

Esterel



**From an “Introduction to Esterel”,
Girish Keshav Palshikar,
Embedded Systems Programming,
November 2001**

Esterel

- ◆ Esterel is a system-design language that can be used to generate complex state machines automatically. This article offers an overview of the syntax and usage.
- ◆ Esterel-invented by G. Berry in INRIA, France-belongs to a family of formal specification languages specialized for reactive systems; other members of this family include StateCharts, Lustre, Signal, and SL.
- ◆ More information about Esterel and its history can be found at <http://www-sop.inria.fr/esterel.org/>.

Reactive Systems

- ◆ A reactive system is one that is in continuous interaction with its environment. Most real-time, embedded systems are reactive. In addition, operating systems, networking protocols, VLSI chips, and even a graphical user interface can be considered partly reactive.
- ◆ The behaviour of a reactive system can be thought of as a black box that continuously receives some input events and reacts by producing some output events. This output may in turn affect the production of later input events by the environment.

- ◆ The task of specifying the behaviour of a reactive system is akin to that of specifying relationships between the input and output events.
- ◆ However, such a task is complicated by the fact that several input events may happen simultaneously and that after the system receives an input event, it may take a finite nonzero amount of time to produce its response in the form of an output event

Synchrony hypothesis

- ◆ To simplify the behavioral specifications of reactive systems, Esterel makes an assumption called the synchrony hypothesis.
- ◆ The synchrony hypothesis says that the underlying machine is infinitely fast and, hence, the reaction of the system to an input event is instantaneous. As a consequence, the reaction intervals are reduced to reaction instants; therefore, the reactions do not overlap with each other.
- ◆ This assumption is also called the hypothesis of atomic reactions. The system is then active only at each instant. "In between" the instants, it is idle and awaiting input events.

Determinism

- ◆ Esterel also assumes that the systems are deterministic. Informally, a non-deterministic system does not have a unique response to a given input event; instead, it chooses its response to the input event from a set of possible responses and an external observer has no way to consistently predict the response that will be chosen by the system.
- ◆ All Esterel statements and constructs are guaranteed to be deterministic. There is no way to introduce any nondeterministic behavior in an Esterel program. The Esterel compiler checks the given program and ensures that it is deterministic. This assumption of determinism greatly simplifies the behavioral specifications.

Parallelism

- ◆ Esterel provides the operator \parallel for a parallel composition of its programs. If **P1** and **P2** are two Esterel programs then **P1** \parallel **P2** is also an Esterel program, with the following characteristics:
 - All inputs received from the environment are available to both **P1** and **P2**.
 - All outputs generated by **P1** (or **P2**) are available in the same instant to **P2** (or **P1**).
 - Both **P1** and **P2** continue execution in parallel and the statement **P1** \parallel **P2** terminates when both **P1** and **P2** terminate.
 - No data or variables can be shared by **P1** and **P2**.

Modules

- ◆ A module in Esterel defines a (reusable) unit of behavior. A module is somewhat like a subroutine with its own local data and behavior. However, modules are quite different from subroutines in the manner of their usage.
- ◆ There is no "module call" facility in Esterel. A module is used like a macro in C; using a module simply means an inline substitution of its entire text at the place of "call."
- ◆ There are other significant differences between a module and a subroutine. For example, no global data is shared by modules, recursive module definitions are not possible, and so on.

- ◆ A module has an interface and a body defining the behavior. Following is the Esterel syntax to define a module.

```
% this is a line comment  
module module-name :  
  declarations and compiler directives  
  % signals, local variables etc.  
  body  
% end of module body
```

- ◆ Each module can be thought of as an independent Esterel program and Esterel provides several constructs to combine modules to build larger reactive systems. In fact, an Esterel program is typically an interconnected network of modules.

- ◆ Each module can be thought of as an independent Esterel program and Esterel provides several constructs to combine modules to build larger reactive systems. In fact, an Esterel program is typically an interconnected network of modules.
- ◆ Esterel does not have any notion of global data or global memory shared by all modules. However, each module can define its local data in terms of variables. Each Esterel statement has associated with it a precise definition about its duration in number of instants; for example, emit terminates instantly and await terminates only when the signal waited for becomes available.

Signals

- ◆ A reactive system reacts to its input events by producing output events. In general, a reactive system needs to interact with its environment; component subsystems of a reactive system also interact with each other. Esterel provides a simple logical concept called a signal to model many such events and interactions
- ◆ Classification attributes for a signal:
 - Visibility: interface signal vs. local signal
 - Information contained in a signal: pure signal vs. valued signal
 - Accessibility of interface signals: input, output, inputoutput, sensor

- ◆ An important concept about signals is that of an occurrence. An occurrence simply refers to one or more happenings of a signal. To deal with an occurrence of the form $\mathbf{N S}$, where \mathbf{N} is an expression that evaluates to a positive integer and \mathbf{S} is an input (or inputoutput) signal, the Esterel system checks for the presence of the signal \mathbf{S} in successive instants and increments the internal counter whenever \mathbf{S} is present; the occurrence is complete in the instant when the internal counter reaches the value \mathbf{N} . Thus an occurrence has a positive duration (in the number of instants).

Istantaneous Broadcast

- ◆ Esterel incorporates an instantaneous broadcast mechanism for signal reception and transmission. This means that a signal cannot have any destination specified; all signals are broadcast and any module may listen to and read an emitted signal. Also, signals do not have any unique identifier.
- ◆ In Esterel, signals are emitted and used much like bus signals in a hardware interconnection bus. Any signal emitted makes a "wire" for that signal come alive with the information contained in that signal. Any module (including the module that emitted the signal) can tap this wire and read the emitted signal. Thus, no copies are made of an emitted signal.

Istantaneous Broadcast

(cont)

- ◆ After the current reaction instant is over, the "bus" is "reset" (that is, cleared of all previous signals) and waits for any further input signals to be put on the wire by the environment. That is, the signals are available only in the current instant. The system's reaction to those input signals starts the next reaction instant.
- ◆ The bus analogy is useful to illustrate another important fact about Esterel signals. A signal **S** emitted by a module **M** at a reaction instant **t** is made available to all other modules in the same reaction instant **t**. The emission of signal **S** constitutes part of the input event at **t** and the reaction instant **t** is not completed until some module reacts to the presently available signal **S**.

Example

- ◆ A Vending Machine...

- ◆ **module VM1 :**

input COIN, TEA, COFFEE;

output SERVE_TEA, SERVE_COFFEE;

relation COIN # TEA # COFFEE;

loop

await COIN;

await

case TEA do emit SERVE_TEA;

case COFFEE do emit SERVE_COFFEE;

end await;

end loop;

More constructs - Relation

- ◆ The relation directive is used to describe restrictions on the possible combinations of the input signals present in an instant. The relation directive has the form:

relation Master-signal-name => Slave-signal-name;

- ◆ and indicates that in any instant where the input signal having the name **Master-signal-name** is present, the input signal having the name **Slave-signal-name** should also be present.

More constructs - Relation

- ◆ Esterel has another kind of relation directive, which is used to declare signals that are pair-wise incompatible. This directive has the form:

**relation Signal-name1 # Signal-name2 # ... #
Signal-namen;**

- ◆ and states that in any instant, at most one of the n input signals **Signal-name1**, **Signal-name2**, ..., **Signal-namen** may be present; that is, no two or more of these n input signals can be simultaneously present in any instant. Use of the relation directive reduces the size of the automaton generated from an Esterel specification.

More constructs - Loop

- ◆ The classical Infinite Loop statement has the format:

loop Body end loop;

- ◆ and it forever executes the enclosed Esterel statements.
The only restriction is that the **Body** must not terminate in the same instant that it started.

More constructs – Multiple await statement

- ◆ The multiple await statement has the form:

```
await  
  case Occurrence1 do Body1  
  case Occurrence2 do Body2  
  ...  
  case Occurrencen do Bodyn  
end await;
```

- ◆ This statement is used to wait for any of the n occurrences: **Occurrence₁, Occurrence₂, ..., Occurrence_n**. If only one of the occurrences, **Occurrence_i**, is complete in the current instant, then the execution of the corresponding case alternative behavior (given by the body **Body_i** of Esterel statements) is started and all other awaits are terminated.

More constructs – Multiple await statement

- ◆ If more than one occurrence is complete in the current input instant, then the case alternative for the occurrence that textually occurs first is executed and the other occurrences are ignored (that is, the behavior for them not executed). This rule makes the multiple await statement deterministic. This await statement terminates whenever the body **Body_i**, whose execution was started, terminates.

More constructs – Await occurrence / Emit signal / ;

- ◆ The statement **await Occurrence**; waits till the **Occurrence** is complete and terminates when it happens.
- ◆ The statement **emit Signal-name**; is used to broadcast the given output signal given by **Signal-name**; the **emit** statement executes instantaneously. That is, the emitted signal is also made available as part of the present instant.
- ◆ The sequential composition **operator** ; terminates an Esterel statement and binds two Esterel statements in a sequential order for execution. Note that the semicolon does not denote an empty statement (unlike in C); thus ;; is illegal.

More constructs – Nothing / Halt

- ◆ The nothing; statement is a null statement that does nothing and terminates instantaneously.
- ◆ In addition, Esterel has the halt; statement, which also does nothing but never terminates. These two statements are surprisingly effective in many situations.

More constructs – Signal declaration

- ◆ The signal statement is used to declare the local signals used within the module and its sub-modules and parts. This statement has the format:

**signal Signal-decl₁, Signal-decl₂, ..., Signal-decl_n in
Body end;**

- ◆ The above statement declares n local signals which are available only within the **Body** and not outside. For pure signals, **Signal-decl_i** contains only a signal name. For a valued signal, the declaration **Signal-decl_i** has the form **Signal-name_i : Signal-type_i**.

More constructs – Every

- ◆ The every statement is the **Periodic Restart** construct in Esterel. The statement:

every Occurrence do Body end every;

- ◆ has the following meaning: whenever the **Occurrence** is complete in the present instant, the execution of **Body** is started and another wait starts for the completion of the next **Occurrence**. If the next **Occurrence** is complete before the execution of **Body** is completed, the current execution of **Body** is terminated and a fresh execution of **Body** is started. The every tick do **Body** end every; statement allows some actions to be performed at every instant.

More constructs – Parallel composition

- ◆ The **Parallel Composition construct** `||` has the following format:

Body₁ || Body₂ || ... || Body_n

- ◆ where each **Body_i** is a group of Esterel instructions. The execution of all the bodies starts at the same instant and continues in parallel. The entire construct terminates only after the execution of each body terminates; the execution of all bodies need not terminate at the same instant.

More constructs - If

- ◆ The If statement has the form:

if boolean-expression then Body₁ else Body₂ end if;

- ◆ When the control comes to this statement, the **boolean-expression** is evaluated. If the expression evaluates to true in the current instant then the execution of **Body₁** is started; otherwise, the execution of **Body₂** is started. In the if-then-else statement, either the **then Body₁** part or the **else Body₂** part can be omitted (but not both).

More constructs – Valued signal

- ◆ If S is a valued signal which is available in the current instant, then the expression $?S$ returns the value of this signal in the current instant.
- ◆ Thus $?$ is the signal value extraction operator. $?$ cannot be used on pure signals. Note that $?$ is a unary operator. The type of the value returned by the expression $?$ is the same as the type of signal $?$.

More constructs - present

- ◆ The present statement is the signal testing construct. The statement:

present Signal-name then Body₁ else Body₂ end present;

- ◆ has the following meaning. If the signal having the name Signal-name is present in the current instant then start the execution of the statement body **Body₁**; otherwise, start the execution of the statement body **Body₂**. The entire statement terminates when the execution of **Body₁** or **Body₂** terminates (whichever body was started). In the present statement, either the then **Body₁** part or the else **Body₂** part can be omitted (but not both).

More constructs – Do watching

- ◆ The **do-watching-timeout** is a basic and important temporal statement in Esterel. This statement has the following format:

**do Body₁ watching
Occurrence timeout Body₂ end;**

- ◆ The timeout clause is optional. Whenever the control arrives at this statement, it executes as follows:

More constructs – Do watching

if the Occurrence is complete in the current instant then
 if Body₂ is given then
 start executing Body₂
 else pass control to the following statement
else start executing Body₁.
if Occurrence becomes complete before Body₁ has finished then
 immediately abort the execution of Body₁
 if Body₂ is given then
 start executing Body₂
 else pass control to the following statement
else Body₁ has finished execution but Occurrence is not complete yet
 pass control to the following statement

Temperature Controller

```
module temp_controller :
input    TEMP : integer, SAMPLE_TIME, DELTA_T;
output  B1_ON, B1_OFF, B2_ON, B2_OFF;
relation SAMPLE_TIME => TEMP;

signal HIGH, LOW in
  every SAMPLE_TIME do
    present TEMP else await TEMP end present;
    if ( ?TEMP >= 250 ) then emit HIGH else emit LOW end if;
  end every;
  ||
  loop
    await LOW;
    emit B1_ON;
    do
      await HIGH;
      emit B1_OFF;
      watching DELTA_T
      timeout
      present HIGH else
        emit B2_ON;
        await HIGH;
        emit B2_OFF;
      end present;
      emit B1_OFF;
    end;
  end loop;
end;
```