The Art Of State Machines

Elias Frantar

October 1, 2014, Vienna

Contents

1 Problem Statement

Implement a component based C-Program to show the difference of the 5 types of state machines presented in the book of Mrs. Elicia White "Making Embedded Systems". To test your implementation you can use simple output functions (e.g. fprintf).

Don't forget to document the differences in your protocol.

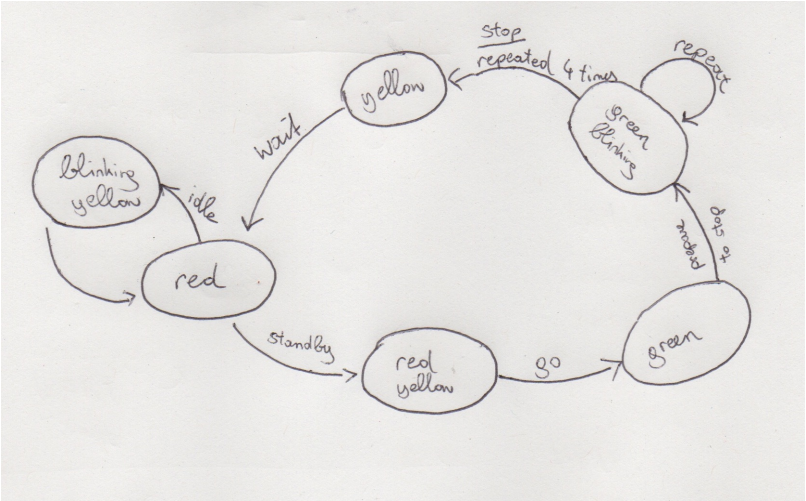

Figure 1: State Machine Diagram; drawn by Gary Ye

2 How to execute the program

To execute the shipped program, simply go into it's root directory (*state_machines/*) and the execute the *makefile* by typing:

```
1 make sm=$(STATE_MACHINE)
```

Where *$(STATE_MACHINE)* is the type of state-machine to run. Must be one of the following:

- STATE_CENTRIC
- EVENT_CENTRIC_WO_TRANSITIONS
- EVENT_CENTRIC
- TABLE_BASED
- STATE_PATTERN

3 Costs

The following table compares the estimated with the actually needed amount of time for completing each task.

Task	Estimation	Actual
Preparation	2h	3h
State Centric	2h	1.5h
State Centric wo. Transitions	1.5h	1.5h
Event Centric	1.5h	1h
State Pattern	2h	2h
Table Based	2h	2h
Protocol & Documentation	3h	3h
Total	10h	14h

4 Design

4.1 Component Based

At first, this has to be a **component based** C program. So I decided to create a header-file (*trafficlight.h*) which defines all constants and function prototypes. In that file there also is the definition of *runTrafficLight()*, which is implemented differently for the individual state-machines (for example *state_centric.c*). I then use the **Linker** to add the wished implementation to my main-file (*trafficlight.c*) by defining an additional constant during compile-time with **-D**.

4.2 Events

Normally when using state-machines, **events** will be triggered by **interrupts**. Since there are no interrupts in plain C, I decided to implement a "pseudo-interrupt"-routine by myself. It works by polling the current system time and comparing it with a preset value. This is implemented in *timer.c*.

```
1  int doneTime; // when the timer is finished
2  enum Event doneEvent = NO_EVENT; // event to return when finished
3
4  void timer(int seconds, enum Event event)
5  {
6    doneTime = (clock() / CLOCKS_PER_SEC) + seconds;
7    doneEvent = event;
8  }
9
10 enum Event checkEvent()
11 {
12   if (clock() / CLOCKS_PER_SEC >= doneTime)
13   {
14     return doneEvent;
15   }
16
17   return NO_EVENT; // return NO_EVENT if not yet finished
18 }
```

5 State Machine Implementations & Analysis

A **state-machine** is a design-pattern often used for programming embedded systems.

The idea behind it is that a system always is in a certain **state** and these states are changed by incoming **events**.

There are many different ways to implement such a state-machine. Five of these are discussed and compared in the following section. They all share a common core though:

```
1 while (1) {
2   event = checkEvent();
3
4   /* state machine */
5 }
```

Basically a state-machine is running the whole time and waiting for **events**. If an event occurred, it checks the current **state** and acts accordingly.

The exact handling of this part is very implementation specific and can hugely decrease the readability of the code if done/chosen incorrectly, so it is very important to make the right choice for a specific use-case.

5.1 State Centric

The general idea of a state centric state-machine is to *switch* on the different **states** and then act depending on the occurred **event**. Here is an excerpt of the code in *state_centric.c*:

```
1  switch (state)
2  {
3    case RED:
4    if (event == STANDBY)
5    {
6      state = RED_YELLOW;
7      handleStandby();
8    }
9    else if (event == IDLE)
10   {
11     state = YELLOW_BLINKING;
12     handleIdle();
13   }
14   break;
15
16   /* ... */
```

As you can see here, we perform a *switch* on **state** (line 1). When the state is *RED* we check which **event** has occurred and then act accordingly. I.e. if the event is *STANDBY* (line 3), we change to the next state, turn on the yellow-light and start a timer for triggering the next event (line 4-7).

The main advantages of this type of state-machine are that it is very easy to implement and that it directly translates the control-flow from a state-machine diagram to code. I.e. in a specific state, there are different branches for the events which lead to other states, and so on. This fact IMO makes a state-centric state-machine to the most readable and easiest to understand option.

5.2 State Centric Without Transitions

The difference between a state-centric state-machine and one without transitions is that the latter does not know where it is coming from (i.e. there are no events), so the state-machine cannot branch out into different cases and therefore there is only a one control flow. Below is part of the code from *state_centric_wo_transitions.c*:

```
1  switch (state)
2  {
3    case RED:
4      state = RED_YELLOW;
5      handleStandby();
6
7      break;
8
9    /* ... */
```

In **state** *RED* the state-machine advances to *RED_YELLOW*, switches on the yellow light and then **blocks** for a few seconds before continuing to the next state.

The usage of this state machine is very limited, since there can only be a single control flow (without decisions). In fact the only difference between a simple structured code and this state-machine is that the actions are cleanly grouped and you could possible skip states. All in all this implementation should only be used in very very specific use-cases!

5.3 Event Centric

The event centric approach is to *switch* on the **events** instead of the states and then decide which actions to perform depending on the **state**. You can find my implementation of this type of state-machine in *event_centric.c*.

```
1  switch (event)
2  {
3    case STANDBY:
4    if (state == RED)
5    {
6      state = RED_YELLOW;
7      handleStandby();
8    }
9    break;
10
11   /* ... */
```

As you can easily see, the switch is performed on **event** (line 1), in the case *STANDBY* (line 2) we check if our **state** is *RED* and then act accordingly (line 3-7). Basically it's the same code as the event-centric one with the roles of state and event exchanged.

Quickly summarized, I can see no general advantages of this approach over an event-centric one, since the only thing it does, is obfuscating the control-flow of the state-diagram. This should truly only be used if there is a large number of possible events and only a very small amount of different states. Otherwise you should always use the state-centric approach.

5.4 State Pattern

The main idea of the *state pattern* is to en capsule the full behavior of a **state** into a state-"Object". This includes the reaction on **events** and the transitions to the next states [1]. You can find my source-code in **state_pattern.c**:

```
1  struct StateClass
2  {
3    struct StateClass (*gotoNextState) (enum Event);
4  };
5
6  /* ... */
7
8  /* state-machine loop */
9  while (1)
10 {
11   event = checkEvent();
12
13   if (event != NO_EVENT) // do only if something happend
14   {
15     state = state.gotoNextState(event); // handle next state
16   }
17 }
```

Since there are no Objects in C, I used *struct* in my implementation (line 1).

The main advantage of this pattern is that the behavior of a single state is in a single Object and thereby can easily be modified and exchanged without effecting the rest of the code. In addition this results in a very short an clean state-machine loop.

5.5 Table Based

In the textottable-based implementation, the full behavior of the states is defined in one (or more) tables. (see *table_based.c*):

```
1  /* the state table containing a new state entry in [state][event] */
2  enum State stateTable[6][8] =
3  {
4    /* ... */
5  };
6
7  void (*actionTable[6][8]) () =
8  {
9    /* ... */
10 };
```

In my implementation, I used two tables, one for the state transitions (*stateTable*; next states) and one for the actions to perform (*actionTable*; function pointers) in a state.

The advantage is again, that the state-machine-loop is completely independent of the state and theirs actions itself. That allows you to create general implementations of the state-machine which can be used for multiple different occasions.

There is a pretty big disadvantage, namely that when there are a lot of states and events the code of the table soon becomes uninterpretable for non-coders.

5.6 Comparison & Conclusion

I am now going to evaluate the individual types of state-machines for this specific use-case of a traffic-light.

The *state-centric without transitions* gets eliminates rather quickly, since it is impossible to implemented the yellow-blinking light for reasons explained above.

In my opinion is *event-centric* in this use-case just inferior to the simple *state-centric*-approach and therefore also doesn't make the cut.

Table-driven and *state-statten* are simply too complicated to implement, so they are not worth it in that specific use-case. In addition they are IMO not readably at all for someone who hasn't written the code by himself.

So that leaves us with the good old **state-centric** type. For this specific use case this is IMO the best choice, since it is easy to implement, easy to understand and the code will probably not have to be expanded to a huge state-machine anyways. Finally, that would be my choice for a real implementation.

6 Problems

6.1 Unclear Requirements

The main problem, and also the reason why this homework took so long, was the very ambiguous problem statement.

In fact it told us to implement the different types of state-machines *as they were described in the provided book*, but which later turned out to *actually not represent the opinion of the instructor*. I.e. after quite some discussion on how to "really" implement the different types of state-machines, I had to redo basically all of them.

That cost me a lot of motivation and therefore resulted in a huge time-loss.

I sincerely hope that this will not happen again during this school year!

7 Testing

For testing I simply executed all different state-machine implementations on a Unix-System as described in the beginning of the document.

Here are the results of the tests in order (of the state-machines described in this report):

```
schueler@debian:~/repositories/state_machines$ make sm=STATE_CENTRIC
Running state-machine type: STATE_CENTRIC
Running the traffic-light infinitely. Press `CTRL + C` to terminate!
RED     ON
YELLOW  ON
RED     OFF
YELLOW  OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
YELLOW  ON
YELLOW  OFF
RED     ON
^Cmake: *** [run] Interrupt
```

```
schueler@debian:~/repositories/state_machines$ make sm=STATE_CENTRIC_WO_TRANSITIO
S
Running state-machine type: STATE_CENTRIC_WO_TRANSITIONS
Running the traffic-light infinitely. Press `CTRL + C` to terminate!
RED     ON
YELLOW  ON
RED     OFF
YELLOW  OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
YELLOW  ON
YELLOW  OFF
RED     ON
^Cmake: *** [run] Interrupt
```

```
schueler@debian:~/repositories/state_machines$ make sm=EVENT_CENTRIC
Running state-machine type: EVENT_CENTRIC
Running the traffic-light infinitely. Press `CTRL + C` to terminate!
RED     ON
YELLOW  ON
RED     OFF
YELLOW  OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
YELLOW  ON
YELLOW  OFF
RED     ON
^Cmake: *** [run] Interrupt
```

```
schueler@debian:~/repositories/state_machines$ make sm=STATE_PATTERN
Running state-machine type: STATE_PATTERN
Running the traffic-light infinitely. Press `CTRL + C` to terminate!
RED     ON
YELLOW  ON
RED     OFF
YELLOW  OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
YELLOW  ON
YELLOW  OFF
RED     ON
^Cmake: *** [run] Interrupt
```

```
schueler@debian:~/repositories/state_machines$ make sm=TABLE_DRIVEN
Running state-machine type: TABLE_DRIVEN
Running the traffic-light infinitely. Press `CTRL + C` to terminate!
RED     ON
YELLOW  ON
RED     OFF
YELLOW  OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
GREEN   ON
GREEN   OFF
YELLOW  ON
YELLOW  OFF
RED     ON
^Cmake: *** [run] Interrupt
```

Summarized, all tests **returned the expected** output.

Because of the way it was implemented , testing the state *YELLOW_BLINKING* was not possible.

References

[1] Eric Gamma and alt. *Entwurfsmuster*.

[2] Kernighan and Ritchie. *Programmieren in C*.