thread T22
        pause;
        y = 2;
      }
      join;
      fork { // Thread T23
        y = 3;
      }
      par { // Thread T24
        x = 2;
      }
      join}
    join;
  goto loop;
}
```

Alternative definition for static thread concurrency:

- Threads are concurrent iff their least common ancestor (lca) in thread tree is a fork node.
Thread Reincarnation – The Reinc Example

Are interested in run-time concurrency, i.e., whether ordering is up to discretion of a scheduler.

Observations:

- T2 exhibits thread reincarnation
- Assignments to \( x \) are both executed in the same tick, yet are sequentialized
- Thus, static thread concurrency not sufficient to capture run-time concurrency!
Statement Reincarnation I

Accesses to $x$ in $L7$ and $L11$ executed twice within tick

Denote this as statement reincarnation

Accesses are (statically) concurrent

Data dependencies $\Rightarrow$ Must schedule $L7$ before $L11$

But only within the same loop iteration!

Not enough to impose an order on the program statements

$\Rightarrow$ Need to distinguish statement instances
Traditional synchronous languages: Reject

- **Instantaneous loops** traditionally forbidden

😊 **SC: Determinate ⇒ Accept**

- One might still want to ensure that a program always terminates
- But this issue is orthogonal to determinacy and having a well-defined semantics.
Macroticks [Def. 2.3 + 2.4]

- Given: SCG $G = (N, E)$
- (Macro) tick $R$, of length $\text{len}(R) \in \mathbb{N}_{\geq 1}$:
  - mapping from micro tick indices $1 \leq j \leq \text{len}(R)$,
  - to nodes $R(j) \in N$

A macro tick is also: Linearly ordered set of node instances

- Node instance: $ni = (n, i)$,
  - with statement node $n \in N$,
  - micro tick count $i \in \mathbb{N}$

- Can identify macro tick $R$ with set
  $$\{(n, i) \mid 1 \leq i \leq \text{len}(R), n = R(i)\}$$
Run-Time Concurrency [Def. 2.5 + 2.6]

Given: macro tick \( R \), index \( 1 \leq i \leq \text{len}(R) \), node \( n \in N \)

Def.: \( \text{last}(n, i) = \max\{j \mid j \leq i, R(j) = n\} \),
retrieves last occurrence of \( n \) in \( R \) at or before index \( i \). If it does not exist, \( \text{last}_R(n, i) = 0 \).

Given: macro tick \( R \), \( i_1, i_2 \in \mathbb{N}_{\leq \text{len}(R)} \), and \( n_1, n_2 \in N \).

Def.: Two node instances \( ni_1 = (n_1, i_1) \) and \( ni_2 = (n_2, i_2) \) are (run-time) concurrent in \( R \), denoted \( ni_1 \mid_R ni_2 \), iff

1. they appear in the micro ticks of \( R \), i.e., \( n_1 = R(i_1) \) and \( n_2 = R(i_2) \),
2. they belong to statically concurrent threads, i.e., \( \text{th}(n_1) \parallel \text{th}(n_2) \), and
3. their threads have been instantiated by the same instance of the associated least common ancestor fork, i.e., \( \text{last}(n, i_1) = \text{last}(n, i_2) \) where \( n = \text{lcafork}(n_1, n_2) \)
Overview

Motivation

Formalizing Sequential Constructiveness (SC)
The SC Language (SCL) and the SC Graph (SCG) [Sec. 2]
Free Scheduling of SCGs [Sec. 3]
The SC Model of Computation [Sec. 4]

Wrap-Up
Continuations & Thread Execution States [Def. 3.1]

A continuation $c$ consists of

1. Node $c.n$ode $\in N$, denoting the current state of each thread, i.e., the node (statement) that should be executed next, similar to a program counter.

2. Status $c.status \in \{active, waiting, pausing\}$

In a trace (see later slide), round/square/no parentheses around $n = c.node$ denote $c.status$, for enabled continuations $c$. 
Continuation Pool & Configuration [Def. 3.2 + 3.3]

Continuation pool: finite set \( C \) of continuations

- \( C \) is valid if \( C \) meets some coherence properties (see [TECS]), e.g., threads in \( C \) adhere to thread tree structure

Configuration: pair \((C, M)\)

- \( C \) is continuation pool
- \( M \) is memory assigning values to variables accessed by \( G \)

A configuration is called valid if \( C \) is valid
module Control
input bool free, req;
output bool grant, pend;
{
  bool checkReq;
  fork {
    // Thread Request
    Request entry:
    pend = false;
    if (req)
      pend = true;
    checkReq = req;
    if (pend && grant)
      pend = false;
    pause;
    goto Request entry;
  }
  par {
    // Thread Dispatch
    Dispatch entry:
    grant = false;
    if (checkReq && free)
      grant = true;
    pause;
    goto Dispatch entry;
  }
  join;
}
module Control
input bool free, req;
output bool grant, pend;
{
    bool checkReq;
    fork {
        // Thread Request
        Request entry:
        pend = false;
        if (req)
            pend = true;
        checkReq = req;
        if (pend && grant)
            pend = false;
        pause;
        goto Request entry;
    }
    par {
        // Thread Dispatch
        Dispatch entry:
        grant = false;
        if (checkReq && free)
            grant = true;
        pause;
        goto Dispatch entry;
    }
    join;
}
Free Scheduling [Sec. 3.2]

Now define **free scheduling**, to set the stage for later defining “initialize-update-read” protocol (\(\rightarrow\) SC-admissible scheduling)

Only restrictions:

1. Execute only \(\prec\)-maximal threads
   - If there is at least one continuation in \(C_{cur}\), then there also is a \(\prec\)-maximal one, because of the finiteness of the continuation pool

2. Do so in an interleaving fashion
Micro Steps I

Micro step: transition $(C_{cur}, M_{cur}) \xrightarrow{c}_{\mu s} (C_{nxt}, M_{nxt})$ between two micro ticks

- $(C_{cur}, M_{cur})$: current configuration
- $c$: continuation selected for execution
- $(C_{nxt}, M_{nxt})$: next configuration

The free schedule is permitted to pick any one of the $\prec$-maximal continuations $c \in C_{cur}$ with $c.status = \text{active}$ and execute it in the current memory $M_{cur}$
Micro Steps II

(Recall:) Micro step: transition \((C_{cur}, M_{cur}) \xrightarrow{c} \mu_{s} (C_{nxt}, M_{nxt})\)

- Executing \(c\) yields a new memory \(M_{nxt} = \mu M(c, M_{cur})\) and a (possibly empty) set of new continuations \(\mu C(c, M_{cur})\) by which \(c\) is replaced, i.e., \(C_{nxt} = C_{cur} \setminus \{c\} \cup \mu C(c, M_{cur})\)

- If \(\mu C(c, M_{cur}) = \emptyset\): status flags set to active for all \(c' \in C_{nxt}\) that become \(\prec\)-maximal by eliminating \(c\) from \(C\)

- Actions \(\mu M\) and \(\mu C\) (made precise in paper) depend on the statement \(c.node.st\) to be executed

- \((C_{nxt}, M_{nxt})\) uniquely determined by \(c\), thus may write \((C_{nxt}, M_{nxt}) = c(C_{cur}, M_{cur})\)
Clock Steps I

**Quiescent** configuration \((C, M)\):
- No active \(c \in C\)
- All \(c \in C\) pausing or waiting

If \(C = \emptyset\):
- Main program terminated

Otherwise:
- Scheduler can perform a global clock step
Clock Steps II

Global clock step: \((C_{\text{cur}}, M_{\text{cur}}) \rightarrow_{\text{tick}} (C_{\text{nxt}}, M_{\text{nxt}})\)

- Transition between last micro tick of the current macro tick to first micro tick of the subsequent macro tick
- All pausing continuations of \(C\) advance from their surf node to the associated depth node:

\[
C_{\text{nxt}} = \{ c[\text{active} :: \text{tick}(n)] \mid c[\text{pausing} :: n] \in C_{\text{cur}} \} \cup \{ c[\text{waiting} :: n] \mid c[\text{waiting} :: n] \in C_{\text{cur}} \}
\]
Clock Steps III

Global clock step updates the memory:

- Let $I = \{x_1, x_2, \ldots, x_n\}$ be the designated input variables of the SCG, including input/output variables.
- Memory is updated by a new set of external input values $V_I = [x_1 = v_1, \ldots, x_n = v_n]$ for the next macro tick.
- All other memory locations persist unchanged into the next macro tick.

Formally,

$$M_{nxt}(x) = \begin{cases} 
  v_i, & \text{if } x = x_i \in I, \\
  M_{cur}(x), & \text{if } x \notin I.
\end{cases}$$
Macro Ticks

Scheduler runs through sequence

$$\begin{align*}
(C_0^a, M_0^a) \xrightarrow{c_1^a} \mu s (C_1^a, M_1^a) \xrightarrow{c_2^a} \mu s \cdots \xrightarrow{c_{k(a)}^a} \mu s (C_{k(a)}^a, M_{k(a)}^a)
\end{align*}$$  \tag{1}


to reach final quiescent configuration \((C_{k(a)}^a, M_{k(a)}^a)\)

Sequence (1) is macro tick (synchronous instant) \(a\):

\[(R^a, V^a_I) : (C_0^a, M_0^a) \xrightarrow{\Rightarrow} (C_{k(a)}^a, M_{k(a)}^a) \tag{2}\]

- \(V^a_I\): projects the initial input, \(V^a_I(x) = M_0^a(x)\) for \(x \in I\)
- \(M_{k(a)}^a\): response of \(a\)

\(R^a\): sequence of statement nodes executed during \(a\)

- \(\text{len}(R^a) = k(a)\) is length of \(a\)
- \(R^a\) is function mapping each micro tick index \(1 \leq j \leq k(a)\) to node \(R^a(j) = c_j^a.node\) executed at index \(j\)
Runs and Traces

Run of $G$: sequence of macro ticks $R^a$ and external inputs $V_i^a$, with

- initial continuation pool $C_0^0 = \{ c_0 \}$ activates the entry node of the $G$'s Root thread, i.e., $c_0.\text{node} = \text{Root.en}$ and $c_0.\text{status} = \text{active}$

- all macro tick configurations are connected by clock steps, i.e., $(C_{k(a)}^a, M_{k(a)}^a) \rightarrow \text{tick} (C_{0}^{a+1}, M_{0}^{a+1})$

Trace: externally visible output values at each macro tick $R$ [TR, Sec. 3.9]
Determinacy

Recall:

\[(C_{0}^{a}, M_{0}^{a}) \xrightarrow{c_{1}^{a}} \mu_{s} (C_{1}^{a}, M_{1}^{a}) \xrightarrow{c_{2}^{a}} \mu_{s} \cdots \xrightarrow{c_{k(a)}^{a}} \mu_{s} (C_{k(a)}^{a}, M_{k(a)}^{a})\] (1)

\[(R^{a}, V_{I}^{a}) : (C_{0}^{a}, M_{0}^{a}) \implies (C_{k(a)}^{a}, M_{k(a)}^{a})\] (2)

- **Macro (tick) configuration**: end points of a macro tick (2)
- **Micro (tick) configuration**: all other intermediate configurations \((C_{i}^{a}, M_{i}^{a})\), \(0 < i < k(a)\) seen in (1)

**Synchrony hypothesis:**

- only macro configurations are observable externally (in fact, only the memory component of those)

**Suffices to ensure that**

sequence of macro ticks \(\implies\) is determinate

- Micro tick behavior \(\xrightarrow{\mu_{s}}\) may well be non-determinate
Active and Pausing Continuations are Concurrent [TR, Prop. 2]

Given:
- \((C, M)\), reachable (micro or macro tick) configuration
- \(c_1, c_2 \in C\), active or pausing continuations with \(c_1 \neq c_2\)

Then:
- \(c_1.node \neq c_2.node\)
- \(th(c_1.node) \parallel th(c_2.node)\)
- No instantaneous sequential path from \(c_1.node\) to \(c_2.node\) or vice versa

(Proof: see [TR])
Concurrency vs. Sequentiality Revisited I

Recall: Want to exploit sequentiality as much as possible
  ▶ Thus, consider only run-time concurrent data dependencies

Recall: Static concurrency $\not\Rightarrow$ run-time concurrency
  ▶ Consider Reinc example
  ▶ Thus, can ignore some statically concurrent data dependencies
Concurrency vs. Sequentiality Revisited II

Question: Does (static) sequentiality preclude run-time concurrency?

- Then we could ignore data dependencies between nodes that are sequentially ordered
- But the answer is: no

Counterexample: Reinc3 (SCG shown on right)

- Assignments to x run-time concurrent? Yes!
- Assignments to x sequentially ordered? Yes!

Thus, concurrency and (static) sequentiality are not mutually exclusive, but orthogonal!

However, (instantaneous) run-time sequentiality (on node instances) does exclude run-time concurrency.
Notes on Free Scheduling I

Key to determinacy:
rule out uncertainties due to unknown scheduling mechanism

- Like the synchronous MoC, the SC MoC ensures macro-tick determinacy by inducing certain scheduling constraints on variable accesses

- **Unlike** the synchronous MoC, the SC MoC tries to take maximal advantage of the execution order already expressed by the programmer through sequential commands

- A scheduler can only affect the order of variable accesses through **concurrent** threads
Notes on Free Scheduling II

Recall:

▶ If variable accesses (within tick) are already sequentialized by $\rightarrow_{seq}$, they cannot appear simultaneously in the active continuation pool.

▶ Hence, no way for thread scheduler to reorder them and thus lead to a non-determinate outcome.

Similarly, threads are not concurrent with parent thread:

▶ Because of path ordering $\prec$, a parent thread is always suspended when a child thread is in operation.

▶ Thus, not up to scheduler to decide between parent and child thread.

▶ No race conditions between variable accesses performed by parent and child threads; no source of non-determinacy.
The Aim

Want to find a suitable restriction on the “free” scheduler which is

1. easy to compute
2. leaves sufficient room for concurrent implementations
3. still (predictably) sequentializes any concurrent variable accesses that may conflict and produce unpredictable responses

In the following, will define such a restriction: the SC-admissible schedules
Guideline for SC-admissibility

- Initialize-Update-Read protocol, for concurrent accesses
- Want to conservatively extend Esterel’s “Write-Read protocol” (must do emit \textit{before} testing)
- But does Esterel \textit{always} follow write-read protocol?
Write After Read Revisited

Esterel version

- Concurrent emit after present test
- But WriteAfterRead is BC – hence should also be SC!
- Observation: second emit is ineffective, i.e., does not change value

SCG

SCG

SCG
module InEffective1
output int x = 2;
    int y;
{
    fork
        if (x == 2) {
            y = 1;
            x = 7
        }
    else
        y = 0
    par
    x = 7
    join
}

If L13 is scheduled before L6:
- L13 is effective
- No out-of-order write
- y = 0

If L13 is scheduled after L8 (and L6):
- L13 is out-of-order write
- However, L13 is ineffective
- y = 1 (→ non-determinacy!)
- The problem: L8 hides the potential effectiveness of L13 wrt. L6!

- Both schedules would be permitted under a scheduling regime that permits ineffective writes
- → Strengthen notion of “ineffective writes”:
- Consider writes “ineffective” only if they do not change read!
Ineffectiveness – 2nd Try

```
module InEffective2
output bool x = false;
  int y;
{
  fork
    if (!x) {
      y = 1;
      x = x xor true
    }
  else
    y = 0
  par
    x = x xor true;
  join
```

“\( x = x \text{ xor true} \)”

- Relative writes
- Equivalent to “\( x = !x \)”

Sequence L13; L6; L11:
- \( y = 0 \)

Sequence L6; L7; L8; L13:
- **Q**: Is L13 ineffective *relative to L6*?
- **A**: Yes!
- L13 is out-of-order . . .
- but writes \( x = \text{true} \), which is what L6 read!
- \( y = 1 \) (→ again non-determinacy!)

- Again, both schedules would be permitted under a scheduling regime that permits ineffective writes
- → Replace “ineffectiveness” by “confluence”
Overview

Motivation

Formalizing Sequential Constructiveness (SC)
   The SC Language (SCL) and the SC Graph (SCG) [Sec. 2]
   Free Scheduling of SCGs [Sec. 3]
   The SC Model of Computation [Sec. 4]

Wrap-Up
Combination Functions [Def. 4.1]

Combination function $f$:
- $f(f(x, e_1), e_2) = f(f(x, e_2), e_1)$
  for all $x$ and all side-effect free expressions $e_1, e_2$
- **Sufficient condition:** $f$ is *commutative* and *associative*
- **Examples:** $\ast, +, -, \max, \text{and, or}$
Relative and Absolute Writes [Def. 4.2]

Relative writes, of type $f$ (“increment” / “modify”): $x = f(x, e)$

- $f$ must be a combination function
- Evaluation of $e$ must be free of side effects
- Thus, schedules

  $x = f(x, e_1); x = f(x, e_2)$’ and
  $x = f(x, e_2); x = f(x, e_1)$’ yield same result for $x$

- Thus, writes are confluent
- E.g., $x++$, $x = 5*x$, $x = x-10$

Absolute writes (“write” / “initialize”): $x = e$

- Writes that are not relative
- E.g., $x = 0$, $x = 2*y+5$, $x = f(z)$
iur Relations [Def. 4.3]

Given two statically concurrent accesses $n_1 \parallel n_2$ on some variable $x$, we define the iur relations

- $n_1 \rightarrow_{ww} n_2$ iff $n_1$ and $n_2$ both initialize $x$ or both perform updates of different type. We call this a $ww$ conflict.
- $n_1 \rightarrow_{iu} n_2$ iff $n_1$ initializes $x$ and $n_2$ updates $x$.
- $n_1 \rightarrow_{ur} n_2$ iff $n_1$ updates $x$ and $n_2$ reads $x$.
- $n_1 \rightarrow_{ir} n_2$ iff $n_1$ initializes $x$ and $n_2$ reads $x$.

Since $n_1 \rightarrow_{ww} n_2$ implies $n_2 \rightarrow_{ww} n_1$:

- abbreviate the conjunction of $n_1 \rightarrow_{ww} n_2$ and $n_2 \rightarrow_{ww} n_1$ with $n_1 \leftrightarrow_{ww} n_2$
- by symmetry $\rightarrow_{ww}$ implies $\leftrightarrow_{ww}$
Confluence of Nodes [Def. 4.4]

Given:

- Valid configuration \((C, M)\) of SCG
- Nodes \(n_1, n_2 \in N\)

\(n_1, n_2\) are conflicting in \((C, M)\) iff

1. \(n_1, n_2\) active in \(C\), i.e., \(\exists c_1, c_2 \in C\) with \(c_i\).status = active and \(n_i = c_i\).node

2. \(c_1(c_2(C, M)) \neq c_2(c_1(C, M))\)

\(n_1, n_2\) are confluent with each other in \((C, M)\), written: \(n_1 \sim_{(C, M)} n_2\), iff

- \(\exists\) Sequence of micro steps \((C, M) \xrightarrow{\mu s} (C', M')\) such that \(n_1\) and \(n_2\) are conflicting in \((C', M')\)
Notes on Confluence

(From definition:) \( n_1 \sim_{(C,M)} n_2 \) iff

- \( \exists \) Sequence of micro steps \((C, M) \rightarrow_{\mu_s} (C', M')\)
  such that \( n_1 \) and \( n_2 \) are conflicting in \((C', M')\)

Observations I

- Confluence is taken *relative* to valid configurations \((C, M)\)
  and *indirectly* as the absence of conflicts

- Instead of requiring that confluent nodes commute with each
  other for *arbitrary* memories, we only consider those
  configurations \((C', M')\) that are *reachable* from \((C, M)\)

- *E.g.*, if it happens for a given program that in all memories
  \( M' \) reachable from a configuration \((C, M)\) two expressions \( ex_1 \)
  and \( ex_2 \) evaluate to the same value, then the assignments \( x = ex_1 \)
  and \( x = ex_2 \) are confluent in \((C, M)\)
Notes on Confluence

(From definition:) \( n_1 \sim (C, M) n_2 \) iff

- There exists a sequence of micro steps \( (C, M) \rightarrow_{\mu s} (C', M') \)
such that \( n_1 \) and \( n_2 \) are conflicting in \( (C', M') \)

Observations II

- Similarly, if the two assignments are never jointly active in any
  reachable continuation pool \( C' \), they are confluent in \( (C, M) \),
too

- Thus, statements may be confluent for some program relative
to some reachable configuration, but not for other configurations or in another program

- However, notice that relative writes of the same type are
  confluent in the absolute sense, i.e., for all valid
  configurations \( (C, M) \) of all programs
Notes on Confluence

(From definition:) \( n_1 \sim_{(C,M)} n_2 \) iff

- \( \nexists \) Sequence of micro steps \((C, M) \rightarrow^{\mu_s} (C', M')\)
  such that \( n_1 \) and \( n_2 \) are conflicting in \((C', M')\)

Observations III

- Confluence \( n_1 \sim_{(C,M)} n_2 \) requires conflict-freeness for all configurations \((C', M')\) reachable from \((C, M)\) by arbitrary micro-sequences under free scheduling
- Will use this notion of confluence to define the restricted set of SC-admissible macro ticks
- Since compiler will ensure SC-admissibility of the execution schedule, one might be tempted to define confluence relative to these SC-admissible schedules; however, this would result in a logical cycle
Notes on Confluence

(From definition:) \( n_1 \sim_{(C,M)} n_2 \) iff

\( \exists \) Sequence of micro steps \( (C, M) \rightarrow_{\mu s} (C', M') \)

such that \( n_1 \) and \( n_2 \) are conflicting in \( (C', M') \)

Observations IV

- This relative view of confluence keeps the scheduling constraints on SC-admissible macro ticks sufficiently weak
- **Note:** two nodes confluent in some configuration are still confluent in every later configuration reached through an arbitrary sequence of micro steps
- However, more nodes may become confluent in later configurations, because some conflicting configurations are no longer reachable
- Exploit this in following definition of confluence of node instances by making confluence of node instances within a macro tick relative to the index position at which they occur
Confluence of Node Instances [Def. 4.5]

Given:

- Macro tick $R$
- $(C_i, M_i)$ for $0 \leq i \leq \text{len}(R)$, the configurations of $R$
- Node instances $ni_1 = (n_1, i_1)$ and $ni_2 = (n_2, i_2)$ in $R$, i.e., $1 \leq i_1, i_2 \leq \text{len}(R)$, $n_1 = R(i_1)$, $n_2 = R(i_2)$

Call node instances confluent in $R$, written $ni_1 \sim_R ni_2$, iff

- for $i = \text{min}(i_1, i_2) - 1$
- $n_1 \sim (C_i, M_i) n_2$
module InEffective2
output bool x = false;
  int y;
{
  fork
  if (!x) {
    y = 1;
    x = x xor true
  }
  else
    y = 0
  par
  x = x xor true;
  join
}

Recall sequence L6; L7; L8; L13:
  ▶ Q: Is L13 ineffective relative to L6?
  ▶ A: Yes!
  ▶ L13 is out-of-order . . .
  ▶ but writes x = false, which is what L6 read!
  ▶ Q: Are L6 and L13 confluent?
  ▶ A: No!
  ▶ L6 and L13 conflict at point of execution of L6

→ Def. of SC-admissibility – specifically, the underlying scheduling relations – uses confluence condition
Scheduling Relations [Def 4.6]

Given:

- Macro tick $R$ with
- Node instances $ni_{1,2} = (n_{1,2}, i_{1,2})$, i.e., $1 \leq i_{1,2} \leq \text{len}(R)$ and $n_{1,2} = R(i_{1,2})$
- $ni_{1,2}$ concurrent in $R$, i.e., $ni_1 |_R ni_2$
- $ni_{1,2}$ not confluent in $R$, i.e., $ni_1 \not\sim_R ni_2$

Then:

- $ni_1 \rightarrow^R_\alpha ni_2$ iff $n_1 \rightarrow_\alpha n_2$ for some $\alpha \in \alpha_{iur}$
- $ni_1 \rightarrow^R ni_2$ iff $i_1 < i_2$; i.e., $ni_1$ happens before $ni_2$ in $R$. 
Sequential Admissibility [Def. 4.7]

A macro tick $R$ is **SC-admissible** iff

- for all node instances $ni_{1,2} = (n_{1,2}, i_{1,2})$ in $R$, with $1 \leq i_{1,2} \leq \text{len}(R)$ and $n_{1,2} = R(i_{1,2})$,
- for all $\alpha \in \alpha_{iur}$

the scheduling condition $SC_{\alpha}$ holds:

if $ni_1 \xrightarrow{R_\alpha} ni_2$ then $ni_1 \xrightarrow{R} ni_2$.

A run for an SCG is **SC-admissible** if all macro ticks $R$ in this run are SC-admissible.
SC-admissibility vs. Determinacy

```
module NonDet
output bool x = false, y = false;
{}
fork {
  // Thread CheckX
  if (!x)
    y = true;
}
par {
  // Thread CheckY
  if (!y)
    x = true
}
join
```

▶ Admissible runs? Yes, multiple
▶ Determinate? No

Thus: **SC-admissibility \(\not\Rightarrow\) Determinacy**
SC-admissibility vs. Determinacy

▶ Admissible runs? No
▶ Determinate? Yes

Thus: Determinacy \( \not\Rightarrow \) SC-admissibility
Sequential Constructiveness [Def. 4.8]

Definition: A program $P$ is sequentially constructive (SC) iff for each initial configuration and input sequence:

1. There exists an SC-admissible run ($P$ is reactive)
2. Every SC-admissible run generates the same determinate sequence of macro responses ($P$ is determinate)
Overview

Motivation

Formalizing Sequential Constructiveness (SC)

Wrap-Up
  Synchronous Program Classes
  Summary
Synchronous Program Classes [TR, Sec. 9]

- **Sequentially Constructive (SC)**
- **Acyclic Schedulable (ASC)**
- **Static cycles**
- **Dynamic scheduling**
- **Sequences of values**
- **NonDet or concurrent writes**

Cycle of concurrent dependencies, All Programs
Synchronous Program Classes

Example $P_{APS} = \text{if } (x) x = 1$
Example: $P_{AS} = \text{if} \ (\neg x) \ x = 1$
Sequentially Constructive (SC) Logically Correct (LC) Pnueli-Shalev (PC) Berry Constructive (BC) Acyclic Schedulable (ASC)

Sequences of values Static cycles Dynamic scheduling

Speculate on absence or presence Speculate on absence

Cycle of concurrent dependencies, or concurrent writes Out-of-order scheduling

NonDet

All Programs

Example $P_{ALS} = \text{if} \ (\neg x) \ x = 1 \ \text{else} \ x = 1$
Synchronous Program Classes

Example $P_{ALPS} = \text{if } (!x \&\& y) \{x = 1; y = 1\}$
Summary

Underlying idea of sequential constructiveness rather simple
- Prescriptive instead of descriptive sequentiality
- Thus circumventing “spurious” causality problems
- Initialize-update-read protocol

However, precise definition of SC MoC not trivial
- Challenging to ensure conservativeness relative to Berry-constructiveness
- Plain initialize-update-read protocol does not accommodate, e.g., signal re-emissions
- Restricting attention to concurrent, non-confluent node instances is key
Conclusions

- Clocked, **synchronous model of execution** for **imperative, shared-memory multi-threading**
- Conservatively extends synchronous programming (Esterel) by **standard sequential control flow** (Java, C)
- $\Longrightarrow$ Deterministic concurrency with synchronous foundations, but without synchronous restrictions
  - 😊 Expressive and intuitive sequential paradigm
  - 😊 Predictable concurrent threads
Future Work

Plenty of extensions/adaptations possible . . .

▶ Alternative notions of sequential constructiveness:
  ▶ A truly “constructive” approach that sharpens SC admissibility to determinate schedules
  ▶ Extension of iur-protocol, e.g., to model ForeC
▶ Improved synthesis & analysis — see also next lecture