Assessment Evaluation

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 Introduction

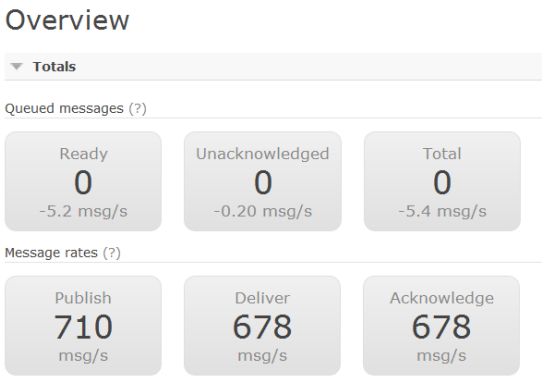
This document presents the results of testing the MACTS system components. 161 minutes were used in the testing activity. This does not include testing time that was specifically part of the development process.

2 References

"Test Plan" available at http://people.cis.ksu.edu/~bnehl/.

3 RabbitMQ

I was able to see that the agents were active by using the RabbitMQ monitoring tool.
Included are some screen shots that show message activity, the channels where the activity was occurring and their related exchanges.



Channels

	Details							Message rates			
Channel	**Virtual host**	**User name**	Mode (?)	**Prefetch**	**Unacked**	**Unconfirmed**	**Status**	**publish**	**confirm**	**deliver / get**	**ack**
127.0.0.1:59306 -> 127.0.0.1:5672 (1)	macts	metrics		1	1	0	Active	0.00/s		657/s	657/s
127.0.0.1:59511 -> 127.0.0.1:5672 (1)	macts	liaison		1	0	0	Idle	0.00/s			
127.0.0.1:59512 -> 127.0.0.1:5672 (1)	macts	liaison		0	0	0	Active	674/s			

macts	**command_response**	direct		0.00/s
macts	**metrics**	fanout		672/s
macts	**sensor-RoseKilnLane**	fanout		16/s
macts	**sensor-SteSaviors**	fanout		16/s

With the administration tool I was also able to create and bind my own queues to the exchanges. In this way I could manually review the traffic flowing through the exchange. In addition I could look at individual queues and read or push values into them. This was useful when I wanted to verify agent operation on receipt of commands.

4 MongoDB

I was able to verify that simulation run metrics were persisted to the database by finding the results with the simulation Id that was presented at the end of a simulation run. This was done by using the mongo shell application to find() the document in the macts.metrics collection

5 VERBOSE_LEVEL

In the "interesting" classes I incorporated the use of a helper method for displaying additional debugging and monitoring information. The helper method would accept a string format pattern along with the variables to display and a verbose level at which to display the message. This consolidated an if-then expression into a single more concise line. I would then set the verbose_level constant in the class to the level of detail I wanted to see displayed. The general rule that I followed was the more detailed/lower level the information being displayed would have a higher level.

6 Inspection Result Details

SR1, SR2, SR3 were verified simply by running the system and seeing the desired road network displayed in the SUMO environment.

7 Black box testing

Per section 6 of the Test Plan, black box testing was done by watching the simulation and checking run metrics.

8 White box testing for System Requirement Validation

These test cases refer to Section 10 of the Test Plan.

Scenario	**System Requirement Tested**	**Passing Criteria, problems and resolutions**
Do basic simulation run	SR4, SR8, SR9	The communication agent puts simulation state and metric data into RabbitMQ queues. I was able to monitor that the correct data was being put into the correct queues.
Do basic simulation run	SR5, SR6, SR7, SR10, SR11	Commands put into RabbitMQ Command Queue are retrieved and submitted to TraCI. The do next simulation step instruction is sent. This initially caused a threading issue when a thread on one loop is trying to read sensor data and metric information another thread was trying to send the TLS command. Resolved by implementing Python threading "events"
Multi-agent run	SR7, SR10, SR11	Communication agent waits for all MAS Nodes to respond before sending do next simulation step. Again, had to make sure that one thread wasn't trying to

		increment to the next simulation step while another was still pulling metrics. Also, implemented additional separate RabbitMQ connection.
Multi-agent run	SR12	Review message queues and output to see that agent network discovery is occurring. Reviewed on screen output and queue contents to verify behavior.
Do basic simulation run with Reactive Agent	SR15, SR16	Review output from Communications Agent, Planning Agent and Safety Agent. Monitored full integration test with verbose_level set to 3.
Unit test simulating calls with unsafe conditions	SR18, SR19	Doesn't permit unsafe TLS configurations. Reason returned to requester via RabbitMQ. Full unit tests on SignalState class to verify behavior and to verify that the correct failure reasons are returned.
Do basic simulation run	SR18, SR20	Safe commands are submitted to command queue. In the process of testing a known good traffic light program, the agent refused to change phases. I determined that this was because the safety agent was configured to not permit the transition from yellow to green. The system code was altered to permit this transition.
Do basic simulation run	SR21, SR22, SR23	Review output log for simulation step and aggregate metrics. Check MongoDB for aggregated metrics. After a simulation run used the displayed Simulation ID to locate the stored metrics in the DB. Noticed not all data fields were filled in. Tracked it down to the use of hard coded constant that was different than the expected key. Changed to use a global constant declaration.

9 Simulation Results

Per section 6 of the Test Plan: I established baseline metrics for comparison by running 5 simulation runs of the default timing based configuration for a simulated hour of time. I used the average of the results as a basis for comparison.

9.1 Post Simulation Run Analysis

At the end of every simulation run I would retrieve the aggregated metrics from the MongoDB. Then I would divide the totals by 3600 seconds to get the average per simulation second values. For the Noise and MeanSpeed I also had to divide out the number of road network segments (43). The mean speed also required the multiplication by 3.6 to get the average km/h value.

```
"_id" : ObjectId("4f9647b6e6286c2f7c000000"),
   "SimulationSteps" : 3600,
```

```
   "CO" : 4609808.32152559 /3600=1280.50231153 ,
   "PMx" : 31626.892771381 /3600=8.785247992 ,
   "CO2" : 473461046.8794881/3600=131516.9574665 ,
   "NOx" : 788364.0258641898 /3600=218.9900071845 ,
   "Fuel" : 188761.543257277 /3600=52.433762016 ,
   "HC" : 204108.23595548145 /3600=56.69673220986 ,
   "NetworkConfiguration" : [
          "JSS_ReactiveAgent",
          "MetricsAgent",
          "JRKL_ReactiveAgent"
   ],
   "Noise" : 3914785.769310845/3600 = 1087.440491475
1087.440491475 /43 = 25.289313755    ,
   "Halting" : 287129 /3600=79.7581 ,
   "SimulationId" : "20120424|012250",
   "MeanSpeed" : 882959.4183915803 / (3600*43) = 5.7038722118 *
3.6 = 20.5339399625
```

9.2 Simulation Run Results

The scenarios are for low (10%), medium (50%) and full traffic loads (100%). Fixed indicates that the simulation was running the default fixed traffic light signal plan—the base line behavior for comparison. SS Solo and RKL Solo indicate that the agents for St Saviors Junction and Rose Kiln Lane junctions were run independently. That is, the opposite junction was running the fixed program. "Both" indicates that both the SS and RKL agents were actively managing their junctions. I initially had a slightly different algorithm for the SS and RKL agents. With the disappointing performance of the RKL agent, I did another set of simulation runs with it reconfigured to the same algorithm as the SS agent.

These numbers are the average (per second) for a 1 hour simulation

Run Type	CO mg	CO2 mg	PMx mg	NOx mg	Fuel ml	HC	Noise dbA	Halting cars	Mean Speed km/h
Fixed - FULL	1,087.70	115,593.40	8.08	195.81	46.09	47.26	25.26	57.38	20.80
Reactive - SS Solo	1,158.38	122,184.23	8.31	204.96	48.71	51.12	25.88	64.25	20.34
Reactive - RKL Solo	1,250.77	128,457.21	8.75	215.18	51.21	54.79	25.13	72.85	20.74
Reactive - Both	1,280.50	131,516.96	8.79	218.99	52.43	56.70	25.29	79.76	20.53
Reactive - Same A	1,255.91	129,790.39	8.80	217.13	51.75	55.28	25.49	72.66	20.59
Fixed - Medium	536.83	64,761.16	4.82	112.70	25.82	23.16	20.66	19.04	22.30
Reactive - SS Solo	442.42	58,014.50	4.48	102.60	23.13	18.95	20.26	8.37	22.85
Reactive - RKL Solo	549.38	66,734.74	4.85	115.15	26.61	24.23	21.05	20.59	22.37
Reactive - Both	438.94	57,937.51	4.46	102.37	23.10	18.89	20.11	8.19	22.93
Reactive - Same A	443.81	58,276.51	4.47	102.78	23.23	19.16	20.01	9.13	22.87
Fixed - Low	82.22	11,356.62	0.87	20.00	4.53	3.64	8.82	1.44	24.73
Reactive - SS Solo	81.91	11,294.60	0.87	20.00	4.50	3.57	8.56	1.17	24.87
Reactive - RKL Solo	78.51	11,196.12	0.85	19.72	4.46	3.51	8.36	1.00	24.96
Reactive - Both	78.36	11,111.72	0.85	19.63	4.43	3.47	8.31	0.97	24.98
Reactive - Same A	77.91	11,082.03	0.85	19.54	4.42	3.47	8.25	1.01	25.00

I was disappointed to not see an improvement over the standard system generated program for the full or heavy load scenario. I was very happy to see that there was improvement for medium and low network load scenarios.

10 Summary

The combination of doing module level testing that was feature or system requirement oriented along the way with systems integration testing was a good practice. It kept the defect rate down and productivity up.