Esterel and the Synchronous Approach

Stephen A. Edwards

Columbia University

April 2009
The Big Picture

- The Digital Approach
  Discretize value to *completely eliminate* noise

- The Synchronous Approach
  Discretize time to *completely eliminate* noise
Digital Is Everywhere

- DNA
- Written Language
- Cellular Structure
- Spoken Language
- Atomic Structure
Synchrony Is Everywhere

Clocks

Railroads

Conductors

Day and Night

Seasons
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
Timeline

1983  How do you program an infinitely fast computer?
1984  First semantics, LISP-based V2 compiler
1988  Better semantics, Efficient V3 compiler
1990  First hardware synthesis to FPGAs (DEC)
1992  BDD-based verification facilities (Dassault)
1995  Causality and cyclic circuits
1997  Sequential optimization
1999  V7 specification started
2001  Esterel Technologies founded
2003  Esterel V7 compiler released
2005  First silicon produced by Esterel V7
2007  Fast code generation, System C backend
2008  IEEE Standardization process started
Standardization Effort

Project P1778 approved by IEEE on March 20th, 2007

G. Berry (Chair, Esterel)  S. Edwards (Vice-Chair, Columbia)
S. Dissoubray (Secretary, Esterel)  C. André (U. Nice)
E. Badi (TI)  R. de Simone (INRIA)
G. Clavé (TI)  K. Schneider (U. Kaiserslautern)
L. Blanc (TI)  R. von Hanxleden (U. Kiel)
B. Bentayibi (ST Micro)  L. Zaffallon (EIG Geneva)
J-P. Cousin (ST Micro)
R. Bernhard (Orange)
M. Duranton (NXP)
M. Kishinevsky (Intel)
M. Perreaut (Esterel)
S. Ramesh (GM)
S. Ramesh (GM)
R.K. Shyamasundar (IBM)
O. Tardieu (IBM)
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.
The output $O$ should occur when inputs $A$ and $B$ have both arrived. The $R$ input should restart this behavior.
The Esterel Version

**module** ABRO :
**input** A, B, R;
**output** O;

**loop**
[  
  **await** A
  ||
  **await** B
] ;

**emit** O

**each** R

**end module**

Esterel programs consist of modules

Interface comprising input and output signals

**loop...each** for reset behavior

**await** waits for the next cycle when a condition is true

**emit** makes a signal present

|| runs statements in parallel; waits for all to terminate
The Big Ideas of Esterel

Global Clock

“Combinational” statements
(e.g., emit, if)

Sequential statements
(e.g., pause)
Processor

SoC Power Management

Serial ATA link layer protocol

High throughput DMA for video processor

Flash card (SD/MMC) controller

Memory architectures, including caches

(From website of Esterel EDA Technologies)
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

Limited support for data, although much better in V7 than V5

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**.
Signals

Esterel programs communicate through signals

```plaintext
input Req, Ack; // Control signals
output Write;

input Addr : unsigned<[16]>; // Data signal

output Dout when Write : unsigned<[8]>; // Controlled signal
```

<table>
<thead>
<tr>
<th>Kind</th>
<th>Status</th>
<th>Value</th>
<th>Usage</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>✓</td>
<td></td>
<td>Control/strobe</td>
<td>reset, req, ack</td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td>✓</td>
<td>Data</td>
<td>address</td>
</tr>
<tr>
<td>Controlled</td>
<td>✓</td>
<td>✓</td>
<td>Strobed data</td>
<td>dout</td>
</tr>
</tbody>
</table>
**Declarations: Data, Interfaces, and Modules**

```pseudocode
data SizeData : // types, constants, functions, and procedures
    constant N : unsigned = 8;
end data

interface DoubleIntf : // adds inputs and outputs
    extends data SizeData;
    input I : signed<N>;
    output O : signed<2*N>;
end interface

module Double : // adds behavior
    extends interface DoubleIntf;
    every I do
        emit ?O <= 2 * ?I
    end every
end module
```
Emitting Signals

Emit: sets the status to “present” for the current cycle

```plaintext
emit Req; // Set status
emit ?DataOut <= 253; // Set status & value
emit ?ConfReg <= 'b100_010_11'; // bitvector
emit ?ConfRegArray[0] <= ?Conf; // use value of another signal
```

Sustain: sets the status to “present” in current and future cycles

```plaintext
sustain Ack; // Ack now and forever
sustain {
    ?Addr <= pre(?Addr) + 1 mod MAX, // Count and wrap around
    Write
}
```
Waiting

Pause: wait for a cycle

```
pause; // Wait for one cycle
```

Await \( expr \): wait for next cycle in which \( expr \) is true

```
await FifoFull;
await Req and (?Addr > 0x00FF); // next valid request
await 3 tick; // like “pause; pause; pause”
```
Esterel (finally!) has the usual if-then-else construct and variants.

```plaintext
if S and (?S > 0) then // Statement
    emit ?DataValid <= ?S
else
    await Ready
end if

emit {
    if S and (?S > 0) then // Within an emit
        ?DataValid <= ?S
    else
        Grant
    end if
} // In a case list

emit {
    ?DataValid <= ?S if S and (?S > 0) // In a case list
    | 0
} // In a case list
```
module Example1:
output A, B, C;
emit A;
if A then emit B end;
pause;
emit C
end module
Signal Coherence Rules

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

All writers run before any readers do

Thus

\[
\text{if } A \text{ else emit } A \text{ end}
\]

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

\[
\begin{align*}
\text{every} 60 \ S & \text{ do } // \text{ invokes its body every 60th S} \\
\text{emit} \ M & \text{ // takes no time (cycles)} \\
\text{end}
\end{align*}
\]

\[
\begin{array}{ccccccc}
S & S & S & S & S & S & S \\
M & M & & & & & \\
1 & \cdots & 59 & 60 & 61 & \cdots & 120
\end{array}
\]
Groups of statements separated by run concurrently and terminate when all groups have terminated

\[
[\
  \text{emit } A; \text{ pause; emit } B; \\
  \text{pause; emit } C; \text{ pause; emit } D \\
]\
\text{emit } E
\]
A signal emitted in a cycle is visible immediately

\[
\begin{align*}
&\text{pause}; \text{emit A; pause; emit A} \\
&\text{pause; if A then emit B end}
\end{align*}
\]
Processes can communicate back and forth in the same cycle

```plaintext
[ pause;
  emit A; if B then emit C end;
  pause;
  emit A
]
```

A A
B
C
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 waits for a particular cycle

await S waits for the next cycle in which S is present

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

A A
B
Await normally waits for a cycle before beginning to check

`await immediate` also checks the initial cycle

```javascript
[ emit A; pause; pause; emit A
  ||
  await immediate A; emit B
]
```

A  A
B
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

---
Instantaneous nature of loops plus await provide very powerful synchronization mechanisms

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

```
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:

```
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

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

**Weak abort**
emit A runs

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

B

C

A

B

C
Important distinction

Something may not cause its own strong preemption

Erroneous

\begin{itemize}
\item abort
\item pause; emit A
\item when A
\end{itemize}

OK

\begin{itemize}
\item weak abort
\item pause; emit A
\item when A
\end{itemize}
Esterel provides an exception facility for weak preemption

Interacts nicely with concurrency

Rule: outermost trap takes precedence
The Trap Statement

\[
\text{trap } T \text{ in} \\
| \text{pause;} \\
| \text{emit } A; \\
| \text{pause;} \\
| \text{exit } T \\
| | \\
| \text{await } B; \\
| \text{emit } C \\
| \text{end trap;} \\
\text{emit } D
\]

A \quad D
---
Normal termination from first process
A
B
C \quad D
---
\text{emit } C \text{ also runs}
A \quad B
C
D
---
Second process allowed to run even though first process has exited
A \quad B
C
D
---

Normal termination from first process

\text{emit } C \text{ also runs}

Second process allowed to run even though first process has exited
Nested Traps

\[
\begin{align*}
\text{outer trap:} & \quad \text{emit } A \\
\text{inner trap:} & \quad \text{emit } B
\end{align*}
\]

Outer trap takes precedence; control transferred directly to the outer trap statement.

emit A not allowed to run.

B
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

suspender loop
  emit A; pause; pause
end
when B

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.,

```
if A else emit A
end

abort
    pause; emit A
when A

if 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;
  if B then emit A end

Emission of B indirectly causes emission of A
```
Definition has evolved since first version of the language

Original compiler had concept of “potentials”

Static concept: at a particular program point, which signals could be emitted along any path from that point

Latest definition based on “constructive causality”

Dynamic concept: whether there’s a “guess-free proof” that concludes a signal is absent
emit A;
if B then emit C end;
if A else emit B end;

Considered erroneous under the original compiler

After emit A runs, there’s a static path to emit B Therefore, the value of B cannot be decided yet

Execution procedure deadlocks: program is bad
Causality Example

```
emit A;
if B then emit C end;
if A else emit B end;
```

Considered acceptable to the latest compiler

After `emit A` runs, it is clear that B cannot be emitted because A’s presence runs the “then” branch of the second present

B declared absent, both present statements run
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
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
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
People Counter Example

Construct an Esterel program that counts the number of people in a room. People enter the room from one door with a photocell that changes from 0 to 1 when the light is interrupted, and leave from a second door with a similar photocell. These inputs may be true for more than one clock cycle.

The two photocell inputs are called ENTER and LEAVE. There are two outputs: EMPTY and FULL, which are present when the room is empty and contains three people respectively.

Conditioner detects rising edges of signal from photocell.
Counter tracks number of people in the room.
Implementing the Conditioner

```
module Conditioner:
  input A;
  output Y;

  loop
    await A; emit Y;
    await [ not A ];
  end

end module
```
Testing the Conditioner

```
# esterel -simul cond.strl
# gcc -o cond cond.c -lcsimul # may need -L
# ./cond
Conditioner> ;
-- Output: Conditioner> A;  # Rising edge
-- Output: Y
Conditioner> A;  # Doesn’t generate a pulse
-- Output: Conditioner> ;  # Reset
-- Output: Conditioner> A;  # Another rising edge
-- Output: Y
Conditioner> ;
-- Output: Conditioner> A;
-- Output: Y
```
module Counter:
input ADD, SUB;
output FULL, EMPTY;

var count := 0 : integer in
loop
    if ADD and count < 3 then
        count := count + 1 end;
    if SUB and count > 0 then
        count := count - 1 end;
    if count = 0 then emit EMPTY end;
    if count = 3 then emit FULL end;
    pause
end
end
end module
Testing the Counter

Counter> ;
   -- Output: EMPTY
Counter> ADD SUB;
   -- Output: EMPTY
Counter> ADD;
   -- Output:
Counter> SUB;
   -- Output: EMPTY
Counter> ADD;
   -- Output:
Counter> ADD;
   -- Output:
Counter> ADD;
   -- Output:
Counter> ADD;
   -- Output: FULL
Counter> ADD SUB;
   -- Output: # Oops: should remain FULL
module Counter:
input ADD, SUB;
output FULL, EMPTY;

var c := 0 : integer in
  loop
    if ADD then
      if not SUB and c < 3 then
        c := c + 1
      end
    else
      if SUB and c > 0 then
        c := c - 1
      end;
    end;
    if c = 0 then emit EMPTY end;
    if c = 3 then emit FULL end;
    pause
  end
end
end module
Testing the second counter

Counter> ;
-- Output: EMPTY
Counter> ADD SUB;
-- Output: EMPTY
Counter> ADD SUB;
-- Output: EMPTY
Counter> ADD;
-- Output:
Counter> ADD;
-- Output:
Counter> ADD;
-- Output: FULL
Counter> ADD SUB;
-- Output: FULL  # Working
Counter> ADD SUB;
-- Output: FULL
Counter> SUB;
-- Output:
Counter> SUB;
-- Output:
Counter> SUB;
-- Output: EMPTY
Counter> SUB;
-- Output: EMPTY
module PeopleCounter:
input ENTER, LEAVE;
output EMPTY, FULL;

signal ADD, SUB in
  run Conditioner[signal ENTER / A, ADD / Y]
  ||
  run Conditioner[signal LEAVE / A, SUB / Y]
  ||
  run Counter
end

end module
Design a vending machine controller that dispenses gum once. Two inputs, N and D, are present when a nickel and dime have been inserted, and a single output, GUM, should be present for a single cycle when the machine has been given fifteen cents. No change is returned.

N = or D = or GUM =

module Vending:
input N, D;
output GUM;

loop
  var m := 0 : integer in
  trap WAIT in
    loop
      if N then m := m + 5; end;
      if D then m := m + 10; end;
      if m >= 15 then exit WAIT end;
    pause
    end
  end;
  emit GUM; pause
end
end module
Alternative Solution

```
loop
  await
    case immediate N do await
    case N do await
    case immediate D do nothing
    end
    case immediate D do nothing
    end
    case immediate D do await
    case immediate N do nothing
    case D do nothing
    end
  end
end;
emit GUM; pause
end
```
Construct an Esterel program that controls the turn signals of a 1965 Ford Thunderbird.

Tail Light Behavior
There are three inputs, LEFT, RIGHT, and HAZ, that initiate the sequences, and six outputs, LA, LB, LC, RA, RB, and RC. The flashing sequence is

<table>
<thead>
<tr>
<th>LC</th>
<th>LB</th>
<th>LA</th>
<th>step</th>
<th>RA</th>
<th>RB</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
module Lights:
output A, B, C;

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

end module
module Thunderbird:
input LEFT, RIGHT, HAZ;
output LA, LB, LC, RA, RB, RC;

...
loop
  await
  case immediate HAZ do
    abort
    run Lights[signal LA/A, LB/B, LC/C]
  | |
    run Lights[signal RA/A, RB/B, RC/C]
    when [not HAZ]
  case immediate LEFT do
    abort
    run Lights[signal LA/A, LB/B, LC/C]
    when [not LEFT]
  case immediate RIGHT do
    abort
    run Lights[signal RA/A, RB/B, RC/C]
    when [not RIGHT]
  end
end
I choose to use Esterel’s innate ability to control the execution of processes, producing succinct easy-to-understand source but a somewhat larger executable.

An alternative: Use signals to control the execution of two processes, one for the left lights, one for the right.

A challenge: synchronizing hazards.

Most communication signals can be either level- or edge-sensitive. Control can be done explicitly, or implicitly through signals.
Traffic-Light Controller Example

This controls a traffic light at the intersection of a busy highway and a farm road. Normally, the highway light is green but if a sensor detects a car on the farm road, the highway light turns yellow then red. The farm road light then turns green until there are no cars or after a long timeout. Then, the farm road light turns yellow then red, and the highway light returns to green. The inputs to the machine are the car sensor C, a short timeout signal S, and a long timeout signal L. The outputs are a timer start signal R, and the colors of the highway and farm road lights.

module Fsm:

input C, L, S;
output R;
output HG, HY, FG, FY;

loop
  emit HG; emit R; await [C and L];
  emit HY; emit R; await S;
  emit FG; emit R; await [not C or L];
  emit FY; emit R; await S;
end

end module
The Traffic Light Controller

module Timer:
input R, SEC;
output L, S;

loop
  weak abort
  await 3 SEC;
  [sustain S
  | | await 5 SEC;
  | | sustain L
  ]
  when R;
end
end module
module TLC:
  input C, SEC;
  output HG, HY, FG, FY;

signal S, L, S in
  run Fsm
  run Timer
end
end module