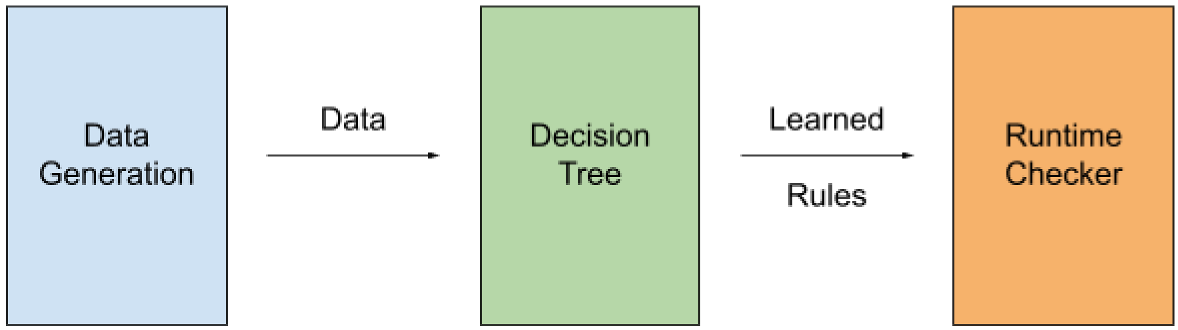
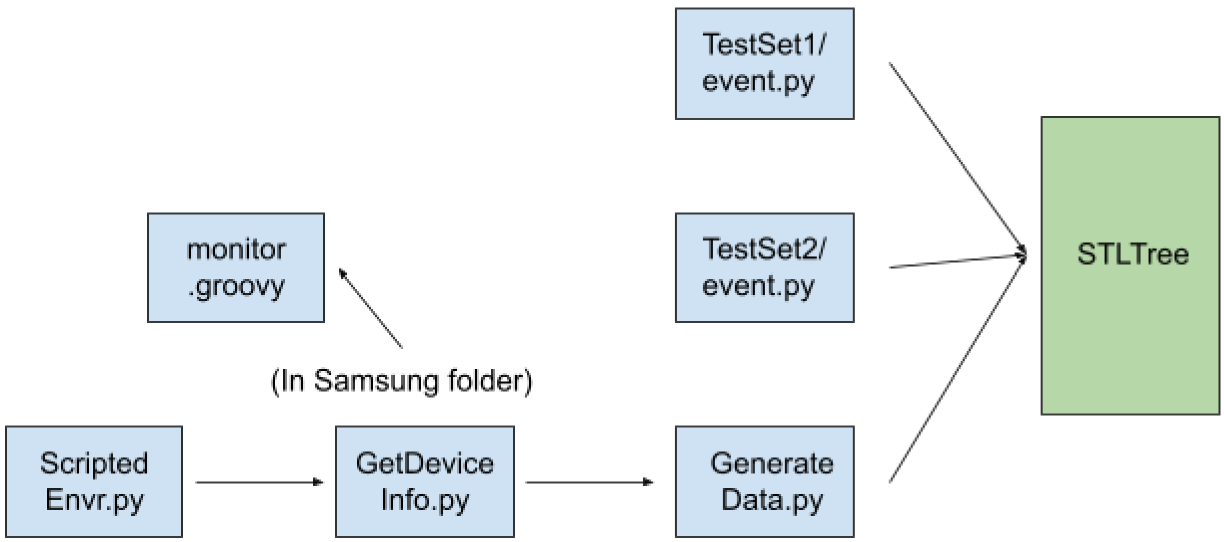
Current Workflow:



Data Generation:



TestSet1 and TestSet2 represented how we initially generated data for our STLTree model. In event.py, we utilized a state machine with hardcoded devices and rules to generate data for an environment with such devices at each timestamp.

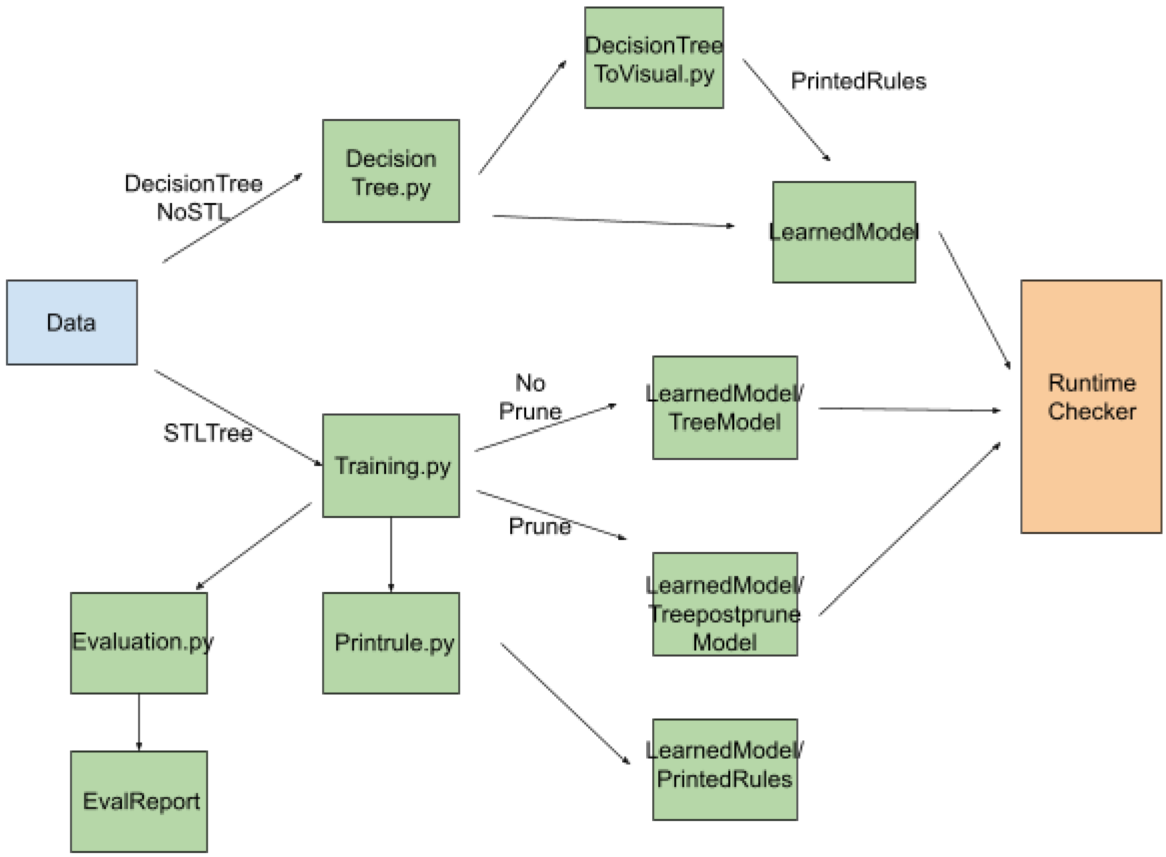
Running **python3 event.py** will generate the data, and the amount of timestamps generated is done through changing the number of iterations in **genEvent()** function.

The Samsung folder generates data on Samsung Smartthings development platform directly through simulations. We hardcoded some interactions with devices and Smartapps, and generated data through automation with Selenium. For new test environments, we would need to make a new **ScriptedEnvr.py** for Selenium to run.

GetDeviceInfo.py calls the monitor we created in 15400 (**Smartapp/monitor.groovy**) to access state changes for each device in the environment, with which we use to generate the test data through generateData.py. In actual use of our runtime checker, we would access for Samsung device state changes with the same workflow.

Note: The APIkey and endpoint in **GetDeviceInfo.py** should match the token and endpoint in **monitor.groovy** when installing the monitor to the environment on Samsung hub.

Learning Rules with Decision Trees:



After data is generated, we learn rules on the environment with our decision tree models:

STLTree: Learns rules based on temporal logic, where we assign a number for each state a device can have, and have the following rules:

- **F[a, b](x < c):** In interval [a, b], device x becomes a state less than c
- **G[a, b](x < c):** In interval [a, b], device x is always a state less than c
- **F[a, b]G[0, d](x < c):** In interval [a, b], device x is in a state less than c for d seconds

Running **python3 training.py** will learn two trees of STL rules for each (Device, state) tuple in the environment, pruned and unpruned version. The learned tree can be found in a file in the format “{deviceName}_{stateName}.pkl”. There is also a readable version under directory **LearnedModel/Treeprint** and **LearnedModel/Treepostpruneprint** with name format “{deviceName}_{stateName}.txt”.

To have a readable format of the learned rules, run **python3 printrule.py** after training model.

To evaluate our model’s accuracy, run **python3 evaluation.py** on the evaluation set.

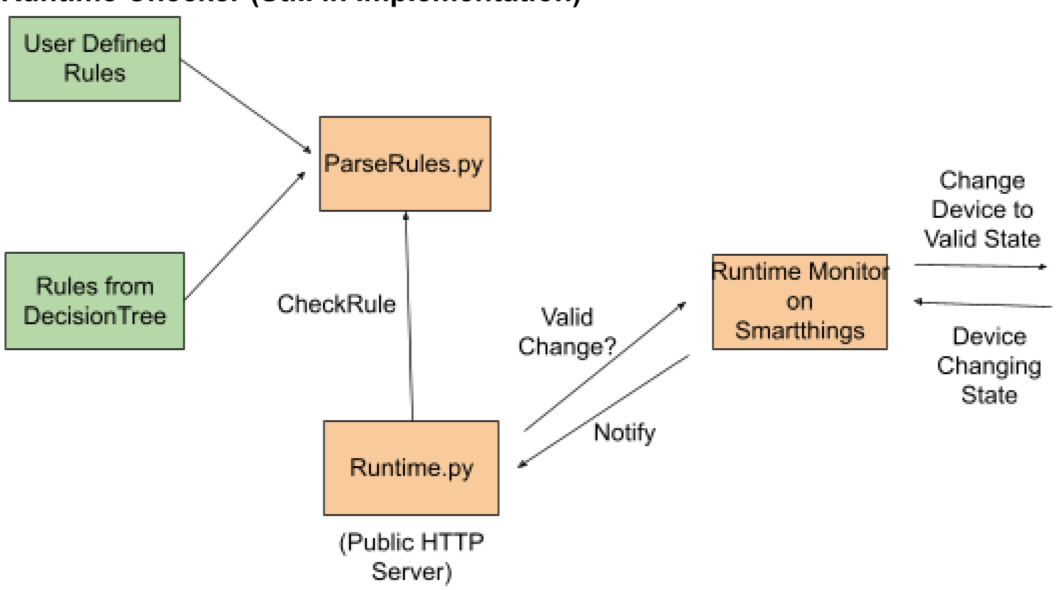
There are multiple parameters can be modified in the STLTree, listed below:

- **TrainingData:** The data used for training and validation for pruning in the STLTree model can be modified in **training.py.** The data used for evaluation is modified in **evaluation.py**
- **Datahandling:** Our data is currently separated into 10 second intervals for our tree to learn with an offset of 2. That is, we start at timestamp 1 to 10 as first data, then 3 to 12 as second, etc. The interval amount and offset for data handling can be changed in **training.py** under calls to function **trainingset** and **evaluationset.**
- **Training Iterations:** Our model utilizes the simulated annealing method, which the number of iterations ran is determined by parameters **Temperature** and **Steps**. This is done by changing the parameter for **self.Tmax** and **self.Steps** for both **FLPrimitiveProblem** and **SLPrimitiveProblem** under **Model/PrimitiveOptProb.py**
- **Error Threshold:** Since there is randomized behavior in our data generation, there are some rules learned by our model that are fairly inaccurate. We would only print out the rules that we are confident with a high accuracy. The accuracy threshold is set in **printRule.py.** Similarly, we would only count mistakes due to rules with a high accuracy in evaluation, the threshold for such is set in **evaluation.py**

DecisionTree NoSTL: Learns rules in the environment that are not time related. I.e. when some device changes to a state, the other device must immediately change to the state also.
The model utilizes the sklearn.tree package by python. The output for learned rule is directly shown under **LearnedModel** folder with format “{deviceName}_{stateName}.pkl” and can be visualized under **LearnedModel** folder with format “{deviceName}_{stateName}.txt”

To learn the rules, run **python3 ModelsNoSTL/DecisionTree.py**
To visualize the rules, run **python3 ModelsNoSTL/DecisionTreeVisualization.py**

Runtime Checker (Still in Implementation)



Our current runtime checker first parses the rules learned from the decision tree and possible user defined rules. Due to the limitation of DONT rules only, we are only supporting user input rules in the format of

THE {Device}_{State} STAYS {State Value} AFTER {number} {SECONDS/MINUTES/HOURS} WHEN {State} OF {Device} IS {State Value} FOR {number} {SECONDS/MINUTES/HOURS}
-- Translates to a G rule for PTSL

THE {Device}_{State} STAYS {State Value} AFTER {number} {SECONDS/MINUTES/HOURS} WHEN {State} OF {Device} BECOME {State Value} IN LAST {number} {SECONDS/MINUTES/HOURS}
-- Translates to a F rule for PTSL

THE {Device}_{State} STAYS {State Value} AFTER {number} {SECONDS/MINUTES/HOURS} WHEN {State} OF {Device} BECOME {State Value} IN LAST {number} {SECONDS/MINUTES/HOURS} FOR {number} {SECONDS/MINUTES/HOURS}
-- Translates to a FG rule for PTSL

Example:
THE Door_lock STAYS unlocked AFTER 3 SECONDS WHEN alarm OF Smoke Alarm IS siren FOR 5 SECONDS

Workflow of Monitor:
We would first start a public http server using **python3 runtime.py** to communicate with a Samsung Smartapp that is created to subscribe to all the device changes in the environment, **Smartapp/runtimeChecker.groovy**. Whenever a device's state gets changed, the RuntimeChecker sends the event to the http server.

The http server then keeps an internal state of devices in the environment, and when a device change is received, it then checks for if the change violates any learned rules. If it does, it will send back to the RuntimeChecker to the actual state of the device it should be in, and the RuntimeChecker automatically changes the device back to its state.

Parameters that can be changed in the workflow:

- HTTP server port. In my implementation, I have used my server at home, which I would need to run Runtime.py when SSH'ed into the server. The general code should work for any server hostname with a working port. This is changed in top of **runtime.py** also in **RuntimeChecker.groovy** for it to connect to the server.
- Error Threshold when converting rules. Similar to the tree model before, only accurate rules would be converted. The error threshold is changed in function call to **convertRules** in **runtime.py**
- "Cap" when converting rules. To make our learned PTSL rules in the model easy to check, we converted the interval **[a, b]** into **[a', b']**, where **a', b'** represents the interval

from a' seconds ago to b' seconds ago from now. The conversion needs the time interval we use to train our model, which is "Cap". The cap parameter should always be the same as the interval for training model, and can be changed in function call to **convertRules** in **runtime.py**

- Only checking import devices. We can also only check for important device changes in the environment, and ignore other device changes. This is done by whenever an important device change gets sent to the monitor, the monitor requests each device state in the environment and analyzes the change based on stored data. However, the Samsung Hub has a ~10 second delay on storing device changes, so such runtime checking may not be accurate. To activate the feature, run **python runtime.py --important=True** instead.

The runtime checker is currently operating under the limitation that we are only able to check whether the device is changed to a state that it should not have, and changing it back to the desired state.

However, It can not check whether the device has changed to a state according to the rule, which we are currently implementing the feature by looking at Samsung Smartthing's automation package with javascript.

Note: The APIkey and endpoint in **runtime.py** should match the token and endpoint in **monitor.groovy** when installing the monitor to the environment on Samsung hub.