## Learning Objectives

- To reimplement tasks using a higher-level finite state machine framework
- To compare and contrast if-statements and the FSM framework in terms of testability
- To familiarize yourself with the TerraBot hardware

Download and unzip the assignment code from Canvas. Place the folder in `~/Desktop/TerraBot/agents/` and copy (**don't link!**) the files `greenhouse_agents.py`, `layers.py`, and `ros_hardware.py` from your ROS_HW directory. If you have not successfully completed the Architectures and ROS assignment, contact us and we can supply you with the refsol (after the final deadline for submitting that assignment has passed).

In this assignment, you will use a finite state machine framework called `pytransitions` to reimplement the tasks that were provided to you in the previous assignment. This new framework will allow you to extend your tasks and test more efficiently if they are properly implemented. `pytransitions` has extensive documentation that you will likely find helpful for this assignment:
https://github.com/pytransitions/transitions/blob/master/README.md.

The lecture slides also have examples of how to use various features of `pytransitions`, including `add_transition`, the use of triggers, `conditions` and `unless` conditions (which are the negation of `conditions`), and `before` and `after` actions. In addition, you may find `on_enter` and `on_exit` actions useful, as well.

`pytransitions` is already loaded onto your virtual machine if you are using it, but if not you should follow the brief instructions at the top of the README to install it onto your Ubuntu distribution before continuing.

## Part 1: Autonomous Coffee Maker (30 points)

Before we edit the greenhouse tasks, let's build a small autonomous coffee maker for a smart home to get you started. Your coffee maker has a pod presence sensor, 3 different size button sensors (although only one can be selected at a time), a water temperature sensor, and a start button sensor. **For this part, you will be editing the `coffee_maker.py` file.**

The smart coffee maker will wait for a coffee pod to be inserted and a coffee size to be selected (in either order). Then, when both of them are sensed, it should wait for a start button to be selected. The user can remove the pod or select a different size coffee at any point before start is selected. When the start button is selected, the coffee maker immediately starts heating water (i.e., outputs the `START HEATING` message) and waits for it to sense that the water is 180 degrees Fahrenheit. Once it reaches 180 or higher, it should output the `START DISPENSING` message and then it can dispense the coffee for a particular duration of time, depending on the selected size (5s for small, 10s for medium, and 15s for large). When the machine is done dispensing after the designated amount of time, the coffee maker should output the `END DISPENSING` message and wait to sense that the pod is removed before resetting to empty. Note that when the pod is removed, the size button is reset to False as well.

Make sure to handle cases when the pod is removed after inserting it and when different size buttons are selected *before the start button is selected.* Once the start button is selected, your coffee maker cannot go back - it should immediately start heating water and waiting for the temperature to be correct. You do not have to handle the case where size buttons are completely deselected before the start button is pressed (i.e., once a size button is selected, a different size button can be selected instead – only one size selected at a time – but you can never transition back to a state with NO size selected).

### Step 1:

Draw the FSM on a sheet of paper. Label the meaning of the states and the sensing conditions for the transitions. Our refsol FSM has 7 states and 10 transitions that change the state in some way (the rest are self-loops). You are free, though, to implement it with more states and transitions, as long as it meets the stated functionality given above.

## Step 2:

The file `coffee_maker.py` defines the `CoffeeMaker` class with a dictionary of sensor data that will be updated through a sense function, a list of states as strings, and a single trigger `doStep`. When a new `CoffeeMaker` is initialized, it creates a new instance of the `Machine` class (an FSM) with an initial state `empty`. You should create distinct strings representing the different states within your FSM and add them to the list of states. Within `__init__()`, you should add transitions between your states that trigger based on defined conditions, where the conditions are functions that return either `True` or `False`. For the coffee maker FSM, implement the needed condition functions within the `CoffeeMaker` class. If you need *conjunctions* of conditions, you can read about them in the `pytransitions readme`.

As you think about your FSM, you will likely need to update state as a result of certain transitions (e.g., remembering which size was selected). You should use `before` or `after` arguments in the `add_transition` function to indicate a function that you want executed after the conditions are checked, but before/after the state transition occurs. These functions do not return anything, but can be used to save state to your `CoffeeMaker` class variables. **Do not put any state-saving code in the condition functions.**

Additionally, you should use `after` arguments to "publish" the message `START HEATING` when the maker starts heating up, `START DISPENSING` after the coffee maker senses that the water is 180 degrees, and `DONE DISPENSING` after it finishes dispensing the coffee for the correct duration, using the `publish` function defined in the `CoffeMaker` class. We will grade your FSM based on whether these messages appear and how many seconds they are apart from each other.

**General Notes:**

- You may notice that we initialized the FSM with `ignore_invalid_triggers=True` so that it just executes a self loop if a trigger occurs when no transition condition is met (thus, you don't have to explicitly define all the self loops).
- You can assume that a button remains pressed until a different button is pressed or the coffee maker returns to the empty state.
- To handle elapsed time for dispensing coffee, use `sensordata['unix_time']` for time, in seconds, since Jan 1 1970. **Do *not* use Python's built-in time functions!**
- `pytransitions` comes with a logger that writes state transitions to a file. We have augmented that to also log sensordata updates and actions taken. You can use `output-cm[datetime].log` to debug your work in addition to printing `self.machine.state`.
- The only place you should have `if` statements in your code is in the `condition` functions, the `perceive` function, or any auxiliary functions that those functions call. You should not have `if` statements anywhere else in your code. We will deduct points from your assignment grade if you do. Additionally, the only places that you should save state or send publish commands are in the `before, after`, or `perceive` functions, not in the conditions. We will deduct points from your assignment if you do.

To test this part, you can run the autograder using `python autograder.py -p1`. You can run a specific test using `python autograder.py -p1 -t[n]`, where n is an integer from 1-11 (don't include the brackets!).

**Testing Notes**:

- The autograder produces a log file that it uses to grade the tests. By default the file is deleted after a run, but if you want to save it to analyze your agent's behavior (see above), use the -k/–keep option.
- The `coffee_maker.py` file that we provide as a baseline should score 14 points when tested, without any modifications. That's because there are tests where the agent is not supposed to do anything, and that's exactly what the baseline code does – nothing.

## Part 2: Greenhouse Tasks (50 points)

Don't forget to copy **don't link** the files you modified in the ROS assignment (`greenhouse_agents.py, layers.py,` and `ros_hardware.py`) into the FSM_HW directory before beginning this part. If you have not successfully completed the Architectures and ROS assignment, contact us and we can supply you with the refsol (after the final deadline for submitting that assignment has passed). **For this part, you will be editing the `greenhouse_behaviors.py` and `ping_behavior.py` files.**

This part is testable without ROS, but does require `limits.py`, which is found in the `agents` directory. If you are not using a virtual machine, make sure that the `agents` directory is on your `PYTHONPATH`.

Use what you learned in Part 1 to reimplement each of the seven behaviors (`Light, RaiseTemp, LowerTemp, LowerHumid, RaiseSMoist, LowerSMoist,` and `Ping`) using seven separate FSMs. Put each FSM in each respective task class. Your FSMs should result in the same greenhouse actuation as the if-statement versions, though it need not repeatedly send action commands (e.g., the old behaviors would send the "wpump on" command over and over until it turns it off, but your behavior should send the command only once, at the right time).

We have given you a bit of head start by defining the `enable` and `disable` functions. The `enable` function invokes the "enable" trigger, which should transition the FSM out of the initial state. Conversely, the `disable` function uses the "disable" trigger, which should transition the FSM *into* the initial state. For all other transitions, the `act` function uses the `doStep` trigger.

As in the coffee maker, you should publish action messages only in `after` or `before` functions (or `on_enter`, `on_exit`), so your `doActions` functionality previously in the `act()` function should instead be called from those functions. Instead, the `act` function calls the `doStep` trigger, as indicated above.

You are strongly encouraged to base your behavior FSM's on the if-else behaviors found in Assignment 1 (ROS_HW). As with Part 1, you are strongly encouraged to draw the FSM on paper first, to understand states, transitions, and actions that are needed. This is especially true for the `RaiseSMoist` behavior, which is quite complex. Note that, in particular, the autograder expects that your `RaiseSMoist` behavior is implemented exactly based on the given if-else behavior. If it differs at all, then it may not pass the test cases even if it is correct. Please let us know if it (or any other behavior) isn't passing and we can manually grade it for you.

You may find the `self.state` variable in the old behaviors helpful in determining what FSM states you may need, however the design is up to you. **You may not use `self.state` in the new file because the `Machine` class uses that variable to save the state of the FSM**.

**General Notes:**

- Don't forget to update `self.states` to include the list of the states you are using for the FSM. Also, in your `add_transition` calls, use only the triggers `doStep, enable`, or `disable` .
- You should add any additional functions that you need in your behavior classes.
- You may find it helpful to copy some of the functionality from the coffeemaker example into your greenhouse behaviors. For example, the logging may come in handy for debugging (we will not grade any of those log files).
- **Remember: the only `if` statements you should have are in the `condition` functions, to check sensor data**, or in the `perceive` function (or in functions that are called by them). We will deduct points from your assignment if you use them elsewhere. **The only places that you should save state or send publish commands are in `before, after, on_enter, on_exit,` or `perceive` functions**.
- Remember, **do *not* use Python's built-in time functions!** Instead use `sensordata['unix_time']` for the time, in seconds, since Jan 1 1970, or `sensordata['midnight_time']` for the time, in seconds, since midnight (note that this value rolls around from 86400 to 0 every 24 hours!).
- Don't forget to write the start and pause functions and triggers. Think about what state information should be saved and what actions should be sent when these functions are called.

- When disabling a behavior, you should send it commands to turn off any actuators it was using, to ensure the physical system is returned to some known state. To be safe, you may want to do the same when enabling the behavior.
- In general, check what how ROS_HW behaviors act when enabled or disabled, and incorporate those actions into your FSM – **do not modify the FSM `enable` or `disable` functions, themselves!**

To test this part, you can run the autograder using `python autograder.py -p2`. As with Part 1, you can run a specific test using `python autograder.py -p2 -t[n]`, where n is an integer from 1-7 (don't include the brackets!). Specifically, test 1 is `Light`, 2 is `LowerTemp`, 3 is `LowerHumid`, 4 is `RaiseSMoist`, 5 is `LowerSMoist`, 6 is `RaiseTemp`, and 7 is `Ping`.

## Part 3: Testing on ROS (10 points)

Use the `assignment2.tst` file to confirm that all of your behaviors are still passing all the tests. In one Terminal, run `python TerraBot.py -m sim -s 1000 -t [path/to/file]/assignment2.tst` In another Terminal, run `python greenhouse_agent.py -L -m sim`.

You don't need to turn in anything for this part, but we will grade you on whether your behaviors work as expected (recall from Assignment 1 that there may still be occasional test failures – we know what to expect).

## Part 4: Using the TerraBot Hardware (10 points)

The first grow period starts on **October 2**, and in preparation we want you to familiarize yourself with the TerraBot hardware. For this part, you will be using the `interactive_agent.py` (found in `TerraBot/agents`) to manually control the actuators and read the sensor values.

**Step 1:** Carefully read the instructions in the file `Using the TerraBots.pdf`, found on the assignment 2 page and the `TerraBots` module. Log into your TerraBot (please use only the TerraBot that your group as been assigned to) and run the `interactive_agent`. Type "h" at the command line to see the actions available to you. Try turning the lights and fans on and off (don't turn the pump on, just yet), and type "v" is see the sensor values and how they change.

**Step 2:** Using the "c <filename>" command, take a picture from the TerraBot camera and copy it (using `scp`) to your own computer. Use any available display tool to look at the image and admire your work!

**Step 3:** With the lights and fans off, record (manually) the various sensor values every 10 seconds, for about a minute. Now, turn the pump on for 10 seconds and then turn it off. Once again, record the sensor values every 10 seconds, for another 60 seconds.

**Step 4:** Do the same with the fans: record sensor values every 10 seconds for 60 seconds, then turn the fans on for 30 seconds this time, and then record sensor values every 10 seconds for 60 seconds.

**Step 5:** Turn the lights on at a level of 50, wait 10 seconds, and record the sensor values. Continue doing the same thing with levels of 100, 150, 200, and 250.

For steps 3-5, analyze the change in sensor readings over time and write up your analysis in a `readme.txt` file. The analysis should indicate why you think that certain sensor values did, or did not, change as expected. If they changed as expected, report on what you think caused those changes; if they didn't change as expected, speculate on why.

In addition, include reflections on how you might use these characterizations of how the sensor values change over time to adjust your agent's behaviors, to better match how things work in the real world.

## Submission

Submit your `coffee_maker.py`, `greenhouse_behavior.py`, and `ping_behavior.py` files. When put in a clean directory with relevant files from Assignments 1 and 2, they should run without error. In addition, submit to Canvas the `readme.txt` file and the picture that you took in Part 4.