A lab introducing the Esterel language

Avinash Malik

Department of Electrical, Computer, and Software Engineering,
The University of Auckland
Auckland, New Zealand

1 Learning outcomes

The main learning outcomes are:

1. Esterel to C interface and data handling.
2. Determinism through synchrony and the atomicity of reactions.
3. The effect of scheduling order.
4. Synchronous broadcast and causality.
5. Reaction to absence.
6. Reincarnation.

These features will be covered further during the lectures and we encourage you to revisit the lab from time to time as these concepts are elaborated in the lectures.

2 Introduction

Esterel is a synchronous programming language [1] ensuring deterministic composition of a set of synchronous threads. This lab provides you with a hands-on introduction to some features of the language through a set of examples. We will start by showing you how to perform the basic steps of compilation, dealing with syntax errors and simulating Esterel programs.

3 Esterel to C interface for data handling

During the COMPSYS303 course, we learnt about race conditions through the following producer consumer example in the prescribed text book [3]. This example is reproduced here for your convenience (Fig. 1). This example has race conditions as the value of `count` could be non-deterministic for arbitrary interleavings of the producer and the consumer thread. Now let us see how to program this in Esterel.

```
01: data_type buffer[N];
02: int count = 0;
03: void processA() {
04:   int i;
05:   while( 1 ) {
06:     produce(&data);
07:     while( count == N );/*loop*/
08:     buffer[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }
13: void processB() {
14:   int i;
15:   while( 1 ) {
16:     while( count == 0 );/*loop*/
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }
23: void main() {
24:   create_process(processA);
25:   create_process(processB);
26: }
```

Fig. 1. Producer consumer example reproduced from [3]

3.1 Design in Esterel

Esterel is a synchronous reactive language for the design of embedded systems. Esterel programs can be compiled to C (software), netlist form (for hardware synthesis) or can be used for hardware-software co-design. In this course we will focus on the software approach for Esterel.

Being a synchronous reactive language, Esterel models all reactive constructs using the native statements of Esterel. All data computations are handled by invoking C functions. This provides a neat separation of control flow (which is captured in Esterel) and data computation (which is captured in C). We use the producer consumer example to elaborate on how this separation can be used.

Each data object before usage must be defined in the module. Data objects could be of any primitive Esterel type or any user defined type. Primitive types are *boolean, integer, float, double, string*. User defined data types can be defined by the programmer in Esterel abstractly and the actual types must be defined in C and linked to the compiled Esterel code. Introduction of user defined types and how to deal with them is introduced in section 5.2.

A module's interface consists of data type declarations, constants, functions or procedures, input and output interface signals. We start by the first variant of the producer consumer as shown in Listing 1.1. We start by defining two procedures `send, recv` and a function `outCount`. Procedures take two sets of parameters specified in two separate parenthesis. The first set of parameters are called by reference (variable parameters) and the second set are called by value (value parameters). For example, the procedure `send` has no variable parameters and has a single value parameter representing the value of the data to be sent. Procedures perform some computation and don't return any thing. Functions, on the other hand, return a value of some type. For example, the function `outCount` returns an integer representing the current `count` value.

The module consists of three separate threads that are composed synchronously using the || operator of Esterel. The first thread is a receiver thread that waits for two ticks before calling the `recv` procedure to receive some data from the buffer. It then displays the received data after a further delay of one more tick. This behaviour is then repeated indefinitely.

The second thread, in every tick, emits the count value by calling the function `outCount`. The third thread awaits a value from the environment through the `svalue` signal. It then sends this value by calling the procedure `send`. This behaviour is repeated indefinitely. The implementations of the C functions are provided in Listing 1.2.

Compilation and Simulation: The Makefile file contains all the instructions for compiling and then generating the files required for simulation. Open up a terminal, and then type the following commands:

1. $ cd /Desktop/lab1/produceConsume1
2. $ make produceConsume.xes
3. $ ./produceConsume.xes

You should see the two windows as shown in Fig 2. The simulation panel (right window) allows you to set inputs by clicking on the input signals (turns from blue to red). Then, by clicking the 'tick' button, you should see the statements that are executed during this tick, and the status of the output signals. The 'reset' button only resets Esterel signals and variables but not the variables stored in host procedures such as C. Note that the value of a signal is only set if the signal is present.

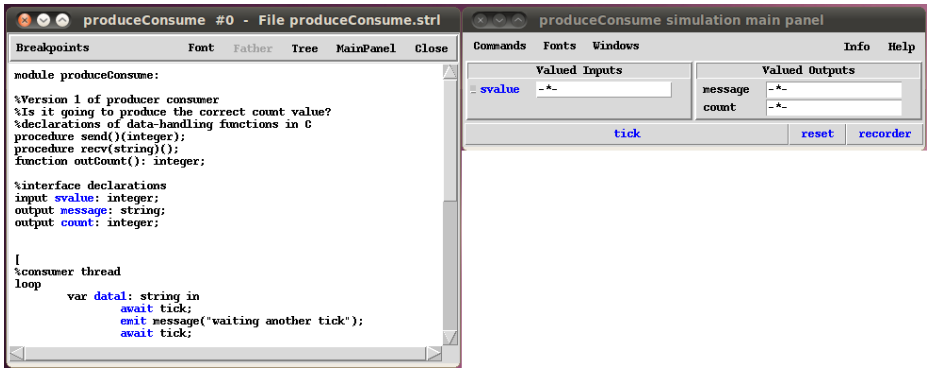

Fig. 2. Simulation environment

Your task Compile and then simulate this program using the XES simulator. Answer the following questions:

1. What is the distinction between the Esterel implementation and the OS-based implementation as suggested in [3]? By understanding this, you will appreciate the distinction between the RTOS approach (taught in the first half) and the synchronous approach (taught in the second half).
2. Is there a data-race like the RTOS approach?
3. Is the count value correctly displayed i.e., the count value in a tick is same as the number of items in the buffer? If not why?

Listing 1.1. A Producer Consumer in Esterel

```
module produceConsume:
%Version 1 of producer consumer
%Is it going to produce the correct count value?
%declarations of data-handling functions in C
procedure send()(integer);
procedure recv1(string)();
function outCount(): integer;

%interface declarations
input svalue: integer;
output message: string;
output count: integer;

[
%Consumer thread
loop
        var data1: string in
                await tick;
                emit message("waiting another tick");
                await tick;
                pause;
                call recv1(data1)();
                emit message(data1);
        end var
end loop
]
||
[
%Counter output thread
loop
        pause;
        emit count(outCount());
end loop
]
||
[
%Producer thread
loop
        await svalue;
        call send()(?svalue);
end loop
]
end module
```

Listing 1.2. Data handling functions in C

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5

void send(int data);
void recv1(char** data);
void adding(unsigned int data);
int remov(void);
int outCount();

int cq[SIZE];
int front=0;
int rear=0;
int cnt=0;

void send(int data){
        adding(data);
}
void recv1(char** data){
        int tmp;
        tmp = remov();
        if(tmp != -1)
                sprintf(&data[0],"consumed %d",tmp);
        else
                sprintf(&data[0],"nothing");
}

void adding(unsigned int data){
        if(cnt!=SIZE){
                cq[rear]=data;
                rear = (rear +1) % SIZE;
                ++cnt;
        }
}
int remov(void){
        int val;
        if(cnt!=0){
                val=cq[front];
                front = (front+1) % SIZE;
                cnt--;
                return val;
        }else
                return -1;
}

int outCount(){
  return cnt;
}
```

3.2 Second variant of the producer consumer

You might have already uncovered that the count value, while being devoid of data races, is still not correct i.e., the invariant that `the number of items in the buffer is same as the displayed count` doesn't hold. This is since the displayed count value is not based on the most up to date value of count in every instant i.e., the counter thread outputs the count value before the updates to the count has been completed.

In order to understand why this happens, we have to understand how threads are scheduled in Esterel. Concurrency is Esterel is logical and is "compiled away" to produce sequential code. The compiler orders threads statically based on producer-consumer dependencies of signals so that all producers (signal emitters) are scheduled before the respective consumers (signal consume / test using await and present).

Consider the second variant of this example as shown in Listing 1.3. By using the local signals A and B, the compiler is forced to generate code such that the scheduling order of threads for this program is $producer \rightarrow consumer \rightarrow counterOputout$.

Your task Compile and then simulate this program using the XES simulator. Answer the following questions:

1. What is the distinction between the first example and this example?
2. Is the count value correctly displayed i.e., the count value in a tick is same as the number of items in the buffer? If so, why?
3. Do you notice, how local signals communicate and synchronize using synchronous broadcast?

3.3 Third variant of the producer consumer

Consider the next variant as shown in Listing 1.4, where the producer and consumer operate in lock-step to produce and consume in the same instant. Simulate this program to determine any difference from the previous versions.

Listing 1.3. A Producer Consumer with thread synchronization using local signals

```
module produceConsume:
%version 2 - what has changed?
%Will the count values be correct always?
procedure send()(integer);
procedure recv1(string)();
function outCount(): integer;

%Interface
input svalue: integer;
output message: string;
output count: integer;

%local signals for thread synchronization
signal A, B in
[
%Consumer thread
loop
        var data1: string in
                await [A or tick];
                emit B;
                emit message("waiting another tick");
                await [A or tick];
                emit B;
                pause;
                call recv1(data1)();
                emit message(data1);
                emit B;
        end var
end loop
]
||
%Counter output thread
loop
        await [A or B];;
        emit count(outCount());
end loop
||
[
%Producer thread
loop
        await svalue;
        call send()(?svalue);
        emit A;
end loop
]

end signal

end module
```

Listing 1.4. A Producer Consumer with lock-step operation

```
module produceConsume:
%Version 3 - lock step version

%external declarations
procedure send()(integer);
procedure recv1(string)();
function outCount(): integer;

%interface
input svalue: integer;
output message: string;
output count: integer;

%local signals
signal A, B in
[
%%Consumer thread
loop
        var data1: string in
                await A;
                emit B;
                call recv1(data1)();
                emit message(data1);
        end var
end loop
]
||
%Counter output thread
loop
        await B;
        emit count(outCount());
end loop
||
[
%Producer thread
loop
        await svalue;
        call send()(?svalue);
        emit A;
end loop
]

end signal

end module
```

4 Synchronous broadcast and causality

You may notice how using synchronous broadcast, threads can communicate and synchronize operations within a single tick in the producer consumer example. However, this example doesn't provide all the nuances of this communication approach.

In Esterel, every signal (except combined valued signals to be introduced in the lectures) is either present or absent in a reaction (tick). A signal is present when it is emitted and is absent when no potential emitter can emit it within this instant.

Due to synchrony, a thread can react to both signal presence and absence (reaction to absence is shown in Listing 1.5). Also, due to instantaneous feedback, we can create programs that are "syntactically correct but semantically nonsense" [2] i.e., it is unclear how to provide a precise meaning to such programs. These are classified as non-causal programs. I will give a few examples in Listing 1.5, some reused from the the Esterel Primer [2], to highlight causality issues Esterel:

4.1 Your task

Your task is to try to compile these programs to see why causality issues exist and suggest remedies so that the causality problems can be resolved.

5 Reincarnation

In an Esterel program, a signal has an unique value in every instant. However, with loops and local signals declared within a loop that are instantaneously restarted, it is feasible that the local signals take several different values in an instant. The example in Listing 1.6, taken from Li Hsien Yoong's thesis [4], highlights the issue. Such programs are termed as *schizophrenic* and the local signal is said to be *reincarnated*. Usually, compilers create two different copies of the same signal to ensure the uniqueness of every signal.

5.1 Your task

Compile and simulate this program. In which instant is the `signal S` is reincarnated and why? What is the behaviour of this program in the instant this local signal is reincarnated? Is the behaviour logical?

5.2 Data handling for user defined types

Note that Esterel V5 does not natively support arrays and other user defined types. For example, if an array needed it must be declared as user defined types. We provide a snippet of the declaration of custom type `BoolArray`, which has to be declared with a header file.

Listing 1.5. Non-causal Esterel programs

```
------ Program 1 ------
module noncausal:
signal A, B, C in
loop
[
        present C then emit A;
        end present;
        pause;
]
||
[
        await A;
        emit B;
]
||
[
        await B;
        emit C;
]
end loop
end signal
end module

------ Program 2 ------
module noncausal:
signal A in
loop
    present A else emit A;
    end present;
    pause;
end loop
end signal
end module

------ Program 3 ------
module BAD_COUNT:
     input I;
     output COUNT := 0 : integer;
     every I do
        emit COUNT(?COUNT+1)
     end every
end module
```

Listing 1.6. Reincarnation

```
module reincarnate:
output O1, O2;

loop
    signal S in
      present S then emit O1 else emit O2 end;
      pause;
      emit S;
      present S then emit O1 else emit O2 end;
    end signal
end loop
end module
```

Whenever a new type is created, we have to create 5 different functions to deal with this new data type in Esterel so that proper code is available during linking. For every new type `T` that is created the following functions need to be defined in C:

1. *Assignment functions* to handle explicit assignment of a variable to a type, emissions of signals carrying some data type (T).

   ```
   x:=exp
   #define _T(x,y) (*(x)=y)
   ```

2. *Equality functions* to handle comparison between objects of type T.
3. *String Conversion Functions* for simulation.

   ```
   char * _T_to_text(T);
   void _text_to_T(char*, T*);
   int _check_T(char*);
   ```

For example, for the `BoolArray` type, we need the following functions. Detailed implementations of these are available to you in the `Simple Lift` example I have developed and made available on Cecil.

```
#define N 4

typedef struct {
    char array[N];
} BoolArray;

void _BoolArray(BoolArray*, BoolArray);
int _eq_BoolArray(BoolArray, BoolArray);
char* _BoolArray_to_text(BoolArray);
void _text_to_BoolArray(BoolArray*, char*);
int _check_BoolArray(char*);
```

For each user defined type declared, five functions similar to the above snippet are expected by the Esterel compiler to handle the type. These are functions to handle assignment (the = operator), comparison, converting to and from a string, and type sanity check, in respective order declared above.

6 Conclusions

I prepared a quick introduction to the Esterel language using some examples to illustrate a few key features of the language. You are encouraged to read the Esterel Primer [2], in addition to the course notes that I have provided to gain better understanding of the language and its semantics.

7 Acknowledgements

The author acknowledges the feedback I received from the teaching assistants.

References

1. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.
2. G. Berry. *The Esterel v5 Language Primer Version v5 91.* INRIA, France, 2000.
3. F. Vahid and T. Givargis. *Embedded System Design – A Unified Hardware/-software Introduction.* John wiley and Sons, 2002.
4. Li Hsien Yoong. Compiling Esterel for distributed execution. Master's thesis, Department of Electrical and Computer Engineering, University of Auckland, 2005.