The Synchronous Language Esterel

COMS W4995-02

Prof. Stephen A. Edwards
Fall 2002
Columbia University
Department of Computer Science
The Esterel Language

Developed by Gérard Berry
starting 1983

Originally for robotics applications

Imperative, textual language

Synchronous model of time like that in digital circuits

Concurrent

Deterministic
A Simple Example

The specification:

The output O should occur when inputs A and B have both arrived. The R input should restart this behavior.
A First Try: An FSM
module ABRO:
input A, B, R;
output O;

loop
  [ await A || await B ];
emit O
each R

end module

Much simpler since language includes notions of signals, waiting, and reset.
module ABRO:
input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O each R
end module
module ABRO:
input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O
  each R
end module
Basic Ideas of Esterel

Imperative, textual language
Concurrent
Based on synchronous model of time:

- Program execution synchronized to an external clock
- Like synchronous digital logic
- Suits the cyclic executive approach

Two types of statements:

- Combinational statements, which take “zero time” (execute and terminate in same instant, e.g., emit)
- Sequential statements, which delay one or more cycles (e.g., await)
Uses of Esterel

Wristwatch

- Canonical example
- Reactive, synchronous, hard real-time

Controllers, e.g., for communication protocols

Avionics

- Fuel control system
- Landing gear controller
- Other user interface tasks

Processor components (cache controller, etc.)
Advantages of Esterel

Model of time gives programmer precise timing control
Concurrency convenient for specifying control systems
Completely deterministic
  • Guaranteed: no need for locks, semaphores, etc.
Finite-state language
  • Easy to analyze
  • Execution time predictable
  • Much easier to verify formally
Amenable to both hardware and software implementation
Disadvantages of Esterel

Finite-state nature of the language limits flexibility

- No dynamic memory allocation
- No dynamic creation of processes

Little support for handling data; limited to simple decision-dominated controllers

Synchronous model of time can lead to overspecification

Semantic challenges:

- Avoiding causality violations often difficult
- Difficult to compile

Limited number of users, tools, etc.
Esterel’s Model of Time

The standard CS model (e.g., Java’s) is *asynchronous*: threads run at their own rate. Synchronization is through calls to `wait()` and `notify()`.

Esterel’s model of time is *synchronous* like that used in hardware. Threads march in lockstep to a *global clock*.

![Diagram of time and clock tick]
Signals

Esterel programs communicate through signals

These are like wires

Each signal is either present or absent in each cycle

Can’t take multiple values within a cycle

Presence/absence not held between cycles

Broadcast across the program

Any process can read or write a signal
**Basic Esterel Statements**

`emit S`

Make signal S present in the current cycle

A signal is absent unless emitted *in that cycle*.

`pause`

Stop for this cycle and resume in the next.

`present S then s₁ else s₂ end`

Run $s₁$ immediately if signal S is present in the current cycle, otherwise run $s₂$. 
module Example1:
output A, B, C;

emit A;
present A then
    emit B
end;
pause;
emit C

end module
Signal Coherence Rules

Each signal is only present or absent in a cycle, never both.

All writers run before any readers do.

Thus

```plaintext
present A else
  emit A
end
```

is an erroneous program. (Deadlocks.)

The Esterel compiler rejects this program.
Advantage of Synchrony

Easy to regulate time

Synchronization is free (e.g., no Bakers’ algorithm)

Speed of actual computation nearly uncontrollable

Allows function and timing to be specified independently

Makes for deterministic concurrency

Explicit control of “before” “after” “at the same time”
Time Can Be Controlled Precisely

This guarantees every 60th S an M is emitted

\[
\text{every } 60 \text{ S do }
\begin{align*}
\text{emit M} \\
\text{end}
\end{align*}
\]

\text{every} invokes its body every 60th S
\text{emit} takes no time (cycles)
The || Operator

Groups of statements separated || by run concurrently and terminate when all groups have terminated

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

A       B
C       D
E
Communication Is Instantaneous

A signal emitted in a cycle is visible immediately

\[
[ \text{pause; emit A; pause; emit A} ]
\]

\[
|| \text{pause; present A then emit B end} ||
\]

A A

B
Bidirectional Communication

Processes can communicate back and forth in the same cycle

```plaintext
[  
    pause; emit A;
    present B then emit C end;
    pause; emit A
  ]
 ||
pause; present A then emit B end
```
Concurrency and Determinism

Signals are the only way for concurrent processes to communicate.

Esterel does have variables, but they cannot be shared.

Signal coherence rules ensure deterministic behavior.

Language semantics clearly defines who must communicate with whom when.
The Await Statement

The await statement waits for a particular cycle await S waits for the next cycle in which S is present

```plaintext
[  emit A ; pause ; pause; emit A ||
    await A; emit B
]
```

A A

B
The Await Statement

Await normally waits for a cycle before beginning to check

```
await immediate also checks the initial cycle
[
  emit A; pause; pause; emit A
||
  await immediate A; emit B
]
```

A A
B B
Loops

Esterel has an infinite loop statement

Rule: loop body cannot terminate instantly

Needs at least one pause, await, etc.

Can’t do an infinite amount of work in a single cycle

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

A A A A A
B B B B B
Loops and Synchronization

Instantaneous nature of loops plus await provide very powerful synchronization mechanisms

```plaintext
loop
    await 60 S;
    emit M
end
```

S S S S S S

M M

1 ... 59 60 61 ... 120
Preemption

Often want to stop doing something and start doing something else

E.g., Ctrl-C in Unix: stop the currently-running program

Esterel has many constructs for handling preemption
The Abort Statement

Basic preemption mechanism

General form:

```plaintext
abort
    statement
when  condition
```

Runs `statement` to completion. If `condition` ever holds, `abort` terminates immediately.
The Abort Statement

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

Normal Termination

Aborted termination

Aborted termination; emit A preempted

Normal Termination

B not checked in first cycle (like await)
Strong vs. Weak Preemption

Strong preemption:

- The body does not run when the preemption condition holds
- The previous example illustrated strong preemption

Weak preemption:

- The body is allowed to run even when the preemption condition holds, but is terminated thereafter
- “weak abort” implements this in Esterel
Strong vs. Weak Abort

Strong abort
emit A does not run

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

Weak abort
emit A runs

weak abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
Strong vs. Weak Preemption

Important distinction

Something may not cause its own strong preemption

Erroneous

abort
    pause; emit A
when A

OK

weak abort
    pause; emit A
when A
The Trap Statement

Esterel provides an exception facility for weak preemption

Interacts nicely with concurrency

Rule: outermost trap takes precedence
The Trap Statement

```
trap T in
  [pause;
   emit A;
   pause;
   exit T
  ]
||
 await B;
 emit C
]
end trap;
emit D
```

- Normal termination from first process
- Emit C also runs
- Second process allowed to run even though first process has exited
Nested Traps

trap T1 in
trap T2 in
[
  exit T1
||
  exit T2
]
end;
emit A
end;
emit B

Outer trap takes precedence; control transferred directly to the outer trap statement. emit A not allowed to run.

B
The Suspend Statement

Preemption (abort, trap) terminate something, but what if you want to resume it later?

Like the unix Ctrl-Z

Esterel’s suspend statement pauses the execution of a group of statements

Only strong preemption: statement does not run when condition holds
The Suspend Statement

suspend
  loop
    emit A; pause; pause
  end
when B

A A B A B A

B prevents A from being emitted here; resumed next cycle
B delays emission of A by one cycle
Causality

Unfortunate side-effect of instantaneous communication coupled with the single valued signal rule

Easy to write contradictory programs, e.g.,

```plaintext
present A else emit A end

abort pause; emit A when A

present A then nothing end; emit A
```

These sorts of programs are erroneous; the Esterel compiler refuses to compile them.
Causality

Can be very complicated because of instantaneous communication

For example, this is also erroneous

```plaintext
abort
  pause;
  emit B
when A
||
  pause;
  present B then emit A end
```

Emission of B indirectly causes emission of A
Compiling Esterel

Semantics of the language are formally defined and deterministic.

It is the responsibility of the compiler to ensure the generated executable behaves correctly w.r.t. the semantics.

Challenging for Esterel
Compilation Challenges

• Concurrency
• Interaction between exceptions and concurrency
• Preemption
• Resumption (pause, await, etc.)
• Checking causality
• Reincarnation
  Loop restriction prevents most statements from executing more than once in a cycle
  Complex interaction between concurrency, traps, and loops allows certain statements to execute twice or more
Automata-Based Compilation

Key insight: Esterel is a finite-state language

Each state is a set of program counter values where the program has paused between cycles

Signals are not part of these states because they do not hold their values between cycles

Esterel has variables, but these are not considered part of the state
void tick() {
    static int s = 0;
    A = B = 0;

    switch (s) {
    case 0:
        A = 1;
        s = 1;
        break;
    case 1:
        if (C) {
            B = 1; s = 0;
        }
        break;
    }
}
Automata Compiler Example

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

switch (s) {
case 0:
  A=1;
  B=1;
  s=1;
  break;
case 1:
  if (C) {
    D=1;
    if (E) B=1;
    s=2;
  }
  break;
case 2:
}
Automata Compilation Considered

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 states

- “these inputs are mutually exclusive”
  \[
  \text{relation } A \# B \# C;
  \]
- “if this input arrives, this one does, too”
  \[
  \text{relation } D \Rightarrow E;
  \]
Automata Compilation

Not practical for large programs

Theoretically interesting, but don’t work for most programs longer than 1000 lines

All other techniques produce slower code
Netlist-Based Compilation

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

Generate code that simulates the netlist
emit A; emit B; await C;
emit D; present E then emit B end
Netlist Compilation Considered

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 irrelevant values
- 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

Currently the most widely-used technique
Control-Flow Graph-Based

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

| loop
| present B then
| emit C end;
| pause
| 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;
      }
    }
}
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 Threads

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
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
Add Code Between 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
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

\begin{itemize}
\item s = 2
\item s = 1
\end{itemize}
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
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
Run First Node
Run First Part of Left Thread
Context Switch
Run Right Thread
Context Switch
Finish Left Thread
Completed Example
Control-flow Approach Considered

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 the others accept.

Only implementation hiding within Synopsys’ CoCentric System Studio. Will probably never be used industrially.

What To Understand About Esterel

Synchronous model of time

- Time divided into sequence of discrete instants
- Instructions either run and terminate in the same instant or explicitly in later instants

Idea of signals and broadcast

- “Variables” that take exactly one value each instant and don’t persist
- Coherence rule: all writers run before any readers

Causality Issues

- Contradictory programs
- How Esterel decides whether a program is correct
What To Understand About 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