

Compiling Esterel into Better Circuits and Faster Simulations

Stephen A. Edwards

Department of Computer Science,
Columbia University

www.cs.columbia.edu/~sedwards

sedwards@cs.columbia.edu

An Overview of Esterel

Synchronous model of time: implicit global clock

Communication through wire-like signals

Two flavors of statement:

Combinational

Execute in one cycle

emit
present / if
loop

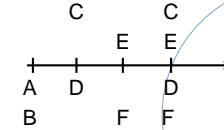
Sequential

Take multiple cycles

pause
await
sustain

Sequencing and Decisions

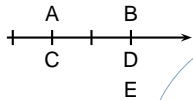
```
emit A;
emit B;
pause;
loop
  present C then emit D end;
  present E then emit F end;
pause;
end
```



Concurrency

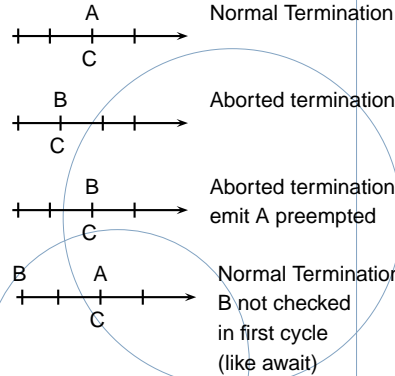
```
[
  await A; emit C
||
  await B; emit D
];
emit E
```

- Parallel statements start in same cycle
- Block terminates once all have terminated



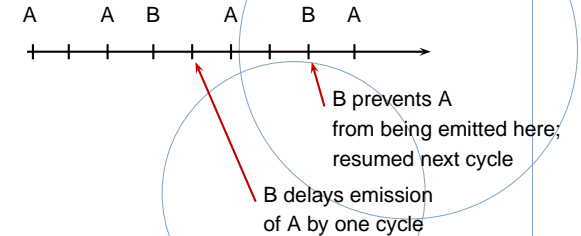
The Abort Statement

```
abort
  pause;
  pause;
  emit A
when B;
emit C
```



The Suspend Statement

```
suspend
  loop
    emit A; pause; pause
  end
when B
```



Esterel for Hardware Synthesis

Esterel for Hardware Specification

Why consider Esterel?

- Semantics more abstract than RTL
More succinct descriptions faster and easier to write
- High-level semantics enable high-level optimizations
State assignment a hierarchical problem
- High-level semantics enable more efficient simulation
Semantics are more like an imperative program
- Esterel's semantics are deterministic
Simulation-synthesis mismatches eliminated

Applications of Esterel

Systems with complex (non-pipelined) control-behavior:

- DMA controllers
- Cache controllers
- Communication protocols

(Not processors)

Verilog More Verbose Than Esterel

```

state (cur_state) // asynchronous parallel case
  if (normal_ack) begin
    next_state = STANDBY_PSR_M0;
  end
  if (cacheable) begin
    case [icu_miss and
          not cacheable] do
      emit A;
    end
  end
  if (error_ack) begin
    case [icu_miss and
          cacheable] do
      emit B;
    end
  end
  if (normal_ack) begin
    case [icu_miss and
          cacheable] do
      emit C;
    end
  end
  if (error_ack) begin
    case [icu_miss and
          cacheable] do
      emit D;
    end
  end
  if (normal_ack) begin
    case [icu_miss and
          cacheable] do
      emit E;
    end
  end
  if (error_ack) begin
    case [icu_miss and
          cacheable] do
      emit F;
    end
  end
  if (normal_ack) begin
    case [icu_miss and
          cacheable] do
      emit G;
    end
  end
  if (error_ack) begin
    case [icu_miss and
          cacheable] do
      emit H;
    end
  end
end
endstate

```

Why is Esterel More Succinct?

```

Verilog:
REQ_STATE2: begin
  if(normal_ack) begin
    next_state = FILL_4TH_WD;
  end
  else if (error_ack) begin
    next_state = IDLE ;
  end
  else next_state = cur_state;
end

Esterel:
abort
  await normal_ack
  when error_ack

```

- Esterel provides cross-clock control-flow
- State machine logic represented implicitly
- Higher-level constructs like *await*

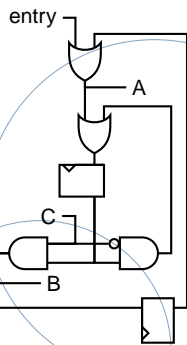
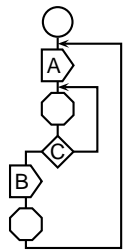
Generating Fast Circuits

Basic Circuit Generation

```

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

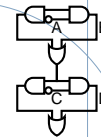
```



Basic Circuit Generation

Berry's technique [1992] works, but is fairly inefficient:

- Many combinational redundancies. E.g.,
present A then emit B end;
present C then emit D end
 produces two redundant OR gates



- Many sequential redundancies
 One flop per pause can be very wasteful
 Touati, Toma, Sentovich, and Berry [1993–1997]
 proposed techniques to eliminate many, but requires
 reachable state space and only works on circuit.

Generating Fast Circuits

Esterel's semantics match hardware. Translation is straightforward.

Nice feature: state space is well-defined and hierarchical (e.g., due to abort and concurrency).

Enables a hierarchical state assignment/synthesis procedure.

A State Assignment Example

```

abort
[
  await A; await B
  ||
  await C
]
when D;
emit E;
pause;
[
  await F
  ||
  await G
]

```

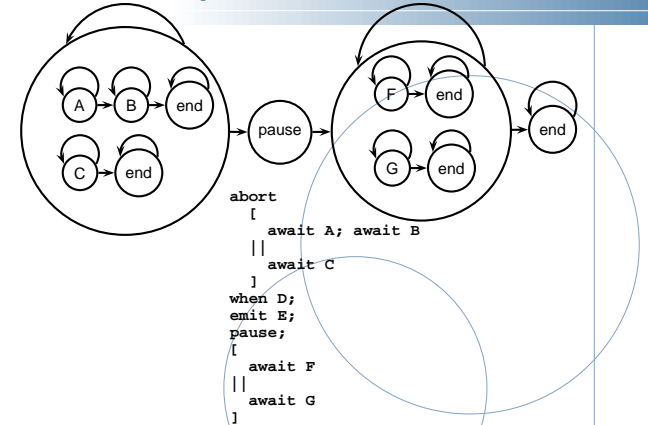
Hierarchical States

```

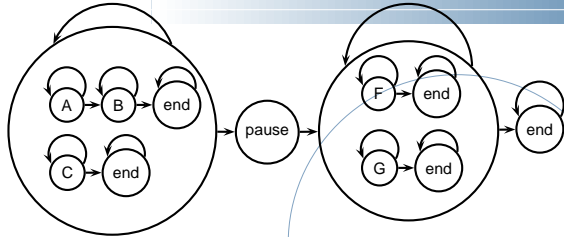
abort
[
  await A; await B
  ||
  await C
]
when D;
emit E;
pause;
[
  await F
  ||
  await G
]

```

Five Simple FSMs



Five Simple FSMs



Obvious questions:

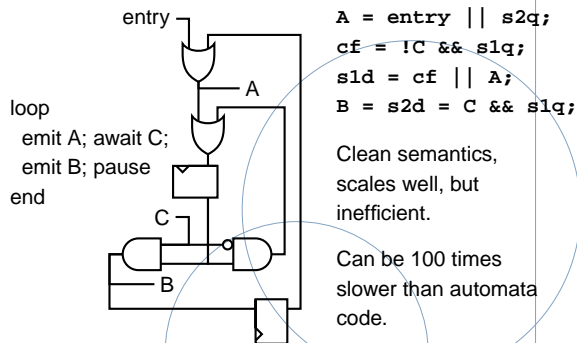
- How should each state machine be encoded?
- Should state be shared between the AB/F and C/G machines?

Choosing a Good Encoding

Goal: The smallest circuit that meets a timing constraint

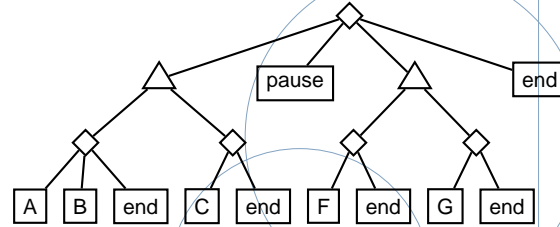
1. Start with largest, fastest circuit (one-hot, no sharing)
2. Estimate the slack at each state decision point by estimating how much the delay could be increased at that point while still meeting the timing requirement
3. Attempt to share states at the lowest decision point with the largest slack or reencode the widest-fan-out decision point with sufficient slack.
4. Repeat steps 2–3 until no further gain possible

Netlist-based Compilers

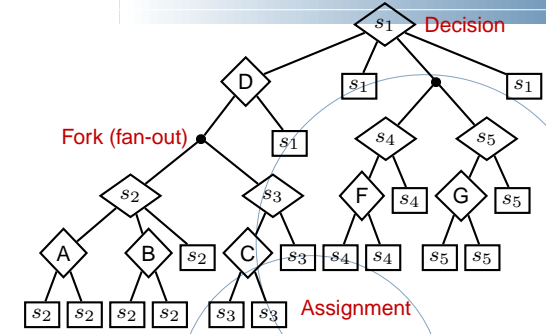


General Problem Statement

States in an Esterel program an arbitrary tree of sequential and parallel state machines.



Choosing an Encoding



- How should s_1, \dots, s_4 be encoded?
- Should s_2 or s_3 be shared with s_4 or s_5 ?

Automata Compilers

Esterel is a finite-state language, so build an automata:

```

loop
  emit A; await C;
  emit B; pause
end
switch (s) {
  case 0: A = 1; s = 1; break;
  case 1: if (C) { B = 1; s = 0; } break;
}
    
```

V1, V2, V3 (INRIA/CMA) [Berry, Gonthier 1992]

Fastest known code; great for programs with few states.

Does not scale; concurrency causes state explosion.

Simulating Esterel

Discrete-Event Based Compilers

SAXO-RT [Weil et al. 2000] Divides Esterel program into event functions dispatched by a fixed scheduler.

```

loop
  emit A; await C;
  emit B; pause
end
unsigned curr = 0x1;
unsigned next = 0;

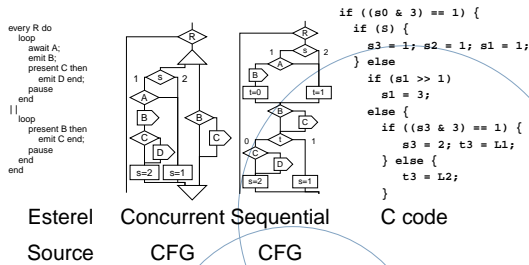
static void f1() {
  A = 1;
  curr &= ~0x1; next |= 0x2;
}

static void f2() {
  if (!C) return;
  B = 1;
  curr &= ~0x2; next |= 0x1;
}

void tick() {
  if (curr & 0x1) f1();
  if (curr & 0x2) f2();
  curr |= next;
  next = 0;
}
    
```

Generating Fast Simulations

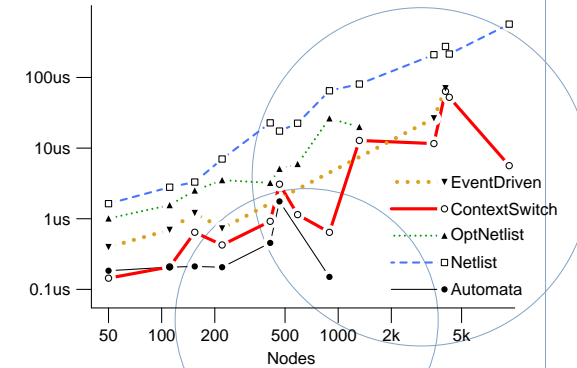
My Previous Technique



My Previous Technique

1. Translate Esterel into a concurrent control-flow graph
2. Analyze static data dependencies
3. Schedule
4. Generate sequential control-flow graph by inserting context-switching code
5. Translate to C

Average Cycle Times on an UltraSparc-II



Comments on Previous Technique

Much more efficient (can be 100×) than netlist simulation.

Currently used within Synopsys' CoCentric System Studio for control-code generation.

Context-switching idea powerful, but does not have much insight into program behavior.

Adheres too closely to control dependencies; many more opportunities to reorder code and simplify scheduling.

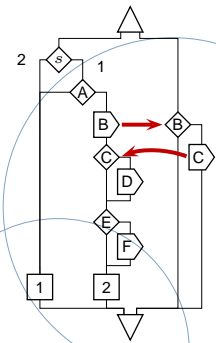
New Technique

1. Translate Esterel into a concurrent control-flow graph
2. Transform into Program Dependence Graph
3. Analyze static data dependencies
4. Insert control predicates to enable scheduling
5. Schedule
6. Generate sequential control-flow graph
7. Translate to C

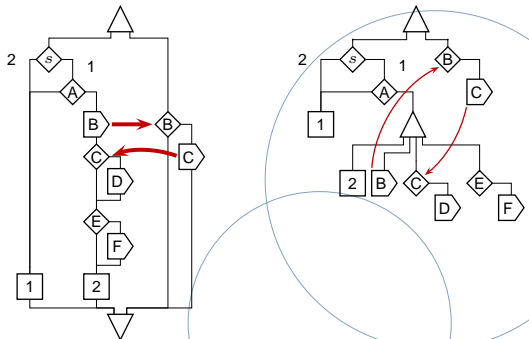
A Code-Generation Example

```

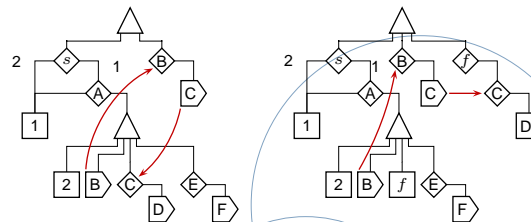
loop
  await A;
  emit B;
  present C then
    emit D end;
  present E then
    emit F end;
  pause
end
||
loop
  present B then
    emit C end;
  pause
end
    
```



Concurrent Control-Flow and the PDG



Splitting the PDG for Scheduling



Generating Code

```

f = 0;
if (s1 == 1 && A) {
  s1 = 2;
  B = 1;
  f = 1;
  if (E) F = 1;
} else {
  s1 = 1;
}
if (f && C) D = 1;
    
```

Summary

Introduction to Esterel

Synchronous, Concurrent, Textual Language
Combinational and Sequential Statements

Esterel for Hardware Synthesis

Higher-level, deterministic
More succinct than Verilog

Generating Fast Circuits

Existing translation produces many redundancies
Hierarchical state assignment problem

Summary

Simulating Esterel

Automata, Netlist, Discrete-Event, and
Context-switching techniques

New Technique: Transform to PDG, insert predicates,
schedule, generate code