MECHENG 313

Assignment 2: Object-Oriented Programming of Real-Time Systems

Overview

Real-time systems (also known as “cyber-physical” systems) are usually comprised of three parts: a controlled object, a real-time computer, and a user interface to that computer[1]. There are several functional requirements of any real-time system: A real-time computer must inform the user of its state and the state of the controlled object, thereby assisting the user in the decision-making process. Data collection -- another requirement -- logs interactions between user and real-time computer, as well as changes in the state of the computer.

In this assignment, you will develop a C# application that implements finite-state machines, data logging, and threading, as well as a simple console-based user interface.

Task 1: Implementing a class for finite-state tables

In lectures, we discussed the “finite-state table” technique which can be used to implement finite-state machines. Using C#, write a definition for `class FiniteStateTable`.

Your class definition should include the following variables:

- A two-dimensional array named `FST`
- A `struct` named `cell_FST` which groups the variables that comprise a cell of the finite state table (specifically, an index to the next state, and the action(s) associated with the transition)

Your class definition should include the following methods:

- `SetNextState()` and `SetActions()`, which can be used to set the contents of any cell (`cell_FST`) comprising the finite state table (`FST`)
- `GetNextState()` and `GetActions()`, which can be used to return the contents of any cell comprising the finite state table

[1] Kopetz (2011), Real-Time Systems: Design Principles for Distributed Embedded Applications, Springer.

Your class definition should also include any useful constructors.

Task 2: Programming a finite-state machine

Using C#, develop a console application that implements the finite-state machine represented in Figure 1. If it proves helpful, use your definition `class FiniteStateTable` from Task 1. (Note, however, that it's not necessary to use that class definition if, indeed, your design doesn't require it.)

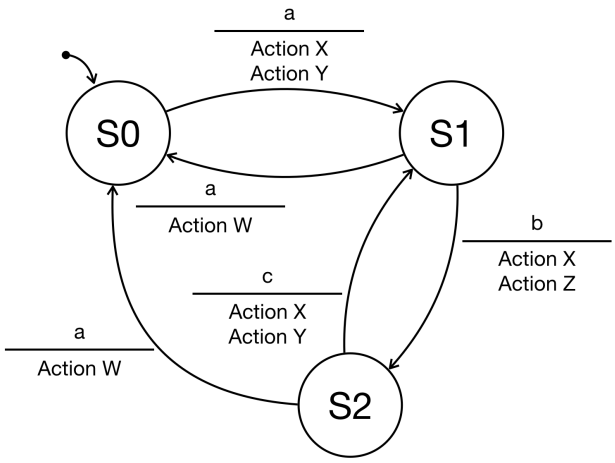

Figure 1. This finite-state machine's transitions are triggered by keyboard events 'a', 'b', or 'c'. All transitions are associated with at least one action.

The application's user will interface to the machine using the keyboard; to trigger the machine, the user presses keys 'a', 'b', or 'c'. When the machine transitions to a new state, this transition should be reported to the user (e.g., when the machine transitions from state S0 to state S1, the string "Now in state S1" should be written to the console). The machine executes actions by writing to the console (e.g., when the machine executes "Action A", the string "Action A" should be written to the console). The key 'q' is used to quit the application.

Your application should log the activity of both user and machine, timestamping trigger events and actions, and writing both to a file. On quit, your application must write the log to file; prompt the user to provide a fully-qualified file name (e.g., on Windows, "c:\temp\log1.txt", or on Mac or Linux, "/tmp/log1.txt"), and notify the user of successful write completion. Your program must appropriately handle erroneous user input.

If necessary, you may develop the functionality of your `class FiniteStateTable` (Task 1).

Task 3: Programming concurrent, dependent, synchronous finite-state machines

Further develop your code from Task 2 so that your console application now implements two concurrent, dependent, synchronous finite-state machines. The first finite-state machine is as in Task 2 (Figure 1). The second finite-state machine is represented below (Figure 2). Again, the user of your application will interface to the machines with the keyboard; when either machine transitions to a new state, this transition should be reported to the user; machines execute actions by writing to the console; the key 'q' quits the application; and your application should log the activity of both user and machines.

As will be discussed in lectures, multi-threading is a feature of C#, maximizing CPU utilization where two or more actions are executed at the same time. In Task 3, program your application so that, when the machine represented in Figure 2 transitions from state SB to state SA, actions J, K, and L are executed using three separate threads.

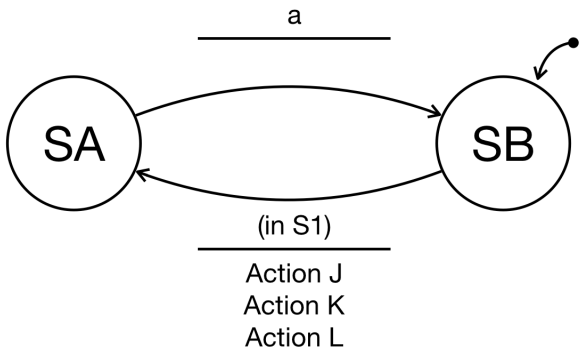

Figure 2. This finite-state machine's transition from state SB to state SA can be triggered by any keyboard event, 'a', 'b', or 'c'. However, there is a simple dependency between this machine and the concurrent machine represented in Figure 1; the transition from state SB to state SA is only triggered if the concurrent machine is in state S1.

Marking Criteria

This assignment is worth 20% of your final grade for MECHENG 313.

Software (15%). Your code will be assessed on its readability (good commenting and clear, consistent style) and accuracy. You are expected to use high-level features of C#, like those described in lectures. TAs will run the code you submit on Canvas, using their own installations of Microsoft Visual Studio, checking that your implementation adheres to the above-described specifications.

Report (5%). Document your `class FiniteStateTable`. Your documentation should be easy to read, and may include the following subsections: (1) an overview of the class, (2) code exemplifying its use, (3) a synopsis of class variables, methods, and constructors, and (4) additional remarks that you think might be useful to anyone implementing your class. Overall, your documentation should be short-and-sweet, preferably 2 pages including any necessary code listings and/or images (12pt font, single-spaced), but certainly no more than 5 pages.

Files to Submit

Files are to be submitted on Canvas.

Software submission. One submission per group. Please submit a zip file. This zip file should contain three C# source code files corresponding to your definition of `class FiniteStateTable`, your solution to Task 2, and your solution to Task 3, respectively. Include a "README.txt" text file briefly explaining the contents of each source code file, as well as a list of names and UPIs of group members.

Documentation. One submission per group. Please submit a PDF.

Peer Review. One submission per person. Please use the form provided.