Search

An Introduction to Esterel

By Girish Keshav Palshikar
Embedded Systems Programming
(11/01/01, 12:36:41 PM EDT)

PRINT THIS STORY   SEND AS EMAIL

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.

Because of Esterel's textual nature (as opposed to graphical) and compositional facilities, it is relatively easy to write compact specifications for systems with very complex state machines. A system with thousands of states can generally be specified by an Esterel program of only a few hundred lines.

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/meije/esterel/esterel-eng.html. A variety of tools for compilation, code generation, simulation, theorem proving, and automata visualization are also available there.
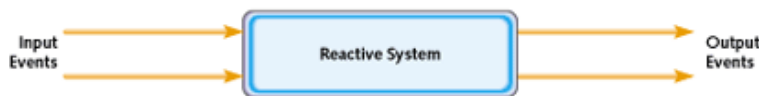
Reactive systems

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 behavior 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 (Figure 1). This output may in turn affect the production of later input events by the environment.

Figure 1: A reactive system as a black box of relationships between input and output events

The task of specifying the behavior 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. We can visualize the trace of the life of a reactive system as a series of overlapping reaction

intervals; each reaction interval begins when the system receives an input event and ends when it generates the corresponding output event (Figure 2).

Figure 2: A trace of an execution of a reactive system
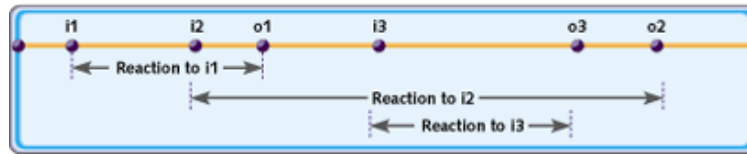
Synchrony hypothesis

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.

At first glance, the synchrony hypothesis may seem unrealistic; however, it considerably simplifies the specifications of reactive systems and is suitable for a large number of application areas. In practice, what is needed is that the machine react to an input event before the next input event arrives.

Esterel allows multiple input events to arrive simultaneously. The reaction instant of Esterel is completed only when the system reacts to all of them; that is, the reaction to all the presently available input events constitutes the reaction instant.

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. For example, suppose an elevator is currently at the third floor of a building and people on both the first and fifth floors simultaneously press the request button. If the relevant behavior of the elevator controller is specified nondeterministically, the response of the system will be either **UP** or **DOWN** movement without any guaranteed way for the users to predict it. Non-determinism corresponds to unlimited parallelism and not to any stochastic behavior. Automata theory defines non-determinism more rigorously.

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 **||** for a parallel composition of its programs. If **P1** and **P2** are two Esterel programs then **P1 || 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 || 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.

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.

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. A signal is a logical unit of information exchange and interaction; its formal meaning is much like its everyday meaning. Examples include **START, STOP, HOUR, ALARM, LIFT_ARRIVED, BUTTON_PRESSED**, and so on. Names of signals in Esterel are conventionally written in capital letters. Esterel provides different kinds of signals.

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

Consider an Esterel module **M**. The module **M** can exchange information with its external world or it may need to exchange information within its parts (or sub-modules) executing in parallel. A signal that is exchanged between a module **M** and its external environment is called an interface signal. An interface signal is available for all parts and sub-modules of **M** due to the instantaneous signal broadcast mechanism in Esterel. For modularity reasons, a module **M** can use signals that are purely local to **M** and its parts and sub-modules and are hidden from the external environment of **M**. Such a signal is called a local signal of module **M**.

A pure signal does not carry any information within it; only its presence or absence can be detected by the system. Thus a pure signal is typeless. A valued signal is a typed signal and carries a value whenever it is emitted; the type of a valued signal indicates the kinds of values that the signal can carry.

A module needs to declare its interface to the external world in terms of input signals, output signals, inputoutput signals, and sensors; also, each such interface signal can be pure or valued.

An input signal for a module can only be "read" by the module; the module cannot generate (or emit) such a signal. An output signal for a module can be generated by the module; the module cannot "read" such a signal. Clearly, an input signal for a module can be an output signal for another module and vice versa. An inputoutput signal for a module is one that can be both emitted as well as read by the module. Typically, such a signal is used only for those signals which the system receives from the environment and sends back to the environment after some filtering or

processing.

A sensor is a special kind of signal that is "read only" and always assumed to be available in every instant; it cannot be waited for nor can it be emitted by any module. A sensor can only be read by accessing its value. Thus a sensor cannot be a part of an occurrence.

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 **N S**, where **N** is an expression that evaluates to a positive integer and S is an input (or inputoutput) signal, the Esterel system checks for the presence of the signal **S** in successive instants and increments the internal counter whenever **S** is present; the occurrence is complete in the instant when the internal counter reaches the value **N**. Thus an occurrence has a positive duration (in the number of instants).

Occurrences are used in most of the temporal statements in Esterel; for example, the await statement. In later versions of Esterel, the definition of an occurrence is somewhat expanded and made more flexible; for example, logical connectives like and, or, and not allow us to describe logical combinations of signals. For example, the occurrence **3 STOP** is complete when the Esterel system counts the presence of three **STOP** signals (once in each instant). Note that the **STOP** signals need not be present in consecutive instants. Thus the duration of this occurrence is at least three instants.

An Esterel program can wait for, read, generate, and check for the presence of a signal. In this sense, a signal is like a message; however, these two concepts differ in some significant ways.

A module can wait for an input or inputoutput signal to occur using the await statement. A module can check for the presence (or absence) of an input signal or an inputoutput signal using the present statement. A module can access (or read) the value of a valued input signal, a valued inputoutput signal, or a valued sensor using the **?** construct. A module can broadcast an output signal or an inputoutput signal using the emit statement. These operations on signals are explained later.

Instantaneous 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 (Figure 3). 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. 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.

Figure 3: Signal broadcast mechanism in Esterel
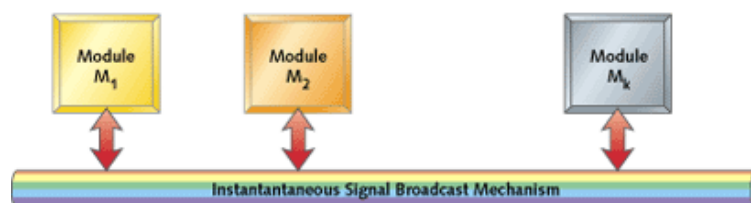
The bus analogy is useful to illustrate another important fact about Esterel signals. A signal **S** emitted by a module M at a reaction instant t is made available to all other modules in the same reaction instant t. The emission of signal **S** constitutes part of the input event at **t** and the reaction instant **t** is not completed until some

module reacts to the presently available signal **S**. Thus an input event at an instant consists of all input signals received from the environment as well as all signals emitted by the system as a part of its reaction at that current instant. Such a composite mixture of input and output signals available at any instant is called an event in Esterel. The input signals within an event constitute an input event. The output signals within an event constitute an output event; thus, an event is a composite of an input and output event.

In summary, all input signals received from the environment (as well as all signals generated by the system as part of its behavior) are available to all modules in the same reaction instant.

Esterel does not have facilities to directly refer to the past or future occurrences of signals. An Esterel program is memory-less in this sense; however, variables can be used to store historical information. Also, Esterel has no facility to refer to any of the past or future instants.

A simple vending machine

We shall begin our exploration of the world of reactive programming with the celebrated example of a vending machine. We shall start with a very simplified version of a vending machine. The vending machine can serve either tea or coffee. The user has to insert a coin and then press either the **TEA** or **COFFEE** button, after which the system delivers the requested drink. Listing 1 depicts a specification of VM1.

Listing 1: A simple 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;
```

The code consists of only one module called **VM1**. The interface of this module consists of three input signals (**COIN, TEA**, and **COFFEE**) and two output signals (**SERVE_TEA** and **SERVE_COFFEE**). The presence of the input signal **COIN** signifies the insertion of a coin in the slot. The presence of the input signal **TEA** (or **COFFEE**) signifies the pressing of the **TEA** (or **COFFEE**) button. The machine indicates its readiness to serve tea (or coffee) by emitting to the environment the output signal **SERVE_TEA** (or **SERVE_COFFEE**). Notice that we are not interested (at this level of abstraction) in a number of details of the machine's operations, for example, the actual mechanism of the delivery of tea or coffee. All our input and output signals are then purely logical signals, intended to represent the behavior of the environment or the system, and as such, do not have anything to do with the technology of switches, sensors, and such used to receive and generate them.

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.

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. The relation directive in Listing 1 indicates that in any instant, at most one of the three input signals **COIN**, **TEA**, and **COFFEE** may be present.

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; this will cause an instantaneous infinite loop and the compiler will usually flag this as an error. Note that in this form, the loop statement never terminates; whenever an execution of **Body** terminates, a new execution of **Body** is started.

The multiple await statement has the form:

await
  case Occurrence$_1$ do Body$_1$
  case Occurrence$_2$ do Body$_2$
...
  case Occurrence$_n$ do Body$_n$
end await;

This statement is used to wait for any of the n occurrences: **Occurrence$_1$, Occurrence$_2$, ..., 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. 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.

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.

Now that the system behavior is specified as an Esterel program, we shall see how to test these behavioral specifications. Following are some examples of test cases for the above program (which can be performed by compiling and executing the Esterel program):

1. T1 = ({COIN},{}), ({TEA},{SERVE_TEA})

This test case consists of two instants; expected input and output events are given for each instant. Initially, the user inserts a coin; the system is not expected to produce any reaction to this input event. Then the user presses the **TEA** button and the system is expected to react by serving tea to him. No other input events are happening during the execution of this test case. **2. T2 = ({COIN},{}), ({COFFEE}, {SERVE_COFFEE})**

This test case is similar to T1 but checks the behavior of the **COFFEE** subsystem.

3. T3 = ({COIN},{}),({COIN},{}), ({TEA},{SERVE_TEA}), ({TEA},{SERVE_TEA})

This test case is similar to T1 but checks whether the system "remembers" the number of coins inserted (it does not).

A temperature controller

The temperature within a chamber (perhaps the site of some chemical process) needs to be maintained at a known constant value (C = 250 degrees). We need to specify the behavior of a temperature controller to do this job. Two boilers, **B1** and **B2**, are available to heat the chamber (Figure 4). The temperature of the chamber is regularly provided by a thermometer at every **N** seconds. As soon as the temperature within the chamber goes below **C**, boiler **B1** should be switched on (unless it is already on). If **B1** does not restore the temperature to C within a known constant time interval **DT = 1 minute**, then boiler **B2** should be switched on. As soon as the temperature rises above **C**, each boiler that is on must be switched off.

Figure 4: A simple heating system and its controller
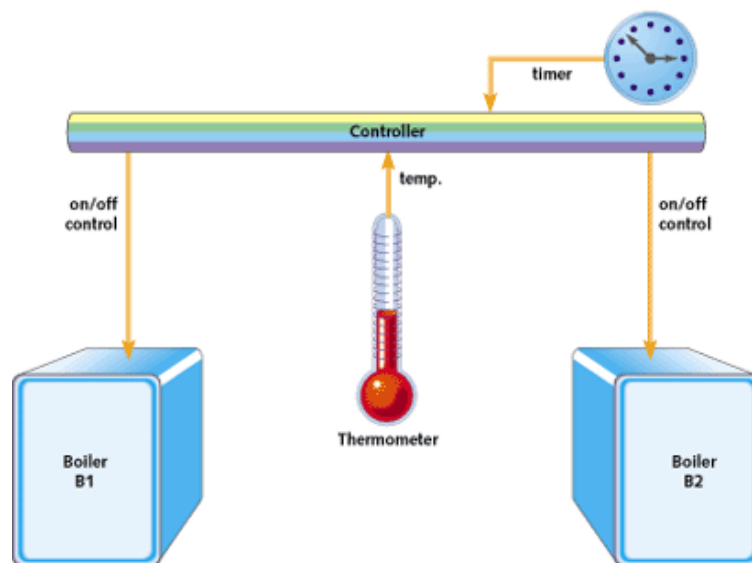
The temperature controller system just described is specified in Listing 2. **TEMP** is an integer valued input signal. The input signal **SAMPLE_TIME** is used to indicate the need to sample the temperature reading; that is, the environment supplies the input signal **SAMPLE_TIME** whenever it is time to receive the **TEMP** signal. **SAMPLE_TIME** is a logical input signal but it can be generated by a clock whenever it is time to see the temperature. Local signals **HIGH** and **LOW** indicate whether the current temperature is above or below C. The output signals **B1_ON, B1_OFF, B2_ON**, and **B2_OFF** are used to indicate the output actions of the system in switching the boilers on or off.

Listing 2: A 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;
.
```

The relation directive states the master-slave combination of the signals **SAMPLE_TIME** and **TEMP**; it states that in any instant where the input signal **SAMPLE_TIME** is present, the input signal **TEMP** should also be present. **HIGH** and **LOW** are pure local signals and are not available to the environment external to the program.

New constructs

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-decl$_1$, Signal-decl$_2$, ..., 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$_i$**.

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.

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.

The **Parallel Composition construct ||** has the following format:

Body$_1$ || Body$_2$ || ... || 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.

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

present Signal-name then Body$_1$ else Body$_2$ 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 **Body$_1$**; otherwise, start the execution of the statement body **Body$_2$**. The entire statement terminates when the execution of **Body$_1$** or **Body$_2$** terminates (whichever body was

started). In the present statement, either the then **Body$_1$** part or the else **Body$_2$** part can be omitted (but not both).

The statement:

present Signal-name else Body$_2$ end present;

has the following meaning: if the signal having the name Signal-name is present in the current instant then do nothing; otherwise, start the execution of the statement body Body$_2$. Thus it is equivalent to the statement:

present Signal-name then nothing else Body$_2$ end present;

Similarly, the statement:

present Signal-name then Body$_1$ end present;

is equivalent to the statement:

present Signal-name then Body$_1$ else nothing end present;

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 **?**.

Esterel provides the standard **if-then-else** control statement which has the form:

if boolean-expression then Body$_1$ else Body$_2$ 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 **Body$_1$** is started; otherwise, the execution of **Body$_2$** is started. In the if-then-else statement, either the **then Body$_1$** part or the **else Body$_2$** part can be omitted (but not both).

The statement:

if boolean-expression else Body$_2$ end if;

has the following meaning: if the **boolean-expression** evaluates to true in the current instant then do nothing and pass the control to the following statement; otherwise start the execution of the statement body **Body$_2$**. Thus it is equivalent to the statement:

if boolean-expression then nothing else Body$_2$ end if;

Similarly, the statement:

if boolean-expression then Body$_1$ end if;

is equivalent to the statement:

if boolean-expression then Body$_1$ else nothing end if;

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

do   Body$_1$ watching Occurrence   timeout   Body$_2$ end;

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

if the Occurrence is complete in the

current instant then
  if Body$_2$ is given then
    start executing Body$_2$
  else pass control to the following
   statement
else start executing Body$_1$.
if Occurrence becomes complete
 before Body$_1$ has finished then
  immediately abort the execution of
   Body$_1$
  if Body$_2$ is given then
    start executing Body$_2$
 else pass control to the following
  statement
else Body$_1$ has finished execution but Occurrence is
not complete yet
  pass control to the following
  statement

As is clear from the above flowchart, this statement terminates either when **Body$_1$** does or when **Body$_2$** does, provided **Occurrence** completes before **Body$_1$** has finished its execution. The statement **await Occurrence;** is equivalent to the statement:

do halt watching Occurrence;

Back to the example

With these new Esterel constructs, we are ready to describe the specification shown in Listing 2. The module consists of two loops which are executing in parallel and which exchange information using local signals **HIGH** and **LOW**; let us call them **Loop1** and **Loop2**.

Loop1 keeps waiting for the input signal **SAMPLE_TIME**. Every instant when this input signal is available, **Loop1** also reads the input signal **TEMP** and emits the local signal **HIGH** or **LOW** depending on the value of the input signal **TEMP**.

Loop2 assumes that, initially, the temperature within the chamber will be less than **C**. If that is not the case, then it simply waits till the temperature falls below **C**. When the temperature falls below **C** (as indicated by the presence of the local signal **LOW** emitted by **Loop1**), **Loop2** switches on boiler **B1** (as indicated by the emission of the output signal **B1_ON**) and then waits until either the temperature again rises above **C** (as indicated by the presence of the local signal **HIGH** emitted by **Loop1**) or time interval **DT** elapses (as indicated by the presence of the input signal **DELTA_T**). This effect is achieved by the **do-watching-timeout** statement. If **HIGH** arrives before **DELTA_T**, boiler **B1** is switched off (as indicated by the emission of the output signal **B1_OFF**). Otherwise, boiler **B2** is switched on and when **HIGH** finally arrives, both **B1** and **B2** are switched off. The above behavior is forever repeated using the loop statement. Notice how we have used logical clock signals **SAMPLE_TIME** and **DELTA_T**.

The specification of the system needs to be tested for a number of typical situations. For each such situation, you can design a test case. Then test the system simulator constructed from Listing 2 using these test cases. Some examples of the test cases are as follows:

**1. ({SAMPLE_TIME,TEMP(200)},{B1_ON}),
({SAMPLE_TIME,TEMP(300)},{B1_OFF})**

Initially, the temperature is below C, so the system should react by switching on boiler **B1**. Then the temperature crosses above **C**, so the system should react by switching off **B1**.

**2. ({SAMPLE_TIME,TEMP(100)},{B1_ON}),
({SAMPLE_TIME,TEMP(150)},{}),
({SAMPLE_TIME,TEMP(300)},{B1_OFF})**

Initially, the temperature is below C, so the system should react by switching on boiler **B1**. Then the temperature is still below **C** (but the time interval **DT** is not yet finished), so the system should not produce any output. Then the temperature crosses above **C**, so the system should react by switching off boiler **B1**.

3. ({SAMPLE_TIME,TEMP(100)},{B1_ON}),
**   ({SAMPLE_TIME,TEMP(200)},{}),**
({SAMPLE_TIME,TEMP(220),DELTA_T},
**   {B2_ON}),**
({SAMPLE_TIME,TEMP(300)},{B1_OFF,
**   B2_OFF})** Initially, the temperature is below **C**, so the system should react by switching on boiler **B1**. Then the temperature is still below **C** (but the time interval **DT** is not yet finished), so the system should not produce any output. Then while the temperature is still below **C**, the time interval **DT** finishes, so the system should react by switching on boiler **B2**. Then the temperature crosses above **C**, so the system should react by switching off both **B1** and **B2**.

The following properties of the above system are of interest:

- It should never happen that **B1** is OFF and **B2** is ON at the same instant.
- It should never happen that a boiler (either **B1** or **B2**) is switched on when it is already on.
- It should never happen that a boiler (either **B1** or **B2**) is switched off when it is already off.

Theorem-proving and verification tools of Esterel can be used to verify that Listing 2 satisfies these properties.

Girish Keshav Palshikar is a scientist at the Tata Research Development and Design Centre in Pune, India. He obtained a M.Sc. in physics from the Indian Institute of Technology (Bombay) and an MS in computer science and engineering from the Indian Institute of Technology (Chennai). E-mail him at girishp@pune.tcs.co.in.

The author would like to thank Professor Mathai Joseph and Dr. Manasee Palshikar for their help.

References

Berry, G. "Esterel v3 Programming Examples: Programming a Digital Watch in Esterel v3," Tech. Report, Ecole des Mines de Mathematiques Appliquees, 1989.

Berry G. and G. Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," Sci. Comp. Prog., 19 (1992), pp. 87-152.

Berry G., "Esterel on Hardware," in *Mechanized Reasoning and Hardware Design.* New Jersey: Prentice-Hall, 1992, pp. 87-104.

Boussinot, F. and R. De Simone. "The Esterel Language," *Proc. IEEE,* Vol. 79, No. 9, Sep. 1991, pp. 1293-1304.

Free Subscription to Embedded Systems Programming

First Name   Last Name

Company Name   Title

Business Address   City

State   Select State/Province ◆   Zip

Email address   CONTINUE ⇥

▸ **Electronics Marketplace**

- **McObject's eXtremeDB in-memory embedded database**

eXtremeDB is the embedded in-memory database for C/C++ programmers. In-memory means amazing performance. Check out the XML-enabled and High Availability versions, too. Download a free trial.

• **Free Technical Publications**
Keep up with advancing technologies. Download the lastest technical publications including topics on IC, FPGA, Functional Verification, Design-for-Test and PCB Design Solutions.

• **The Fastest Embedded Processor Ever - Xtensa V**
Test drive Tensilica's Xtensa V processor, the fastest ever according to EEMBC Certification Labs.

• **PEG (Portable Embedded GUI) Graphics Library/Tools**
PEG is a GUI Development Package comprised of a full set of tools for embedded systems. PEG comes with a complete class library, runs fast, is small in size and completely rommable. PEG is delivered with C++ source code and is ROYALTY FREE!

• **PCB123-the FREE PCB Solution, Design to Order**
Remove engineering bottle-necks with our FREE and easy to use schematic and layout software. Quotes your boards as you design! Quick and easy solution for up to 4-layer circuit board designs.

Buy a link NOW: