Component Design

For Multiagent Control of Traffic Signals

Version 1.0

Submitted in partial fulfillment of the requirements of the degree of MSE

Bryan Nehl
CIS 895 – MSE Project
Kansas State University

Table of Contents

1 Intro

In this document I will review the components that make up the MACTS system. Recall the System Context diagram from the System Architecture Document. The classes covered here would fit inside of the yellow boxes in the System Context diagram.

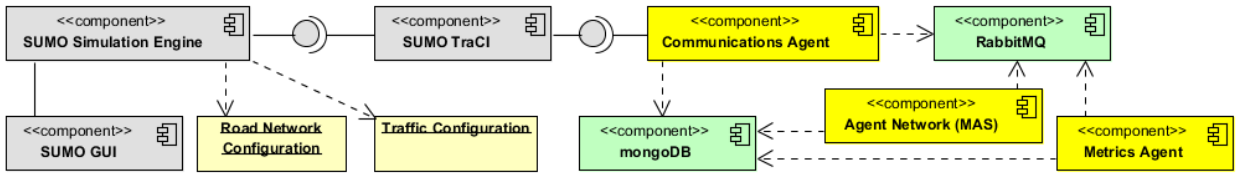

Figure 1System Context Diagram

2 Components

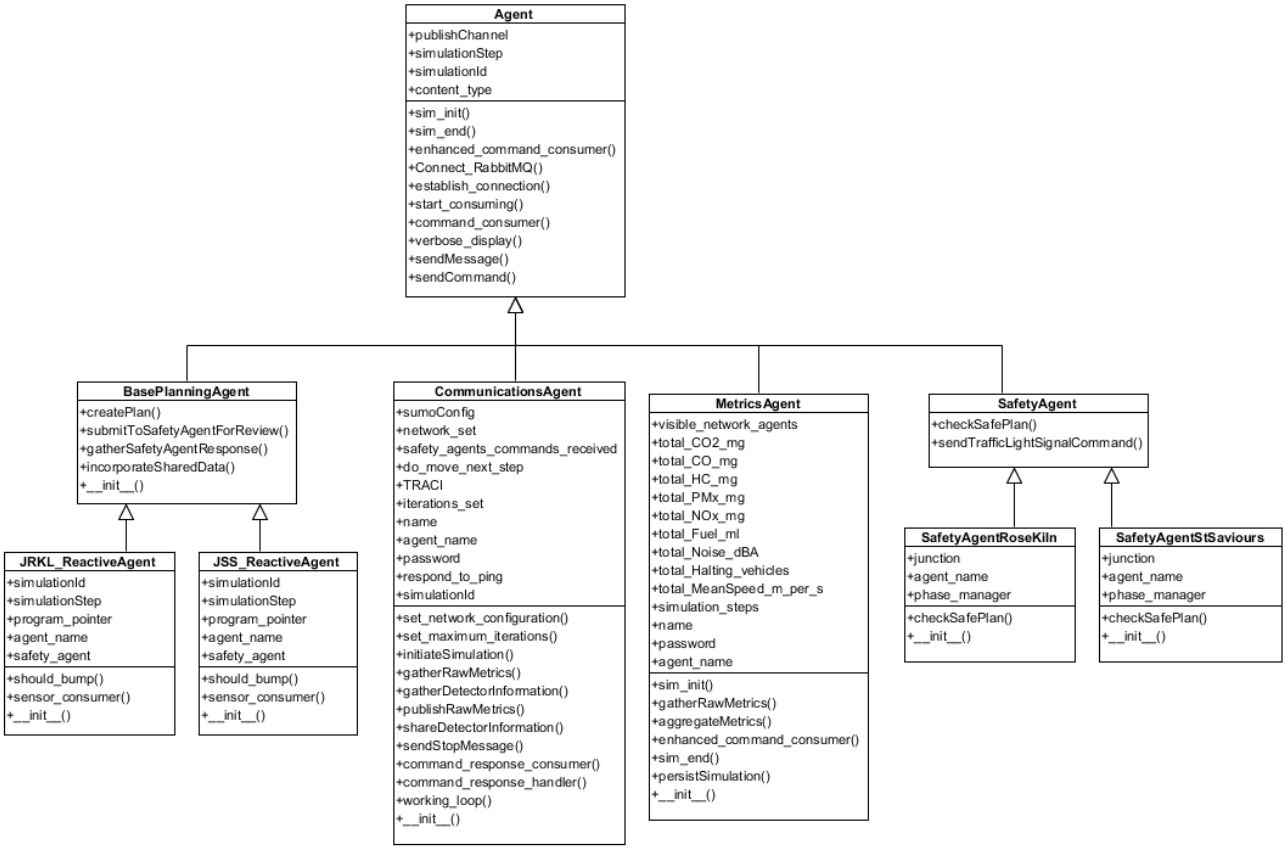

Figure 2Agent Classes

2.1 Agent

The agent class serves as a base class for the other agents in the system it is located in the Python Core.py file. It contains methods for connecting to RabbitMQ, sending messages and receiving and handling some common simulation commands. It has stubs for enhanced_command_consumer, sim_init and sim_end which may be used by the implementing agent to enhance and extend operation of the agent when it receives commands during operation.

In addition the extending code will be able to hook in code that should be run at simulation initialization and end.

The helper method verbose_display is found here as well as sendMessage and sendCommand. The sendMessage method is used to send a message to any RabbitMQ exchange after you have established a connection with Connect_RabbitMQ. The sendCommand method sends a "command" formatted message to through the sendMessage method to the COMMAND_DISCOVERY exchange.

2.2 BasePlanningAgent

This class serves more as a specification or abstract class for what methods should be implemented in a planning agent.

2.3 JRKL_Reactive Agent and JSS_ReactiveAgent

These classes are at the heart of the TLS control. They are responsible for deciding the next TLS phase based on current state and sensor input.

The algorithm for controlling the traffic light signals is based on the default generated by the SUMO system. I altered it by shortening the long phases and providing for "bumping" the phase longer if there were vehicles detected on the sensors that were appropriate for the active light phase. However, there are a maximum number of times to bump per phase that is also specified. You wouldn't want to keep a light green all the time blocking other traffic. This keeps the phase shorter if the network isn't active and lengthens the phase for an active network. I also implemented a SLIDING_WINDOW variable that is used to specify don't bump if the program pointer isn't within *n* seconds/steps of the end of that phase. That is, don't spend our bump if the car will already clear the intersection.

2.4 Safety Agent

The safety agent is responsible for checking the submitted phase and verifying that it is a safe transition to make. If the transition is not acceptable it will return a status array which indicates the offending signal. The sendTrafficLightSignalCommand method accepts a phase plan, decorates it appropriately for turning into a JSON document that will be sent to the COMMAND_RESPONSE exchange.

2.5 Safety Agent Rose Kiln and Safety Agent St Saviours

These agents are concrete implementations of the SafetyAgent class. There is enough similarity here that the SafetyAgent class could probably implement the same behavior by taking a few parameters in the constructor.

2.6 Metrics Agent

The metrics agent is responsible for consuming all simulation run metrics and saving the total values and network configuration to MongoDb. It is located in the Python Core.py file. The variables for the metrics being captured include the unit of measure. The metrics agent simply sums. It is up to you to do the division by the simulation steps. Also, for some of the variables like Noise_dBA and MeanSpeed_m_per_s you will need to do the division by the number of steps multiplied by the number of segments being captured.

When a simulation init is triggered, the metrics are reset. The agent exits when it receives a simulation end command. The enhanced_command_consumer is an extension on the base command_consumer. It is used to learn the identities of the agents on the network. This agent utilizes both the Metrics exchange and the COMMAND_DISCOVERY exchange.

2.7 CommunicationsAgent

The set_network_configuration method is used to determine which network configuration to load. The full, medium or low traffic load configuration.

The set_maximum_iterations method is used to obtain the intended number of simulation steps to be run.

If the prior two methods are successful, the application then calls initiateSimulation. This method launches the SUMO GUI executable.

After establish a connection with TraCI and doing the initial Connect_RabbitMQ the Communications Agent broadcasts a new simulation Id to the COMMAND_DISCOVERY exchange. Next, a new thread is created which is the working_loop of the program.

The command_response_handler is run which is an event based consumer that is triggered by RabbitMQ messages on the COMMAND_RESPONSE exchange.

In the working loop the Communications Agent gathers run metrics from SUMO via the TraCI TCP/IP connection, packages them and sends them on to the Metrics exchange. Since the working loop is on a different thread and high volume it creates and uses its own connection to RabbitMQ.

If the variable FIXED_PLAN_NO_SAFETY_AGENTS is set to True, the working loop will not wait for responses from safety agents. This is useful when you want to gather the metrics associated with the default plan that is specified in the configuration files.

Because the methods within the working loop (gatherDetectorInformation and gatherRawMetrics) need to work with the TraCI connection and the command_response_handler also wants to send TLS instructions through TraCI, it was necessary to use Python events from the threading library to control access to the single TraCI connection.

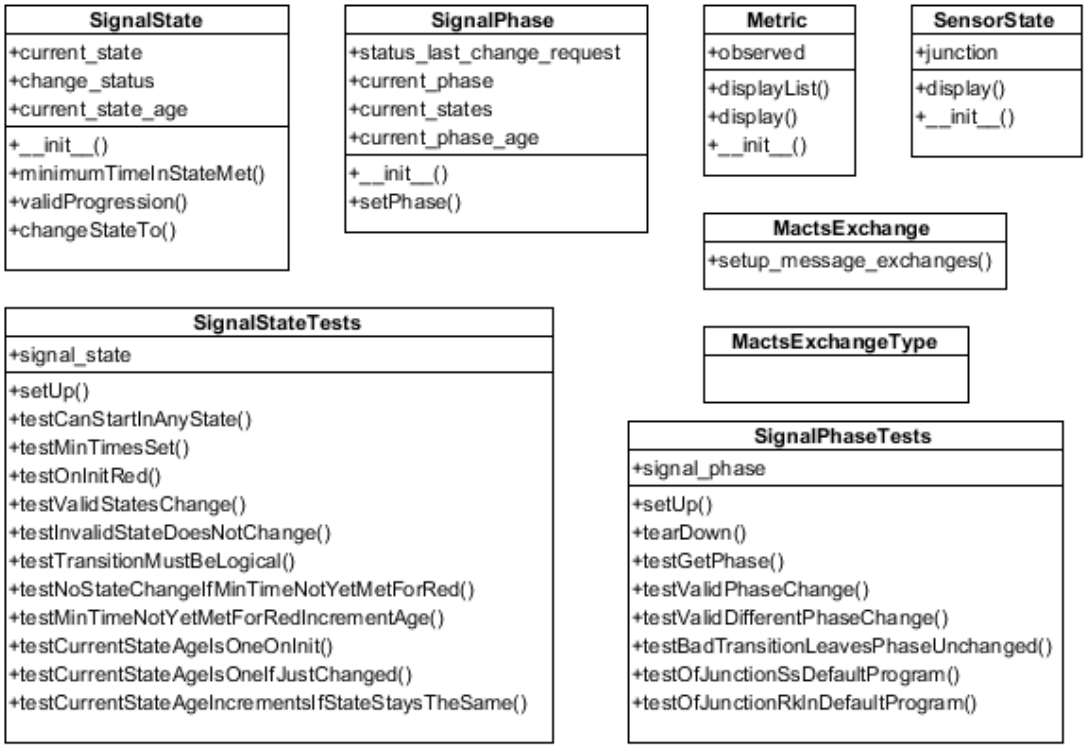

Figure 3Supporting Classes

2.8 SignalState

The SignalState class is located in the TrafficLightSignal.py file. The constructor for this code requires that minimum times be specified for all light states. The primary method of interest here is the changeStateTo method which accepts a parameter of desired state. This method checks with the helper methods of minimumTimeInStateMet and validProgression before changing the state. The change_status property can be check to see the status of the method call. If the request was not valid, the timer increments and the current_state does not change.

2.9 SignalStateTests

The SignalStateTests class is located in the TrafficLightSignalsTests.py file. The signal state tests are based on the formal specification post conditions of the System Architecture Design Document. Tests include: only valid light progressions permitted, minimum times in signal met before changing and an invalid state change is not permitted. So, for instance you cannot set the state to “blue”. You also cannot change from green directly to red.

2.10 SignalPhase

The SignalPhase class is located in the TrafficLightSignal.py file. This class builds on the SignalState class. For each light that is a part of the signal phase a backing SignalState object is created by the init method. The setPhase method works by breaking down the desired_phase request into its individual light components. The appropriate light change request is then made of each individual SignalState object. The result and status of each change request is captured. Finally a check is done to see if the new_phase is equal to the desired_phase and the appropriate status response is returned.

There is room for extending this class to incorporate not allowing “foes” to be active at the same time. Foes are when having two lights green would result in a crossed paths condition.

2.11 SignalPhaseTests

The SignalPhaseTests class is located in the TrafficLightSignalsTests.py file. This test class verifies that valid phase change requests are honored and bad requests do not change the current phase. In addition it verifies that the default generated TLS programs are valid according to our code.

2.12 Metric

The metric class is used to store metrics information in a dictionary. The display method does a pretty display to the screen for an individual item. The class method displayList accepts a list of metrics iterates through them and displays them.

2.13 SensorState

SensorState is used for aggregating the sensor states for each simulation step. It is located in the Python Core.py file. It contains definitions for the sensors related to each junction. It has a dictionary for holding the key of the sensor id and its corresponding metric value. On init, the simulation id, the simulation step and the junction id are automatically added to the dictionary.

2.14 MactsExchange

Is located in the Python Core.py file and has the class method: setup_message_exchanges which is critically important to be run whenever a machine has been restarted. The exchanges are not set up to be durable. The method is called by the rabbitmq_create_exchanges.py script.

2.15 MactsExchangeType

This class is a simple container for constant values used when declaring a RabbitMQ exchange. It is located in the Python Core.py file. These constants should probably be refactored into part of MactsExchange.

3 References

Nehl, B. (2012). System Architecture Design version 2.0.