A Traffic Control Project for the Practicum

Problem Statement
This project will build a control system for traffic lights at an intersection. The system will use a design based on the Finite State Machine (FSM) pattern and will be demonstrated via a User Interface. A FSM language will be used to define any machine which can be executed by the controller.

Requirements
This document is NOT a full requirements document for the system. That will be developed by the student practicum team for approval by the instructor as part of the project. There will be high level or general directions stated here followed by constraints or design directions in the next section. The instructor will work with the team to formulate a full requirements document.

Feature Ideas
These are the initial ideas on how the system should work. These concepts will be refined into a final feature set as part of the requirements phase.

1. There will be a desktop application which is a programmable traffic controller that controls a traffic intersection with traffic lights for each direction and road sensors that cars would trip. The traffic sensors will be buttons for demo purposes.
2. The design will use a finite state machine (FSM) approach to constructing the controller logic. The design MUST first show a valid state transition diagram before coding can begin. This will be reviewed with the instructor. This will demonstrate that the students understand how the FSM works and thus how a controller can be programmed.
3. The controller will use a programmable FSM machine that will utilize a FSM language that can be loaded to define the logic. A simple language instruction set is provided.
4. The first test case to be built will be a simple intersection with a main road and one side road. There will be one road sensor (for the side road) as well as any timers needed. The rules for the traffic light operation are as follows:
    - The main road is green by default.
    - The side road light is red until a car appears. If the main road light has been green for more than 2 minutes, turn the main yellow for three seconds. Then turn the side road light green and the main road light red.
    - When the side road is green, keep it green for a maximum of 1 minute no matter how many cars appear.
    - When either direction shows a yellow light, keep it yellow for 3 second before turning it red and the other side green.

- On a “just miss” where the side road car trips the sensor during a yellow, wait for 2 minutes on main road green and then begin the cycle to turn the side road green.
- The team must first describe this setup in an FSM transition diagram, then in a set of FSM tables defined below and finally using the FSM language given below.

The final feature set will be determined as part of the requirements process. The amount of work to be committed will depend on the experience of the project team and the allotted time available in the semester. The instructor will serve as customer/user.

Design Constraints

The following are design constraints

1. The application will be coded using Java and the Java Swing GUI library.
2. The FSM state transition diagram will be critical to success. this must be completed and reviewed with the instructor before major coding can begin. Background coding, definition of GUI, etc can begin before this as determined by the project plan.
3. A good Object Oriented design is needed for ease of understanding, good practice and ease of extension. This will also serve to divide up the assignments for team members.
4. The user will supply some minimal test programs to execute. The team should also design and run some example test programs as part of their delivery.
5. The implementation will follow the suggested design approach given by the instructor. Since Finite State Machines are not taught at the undergraduate level, the first week or 2 of the term will be available to leave FSM theory.

Suggested User Interface

This is a GUI suggested interface. Other ideas can be discussed with the instructor as the design is approached. Since the FSM controller is generic and can be wired up according to the definitions in the language. the GUI must then be defined according to the definitions contained in the input program. The FSM controller will thus have N possible states, M possible output lines, S possible sensors, and T possible timers. These can be chosen and set up at compile time. Because the FSM is controlled by an input program, there is no preset traffic or road layout. Thus the display would likely be tabular in nature showing the state transitions, values of output lines, and values of timers. A GUI design proposal must be approved before coding can begin in earnest.

Traffic Controller - A Design using Finite State Machine Engine

The following is a design approach to be followed when building the traffic controller finite state machine. This approach defines a table based sequencer that will allow for the subsequent easy implementation of a language to define any traffic control or finite state machine actions via an input text file.

The basic approach will be a clocked sequencer design that uses 2 tables for the finite state machine definition. By storing all of the specifics into tables, any traffic control requirement can be met simply by changing the content of the tables. In addition, the tables will allow for a simple text based language to be used to provide a method to define, at runtime, any new control operation.

First we will define 2 tables - the output table and the state sequence table. The output table defines the value of each output of the machine at each state. There will be output lines and timer set values. Here is an example of the table:

State	Line1	Line2	Line3	Line4	Line5	Timer1	Timer2
1	0	0	1	0	1	0	120
2	0	1	1	1	0	0	0
3	0	0	0	1	0	60	0
4	1	0	0	1	1	0	0

In this example, we have a machine with 4 states, 5 binary output lines (on/off) and 2 timers. In an FSM, each state is logically defined by it's outputs. Thus in state 1, Line3 and Line5 are 'ON' . In addition, a timer is defined as a set state, i.e. a starting countdown. This as the state is entered, this value is set into a timer and a countdown started. When the timer expires, a state transition is triggered as seen in the state sequence table. The output lines are then wired or associated with the individual lights on each traffic light - the red, yellow, or green light. In a physical implementation, these lines would be wired to the lights to turn them on and off.

The state sequence table defines the transitions from state to state based on the set of inputs using the basic FSM concept: State[next] = function (state[present], inputs).

Current State	Sensor1	Timer1	Timer2
1	2	1	1
2	2	3	2
3	3	3	4
4	4	1	4

In the example, we have all possible state transitions specified for a 4 state machine and 3 inputs which in this case is 1 sensor and 2 timers. The first column shows all possible states. Then for each input, that column shows the next state given a firing of the sensor. For example, for a current state of 1, a sensor1 input causes a transition to state 2. Note that the timer input is implied as the expiration of a timer. As we saw above, as the machine enters a state, the timer is loaded with a value and it starts to countdown. When it reaches zero, the timer has expired and forms an input to the sequencer and the state transition as defined by the table is performed. The sequencer thus steps the machine through each state as triggered by inputs. As each new state is entered, the machine performs the setting of the output lines and sets initial values to the timers all as directed by the information in the output table. Note also that if the next state is equal to the present state, no state outputs need be set. A transition from one state to itself is thus treated as a “no-operation”. In this way, a non-transition prevents the timer from being reset to its initial value.

The sequencer can best be implemented as a simple clocked machine using 1 clock input. Sensors will be simple scanned devices with no latching or asynchronous operation themselves. Timers can be implemented as simply a countdown object. When the sequencer sees a timer reach a value of zero, it performs the state transition logic. The sensor is a scanned device; it is passive and performs no actions on its own. It is examined to see if it is on or off. It is also helpful to have a minimum period for the sensor, say three intervals before it is declared as ‘on’. So, for all types of inputs, the sequencer does all logic alone at each clock interval because each device is a passive device and does no direct operation itself.

As an example, consider a clocked sequencer with a clock of 1 second interval. The machine , by convention, starts in state 1. At every clock cycle, the sequencer would do the following:

1. Scan the sensor. If the sensor has been set for N periods (say 3), set the sensor to “ON’. If the sensor is not set, set it to “OFF”.
2. For each timer, check if the timer is active (i.e. has a non-zero value). If so, decrement the timer. If the value now becomes zero, the timer has fired.

3. Now consult the sequence table. Given the current state and any newly changed sensors and timers, execute a state transition.
4. If the new state is different from the old state, use the state output table to define the new values of the output lines and jam in new initial values for the timers if non-zero.
5. Return to idle waiting for the next clock tick.

Using this approach, we can see that we can implement a simple clocked FSM machine.

A Finite State Machine Definition Language

It is highly desirable to have a text based definition for a Finite State Machine sequencer. In this way, a text based description can be compiled into a binary set of tables that will allow a FSM sequencer to operate any machine without the need to reprogram or recompile the basic sequencer. The language will generate values for the 2 tables needed for a FSM as defined in the memo - A FSM design.

There are a few basic statements needed for the language:

Statement	Example	Purpose
# comment	# This is a comment	Comment line - ignored by system
define	define MAX_TIME 120	Assignment of a constant to a pneumonic
states	states IDLE WAIT GO	Defines all possible states. By convention, first state is initial state.
outputs	outputs MR MY MG SG SY SR	Defines all output lines
timers	Timers YELLOW MAXON	Defines all of the timers
sensors	sensors SIDEROAD	define sensors
at	at IDLE MR=1 SG=1 MAXON=MAXTIME	Defines state of outputs at a given state
when	when SIDEROAD WAIT > GO	Defines a state transition on firing of sensor or timer

The ‘comment’ command simply allows for commentary in the specification. The ‘define’ command allows for variable specification through the definition so that constants can be easily changed throughout the FSM spec.
The commands - states, outputs, timers, sensors - define the full set of items to be used. These are all defined up front so that the tables can be built by the compiler before population. The ‘at’ command specifies the state of output lines and timer initial values as the machine enters that state. This command is thus used to populate the state output table. SInce the outputs can all initially be set to zero, only the ‘on’ values need be specified for any given state. Finally, the ‘when’ command specifies the state transition on an event. Again, only transitions that cause a state change need be specified.

In order to best show the operation of the language, an example follows. The set of language statements is presented and then the corresponding output tables are given for that machine.

```
# This is a simple FSM example. No attempt is made to validate this machine
define WAIT_TIME 25
define SAFE_TIME 3
define ON 1
states  IDLE WAIT GO
outputs L1 L2 L3 L4
timers  GUARD
sensors RESET

at IDLE L1=ON GUARD=SAFE_TIME
at WAIT L2=ON GUARD=WAIT_TIME
at GO L3=ON L4=ON

when RESET WAIT > IDLE
when RESET GO > IDLE
when GUARD WAIT > GO
when GUARD IDLE > WAIT
```

Output Table

State	L1	L2	L3	L4	GUARD
IDLE	1	0	0	0	3
WAIT	0	1	0	0	25
GO	0	0	1	1	0

State Transition Table

State	RESET	GUARD
IDLE	IDLE	WAIT
WAIT	IDLE	GO
GO	IDLE	GO