CS2303 In Class Exercises

Jan 28, 2020

February 1, 2020

Abstract

Today's goal is that you can use the state machine paradigm as one of your design tools. Use means that the thoughts you have at the level of abstraction of the state machine guild you to writing your code.

Name and WPI email: Kush Shah kshah2@wpi.edu

1 Applying State Charts

The first step in applying this paradigm is to draw the diagram.

Examples:

See Figure 1.

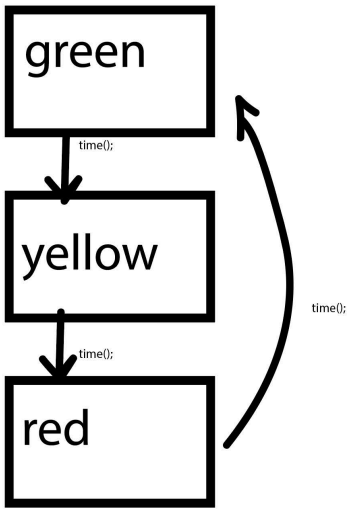
Note in the first example above that there are three different states, or modes, and that they are mutually exclusive. They form a partition, which is to say, any element of the set (in this case, a time instant) is found in exactly one of the compartments (states).

The second example depicts buying lunch.

The third example depicts switching between in class and between classes, according to the time of day, during normal time, and the response to a fire alarm.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Draw a state diagram for an ordinary traffic signal. Draw the states, and the arrows showing possible state transitions. Then, label the arrows with the item that provokes the transition.




```
 8 -     if power
 9 -         switch currentState
10 -             case "off"
11 -                 currentState = "seeking";
12 -             case "seeking"
13 -                 currentState = "off";
14 -             case "tracking"
15 -                 currentState = "off";
16 -         end
17 -     end
18
19 -     if strcmp(currentState,"seeking")
20 -         foundObject = false;
21           % Use seeking algorithm
22 -         while ~foundObject
23 -             foundObject = seekObject();
24 -         end
25 -         if foundObject
26 -             currentState = "tracking";
27 -         end
```

Figure 1: A state machine diagram helps us structure our code. It is a higher level of abstraction, which makes us faster, and less error prone.

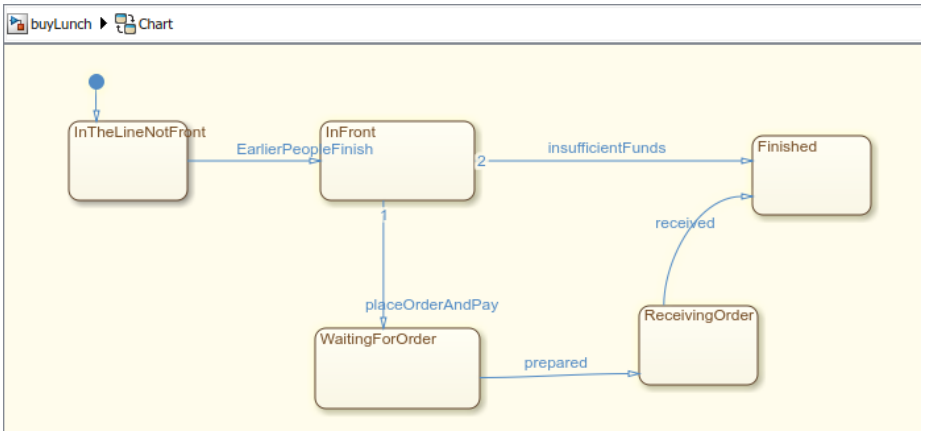


Figure 2: Two routes away from a state

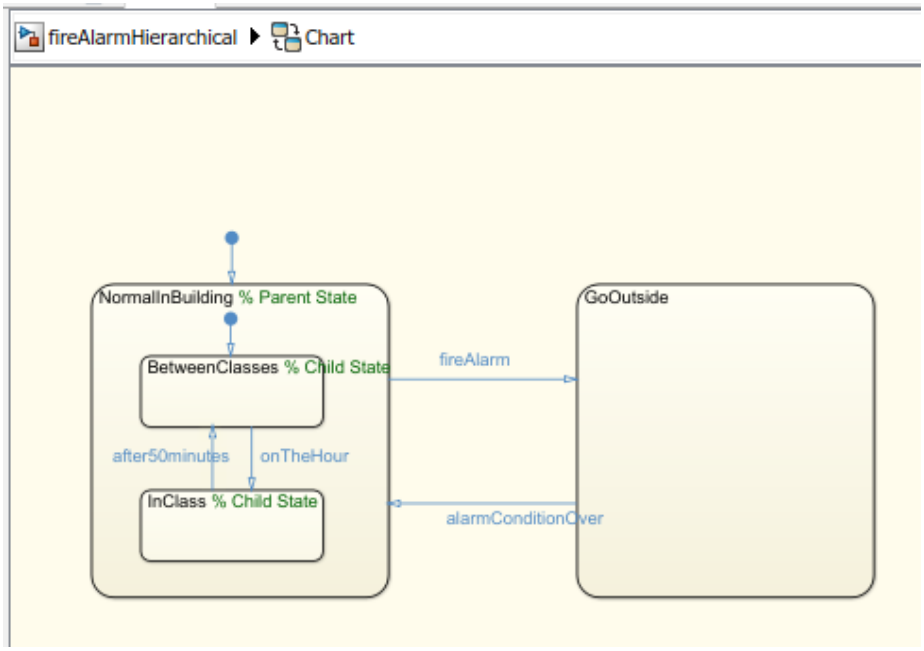


Figure 3: Hierarchical state charts have states within states.

Recall the switch/case construct offered in C. Recall the typedef enum keywords.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Define an enumerated type, such that each state of your traffic light is a element of the set described by your enumerated type.

```
typedef enum{
Green = 2,
Yellow = 1,
Red =0
}LightStates;
```

```
LightStates l;
```

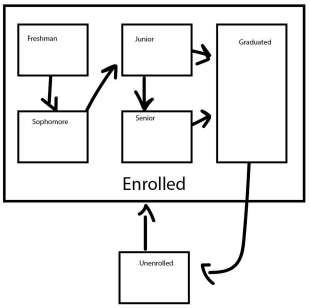
2. Write a switch/case construct that has one case for each of your values in your enumerated type (also include the default case).

```
switch(l){
case 0:
puts("red");
break;
case 1:
puts("yellow");
break;
case 2:
puts("green");
break;
default:
puts("Light System");
break;
}
```

Stages of a progression can be looked at as a state machine.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Draw a state diagram for moving through a four year undergraduate program. Draw the states, and the arrows showing possible state transitions. Then, label the arrows with the item that provokes the transition.



2. Augment your diagram to switch to BS/MS. Draw the states, and the arrows showing possible state transitions. Draw the diagram as if it were possible to switch into BS/MS at several points. Then, label the arrows with the item that provokes the transition.

https://www.mathworks.com/videos/using-state-machines-part-1-supervisory-control-98578.html

2 Learning Design Style from Good Examples

These algorithms are chosen for you to observe, not to memorize. They are beautiful in a way, and examining them might help you develop a sense of well-written functions. Then you can use that sense as you write your own functions.

2.1 QuickSort

In your algorithms course you will have the opportunity to see good examples. Here is one:

```
algorithm quicksort(A, lo, hi) is
if lo < hi then
p := partition(A, lo, hi)
quicksort(A, lo, p)
quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
pivot := A[(hi + lo) / 2]
i := lo - 1
j := hi + 1
loop forever
do
i := i + 1
while A[i] < pivot
do
j := j - 1
while A[j] > pivot
if i >= j then
return j
swap A[i] with A[j]
```

The first part,

```
algorithm quicksort(A, lo, hi) is
if lo < hi then
p := partition(A, lo, hi)
quicksort(A, lo, p)
quicksort(A, p + 1, hi)
```

calls partition to split the items to be sorted into two groups, and then recursively calls itself on the two groups.

Once we know that partition arranges for all of the members of the first group to be less than any of the members of the second group, we see how, at a high level, this recursive sorting is going to go. Eventually the groups will be so small, they are sorted. So, we should then see how partition works.

```
algorithm partition(A, lo, hi) is
pivot := A[(hi + lo) / 2]
i := lo - 1
j := hi + 1
loop forever
do
i := i + 1
while A[i] < pivot
do
j := j - 1
while A[j] > pivot
if i >= j then
return j
swap A[i] with A[j]
```

First a pivot is chosen. Variations on this algorithm consider different methods of choosing a pivot. When the chosen pivot is good, better performance results.

Two cursors, i and j are initialized; i on the low side, and j on the high side. Then i moves to higher positions in the array, until it encounters a value in the array that exceeds the pivot. Next, j moves to lower positions in the array, until it encounters a value in the array that the pivot exceeds. If these resulting values of the cursors i and j still have i to the lower side of j, the values in the array where i and j are pointing are swapped. Then, i and j continue as before, possibly identifying another pair of values to be swapped. Eventually i and j will meet.

When i and j meet, all the values of array elements to the left of where they meet are lower than any of the values to the right of where they meet. This is the behavior needed from partition.

The purpose of this exercise is to give you practice with the code. Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Create an array of length 8, and place values in the array in unsorted order. Show successive iterations of performing quicksort on it.

3,8,7,4,6,1,5,2
pivot = 4
i = 3
i = 8
j = 2
3,2,7,4,6,1,5,8
i = 7
j = 5
j = 1
3,2,1,4,6,7,5,8
pivot = 2
1,2,3
pivot = 5
i = 6
j = 5
5,7,6,8
pivot = 6
i = 7
j = 8
j = 6
6,7,8
1,2,3,4,5,6,7,8

2.2 Splay Tree

Recall that we have looked at searching through a graph; we made the graph into a tree as we worked on it. In that context, there was no requirement that the values at the nodes were sorted into any particular order.

A special case of a tree (that, a subset of trees) has the property that the values contained in left child nodes are less than the values contained in right child nodes. When this is true, we can arrive at a node, and search at most one of the child subtrees, knowing the value we seen can only be on one of the sides. These are called search trees. There are several kinds of search tree.

One kind of search tree is a splay tree.

In the case of a splay tree, concern for how long the search may take is present. To perform well, the tree should remain at least approximately balanced, which is to say, the depth of the tree is about the same, no matter which leaf we choose.

We are thinking about the tree data structure being dynamic, that is, items get added to the tree, items get deleted from the tree, and all the while, we wish the tree to remain roughly balanced. There are operations that can be performed on a search tree.

`https://en.wikipedia.org/wiki/Splay_tree`

The purpose of this exercise is to give you practice with the code. Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Draw an instance of a binary search tree containing 8 nodes.
2. Perform a zig step.
3. Perform a zig-zig step.
4. Perform a zig-zag step.