문서 이력				
Version	작성일	변경사유	작성자	변경 내용 요약
1.2	2022.11.08		고영배 외	

ADS 연동 V2X 프로토콜 검증 시뮬레이터 제작

- 용역 보고서 -

목차

1. 개요

본 문서는 ADS 연동 V2X 프로토콜 검증 시뮬레이터 용역을 수행하며 제작한 시뮬레이터의 소스코드 및 사용 설명서이다. 시뮬레이터는 전자통신연구원에서 개발한 ADS (Automated Driving System) 과 연동되어 다양한 환경에서 협력주행 동작 검증 및 알고리즘 개발에 활용된다. 이를 통해서 실제 도로에서 확인하기 전 시뮬레이터를 통해서 다양한 환경을 재현하여 협력주행 알고리즘을 검증 및 평가할 수 있다. 이를 위해 본 용역에서는 표 1.1 과 같이 총 5개의 협력주행 Class (A~E)를 정의하고, 각 Class에 대하여 미리 협의된 실제 도로 환경 (전자통신연구원 주변 도로)을 모델링하여 시뮬레이터에 반영한다.

[표 1.1] 시뮬레이터의 협력주행 시나리오 및 상황

협력주행 시나리오	협력주행 상황
Class A	우합류
Class B	끼어들기
Class C	추월
Class D	양보요청
Class E	양보요청

본 보고서의 구성은 다음과 같다. 2장에서 시뮬레이터에 적용된 협력주행 프로토콜에 대해 자세히 설명한다. 3장에서는 시뮬레이터 개발 및 소스코드에 대해 자세히 설명한다. 4장에서는 시뮬레이터의 사용 방법 자세히 기술한다.

2. V2X 협력 주행 프로토콜 및 알고리즘

본 장에서는 시뮬레이터에서 사용된 협력주행 프로토콜에 대해서 자세하게 설명한다. 프로토콜은 SAE (Society of Automotive Engineers)에서 협력주행을 위해서 J2735 (Dedicated Short Range Communications Message Set Dictionary)에서 정의한 메시지들을 기반으로 협력주행 상황에 맞도록 구성한다. 본 장의 1절은 협력주행을 위하여 사용되는 메시지에 대하여 기술하며, 2절은 각 시나리오에서 제안된 협력 주행 프로토콜에 대하여 기술하고, 3절은 각 시나리오에서 제안된 협력 주행 알고리즘에 대하여 자세히 기술한다.

2.1 협력주행 메시지

본 시뮬레이터의 협력주행 프로토콜에서 사용된 메시지는 표 2.1과 같이 총 7가지이며 각 협력주행 Class에서 협력주행을 수행하기 위해서 적절하게 활용된다. 각 메시지의 Format은 SAE J2735 문서에 정의된 형식을 이용하되, 전자통신연구원과의 협의를 통해 ADS에 맞춰서 일부 내용을 변경하여 활용한다. 협력주행 메시지는 기본적으로 Header와 Payload로 구성되며 각 메시지의 시작인 Header는 각 메시지 구분하고 메시지의 총 길이가 담기는 것을 목적으로 한다. 메시지 구분을 위하여 사전에 정의된 각 메시지의 ID를 사용하여 다른 메시지로 전송되지 않도록 각 메시지가 정의되었다. 메시지 헤더의 구성은 표 2.2와 같다.

[표 2.1] 협력주행 메시지

Message Type	주요 기능	사용 Class
BSM (Basic Safety Message)	차량 주행 정보 교환	A, B, C, D, E
PIM (Perception Information Message)	차량 주변 오브젝트 정보 교환	A
DMM (Driving Maneuver Message)	차량 주행 의도 교환	B
DNM (Driving Negotiation Message) Request	협력주행 협의 요청	B, C
DNM Response	협력주행 협의 응답	C
EDM (Emergence Driving Message)	양보 요청	D, E

[그림 2.1] Header 프레임 포맷

Header			Message
Msg_ID (1byte)	CRC-16 (2byte)	Packet Length (2byte)	

[표 2.2] Header 프레임 Field 설명

Field	변수형 및 크기	Desciption
Msg_ID	Uint8, 1byte	1 : BSM 2 : PIM 3 : DMM 4 : DNM_Req 5 : DNM_Rep 6 : DNM_Ack 7 : EDM
CRC-16	Uint16, 2byte	
Packet Length	Uint16, 2byte	Header + payload msg 크기

- BSM(Basic Safety Message)

- BSM은 차량 상태에 대한 안전 데이터를 교환하기 위해 다양한 어플리케이션에서 사용된다. 주변 차량들에게 알리기 위해 메시지로써 broadcast되며, 일반적으로 초당 10회의 전송 속도로 전달되는 것을 목표로 한다. 메시지를 전송하면서 MsgCnt와 DSecond를 업데이트될 수 있도록 동작한다.

- MsgCnt는 메시지의 시퀀스를 제공하기 위해 사용된다. 0에서 127 범위의 값으로 초기화할 수 있습니다. BSM이 전송되면서 1씩 증가하며 127 이후의 값은 0으로 초기화된다.

- TmpId는 4byte로 차량 식별을 위해 사용된다. 어떠한 차량이 BSM을 보냈는지 구분할 수 있으며, 차량 식별을 통해 메시지가 겹치는 것을 방지할 수 있다.

- Lat은 차량의 물리적 위도 값을 뜻한다. 31비트 값으로 표현되

며, 사용 중인 수평 기준과 함께 1/10 정수 microdegree로 표현된다. 값의 표현범위는 -90도에서 +90도까지 가능하다.

- Long은 차량의 물리적 경도 뜻한다. 32비트 값으로 표현되며, 사용 중인 수평 기준과 함께 1/10 정수 microdegree로 표현된다. 값의 표현범위는 -180도에서 +180도까지 가능하다.

- Speed는 차량의 속도를 뜻한다. m/s로 계산되며 정밀한 차량 속도 정보가 담긴다.

- Heading은 차량이 주행하는 방향을 뜻한다. 360도 표현을 통해 차량이 정확히 어떠한 방향으로 주행하는 알 수 있다.

							Transm&Speed (2byte)	
MsgCnt (1byte)	TmpId (4byte)	DSecond (2byte)	Lat (4byte)	Long (4byte)	Elev (2byte)	PosAccy (4byte)	Trasm (3bit)	Speed (13bit)

[그림 2.1] BSM 메시지 프레임 포맷

Heading (2byte)	Angle (1byte)	Acc (7byte)	Brake (2byte)	Size (3byte)

- PIM(Perception Information Message)

- PIM은 차량 주변의 오브젝트 정보를 전달하기 위한 메시지다. PIM이 한번 전송될 때 차량 주변의 오브젝트의 모든 정보가 담겨서 전달되는 것을 목표로 한다.

- PIM은 전송하는 차량의 정보와 차량이 인지한 오브젝트 수가 ObjNum에 채워지고 수만큼 각 오브젝트의 상세 정보를 담아

서 다른 차량에게 전달하여 다른 차량이 탐지하지 못한 오브젝트에 대하여 정보를 제공할 수 있다.

				반복		
MapOrigin (8*2 byte)	CrntLoc (4*4byte)	DstLoc (4*2byte)	ObjNum (2byte)	ObjType (1byte)	Accuracy (4byte)	ObjID (2byte)

[그림 2.2] PIM 메시지 프레임 포맷

반복				
Class (1byte)	Outlier (4*2*4byte)	Velocity (4*2byte)	TraffLgt (1byte)	ExtLgt (1byte)

- DMM(Driving Maneuver Message)

- DMM 메시지는 차량이 앞으로 할 의도를 전달하기 위한 메시지다. 차로 변경, 끼어들기, 추월할 때 앞으로 할 주행 의도를 전송하면서 실행될 곳까지의 거리를 전달하는 것을 목표로 한다.

- 실질적으로 주행 의도가 실행되는 곳은 진행중인 차와 주변 차의 속도에 따라서 변경될 수 있다.

- 각 필드의 값은 아래의 표 2.3을 참고하여 결정된다.

[표 2.3] DMM 메시지 Field 설명

Field	변수형 및 크기	Description
Maneuver Type	UInt16, 2byte	차로주행 : 차선내 직진주행(1) 차로변경 : 좌차로 변경(2), 우차로 변경(3) 교차로통과 : 직진(4), 좌회전(5), 우회전(6) 기타 : 유턴(7)
RemainDistance	UInt8, 1byte	현재 위치에서 주행의도 장소까지의 거리 단위 : m(미터)

Maneuver Type (2 byte)	RemainDistance (1 byte)

[그림 2.3] DMM 메시지 프레임 포맷

- DNM(Driving Negotiation Message)

- DNM은 차량 협력 주행을 맺기 위한 메시지다. TCP의 3-hand-shaking과 유사하게 요청, 응답, 마무리 3개의 메시지 형태가 있다. 앞차와의 충돌이 발생할 것 같을 때 협력 주행을 맺어 앞 차를 추월하는 것을 목표로 한다.

- DNM에서는 누가 누구와 협력 주행을 맺는지 중요하기 때문에 모든 DNM 메시지에서 Sender와 Receiver의 TmpID가 일치해야 한다.

- RemainDistance는 충돌 예상 지점까지의 남은 거리를 뜻하며, AgreementFlag는 협상 동의 여부를 뜻하고 0은 비동의, 1은

동의를 하고, NegoDrivingDone은 협력 주행 완료되었음을 전달한다.

Sender (4 byte)	Receiver (4 byte)	RemainDistance (1 byte)

[그림 2.4] DNM_Request 메시지 프레임 포맷

Sender (4 byte)	Receiver (4 byte)	AgmtFlag (1 byte)

[그림 2.5] DNM_Response 메시지 프레임 포맷

Sender (4 byte)	Receiver (4 byte)	Negotiation Driving Done (1 byte)

[그림 2.6] DNM_Done 메시지 프레임 포맷

- EDM(Emergence Driving Message)

- EDM은 차량 주행 양보를 전달하기 위한 메시지다. 응급한 상황에서 다른 차량에게 메시지를 전달하여 수신된 모든 차가 EDM 메시지를 받은 차에게 주행을 양보하는 것을 목표로 한다.

- 각 필드의 값은 아래의 표 2.4를 참고하여 결정된다.

[표 2.3] EDM 메시지 Field 설명

Field	변수형 및 크기	Description
Maneuver Type	UInt16, 2byte	차로변경 : (1) 교차로통과 : 직진(2), 좌회전(3), 우회전(4) 기타 : 유턴(7)
RemainDistance	UInt8, 1byte	현재 위치에서 주행의도 장소까지의 거리 단위 : m(미터)

• EDM은 양보를 요청하는 차량만 전송을 하며, 다른 차량은 수신 한다.

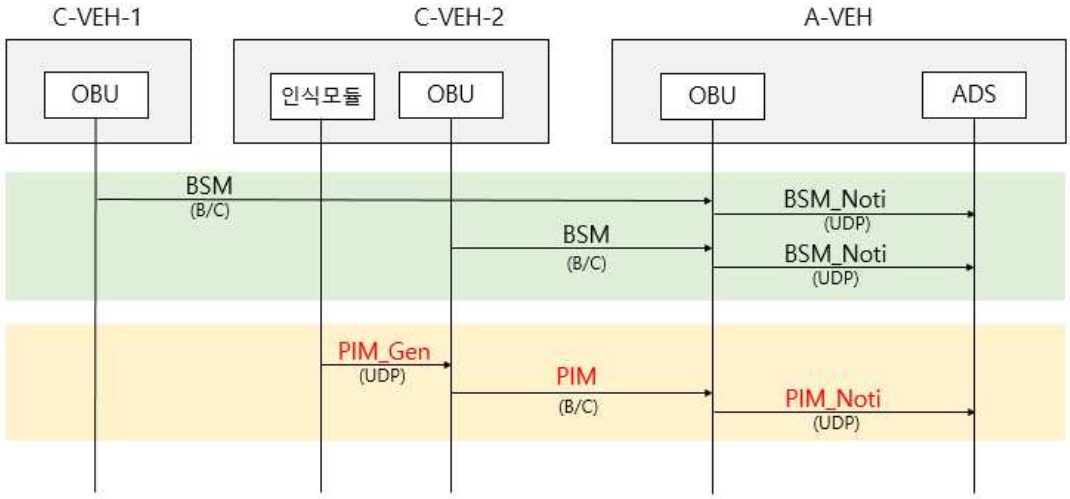
Maneuver Type (2 byte)	RemainDistance (1 byte)

[그림 2.7] EDM 메시지 프레임 포맷

2.2 협력 주행 프로토콜

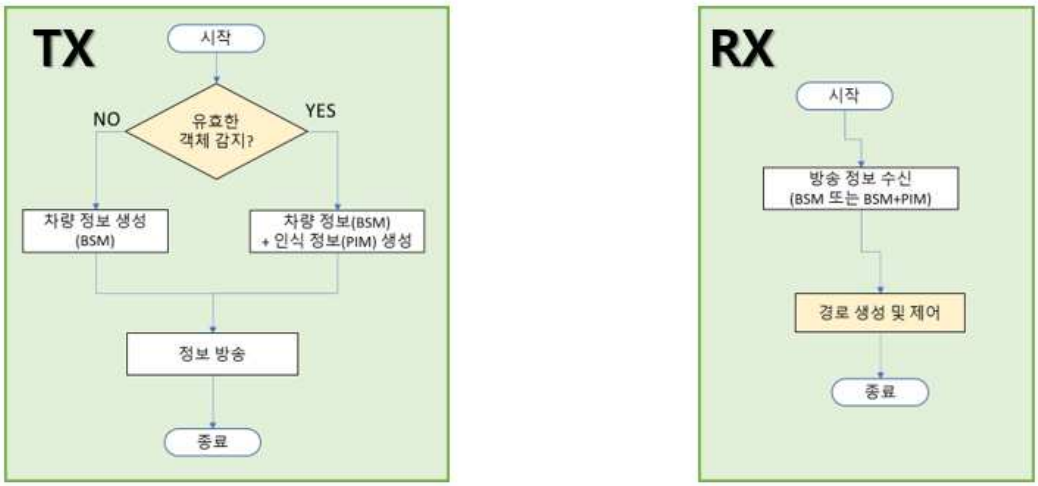
본 절에서는 순서대로 각 시나리오에서 사용되는 프로토콜에 대하여 기술한다. 2.1에서 기술된 메시지들을 사용하여 프로토콜이 동작하는 과정을 자세하게 기술한다.

그림 2.8은 Class A(이하 우합류) 시나리오 프로토콜을 나타낸다. 총 차량 3대 중 C-VEH 2대 사이로 A-VEH이 합류하려는 시나리오에서 사용되는 프로토콜이다. 모든 차량들이 BSM을 주기적으로 1초당 10번씩 broadcast하도록 하며, C-VEH(2)이 차량이 PIM에 C-VEH(1)의 정보를 인식하여 A-VEH에게 전달하는 과정을 수행하는 것을 목표로 한다.



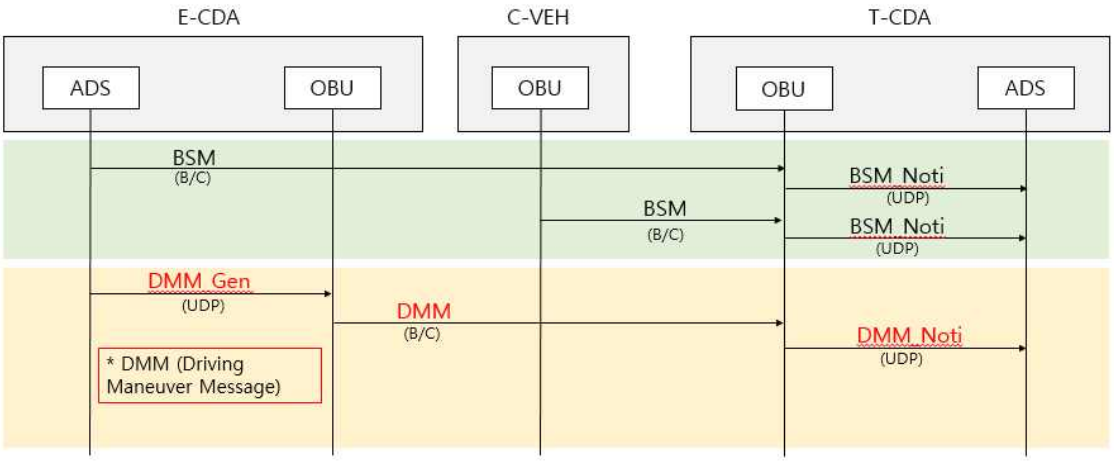
[그림 2.8] Class A (우합류) 시나리오 프로토콜

그림 2.9은 우합류 시나리오 알고리즘을 표현한다. Tx는 주행을 하면서 진행하는 차량의 주변 객체를 탐지되었을 경우, 탐지한 객체에 대해서 PIM에 정보를 담아서 BSM과 같이 생성하며, broadcast 방송을 한다. 만약 Tx가 주변 객체를 탐지되지 않은 경우, 송신 차량의 정보가 담긴 BSM만 생성해서 broadcast를 한다. Rx는 BSM이나 BSM과 PIM이 들어오는 경우에 따라 경로 생성과 차량 제어를 한다.



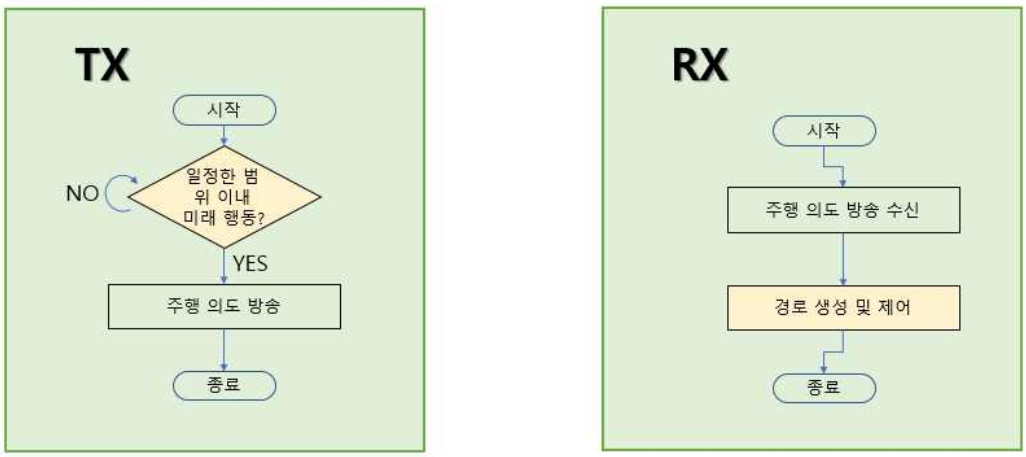
[그림 2.9] Class A (우합류) 시나리오 알고리즘

그림 2.10은 Class B(이하 끼어들기) 시나리오 프로토콜을 나타낸다. 2대의 차량이 나란히 주행하고 있을 때, 옆차로에 있는 차량 1대가 2대의 차량 사이로 끼어드는 시나리오에서 사용되는 프로토콜이다. 모든 차량들이 BSM을 주기적으로 1초당 10번씩 broadcast하도록 하며, E-CDA 차량이 T-CDA 차량에게 주행의도를 방송하고 해당 차량이 끼어들 수 있도록 T-CDA차량은 감속하는 과정을 수행하는 것을 목표로 한다.



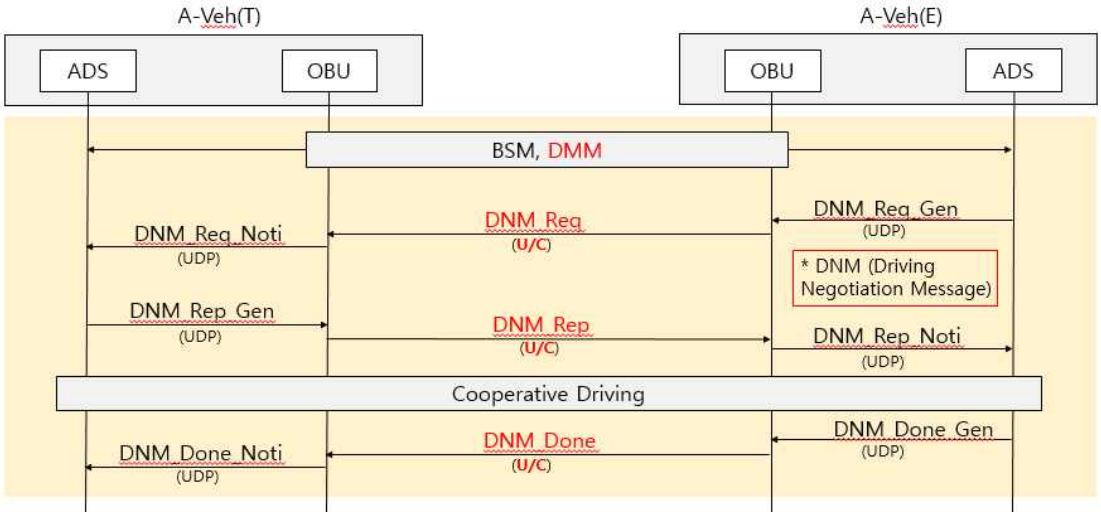
[그림 2.10] Class B (끼어들기) 시나리오 프로토콜

그림 2.11는 끼어들기 시나리오 알고리즘을 표현한다. Tx는 일정한 범위 이내 미래 행동이 있다면 주행 의도 방송을 하고, 일정한 범위 이내 미래 행동이 없다면 주행 의도는 방송을 하지 않고, BSM만 방송한다. Rx는 주행 의도 방송을 수신한 경우, Tx 차량의 주행 의도를 참고하여 경로 생성 및 차량의 속도를 제어한다.



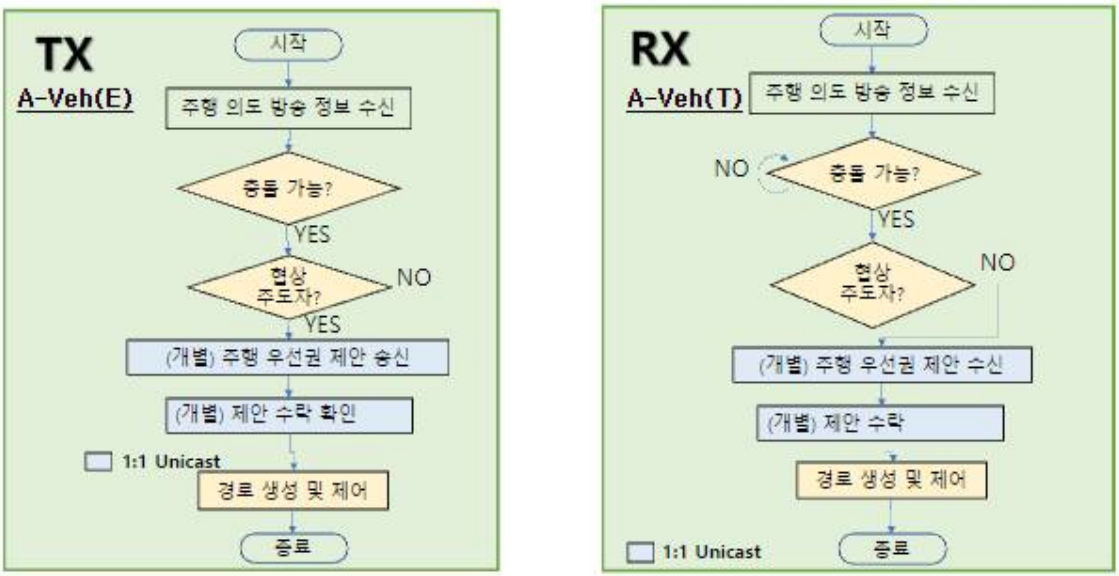
[그림 2.11] Class B (끼어들기) 시나리오 알고리즘

그림 2.12는 Class C(이하 추월) 시나리오 프로토콜을 나타낸다. ADS 차량과 시뮬레이터 차량(이하 C-CAR)이 앞뒤로 나란히 주행하고 있을 때, 뒷 차량이 앞 차량을 추월하는 시나리오에서 사용되는 프로토콜이다. 모든 차량들이 BSM을 주기적으로 1초당 10번씩 broadcast하도록 하며, ADS 차량이 C-CAR 차량에게 주행의도를 방송하여, C-CAR 차량이 ADS 차량의 주행 의도를 파악한 후, 추월을 하기 위해 협력 주행을 맺는다. 이때 DNM 메시지를 사용하여, ADS 차량이 C-CAR 차량과의 협력 주행을 맺는 과정을 통하여, 최종적으로 ADS 차량이 C-CAR 차량을 추월하는 것을 목표로 한다.



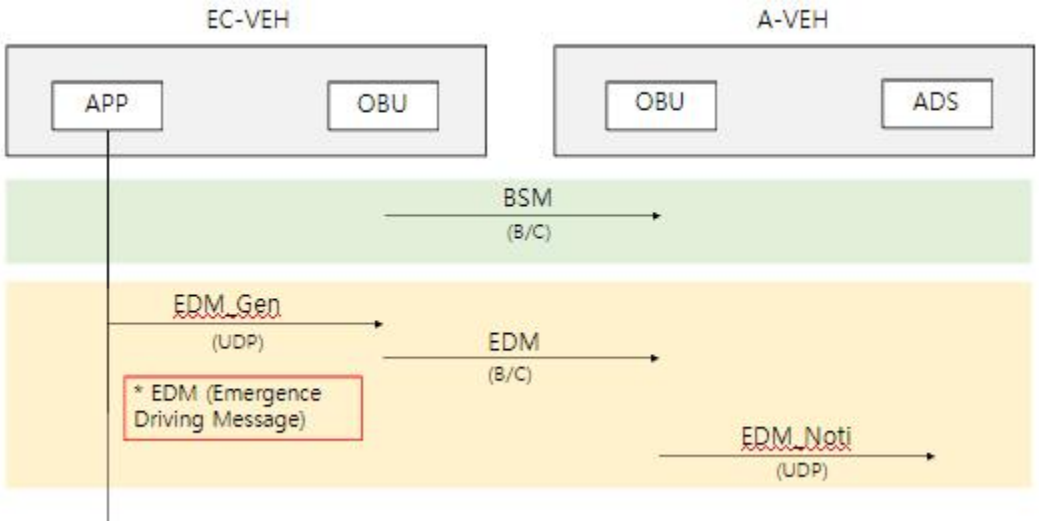
[그림 2.12] Class C (추월) 시나리오 프로토콜

그림 2.13는 추월 시나리오 알고리즘을 표현한다. TX는 끼어들기 알고리즘을 거치고, 추월하고자 하는 차량(RX)에게 DNM 메시지를 통하여 협력 주행을 제안하고, 협력 주행을 마무리하는 것까지입니다. RX는 TX로부터 들어오는 DNM 협력 요청 메시지를 수락하거나 거절하여, 두 차량 간 협력 주행을 맺도록 한다.



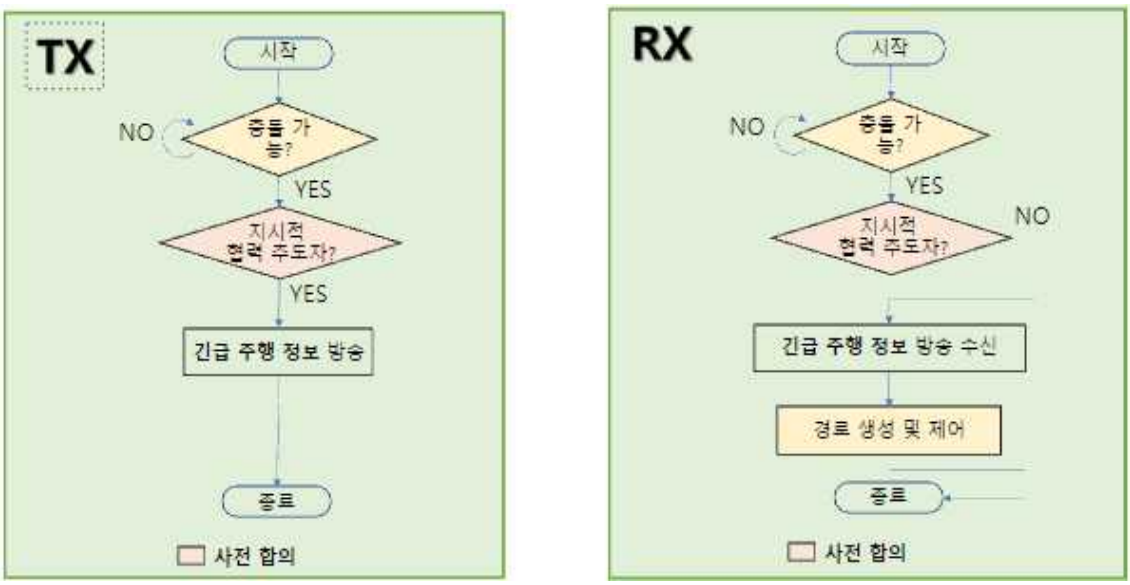
[그림 2.13] Class C (추월) 시나리오 알고리즘

그림 2.14는 Class D(이하 양보요청) 시나리오 프로토콜을 나타낸다. ADS 차량과 시뮬레이터 차량(이하 긴급차량)이 우합류하거나 삼거리에서 따로 주행하고 있을 때, 긴급차량이 계속 EDM 메시지를 전송하여, EDM 메시지를 수신받는 차량이 주행을 멈추게 된다.



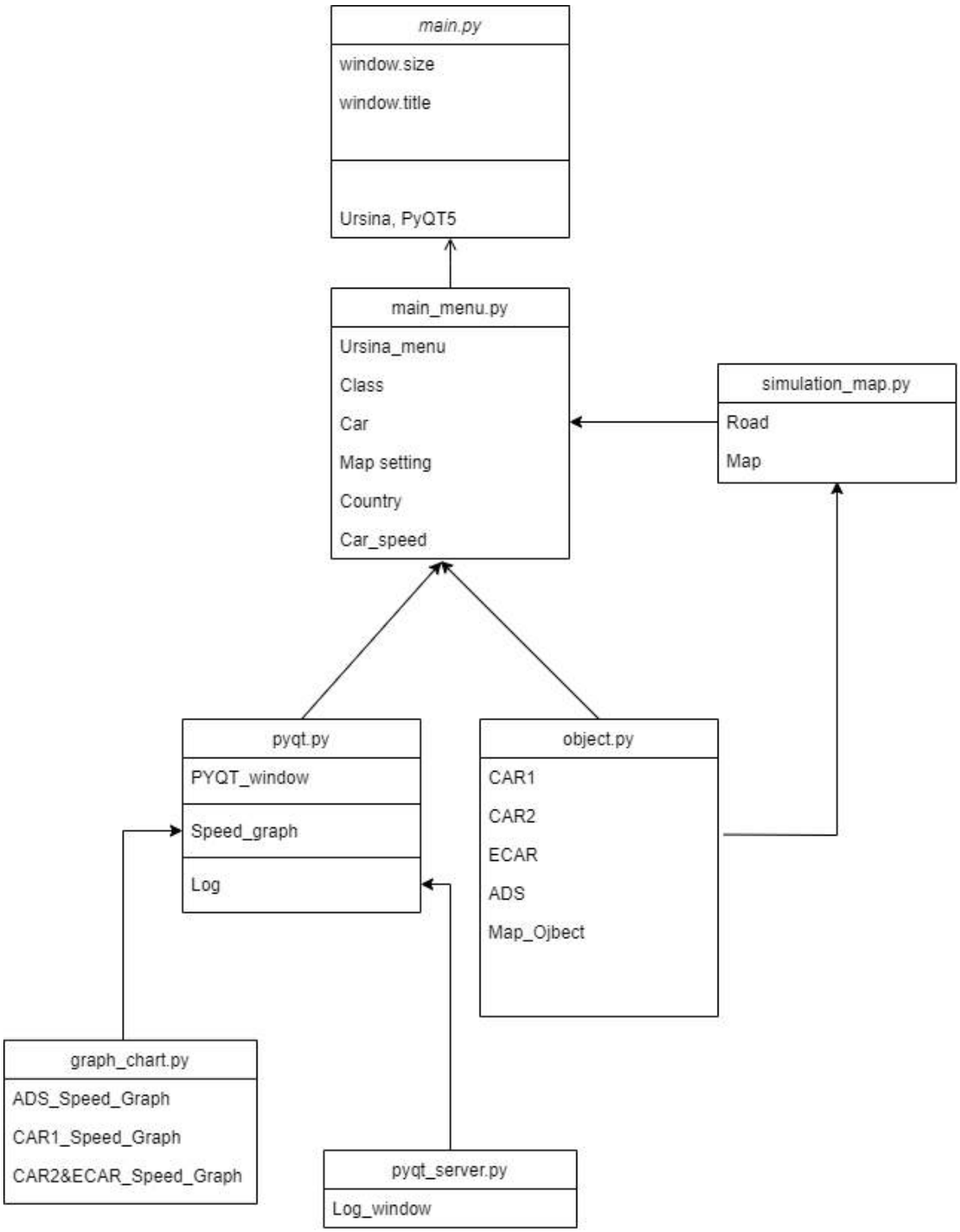
[그림 2.14] Class D (양보요청) 시나리오 프로토콜

그림 2.15은 양보요청 시나리오 알고리즘을 표현한다. 긴급차량이 Tx가 되어, 계속하여 EDM 메시지를 발송하고, RX는 EDM 메시지가 수신되었을 경우, 경로를 새롭게 생성하거나 주행을 멈추게 한다.



[그림 2.15] Class D (양보요청) 시나리오 알고리즘

3. 소스코드 설명



[그림 3.1] 프로그램 구성도

본 시뮬레이터는 Python의 Ursina Engine 라이브러리를 기반으로 GUI (Graphic User Interface)를 제공한다. Ursina Engine은 보통 게임 개발에 많이 활용되며 쉽고, 자유도가 높으며, 오픈 소스라는 장점이 있다. 본 시뮬레이터는 이러한 게임엔진을 이용하여 도로환경과 자동차의 움직임을 구현했다.

그림 3.1은 시뮬레이터 소스코드의 프로그램 구성도이다. 소스코드의 각 파일의 개요는 다음과 같다.

- main.py : 프로그램 창의 이름과 크기를 결정할 수 있는 코드파일이다. 프로그램의 시작이 되는 부분이다.

- object.py : 프로그램에서 사용되는 object들이 정의된 코드파일이다. 차량, 도로, 맵을 구성하기 위해 필요한 텍스쳐, 이미지, 사이즈 등 원하는 대로 프로그램의 오브젝트를 설정 가능하다. 각 오브젝트의 기본적으로 오브젝트를 자세히 구현 가능하다.

- simulation_map.py : 맵을 만들어 놓은 코드 파일이다. object.py에서 정의된 object를 가져와서 위치와 방향들을 설정하여 map을 구성한다.

- main_menu.py : 프로그램의 부가적인 기능과 설정이 진행되는 부분이다. 원하는 시뮬레이터 상황을 고르고, 해당위치에서 시작되게 할 수 있고, 프로그램의 카메라 각도, 차량의 속도 변경을 위해 작성된 코드파일이다.

- pyqt.py : log와 스피드를 표현하기 위해 사용되는 다른 프로그

램 창이다. log는 pyqt_server.py와 스피드는 graph_chart.py와 연결되어 있다.

- pyqt_server.py : log창에서 log가 버튼을 클릭함으로써 기록되게하고 기능을 멈추게 할 수 있다.

- graph_chart.py : 각 차량들의 속도를 각기 다른 그래프로 속도를 표현하도록 할 수 있다. 또한 계속해서 갱신되도록 pyqt.py에서 설정한다.

3.1 시뮬레이터 구조

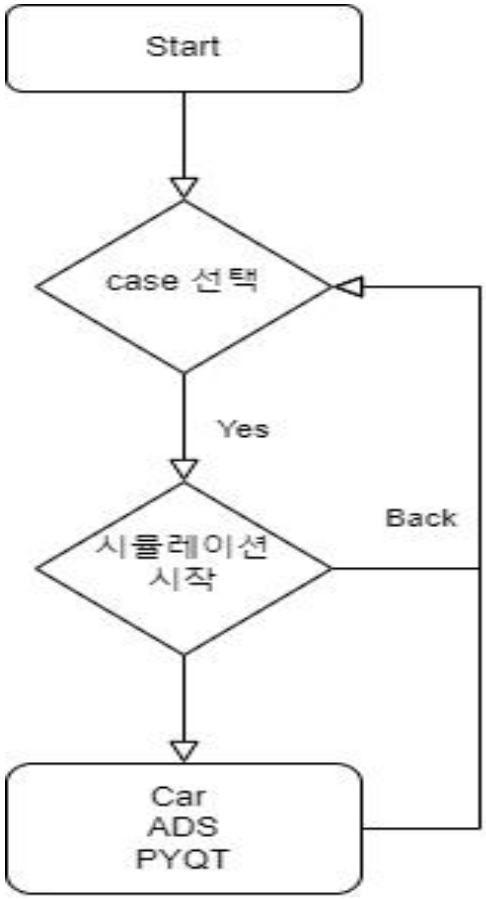
본 절에서는 시뮬레이터 소스코드의 전체적인 구조 및 동작 방식을 간략하게 설명하고, 각 기능에 대하여 모듈로 나누어 자세하게 설명한다.

● 시뮬레이터 동작 방식 개요

1) 시뮬레이터 시작 후, MainMenu에서 만들어진 버튼들로 case를 선택

2) 선택된 case를 통해 반영되는 변수를 사용하여 시뮬레이터가 시작

3) 해당 case에서 필요한 object가 작동



[그림 3.2] simulation flow

● 기능별 모듈 구분

1) ADS 모듈: 소켓 통신을 통해 데이터를 수신하는 ADS 클래스와 수신된 데이터를 parsing하여 사용하는 ADS_car 클래스가 있다.

- ADS 클래스 : ADS 차량으로부터 수신하려는 데이터를 소켓으로 받기 위해 UDP 소켓으로 데이터를 수신
- ADS_car 클래스 : UDP 소켓으로 수신된 데이터를 parsing하여 시뮬레이터 GUI에 보이도록 표시

2) CAR 모듈: 각 case에서 사용되는 차량들이 주행할 수 있도록 설정되어 있고, BSM과 사용되는 메시지들이 정의되어 있다.

[표 3.1] CAR class 정리

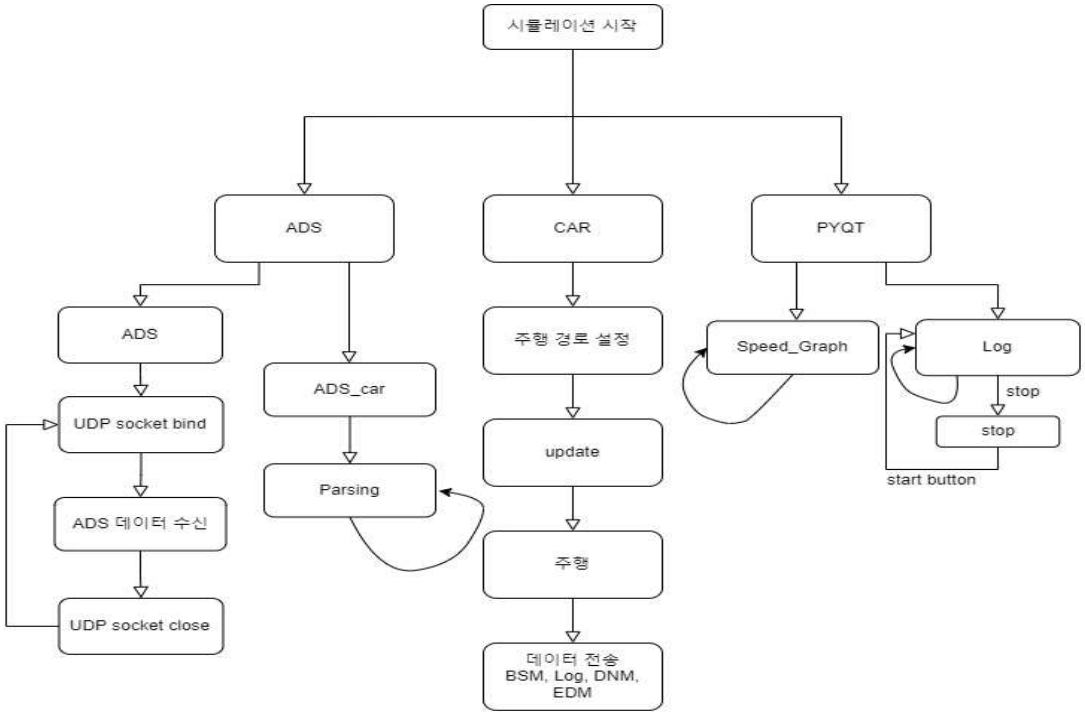
Class	적용 시나리오	메시지	PYQT Log 전송
Car1	우합류, 끼어들기	BSM	o
Car2	우합류, 끼어들기, 추월	BSM, DNM	o
E_car	양보요청	BSM, EDM	o

3) PYQT 모듈: 차량의 스피드를 그리고, Log 데이터가 들어오는 데로 Log창에 표시한다.

❑ Speed_Graph : 총 4 종류의 차량(ADS, Car1, Car2, E_car)에 대해서 그래프 3개로 속도를 표현
- 맨 위 그래프 : ADS 속도 정보
- 가운데 그래프 : Car1 속도 정보
- 아래 그래프 : Car2 or E_car 속도 정보

우합류, 끼어들기, 추월 시나리오에서는 Car2가 양보요청 시나리오에서는 E_car 속도 정보가 반영

❑ Log : 시뮬레이터에서 parsing되는 모든 데이터를 요약해서 PYQT로 전송된 메시지를 Log창에 투영
- 켜지면서, 자동으로 들어오는 데이터를 Log창에 투영
- 버튼을 누를 경우, Stop한 상태에서 들어오는 데이터를 투영하지 않는다.
- 다시 버튼을 누를 경우, 자동으로 들어오는 데이터를 Log창에 투영



[그림 3.3] simulation flow

3.2 세부 모듈 설명

1) CAR : 시뮬레이터 시작 이후, 사용되는 모든 차량 오브젝트들은 각 정의된 class를 통해 움직인다.

- 먼저 메인 흐름에서 각 case를 선택할 때, 다른 case 변수의 값 지정
- case 변수를 차량 오브젝트의 파라미터로 사용하여 차량 오브젝트 enable
- 각 Car class에서 case 파라미터 값을 통해 주행 경로 설정 함수 실행

※ case A, B는 Car1, Car2 작동, C는 Car2만 작동, D는 E_car만 작동

- 자동실행되는 update를 통해, 시간차와 속도를 이용하여 동적인 주행을 구현
- 0.1초마다 bms이 전송되고, 다른 case에 쓰이는 차량에 case에

서 사용되는 메시지가 전송되도록 구현

```
if self.case == 1 or self.case == 2:
    self.car1 = Car1(case = self.case, speed= self.car1_speed)
    self.car2 = Car2(case=self.case, speed= self.car2_speed)
elif self.case == 3:
    self.car2 = Car2(case=self.case, speed= self.car2_speed)
    self.car1_speed = 0
else:
    self.e_car = E_car(case = self.case, speed= self.car2_speed)
    self.e_car_speed = self.car2_speed
    self.car1_speed = 0
```

[그림 3.3] 차량 object enable in main_meny.py

```
class Car1(Entity):

    def __init__(self, case, speed):
        super().__init__(...
        self.case = case
        self.__speed = speed
        self.result = []
        # 경로 선택
        if self.case == 1:
            self.case_a()
            self.car1_ri = 70
        elif self.case == 2:
            self.case_b()
            self.car1_ri = 50
```

[그림 3.4] case에 맞는
주행 설정

```
def case_a(self):
    with open("convert_a.txt", "r") as fp:
        j = 0
        ri = 0
        for i in fp.readlines():
            tmp = i.split(",")
            tmp[2] = tmp[2].replace("\n", "")
            if j == 0:
                self.result.append([float(tmp[0]) / 10000000, float(tmp[1])\
                                    / 10000000, float(tmp[2]), 0.0])
                ri += 1
                j += 1
            else:
                if 1 <= ri:
                    first = (float(self.result[ri - 1][0]), float(self.result[ri - 1][1]))
                    second = (float(tmp[0]) / 10000000, float(tmp[1]) / 10000000)
                    c = haversine(first, second, unit=Unit.METERS)
                    self.result.append([float(tmp[0]) / 10000000, float(tmp[1])\
                                        / 10000000, float(tmp[2]), c])
                    ri += 1
```

[그림 3.5] self.result에 리스트 형태로 파일 데이터를
읽어서 변환

❑ 주행

- main_menu.py에서 시뮬레이터 시작 버튼을 눌렀을 때, 해당 case에서 차량 오브젝트에 파라미터로 속도와 case 변수 값이 들어간다.
- 들어온 case 변수 값을 통하여, 어떤 케이스인지 만약 case의 값이 1이라면 case_a를 실행
- 우합류 주행 경로를 self.result에 리스트로 만들어 넣는다.

```
def update(self):
    self.key_control()
    self.car1_tick2 = self.car1_tick1
    self.car1_tick1 = dt.datetime.now().strftime('%S%f')[:-3]
    self.car1_tick = (float(self.car1_tick1) - float(self.car1_tick2))
    if self.car1_tick < 0:
        self.car1_tick += 60000
    self.timer += self.car1_tick
    self.car_moving()
```

[그림 3.6] update 함수

- 기본 설정이 마무리 되었다면, update함수가 자동적으로 실행
- update가 돌아가는 시간의 간격의 tick 구하여 해당 시간 만큼 움직이도록 구현

```
# 시작점
start_dot = [self.result[self.car1_ri][0], self.result[self.car1_ri][1]]
# 다음점
next_dot = [self.result[self.car1_ri + 1][0], self.result[self.car1_ri + 1][1]]
# 점 사이의 거리
d2d_distance = haversine(start_dot, next_dot, unit=Unit.METERS)
# 차량의 이동 거리
if self.state == 0:
    car1_md = 0
else:
    car1_md = float(self.__speed / 3600 * self.car1_tick)
# 남은 거리
car1_rd = car1_md - d2d_distance + self.car1_rd
```

[그림 3.7] car_moving 함수

- car_moving()함수를 통하여 차량이 실제로 이동한 후의 좌표로 변환
- self.car1_ri로 시작 위치와 다음 찍힌 점 사이의 거리(d2d_distance)를 구한 후, 차량의 이동 거리(car1_md) 계산

```
car1_rd = car1_md - d2d_distance + self.car1_rd
line_check = self.car1_ri
while car1_rd > 0:
    self.car1_ri += 1
    # out of range 방지
    if self.case == 1 and self.car1_ri == len(self.result) - 1:
        self.car1_ri = 70
        self.car1_rd = 0
    elif self.case == 2 and self.car1_ri == len(self.result) - 1:
        self.car1_ri = 50
        self.car1_rd = 0
    else:
        start_dot = [self.result[self.car1_ri][0], self.result[self.car1_ri][1]]
        next_dot = [self.result[self.car1_ri + 1][0], self.result[self.car1_ri + 1][1]
        self.new_p2p_distance = haversine(start_dot, next_dot, unit=Unit.METERS)
        car1_md = car1_rd
        car1_rd -= self.new_p2p_distance
        self.ch_while = 1
```

[그림 3.8] 남은 거리 > 0

```
    if car1_rd <= 0:
        dx = next_dot[0] - start_dot[0]
        dy = next_dot[1] - start_dot[1]
        if self.ch_while == 0:
            car1_x = (car1_md + self.car1_rd) * dx / d2d_distance
            car1_y = (car1_md + self.car1_rd) * dy / d2d_distance
        elif self.ch_while == 1:
            car1_x = car1_md * dx / self.new_p2p_distance
            car1_y = car1_md * dy / self.new_p2p_distance
        wgs_x = start_dot[0] + car1_x
        wgs_y = start_dot[1] + car1_y
        x = float((geo.wgs842tm(wgs_x, wgs_y)[0] - 232804.0) / 50)
        y = -18.95
        z = float((((geo.wgs842tm(wgs_x, wgs_y)[1]) - 420248) / 50) - 1.71)
        self.position = (x, y, z)
        if line_check == self.car1_ri:
            self.car1_rd += car1_md
        else:
            self.car1_rd = car1_md
        self.rotation_y = self.result[self.car1_ri][2]
```

[그림 3.9] 남은 거리 <= 0

- 만약 car1_md - d2d_distance를 뺐을 때 남은 거리(ca1_rd)가, 0보다 작으면, 아직 두 점 사이에 있는 것으로 여기고, 직선 위에 해당 car1_md만큼의 이동된 좌표를 구해서 차량 좌표로 변환
- 만약 car1_rd가 0보다 크다면, car1_md가 d2d_distance보다 커서 다음 그 좌표 사이에 있을 가능성을 생각하여, self.car1_ri를 1을 더하여 다시 위의 방식으로 계산
- car1_rd가 0보다 작아질 때 까지, 만약 self.car1_ri가 self.result의 범위를 벗어난 경우, self.car1_ri는 처음 정의된 값으로 되어 초기 주행 시작 위치로 차량의 위치 초기화
- car1_rd는 car1_md와 d2d_distance 계산으로 나오는 값
- self.car1_rd는 현재 위치를 정확하게 해주기 위해 사용하는 변수
- 그래서 거리 계산할 때, self.car1_rd를 더한다.
- 맨 처음 계산 시에는 self.car1_rd는 0이므로 영향을 주지 않는다.

❑ 메시지 처리

① BSM : ADS를 제외한 모든 차량에서 전송하는 과정은 다음과 같은 과정으로 진행되었다.

※ BSM의 모든 데이터는 SAE J2735 문서를 참고

[표 3.2] BSM 변수 정리

변수	값	설명
bsm_lat	정수	시뮬레이터 차량 latitude * 10000000 정수 값 [시뮬레이터 차량 position x값 * 50(시뮬레이터 배율) + 232804(TM좌표)] 값을 WGS로 변환 후 * 10000000(소수점 없애기)
bsm_long	정수	시뮬레이터 차량 longitude * 10000000 정수 값 [{시뮬레이터 차량 position z값 + 오차} * 50(시뮬레이터 배율) + 420248(TM좌표)] 값을 WGS로 변환 후 * 10000000(소수점 없애기)
bsm_Trans_Speed	정수	3bit 변속기 + 13bit 속도(단위 m/s) [차량의 속도 값(단위 km/h) / 3.6(km/h -> m/s) / 0.02(단위변환)]의 정수 값
bsm_Heading	정수	차량의 heading 값 [차량의 rotation_y 값 / 0.0125(단위변환)]의 정수 값

```
# LAT, LONG 변환
self.x1 = (x * 50) + 232804
self.z1 = (z + 1.02 + 0.69) * 50 + 420248
self.lat_car1 = int(round(geo.tm2wgs84(self.x1, self.z1)[0], 7) * 10000000)
self.long_car1 = int(round(geo.tm2wgs84(self.x1, self.z1)[1], 7) * 10000000)
self.bsm_lat1 = self.lat_car1.to_bytes(4, byteorder="big", signed=True)
self.bsm_long1 = self.long_car1.to_bytes(4, byteorder="big", signed=True)
self.bsm_ELev = (0).to_bytes(2, byteorder="big", signed=True)
self.bsm_PosAccy = (0).to_bytes(4, byteorder="big", signed=True)
self.msg_speed_car1 = self.__speed / 3.6 / 0.02
self.trans_speed_car1 = self.trans + int(self.msg_speed_car1)
self.bsm_Trans_Speed1 = (self.trans_speed_car1).to_bytes(2, byteorder="big", signed=True)
self.msg_heading_car1 = int(self.rotation_y / 0.0125)
self.bsm_Heading1 = (self.msg_heading_car1).to_bytes(2, byteorder="big", signed=True)
```

[그림 3.10] BSM 데이터 변환

② DMM : 끼어들기 시나리오에서 ADS가 DMM 메시지를 전송하면, Car2의 속도를 10km/h 감속한다.

[표 3.3] DMM 변수 정리

변수	값	설명
dmm_recv (전역 변수)	정수	ADS에서 DMM이 수신되었을 때, 해당 변수의 값이 1로 바뀌게 되고, Car2에서 dmm_recv의 값이 1이 되었다면 현재 속도에서 10km/h 감속한 후, 값을 0으로 변경
__speed	정수	Car2 차량의 속도 값을 나타내는 변수 main_menu.py에서도 접근하여 변수 값 변경

```
if dmm_msg[4] == 2 or msg[3] == 3:
    if self.case == 2 and self.speed_ch == 0:  # 속도가 바뀌지 않았을때만 작동하도록
        # 차량 속도 변환 -10km/h
        print("차량의 속도가 변경됩니다.")
        dmm_recv = 1
        self.speed_ch = 1
```

[그림 3.11] ADS 파싱 과정에서 dmm_recv 값이 1로 변경

```
global ads_tmp_id, dmm_recv
if dmm_recv == 1:
    self.car2_fir_sp = self.__speed
    self.__speed -= 10
    print(self.car2_fir_sp, " >>>> ", self.__speed)
    msg = "[CAR1,2 >> ADS] MSG_TYPE : DMM, 차량의 속도 변경\nCAR2 : " + str(
        self.car2_fir_sp) + ">>> " + str(self.__speed)
    self.qt_log_server.sendto(msg.encode(), self.qt_iport)
    dmm_recv = 0
```

[그림 3.12] Car2 속도 감속

③ DNM : ADS로부터 협력 주행 메시지가 수신되었을 때, 협력 주행을 동의하는 메시지를 전송하고, 완료 메시지가 수신되는 과정을 Log로 기록한다.

[표 3.4] DNM 변수 정리

변수	값	설명
ads_tmp_id	바이트	들어온 DNM 메시지가 Request이면 첫 바이트 값이 1, Done이면 첫 바이트 값이 0.
Sender	정수	시뮬레이터 차량의 Tmp_id
Receiver	정수	ADS 차량의 Tmp_id (수신된 DNM에서 ads_tmp_id를 이용)

```
# DNM Req메시지
if msg_id[0] == 4:
    self.dnm_msg = struct.unpack('<B H H I I B', self.msg)
    ads_tmp_id = struct.pack('>B I', 1, self.dnm_msg[3])
    msg = "[ADS -> Car2] MSG_TYPE : DNM Req"
    self.qt_log_server.sendto(msg.encode(), self.qt_iport)
# DNM Done메시지
if msg_id[0] == 6:
    try:
        msg = "[ADS -> Car2] MSG_TYPE : DNM Ack, 협력주행 끝"
        self.qt_log_server.sendto(msg.encode(), self.qt_iport)
        ads_tmp_id = struct.pack('>B I', 0, self.dnm_msg[3])
    except:
        pass
```

[그림 3.13] DNM 협력 주행 동의 및 완료 메시지 parsing

```
def DNM(self):
    DNM_msg = struct.unpack('>B I', ads_tmp_id)
    if DNM_msg[0] == 1:
        self.DNM_ID = (5).to_bytes(1, byteorder="big", signed=True)
        self.DNM_CRC = (0).to_bytes(2, byteorder="big", signed=True)
        self.DNM_PL = (14).to_bytes(2, byteorder="big", signed=True)
        self.Sender = (2).to_bytes(4, byteorder="big", signed=True)
        self.Receiver = (DNM_msg[1]).to_bytes(4, byteorder="big", signed=True)
        # self.Receiver = (5678).to_bytes(4, byteorder="big", signed=True)
        self.Agreement = (1).to_bytes(1, byteorder="big", signed=True)
        self.DNM_Rep = (self.DNM_ID + self.DNM_CRC + self.DNM_PL + self.Sender + self.Receiver + self.Agreement)
        self.client.sendto(self.DNM_Rep, self.client_addr)
        msg = "[CAR2 >> ADS] MSG_TYPE : DNM Rep, 협력 주행 동의하였습니다. "
        self.qt_log_server.sendto(msg.encode(), self.qt_iport)
    elif DNM_msg[0] == 0 and self.dnm_done_check < 5:
        msg = "협력 주행이 마무리되어 DNM_reponse를 보내지 않습니다."
        self.qt_log_server.sendto(msg.encode(), self.qt_iport)
        self.dnm_done_check += 1
```

[그림 3.13] DNM 협력 주행 동의 메시지 전송

- ADS로부터 DNM Request 메시지가 오게 되면, ads_tmp_id의 첫 바이트 값이 1로 하고, Done 메시지가 오게 되면 0으로 설정
- ads_tmp_id의 나머지 4바이트는 Request에 있는 ADS Tmp_ID를 가져온다.
- 첫 바이트의 값이 1이면, Car2는 ADS에게 DNM Response 메시지로 협력 주행 동의하는 의사를 전달
- 협력 주행이 끝나고 Done 메시지가 수신되면, ads_tmp_id의 첫 바이트 값이 0으로 바뀐다.

- ads_tmp_id의 첫 바이트가 0인 것을 통해, Response 메시지는 전송되지 않고, 협력 주행이 마무리 된 Log를 PYQT로 전송

④ EDM : E_car 차량이 주행을 하면서 계속 전송하는 메시지이다.

[표 3.4] EDM 변수 정리

변수	값	설명
temp_id	정수	E_car의 Tmp_id
edm_heading	정수	E_car의 주행 의도
re_dis	정수	목적지까지의 남은 거리

```
if self.re_dis >= 0:
    edm_heading = 2
    try:
        if self.result[self.ri][2] - self.result[self.ri + 10][2] >= 2.5:
            edm_heading = 4
        elif self.result[self.ri][2] - self.result[self.ri + 10][2] <= -2.5:
            edm_heading = 3
    except:
        pass
    edm_msg = struct.pack('>B H H I H B', 7, 0, 12, self.temp_id, edm_heading, int(self.re_dis))
    self.client.sendto(edm_msg, self.client_addr)
    self.ui_client.sendto(edm_msg, self.ui_port)
    msg = "[E_CAR >> ADS] MSG_TYPE : EDM, 남은 주행 거리 = " + str(self.re_dis) + "m"
    self.qt_log_server.sendto(msg.encode(), self.qt_iport)
```

[그림 3.14] EDM 메시지 전송

- re_dis는 목적지까지의 주행거리를 나타내는 변수
- 주행하면서 실질적으로 남아있는 거리가 계속해서 갱신
- edm_heading은 주행의도를 전달하는 변수
- 경로를 할당하면서, 처음부터 끝까지의 거리를 계산하고 헤딩값을 비교
- 주행하면서 5번 뒤에 있는 경로 데이터의 헤딩 값과 현재 헤딩 값을 비교
- 차이가 2.5보다 크거나 같으면, 우회전(4) 값

- 차이가 -2.5보다 작거나 같으면, 좌회전(3) 값

2) ADS : 시뮬레이터 시작 이후, ADS로부터 수신되는 데이터를 수신하고 parsing하여, GUI에 보이도록 한다.
- ADS 클래스 : 데이터를 수신하기 위해 소켓을 관리
- ADS_car 클래스 : 수신된 데이터를 parsing하여 GUI에 반영

❑ ADS

```
class ADS(Entity):
    def __init__(self):
        super().__init__()

    def update(self):
        global msg
        self.socket = socket(AF_INET, SOCK_DGRAM)
        self.socket.bind(("192.168.1.202", 31252))
        ready = select.select([self.socket], [], [], 0.02)
        if ready[0]:
            msg, addr = self.socket.recvfrom(1024)
            print("receive")

        self.socket.close()
```

[그림 3.15] 데이터 수신을 위한 ADS class

- ADS와의 UDP 소켓 통신을 통하여, 데이터를 수신
- 통신 시, 시뮬레이터가 server로 작동하여 bind가 필수
- Thread 방식으로 update가 작동하지 않아 timeout 설정
- ready 변수에서 0.02초 동안 데이터가 수신되지 않으면 timeout으로 통과
- 소켓의 데이터가 쌓이는 것을 방지하기 위해 소켓 close
- update는 자동으로 계속 호출되는 함수
- 계속 소켓이 만들어지고 수신되고 close하는 과정이 반복
- 수신되는 데이터는 전역 변수인 msg에 저장

❑ ADS_car

```
class ADS_car(Entity):
    def __init__(self, case):
        super().__init__(...)
        self.case = case
        self.qt_log_server = socket(AF_INET, SOCK_DGRAM)
        self.qt_iport = ('localhost', 55555)
        self.dnm_msg = 0
        self.__speed = 0
```

[그림 3.16] ADS_car class 기본 설정

- case 파라미터를 받는 이유는 case b, c 모두 DMM 메시지가 사용되지만, b에서만 속도가 감속되도록 하기 위해
- BSM이 수신되기 전까지 속도는 0으로 설정
- 테스트 도중 DNM Done 메시지가 먼저 발생하게 되어 오류가 발생되는 것을 해결하기 위하여, self.dnm_msg에 더미 값 저장

```
def update(self):
    global dmm_recv, msg, ads_tmp_id
    self.msg = msg
    if type(self.msg).__name__ == "bytes":
        msg_id = struct.unpack('>b', self.msg[:1])
        # BSM 메시지
        if msg_id[0] == 1:...
        # DMM
        if msg_id[0] == 3:...
        # DNM Req메시지
        if msg_id[0] == 4:...
        # DNM Done메시지
        if msg_id[0] == 6:...
```

[그림 3.17] ADS_car update 함수

- 전역 변수로 지정된 msg의 데이터를 통하여, 소켓으로 수신된 byte 데이터이면 parsing을 시작
- 맨 처음 msg의 header부분의 Msg_id 부분 1바이트를 parsing

```
# BSM 메시지
if msg_id[0] == 1:
    self.ads_bsm = struct.unpack('<B H H B I H i i H i h h b', self.msg[0:31])
    if self.ads_bsm[10] >= 8192:
        de1_Trans_Speed = bin(self.ads_bsm[10] >> 13)
        int_de1_Trans = int(de1_Trans_Speed)
        de2_Trans_Speed = bin(int_de1_Trans << 13)
        self.__speed = (self.ads_bsm[10] - int(de2_Trans_Speed, 2)) * 0.02
    else:
        self.__speed = self.ads_bsm[10] * 0.02 * 3.6
    self.position = (
        (geo.wgs842tm(self.ads_bsm[6], self.ads_bsm[7])[0] - 232804.0) / 50, -18.95,
        (((geo.wgs842tm(self.ads_bsm[6], self.ads_bsm[7])[1]) - 420248) / 50) - 1.02 - 0.69)
    self.rotation_y = self.ads_bsm[11] * 0.0125
    msg = "[ADS >> SIMULATOR] MSG_TYPE : BSM, LAT : " + str(self.ads_bsm[6]) + ", LONG : " + str(
        self.ads_bsm[7]) + ", HEADING : " + str(self.rotation_y) + ", SPEED : " + str(self.__speed)
    self.qt_log_server.sendto(msg.encode(), self.qt_iport)
```

[그림 3.18] BSM parsing

① BSM

- ADS 차량의 속도, 헤딩, 위치만 parsing하여, 시뮬레이션에 맞춰서 변형
- 속도에 경우, 변속기 값이 있는 경우와 없는 2 가지 경우 중 하나로 변환
- 헤딩 값은 parsing 후 J2735 문서의 있는 단위로 변환
- 위치 값은 WGS 좌표계 값을 TM 좌표계로 변환하고 시뮬레이터 좌표에 맞춰 변환
- 들어온 데이터를 PYQT에 정리하여 전송

```
# DMM
if msg_id[0] == 3:
    dmm_msg = struct.unpack('<B H H I H B', self.msg)
    # DMM 메시지의 파싱이 시작되었다는 표시
    print("DMM is arrived")
    # ADS가 payload에 추가한 ads Tmp_id
    self.ads_id = dmm_msg[3]
    # 좌차로 변경 or 우차로 변경 메시지가 수신되면 car2의 차량이 속도를 늦춥니다.
    if dmm_msg[4] == 1:...
    elif dmm_msg[4] == 2:...
    elif dmm_msg[4] == 3:...
    elif dmm_msg[4] == 4:...
    elif dmm_msg[4] == 5:...
    elif dmm_msg[4] == 6:...
    elif dmm_msg[4] == 7:...

    if dmm_msg[4] == 2 or msg[3] == 3:
        if self.case == 2 and self.speed_ch == 0:  # 속도가 바뀌지 않았을때만 작동하도록
            # 차량 속도 변환 -10km/h
            print("차량의 속도가 변경됩니다.")
            dmm_recv = 1
            self.speed_ch = 1
```

[그림 3.19] DMM parsing

② DMM

- ADS의 주행 의도 데이터가 들어있어, 값에 따른 parsing을 진행
- 어떠한 값이 들어왔는지 PYQT로 Log 전송
- case b에서 ADS 차량의 주행 의도가 좌차로 변경 또는 우차로 변경일 때, 속도 감속
- update로 계속 반복하는 과정에서 감속이 한번 되도록 self.speed_ch의 값이 0이 아니라면 감속하지 않는다.
- DMM의 감속과정은 위에 Car에서 DMM 부분을 찾기

```
# DNM Req메시지
if msg_id[0] == 4:
    self.dnm_msg = struct.unpack('<B H H I I B', self.msg)
    ads_tmp_id = struct.pack('>B I', 1, self.dnm_msg[3])
    msg = "[ADS -> Car2] MSG_TYPE : DNM Req"
    self.qt_log_server.sendto(msg.encode(), self.qt_iport)
# DNM Done메시지
if msg_id[0] == 6:
    try:
        msg = "[ADS -> Car2] MSG_TYPE : DNM Ack, 협력주행 끝"
        self.qt_log_server.sendto(msg.encode(), self.qt_iport)
        ads_tmp_id = struct.pack('>B I', 0, self.dnm_msg[3])
    except:
        pass
```

[그림 3.20] EDM parsing

③ DNM

- ADS가 협력 주행을 맺는 과정에서 보내지는 메시지를 구분
- DNM Request 메시지가 수신되었을 때, 시뮬레이터 차량이 협력 주행 수락 메시지를 전송할 수 있도록 전역 변수 ads_tmp_id 변수 값 수정
- DNM Done 메시지가 수신되었을 때, 시뮬레이터 차량이 협력 주행 수락 메시지를 전송하지 못하도록 전역 변수 ads_tmp_id 변수 값 수정
- try – except를 이용하여, DNM Done 메시지가 먼저 들어온 경우, 오류가 발생하는 것을 예방

3) PYQT : 시뮬레이터의 차량들의 속도 정보와 주고 받는 데이터의 기록을 위하여 작동한다.

- 속도 그래프는 ADS, Car1, Car2, E_car의 속도 표시
- Log 창은 주고받는 데이터를 표시

❑ 속도 그래프

```
def send_speed(self):
    if self.case == 4 or self.case == 5:...
    else:...

    speed = "speed" + ads_speed + car1_speed + car2_speed

    self.qt_log_server.sendto(speed.encode(), self.qt_iport)
```

[그림 3.21] 속도 전송 send_speed 함수

```
def receive(self):
    global ads_speed, car2_speed, car1_speed
    while True:
        try:
            msg, self.addr = self.server.recvfrom(90000)

        except:
            print('Logging stop')
            break
        else:
            if msg:
                if msg.decode()[:5] == "speed":
                    ads_speed = int(msg.decode()[5:8])
                    car1_speed = int(msg.decode()[8:11])
                    car2_speed = int(msg.decode()[11:14])
                else:
                    self.recv_signal.emit(str(msg.decode()))
```

[그림 3.22] 속도 정보 업데이트

- 3개의 속도 그래프는 0~100까지의 정수를 표현 가능
- main_menu.py에서 send_speed() 함수를 통해 전송되는 속도 정보를 그래프에 표시

```
def ads_plot_data(self):

    self.x = self.x[1:]  # Remove the first y element.
    self.x.append(self.x[-1] + 1)  # Add a new value 1 higher than the last.

    self.y = self.y[1:]  # Remove the first
    self.y.append(pyqt_server.ads_speed) # Add a new random value.
    self.data_line.setData(self.x, self.y)  # Update the data.
```

[그림 3.23] ADS 속도 업데이트

```
def car1_plot_data(self):

    self.x = self.x[1:]  # Remove the first y element.
    self.x.append(self.x[-1] + 1)  # Add a new value 1 higher than the last.

    self.y = self.y[1:]  # Remove the first
    self.y.append(pyqt_server.car1_speed) # Add a new random value.
    self.data_line.setData(self.x, self.y)  # Update the data.
```

[그림 3.24] Car1 속도 업데이트

```
def car2_e_plot_data(self):

    self.x = self.x[1:]  # Remove the first y element.
    self.x.append(self.x[-1] + 1)  # Add a new value 1 higher than the last.

    self.y = self.y[1:]  # Remove the first
    self.y.append(pyqt_server.car2_speed) # Add a new random value.
    self.data_line.setData(self.x, self.y)  # Update the data.
```

[그림 3.25] Car2 or E_car 속도 업데이트

- receive되는 send_speed() 데이터를 log 창에 출력하지 않고 변수로 저장
- Multi-Thread로 작동하는 3개의 그래프가 변수의 값을 읽는다.
- case a, b에서는 3개의 그래프가 모두 작동
- case c, d, e에서는 ADS 그래프와 맨 아래 그래프만 작동

- case c에서는 Car2의 속도를 표시
- case d, e에서는 E_car의 속도를 표시

❑ Log 창

- 시뮬레이터에서 주고 받는 데이터의 Log를 PYQT에서 전송받아 Log 창에 표시
- 시뮬레이터를 시작하면 바로 Log 창에 데이터가 표시
- pyqt_server.py와 graph_chart.py의 log_widet이 부모와 자식 관계
- [그림 3.22]에서 위의 PYQT receive시 데이터가 받아지고 signal을 통해 받아진 데이터가 전송

```
self.recv_signal.connect(self.parent.updateMsg)
```

[그림 3.26] signal 선언 in pyqt_server.py

```
def updateMsg(self, msg):
    self.msg.addItem(QListWidgetItem(msg))
    self.msg.setCurrentRow(self.msg.count() - 1)
    self.f.writelines(msg)
    self.f.writelines("\n")
    if self.msg.count() > 350:
        self.msg.clear()
```

[그림 3.27] updateMsg() 함수

- updateMsg를 통하여, Log창에 맨 밑줄로 이동하여 receive된 데이터 추가
- Log 창의 줄이 350을 넘으면 자동으로 초기화
- Log 창에 데이터가 쌓이면 시뮬레이터가 느려지는 현상이 발생

```
self.f = open(r"log_file.txt","a")
```

- Log 창에 추가되는 데이터는 Log 파일로도 추가되어 저장
- log_file.txt 파일로 저장되는데, 이어쓰기 모드이기 때문에 시뮬레이터를 종료하고 삭제하거나 다른 이름으로 저장 필요

4. 결론

본 용역 과제에서는 ADS의 협력주행 알고리즘 개발 및 동작 검증을 위한 시뮬레이터를 개발하였다. 이를 위해서 5개의 협력주행 Class를 정의하고, 실 환경 테스트를 진행할 전자통신연구원 주변도로를 시뮬레이션 환경으로 구축하고, ADS로부터 발생한 데이터를 수신하여 시뮬레이션에 반영하고, C_Vehicle들의 움직임을 구현하고, 시뮬레이션을 분석하기 위한 다양한 방법들을 제공할 수 있는 기능을 개발했다. 전자통신연구원에서 개발한 ADS와의 연동 테스트를 통해 시뮬레이터의 동작을 검증하였다. 본 보고서는 이렇게 개발된 시뮬레이터의 협력 주행 프로토콜과 소스코드에 대해서 자세하게 기술하였다.