


CAV Simulation Observer Framework

 This work is part of a ME 599 project, if you have any questions or are confused about something please feel free to reach out to Ansar Khan at anytime. I am always open to answering any questions or clearing things up.

az3khan@uwaterloo.ca

khanzansar@gmail.com

The observer framework is a system to monitor a simulation and assess its performance. Observers subscribe to a topic and watch each message published, at the end of a simulation run the observer is expected to return a PASS/FAIL result.

Using Existing Observers

There are many observers already built into the [structured testing repo](#) that can be used out of the box in `.sim` files.

Schema:

Observers are added to a `.sim` file by putting them in a list under the `observers` section, specific examples are shown below. Each type of observer has 5 common fields.

`name`: Name of the observer, can be anything used to associate results file with observer

`observerClass`: Class defining the functionality of the observer, has to be present in the [observers](#) directory of structured testing

`observerType`: Type of observer must be one of {ALWAYS_TRUE, TRUE_ONCE, TRUE_AT_END, TRUE_AT_START} NOTE: Not required for all observers i.e Frequency and Heartbeat do not need this field

`topic`: Name of topic to subscribe to (using the global namespace)

`msgType`: Data type of msg, in the format it would be imported in python and not C++ i.e use `std_msgs.msg.Float32` and NOT `std_msgs/Float32`

`field`: Subfield of the msg that is being consumed indexed through a `.` (see examples below to clear up understanding)

Frequency Observer:

Used to ensure topic is published above an average frequency over the simulation.

This is useful to add to critical control signals are topics that are known to be slow to compute ensuring that upstream nodes are getting data at the expected rate.

Example Schema:

```
#Ensures topic "/cmd_vel" is published at 10Hz (or more)
- name: CmdVelFrequency
  observerClass: FrequencyObserver
  topic: "/cmd_vel"
  msgType: "geometry_msgs.msg.Twist"
  minFreq: 10
```

Heartbeat Observer:

Used to ensure a topic is published at least once during a simulation. Generally used when you want to make sure the dependencies for a required topic have been met and it is able to publish something without caring about its contents.

Example Schema:

```
# Make sure that radar publishes tracked objects at least one
- name: RadarHeartbeat
  observerClass: HeartbeatObserver
  topic: "/radar_tracked_objects"
  msgType: "nav_msgs.msg.Odometry"
```

In Range Observer:

Ensures that a numeric value is within an acceptable range. Paradigms of {ALWAYS_TRUE, TRUE_ONCE, TRUE_AT_END, TRUE_AT_START} apply and are used to determine the result.

Example Schema:

```
# Make sure cmd vel in x dir is always within [0, 10]
- name: CmdVelX
  observerClass: InRangeObserver
  topic: "/cmd_vel"
  msgType: "geometry_msgs.msg.Twist"
  field: "linear.x"
  observerType: ALWAYS_TRUE
  minVal: 0
  maxVal: 10
```

Min Max Observer:

Similar to the in range observer but only checks value in one direction. (Value is always greater than min OR value is always less than max).

Example Schema:

```
# Ensure radar sequence number is always less than 120
- name: RaderObjectsIDObserver
  observerClass: MinMaxObserver
  observerType: ALWAYS_TRUE
  topic: "/radar_tracked_objects"
  msgType: "nav_msgs.msg.Odometry"
  field: "header.seq"
  value: 120 #in m
  isMin: False #Max Value (if true becomes min val observer)
```

Monotonic Observer:

Used to ensure that a value is always increasing or decreasing throughout the simulation. Useful to determine if fragmented data is coming in order.

Example Schema:

```
#Ensure sequence number of radar msg is always increasing to confirm
data is received in order
- name: RaderObjectsIDObserver
  observerClass: MinMaxObserver
  observerType: ALWAYS_TRUE
  topic: "/radar_tracked_objects"
  msgType: "nav_msgs.msg.Odometry"
  field: "header.seq"
  isIncreasing: True #Check monotonic increase if false checks for
decrease
```

Adding New Observers:

The observer framework is designed to support rapid development of new custom observers targeted to specific use cases. In order to make a new observer define a new class in the [observers directory](#) of structured testing. The class should inherit from the [BaseObserver](#).

 Once an observer is added and pushed to the repo ensure to update this page with its schema above.

Defining an Initializer:

An initializer should take following form. Ensuring that the super initializer is called.

```
def __init__(self, *, name, topic, msgType, observerType,
customArg1, customArg2, ..., **kwargs):
    super().__init__(name, [topic], [msgType], observerType) #Note
observerType can be passed in or set to a fixed value (i.e Frequency
Observer)
```

Custom arguments specific to the new observer (`customArg1, customArg2` above) can be specified in the sim YAML keyed exactly the same as the variable name in function signature of the initializer.

Define consumeMsg:

This function is called every time a msg is published, the data can be parsed and understood within the function.

For observers that are memoryless (Truth value is able to be determined with msg at current time step) the `logResult` function can be used. This is a function that logs the truth value of the observer overtime and at the end when the default `getResult` is called returns an overall truth value based on the specified observer type (`{ALWAYS_TRUE, TRUE_ONCE, TRUE_AT_END, TRUE_AT_START}`).

For observers that are NOT memoryless a custom implementation of `getResult` will need to be written.

Define getResult:

This function is called once at the end of the simulation, this function is expected to return a Boolean that corresponds to Pass/Fail of the test. As defined above if an observer is memoryless and `logResult` is used this does not have to be implemented.

Define metadataDict:

This function is called once at the end of the simulation and is expected to return a dictionary, the contents of this dictionary are dumped to the results file. It is useful to use this function to specify more information about the final result other than just PASS/FAIL