System Architecture Design

For Multiagent Control of Traffic Signals

Version 2.0

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 3
2 References 3
3 Architecture 3
3.1 System Analysis 3
3.2 System Context Diagram 4
3.3 Component Design 5
3.4 Component Interface Specification 6
3.4.1 IPersistPlan 6
3.4.2 IPersistSafetyRequest 7
3.4.3 IPersistMetrics 7
3.5 System Design Rationale 8
3.6 High-Level Design 9
3.7 Mid-Level Design 10
4 Component Interaction 11
4.1 Initialization interaction 12
4.2 In simulation processing interaction 12
4.3 Simulation end interaction 13
4.4 Discovery Protocol 13
5 USE/OCL Model 14
5.1 Overview 14
5.2 USE OCL Code 16
5.3 Initialization Script 20

1 Introduction

This document provides system design information for the MultiAgent Control of Traffic Signals (MACTS) system. This system is used to simulate agent based control of traffic light signals. This document covers the system components and component interfaces. However, it does not cover all of the interfaces methods in detail. A system analysis diagram as well as a high-level overview of the whole system is included in this document. Mid-Level design is also included for all of the components. A sequence diagram is included which shows how the system components interact during run time.

2 References

1. "Vision Document" available at http://people.cis.ksu.edu/~bnehl/.

3 Architecture

This section documents the system component design, the interfaces of those components and provides high-level design with rationale for design within the system context.

3.1 System Analysis

Clearly describe the high level relationship between model elements

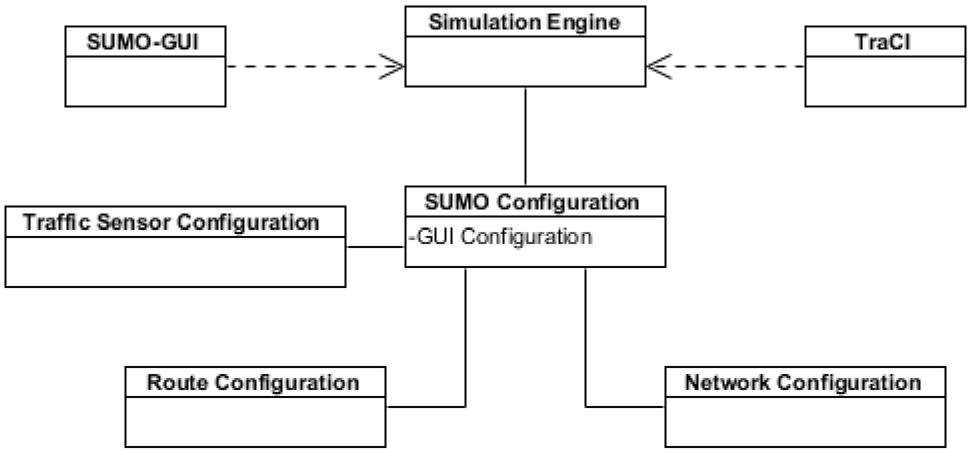


Figure 1 System Analysis Diagram

Referring to Figure 1, the Simulation Engine takes care of the work of simulating the movement of the vehicles in the system. The SUMO-GUI is the front-end graphical user interface that displays the state of the simulation. TraCI is the TCP/IP interface to the simulation engine. TraCI is how external entities can interact with the simulation. The Simulation Engine relies on a SUMO Configuration file. The SUMO Configuration file includes specific settings information for the GUI as well as references or pointers to three other configuration files. The Traffic Sensor Configuration file contains information about sensors like the e1 inductor that are on the road network. The Route Configuration includes information about the routes that cars take. Details regarding the types of vehicles, vehicle distribution are specified. For the routes, the flow rates and probabilities are specified. The Network Configuration file is the result of running three files: Nodes, Edges and Connectors through the NETCONVERT utility. NETCONVERT is a SUMO utility. The Nodes, Edges and Connectors files detail where connections happen (nodes), streets are described by the edges and connectors handle the mapping from one edge to another at a junction node.

3.2 System Context Diagram

This system context diagram shows how the components of the MACTS system interact with each other and with the external systems.

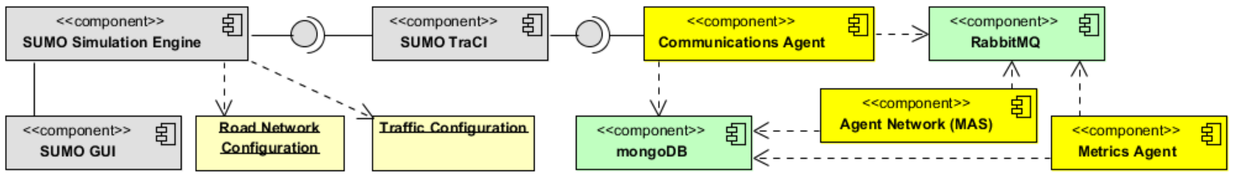

<<component>> SUMO Simulation Engine
<<component>> SUMO TraCI
<<component>> Communications Agent
<<component>> RabbitMQ
<<component>> SUMO GUI
Road Network Configuration
Traffic Configuration
<<component>> mongoDB
<<component>> Agent Network (MAS)
<<component>> Metrics Agent

Figure 2 System Context Diagram

In Figure 2, the System Context Diagram, grey components are SUMO components, the ivory components are SUMO configuration files created by me. The green components represent third party infrastructure servers. The yellow components (Communications Agent, Agent Network (MAS) and Metrics Agent) are the aspects that I will be creating. What we don't see at this level is the possibility for multiple Agent Networks. That would come in to play when we have a MAS working at every intersection.

In Figure 3, the basic processing for a single simulation step is shown. Data is received from the simulator and sent to RabbitMQ. From there, there the Metrics Processing pulls data from its queue and does its own parallel operations. In the Analyze Data step, the MAS node planning agent uses sensor information that it received from specific queues. The planning agent then creates a suggested plan and sends it to the safety officer for checking that the command is safe. At that point, if the plan is safe, operation continues on the happy path sending the commands to the communications agent. Otherwise, the planning agent is informed that the plan isn't safe.

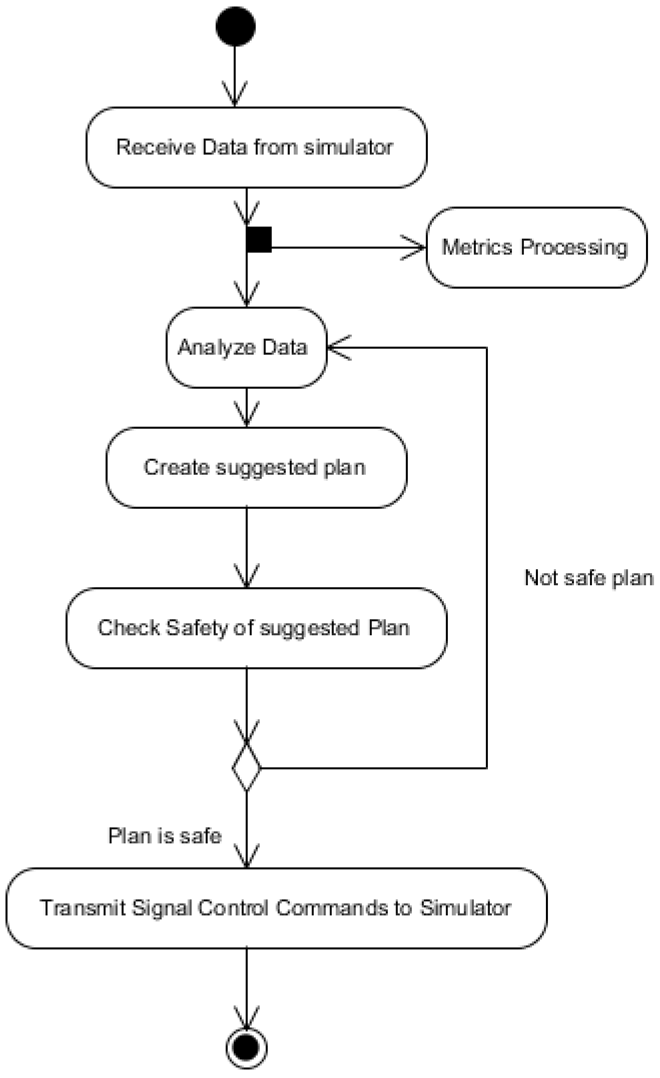

Receive Data from simulator
Metrics Processing
Analyze Data
Create suggested plan
Not safe plan
Check Safety of suggested Plan
Plan is safe
Transmit Signal Control Commands to Simulator

Figure 3Basic Processing for single simulation step

3.3 Component Design

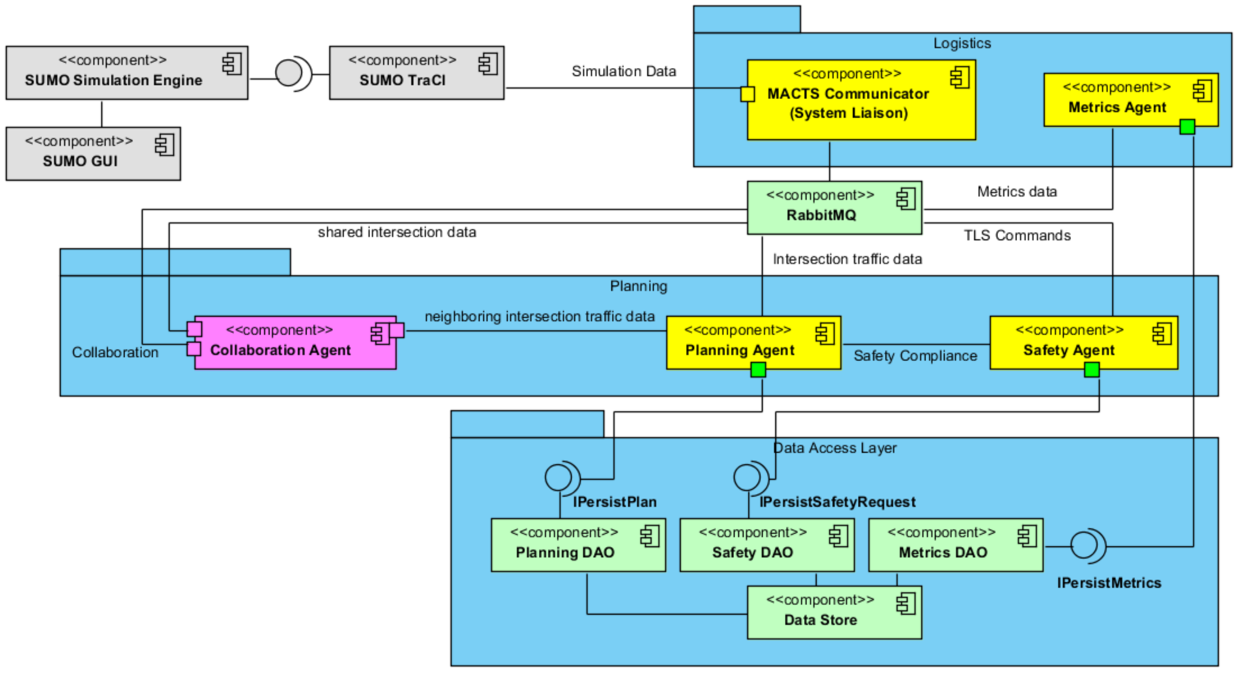


Figure 4 MACTS with single MAS Node

In the Figure 4 the MACTS with a single MAS is portrayed. The grey components, SUMO Simulation Engine, SUMO GUI and SUMO TraCI are all part of the Simulation for Urban MObility software. The light green components RabbitMQ and DataStore are third party servers for message queuing and data persistence. The yellow components Communicator or System Liaison and the Metrics agent are part of the MACTS system. No matter how many MAS Nodes are in the system, there will only be one of each of these.

The communicator component has the responsibilities of: initiating a session, retrieving and publishing sensor data, retrieving and publishing metrics data, retrieving and submitting simulation commands, telling the simulation to proceed with the next step and finally of notifying all participants that the simulation has ended.

The components that constitute a MAS Node are enclosed in the light blue container. They are the yellow Planning Agent and the yellow Safety Agent. The pink component, Collaboration Agent is an optional part of the MAS Node. Concrete instances of the planning agent will propose a signal light plan given sensor data. The concrete safety agent will make the determination if the proposed plan is safe for the intersection.

In figure five we see a multiple MAS node system. Some lines from the second MAS node to the RabbitMQ and Data Store servers have been eliminated for clarity. If draw in, they would originate and terminate the same as the connections from the first MAS node. In a multi-node system with collaboration, the pink collaboration agents share or publish information about what is happening in this intersection with neighboring collaboration agents.

Because I know the topology of the test network, I know that there will at most be two MAS nodes operating simultaneously. If a collaboration agent is enabled, it will be configured to share relevant information with its neighbor. It will also be configured to receive shared information from a predetermined queue.

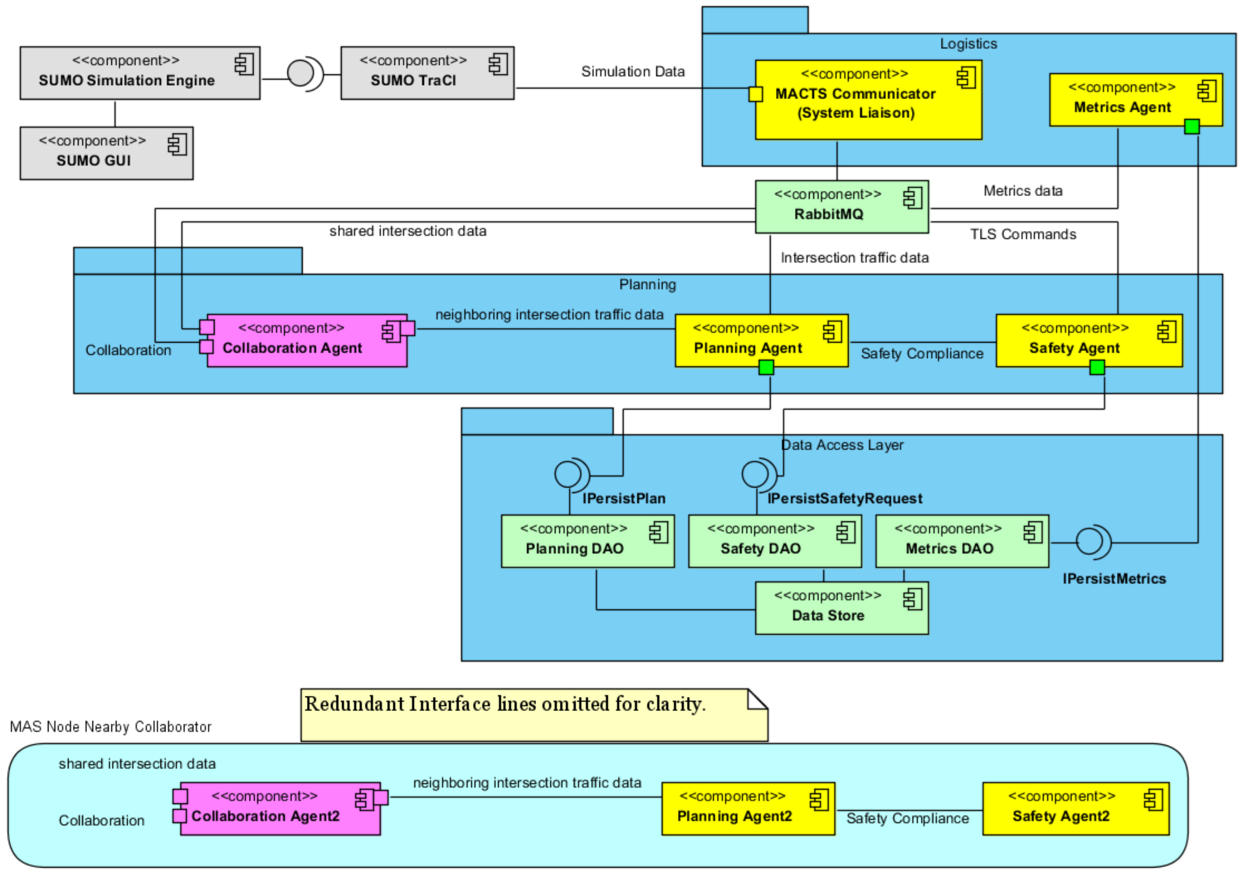

Logistics
<<component>> SUMO Simulation Engine
<<component>> SUMO TraCI
Simulation Data
<<component>> MACTS Communicator (System Liaison)
<<component>> Metrics Agent
<<component>> SUMO GUI
<<component>> RabbitMQ
Metrics data
shared intersection data
TLS Commands
Intersection traffic data
Planning
neighboring intersection traffic data
Collaboration
<<component>> Collaboration Agent
<<component>> Planning Agent
Safety Compliance
<<component>> Safety Agent
Data Access Layer
IPersistPlan
IPersistSafetyRequest
<<component>> Planning DAO
<<component>> Safety DAO
<<component>> Metrics DAO
IPersistMetrics
<<component>> Data Store
Redundant Interface lines omitted for clarity.
MAS Node Nearby Collaborator
shared intersection data
neighboring intersection traffic data
Collaboration
<<component>> Collaboration Agent2
<<component>> Planning Agent2
Safety Compliance
<<component>> Safety Agent2

Figure 5 MACTS with collaborating MAS Node

3.4 Component Interface Specification

While there are traditional class interfaces in the project, MACTS interfaces are more service oriented application programming interfaces, SOA APIs. RabbitMQ is the communications server used with JSON messages. In the forthcoming Component Design Document I will expand on the message exchanges used, their purpose and the type of messages being passed.

3.4.1 IPersistPlan

Signature	+persistPlan(simulationId : string, step : integer, agentId : string, plan : string)
Purpose	To store the plans that the agent creates during the simulation for future review or analysis.
Pre-Conditions	Established simulation and planning agent. Initialized data store.
Post-Conditions	The plan information is persisted in the data store.

3.4.2 IPersistSafetyRequest

Signature	+persistSafety(simulationId: string, step: integer, safetyAgentId: string, planAgentId: string, request:string, response:string)
Purpose	To store the requests that was made of the safety agent, by whom and the response given.
Pre-Conditions	Established simulation, planning and safety agents running. Initialized data store.
Post-Conditions	The safety information is persisted in the data store.

3.4.3 IPersistMetrics

Signature	+persistStep(simulationId : string, step : integer, simulationStepMetrics : Metrics)
Purpose	To store metric information for every simulation step. This will enable further review, analysis and potential reprocessing of the metrics.
Pre-Conditions	Established simulation and communications agent running. Initialized data store.
Post-Conditions	The metric data for every simulation step is persisted along with a simulation identifier and step identifier.

Signature	+persistSimulation(simulationId : string, aggregatedMetrics : Metrics, systemConfiguration : List<string>)
Purpose	The purpose of this method is to store aggregated simulation metrics along with the system configuration. The system configuration is the list of agents that constituted the realized system. Having the simulation identifier allows for associating the individual step metrics with the overall aggregated metrics.
Pre-Conditions	Established simulation and communications agent running. Initialized data store.
Post-Conditions	The aggregated metric data and system configuration are stored along with the simulation identifier.

3.5 System Design Rationale

There were several design criteria which led me to existing architecture. A key problem that had to be resolved was how to enable a distributed architecture to work with the direct connection, single client interface of TraCI. This design makes that possible by using a Communications Agent/System Liaison which interacts with the other parts of the system in a decoupled way through RabbitMQ message queues.

I chose to work with RabbitMQ and MongoDB because of their easy interfaces and ability to work with JSON documents. Also, since the core of this project isn't messaging or data persistence this seemed to be a good way to avoid reinventing the wheel.

I knew that I wanted to be able to test various types of agents as well. So, I wanted to be able to easily configure a network. This led me to a decoupled design where the agents communicate with each other through message queues. Because there is some shared behavior amongst agents, I thought it would be useful to have an abstract Agent class which could be used by the other types of agents. Planning Agent, Collaboration Agent and Safety Agent are all more specialized, yet still abstract classes that build upon the Agent class.

Why didn't I have all of the planning agents going directly to the Communication Agent for shared information? Because I wanted to reflect a bit of reality in that at a given intersection there likely won't be the connectivity back to a central all-knowing authority. The agent has to interact with its neighbors.

To obtain system metrics, TraCI must be used to query the simulation for a specific metric on every lane in the system. To me, it made the most sense for the Communications Agent to gather all of this information and publish it to its own queue. From there, the Metrics Agent gathers it, does any necessary analysis and aggregation and saves the data. The Metrics Agent stores the network configuration along with all of the aggregated metrics at the end of a simulation run.

3.6 High-Level Design

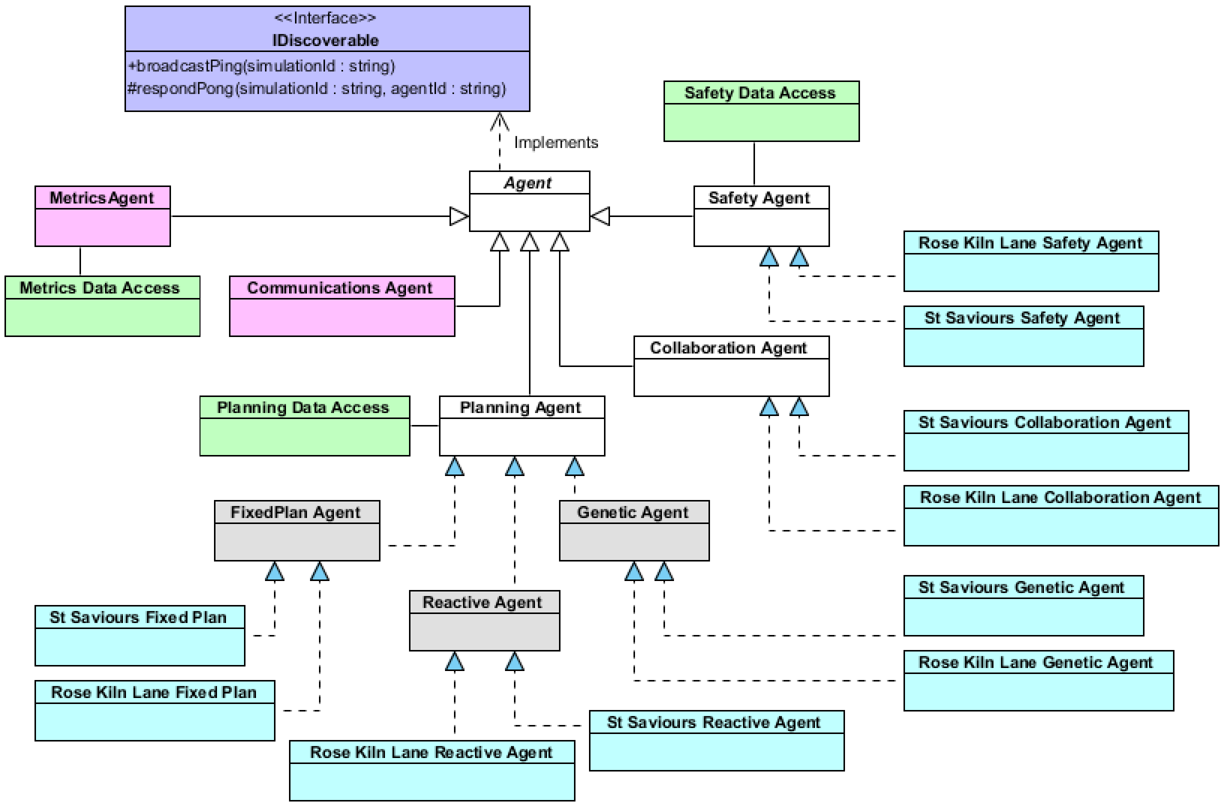

<<Interface>>
IDiscoverable
+broadcastPing(simulationId : string)
#respondPong(simulationId : string, agentId : string)
Implements
Agent
Safety Data Access
Safety Agent
Metrics Agent
Metrics Data Access
Communications Agent
Rose Kiln Lane Safety Agent
St Saviours Safety Agent
Collaboration Agent
Planning Data Access
Planning Agent
St Saviours Collaboration Agent
Rose Kiln Lane Collaboration Agent
FixedPlan Agent
Genetic Agent
Reactive Agent
St Saviours Genetic Agent
Rose Kiln Lane Genetic Agent
St Saviours Fixed Plan
Rose Kiln Lane Fixed Plan
St Saviours Reactive Agent
Rose Kiln Lane Reactive Agent

Figure 6 High Level Class Diagram

In this high level view of the design we can see how the design provides for reusability and scalability of common components. For instance, a base abstract class of Agent is inherited by the Safety Agent, the Collaboration Agent and the Planning Agent classes which are also abstract. Further concrete classes are created off of these specialized classes. The Metrics Agent and Communications Agent also take advantage of base behavior implemented in the Agent class. We also see that all Agents will implement the discoverable interface which is used to find other agents participating in the simulation. The discoverable interface is closely related to the discover protocol outlined in section 4.4. The diagram also shows us the relationship between the agent classes and the data access classes used for persisting simulation information.

3.7 Mid-Level Design

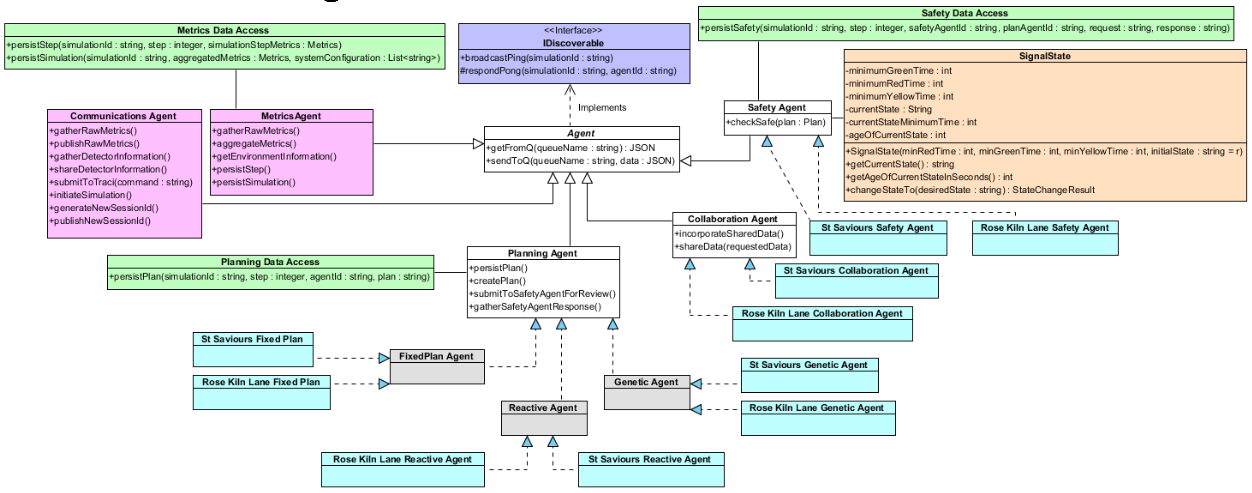

Metrics Data Access
+persistStep(simulationId : string, step : integer, simulationStepMetrics : Metrics)
+persistSimulation(simulationId : string, aggregatedMetrics : Metrics, systemConfiguration : List<string>)
<<Interface>>
IDiscoverable
+broadcastPing(simulationId : string)
#respondPong(simulationId : string, agentId : string)
Safety Data Access
+persistSafety(simulationId : string, step : integer, safetyAgentId : string, planAgentId : string, request : string, response : string)
SignalState
-minimumGreenTime : int
-minimumRedTime : int
-minimumYellowTime : int
-currentState : String
-currentStateMinimumTime : int
-ageOfCurrentState : int
+SignalState(minRedTime : int, minGreenTime : int, minYellowTime : int, initialState : string = r)
+getCurrentState() : string
+getAgeOfCurrentStateInSeconds() : int
+changeStateTo(desiredState : string) : StateChangeResult
Implements
Communications Agent
+gatherRawMetrics()
+publishRawMetrics()
+gatherDetectorInformation()
+shareDetectorInformation()
+submitToTraci(command : string)
+initiateSimulation()
+generateNewSessionId()
+publishNewSessionId()
MetricsAgent
+gatherRawMetrics()
+aggregateMetrics()
+getEnvironmentInformation()
+persistStep()
+persistSimulation()
Agent
+getFromQ(queueName : string) : JSON
+sendToQ(queueName : string, data : JSON)
Safety Agent
+checkSafe(plan : Plan)
Collaboration Agent
+incorporateSharedData()
+shareData(requestedData)
St Saviours Safety Agent
Rose Kiln Lane Safety Agent
St Saviours Collaboration Agent
Rose Kiln Lane Collaboration Agent
Planning Data Access
+persistPlan(simulationId : string, step : integer, agentId : string, plan : string)
Planning Agent
+persistPlan()
+createPlan()
+submitToSafetyAgentForReview()
+gatherSafetyAgentResponse()
St Saviours Fixed Plan
Rose Kiln Lane Fixed Plan
FixedPlan Agent
Genetic Agent
St Saviours Genetic Agent
Rose Kiln Lane Genetic Agent
Reactive Agent
Rose Kiln Lane Reactive Agent
St Saviours Reactive Agent

Figure 7Class Diagram

I the mid-level design we start to see some of the methods that I anticipate will be necessary for the project. For instance, with the communications agent the proposed methods relate back to the System Requirements in the project Vision document. In this diagram we also see the emergence of a Signal State container which will be used by the Safety Agent for managing the current state of the signal, properties of the signal and requested changes in state.

4 Component Interaction

In this section I’ll review the typical system initialization sequence, the looping of the doing simulation sequence of events and finally the teardown or finish sequence.

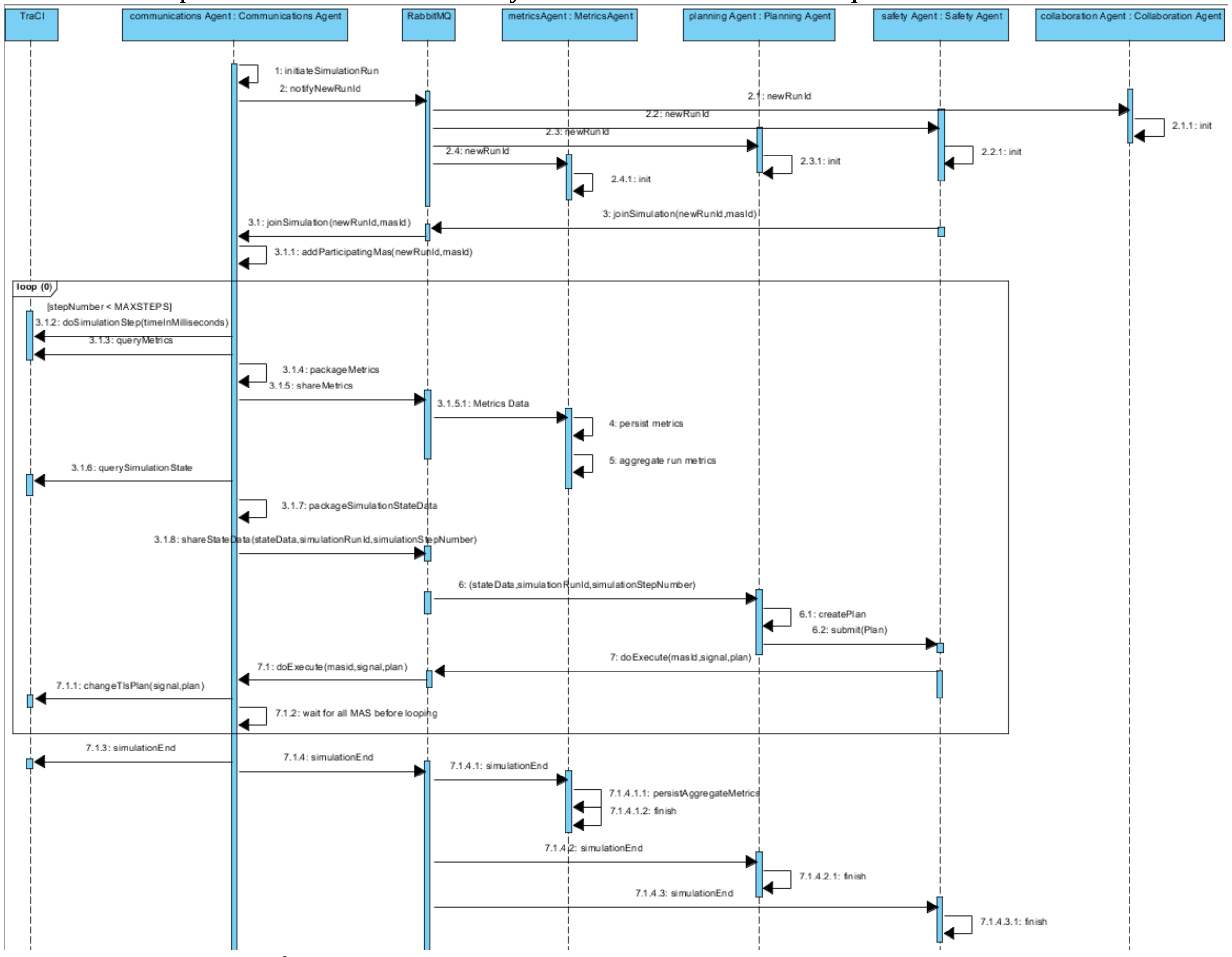
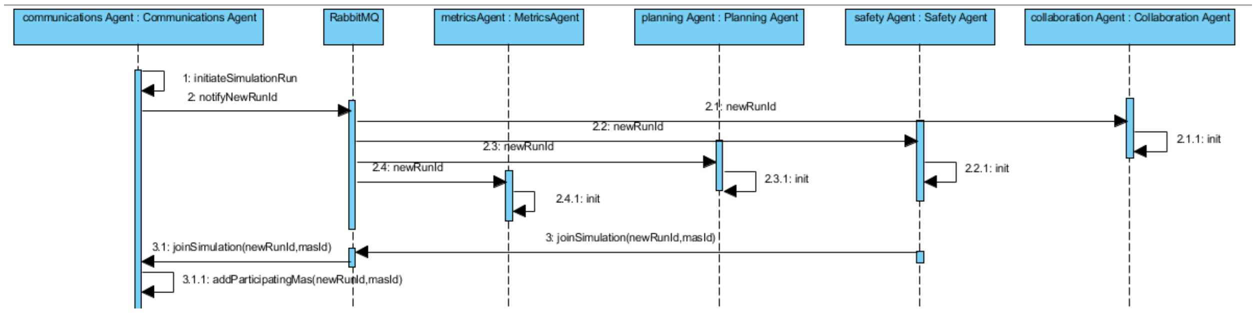

TraCI
communications Agent : Communications Agent
RabbitMQ
metricsAgent : MetricsAgent
planning Agent : Planning Agent
safety Agent : Safety Agent
collaboration Agent : Collaboration Agent
1: initiateSimulationRun
2: notifyNewRunId
2.1: newRunId
2.1.1: init
2.2: newRunId
2.2.1: init
2.3: newRunId
2.3.1: init
2.4: newRunId
2.4.1: init
3: joinSimulation(newRunId,masId)
3.1: joinSimulation(newRunId,masId)
3.1.1: addParticipatingMas(newRunId,masId)
loop (0)
[stepNumber < MAXSTEPS]
3.1.2: doSimulationStep(timeInMilliseconds)
3.1.3: queryMetrics
3.1.4: packageMetrics
3.1.5: shareMetrics
3.1.5.1: Metrics Data
4: persist metrics
5: aggregate run metrics
3.1.6: querySimulationState
3.1.7: packageSimulationStateData
3.1.8: shareStateData(stateData,simulationRunId,simulationStepNumber)
6: (stateData,simulationRunId,simulationStepNumber)
6.1: createPlan
6.2: submit(Plan)
7: doExecute(masId,signal,plan)
7.1: doExecute(masid,signal,plan)
7.1.1: changeTlsPlan(signal,plan)
7.1.2: wait for all MAS before looping
7.1.3: simulationEnd
7.1.4: simulationEnd
7.1.4.1: simulationEnd
7.1.4.1.1: persistAggregateMetrics
7.1.4.1.2: finish
7.1.4.2: simulationEnd
7.1.4.2.1: finish
7.1.4.3: simulationEnd
7.1.4.3.1: finish

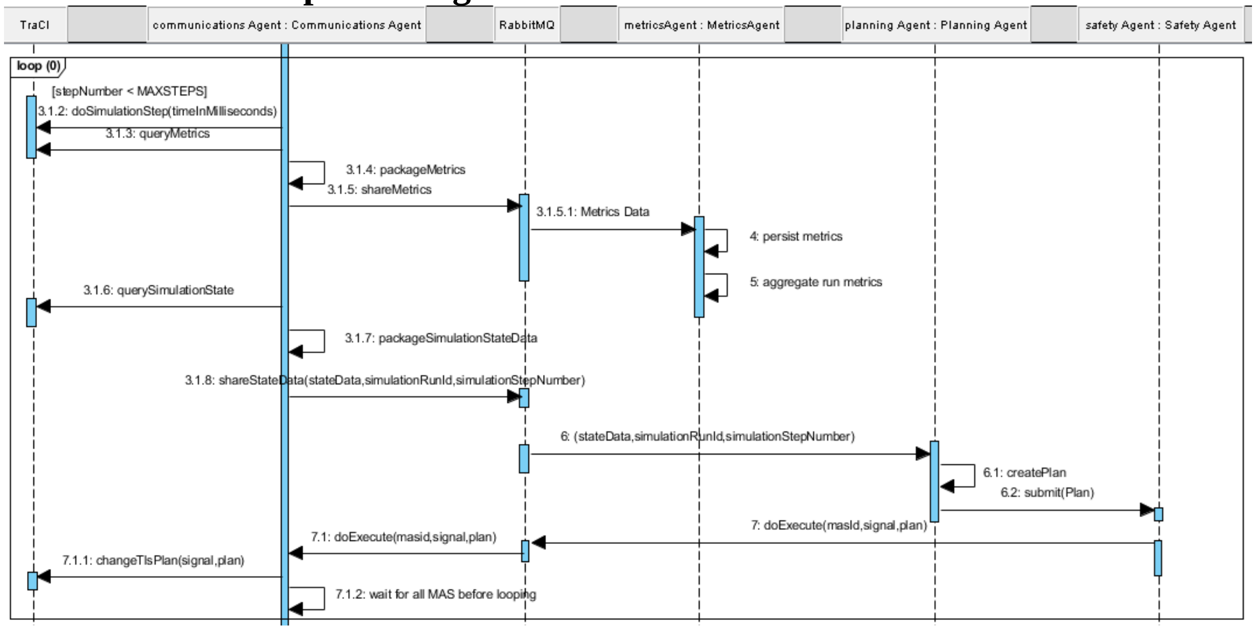
Figure 8Sequence diagram for process interactions

4.1 Initialization interaction


communications Agent : Communications Agent
RabbitMQ
metricsAgent : MetricsAgent
planning Agent : Planning Agent
safety Agent : Safety Agent
collaboration Agent : Collaboration Agent
1: initiateSimulationRun
2: notifyNewRunId
2.1: newRunId
2.1.1: init
2.2: newRunId
2.2.1: init
2.3: newRunId
2.3.1: init
2.4: newRunId
2.4.1: init
3: joinSimulation(newRunId,masId)
3.1: joinSimulation(newRunId,masId)
3.1.1: addParticipatingMas(newRunId,masId)

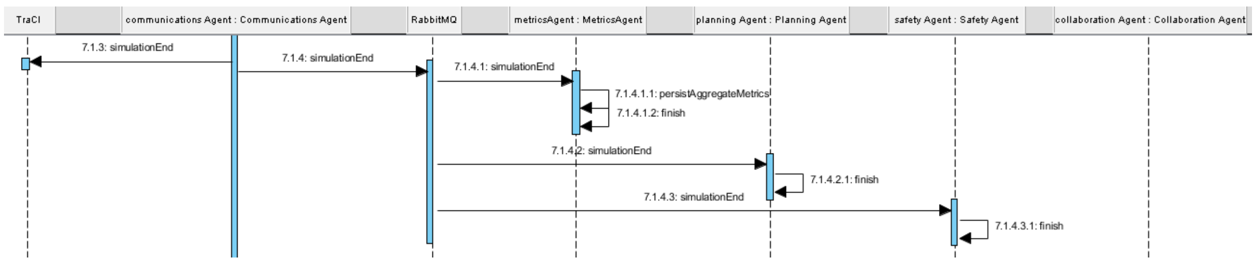
In the system startup process, the Communications Agent creates a new simulation run identifier. It then shares that on a broadcast exchange on RabbitMQ. The subscribed (already started) metrics agent, planning agent, safety agent and potentially collaboration agent all receive the new session identifier and perform any required local initialization. The safety agent also responds back to the communications agent via RabbitMQ that it is “joining” the simulation. The communication agent picks up this message/command and incorporates that knowledge. Now, the communications agent will know when all agents have reported in for each system simulation step.

4.2 In simulation processing interaction


TraCI
communications Agent : Communications Agent
RabbitMQ
metricsAgent : MetricsAgent
planning Agent : Planning Agent
safety Agent : Safety Agent
loop (0)
[stepNumber < MAXSTEPS]
3.1.2: doSimulationStep(timeInMilliseconds)
3.1.3: queryMetrics
3.1.4: packageMetrics
3.1.5: shareMetrics
3.1.5.1: Metrics Data
4: persist metrics
5: aggregate run metrics
3.1.6: querySimulationState
3.1.7: packageSimulationStateData
3.1.8: shareStateData(stateData,simulationRunId,simulationStepNumber)
6: (stateData,simulationRunId,simulationStepNumber)
6.1: createPlan
6.2: submit(Plan)
7: doExecute(masId,signal,plan)
7.1: doExecute(masid,signal,plan)
7.1.1: changeTlsPlan(signal,plan)
7.1.2: wait for all MAS before looping

While the maximum number of simulation steps has not been reached, the communications agent (CA) queries TraCI for simulation metrics. The metrics are packaged and shared via RabbitMQ with the metrics agent. The metrics agent does any necessary processing/aggregation and saves the metrics. Next the CA queries TraCI for the current sensor simulation information. That information is shared with subscribing planning agents via RabbitMQ. Based on the received information, the planning agents create a plan and submit it to the Safety Agent for review. If the plan is safe, the plan is sent by the Safety Agent to the CA via RabbitMQ. The CA then executes the command. The CA waits for all joined agents to report in before kicking off the next simulation step.

4.3 Simulation end interaction


TraCI
communications Agent : Communications Agent
RabbitMQ
metricsAgent : MetricsAgent
planning Agent : Planning Agent
safety Agent : Safety Agent
collaboration Agent : Collaboration Agent
7.1.3: simulationEnd
7.1.4: simulationEnd
7.1.4.1: simulationEnd
7.1.4.1.1: persistAggregateMetrics
7.1.4.1.2: finish
7.1.4.2: simulationEnd
7.1.4.2.1: finish
7.1.4.3: simulationEnd
7.1.4.3.1: finish

In the simulation end the communications agent has reached the max number of simulation steps. Therefore, it broadcasts a simulation end to all participating parties. The Metrics Agent performs final aggregation of metrics and stores them. The planning agent and safety agent perform any finishing tasks.

4.4 Discovery Protocol

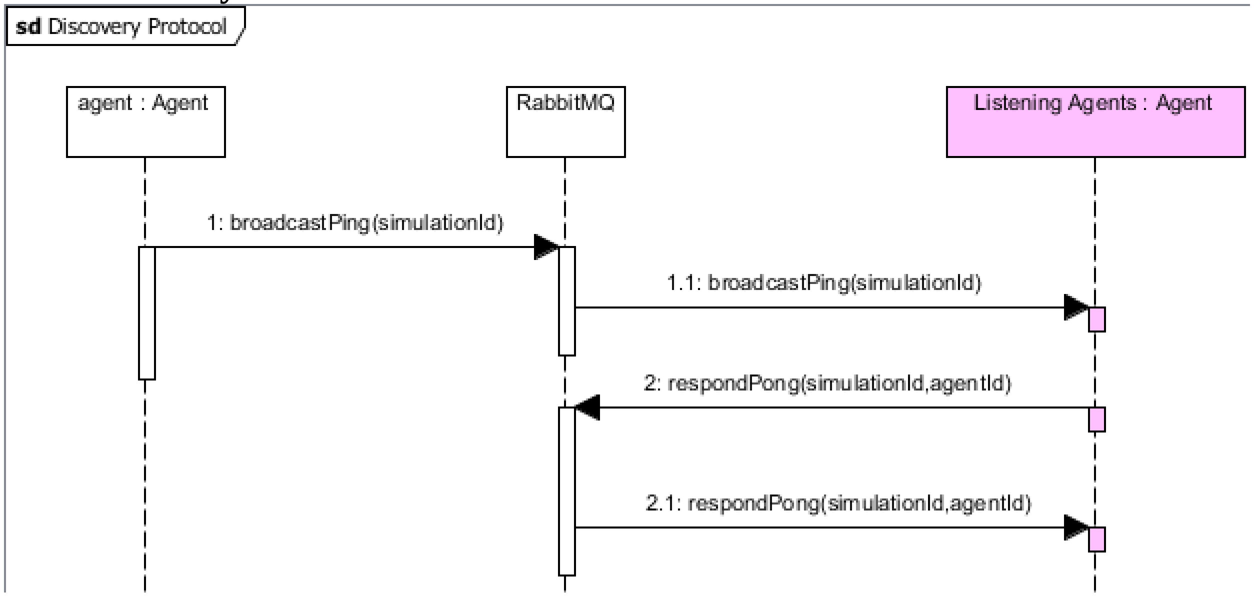

sd Discovery Protocol
agent : Agent
RabbitMQ
Listening Agents : Agent
1: broadcastPing(simulationId)
1.1: broadcastPing(simulationId)
2: respondPong(simulationId,agentId)
2.1: respondPong(simulationId,agentId)

Figure 9 Discovery Protocol

In the discovery protocol, an agent sends a broadcast ping along with the current simulation Id to a discovery queue. All agents are monitoring the discovery queue. On “hearing” a ping, the listening agent responds with the simulation Id as well as its agentId. The agentId is made up of the type of Agent and the MAS Node Id. Any agent that is listening to the pong response queue will “hear” other agents on the network.

5 USE/OCL Model

5.1 Overview

This section will provide a formal specification that the safety agent enforces minimum time per light color and that the lights must change in a rotation of green, yellow, red. The system combines all traffic light signals at an intersection into a single command.

The interactions involved are Planning Agent sends plan to Safety Agent. Safety Agent evaluates. If ok, the Safety Agent sends the plan on to the Communications Agent. If not ok, the Safety Agent notifies the planning agent that the plan is not acceptable and the cause/reason why.

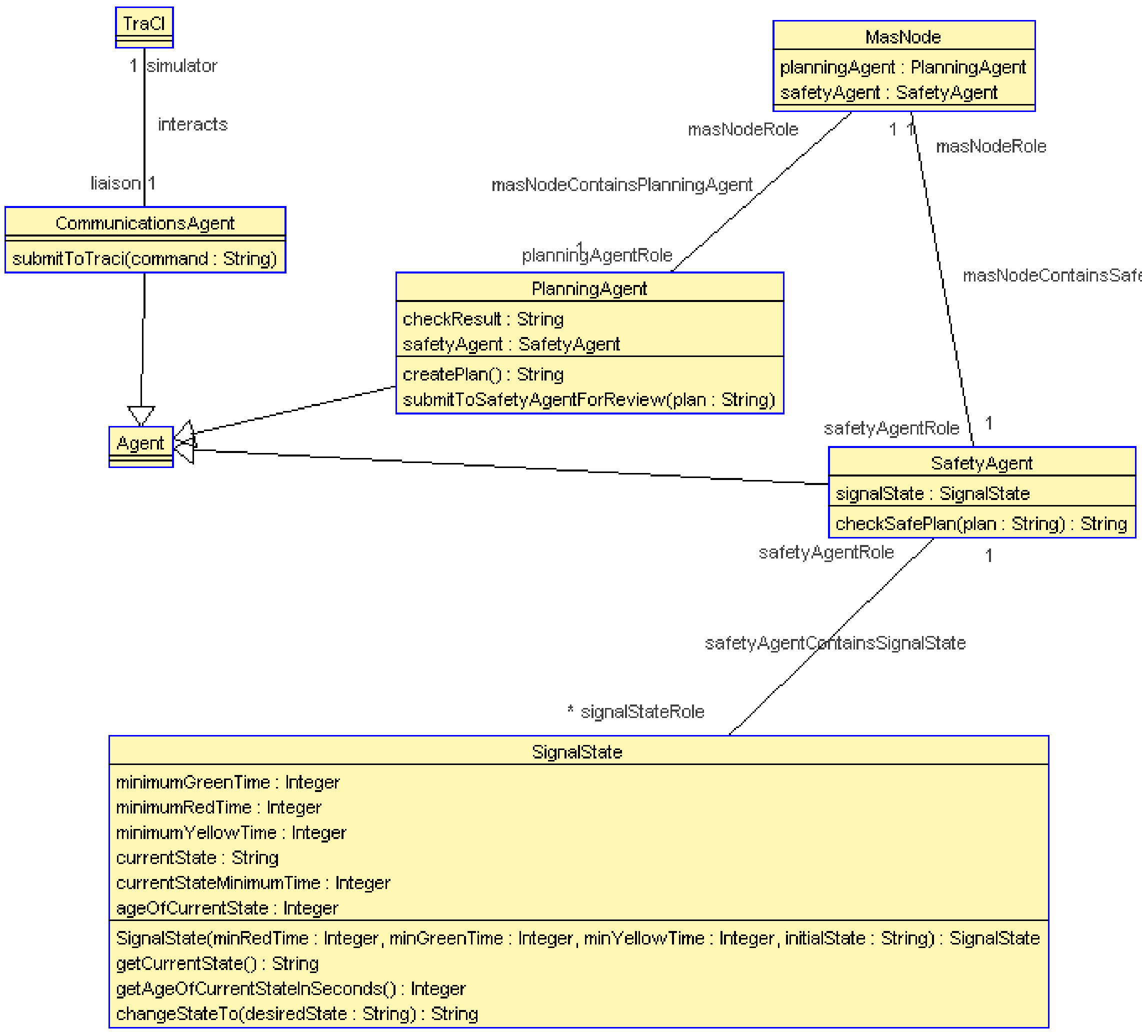

TraCI
1 simulator
interacts
liaison 1
CommunicationsAgent
submitToTraci(command : String)
Agent
MasNode
planningAgent : PlanningAgent
safetyAgent : SafetyAgent
masNodeRole
1 1
masNodeRole
masNodeContainsPlanningAgent
1
planningAgentRole
masNodeContainsSafetyAgent
PlanningAgent
checkResult : String
safetyAgent : SafetyAgent
createPlan() : String
submitToSafetyAgentForReview(plan : String)
safetyAgentRole
1
SafetyAgent
signalState : SignalState
checkSafePlan(plan : String) : String
safetyAgentRole
1
safetyAgentContainsSignalState
* signalStateRole
SignalState
minimumGreenTime : Integer
minimumRedTime : Integer
minimumYellowTime : Integer
currentState : String
currentStateMinimumTime : Integer
ageOfCurrentState : Integer
SignalState(minRedTime : Integer, minGreenTime : Integer, minYellowTime : Integer, initialState : String) : SignalState
getCurrentState() : String
getAgeOfCurrentStateInSeconds() : Integer
changeStateTo(desiredState : String) : String

Figure 10Class Diagram for formal specification

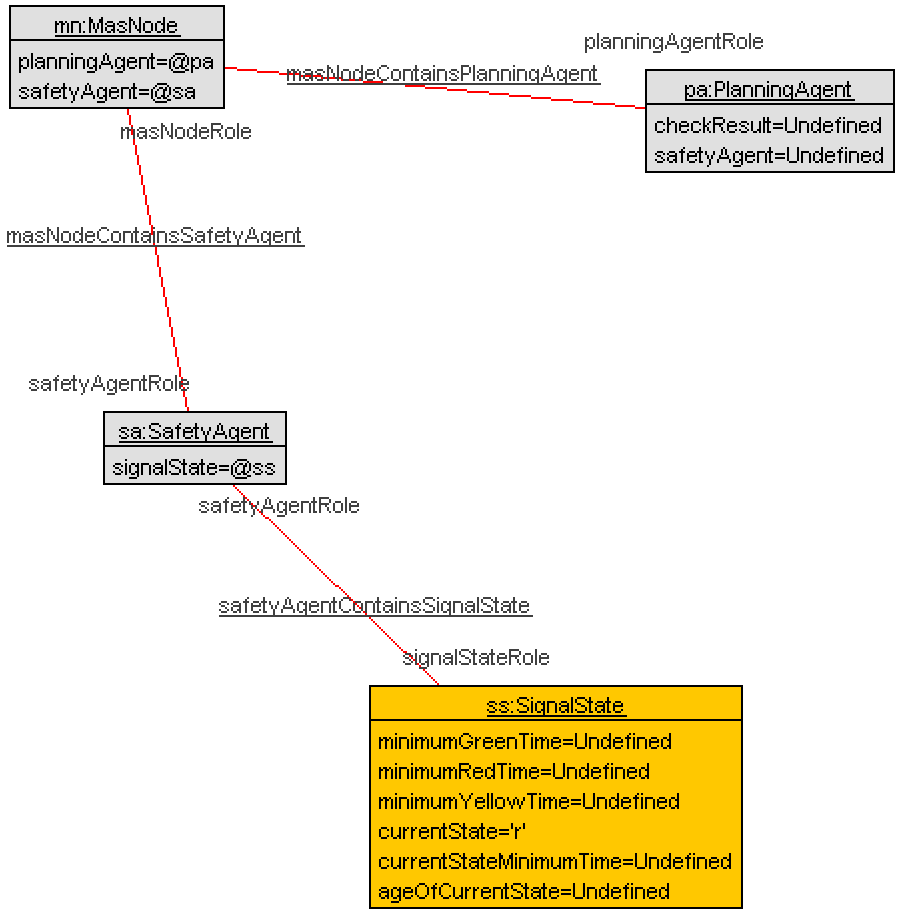


Figure 11Instance Diagram

5.2 USE OCL Code

```
-- CIS 895 MSE Project Formal Specification MACTS Architecture
-- File: macts.use
-- Author: Bryan Nehl
--
-- Description: Aspects of the MultiAgent Control of Traffic Signals
-- models specified in USE OCL
--
-- This is a formal specification that:
--    The safety agent enforces minimum time per light color and
--    The lights must change in a rotation of green, yellow, red.
--    The system combines all traffic light signals at an intersection into a
single command.
--
-- The interactions involved are:
-- Planning Agent sends plan to Safety Agent.
-- Safety Agent evaluates.
--          If ok, the Safety Agent sends the plan on to the Communications
Agent.
--          If not ok, the Safety Agent notifies the planning agent that
--                the plan is not acceptable and the cause/reason why.

model Macts

-- classes -------------------

class TraCI
end

-- abstract, no instances of
class Agent
end

-- one
class CommunicationsAgent < Agent
operations
      submitToTraci(command : String)
end

-- MAS Node --------------------------------------------------------
-- abstract
-- may only have one "PlanningAgent" type per node
class MasNode
attributes
      planningAgent : PlanningAgent
      safetyAgent : SafetyAgent
end

-- SAFETY AGENT ----------------------------------------------------
class SafetyAgent < Agent
```

```
attributes
      signalState : SignalState
operations
      checkSafePlan(plan : String) : String
end

-- PLANNING AGENT -----------------------------------------------------
class PlanningAgent < Agent
attributes
      checkResult : String
      safetyAgent : SafetyAgent
operations
      createPlan() : String
      submitToSafetyAgentForReview(plan : String)
end

-- SIGNAL STATE -------------------------------------------------------
class SignalState
attributes
      minimumGreenTime : Integer
      minimumRedTime : Integer
      minimumYellowTime : Integer
      currentState : String
      currentStateMinimumTime : Integer
      ageOfCurrentState : Integer
operations
      SignalState(minRedTime : Integer, minGreenTime : Integer, minYellowTime
: Integer, initialState : String) : SignalState
      getCurrentState() : String
      getAgeOfCurrentStateInSeconds() : Integer
      changeStateTo( desiredState : String) : String
end

-- associations -----------------

association interacts between
  TraCI[1] role simulator;
  CommunicationsAgent[1] role liaison;
end

association safetyAgentContainsSignalState between
      SafetyAgent[1] role safetyAgentRole;
      SignalState[0..*] role signalStateRole;
end

association masNodeContainsPlanningAgent between
      MasNode[1] role masNodeRole;
      PlanningAgent[1] role planningAgentRole;
end
```

```
association masNodeContainsSafetyAgent between
      MasNode[1] role masNodeRole;
      SafetyAgent[1] role safetyAgentRole;
end

-- constraints --------------------

constraints

-- there is only one communications agent
context CommunicationsAgent inv OneCommAgent:
      CommunicationsAgent.allInstances->size() = 1

-- the mas node contains two agents.
-- one is a planning agent and the other a safety agent
context mn:MasNode
      inv planningAgentIsAPlanningAgent:
        mn.planningAgent.oclIsKindOf(PlanningAgent)
      inv safetyAgentIsSafetyAgent:
            mn.safetyAgent.oclIsKindOf(SafetyAgent)

context SignalState
      -- check that the current state is in the set of valid states
      inv validCurrentState:
            Set{'G','r','y'} -> includes(self.currentState)

-- SAFETY AGENT -----------------------------------------------------
-- LIGHTS MUST CHANGE IN CORRECT ORDER: green, yellow, red ------------------
--
-- valid light colors are: { G, g, r, y }
-- post condition checks to the changeStateTo operation of the SignalState
-- the current state should either be the same as the previous state OR
-- it should be the next state in the cycle
context SignalState::changeStateTo(desiredState:String):String

post yellowFollowsGreen:
      self.currentState@pre = 'G' implies Set{'G','y'} ->
includes(self.currentState)
post redFollowsYellow:
      self.currentState@pre = 'y' implies Set{'y','r'} ->
includes(self.currentState)
post greenFollowsRed:
      self.currentState@pre = 'r' implies Set{'r','G'} ->
includes(self.currentState)

-- MINIMUM TIME PER LIGHT COLOR
-- the currentStateMinimumTime should correspond to the current state
-- Green, yellow and red have different minimum state times
pre greenMinTime:
      self.currentState = 'G' implies currentStateMinimumTime =
self.minimumGreenTime

pre yellowMinTime:
      self.currentState = 'y' implies currentStateMinimumTime =
self.minimumYellowTime

pre redMinTime:
      self.currentState = 'r' implies currentStateMinimumTime =
self.minimumRedTime

-- after every state change the new age of current state should be
-- 0 if the state changed or incremented by 1 if the state stayed the same
post ageOfCurrentStateIncrements:
      Set{0, 1 + self.ageOfCurrentState@pre } ->
includes(self.ageOfCurrentState )

-- stated another way, if the state set to is the same as the previous state
-- then increment by 1, otherwise it is a new state so it should be set to 0
post ageIncrements: if self.currentState@pre = desiredState then
                  self.ageOfCurrentState = self.ageOfCurrentState@pre + 1
            else self.ageOfCurrentState = 0 endif

-- again, if the current state is different than the previous current state,
the age should be 0
post lightStateChangingZerosAge:
      self.currentState <> self.currentState@pre implies
self.ageOfCurrentState = 0

-- if the age of the current state is less than the previous minimum time
-- then the current state should be the same as the previous
post minimumStateTimeEnforced:
      self.ageOfCurrentState < currentStateMinimumTime@pre implies
self.currentState = self.currentState@pre

-- context SafetyAgent::checkSafePlan(plan : String) : String
      -- safetyAgent.checkSafePlan(plan)
      -- receives messages with error issues from safety agent

-- SAFE PLAN: let the planning agent know if plan is acceptable
-- UNSAFE PLAN: if not, give reason why

-- SUBMIT PLAN: to the Communications Agent

-- COMMUNICATIONS AGENT --------------------------------------------
-- receives plan from Safety Agent and executes/submits it to TraCI
```

5.3 Initialization Script

```
!create ss : SignalState
!set ss.currentState := 'r'
!create sa : SafetyAgent
!create mn : MasNode
!create pa : PlanningAgent
!set mn.planningAgent := pa
!set mn.safetyAgent := sa
!set sa.signalState := ss
!insert (mn,pa) into masNodeContainsPlanningAgent
!insert (mn,sa) into masNodeContainsSafetyAgent
!insert (sa,ss) into safetyAgentContainsSignalState

!set ss.minimumGreenTime := 3
!set ss.minimumYellowTime := 2
!set ss.minimumRedTime := 1
!set ss.currentStateMinimumTime := 1
!set ss.ageOfCurrentState := 0
```