Appendix 2: Case Study

Designing A Lift Controller in Esterel

Partha S. Roop
Department of Electrical and Computer Engineering
University of Auckland
New Zealand

Abstract

We present the design of a lift controller [3] in Esterel. We first create the specification of a lift controller using the well-known control data-flow diagrams. We then show how to implement such a specification, where control-data decomposition is already done, in Esterel [1]. In doing this design, we demonstrate an effective way of mapping state-machines in the specification to Esterel code efficiently. We also show the mapping of the data-flow aspects to C functions.

1 Introduction

This example is based on the lift controller taught in COMPSYS-303 [3]. The controller controls the direction of the motor based on the current direction, the floor sensor value and the highest priority requested floor. Control operations include setting the appropriate motor direction, stopping the lift when the requested floor is reached and opening and closing the doors. Once the elevator reaches the requested floor, the controller must clear the pending request that was serviced. In this report, we will illustrate the design of such a lift controller by first creating a specification and subsequently showing how to map this specification in Esterel.

This example is used to illustrate several features of the Esterel language. We illustrate the use of concurrency and synchronous synchronization using signals, the use of data handling functions in C, the use of multiple modules and the connection of ports, the use of interfaces and data objects. The entire project in Esterel, is available on Cecil.

The organization of this report is as follows. In Section 2 we present the specification of the lift controller using control data-flow diagrams. In Section 3 we present the mapping to Esterel. In Section 3.1 we present the use of Esterel data and interfaces. The top level module is presented in Section 3.2 and some comments on how to effectively debug Esterel programs is presented in Section 3.3. Finally in Section 4 we make some concluding remarks.

2 Specification

The overall input-output behaviour of the lift controller is described using the context diagram as shown in Figure 1. The lift reads the user inputs entered through the call buttons. These consists of the up call and down call buttons on each level of the building and also call buttons inside the cabin to request a given floor. Other inputs to the lift are a floor sensor input indicating the current position of the lift and a timer expired input indicating that the timer that was started to determine the duration of opening of the lift door (once the lift arrives at the requested floor) has expired.

The outputs from the controller are the door open and door close outputs to control the door, the motor direction output to control the direction of movement, a timer output to control the duration of door opening, some pending call outputs indicating which requests are still pending to be processed and a stopped at floor (SAF) output indicating that the lift has arrived at some requested floor.

The top-level context diagram can be further refined into the next level diagram as shown in Figure 2. Here the simple lift functionality is further partitioned into a request resolver and a unit controller. The request resolver determines the highest priority request from a list of pending requests and forms a queue of such pending requests (a producer).

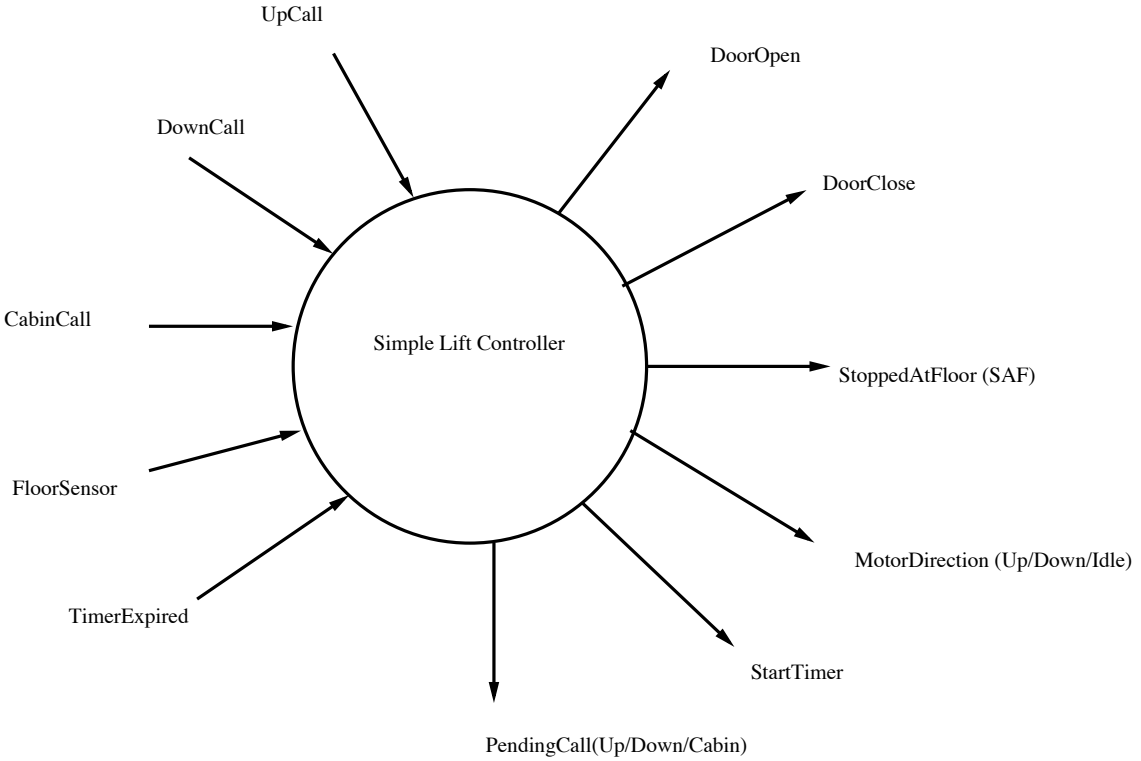

Fig. 1. Context Diagram of the Lift Controller

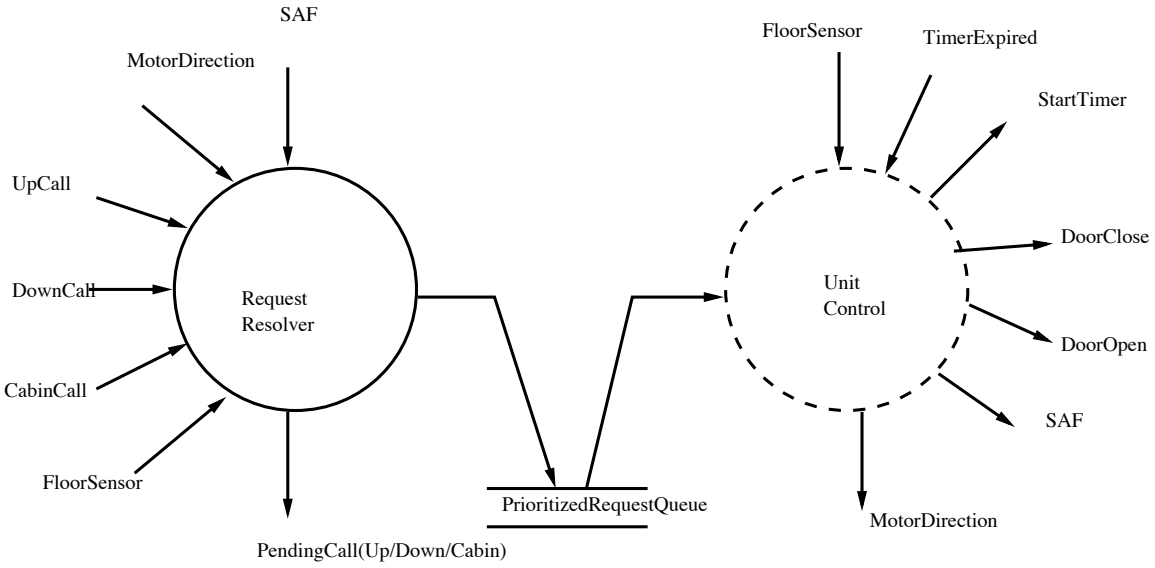

Fig. 2. First Level Refinement

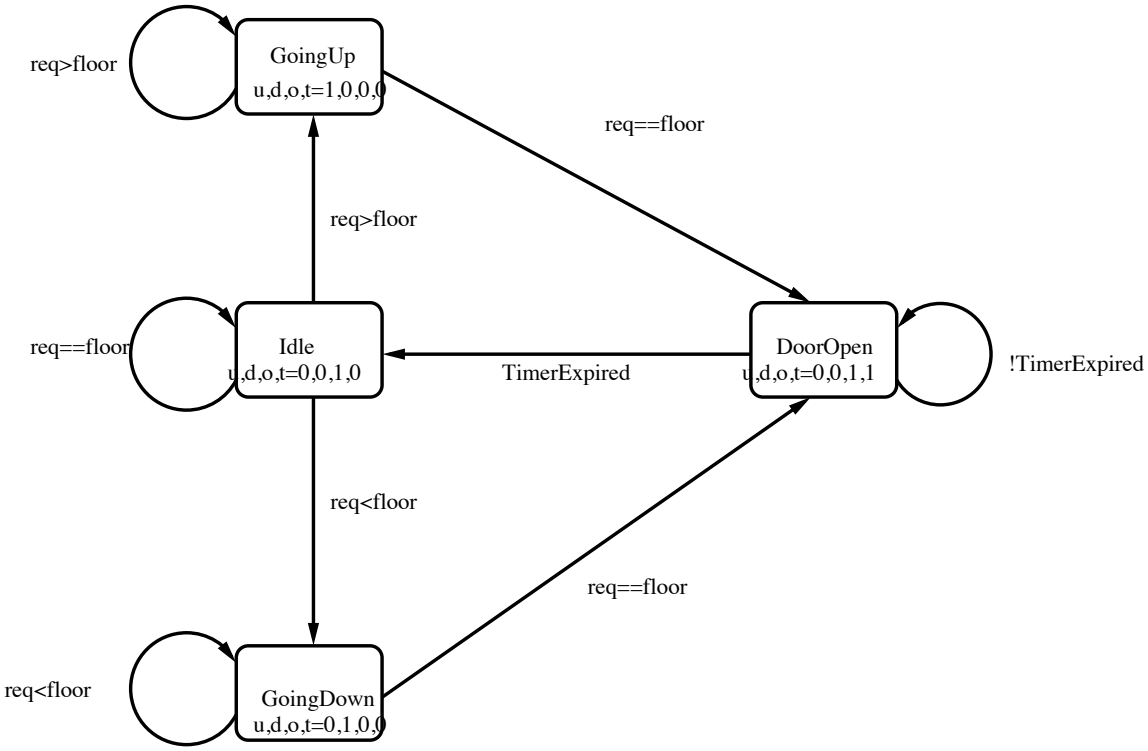

Fig. 3. The Unit Controller Behaviour as an FSM

The unit controller reads the highest priority request from this queue and moves the lift to this requested floor. The unit controller is a pure control-flow functionality and is described as an FSM in Figure 3.

The request resolver is further refined into two different functions: request generator and resolve priority. The request generator function reads the call buttons and creates a list of pending requests. These pending requests are read by the resolve priority function that creates the next highest priority request based on the current floor sensor value, the motor direction and the pending requests. This is described in Figure 4.

3 Design in Esterel

The Esterel design of the lift controller follows directly from the specification described in Section 2. The Esterel program consists of two modules, namely the Unit Control module and the Request Resolver module. These modules are composed using a top-level called SimpleLift.

- The Unit Control Module: This module executes an FSM that receives the next requested floor, called request (a valued signal), from Request Resolver and outputs `SAF=i` (where `i` is the `i`-th requested floor that the Unit Controller has serviced (`SAF` stands for the valued signal `StoppedAtFloor`). The Unit Control module is a simple FSM that executes on a given state based on the Floor Sensor value and the current Request it is processing. The FSM is depicted in Figure 3. We have already learnt the naive approach (token passing) of encoding FSMs. In this example, we encode FSMs efficiently using state variables and traps to emulate go-tos.
- The Request Resolver Module: This module receives call button requests (up/down/cabin) and based on the requests creates some pending requests to process. Its operation may be envisaged as three concurrent threads that do the following:
  - The pending request generator thread: this thread reads call button requests (which may be modelled as an array of pure signals). Based on the requests in a given tick this thread creates an array of Pending Requests whenever some call buttons have been pressed on a given level that is yet to be processed. This thread sustains these pending requests until they are serviced. A given pending request is serviced when the lift arrives at the pending floor (i.e., `SAF=i`, where `i` is the `i`-th pending request). This approach is borrowed from Berry's lift controller available in Esterel studio [2].
  - The request producer thread: This thread, in every tick, reads the current pending requests and calls a C function called `ResolvePriority` that determines the highest priority request based on the array

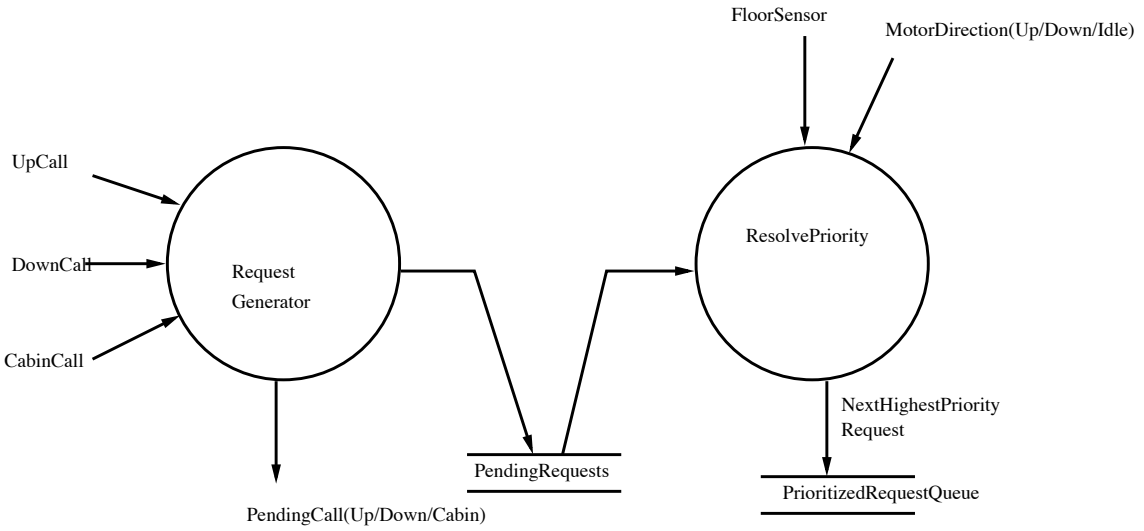

Fig. 4. Refinement of the Request Resolver

of pending requests, the current direction of the motor (up/down/idle) and the current floor. It then outputs the highest priority request to be serviced. This request is then sent to a circular queue for consumption by the request consumer thread using a C function called send.

- The request consumer thread: This thread calls a C function to consume the next highest priority request and if there is a valid request, it sustains it as a valued signal called `request` for consumption by the Unit Control module. This request is withdrawn once this is serviced by the Unit Control module (i.e, when the Unit Control outputs `SAF` with a value which is equal to the current request).

3.1 The Use of Data and Interfaces

A module's interface consists of data type declarations, constants, functions or procedures, input and output interface signals. In our example, we have the following data and interfaces for each module:

```
module RequestResolver:
type BoolArray;
constant initialBoolArray : BoolArray;

input FloorSensor := 0 : integer;
output PendingReq := initialBoolArray : BoolArray;
output CurrentPriority := 0 : integer;

procedure ResolvePriority(integer)(BoolArray, integer, integer);
procedure send()(integer);
procedure recv(integer)();
function orArray(BoolArray) : boolean;
function orArrays2(BoolArray, BoolArray) : BoolArray;
function orArrays3(BoolArray, BoolArray, BoolArray) : BoolArray;

% SensorActuatorInterface
input MotorDirectionUp;
input MotorDirectionDown;
input MotorDirectionIdle;
input StoppedAtFloor := 0 : integer;
output request: integer;

% ButtonInterface
input CabinCall := initialBoolArray : BoolArray; % Buttons inside the cabin
input UpCall := initialBoolArray : BoolArray;    % up call buttons at floors including
    a fake one at floor N-1
input DownCall := initialBoolArray : BoolArray;  % down call buttons at floors
    including a fake one at floor 0
```

```
% VisibleOutputsRequestResolver
output PendingCabinCall := initialBoolArray : BoolArray;
output PendingUpCall := initialBoolArray : BoolArray;
output PendingDownCall := initialBoolArray : BoolArray;
output PendingCall := initialBoolArray : BoolArray;

output DoorOpen;
output DoorClose;
```

Note that these inputs are actually the outputs of the Unit Control module and request is an input in the Unit Control module. Hence the same interface needs to be mirrored in Unit Control by declaring corresponding signals as the inverse of Request Resolver's interface. The other interfaces used by Unit Control in addition to the mirrored interface are as follows:

```
module UnitControl:
input FloorSensor: integer;

% Mirrored SensorActuatorInterface
output MotorDirectionUp;
output MotorDirectionDown;
output MotorDirectionIdle;
output StoppedAtFloor : integer;
input request : integer;

% TimerInterface
output StartTimer;
input TimerExpired;

% VisibleOutputsUC
output DoorOpen;
output DoorClose;
```

Below is the data and interface of the top level module:

```
module SimpleLift:
type BoolArray;
constant initialBoolArray : BoolArray;

input FloorSensor1 := 0 : integer;
output PendingReq1 := initialBoolArray : BoolArray;
output CurrentPriority1 : integer;
output MotorDirectionUp1, MotorDirectionDown1, MotorDirectionIdle1;
output StoppedAtFloor1 : integer;

% ButtonInterface
input CabinCall := initialBoolArray : BoolArray; % Buttons inside the cabin
input UpCall := initialBoolArray : BoolArray;    % up call buttons at floors including
    a fake one at floor N-1
input DownCall := initialBoolArray : BoolArray;  % down call buttons at floors
    including a fake one at floor 0

% TimerInterface
output StartTimer;
input TimerExpired;

% VisibleOutputsRequestResolver
output PendingCabinCall := initialBoolArray : BoolArray;
output PendingUpCall := initialBoolArray : BoolArray;
output PendingDownCall := initialBoolArray : BoolArray;
output PendingCall := initialBoolArray : BoolArray;

% VisibleOutputsUC
output DoorOpen;
```

```
output DoorClose;
```

3.2 The Simple Lift Module

This is the top level Esterel module that runs the Unit Control module and the Request Resolver module in parallel. Note that we need to interconnect appropriate input/output ports so that specific signals from one module that are output of the module become inputs for the other module and vice versa. This is achieved in Esterel using the signal renaming approach already introduced in the lectures (see the speed grabber example). So all input and output signals visible at the top level must be declared at the interface of this top level module. For example, the top level module imports ButtonInterface as all these input signals are visible at the top level. Since request signal is completely internal, it is handled using a local signal declaration. Next, we rename signals in both modules using the name used at top level to connect them. For example, `MotordirectionUp` is renamed as `MotorDirectionUp1` (which is an interface signal at the top-level) in both the Unit Control and the Request Resolver modules. This connects the `MotorDirectionUp` output signal from Unit Control with the `MotorDirectionUp` input signal in Request Resolver.

3.3 Causality and Data Handling

Finally, when signals are shared between modules and connected through renaming to the same signal, care must be taken to ensure that the composition is causal. This is usually achieved by accessing the *pre* of a signal or the *pre* of its value in the consumer module to ensure that causal cycles are removed. For example, we always access the `pre(?request)` in the Unit Control module. This effectively breaks cyclic dependencies between the two modules.

Note that Esterel V5 does not natively support arrays, arrays must be declared as user defined types. Below is a snippet of the declaration of custom type `BoolArray` found in `SimpleLift_data.h`:

```
#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*);
```

The number of floors is defined by `N`, this number can be changed, but you must also change `SimpleLift_data.c` where an instance is declared with an initial value, its length must match `N` defined here. The boolean array is embedded inside a struct, this allows gcc to make a copy when passed by value. This can avoid unwanted side effects when passing to functions.

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.

4 Conclusions

I prepared a pedagogic example of designing a small scale embedded system using Esterel V5. The example of the simple lift controller is based on [3] and illustrates several features of the Esterel language including concurrency, synchronization and data handling through C. Esterel studio also has a lift controller design by G. Berry [2]. His design is based on pure Esterel and has no data handling. Hence, I decided to develop my own. Finally, for future work, I am yet to properly verify this model except simulation. It would be nice to write some observers and environment constraints. This is left as a small exercise to students. For any suggestions on improving this model, please contact me directly.

5 Acknowledgements

The author acknowledges the help from Simon Yuan to map the original V7 example to V5. Also, thanks to Li Hsien Yoong for help with debugging the V7 example.

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. Programming and verifying an elevator in esterel v7. Technical report, Esterel Technologies, 2004.
3. F. Vahid and T. Givargis. *Embedded System Design A Unified Hardware/software Introduction.* John wiley and Sons, 2002.