Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial
15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática
ESPECIALIDAD EN SISTEMAS EMBEBIDOS

CAN and Debugger Interface with Python

Trabajo recepcional para obtener el grado de
Especialista en Sistemas Embebidos

Presenta: DIEGO PRECIADO BARÓN

Asesor PH.D. LUIS RIZO DOMÍNGUEZ

Tlaquepaque, Jalisco. agosto de 2019.

Acknowledgment

Abstract

Nowadays, Python is one of the most popular programming languages and it is becoming little by little the favorite one for many people. On this project, we will describe its advantages on an important sector, test automation for embedded systems. The automotive market has a lot of challenges which add complexity to the development of any product and OEM's are demanding not only more features to their product but also, they want those features fulfilled within a short period of time due to the high competition and customer demands. To keep up the pace, the automation process must be improved, so with more features and updates added to the product, more tests and verifications are needed. Even though visual inspection is always necessary, the need for test automation in embedded system is now a priority to conclude faster verification steps in the development path of the product, improving efficiency and effectiveness avoiding and pointing out human mistakes that no matter what they will be always present. Python is now supported in many tools for communication (for CAN and LIN protocols) and debugging in embedded systems, therefore, combined with their general programming language advantages it can be quite important and useful in the toolset of any software engineer.

Contents

Figures

1 Introduction

The goal of this project is to develop an interface for communication through the CAN protocol and an interface to control the application variables through the iSystem debugger with the programming language Python. This interface has as final objective to ease the realization of test automation for white and black box on any embedded system.

There are three main concepts for this project, CAN protocol, debugger and Python. All essential features and importance will be explained. Furthermore, the reasons of selection of Python instead of another programming language will be discussed with more detail.

After all the explanations, an application will be developed to demonstrate the wide range of test cases that can be created on any project, making emphasis on the advantages of using Python.

1.1 Development Tools

The following tools and libraries have been selected for the development of the project and they are going to be mentioned and described throughout the entire document:

SW tools:

- Spyder from Anaconda: IDE for Python 3.6

- CANoe from Vector (version: 9.0.53 64bit)

- winIDEA from iSystem (version: 9.17): only for comparison purposes.

HW tools:

- CANcase XL

- iSystem Debugger IC5000

Libraries:

- win32com.client: Dispatch interface, who contains properties and methods for COM (Component Object Model) automation.

- isystem.connect: API to connect winIDEA with Python.

- Matplotlib: for data plotting.

2 Python

Python has been evolved as one of the most demanding programming languages with fast growing nowadays. It exists a development community that everyday keeps increasing the number of members, turning this into a great advantage since it is more easy and likely to find a module or library for any specific topic. For example, there are libraries for signal processing, computer vision, data processing, machine learning and so on.

This project is developed on Python 3.6 which has now more technical support than Python 2.7, and it is going to be completely displaced at some point.

The modules that will be integrated are the ones for CANoe and winIDEA, but additionally to these modules, Python has several other libraries that can be integrated on this or any project. One of these is Matplotlib that will be described briefly in the following sections, but some others are out of the scope of this document. For instance, tkinter and pyQt are modules created to design graphic user interfaces, that are great to improve our test tool and ease the way we interact with the machine. Another good example is Pandas, that will be explained in the application section to show only how it can be integrated on a big project when there is a lot of configurations and scenarios to test. Also, Python can work with XML files and create Word or Excel documents with their respective library, in case a final test report is needed.

Besides the wide number of libraries, there are two other important advantages of Python versus other programming languages like C++ or Java. First, Python is dynamically typed, so there is no need to control the data types of your code (e.g. specify if the variables are an unsigned byte, signed 16-bit, string, array etc.) since Python interpreter recognizes it during runtime. Second, in this case like Java, Python has a garbage collector so even if you do not delete a created object when you don't need it anymore, the garbage collector will do that job for you when the object is out of the application scope.

3 CAN

The communication protocol CAN (Controller Area Network) is a method of communication between different electronic systems and it is being used on different industry sectors. The automotive industry is mainly integrating CAN because it has many advantages like being a multi-master protocol with a collision detection and have a good noise immunity. All these and more features make of CAN a very reliable communication protocol.

On this project, CANoe is being selected instead of CANalyzer as CAN framework from Vector, due to the reasons that CANoe has all features from CANalyzer, however, CANalyzer has some restrictions when using external sources like win32 on which the most relevant one is that it is not possible to send messages through it, so even though the Python module supports CANalyzer it can be only used to read messages. [1]

3.1 Python and CAN

As stated before, CANoe is going to be used on this project as a COM server. The Component Object Model of Microsoft is a binary standard which allows application control. COM is independent of the programming language, so modules can be created under different languages like Java, C/C++ and Python. [2]

It's important to mention that to get familiar with the properties and methods of any COM application, it is necessary to investigate the documentation of the product.

Anaconda has already included the win32com package. Furthermore, the COM server is registered once CANalyzer or CANoe are installed.

The object data from the COM server is organized in a hierarchy structure. CANoe has this main table as reference in the Help section, on which anyone can navigate through each object to find more information. The table below shows a part of the entire structure.

Figure 1 COM Object Hierarchy

On Figure 1, the first column indicates the Application object, followed by the second column which has Bus, CAPL, Configuration, etc. This project only focuses on the Bus object since there we can configure the transmission and reception of messages. The rest of objects and Configuration won't be part of the development of the project.

3.1.1 CANoe class

To start using the CANoe object, the module win32com.client has to be imported in the Python script:

import win32com.client

After importing all needed libraries, a class called CANoe will be created and as constructor only the object self.Application is defined calling their own constructor of the class CANoe COM:

self.Application = win32com.client.Dispatch('CANalyzer.Application')

The above line is the most important which will allow us to start developing the project considering the reference table Object Hierarchy from COM server (Figure 1).

The selection of any object from Figure 1 will show a description like Figure 2 with more information regarding the software design of each object, such as syntax, return values, parameters and in most of them, some code examples. The examples provided are written in VB.Net and C#, but as stated before COM server is independent of the implemented programming language.

Implemented By	| **Buses** object |	
Syntax	object.GetSignal(channel, message, signal)	
Function	Returns a **Signal** object.	
Remarks	Other bus types (**LIN, FlexRay, MOST, AFDX, Ethernet**) can be accessed in the same manner as **CAN**.	
Parameters	**Parameter**	**Description**
	object	The **Buses** object.
	channel	The channel on which the signal is sent.
	message	The name of the message to which the signal belongs.
	signal	The name of the signal.
Return Values	The **Signal** object.	
See Also	| **Bus** object | | COM Samples: CANSystemDemo — CANoe Signal.vbs |	
	Example	
| | ```
Set App      = CreateObject( "CANoe.Application")

// Access to the signal value
Set Signal   = App.Bus( "CAN").GetSignal( 1, "ABSData", "CarSpeed")
value        = Signal.Value
Signal.Value = 12
``` | |

Figure 2 Object Design

3.1.2 Main methods

The GetSignal method from the Bus object allows, in a straight forward way, to retrieve the value of a desired signal. It needs three parameters when the method is called: channel, message and signal.

In the line below the channel on which we are transmitting is the number 1, the message is *Steering_Data_HS1* and the signal is *TurnLghtSwtch_D_Stat*. The channel is an integer, message and signal are a string type.

Steering_data = self.Application.Bus.GetSignal(channel, message, signal)

Steering_data is an object of type Signal and this object has a property called Value. So now through this *Steering_data* object we can set any value we want. When this object was created it points to the original main object application.

Steering_data = self.Application.Bus.GetSignal(channel, message, signal)
Steering_data.Value = value

3.1.3 Other methods

Here we are going to describe briefly some other methods that can be useful on any application:

- **Start**: It begins the measurement. The Running method must be invoked first to check if the application measurement was already running or not.

- **Stop**: It stops the application measurement from CANoe

- **Running**: Returns True or False if the measurement is running.

4 iSystem debugger

The iC5000 debugger is the development tool of iSystem company and it is used in this project for a 32-bit microcontroller. The framework supported is winIDEA, on which application variables can be read and written. Also, winIDEA has the daqIDEA interface to plot any variable.

4.1 Python and iSystem

The other important module for this project is isystem.connect which is quite useful when creating scripts for test automation. It allows us to control application variables through the debugger similar as we normally do in winIDEA. The Software Development Kit has been developed by the iSystem company supporting several programming languages like C++, C#, Java, Python and even LabView and Matlab.

The complete reference is found in the iSystem web page under the SDK download section, but on this document the main functionalities are explained. [3]

Even though the latest Python version is 3.7, iSystem.connect has not been updated, so the latest installation package supports Python 3.6.

To start having control of the debugger through Python, three methods must be invoked. First the connection to winIDEA must be performed by creating an object of class *ConnectionMgr* and then calling the method *connectMRU*. Lastly, the *CdebugFacade* and *CdataController* methods from isystem.connect class are called by passing as parameter the *ConnectionMgr* object previously created.

```
self.cmgr = ic.ConnectionMgr()
self.cmgr.connectMRU('')
self.dbg = ic.CDebugFacade(self.cmgr)
self.dataCtrl = ic.CDataController(self.cmgr)
self.CPUStat = self.cpuStatus()
```

Once the connection is established and the object controller is created in constructor, the methods are integrated.

As noted in the **Python and CANoe** section only some functionalities will be described and used in the **Application** section. The general methods are reset, stop, and run that control winIDEA and don't need further explanation. Read and write functions are the most common methods from the API, and they are quite easy to implement.

4.1.1 Read method

The read method comes from the object created from *CdataController* and returns the value obtained by using its method *evaluate* passing the parameters *IconnectDebug.fRealTime* and the name of the variable only.

```
self.dataCtrl = ic.CDataController(self.cmgr)
value = self.dataCtrl.evaluate(ic.IConnectDebug.fRealTime, varName)
```

4.1.2 Write method

For the write method the implementation is quite similar as the read one, but instead of the *evaluate* method we use the *modify* method and as additional parameter we include a string argument as the new value.

```
self.dataCtrl = ic.CDataController(self.cmgr)
self.dataCtrl.modify(ic.IConnectDebug.fRealTime, varName, str(value))
```

5 Python and Matplotlib

Regarding data visualization, CANoe and CANalyzer have their own functionalities, however, Python has as well some libraries to create and customize plots. One of the most relevant is Matplotlib which can be integrated in the script to get a visual inspection

of the test application in real time or for statics with any kind of representation, like plots, histograms, bar charts, scatterplots etc. [3]

The two figures below (Figure 3 and 4) are examples of scatter plots of two different signals: Steering data and Alarming data respectively (The values were changed in a short time interval for visualization purposes).

Figure 3 Steering data in Matplotlib

Figure 4 Alarm data in Matplotlib

6 Application

A vehicle has different electronic modules with several applications running and all of them must work as intended, if not, some alternative actions must be performed. To identify one or the another, diagnostics are found on any application.

One specific type of diagnostic is DTC (Diagnostic Trouble Code) which is formed of many conditions, but the most important ones are the identification number or ID, the required conditions or preconditions, and the conditions for incrementing or decrementing. Furthermore, there are DTCs of different kind and they are distinguished by their ID number (e.g. 781412). In most of the cases, they can have a precondition that needs to be fulfilled before the DTC is evaluated, these conditions could be a specific configuration of each vehicle (e.g. whether a vehicle version has fog lamps or not) or general ones like having the ignition status in "On" or "Run".

A DTC has a debounce strategy, so once the preconditions are fulfilled, the evaluation of the DTC begins. This evaluation reads a specific variable and compares it to a constant value or range, so normally when the value exceeds the thresholds or fulfills the evaluation, a counter is incremented or decremented depending on the monitoring strategy. The increment and decrement counter can vary on each DTC and when the counter gets to +128, it is said that the DTC is set, on the other hand, when the counter reaches -127 it is said that the DTC is cleared.

The DTC can be of any kind, for instance, the lack of any specific message on CAN, an open circuit, a short-to-battery circuit, a short-to-ground circuit detected on the output, a stuck button etc. Also, a DTC can be evaluated on-going or on-demand.

Depending on the module of the vehicle the number of total DTCs can vary. A small module can have 20 DTCs but others like the body control module has 420 DTCs. Therefore, an optimal way to test all these methods for white and/or black box testing should be developed.

For this application, the selected DTC to test evaluates if the left rear turn light circuit has an open circuit condition or if there is a short-to-battery. Hence, before running the program the circuit will be set to open to get the confirmation of the DTC set.

6.1 Code explanation

We start by importing all the previously discussed modules into our project:

```
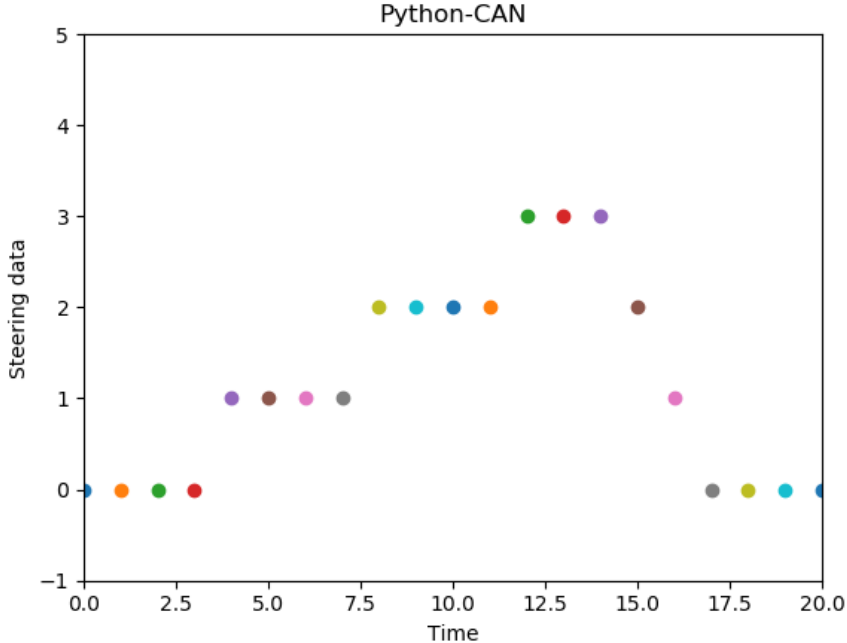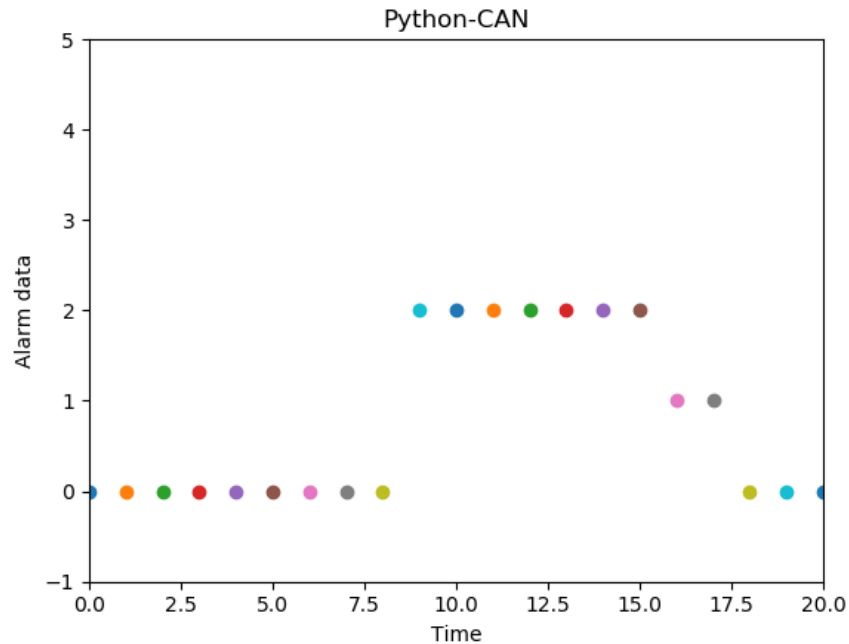import CANOE_Interface as can
import isystem_class as iSys
import matplotlib.pyplot as plt
```

Then, two Python dictionaries are created, first *DTC_924715* which stores the variables and their test values. CAN_data dictionary has the information of the name of the CAN message and signal, that will be plotted in the figure.

```
DTC_924715 = {'BlocksRamImage.struct_DE1A_1B_Configs.LR_Turn_Lamp_Ckt_Usage_Cfg': 2,
'BlocksRamImage.struct_z048_NVMConfigs.OC_Detect_Cfg[9]': 0,
'PIDCtrlLR_Turn_Lamp': 2}
CAN_data = {"Body_Info_6": "TurnLghtSwtch_D_Stat2"}
```

However, if we wanted to include hundreds of DTCs, one way to improve the previous examples is integrating the Pandas library in the Python script to store and get the relevant information of all DTCs. The Pandas data frame could be obtained from a text file, xml file or any other format.

Some constant and global variables are:

```
dtc_data = 0
counter = 0
TURN_LEFT = 1
TURN_RIGHT = 2
CAN_CHANNEL = 1
ID_924715 = 242
CONF_DTC_BIT = 0x08
```

The function below defines the plot features like dimensions and labels of the matplotlib figure we will create to show the CAN signal.

```
def figure():
```

```
plt.axis([0, 20, -1, 5])
plt.ion()
plt.ylabel('Steering 2 data')
plt.xlabel('Time')
plt.title("Python-CAN")
plt.show(block = False)
```

Under the *main* section, we create the class object instance of isystem.connect and CANoe, both applications are set to run. Besides, the function *figure()* is called:

```
if __name__ == '__main__':
        myIsystem = iSys.IC5XXXX()
        CANObj = can.CANoe()
        myIsystem.run()
        if(CANObj.Running()):
                print ("Running State: ", CANObj.Running())
        else:
                CANObj.Start()
        figure()
```

On the following loop, the variable that controls the turn left and turn right light of a vehicle is modified.

```
for key in DTC_924715:
        myIsystem.modifyIntegerVar(key, DTC_924715[key])
```

In an infinite loop, we are going to start reading the variables modified in iSystem and plot the output of the CAN signal. A pause must be set to give time, so the plot is updated and refreshed.

```
while True:
        counter += 1
        steering_data = CANObj.GetSignal(CAN_CHANNEL, "Body_Info_6",
        CAN_data["Body_Info_6"])
        dtc_data = myIsystem.readIntegerVarbyIdx("Dem_Cfg_StatusData.EventStatus",
        ID_924715)
        turn_light = myIsystem.readIntegerVar("PIDCtrlLR_Turn_Lamp")

        plt.scatter(int(counter), int(steering_data.Value))
        plt.draw()
        plt.pause(0.5)
```

dtc_data is a byte that contains information about the DTC state and the bit number 3 indicates if the DTC is set (1 for set and 0 for not-set). For that reason, the result of this value is masked to get the confirmation bit and evaluate the open circuit.

For this test only 5 seconds are needed for evaluation. The configurations and the result are printed in the console indicating if the test passed or not.

```
if counter > 10:
        cfg =
        myIsystem.readIntegerVar("BlocksRamImage.struct_DE1A_1B_Configs.LR_Turn_Lamp_
        Ckt_Usage_Cfg")
        print("EventStatus: ", dtc_data)
        print("LR_Turn_Lamp_Ckt_Usage_Cfg: ", cfg)
        print("LR_Turn_Lamp_Ckt: ", turn_light)
        validation = dtc_data & CONF_DTC_BIT
        if validation:
                print('Test Passed!')
        else:
                print('Test Not Passed!')
        CANObj.Stop()
        myIsystem.stop()
        break
```

When we run the test, we can identify the change in the variable by looking at the winIDEA watch window:

Name	Value
BlocksRamImage.struct_DE1A_1B_Configs.LR_Turn_Lamp_Ckt_Usage_Cfg	\x02 (2)
PIDCtrlLR_Turn_Lamp	\x02 (2)

Figure 5 winIDEA watch window

Changing the value of those variables make the module to activate a toggling signal between Left and Off state through CAN. The Figure 6 below shows the signal from our customized plot and Figure 7 the same signal but from the data signal analysis feature in CANoe. On this way we verify that the module is responding correctly.

Figure 6 Steering data output

Figure 7 Steering data output (CANoe)

Finally, the Figure 8 shows the console results. The event status byte shows 47, this value converted to binary is 10 1111, so the bit number three from left to right is 1 as expected for the test.

```
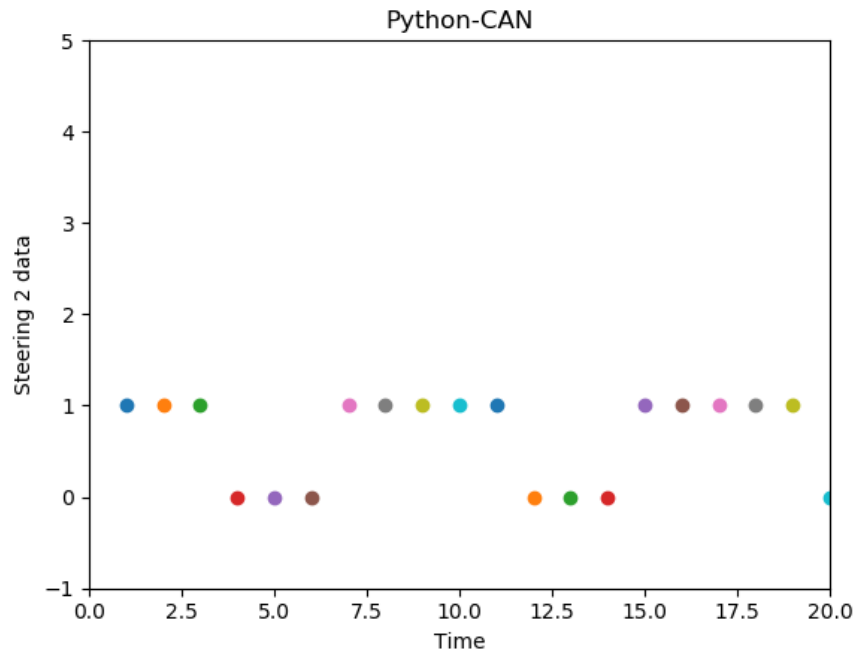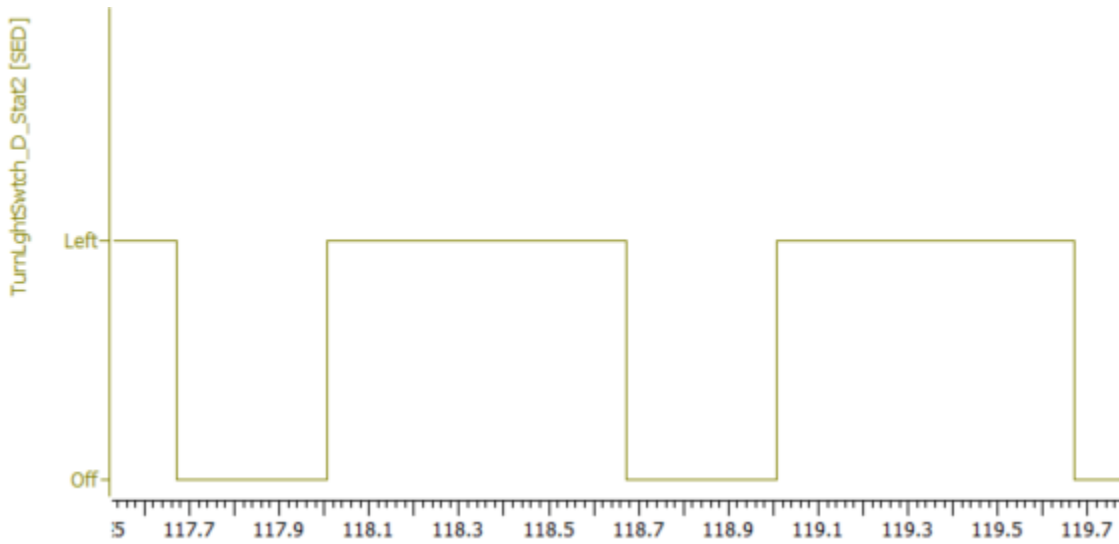In [138]: runfile('C:/Users/uidv7259/Documents/Especialidad/Application.py',
wdir='C:/Users/uidv7259/Documents/Especialidad')
Reloaded modules: CANOE_Interface, myTkinter, isystem_class
Running State:  True
EventStatus:   47
LR_Turn_Lamp_Ckt_Usage_Cfg:  2
LR_Turn_Lamp_Ckt:  2
Test Passed!
```

Figure 8 Console results

7 Conclusions

- Test automation is an area with many opportunities of improvements, which it can help to optimize and reduce the time development of any product.

- Python is dynamically-typed programming language with garbage collector, these advantages result in a quick integration of modules and strategy development.

- Python is a very powerful tool which can homogenize the development environment, where the embedded system makes use of multiple tools.

- Due to its syntax, Python is easy to get familiar with. For complex applications the development time is reduced compared to Java or C++.

- Like C++ or Java, Python can be treated as object-oriented programming language which helps to control the different classes of the different systems in a well-structured way.

- Python is a very popular programming language on which you can find plenty libraries for different purposes.

- We discussed the CANoe/CANalayzer modules from vector and the debugger modules from iSystem, but additionally, other Python libraries can be integrated to move the application script to the next level. These libraries could be Matplotlib, Pandas, Tkinter, Numpy etc.

8 References

[1]   *CANalyzer/CANoe as a COM ServerVersion 4.12017-03-20 Application Note AN-AND-1-117*

[2]   *Component Object Model (COM) https://docs.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal*

[3]   *Matplotlib Version 3.1.1 https://matplotlib.org/*