What Are State Machines and How Can They Be Helpful in Botball?
Tim McCabe
Dead Robot Society
tmccabe@vt.edu

What Are State Machines and How Can They Be Helpful in Botball?

1. Intro to State Machines

What are state machines? State machines are a structure of code that is made up of states. States are small blocks of code that perform very specific tasks and are linked together through the main program. According to the National Institute of Standards and Technology[1] a state machine, or finite state automaton, is a method of computation involving states, a start state, and input. Even the most complex task for the robot can be broken down into a series of states with very simple programming in each state, insuring that each state is very small. Each state has small amounts of code that allows for a much greater understanding of the program. State machines are very good for Botball [2] because they make the code easy to read, test, and fix.

1.1 How State Machines can be used in Botball

State machines allow for the code to be written one action at a time. Each state is written with only one simple action such as raising an arm on a servo or driving the robot straight. Each state then returns a variable to the main program which uses that variable to lead into the next state. The main program is a series of if() statements that read the variable from the previously run state and then run the state that the if() statement leads into it (see section 2.1 for an example). As the code grows to include more actions or tasks, the state machine can grow to include more states. If a large section of code is filled out in the main code (adding in several states before the state is actually written), the programmer may run into the problem of calling a state that does not exist yet. This problem can be overcome by placing a beep() command at the end of the main program right before the closing brackets of the while statement that contains the state machine; this can be seen at the bottom of the state machine in section 2.1. This produces a repetitive beeping noise that alerts the programmer that a non-existing state has been called. This beeping is caused because the non-existing state is not present in the state machine so the main program continuously loops through the code looking for the state. Since the beep() is inside of the closing brackets, a beep is produced every time the main program goes through the code looking for the non-existing state.

1.2 Why State Machines Are Helpful

A large advantage of using a state machine is the ability to reuse entire sequences of code. If a robot has to repeat the same task for the entire given time then the ending state simply returns the variable for the top state. Finally, the entire sequence of actions will repeat itself until the robot is shut down due to a programmed timer or some mechanical stop. Repeating a sequence can also be done inside of the code if the ending of the final repetition triggers a sensor leading to a

different return. States can also be very helpful by allowing the programmer, or another member of the team, to easily create a diagram of the robot's actions on the game board with lines representing each state. This allows for a visual representation of what the code is supposed to be doing and how close it is to completion.

1.3 Why State Machines Are Useful For Testing

State machines are very helpful for many reasons. State machines compartmentalize the code. This compartmentalization allows for many programmers to modify and build on the same code. Each new programmer can read the name of the state and know what the state is doing, and they also only have to sift through a small number of lines of code to fully understand the state. Compartmentalization is also very helpful because it makes the code easy to test. Rather then having to run the entire code from the beginning to end in order to check something, the first state variable, which tells the main program where to start, can be changed to the nearest known point to where the action occurs. This makes both the testing and modification of the code very convenient. If the programmer would like to modify a specific part of the code, he can make his change and then run the state that the modification where made. This ensures that the change did in fact work and not just make the problem worse. State machines can also help with finding the location of the error in the code. When a print statement is placed in each state the programmer can know exactly which state that the error is located in, and then only has to look through five to ten lines of code instead of twenty to thirty. State machines are also very helpful because states can easily be removed from the program. If, during competition, the team decides that a state isn't working for some reason, it can simply be removed from the main program. State machines make the code much easier to read overall. Having the code separated into small simple segments and having clear names for each state makes the code very readable. An example of a suitable name for the state would include the action being done as well as the initial direction that the bot is in at the start of the state. The direction portion of the state name allows for an easier visualization of where the robot is in its sequence of actions. It may also help in the placement of the robot for testing when starting from a state other then the first state.

2.1 Example of a State Machine

This is only an example of the state machine and does not compile.

These are globals to define states. Globals are integers to hold the input of the state machine:

```
//globals
int state;     //main variable for state machine
```

This is the actual state machine:

```
void main(){
   while(state != finished_U){
      if (state == startPos_E)           state = getOutOfStartbox();
      if (state == outOfStartBox_E)      state = goToBridgeLine();
      if (state == atBridgeLine_E)       state = turnToBridge();
      if (state == facingBridge_N)       state = knockDownBridge();
      if (state == bridgeDown_N)         state = crossLine();
      if (state == crossedLine_N)        state = crossBridge();
```

```
    if (state == acrossBridge_N)        state = turnToBridgeLine();
    if (state == atBridgeLine_NW)       state = sweepGreenTribble();
    if (state == atBridgeLine2_NW)      state = disturbTribbles();
    if (state == atCenterOfPiles_W)     state = goToCups();
    if (state == atCups_W)              state = driveIntoShelter();
```

This loop helps the programmer realize, during testing, if a non-existent state is called:

```
    //beep signals infinite loop
    beep();
  }
```

This function shuts down the create and ensures that all states is completed and turned off:

```
  shut_down();
}
```

2.1 Example of a State

```
int knockDownBridge(){
```

prints the function of the state so the programmer knows what the robot is accomplishing:

```
  printf("\n knock down bridge");
```

runs the simple code of the state:

```
  moveServoSlow(ARMS_MID);    //move arms to knock down bridge
  msleep(750L);
  moveServoSlow(ARMS_UP);   //move arms back up
  msleep(500L);
```

returns the variable of the next state:

```
  return bridgeDown_N;
}
```

3. Why State Machines Are Good For Botball

State machines are very useful for programming robots because they have many benefits and few pitfalls. They are not much harder to write then standard code and they allow for much easier use of the code overall. State machines make programming easier because even the most complex programs can be written out in small simple steps. These small steps allow the code to easily be tested, fixed and understood.

References

[1] National Institute of Standards and Technology. “Finite State Machine”. [Online] 12 May 2008. URL <http:// www.nist.gov/dads/HTML/finiteStateMachine.html>.

[2] KIPR. Botball robotics competition. http://www.botball.org, 2002.