Final Documentation

Autonomous Systems A Lab

Team Members:

Stephanie Chinenye Okosa

Maxim Emile Speczyk

Muhammad Umer Bin Yaqoob

Abstract:

This project presents an autonomous truck platooning system's design, implementation, and testing. This technological solution promises to revolutionise the freight industry by increasing efficiency and safety while reducing costs. The system, modelled using UML, incorporates a machine learning algorithm to select the platoon leader based on multiple parameters. It considers various scenarios like steering, lane change, braking, joining, and leaving the platoon, including potential communication failures implemented in UPPAAL. The project will culminate in a comprehensive simulation environment using Python.

Introduction:

Autonomous vehicles have revolutionised our society, bringing with them a myriad of opportunities and challenges. Our team has embarked on a project to develop an autonomous truck platooning system to address these.

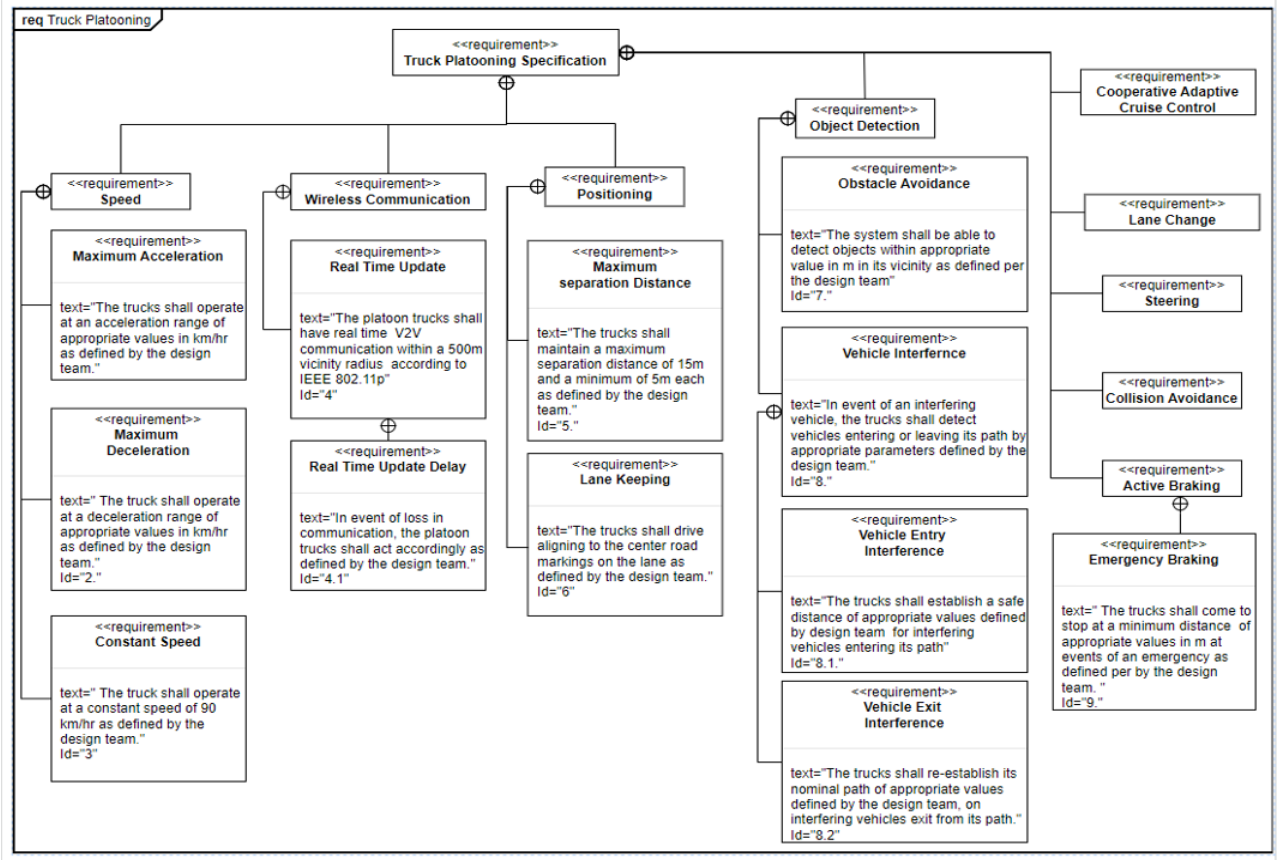
Truck platooning essentially involves several trucks equipped with advanced driver assistance systems. These trucks follow each other closely on a motorway, literally forming a convoy or 'platoon'. The importance of this arrangement lies in the many benefits it offers. These include improved road safety, reduced fuel consumption, lower CO2 emissions and more efficient traffic flow.

Brainstorming and UML designing

In the early stages of our project, each member of our team brainstormed and generated detailed scenarios for our autonomous truck platooning system. This step helped us to identify key functional requirements and gain a deep understanding of the problem from the user's perspective. We then used UML (Unified Modelling Language),to visually represent these scenarios and the system as a whole, detailing the relationships and data flow between entities. The UML designs served as blueprints for the architecture of our system, providing a solid foundation for model specification and final implementation, while facilitating efficient communication and shared understanding among team members and stakeholders.

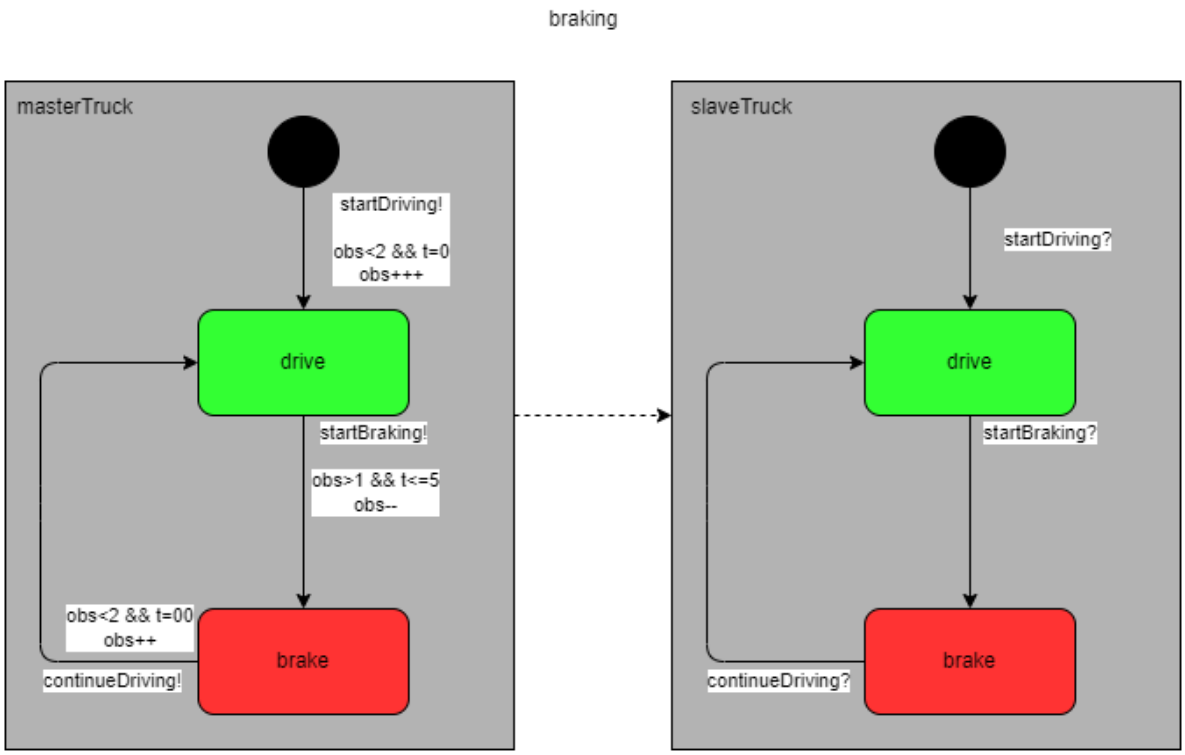
Requirement Diagram

Initial requirements of the truck platooning system were discussed, and the following requirements were identified as critical for the system to function properly.



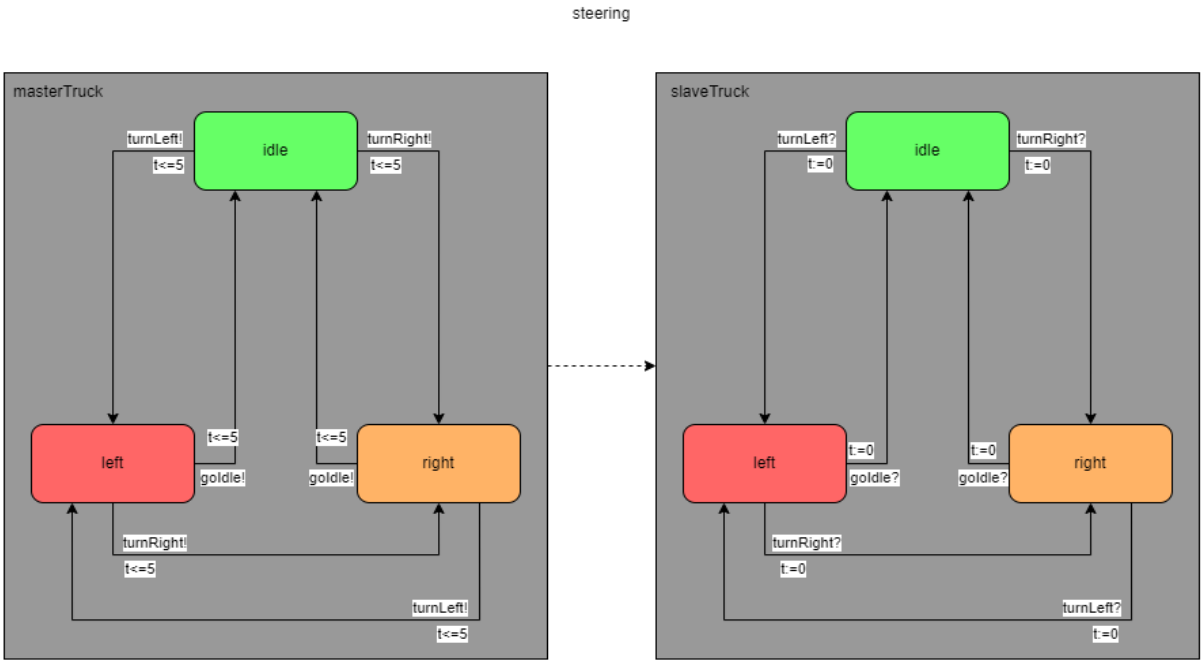
Scenario 1

Braking: Our autonomous truck platooning project focuses on synchronised braking mechanisms across the fleet. When the master truck initiates braking, every other truck must follow suit to maintain safe following distances and avoid collisions. This is achieved through efficient communication paths between the lead and following trucks. The behaviour of this brake control system is illustrated by a state machine diagram in our system design.



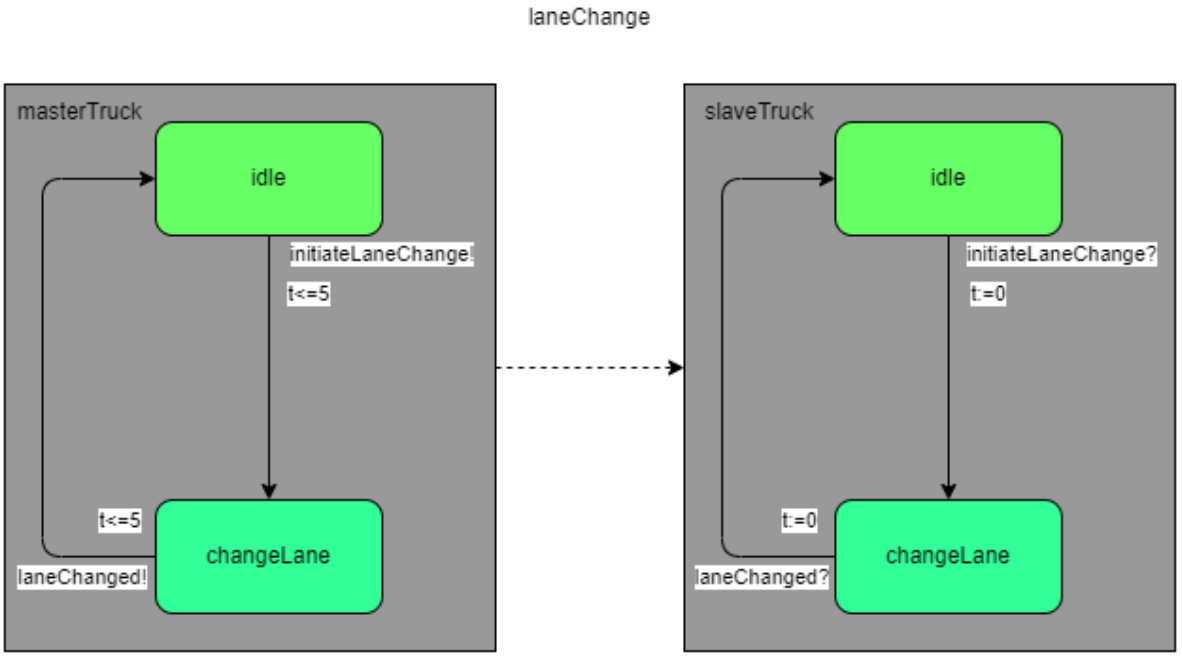
Scenario 2

Steering: In our pursuit to design an autonomous truck platooning system, a fundamental goal was to ensure the efficient 'following' capability of subordinate, or 'slave', trucks within the platoon. This is achieved by implementing a communication protocol wherein the leading, or 'master', truck transmits directional commands to the follower trucks. The communication between the master and slave trucks is pivotal in managing coordinated manoeuvres such as turns, facilitating seamless and synchronized transitions between directional states, e.g., initiating rightward or leftward shifts. The behaviour of this brake control system is illustrated by a state machine diagram in our system design.



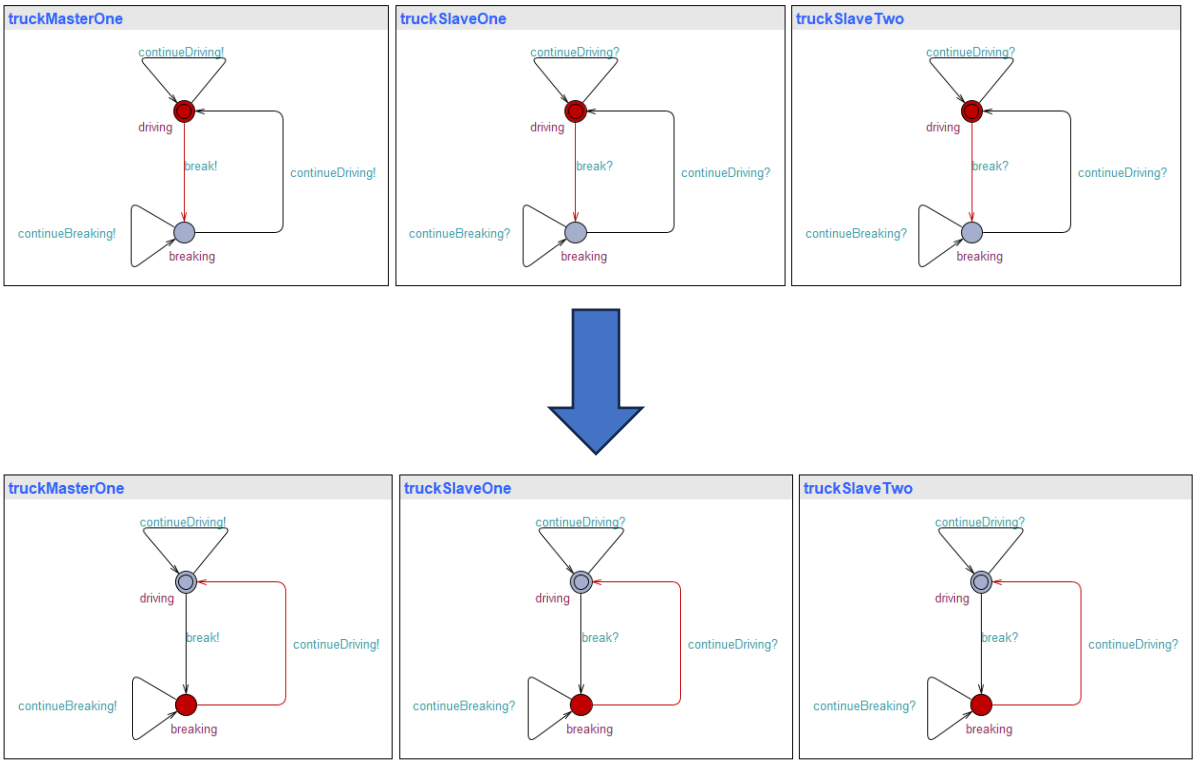
Scenario 3

Lane Change: A key element of our autonomous truck platooning project is the coordination of lane changes within the fleet. The aim is to ensure that when the master truck makes a lane change, the slave trucks systematically reproduce the action to maintain the integrity of the platoon. These lane changes are managed by a highly responsive communication system between the trucks. The detailed implementation of the lane-changing mechanism is represented by a state machine diagram in our system model.



UPPAAL Implementation

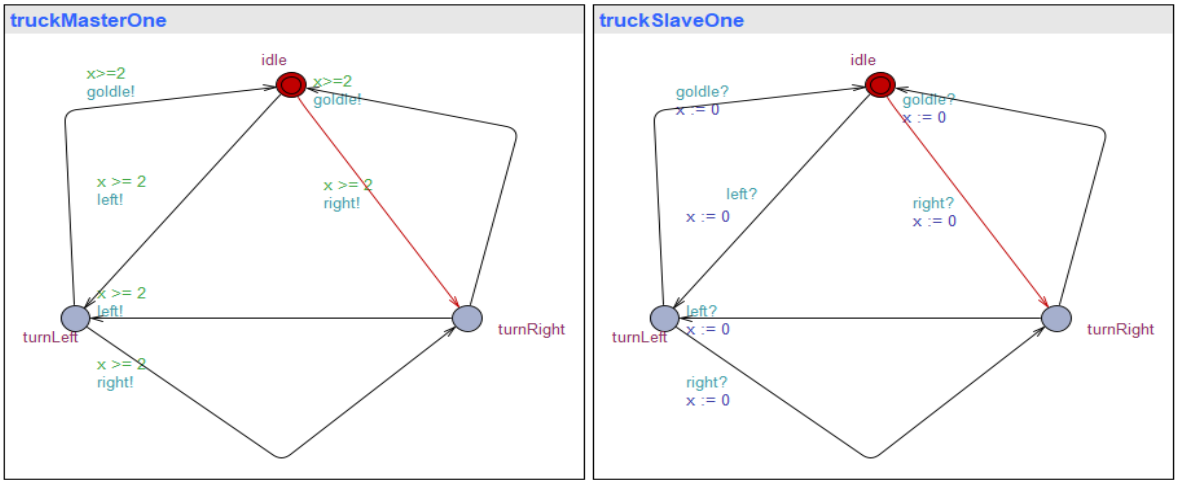
Braking:

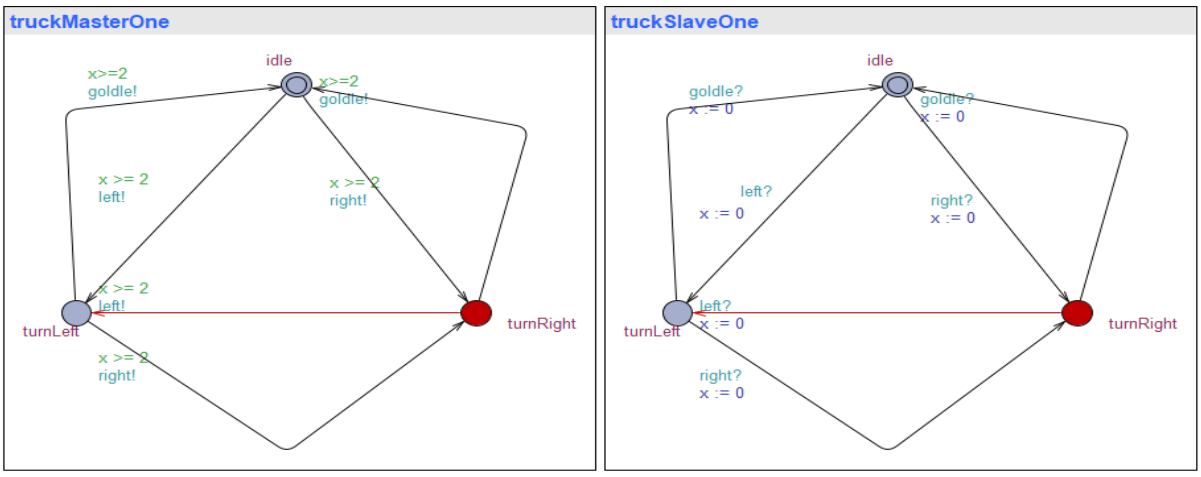


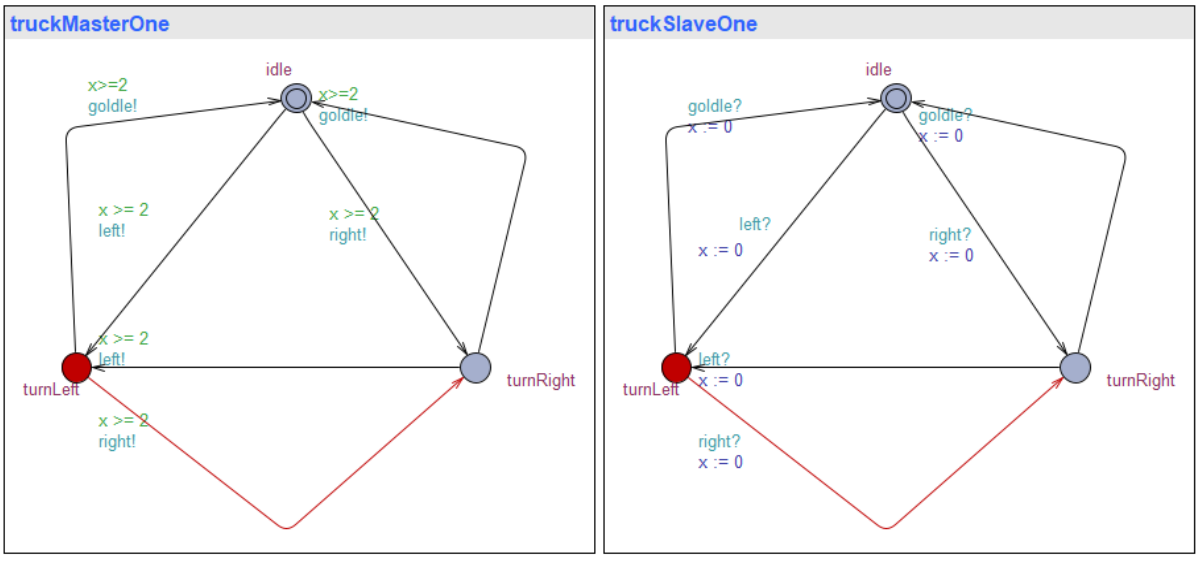
The scenario encompasses an autonomous truck platoon comprising one master and two slave trucks, each modelled in UPPAAL as 'truckMaster' and 'truckSlave' respectively. The master truck leads the platoon, broadcasting driving instructions via three primary channels - 'continueDriving', 'Brake', and 'continueBraking'. Depending on these signals, the slave trucks respond by maintaining speed, applying brakes, or continuing to brake. This synchronization enables the platoon to operate harmoniously and safely. Future queries can help validate and verify model performance and safety.

Steering:

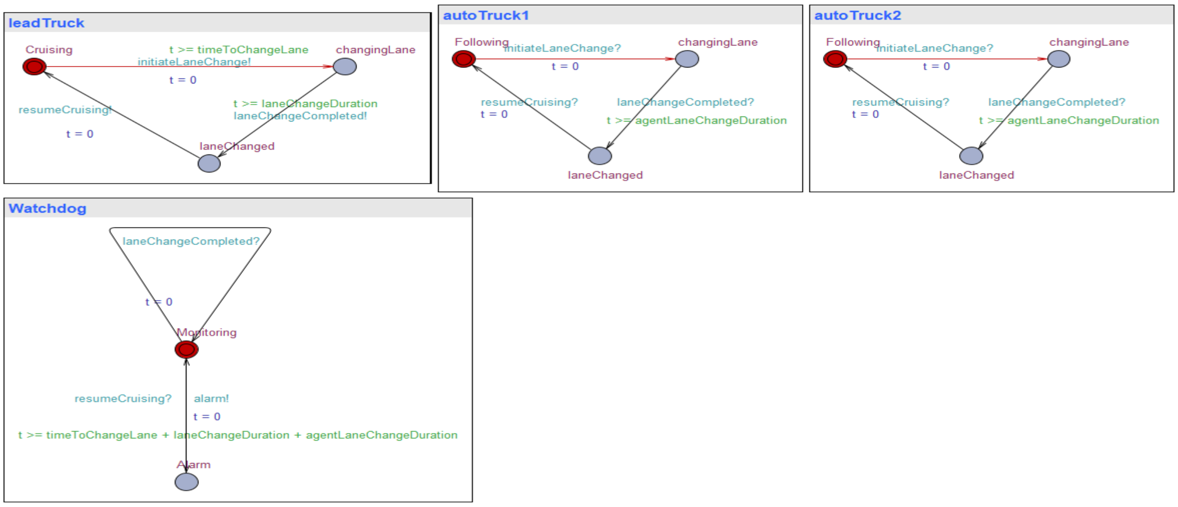
This scenario models a two-truck platoon system where one truck acts as the master and the other follows as the slave. Using UPPAAL, the steering behaviour of both trucks is defined by three states: idle, turnLeft and turnRight. The master truck's state transitions are determined by a global clock and its own synchronisation events, while the slave truck mimics the master's steering actions. The model thus demonstrates a coordinated steering system in a truck platoon, ensuring that the slave truck accurately replicates the directions of the master.

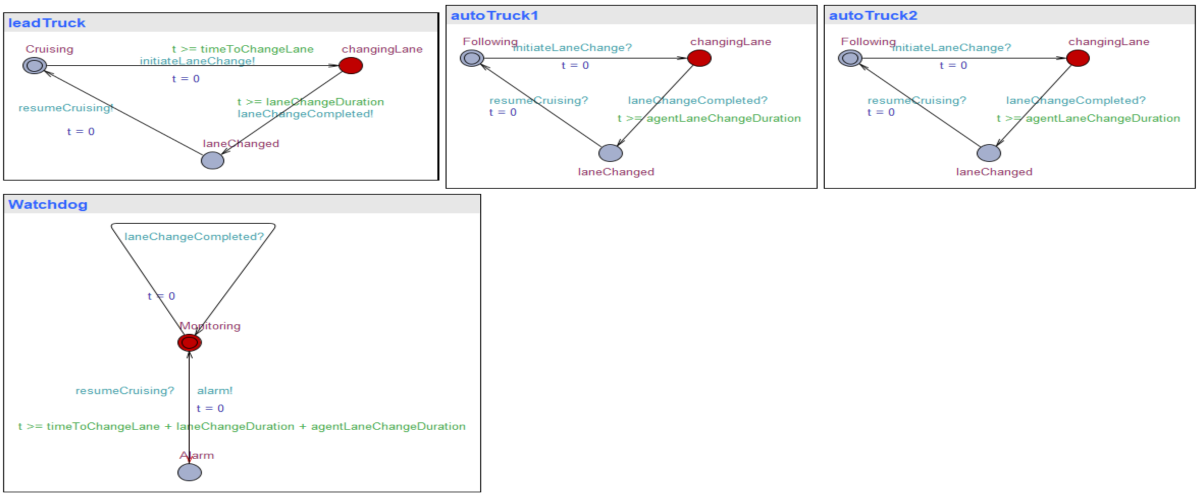

truckMasterOne
idle
x>=2
goldle!
x>=2
goldle!
x >= 2
left!
x >= 2
right!
x >= 2
left!
turnLeft
turnRight
x >= 2
right!
truckSlaveOne
idle
goldle?
x := 0
goldle?
x := 0
left?
x := 0
right?
x := 0
left?
x := 0
turnLeft
turnRight
right?
x := 0

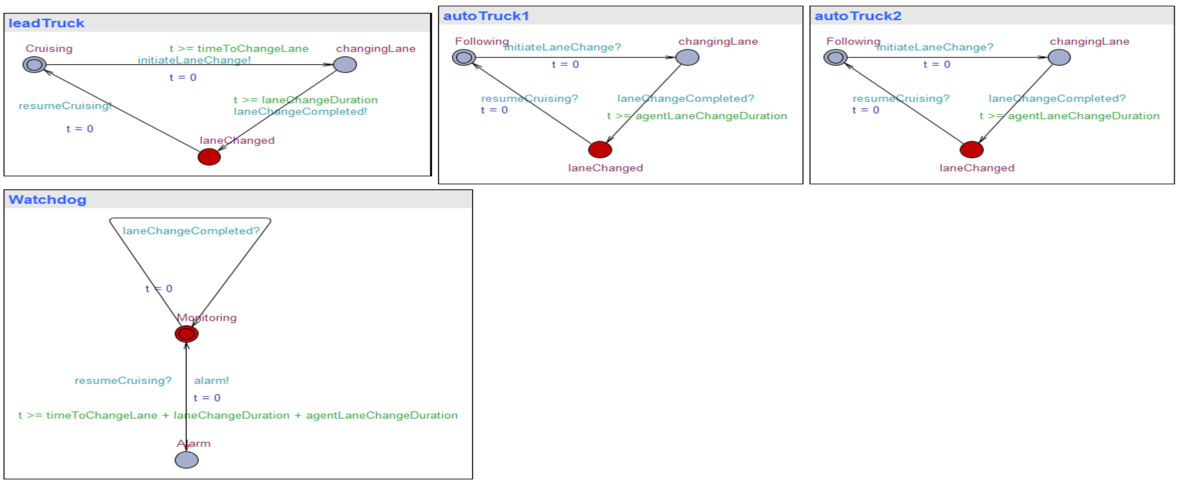

truckMasterOne
idle
x>=2
goldle!
x>=2
goldle!
x >= 2
left!
x >= 2
right!
x >= 2
left!
turnLeft
turnRight
x >= 2
right!
truckSlaveOne
idle
goldle?
x := 0
goldle?
x := 0
left?
x := 0
right?
x := 0
left?
x := 0
turnLeft
turnRight
right?
x := 0


truckMasterOne
idle
x>=2
goldle!
x>=2
goldle!
x >= 2
left!
x >= 2
right!
x >= 2
left!
turnLeft
turnRight
x >= 2
right!
truckSlaveOne
idle
goldle?
x := 0
goldle?
x := 0
left?
x := 0
right?
x := 0
left?
x := 0
turnLeft
turnRight
right?
x := 0

Lane Change:

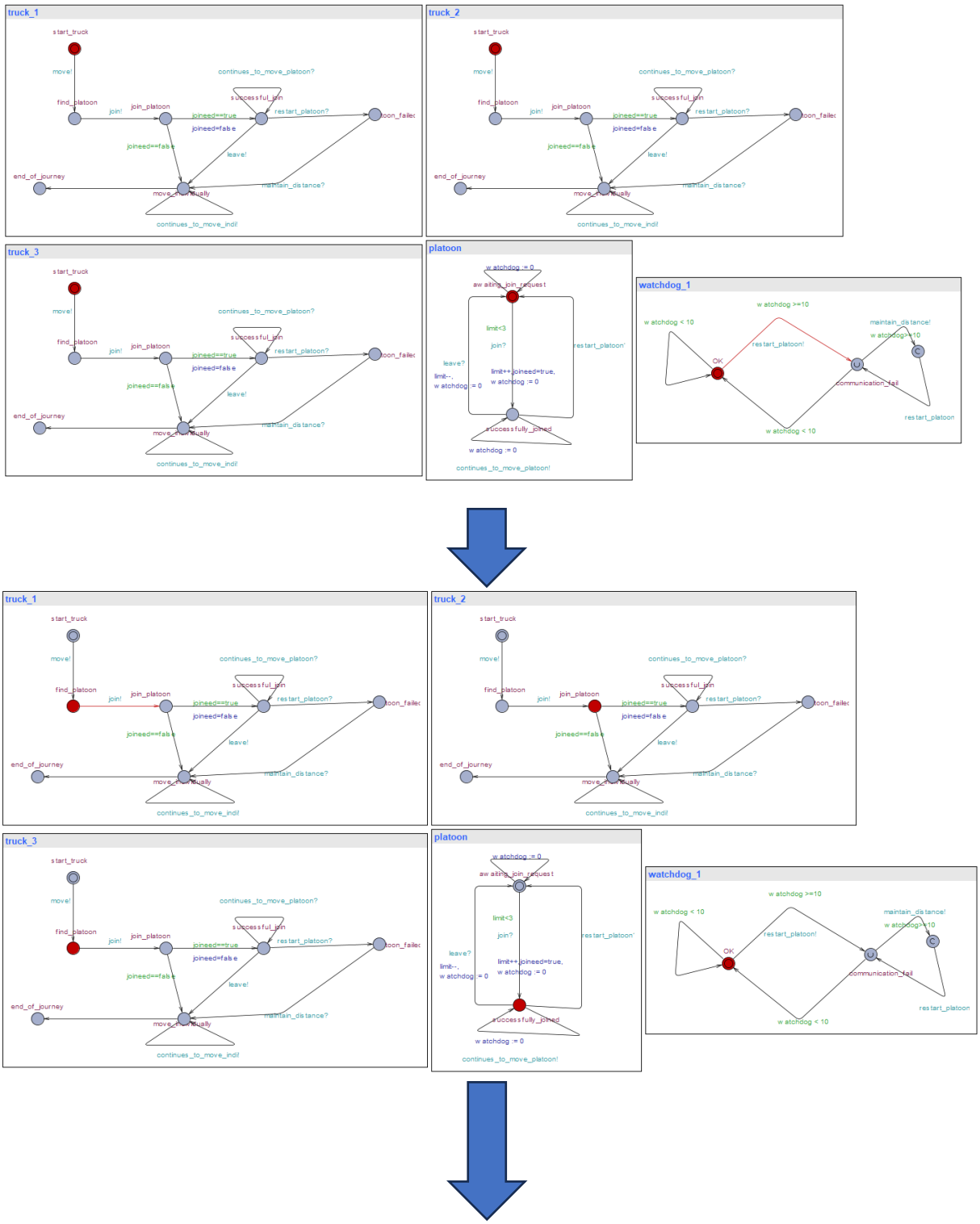

leadTruck
Cruising
t >= timeToChangeLane
initiateLaneChange!
changingLane
t = 0
resumeCruising!
t = 0
t >= laneChangeDuration
laneChangeCompleted!
laneChanged
autoTruck1
Following
initiateLaneChange?
changingLane
t = 0
resumeCruising?
t = 0
laneChangeCompleted?
t >= agentLaneChangeDuration
laneChanged
autoTruck2
Following
initiateLaneChange?
changingLane
t = 0
resumeCruising?
t = 0
laneChangeCompleted?
t >= agentLaneChangeDuration
laneChanged
Watchdog
laneChangeCompleted?
t = 0
Monitoring
resumeCruising?
alarm!
t = 0
t >= timeToChangeLane + laneChangeDuration + agentLaneChangeDuration
Alarm

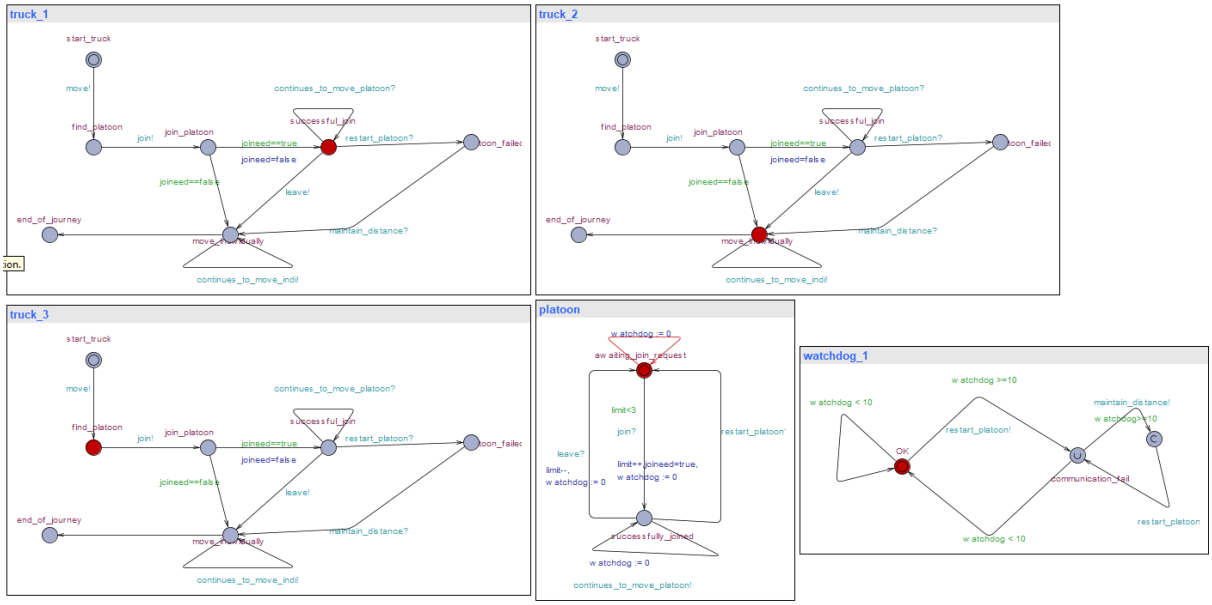

leadTruck
Cruising
t >= timeToChangeLane
initiateLaneChange!
changingLane
t = 0
resumeCruising!
t = 0
t >= laneChangeDuration
laneChangeCompleted!
laneChanged
autoTruck1
Following
initiateLaneChange?
changingLane
t = 0
resumeCruising?
t = 0
laneChangeCompleted?
t >= agentLaneChangeDuration
laneChanged
autoTruck2
Following
initiateLaneChange?
changingLane
t = 0
resumeCruising?
t = 0
laneChangeCompleted?
t >= agentLaneChangeDuration
laneChanged
Watchdog
laneChangeCompleted?
t = 0
Monitoring
resumeCruising?
alarm!
t = 0
t >= timeToChangeLane + laneChangeDuration + agentLaneChangeDuration
Alarm


leadTruck
Cruising
t >= timeToChangeLane
initiateLaneChange!
changingLane
t = 0
resumeCruising!
t = 0
t >= laneChangeDuration
laneChangeCompleted!
laneChanged
autoTruck1
Following
initiateLaneChange?
changingLane
t = 0
resumeCruising?
t = 0
laneChangeCompleted?
t >= agentLaneChangeDuration
laneChanged
autoTruck2
Following
initiateLaneChange?
changingLane
t = 0
resumeCruising?
t = 0
laneChangeCompleted?
t >= agentLaneChangeDuration
laneChanged
Watchdog
laneChangeCompleted?
t = 0
Monitoring
resumeCruising?
alarm!
t = 0
t >= timeToChangeLane + laneChangeDuration + agentLaneChangeDuration
Alarm

This scenario encapsulates a platoon of trucks, comprising a lead truck and three following trucks, conducting a coordinated lane change. The lead truck initiates the lane change after a set time, prompting the following trucks to begin the same procedure. A 'Watchdog' entity monitors the operation, ready to raise an alarm if the lane change takes longer than expected. Upon completion, the lead truck signals the end of the lane change, upon which all trucks resume cruising. This system enables safe and synchronized lane changes within a truck platoon.

Truck coupling and decoupling:





In this autonomous truck platooning scenario, each truck initially operates independently until it identifies a suitable environment for platooning, such as a motorway. When it detects a potential platoon, the truck sends a request to join, which is evaluated by the existing platoon, taking into account factors such as the size of the platoon, the compatibility of the new truck, and the potential benefits and risks. If the request is approved, the truck joins the platoon, adjusting its speed and positioning to form a coordinated and efficient transport unit.

Critical to this operation is the constant communication between the trucks, which is monitored by a watchdog unit that ensures consistent data exchange and monitors for potential anomalies. If the watchdog detects a communication disruption within a truck that exceeds a certain threshold, the affected truck is removed from the platoon until the problem is resolved. In the event of a major malfunction affecting the entire platoon, the platoon disbands and each truck returns to independent operation, demonstrating the resilience and adaptability of the system.

Python Implementation:

Braking:

The Python code provided uses the asyncio library to simulate a platoon of vehicles with a leader and several followers. Each vehicle is represented by a class, with the leader switching between cruising and braking states based on specified time intervals, and the followers mimicking the current state of the leader. The simulation runs to completion with all vehicles operating simultaneously, illustrating how an autonomous vehicle platoon might operate in real-world conditions.

```python
import asyncio

class Vehicle:
    def __init__(self, name):
        self.name = name
        self.state = "Following"

    async def run(self, leader):
        while True:
            if self.state == "Following" and leader.state == "Braking":
                self.state = "Braking"
            elif self.state == "Braking" and leader.state == "Cruising":
                self.state = "Following"
            print(f"{self.name} is {self.state}")
            await asyncio.sleep(1)

class LeaderVehicle:
    def __init__(self):
        self.state = "Cruising"
        self.t = 0

    async def run(self):
        while True:
            if self.state == "Cruising":
                if self.t >= timeToBrake:
                    self.state = "Braking"
                    self.t = 0
            elif self.state == "Braking":
                if self.t >= brakingDuration:
                    self.state = "Cruising"
                    self.t = 0
            print(f"Leader is {self.state}")
            self.t += 1
            await asyncio.sleep(1)

```

Result

```
Leader is Cruising
Agent4 is Following
Agent3 is Following
Agent2 is Following
Agent1 is Following
Leader is Braking
Agent3 is Braking
Agent1 is Braking
Agent4 is Braking
Agent2 is Braking
Leader is Braking
Agent1 is Braking
Agent2 is Braking
Agent3 is Braking
Agent4 is Braking
Leader is Braking
Agent2 is Braking
Agent4 is Braking
Agent1 is Braking
Agent3 is Braking
Leader is Braking
Agent4 is Braking
Agent3 is Braking
Leader is Braking
Agent3 is Braking
Agent1 is Braking
Agent4 is Braking
Agent2 is Braking
Leader is Cruising
Agent1 is Following
Agent2 is Following
Agent3 is Following
Agent4 is Following
Leader is Cruising
Agent2 is Following
Agent4 is Following
Agent1 is Following
Agent3 is Following
```

```python

# Constants
N = 4
timeToBrake = 15
brakingDuration = 5

# Create leader and vehicles
leader = LeaderVehicle()
vehicles = [Vehicle(f"Agent{i}") for i in range(1, N+1)]

# Run the simulation
async def main():
    await asyncio.gather(
        leader.run(),
        *[vehicle.run(leader) for vehicle in vehicles]
    )

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
try:
    loop.run_until_complete(main())
finally:
    loop.close()

```

Steering:

This Python script creates a simulation of two vehicles, a leader and a follower, each represented as an instance of the Vehicle class. The leader cyclically changes its state (idle, turn left, turn right) every second and stores these states in a queue. The follower mimics the leader's states, but with a delay of 2 seconds. The script uses the asyncio library to allow the leader and follower to operate simultaneously in a non-blocking manner, providing a basic model of a delayed vehicle platooning scenario.

```python
import asyncio
from collections import deque

class Vehicle:
    def __init__(self, name):
        self.name = name
        self.state = "idle"

    async def run(self, leader):
        while True:
            # Wait for the leader to have enough history
            if len(leader.history) > delay:
                self.state = leader.history.popleft()
            print(f"{self.name} is {self.state}")
            await asyncio.sleep(1)

class LeaderVehicle(Vehicle):
    def __init__(self, name):
        super().__init__(name)
        self.history = deque()  # Store state history of leader so agent can do the same

    async def run(self):
        while True:
            if self.state == "idle":
                self.state = "turnLeft"
            elif self.state == "turnLeft":
                self.state = "turnRight"
            elif self.state == "turnRight":
                self.state = "idle"

            self.history.append(self.state)
            print(f"{self.name} is {self.state}")
            await asyncio.sleep(1)

# Create leader and vehicles
leader = LeaderVehicle("Master")
follower = Vehicle("Slave")

# Set delay
delay = 2

# Run the simulation
async def main():
    await asyncio.gather(
        leader.run(),
        follower.run(leader)
    )

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

Result

```
Master is turnRight
Slave is idle
Master is idle
Slave is turnLeft
Master is turnLeft
Slave is turnRight
Master is turnRight
Slave is idle
Master is idle
Slave is turnLeft
Master is turnLeft
Slave is turnRight
Master is turnRight
Slave is idle
Master is idle
Slave is turnLeft
Master is turnLeft
Slave is turnRight
Master is turnRight
Slave is idle
Master is idle
Slave is turnLeft
Master is turnLeft
Slave is turnRight
Master is turnRight
Slave is idle
Master is idle
Slave is turnLeft
Master is turnLeft
Slave is turnRight
Master is turnRight
Slave is idle
Master is idle
Slave is turnLeft
Master is turnLeft
Slave is turnRight
```

Lane Change:

This Python code simulates a platoon of vehicles, with a leader vehicle and several followers. The leader changes lanes after a certain time, and the followers mimic this action, but with a different delay. The state of each vehicle is monitored and updated using an asynchronous event loop from the asyncio library. After a successful lane change, the vehicles return to their initial state of following the leader, who returns to cruising mode. This code models a truck platooning system with synchronised lane changes between the platoon vehicles.

```python
import asyncio

class Vehicle:
    def __init__(self, name):
        self.name = name
        self.t = 0
        self.state = "Following"

    async def run(self, leader):
        while True:
            if self.state == "Following" and leader.state == "changingLane":
                self.state = "changingLane"
                self.t = 0
            elif self.state == "changingLane":
                if self.t >= agentLaneChangeDuration:
                    self.state = "laneChanged"
                    self.t = 0
            elif self.state == "laneChanged" and leader.state == "Cruising":
                self.state = "Following"
                self.t = 0
            print(f"{self.name} is {self.state}")
            self.t += 1
            await asyncio.sleep(1)

class LeaderVehicle:
    def __init__(self):
        self.t = 0
        self.state = "Cruising"

    async def run(self):
        while True:
            if self.state == "Cruising":
                if self.t >= timeToChangeLane:
                    self.state = "changingLane"
                    self.t = 0
            elif self.state == "changingLane":
                if self.t >= laneChangeDuration:
                    self.state = "laneChanged"
                    self.t = 0
            elif self.state == "laneChanged":
                # Assume some condition here to return to "Cruising"
                self.state = "Cruising"
                self.t = 0
            print(f"Leader is {self.state}")
            self.t += 1
            await asyncio.sleep(1)

```

```python
# Constants
N = 4
timeToChangeLane = 20
laneChangeDuration = 8
agentLaneChangeDuration = 10

# Create leader and vehicles
leader = LeaderVehicle()
vehicles = [Vehicle(f"Agent{i}") for i in range(1, N)]

# Run the simulation
async def main():
    await asyncio.gather(
        leader.run(),
        *[vehicle.run(leader) for vehicle in vehicles]
    )

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

Result

```
Leader is Cruising
Agent1 is Following
Agent3 is Following
Agent2 is Following
Leader is Cruising
Agent3 is Following
Agent1 is Following
Agent2 is Following
Leader is Cruising
Agent1 is Following
Agent3 is Following
Agent2 is Following
Leader is Cruising
Agent3 is Following
Agent1 is Following
Agent2 is Following
Leader is Cruising
Agent1 is Following
Agent3 is Following
Agent2 is Following
Leader is Cruising
Agent3 is Following
Agent1 is Following
Agent2 is Following
```

```
Leader is changingLane
Agent3 is changingLane
Agent1 is changingLane
Agent2 is changingLane
Leader is changingLane
Agent1 is changingLane
Agent3 is changingLane
Agent2 is changingLane
Leader is changingLane
Agent3 is changingLane
Agent1 is changingLane
Agent2 is changingLane
Leader is laneChanged
Agent1 is changingLane
Agent3 is changingLane
Agent2 is changingLane
Leader is Cruising
Agent3 is changingLane
Agent1 is changingLane
Agent2 is changingLane
Leader is Cruising
Agent1 is laneChanged
Agent3 is laneChanged
Agent2 is laneChanged
Leader is Cruising
Agent3 is Following
Agent1 is Following
Agent2 is Following
Leader is Cruising
Agent1 is Following
Agent3 is Following
Agent2 is Following
```