CAN Data Generation

Lewis Koplon, Fernando Gutierrez, William Lund

Emails: lewisk1899@email.arizona.edu, afgutierrez8@email.arizona.edu,

wmund@email.arizona.edu

ECE 473 Software Engineering Concepts Spring ‘21

March 5, 2021

Executive Summary:

A Controller Area Network (CAN) allows for devices to communicate with other devices' applications without the need for a central host computer. This application is used in all vehicles used today and through a wide variety of other applications. The goal of this project is to develop an efficient software that will utilize DBC rules to provide the user with humanly readable CAN data given a rosbag file containing sensor data from a vehicle. Tools such as CAT Vehicle Simulator and CANTools Python library will provide us with useful features that can help us collect and present CAN data in various ways.

Table of Contents

Table of Figures

1. Project Overview:

Controller Area Network (CAN) “is one of the most dominating bus protocols” that has been created for in-vehicle networks. This was due to the fact that they were able to greatly reduce wiring, weight, and cost while still using more electronics in vehicles. CAN buses can be found in a wide variety of applications from the vehicles that we see on the road every day, to streetcars, trains, and even aircraft control systems. CAN devices are able to transmit messages to one another through a network in packets called frames. These frames contain streams of bits which we can extract to gain useful information about the system we are using such as how our engine is performing. Further analysis into our project will determine how we can use these data fields to convert into a humanly readable format, using standard DBC rules.

2. Requirements:

2.1 Application/Functional Requirements:

2.1.1 Overview:

This section will discuss the requirements that the program will have to meet. There is class B that will represent the minimum requirements that would still produce an effective program, and there is class A which specifies quality of life improvements for the user. These requirements are subject to change over the term as the project progresses.

2.1.2 ‘B’ requirements:

Class B: The program will be coded in python as to utilize the CANtool python library. We will be taking data from sensors and converting it into raw CAN data in the form of the CAN bus. The raw CAN data can be converted into readable data by utilizing a DBC file. DBC files are used to translate raw CAN data. We will be utilizing the ”acura ilx 2016 can generated.dbc” to encode our sensor data. We will develop a command line tool that takes a rosbag file, which would be read using bagpy, and outputs a text file filled with CAN data organized by time the message was sent. The sensor data that will be

converted to CAN, will be the ones that are useful to autonomous navigation.

2.1.3 'A' requirements:

Class A: We will make a linux application that allows the user to input the rosbag file, and specify a filename as the output. Class A will include all the requirements from Class B, however it will provide the user with a UI. There will be no memory leaks.

3. Application Analysis:

3.1 Overview:

The scope of this class will allow us to focus on signals that are especially useful in navigation control, specifically within autonomous vehicles. DBC files contain rules for encoding and decoding raw CAN data streams, which we can then use to extract relevant information from the bits. Each CAN message in a DBC file contains a unique identifier, name, bit length, as well as a variable number of signals. The DBC provides details on how to decode each of the signals. This means that we will have to match each CAN identifier to its corresponding set of rules in the DBC file. For 11-bit identifiers, we can look them up directly in the DBC. For 29-bit identifiers, we will need to do some bit-masking in order to correctly match to the corresponding 32-bit identifier in the DBC file.

3.2 User Interface Design:

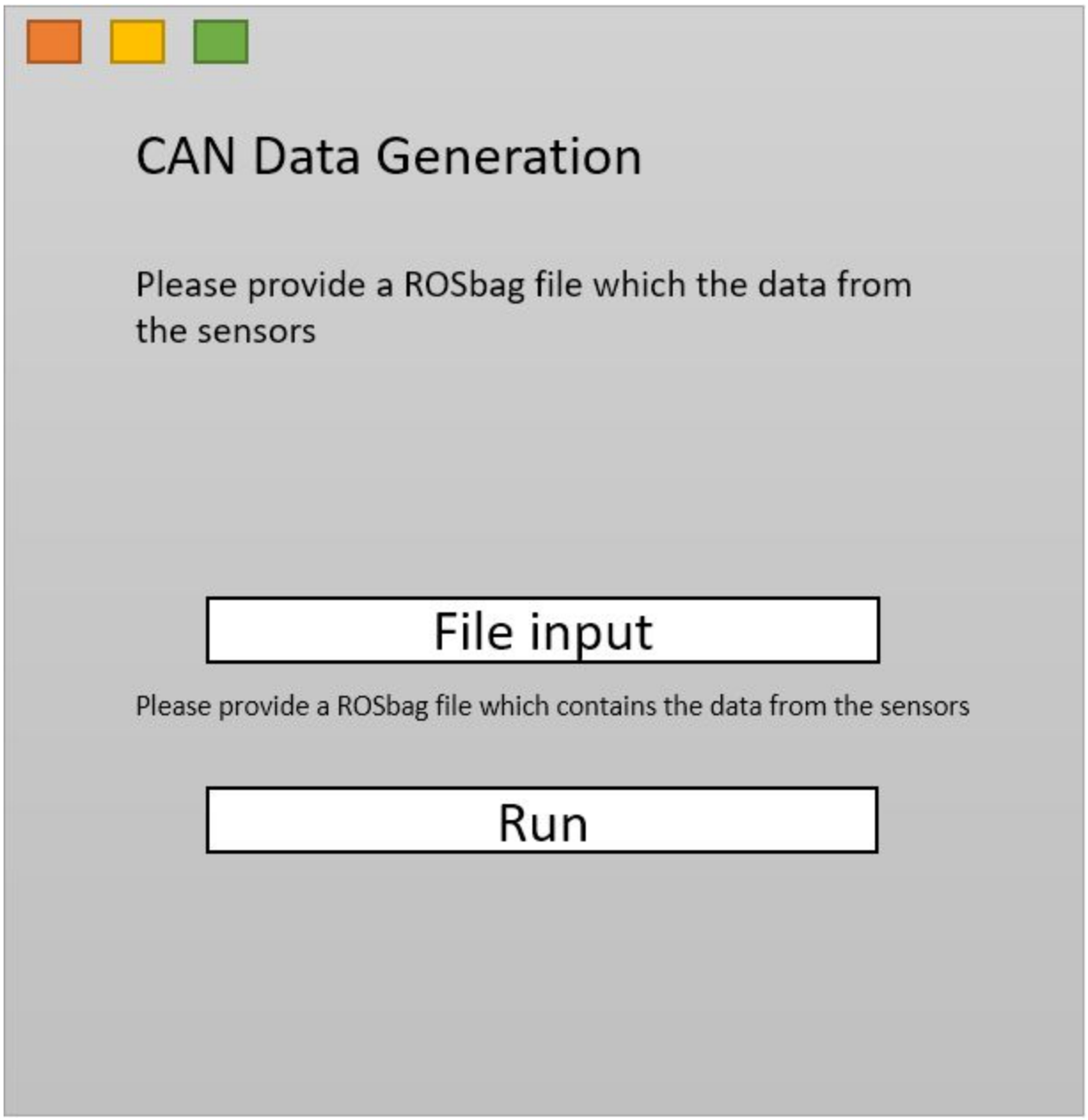


Figure 1: User Interface for the desktop application specified in the requirements of Class ‘A’

The user interface will ask for an input from the user, this input will specify where in the directory the program can find the ROSbag file. The

desired type of file is specified under the input box, it informs the user to provide a rosbag that contains data from the sensors. After the file is provided, the user will be able to press the run button, providing a window asking what the user wants to name the output text file.

3.3 State Model:

3.3.1 State Model According to Class ‘B’:

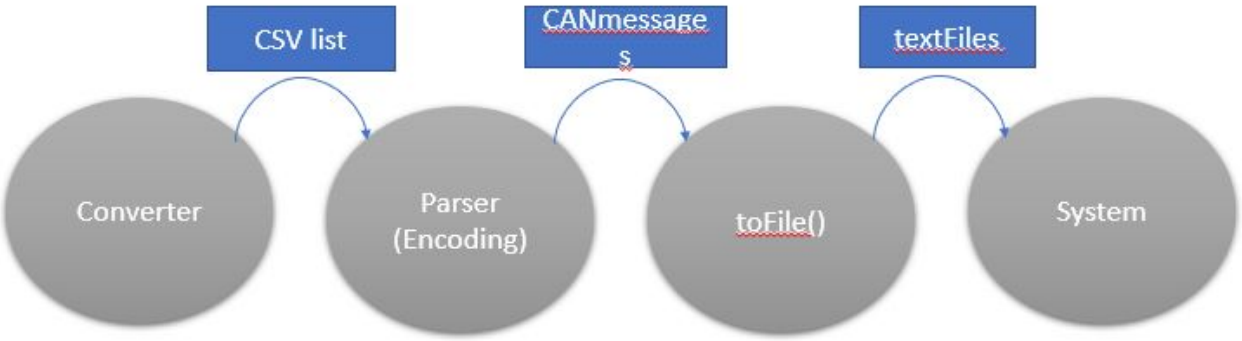


Figure 2: Requirement Class ‘B’ state diagram

This state diagram will take the ROSbag file, and create an array of csv files, which will be parsed and encoded. A list of lists will be passed to an object called toFile(), this will create text files out of each respective sensor's converted data and append it to a file.

3.3.2 State Model According to Class ‘A’:

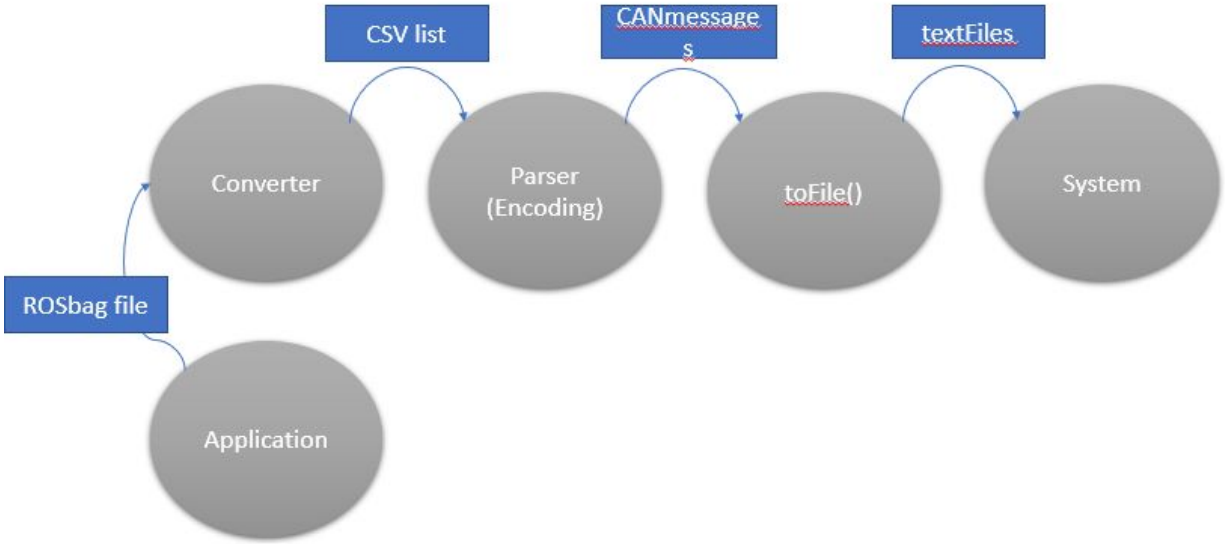


Figure 3: Requirement Class ‘A’ state diagram

Same explanation as figure 2, however there will not be an interface for the user to utilize.

4. Domain Analysis:

4.1 Overview:

This section will provide a brief overview on ROS messages and how our software will be interacting with these messages. The ROS messages that we will be interacting with in our software will be messages containing relevant CAN data. This section will also provide interaction models pertaining to our system.

4.2 ROS Messages

ROS messages are simple data structures that are made up of typed fields and can also include primitive data types as well as arrays of primitive types. Nodes are able to communicate with other other nodes by publishing messages on topics. ROS messages can also be found in ROSbag files. These rosbag files can be created by subscribing to the topics that the nodes publish to and storing the collected data in an efficient file. Our software will utilize rosbags containing relevant information (CAN data) to make full use of the tools offered within ROS.

4.3 Use Cases:

Case - User inputs rosbag file

Description - The user will input a rosbag file and pass it as an argument into our program. The program will read the rosbag file and the file will be decoded using CANTools python library. Using the selected dbc file and following the specified rules to decode the CAN data, we will be able to output a .txt file that presents the CAN data in a user friendly format.

Exceptions - file that user provides does not follow proper encoding, encoding does not match dbc file and generates error

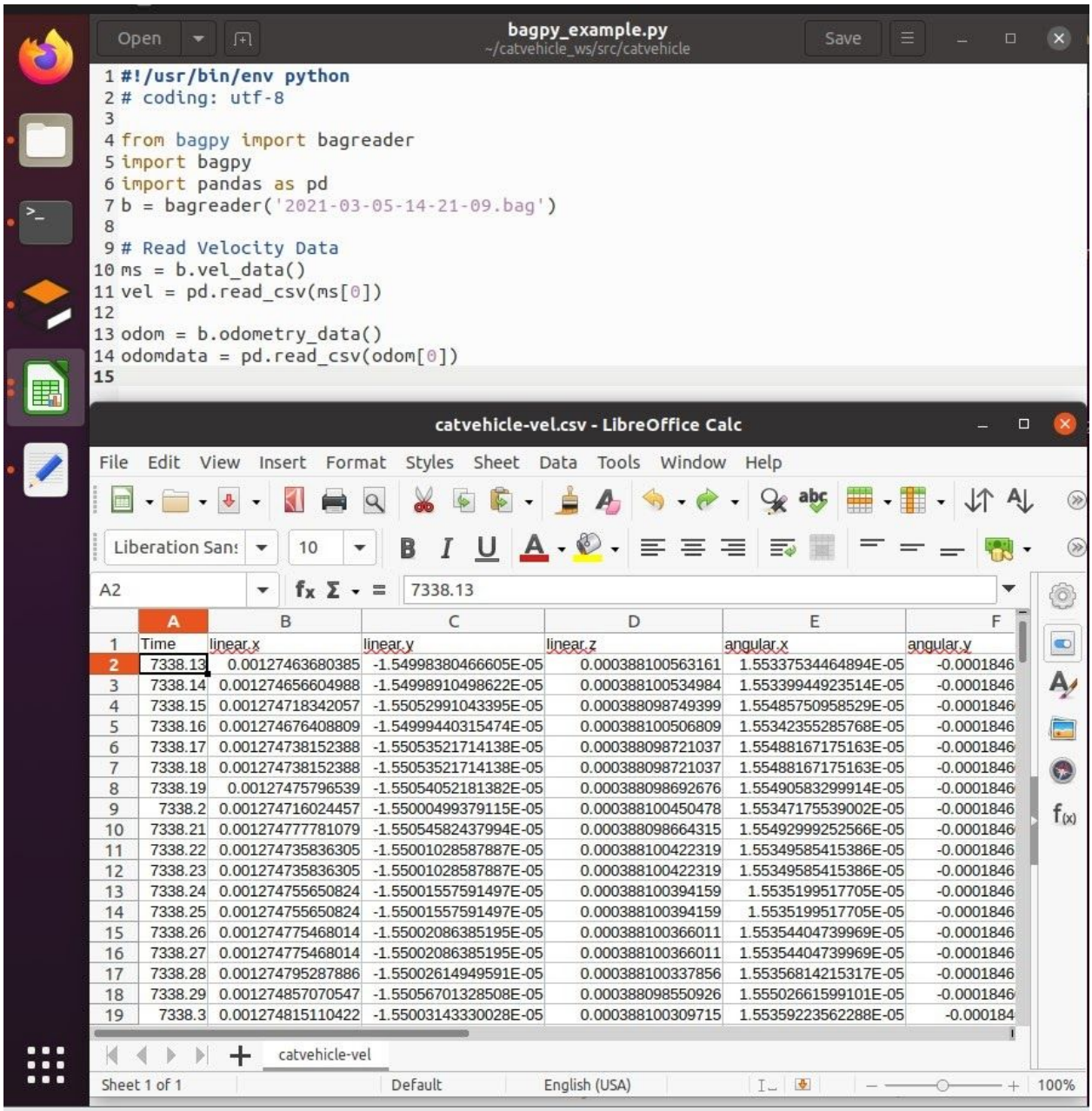
5. Important Algorithms:

5.1 Overview:

CANtools is a CAN Bus Python library that allows for a user to parse DBC files, encode/decode CAN messages, and monitor a CAN Bus, along with many other useful operations. For the purpose of this project, we will utilize CANtools mainly for parsing files and encoding/decoding data we gather from our simulator. Bagpy is another Python library that facilitates the reading of a rosbag file. This will be especially useful to us in properly reading the input from the user and extracting valuable information needed to convert the given CAN data.

5.2 Python Bagpy :

We will use bagpy to read the data from our rosbag file in order to then convert it to CAN data. Below is example code for generating a .csv file full of messages on a rostopic.



	A	B	C	D	E	F
1	Time	linear.x	linear.y	linear.z	angular.x	angular.y
2	7338.13	0.00127463680385	-1.54998380466605E-05	0.000388100563161	1.55337534464894E-05	-0.0001846
3	7338.14	0.001274656604988	-1.54998910498622E-05	0.000388100534984	1.55339944923514E-05	-0.0001846
4	7338.15	0.001274718342057	-1.55052991043395E-05	0.000388098749399	1.55485750958529E-05	-0.0001846
5	7338.16	0.001274676408809	-1.54999440315474E-05	0.000388100506809	1.55342355285768E-05	-0.0001846
6	7338.17	0.001274738152388	-1.55053521714138E-05	0.000388098721037	1.55488167175163E-05	-0.0001846
7	7338.18	0.001274738152388	-1.55053521714138E-05	0.000388098721037	1.55488167175163E-05	-0.0001846
8	7338.19	0.00127475796539	-1.55054052181382E-05	0.000388098692676	1.55490583299914E-05	-0.0001846
9	7338.2	0.001274716024457	-1.55000499379115E-05	0.000388100450478	1.55347175539002E-05	-0.0001846
10	7338.21	0.001274777781079	-1.55054582437994E-05	0.000388098664315	1.55492999252566E-05	-0.0001846
11	7338.22	0.001274735836305	-1.55001028587887E-05	0.000388100422319	1.55349585415386E-05	-0.0001846
12	7338.23	0.001274735836305	-1.55001028587887E-05	0.000388100422319	1.55349585415386E-05	-0.0001846
13	7338.24	0.001274755650824	-1.55001557591497E-05	0.000388100394159	1.5535199517705E-05	-0.0001846
14	7338.25	0.001274755650824	-1.55001557591497E-05	0.000388100394159	1.5535199517705E-05	-0.0001846
15	7338.26	0.001274775468014	-1.55002086385195E-05	0.000388100366011	1.55354404739969E-05	-0.0001846
16	7338.27	0.001274775468014	-1.55002086385195E-05	0.000388100366011	1.55354404739969E-05	-0.0001846
17	7338.28	0.001274795287886	-1.55002614949591E-05	0.000388100337856	1.55356814215317E-05	-0.0001846
18	7338.29	0.001274857070547	-1.55056701328508E-05	0.000388098550926	1.55502661599101E-05	-0.0001846
19	7338.3	0.001274815110422	-1.55003143330028E-05	0.000388100309715	1.55359223562288E-05	-0.000184

Figure 4: bagpy example

5.3 Python CANTools library:

We will be using the cantools python library to encode and decode our CANdata with our chosen .dbc file. Below is an example from their documentation on how we will do this.

The example continues encoding a message and sending it on a CAN bus using the python-can package.

```
>>> import can
>>> can_bus = can.interface.Bus('vcan0', bustype='socketcan')
>>> data = example_message.encode({'Temperature': 250.1, 'AverageRadius': 3.2, 'Enable': 1})
>>> message = can.Message(arbitration_id=example_message.frame_id, data=data)
>>> can_bus.send(message)
```

Alternatively, a message can be encoded using the encode_message() method on the database object.

The last part of the example receives and decodes a CAN message.

```
>>> message = can_bus.recv()
>>> db.decode_message(message.arbitration_id, message.data)
{'AverageRadius': 3.2, 'Enable': 'Enabled', 'Temperature': 250.09}
```

Figure 5: cantools example

6. Class Design:

6.1 Overview:

The classes in the project are the same as our states shown in figure 2. We have a converter class which is responsible for taking our ROSbag file and converting all the data into their respective csv files partitioned based on the signal;. Next we have our parser which will go through each csv and encode the raw sensor data to the CAN message format, this will be done using CANtools. The class toFile will be responsible for taking the CANmessages and outputting them to their own respective files once again based on the signal.

6.2 Class Diagrams:

Converter
-csv_list: String
+rosbag_to_csv(file: String): void

Parser
-csv_list: String **-CANmessage: String[][]**
+encode(): void

ToFile
-CANmessage: String[][]
+csv_to_text():File //write to file

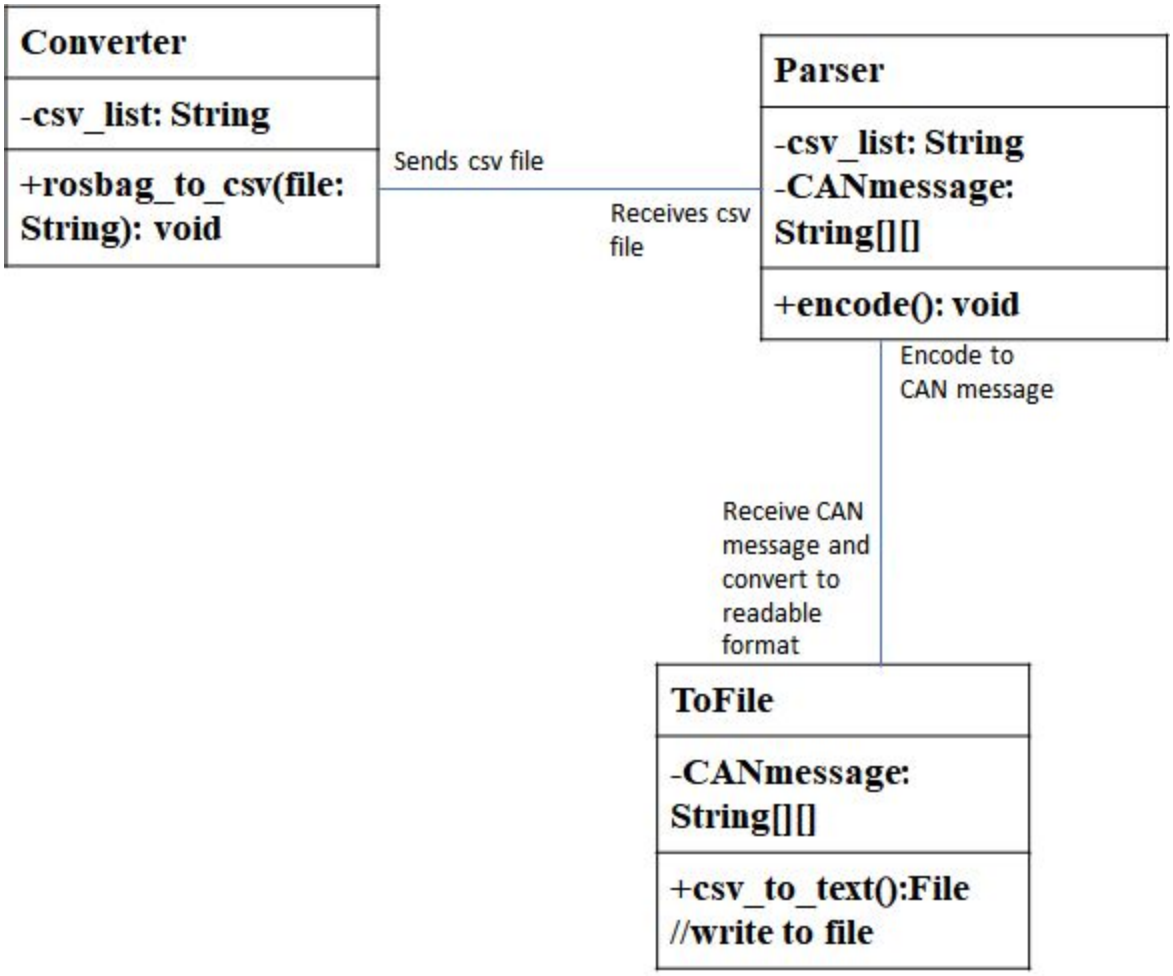


Figure 6. Class Diagram

7. Testing Strategy:

Our testing strategy will involve decoding the CAN data we generate and comparing it to the input data. Our goal is to ensure that the output data matches the input data, meaning nothing is lost during encoding and decoding. We will do this for every type of message, starting simply with velocity on the topic /vel. Anytime we implement a new signal type, we will go through the testing procedure.

8. **Integration with Platform:**

The use of any sensors will be limited to the sensors in the CAT Vehicle Simulator. These sensors will be used to generate data regarding a nonspecific vehicle for simplicity's sake. We can monitor and extract relevant data that we obtain from the CAT Vehicle Simulator. The CAT Vehicle Simulator also provides us with other data regarding camera and lidar features.

9. **Task allocation with Breakdown:**

9.1 'B' requirement task allocation: All members will contribute equal amount of work/ideas to satisfy each portion of the 'B' requirements

9.2 'A' requirement task allocation: All members will contribute equal amount of work/ideas to satisfy each portion of the 'A' requirements

10. **Timeline for Completion:**

March 10th: Converting All ROSbag files into CSV using bagpy

March 16th: Converting data in CAN bus message form

March 19th: Updated Project Design Document due by 5:00 pm

March 20th: Output CAN data to a text file

April 2nd: Alpha Release due by 5:00 pm

April 9th: Requirements Verification due

April 23rd: Beta Release due by 5:00 pm

April 30th: Initial Online Documentation due

May 3rd-5th: Project Presentations

May 5th: Final Release due by 5:00 pm