The 5-Minute Review Session

1. How does the constructive Boolean logic (intuitionistic logic) differ from classical Boolean logic?
The 5-Minute Review Session

1. How does the constructive Boolean logic (intuitionistic logic) differ from classical Boolean logic?
2. What is the relationship between 1. logical correctness, 2. acyclicity, 3. constructiveness, 4. delay insensitivity?
The 5-Minute Review Session

1. How does the constructive Boolean logic (intuitionistic logic) differ from classical Boolean logic?
2. What is the relationship between 1. logical correctness, 2. acyclicity, 3. constructiveness, 4. delay insensitivity?
3. In hw synthesis, which Esterel statements introduce registers?
The 5-Minute Review Session

1. How does the constructive Boolean logic (intuitionistic logic) differ from classical Boolean logic?

2. What is the relationship between 1. logical correctness, 2. acyclicity, 3. constructiveness, 4. delay insensitivity?

3. In hw synthesis, which Esterel statements introduce registers?

4. In the context of Esterel, what is reincarnation?
The 5-Minute Review Session

1. How does the constructive Boolean logic (intuitionistic logic) differ from classical Boolean logic?

2. What is the relationship between 1. logical correctness, 2. acyclicity, 3. constructiveness, 4. delay insensitivity?

3. In hw synthesis, which Esterel statements introduce registers?

4. In the context of Esterel, what is reincarnation? What is schizophrenia?
The 5-Minute Review Session

1. How does the constructive Boolean logic (intuitionistic logic) differ from classical Boolean logic?

2. What is the relationship between 1. logical correctness, 2. acyclicity, 3. constructiveness, 4. delay insensitivity?

3. In hw synthesis, which Esterel statements introduce registers?

4. In the context of Esterel, what is reincarnation? What is schizophrenia?

5. How is schizophrenia dealt with in classical programming languages?
The 5-Minute Review Session

1. How does the constructive Boolean logic (intuitionistic logic) differ from classical Boolean logic?

2. What is the relationship between 1. logical correctness, 2. acyclicity, 3. constructiveness, 4. delay insensitivity?

3. In hw synthesis, which Esterel statements introduce registers?

4. In the context of Esterel, what is reincarnation? What is schizophrenia?

5. How is schizophrenia dealt with in classical programming languages? Which problems does schizophrenia cause in hw synthesis?
The 5-Minute Review Session

1. In the context of Esterel, what is reincarnation?
1. In the context of Esterel, what is reincarnation?
2. What is schizophrenia?
The 5-Minute Review Session

1. In the context of Esterel, what is reincarnation?
2. What is schizophrenia?
3. What is a simple solution to the schizophrenia/reincarnation problem?
The 5-Minute Review Session

1. In the context of Esterel, what is reincarnation?
2. What is schizophrenia?
3. What is a simple solution to the schizophrenia/reincarnation problem?
4. What is the approach by Tardieu and de Simone?
The 5-Minute Review Session

1. In the context of Esterel, what is *reincarnation*?
2. What is *schizophrenia*?
3. What is a simple solution to the schizophrenia/reincarnation problem?
4. What is the approach by Tardieu and de Simone?
5. How do these approaches compare?
Overview

Esterel Compilation

Automata-Based Compilation
Netlist-Based Compilation
Control-Flow Graph-Based Compilation
Experimental Comparison
Compiling Esterel

- Semantics of the language are formally defined and deterministic
- Compiler must ensure that generated executable behaves correctly w.r.t. the semantics
- Challenging for Esterel

The following material is adapted with kind permission from Stephen Edwards
(http://www1.cs.columbia.edu/~sedwards/)
Compilation Challenges
Compilation Challenges

- Concurrency
Compilation Challenges

- Concurrency
- Interaction between exceptions and concurrency
Compilation Challenges

- Concurrency
- Interaction between exceptions and concurrency
- Preemption
- Resumption (pause, await, etc.)
Compilation Challenges

- Concurrency
- Interaction between exceptions and concurrency
- Preemption
- Resumption (pause, await, etc.)
- Checking causality
Compilation Challenges

- Concurrency
- Interaction between exceptions and concurrency
- Preemption
- Resumption (pause, await, etc.)
- Checking causality
- Reincarnation (schizophrenia)
  - Loop restriction generally prevents any statement from executing more than once in a cycle
  - Complex interaction between concurrency, traps, and loops can make certain statements execute more than once
Automata-based Compilation

- Given Esterel program $P$ and an input event $I$, the SOS inference rules introduced earlier produce an output event $O$ and a program derivative $P'$
  - From $P'$ and subsequent input event $I'$, can produce another program derivative $P''$ and further output event $O'$
  - Can view this as sequence of state transitions—from state $P$ to state $P'$ to state $P''$ etc.
Automata-based Compilation

- Given Esterel program $P$ and an input event $I$, the SOS inference rules introduced earlier produce an output event $O$ and a program derivative $P'$
  - From $P'$ and subsequent input event $I'$, can produce another program derivative $P''$ and further output event $O'$
  - Can view this as sequence of state transitions—from state $P$ to state $P'$ to state $P''$ etc.

- Inference rules guarantee that set of states is finite (Finite State Machine, FSM)
Automata-based Compilation

- Given Esterel program $P$ and an input event $I$, the SOS inference rules introduced earlier produce an output event $O$ and a program derivative $P'$
  - From $P'$ and subsequent input event $I'$, can produce another program derivative $P''$ and further output event $O'$
  - Can view this as sequence of state transitions—from state $P$ to state $P'$ to state $P''$ etc.

- Inference rules guarantee that set of states is finite (Finite State Machine, FSM)

- First compiler simulated an Esterel program in every possible state and generated code for each one
Automata-Based Compilation

Note: Strictly speaking, the state of an Esterel program—i.e., what must be remembered from one tick to the next—includes the following:

1. The set of program counter values where the program has paused between cycles
Automata-Based Compilation

Note: Strictly speaking, the state of an Esterel program—i.e., what must be remembered from one tick to the next—includes the following:

1. The set of program counter values where the program has paused between cycles
2. Presence status of signals accessed via \texttt{pre} operator
Automata-Based Compilation

Note: Strictly speaking, the state of an Esterel program—i.e., what must be remembered from one tick to the next—includes the following:

1. The set of program counter values where the program has paused between cycles
2. Presence status of signals accessed via \texttt{pre} operator
3. Values of valued signals
Automata-Based Compilation

**Note:** Strictly speaking, the state of an Esterel program—i.e., what must be remembered from one tick to the next—includes the following:

1. The set of program counter values where the program has paused between cycles
2. Presence status of signals accessed via `pre` operator
3. Values of valued signals
4. Values of variables
Automata-Based Compilation

Note: Strictly speaking, the state of an Esterel program—i.e., what must be remembered from one tick to the next—includes the following:

1. The set of program counter values where the program has paused between cycles
2. Presence status of signals accessed via \texttt{pre} operator
3. Values of valued signals
4. Values of variables
5. Any state kept in the host language
Automata-Based Compilation

Note: Strictly speaking, the state of an Esterel program—i.e., what must be remembered from one tick to the next—includes the following:

1. The set of program counter values where the program has paused between cycles
2. Presence status of signals accessed via \texttt{pre} operator
3. Values of valued signals
4. Values of variables
5. Any state kept in the host language

Only the program counters are reflected in states of FSM
Automata Example

```
loop
  emit A;
  await C;
  emit B;
  pause
end;
```

≡

```c
void tick() {
  static int state = 0;
  sigtype A = B = 0;
  switch (state) {
    case 0:
      A = 1;
      state = 1;
      break;
    case 1:
      if (C) {
        B = 1;
        state = 0;
      }
      break;
  }
}
```
Automata Example

\[
\text{loop}
\begin{align*}
\text{emit } & A; \\
\text{await } & C; \\
\text{emit } & B; \\
\text{pause} & \\
\text{end};
\end{align*}
\]

\[
\begin{align*}
\text{void tick()} \{} & \\
& \begin{align*}
\text{static int } & \text{state} = 0; \\
\text{sigtype } & A = B = 0;
\end{align*} \\
\text{switch (state)} \{} & \\
& \begin{align*}
\text{case 0:} & \\
& \begin{align*}
A & = 1; \\
\text{state} & = 1;
\end{align*} \\
& \text{break};
\end{align*} \\
& \begin{align*}
\text{case 1:} & \\
& \begin{align*}
\text{if (C)} \{} & \\
& \begin{align*}
B & = 1; \\
\text{state} & = 0;
\end{align*} \\
& \text{break};
\end{align*} \\
& \}
\end{align*}
\]
Automata Example

```esterel
emit A;
emit B;
await C;
emit D;
present E then
  emit B
end;
```

```java
switch (state) {
  case 0:
    A = 1;
    B = 1;
    state = 1;
    break;
  case 1:
    if (C) {
      D = 1;
      if (E) B = 1;
    }
    state = 2;
    break;
  case 2:
    }
}
```

First State
▶
A, B, emitted, go to second state

Second state
▶
if C is present, emit D, check E & emit B & go on
▶
otherwise, stay in second state

Third state
▶
Terminated
Automata Example

```plaintext
emit A;
emit B;
await C;
emit D;
present E then
  emit B
end;
```

```
switch (state) {
  case 0:
    A = 1;
    B = 1;
    state = 1;
    break;
  case 1:
    if (C)
      D = 1;
    if (E) B = 1;
    state = 2;
    break;
  case 2:
}
```

First State

- A, B, emitted, go to second state
Automata Example

```c
switch (state) {
    case 0:
        A = 1;
        B = 1;
        state = 1;
        break;
    case 1:
        if (C) {
            D = 1;
            if (E) B = 1;
            state = 2;
        }
        break;
    case 2:
}
```

First State
- A, B, emitted, go to second state

Second state
- if C is present, emit D, check E & emit B & go on
- otherwise, stay in second state
Automata Example

```c
switch (state) {
    case 0:
        A = 1;
        B = 1;
        state = 1;
        break;
    case 1:
        if (C) {
            D = 1;
            if (E) B = 1;
            state = 2;
        }
        break;
    case 2:
}
```

First State
- A, B, emitted, go to second state

Second state
- if C is present, emit D, check E & emit B & go on
- otherwise, stay in second state

Third state
- Terminated
Assessment of Automata Compilation

- Very fast code
- Internal signaling can be compiled away
Assessment of Automata Compilation

- Very fast code
- **Internal signaling can be compiled away**
- Can generate a lot of code because

- Concurrency can cause exponential state growth
  - $n$-state machine interacting with another $n$-state machine can produce $n^2$ states
  - Language provides input constraints for reducing state count
  - “these inputs are mutually exclusive” relation $A \# B \# C$
  - “if this input arrives, this one does, too” relation $D = E$
Assessment of Automata Compilation

- Very fast code
- **Internal signaling can be compiled away**
- Can generate a lot of code because
  - Concurrency can cause exponential state growth
  - $n$-state machine interacting with another $n$-state machine can produce $n^2$ states
Assessment of Automata Compilation

- Very fast code
- **Internal signaling can be compiled away**
- Can generate a lot of code because
  - Concurrency can cause exponential state growth
  - An $n$-state machine interacting with another $n$-state machine can produce $n^2$ states
- Language provides input constraints for reducing state count
  - "these inputs are mutually exclusive"
    relation $A \# B \# C$
  - "if this input arrives, this one does, too"
    relation $D \Rightarrow E$
Automata Compilation

- Not practical for large programs
- Theoretically interesting, but doesn’t work for most programs longer than 1000 lines
- All other techniques produce—in general—slower code
Netlist-Based Compilation

Second key insight:
- Esterel programs can be translated into Boolean logic circuits
Netlist-Based Compilation

Second key insight:

- Esterel programs can be translated into Boolean logic circuits

Netlist-based compiler:

- Translate each statement into a small number of logic gates
  - A straightforward, mechanical process
  - Follows circuit semantics defined earlier
Netlist-Based Compilation

Second key insight:
- Esterel programs can be translated into Boolean logic circuits

Netlist-based compiler:
- Translate each statement into a small number of logic gates
  - A straightforward, mechanical process
  - Follows circuit semantics defined earlier
- Generate code that simulates the netlist
Netlist Example

```esterel
emit A;
emit B;
await C;
emit D;
present E then
  emit B
end;
```

≡

Entry
A
B
D
C
E
Exit
Netlist Example

emit A;
emit B;
await C;
emit D;
present E then
  emit B
end;

≡

Entry

E

Exit

A
B
C
D
E
Assessment of Netlist Compilation

😊 Scales very well
  ▶ Netlist generation roughly linear in program size
  ▶ Generated code roughly linear in program size
Assessment of Netlist Compilation

😊 Scales very well
▶ Netlist generation roughly linear in program size
▶ Generated code roughly linear in program size

😊 Good framework for analyzing causality
▶ Semantics of netlists straightforward
▶ Constructive reasoning equivalent to three-valued simulation
Assessment of Netlist Compilation

😊 Scales very well
  ▶ Netlist generation roughly linear in program size
  ▶ Generated code roughly linear in program size

😊 Good framework for analyzing causality
  ▶ Semantics of netlists straightforward
  ▶ Constructive reasoning equivalent to three-valued simulation

😢 Terribly inefficient code
  ▶ Lots of time wasted computing ultimately irrelevant results
  ▶ Can be hundreds of time slower than automata
  ▶ Little use of conditionals
Netlist Compilation

- Currently the only solution for large programs that appear to have causality problems
- Scalability attractive for industrial users
Control-Flow Graph-Based

- Third key insight:
  - Esterel looks like an imperative language, so treat it as such
Control-Flow Graph-Based

- Third key insight:
  - Esterel looks like a imperative language, so treat it as such
  - Esterel has a fairly natural translation into a concurrent control-flow graph
Control-Flow Graph-Based

- Third key insight:
  - Esterel looks like a imperative language, so treat it as such
  - Esterel has a fairly natural translation into a concurrent control-flow graph
  - Trick is simulating the concurrency
  - Concurrent instructions in most Esterel programs can be scheduled statically
  - Use this schedule to build code with explicit context switches in it
The CFG Approach

every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
  loop
    present B then
      emit C end;
    pause
  end
end

if ((s0 & 3) == 1) {
  if (S) {
    s3 = 1;
    s2 = 1;
    s1 = 1;
  } else
    if (s1 >> 1)
      s1 = 3;
    else {
      if ((s3 & 3) == 1) {
        s3 = 2; t3 = L1;
      } else {
        t3 = L2;
      }
    }
}

Esterel Source

Concurrent

CFG

Sequential

CFG

C code

if ((s0 & 3) == 1) {
  if (S) {
    s3 = 1;
    s2 = 1;
    s1 = 1;
  } else
    if (s1 >> 1)
      s1 = 3;
    else {
      if ((s3 & 3) == 1) {
        s3 = 2; t3 = L1;
      } else {
        t3 = L2;
      }
    }
}
Step 1: Build Concurrent CFG

→ every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
  ||
  loop
    present B then
      emit C end;
    pause
  end
→ end
Add Threads

```plaintext
every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
→||
loop
  present B then
    emit C end;
  pause
end
end
```
Split at Pauses

every R do
    loop
    →await A;
    emit B;
    present C then
        emit D end;
    →pause
    end
end

||
loop
    present B then
        emit C end;
    pause
end
end
Add Code Between Pauses

every R do
→ loop
→ await A;
→ emit B;
→ present C then
→ emit D end;
→ pause
→ end
||
loop
→ present B then
→ emit C end;
→ pause
→ end
→ end
Build Right Thread

every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
end

[Diagram of a control-flow graph with nodes labeled A, B, C, D, R, and edges connecting them. The diagram shows a loop structure with conditions for emitting and pausing.]
Step 2: Schedule

every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
  loop
    present B then
      emit C end;
    pause
  end
end

---

Esterel Compilation

Synchronous Languages Lecture 9
Step 3: Sequentialize

- Hardest part: Removing concurrency
- Simulate the Concurrent CFG
- Main Loop:
  - For each node in scheduled order,
  - Insert context switch if from different thread
  - Copy node & connect predecessors
Run First Node
Run First Part of Left Thread

\begin{itemize}
  \item \texttt{R}
  \item \texttt{s=2}
  \item \texttt{A}
  \item \texttt{B}
  \item \texttt{C}
  \item \texttt{D}
  \item \texttt{s=1}
  \item \texttt{s=2}
\end{itemize}
Context switch: Save State

![Diagram showing context switch and state saving](image-url)
Run Right Thread
Context Switch: Restore State
Resume Left Thread
Step 3: Finished
Assessment of Control-flow Approach
Assessment of Control-flow Approach

😊 Scales as well as the netlist compiler, but produces much faster code, almost as fast as automata
Assessment of Control-flow Approach

😊 Scales as well as the netlist compiler, but produces much faster code, almost as fast as automata
😊 Not an easy framework for checking causality
Assessment of Control-flow Approach

😊 Scales as well as the netlist compiler, but produces much faster code, almost as fast as automata

😊 Not an easy framework for checking causality

😊 Static scheduling requirement more restrictive than netlist compiler
  ▶ This compiler rejects some programs that others accept
Assessment of Control-flow Approach

😊 Scales as well as the netlist compiler, but produces much faster code, almost as fast as automata

😢 Not an easy framework for checking causality

😢 Static scheduling requirement more restrictive than netlist compiler
  ▶ This compiler rejects some programs that others accept

▶ Extension: Pre-process constructive Esterel programs with cycles into equivalent non-cyclic programs [Lukoschus/von Hanxleden 2007]
  ▶ Extends applicability of compilation approaches such as the CFG-based approach
Existing Esterel Compilers

Simulation Speed

Capacity

Automata V3 [Berry ‘87], Polis [DAC ‘95]

switch (st) {
    case 0:
        st = 1;
        break;
    case 1:

Edwards 2001
Existing Esterel Compilers

Logic gates
V4, V5 [Berry ‘92, ‘96]

A = B && C;
D = A && E;

Automata
V3 [Berry ‘87],
Polis [DAC ‘95]

Capacity

Simulation Speed

Edwards 2001
Existing Esterel Compilers

- Logic gates: V4, V5 [Berry '92, '96]
- Automata: V3 [Berry '87], Polis [DAC '95]
- CFG: CNET [CASES 2k]

Simulation Speed vs. Capacity

Edwards 2001
Speed of Generated Code

- Logic Gates
- + Logic Optimization
- CFG
- Automata

Edwards 2001
Esterel Compilation

Size of Generated Code

<table>
<thead>
<tr>
<th>Size (source lines)</th>
<th>Object code size (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>

- Logic Gates
- + Logic Optimization
- CFG
- Automata

Edwards 2001
Summary

Esterel compilation techniques:
Summary

Esterel compilation techniques:

- Automata
  - Fast code
  - Doesn’t scale

Netlists
- Scales well
- Slow code
- Good for causality

Control-flow
- Scales well
- Fast code
- Bad at causality
Summary

Esterel compilation techniques:

- Automata
  - Fast code
  - Doesn’t scale

- Netlists
  - Scales well
  - Slow code
  - Good for causality
Summary

Esterel compilation techniques:

▸ Automata
  ▶ Fast code
  ▶ Doesn’t scale

▸ Netlists
  ▶ Scales well
  ▶ Slow code
  ▶ Good for causality

▸ Control-flow
  ▶ Scales well
  ▶ Fast code
  ▶ Bad at causality
To Go Further


