The Synchronous Language Esterel

Prof. Stephen A. Edwards

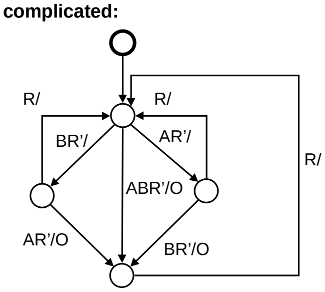
A Simple Example

- 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

- **Fairly complicated:**



The Esterel Version

- **Much simpler**
  - **Ideas of signal, wait, reset part of the language**

```
module ABRO
input A, B, R;
output O;

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

end module
```

Means the same thing as the FSM

The Esterel Version

Esterel programs built from modules

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

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

end module
```

Each module has an interface of input and output signals

The Esterel Version

loop … each statement implements the reset

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

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

end module
```

await waits for the next cycle in which its signal is present

|| operator means run the two awaits in parallel

The Esterel Version

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

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

end module
```

Parallel statements terminate immediately when all branches have

Emit O makes signal O present when A and B have both arrived

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
  - Those that take "zero time" (execute and terminate in same instant, e.g., emit)
  - Those that delay for a prescribed number of cycles (e.g., await)

Uses of Esterel

- Wristwatch
  - Canonical example
  - Reactive, synchronous, hard real-time
- Controllers
  - 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 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 implementation in both hardware and software

Disadvantages of Esterel

- Finite-state nature of the language limits flexibility
  - No dynamic memory allocation
  - No dynamic creation of processes
- Virtually nonexistent support for handling data
- Really suited for 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.

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

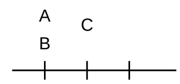
Basic Esterel Statements

- **emit S**
  - **Make signal S present in the current instant**
  - **A signal is absent unless it is emitted**
- **pause**
  - **Stop and resume after the next cycle after the pause**
- **present S then *stmt1* else *stmt1* end**
  - **If signal S is present in the current instant, immediately run stmt1, otherwise run stmt2**

Basic Esterel Statements

- **Thus**

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



- **Makes A & B present the first instant, C present the second**

Signal Coherence Rules

- **Each signal is only present or absent in a cycle, never both**
- **All writers run before any readers do**
- **Thus**

present A else
emit A
end

is an erroneous program

Advantage of Synchrony

- **Easy to control time**
- **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

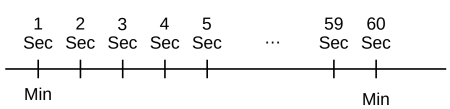
- **This guarantees every 60th Sec a "Min" signal is emitted**

every 60 Sec do
emit Min
end

"every" invokes its body every 60 Sec exactly

emit takes no time

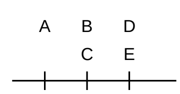
- **Timing diagram:**



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



Communication Is Instantaneous

- A signal emitted in a cycle is visible immediately

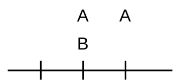
[

pause; emit A; pause; emit A

||

pause; present A then emit B end

]



Bidirectional Communication

- Processes can communicate back and forth in the same cycle

[

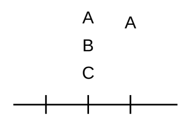
pause; emit A; present B then emit C end;

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

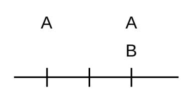
[

emit A ; pause ; pause; emit A

||

await A; emit B

]



The Await Statement

- Await normally waits for a cycle before beginning to check
- await immediate also checks the initial cycle

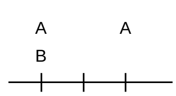
[

emit A ; pause ; pause; emit A

||

await immediate A; emit B

]



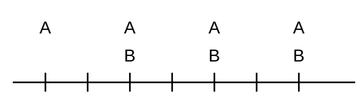
Loops

- 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

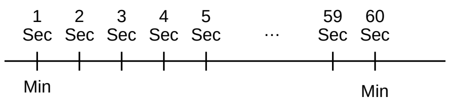
end



Loops and Synchronization

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

loop
await 60 Sec;
emit Min
end



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

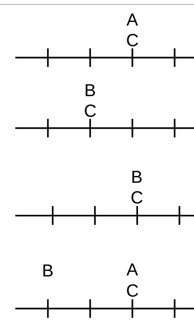
- Basic preemption mechanism
- General form:

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

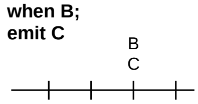


Strong vs. Weak Preemption

- 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

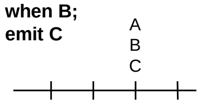
Strong vs. Weak Abort

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



Strong abort: emit A not allowed to run

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



Weak abort: emit A allowed to run, body terminated afterwards

Strong vs. Weak Preemption

- **Important distinction**
- **Something cannot cause its own strong preemption**

```
abort
  emit A
when A
```

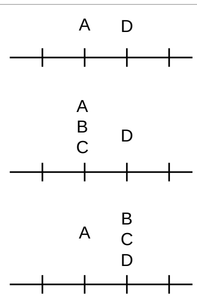
- **Erroneous: if body runs then it could not have**

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

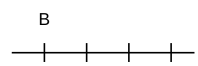
Second process allowed to run even though first process has exited

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 outer trap statement.

emit A not allowed to run

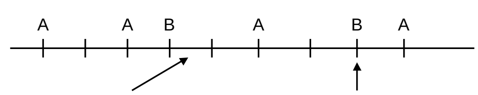


The Suspend Statement

- **Preemption (abort, trap) terminate something, but what if you want to pause it?**
- **Like the unix Ctrl-Z**
- **Esterel's suspend statement pauses the execution of a group of statements**
- **Strong preemption: statement does not run when condition holds**

The Suspend Statement

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



B delays emission of A by one cycle

B prevents A from being emitted; resumed the next cycle

Causality

- Unfortunate side-effect of instantaneous communication coupled with the single valued signal rule
- Easy to write contradictory programs, e.g.,

- present A else emit A end
- abort emit A when A
- present A then nothing end; emit A

- These sorts of programs are erroneous and flagged by the Esterel compiler as incorrect

Causality

- Can be very complicated because of instantaneous communication
- For example: this is also erroneous

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

Emission of B indirectly causes emission of A

Causality

- 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

Causality Example

- Consider the following program

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

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 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

- First 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

Automata-based Compilation

- First compiler simulated an Esterel program in every possible state and generated code for each one
- For example

Automata Example

```
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 = 3; }
  else { state = 1; }
}
```

First state: A, B, emitted, go to second

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

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 state count
  - “these inputs are mutually exclusive,”
  - “if this input arrives, this one does, too”

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

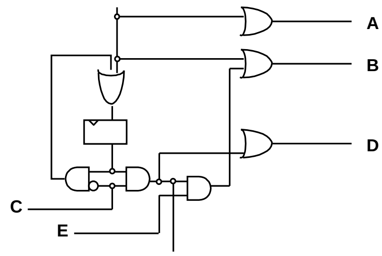
Netlist-Based Compilation

- 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
- Generate code that simulates the netlist

Netlist Example

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 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
- 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

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.
- See my recent IEEE Transactions on Computer-Aided Design paper for details

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
- Compilers, documentation, etc. available from www.esterel.org