

POLITECNICO DI TORINO

M. Sc. In Mechatronic engineering



Master Thesis

Using image recognition to automate the testing process of instrument panels on commercial vehicles

Advisor

Prof. Maurizio Morisio

Candidate

Raffaele Solimene

Tutor

Eng. Luca Galeone

Company

KGR Elettronica S.r.l.

December 2020

ACKNOWLEDGMENTS

A special thanks to my family for giving me the opportunity to be free to choose what to do with my life and for supporting me in every choice I made. Thank you to love me every day.

Thanks to all my relatives and friends for having filled with affection and loyalty the moments we spent together. Thank you for offering me your precious time and for sharing with me all the experiences that have made me who I am.

Thanks to those who actively contributed to the writing of this thesis. Thank you, Luca, for supporting me with your knowledge and experience. Thank you, Giulio, for giving me useful advice as a colleague and as a friend. Thank you, Enea, to be a peaceful and supportive leader. Thank you, Livio, for teaching me how to become a great hacker. Thank you, Maurice, for the experience onboard the washing machine. Thank you, Maria, to be the best impostor “among us” . Thank you, Mr. Plancetta, for telling us amazing stories, you will be a great two-times father!

A thank you to my girlfriend, for giving me her time and for filling mine with love. Thank you for making me happy every day!

Abstract

In the software design field, and particularly in automotive applications, a great amount of time is nowadays dedicated to testing.

Methods to speed up and secure all the test processes that require repetitive and complex operations are needed. Moreover, for companies whose core business is focused on testing, the results achievable by the automation of the tests, become particularly valuable from different points of view: economically, a reduction in time means better use of resources (materials and employees); regarding safety, it should avoid a drop in attention by technicians and consequently, it should be improved the quality of the work; for competitiveness, an improvement in the quality of the job would lead to better opportunities for new customers. The host company (KGR Elettronica) for this thesis project, has a particularly active role in the field of tests of instrument panel cluster (IPC) and radio connectivity modules onboard vehicles. Until today, these tests are manually executed using specific software and hardware, both belonging to the Vector Group. *CANalyzer* is the software responsible for simulating the vehicle ECUs (Electronic Control Units), while the *CANcaseXL* is responsible for the interfacing between the simulated system on *CANalyzer* and the IPC. The communication between the various electronic components onboard the vehicle takes place mainly through a specific vehicle bus standard, the “Controller Area Network (CAN)”, which is a message-based protocol. It means that communication takes place with the exchange of messages through the network.

Nowadays, a technician is in charge of interpreting and solving the specifications related to the execution of the test. He must: configure the *CANalyzer* software and simulate the environmental conditions (he must simulate bugs too), then verify, visually, that the expected behavior on the IPC occurs.

For what concerns the problem of the automation of repetitive tests, a solution has been found programming with:

- CAPL (CAN Access Programming Language), a C-based programming language integrated with *CANalyzer*.
- Python, an open-source, high-level, and general-purpose programming language, easy to learn and, for its widespread use, full of libraries useful for the project purpose.

I wrote software, using CAPL language, able to send specific messages from the system simulated on *CANalyzer* to the IPC that, in turn, generates visual outputs (e.g. icons or pop-up messages) on its HMI (Human Machine Interface). Moreover, by writing and reading a text message, the CAPL software communicates with the Python one, establishing a sort of synchronization and notifying when the test is about to start or finish. The Python software, instead, turns on a camera to capture frames of the current status of the IPC and verifies that it corresponds to the expected one (uploaded in a database prepared during the initial phase of test configuration). At the end of the test, a new folder is created, containing: a text file with the main information about the test (if it is passed or failed and when it was executed); a collection of frames captured when the HMI of the IPC was in the expected status. The whole automatic test system I have created so far works correctly and allows me to achieve the expected result, although with ample room for improvement.

Contents

1.	Introduction	12
2.	Theoretical background	13
2.1	Verification and Validation	13
2.1.1	Verification	13
2.1.2	Validation	13
2.1.3	Fault.....	13
2.2	Test	14
2.2.3	Test Principles	14
2.3	Debugging	15
2.4	Testing	15
2.4.1	Testing Features	16
2.5	Hardware-in-the-loop testing	17
2.6	Controller Area Network (CAN)	17
2.6.1	CAN Network features:	18
3	Materials and Methods	20
3.1	Instrumentation	20
3.2	Test bench scheme	24
3.3	Implementation	25
3.3.1	Observation and study of manual or semi-automatic tests	25
3.3.2	Project Design	27

3.3.3	Configuration and Procedure	28
4.	Results	44
4.1	Automatic tests Results	44
4.2	Image Recognition Results	47
5.	Conclusion	49
5.1	Goals	49
5.2	Limits	49
5.3	Improvements	50
	References.....	52

List of Figures

Figure 1: Generic scheme of a debugging process	15
Figure 2: Generic scheme of a test process	16
Figure 3: Hardware-in-the-loop scheme.....	17
Figure 4: Example of CAN Network on a vehicle	18
Figure 5: CAN bus with 120 Ohm termination resistance	19
Figure 6: E.g. of Dominant (“0” level) and Recessive (“1” level) states for the CAN bus	19
Figure 7: A stabilized power supply	20
Figure 8: Home-made Breakout Box	21
Figure 9: The hardware used to interface the Electronic Control Units simulated through the CANalyzer software, with the Instrument Panel Cluster (IPC)	22
Figure 10: An Older version of the IPC effectively used in project.....	22
Figure 11: System composed of IPC, Breakout box, CANcaseXL	23
Figure 12: The camera used for the project	23
Figure 13: Test bench scheme	24
Figure 14: Example of a scheme extracted from the Vehicle Function description.....	26
Figure 15: Hand brake warning lamp.....	26
Figure 16: A test case example	26
Figure 17: Original project scheme	27
Figure 18: Vector interface hardware with two/four configurable channels	29
Figure 19: CAN communication channels.....	29
Figure 20: Signals contained in the TELEMATIC_DISPLAY_INFO message	29
Figure 21: CAN network representation on CANalyzer	33
Figure 22: Example of the message contained in an IG block	34
Figure 23: Steering wheel controls simulation	34
Figure 24: CAN Network managed with CAPL and simulated on CANalyzer.....	35

Figure 25: Declaration of different type variables.....	35
Figure 26: Initialization of cyclical messages	36
Figure 27: Initialization of key-status.....	36
Figure 28: Cyclic timer.....	36
Figure 29: Configuration panel inside CANalyzer	37
Figure 30: Dialog window to create new System Variables	37
Figure 31: System Variables customization	38
Figure 32: Graphic panel with the button to start the simulation	38
Figure 33: System Variable assignment.....	39
Figure 34: Template images used for comparisons by the image recognition software	41
Figure 35: Python function responsible for image recognition.....	42
Figure 36: Text file used to communicate with Python the beginning of the execution of the automatic test.....	44
Figure 37: First popup	44
Figure 38: Second popup.	45
Figure 39: Third popup.....	45
Figure 40: Fourth popup.	45
Figure 41: Fifth popup.....	46
Figure 42: Passed test status on the graphic panel	46
Figure 43: Failed test status on the graphic panel.....	46
Figure 44: Text file used to communicate with CAPL the beginning of the execution of image recognition software.....	47
Figure 45: Detected templates in the frames captured by the camera	47
Figure 46: Example of a Passed Test recorded in the log file	48
Figure 47: Example of a Failed Test recorded in the log file.....	48
Figure 48: The DSLR Camera, CANON EOS 200D	50

List of Tables

Table 1: Example of multi-frame message use	30
Table 2: InfoCode signal values interesting for project	31
Table 3: First frame, containing the three characters "E", "N", "E", to form the name "EAEA"	31
Table 4: Second frame, containing the last three characters "A", " ", " ", to form the name "EAEA"	32

Abbreviations and acronyms

CAPL	CAN Access Programming Language
CAN	Controller Area Network
CSMA	Carrier Sense Multiple Access
DSLR	Digital Single-Lens Reflex
DUT	Device Under Test
ECU	Electronic Control Unit
EOL	End of Line
HMI	Human-Machine Interface
IG	Interactive Generator Block
IPC	Instrument Panel Cluster
P	Program node
R&D	Research and Development
V&V	Verification and Validation
VF	Vehicle Function

1. Introduction

As can be guessed from the project's title, the research activity is aimed at the development of test automation techniques concerning digital instrumentation onboard vehicles (e.g. Instrument Panel Clusters). For the whole project, it was ensured that some specific requirements were met: it has been chosen to design a low-cost solution, avoiding the purchase of sophisticated hardware and software with functionalities not necessary for the development of the project; after being configured, the test must be able to be started intuitively, so that even an inexperienced user can do it; the outcomes of the automatic tests must be collected in a database, in order to be able to produce a *test report*, easy to read for the final user.

Starting from the manual or semi-automatic tests carried out by company technicians, the goal was to achieve a good level of automation of tests so that it can be possible to speed up processes that require repetitive and complex operations. The benefits cover various fields of interest:

- Economic

The time needed to execute each test process is significantly reduced, with the consequent possibility to manage in a more profitable way the company resources (employees and materials).

- Safety

The test technicians will be able to work on diversified operations, avoiding probable drops in attention deriving from the execution of repetitive operations. This improves working conditions and reduces the probability of sporadic or systematic test errors.

- Reliability

Avoiding human errors caused by a lack of attention, the quality of the service provided is improved and the overall image of the company becomes more reliable for any customers who will have to choose who to entrust new work commissions.

2. Theoretical background

2.1 Verification and Validation

To understand the functionality of a program, you must look at its specifications. Commonly, it can present some faults, and, through the unusual behavior of the software, one can discover them. The goal of Verification and Validation (V&V) is to ensure that the final program meets the needs of the customers.

2.1.1 Verification

Static Verification refers to the analysis of a static representation of the system to detect defects. It can be used at any stage of the development process, even before the system is implemented.

2.1.2 Validation

Validation can only be dynamic. Indeed, you cannot be sure that the system meets the requirements of the customer simply by looking at the structure.

2.1.3 Fault

The V&V process should be applied to every step of the software development and, between its main goals, it should discover the system faults and evaluate if the system is usable during a dynamic situation. There is a different kind of faults and they can be grouped in classes, accordingly to the different phases of the software development:

- Specifications faults:

The description of the software function could be ambiguous, contradictory, or just inaccurate.

- Design faults:

The components or the interaction between them are not suitable for the design of algorithms, data structures, and interface.

- Code faults:

Errors due to a bad implementation of the code, for a lack of understanding of the design or the syntax of the programming language.

- Test faults:

The test cases, the plans for tests, etc. can present faults.

2.2 Test

Definitions:

Test data: Input data were chosen to test the system.

Test case: Input data for the system and estimated output for these inputs in case the system works following its specifications.

Test suite: A collection of tests case.

The test of the software:

- Can show the presence of the errors, not their absence.
- Can be done together with the verification of the static code of the system.

2.2.3 Test Principles

1. Testing is the process of exercising the component using a specific set of test cases with the intent of detecting defects or evaluating the quality.
2. If the goal of a test is to find errors, then a good test case is the one with a good probability to find unknown errors.
3. The test results must be read carefully.
4. A test case should include the estimated output.
5. A test case should be implemented for both valid and not valid input conditions.
6. The probability to find additional errors for a software component is proportional to the number of errors already found
 - Errors are often found in groups.
 - A code too complex is a sign of a bad design.
7. The tests should be executed by a group of technicians independent of the developer group.

8. The tests shall be repeatable and reusable

- Regression tests.
- Unambiguously described.
- Objectively measurable outcome.
- In the description of a test case, there must be all the information a tester needs (human or automatic tester) allowing to evaluate the result of the test unambiguously.

9. The test plan should specify the goals, allocate the right time and resources for the project, and then supervise the outcomes.

10. Testing activities should be carried out throughout the test cycle.

2.3 Debugging

It consists of:

- Making assumptions about program behavior.
- Verify such assumptions and find errors.

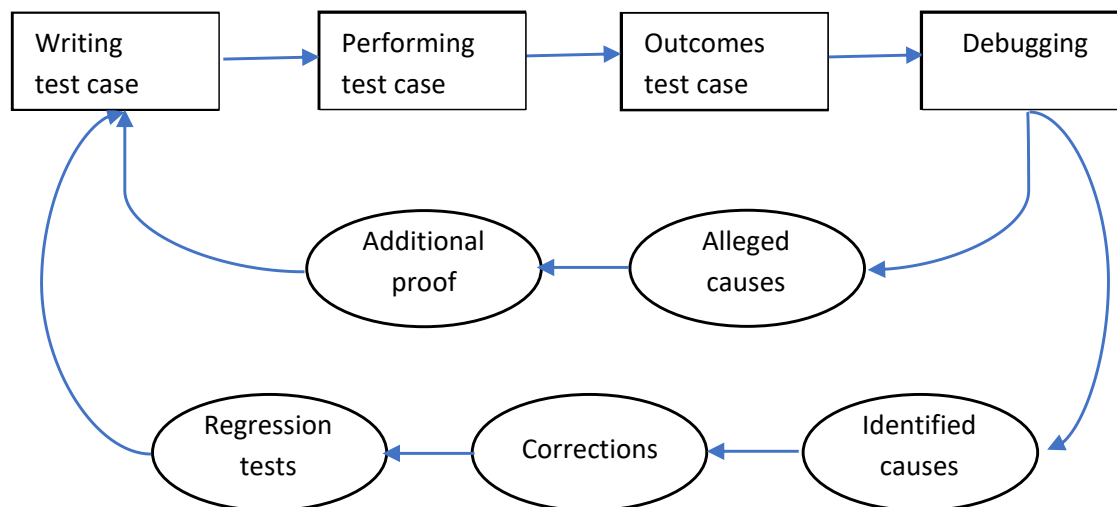


Figure 1: Generic scheme of a debugging process

The correction of an error can often provoke new ones: always apply the regression test, that is the control on pre-existing and already tested functionalities, to verify that modifications to the software have not compromised the quality.

2.4 Testing

The goal of testing is to establish the presence of errors in a system; it can often happen that a test causes the program to behave abnormally.

There are:

- Unit test
 - Testing of individual fragments (methods, classes, etc.).
 - Executed by the developer.
- Integration test
 - Testing of interacting components forming a system or subsystem.
 - Performed by a test team.
 - Tests based on specifications.

Only an exhaustive test can show if a program has no errors, but this is impractical, for example, due to limitations dictated by excessively long test times.

2.4.1 Testing Features

Effective:

Carried out through strategies that allow us to find as many errors as possible.

Efficient:

Able to find errors by testing as few tests as possible. About 40% of software production costs, to achieve reasonable quality levels, are required by testing.

Repeatable:

It may not be possible if the execution of the test case affects the execution environment without the ability to restore it or if in the software there are indeterministic elements.

The software testing process can be schematically represented:

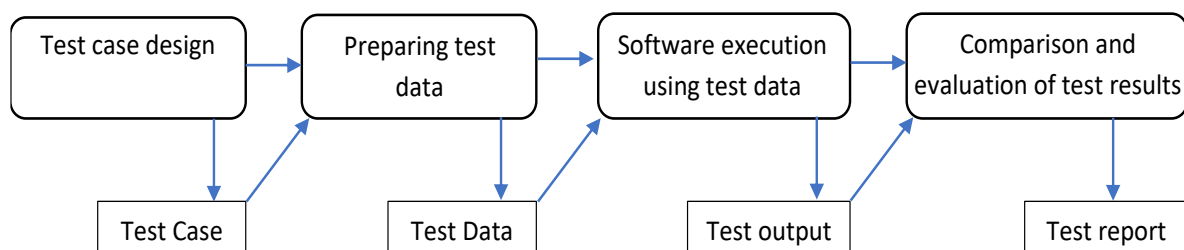


Figure 2: Generic scheme of a test process

2.5 Hardware-in-the-loop testing

From the model design flow, the Hardware-in-the-loop testing is particularly interesting in the automotive field, because it allows us to test a hardware device, such as the Electronic Control Unit of a vehicle, even without having available all the electronic system for which it is intended.

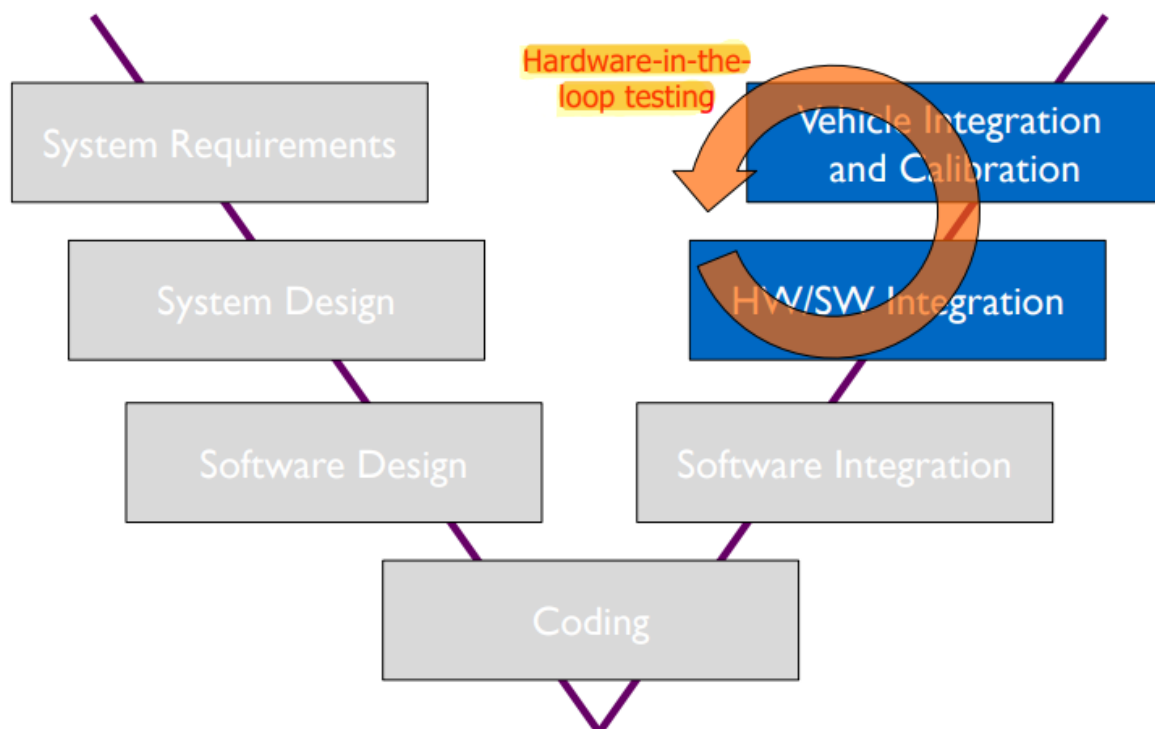


Figure 3: Hardware-in-the-loop scheme

These are particular testing techniques where part of the model runs in a real-time simulator (e.g. CANalyzer), and part may exist as physical hardware (e.g. Electronic Control Units).

2.6 Controller Area Network (CAN)

Invented by Robert Bosch in 1980 for automotive applications, their purpose is to allow simultaneous control of all electronic systems in the vehicle. The different Electronic Control Units, called “nodes”, are connected to one or more CAN bus. This allowed:

- To exchange information between different nodes.
- The realization of complex functionalities which would otherwise not be feasible.
- The simplification of wiring and test procedures.

- Greater flexibility in car configurations.
- Improving system comfort, quality, and reliability.
- Greater coverage of diagnostic functions.

Below is an example of a CAN network:

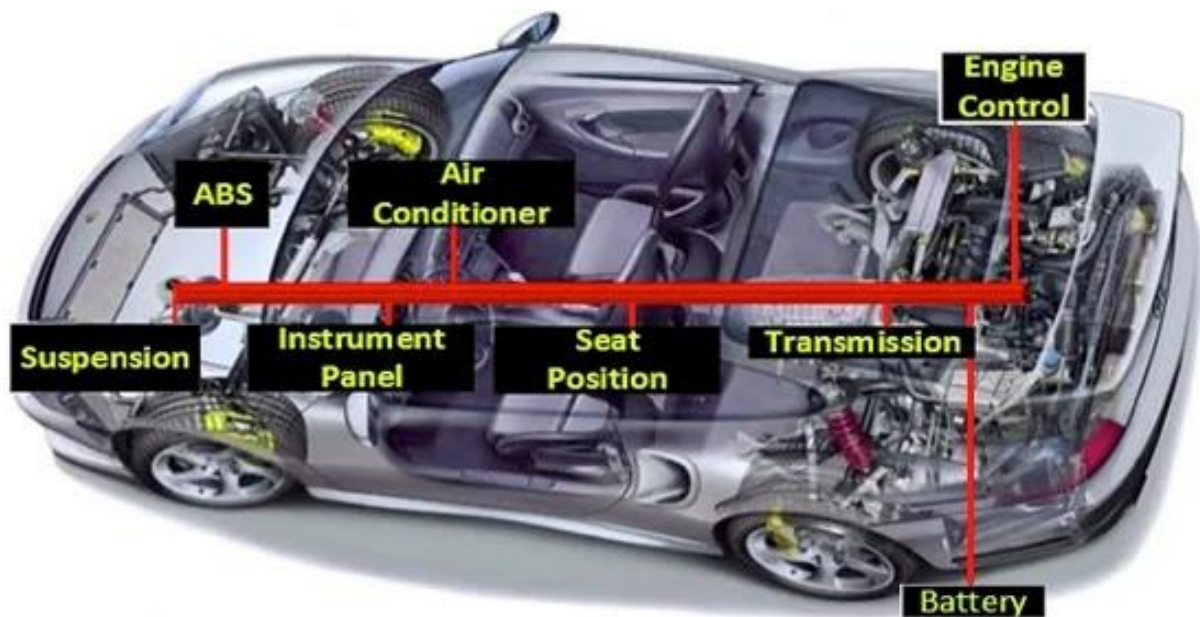


Figure 4: Example of CAN Network on a vehicle

2.6.1 CAN Network features:

- B-CAN protocol (ISO 11519)
 - Low-speed communication (from 50 to 125Kbit/s).
- C-CAN protocol (ISO 11898)
 - High-speed communication (up to 1Mbit/s).
- CAN messages identifiers
 - Standard: 11-bit identifiers.
 - Extended: 29-bit identifiers.
- Message identifier specifies contents and priority
 - The lowest message identifier has the highest priority.
 - Non-destructive arbitration system by CSMA with collision detection.
- Two-wire bus: CAN-H, CAN-L, terminated with a 120-Ohm resistance.

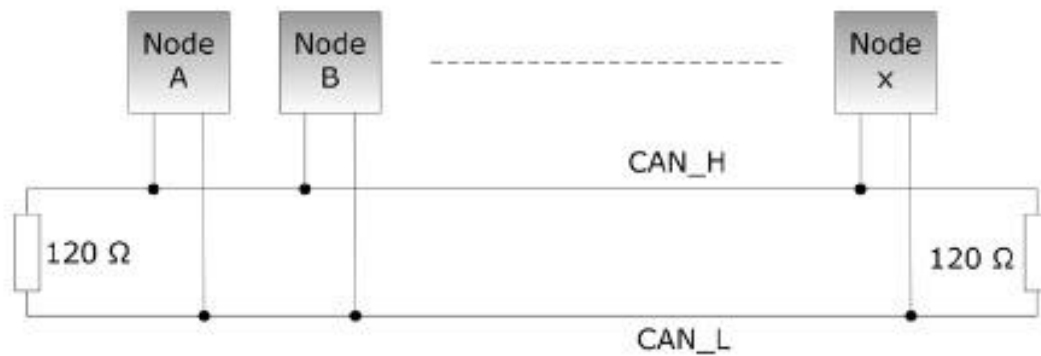


Figure 5: CAN bus with 120 Ohm termination resistance

- Two logic state for the CAN bus: the “zero” level is dominant, the “one” level is recessive

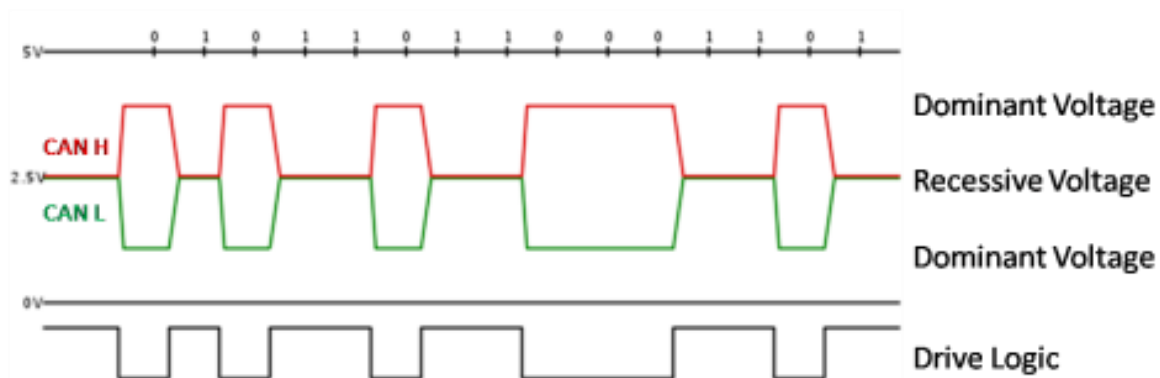


Figure 6: E.g. of Dominant (“0” level) and Recessive (“1” level) states for the CAN bus

The CAN bus is an Asynchronous Serial Bus. A mean of synchronizing the node is necessary as each node at a particular time must be able to see both its transmitted data and the other nodes transmitted data at the same time. The Synchronization starts with a hard synchronization on the first recessive-to-dominant transition after a period of bus idle (the start bit). Then, a resynchronization occurs on every recessive-to-dominant transition during the frame. When it does not happen, this is forced by adding a Stuff Bit (a bit with the inverse level of the previous one) after five consecutive bits of the same level.

3 Materials and Methods

3.1 Instrumentation

As mentioned in the introduction, to minimize costs and thus optimizing the resources, hardware already present in the company is used.

- Bench-stabilized power supply:



Figure 7: A stabilized power supply was used to power the whole system

- Breakout Box :



Figure 8: Home-made Breakout Box used to interface the CANcaseXL with the Instrument Panel Cluster (IPC)

- Vector CANcaseXL:



Figure 9: The hardware used to interface the Electronic Control Units simulated through the CANalyzer software, with the Instrument Panel Cluster (IPC)

- Instrument Panel Cluster (IPC):



Figure 10: For industrial secrecy problems, it is shown an older version of the IPC effectively used in the project

Example:

For demonstrative purposes, it is represented to the side, the system constituted from IPC, Breakout box, and CANcaseXL. The Instrument Panel in the image does not correspond, for industrial secrecy, to the one used for the project.



Figure 11: System composed of IPC, Breakout box, CANcaseXL

- Camera:



Figure 12: The camera used for the project has a maximum resolution of 720p and 30fps

No new software licenses were purchased for the thesis project. The software programs are the same used by the technicians of the company:

- The Vector program called CANalyzer was able to simulate the Electronic Control Units of the vehicle, useful to allow the proper functioning of the cluster.
- The CAPL Browser (tool integrated with CANalyzer) was the programming environment for the CAPL programming language.
- PyCharm was the programming environment for Python programming language.

3.2 Test bench scheme

A graphic representation of the whole test rig is reported below:

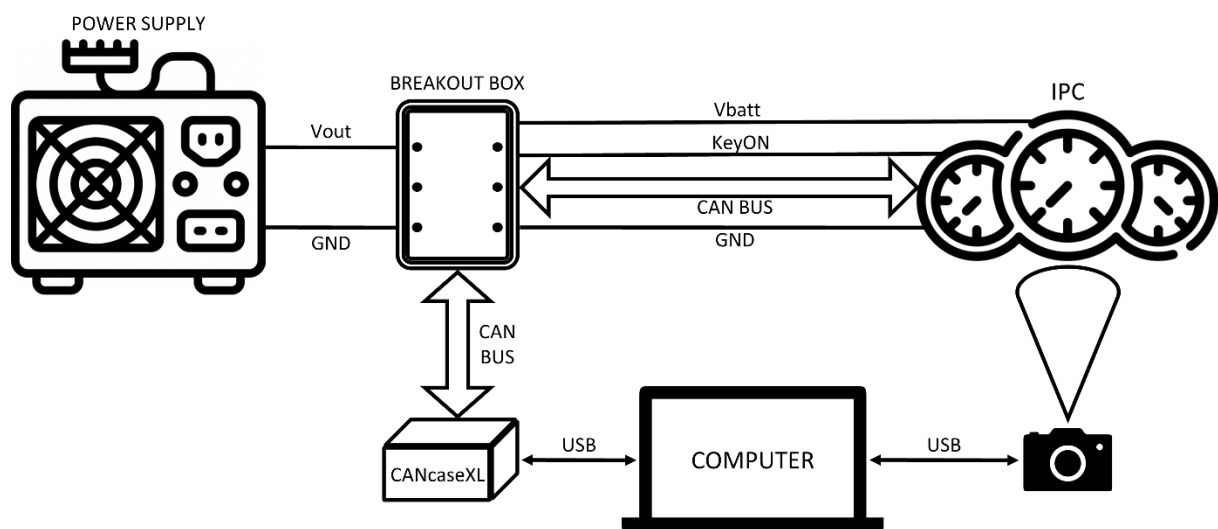


Figure 13: Test bench scheme

3.3 Implementation

3.3.1 Observation and study of manual or semi-automatic tests

To achieve a better understanding of the path to be taken for the project, I started by supporting the technicians during the various manual testing processes carried out within the validation area of the company. In particular, it is possible to divide the IPC test process into four main phases:

1. System Requirements:

Macroscopic study of System functionality, which must then be described in detail into Vehicle Functions.

2. System/Functional requirements analysis:

Functional requirements describe the implementation of specific vehicle functions (VF) of the Electronic Control Unit. In particular, related to the correct execution of a function, it specifies which signals will need to be transmitted and according to which algorithms. On the VF there will also be described any initial configuration parameters of the IPC (EOL calibration parameters), which must be set before starting communication with the CANalyzer software.

3. Testing:

The practical execution of the test can begin, activating the chosen functions through specific signals and verifying that the expected behavior respects the real one. For the Testing phase, specific test-cases are written previously by the test engineer himself or assigned to him by the client who commissioned the work.

4. Test-Report:

Any success or failure of a test shall be noted. In particular, failure cases shall be reported with a detailed description: the type of problem encountered, a list of signals transmitted or received when the problem occurs, the configuration parameters used, and possibly, some multimedia data (video, photo, audio) as records of the problem encountered. In this way, the error can be easily repeated and corrected by the software developers.

EXAMPLE:

“HAND BRAKE MANAGEMENT”

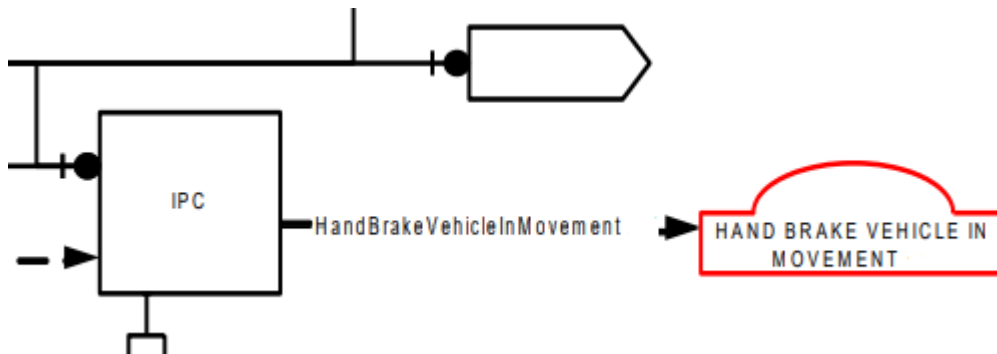


Figure 14: Example of a scheme extracted from the Vehicle Function description

The VF under analysis deals with the management of the “hand brake in motion” function. In particular, in that condition, it is intended to verify that the following warning lamp is displayed on the Instrument Panel:



Figure 15: Hand brake warning lamp

The specification also requires an audible warning (beep) and no text message to be displayed. Finally, it is required that the functionality of this VF is verified in three different time moments and, for each of them, it follows three different behaviors:

Key-on	Turn on: -synchronized with key-on - driven by the cluster
Steady-state	Turn on/Turn off: - driven by the signal
Key-off	Turn off: -synchronized with key-off

Figure 16: A test case example

“key-on”, “Steady-state” and “Key-off” represent the three key-states (turn-on, idle, turn-off) of the IPC, while the right column describes the behavior of the studied functionality for each different key status.

3.3.2 Project Design

After participating actively in some tests, I immediately realized the time limits imposed by the manual performance of some operations. It is easy to find yourself having to perform a test that requires relatively repetitive and tedious actions to be completed. From problems of this kind and the idea of a possible greater profit, was born the need to automate. Before the actual implementation of the project, I started, with the R&D team of the company, a first brainstorming phase, whose result was to develop a scheme that can describe synthetically and intuitively which logical and operational flow to follow.

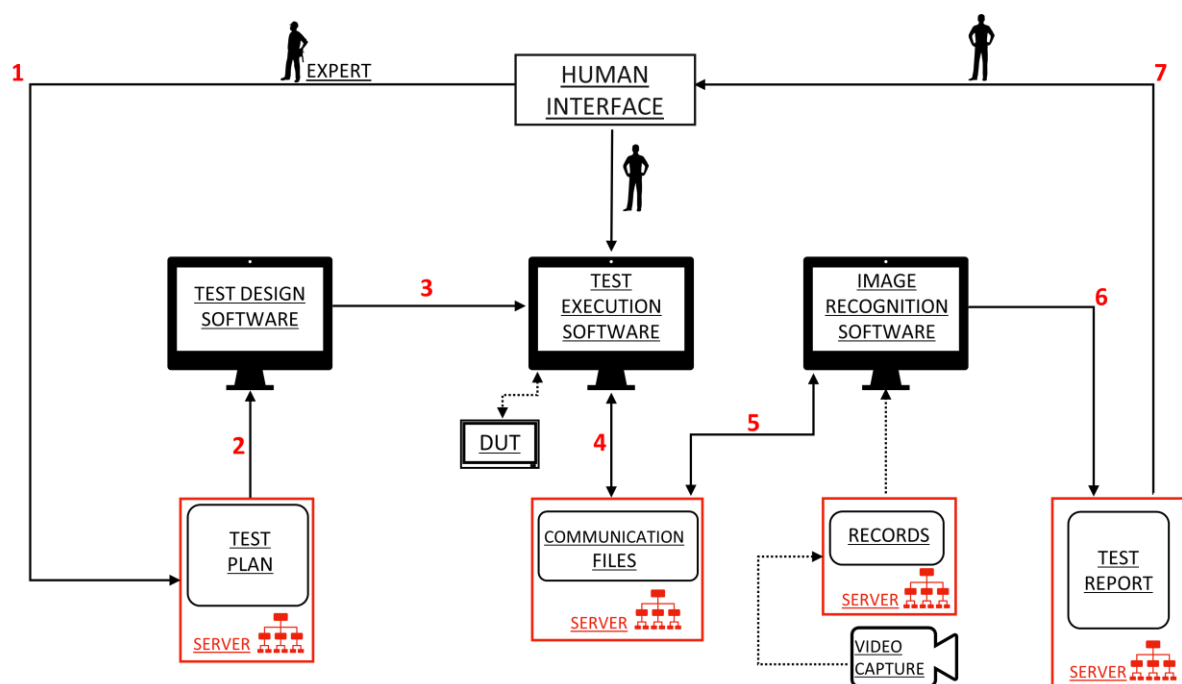


Figure 17: Original project scheme. The elements connected by dotted arrows in the graph represent Inputs necessary for the proper functioning of the system

1. A technician is in charge of writing the Test Plan and preparing the configuration files necessary for the automatic test to be started. The documentation produced is then saved to be made accessible at any time.
2. When the technician decides to start the test, the configuration files are automatically uploaded and opened.
3. The test, with the configuration files already present, is started automatically with the user intervention (no technical expertise is needed).
4. The test execution software saves on files the test progress and reads information about the image recognition software status.
5. The image recognition software saves on files its status and reads information about the test execution software progress.

6. The image recognition software compares the expected behavior of the device under test (DUT) with the observed one (real-time frames, of the device under test, captured by the camera). The right behavior of the DUT was defined through configuration files before the test started.
7. A Test report is made available for the final user (e.g. the tester wants to know the outcome of the test: "Passed" or "Failed").

The project implementation was based on this concept. During the realization, The test data and the test outcomes, have been saved locally on the computer memory, but it is easy to imagine, as shown in the figure, that they can be made usable through servers, for greater accessibility.

3.3.3 Configuration and Procedure

For the thesis project, I decided to automate a specific test case, the one related to the management of calls, in which the contribution given by automation is evident.

Before it is possible to manage a calling routine on the IPC, it is necessary that the whole vehicle electronic system (then the other Electronic Control Units onboard the vehicle) interacting with the IPC, is simulated through the CANalyzer program. The signals used for this project are exchanged, through the vehicle, following the CAN protocol. In addition to the B-CAN protocol, mentioned in the chapter on the theoretical background, in the specific case of the IPC used for the project, it is necessary to define a new channel with a higher transmission rate than B-CAN, but lower than C-CAN. So, two different channels type are used:

- CAN_BH_Vehicle: for lower speed transmission.
- CAN_C_Vehicle: for higher speed transmission.

The Vector hardware (Figure 9) used to interface the IPC with the CANalyzer software present the possibility to use two configurable channels, as shown in a more detailed figure of a similar product:



Figure 18: Vector interface hardware with two/four configurable channels

It is, therefore, necessary to assign the desired communication protocol to each channel, before it is possible to start the transmission of the signals

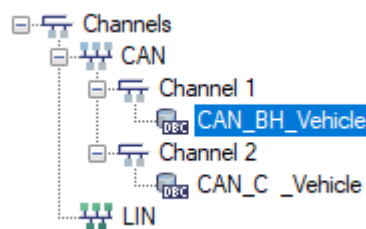


Figure 19: CAN communication channels

We are interested in the message `TELEMATIC_DISPLAY_INFO`, which travels on the channel highlighted in Figure 19 (`CAN_BH_Vehicle`). This message contains the signals useful for configuring the displayed text message on the HMI (Human Machine Interface) of the IPC whenever one or more calls are managed.

Name	Message
TotalFrameNumber	TELEMATIC_DISPLA...
InfoCode	TELEMATIC_DISPLA...
FrameNumber	TELEMATIC_DISPLA...
UTF_Text_1	TELEMATIC_DISPLA...
UTF_Text_2	TELEMATIC_DISPLA...
UTF_Text_3	TELEMATIC_DISPLA...

Figure 20: Signals contained in the `TELEMATIC_DISPLAY_INFO` message

To automate the creation of pop-up messages on the IPC, it was essential to understand how this was handled manually or semi-automatically by the technicians in charge of validation.

They assign specific values, through the CANalyzer program, to the signals contained in the TELEMATIC_DISPLAY_INFO message:

- *TotalFrameNumber:*

This represents the total number of frames to send for a multi-frame message. For the case under analysis, messages with a single frame are used to show: "Connected Phone" or "Disconnected Phone" on the IPC. Multi-frame Messages are used instead to show: "Call in progress...", "Call Pending" and "Call Terminated". This is because the last three messages are displayed together with the name of the person called, and more frames are needed to compose that name. In particular, each sent frame can contain up to three characters.

EXAMPLE:

Name	Characters number	Needed frames
ENE A	4	2
RAFFAELE	8	3

Table 1: Example of multi-frame message use

- *FrameNumber:*

Represents the number of the current frame under analysis, compared to the total number of frames present.

- *InfoCode*:

It represents the code, expressed in binary, of the text message that you want to display. For the project, the following values have been taken into consideration for the *InfoCode* signal

Binary value	Decimal value	Displayed message
010010	18	Telefono connesso
010011	19	Telefono disconnesso
010101	21	Chiamata in corso...
010111	23	Chiamata in attesa
011000	24	Chiamata terminata
010001	17	"Clear Display"

Table 2: *InfoCode* signal values interesting for the project

- *UTF_Text_1, UTF_Text_2, UTF_Text_3*:

It represents the package of three characters in UTF-16 encoding that is sent with each message *TELEMATIC_DISPLAY_INFO*. It is chosen to show the name "ENEA" on the pop-up related to the call notice. To do this, after assigning the right value to each signal, two messages of the type *TELEMATIC_DISPLAY_INFO* are sent in quick succession:

The first frame, with signals values used to display the message "Chiamata in corso..." (*InfoCode*: 010101 = 21) contains the characters "E" (*UTF_Text_1* = 69), "N" (*UTF_Text_2* = 78), "E" (*UTF_Text_3* = 69)

Signal Name	Value
<i>FrameNumber</i>	1
<i>InfoCode</i>	21
<i>TotalFrameNumber</i>	2
<i>UTF_Text_1</i>	69
<i>UTF_Text_2</i>	78
<i>UTF_Text_3</i>	69

Table 3: First frame, containing the three characters "E", "N", "E", to form the name "ENEA"

Then the second frame is sent quickly. It is also related to the message “Chiamata in corso...” (*InfoCode*: 010101 = 21) and contains the characters “A” (*UTF_Text_1* = 65), “ ” (*UTF_Text_2* = 32), “ ” (*UTF_Text_3* = 32).

Signal Name	Value
<i>FrameNumber</i>	2
<i>InfoCode</i>	21
<i>TotalFrameNumber</i>	2
<i>UTF_Text_1</i>	65
<i>UTF_Text_2</i>	32
<i>UTF_Text_3</i>	32

Table 4: Second frame, containing the last three characters "A", " ", " ", to form the name "ENEA"

This example is enough to give an idea of how cumbersome and repetitive the work of the Tester could be to verify that the message `TELEMATIC_DISPLAY_INFO` is correctly sent.

The scheme of the whole CAN network represented on CANalyzer is shown in the figure below, to better understand it:

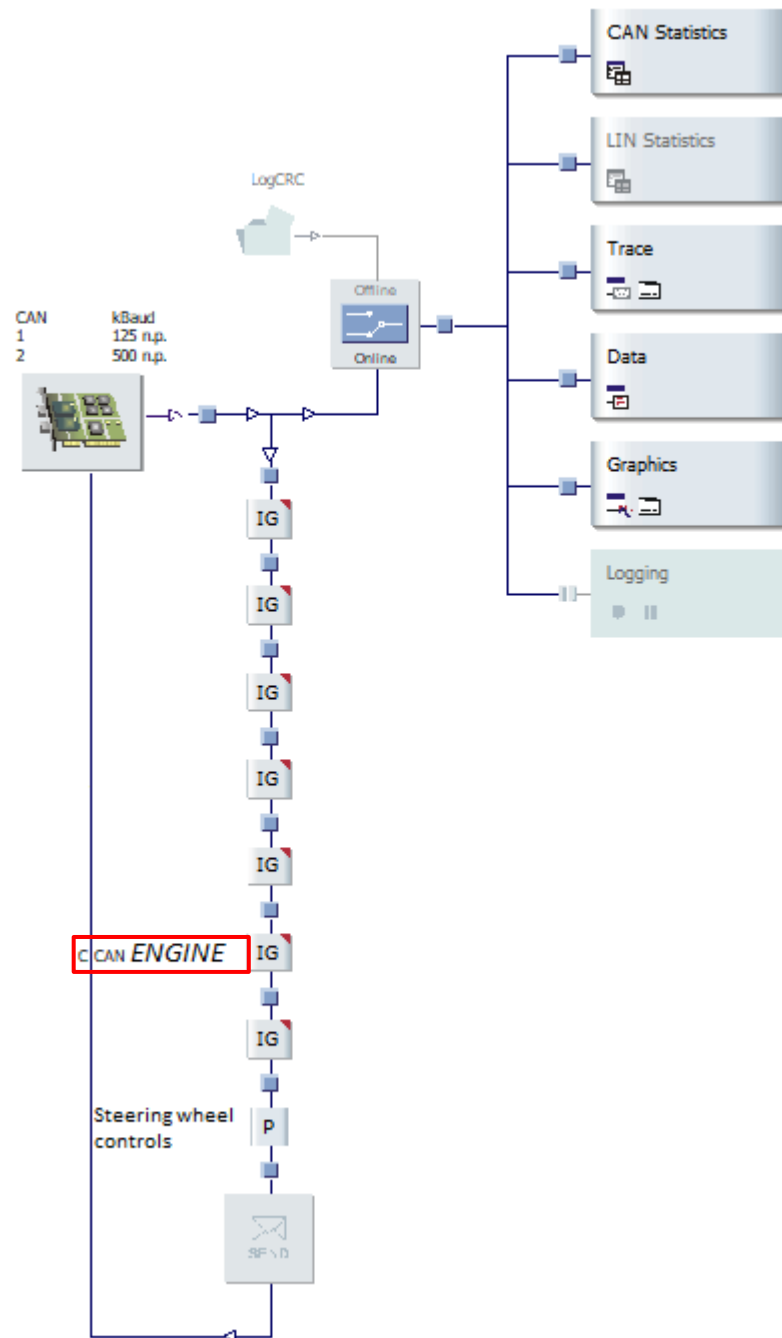


Figure 21: CAN network representation on CANalyzer

The messages of the Electronic Control Units of the vehicle simulated by CANalyzer are collected into more Interactive Generator Blocks (IG). Entering in one of them, a technician can send the desired signals to simulate the activation of a specific function of an ECU. Then, he can look at the IPC and verify if the function under test works or not.

EXAMPLE:

Entering in the IG block called "ENGINE" (Figure 23) it is possible to simulate engine-on and engine-off conditions. It is sufficient to modify the value of the signal "OperationalModeSts" contained in the message "CAN_C_Vehicle::ENGINE_STATUS" as shown in the following image:

	Message Name	Msg Params		Triggering		Data Field							
		Channel	DLC	Send	Cycle Time [ms]	0	1	2	3	4	5	6	7
>	CAN_C1_Vehicle::ENGINE_STATUS	CAN 2	8	now	10	80	0	0	0	0	8	0	0

27	0	0	-	-	1	+	None
60	OperationalModeSts	8	Ignition_On_EngOn	-	1	+	None
24	0	Default	-	1	+	None	

Figure 22: Example of the message contained in an IG block

Every simulated ECU has its IG block to which is possible to enter to manage its signals. Only for the steering wheel controls, a graphic panel is created to make easier the interaction with the IPC:

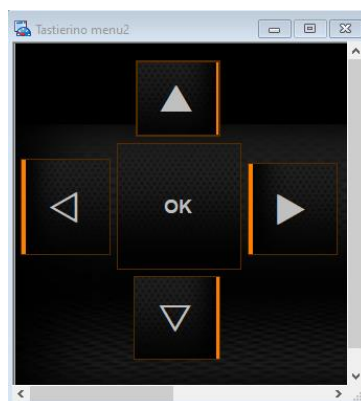


Figure 23: Steering wheel controls simulation

This is the first example of a Program Node (P) use. To create that graphic panel, CAPL (CAN Access Programming Language) is used. It is a C-based programming language integrated with *CANalyzer*. Using it, one can create a custom application for user-defined behavior. The Program node contains the CAPL code to run simultaneously with the other IG blocks. The next step to automate testing was then to plan the automatic sending of messages, programming in CAPL.

The CAN network represented on CANalyzer, therefore, has a different form. All the messages that activate the functions of the various ECUs are managed by a single Program Node (P), entirely programmed in CAPL. No more IG blocks are needed:

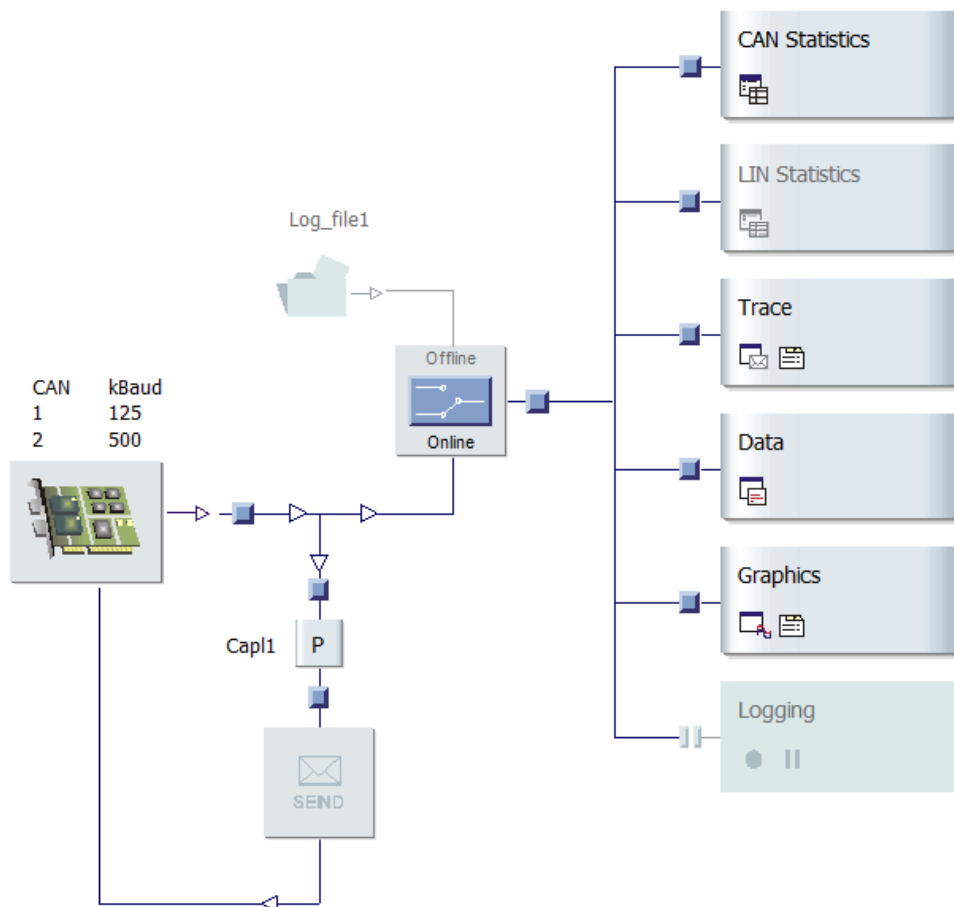


Figure 24: CAN Network managed with CAPL and simulated on CANalyzer

It is used “CAPL Browser” (tool integrated with CANalyzer) as a programming environment for writing CAPL code. Just like with C programming language, first of all, have been declared and initialized the variables that will be used, as in the example below:

```

77 | Timer delay3;
78 | Timer ClearPhone;
79 | //file variables
80 | dword handle;
81 | dword handle2;
82 | dword handle3;
83 | char buffer [64];
84 | char buffer_2 [64];
85 | char buffer_3 [64];
86 | long randomValue;
87 | int msg_arrived;

```

Figure 25: Declaration of different type variables

Although the project, aimed at implementing an automatic test for the call management system, needs to work with a limited number of signals, many other variables need to be

initialized at the start of the program. For example, some messages from other ECUs simulated by Canalyzer need to be sent cyclically, so that the IPC does not recognize any kind of error caused by critical signals missing. This is to prevent alert messages from being displayed on the HMI of the IPC when testing other features.

```
139 on Start
140 {
141     setTimer(tengine,10);
142     setTimer(tbody,10);
143     setTimer(tinfo,1000);
144     setTimer(tbrake,10);
```

Figure 26: Initialization of cyclical messages, important for the proper functioning of the IPC at startup

During this phase of initialization are also set the parameters that we want to assign to the system. For example, it is imposed, at the start of the automatic simulation, the "key-off" configuration for the IPC. It means that the engine and the instrument panel of the vehicle are both turned off. This is done by setting to "2" the value of the "ENGINE_STATUS" signal into the "ENGINE" message.

```
161 //Initial key status
162 ENGINE.ENGINE_STATUS = @Engine::EngineStatus;
163 ENGINE.ENGINE_STATUS = 2;
```

Figure 27: Initialization of key-status

CAPL software can be created to respond to:

- Time events (timers).
- I/O events (keyboard, parallel, or series ports).
- CAN communication events (messages, errors).

It is now possible to pass to the description of the routines that must be performed as a result of a triggering event or after calling a timer. This can be invoked cyclically

```
211 //START TIMER
212 on timer tengine
213 {
214     output(ENGINE);
215     setTimer(tengine,10);
216 }
```

Figure 28: Cyclic timer

Or because it is called by another function. These two types of timers are both used to develop the algorithm of this thesis project.

The next step was to create new System Variables (via configuration dialog into the Analyzer interface) and collect them within the Namespace “Telematic_Display”:

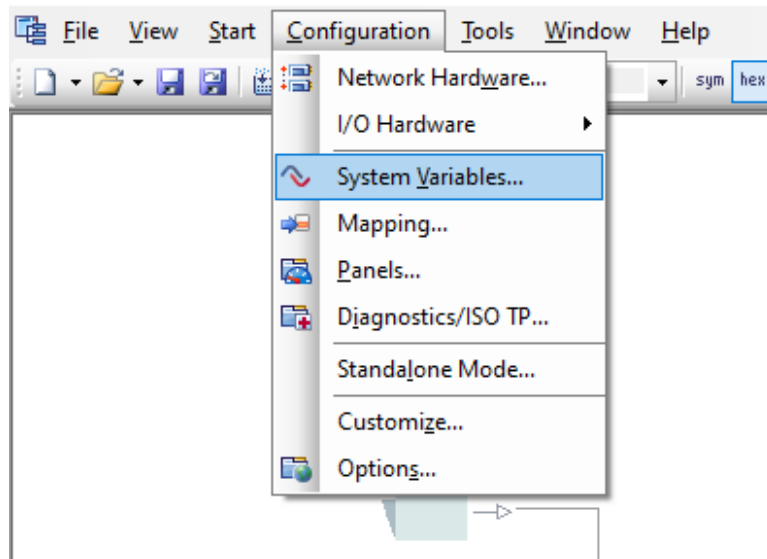


Figure 29: Configuration panel inside CANalyzer

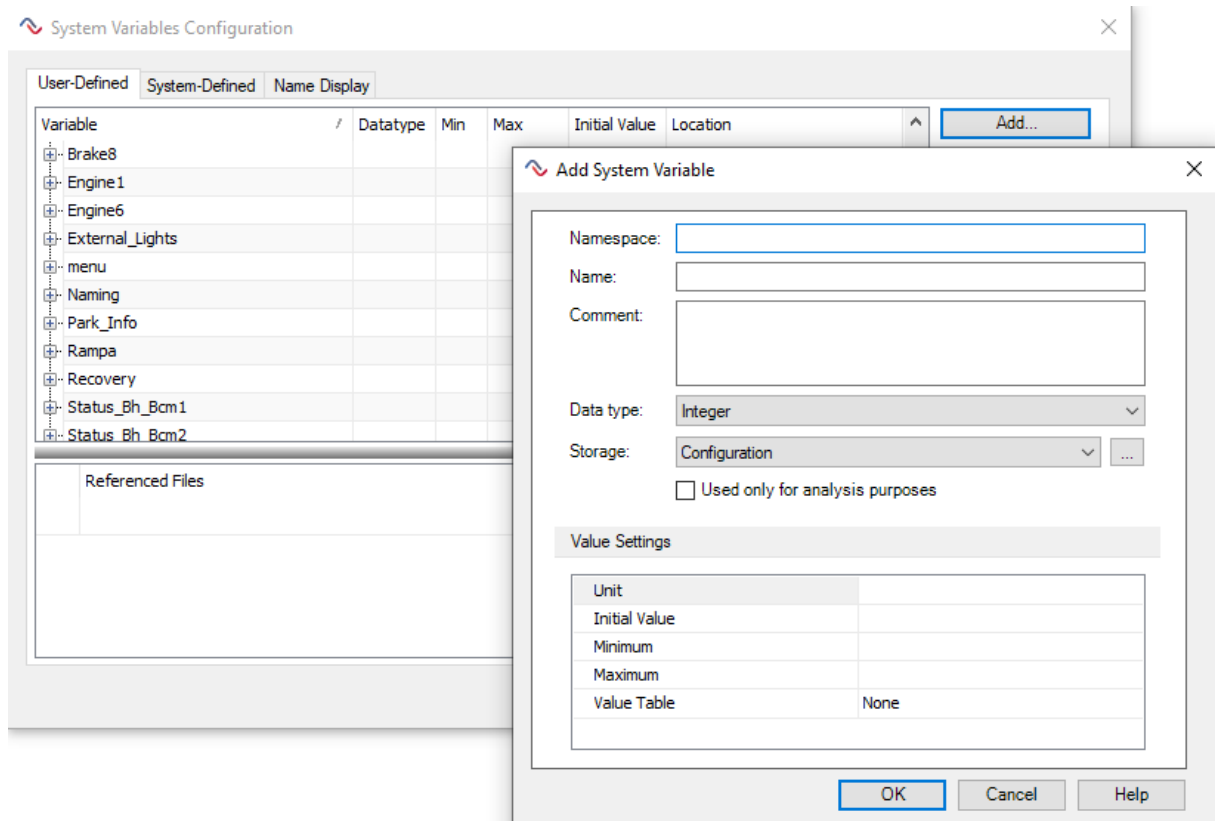


Figure 30: Dialog window to create new System Variables

Specifically for the calling routine, it was important to create system variables linked to the signals contained in the message `TELEMATIC_DISPLAY_INFO` (in *Figure 20*). In this case, to make everything more intuitive, the names of the signals are adopted for the system variables ones:

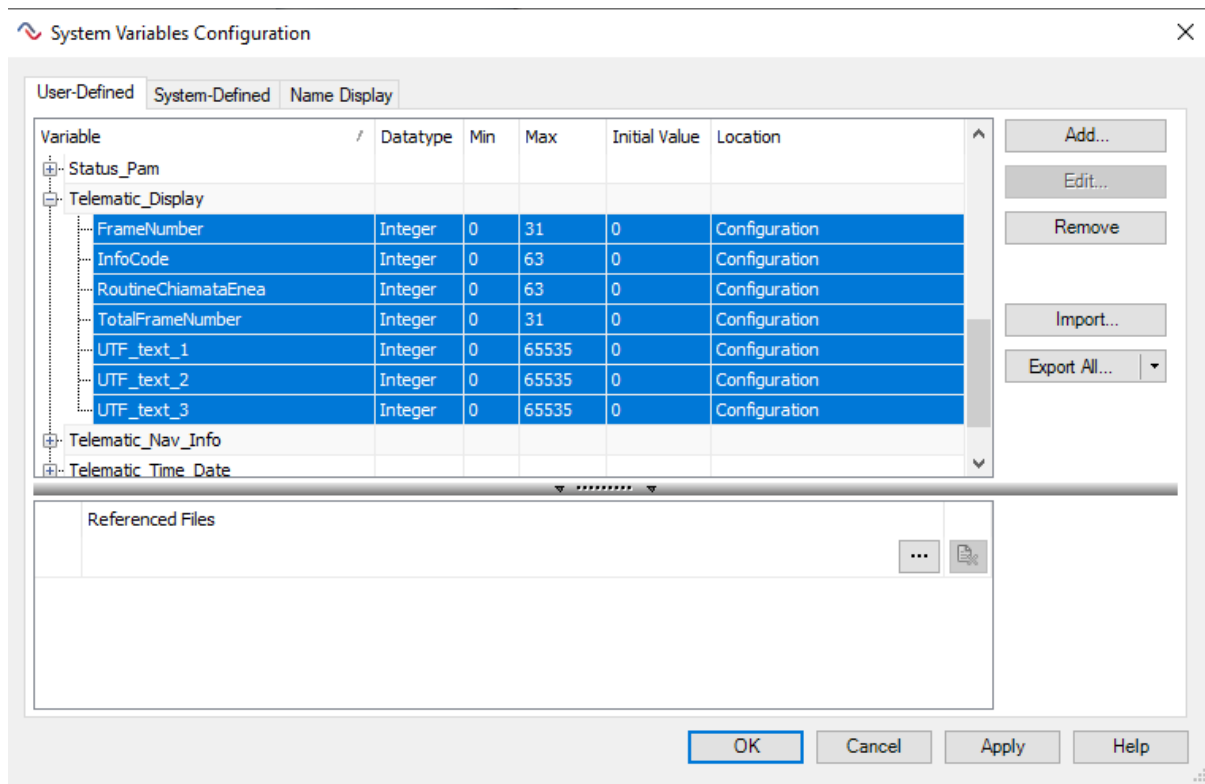


Figure 31: System Variables customization

At this point, through “Panel Designer” (tool integrated with CANalyzer), is possible to design a graphic panel and each created System Variable can be associated with a specific Input/Output box (e.g. button) to use in real-time during the simulation. In this way, by changing the value of each System Variable, it is possible to activate a customized function. I decided to create a graphic panel with a single button, which intuitively starts the automatic test I programmed before:

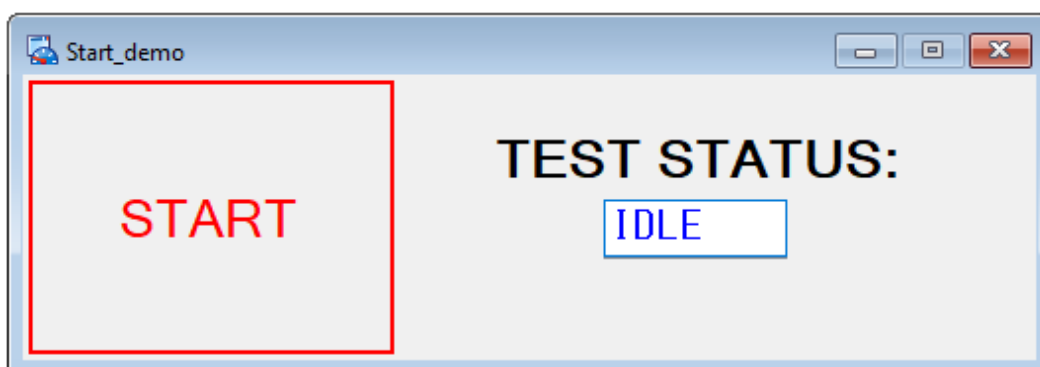


Figure 32: Graphic panel with the button to start the simulation

Only the previously created System Variable “RoutineChiamataEnea” is used. It is assigned to the "START" button as shown in the figure below. It is responsible for starting the whole test process.

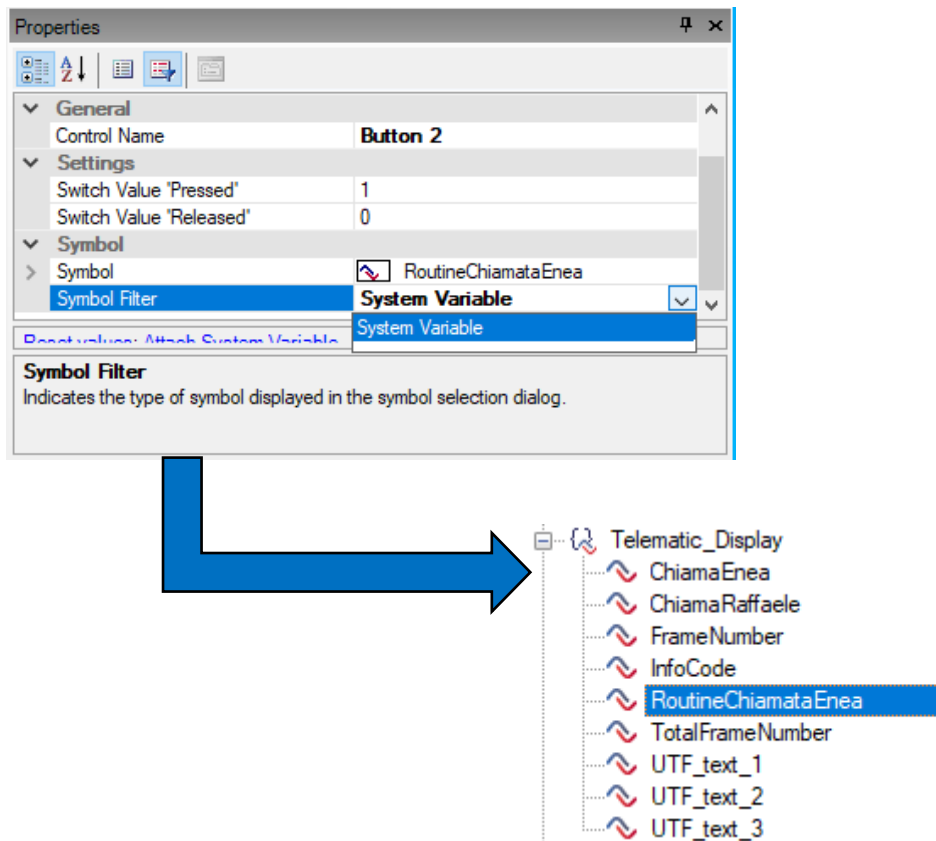


Figure 33: System Variable assignment

The instruction routine is defined when the user decides to press the "START" button:

- The file directory in which is contained the communication file is defined.
- The communication file (a text file) is opened to be written.
- In the text file is written "start process" to warn the Python software that the test is started.
- The text file is closed.
- The key-status of the engine is passed to “key-on”.

- The signals responsible to show the message “Telefono connesso” on the IPC are set as follow:
 - *InfoCode* = 18.
 - *FrameNumber* = 0.
 - *TotalFrameNumber* = 0.

After a delay of two seconds, a timer is evoked. It shall carry out the following instructions:

- The file directory in which is contained the new communication file is defined.
- The communication file is opened to be read.
- If the message “PythonReady” is present, the next instruction can be executed, otherwise, the communication file continues to be read cyclically waiting for the Python program to be ready.
- A new timer is evoked.

The new timer simply sends the `TELEMATIC_DISPLAY_INFO` message to the IPC to show “Telefono connesso” (the message previously set) on the display and then calls a series of new timers, all with similar routine instructions:

- The values of the signals belonging to `TELEMATIC_DISPLAY_INFO` are set:
 - *InfoCode*.
 - *FrameNumber*.
 - *TotalFrameNumber*.
 - *UTF_Text_1*, *UTF_Text_2*, *UTF_Text_3*.
- Then the `TELEMATIC_DISPALY_INFO` message is sent to the IPC.
- The software waits a certain time before calling the next timer.

Only the last timer has a different routine

- The values of the signals belonging to `TELEMATIC_DISPLAY_INFO` are set to clear the display, In particular: *InfoCode* = 17.

- The message “Start process” in the communication file is deleted.
- The TELEMATIC_DISPLAY_INFO info is sent to clear the HMI of the IPC.

To have a graphic outcome of the test as shown in *Figure 3*, a System Variable called “Test_Result” is created. This is associated with a drop-down window in which three possible choices can be selected: “IDLE”, “PASSED” and “FAILED”. The first outcome (“IDLE”) is the default one (box 0 of the drop-down window) and it is shown every time a new test is started. Then another timer has to be evoked at a certain point during the execution of the automatic test. I preferred to do it right after the message “PythonReady” is read by CAPL software. The tasks of this timer are:

- Open a file in which Python software saved the outcome of the test: “Failed” or “Passed”.
- Read cyclically the content of the test result.
- Depending on the test result, a number corresponding to the box of the drop-down window is chosen:
 - Number 1 if the test is “PASSED”.
 - Number 2 if the test is “FAILED”.
- The chosen number is saved as the value of the “Test_Result” System Variable.

As regards the Python software, as mentioned above, it is written to be able to perform image recognition and compare the result obtained with the expected one. to obtain these results, template images are created first. Then, they are used by the camera for comparisons. Through a short program in Python:

- The camera connected to the computer via USB port is activated.
- Frames used as template images are saved in a folder.



Figure 34: Template images used for comparisons by the image recognition software

It is now possible to describe the code responsible for image recognition and comparison. The first step was to import useful libraries. They contain functions created by other programmers and essential to run the program for this thesis project. Then it is possible to describe the main steps of the software algorithm:

- The text file used to communicate with CAPL is open and its content is read.
- The Python program controls if “start process” is read in the text file, as it was a flag, otherwise, it repeats the check until the outcome is positive. This choice was made because the basic design idea was that the whole image recognition program would be activated only once at the beginning. It automatically manages image recognition only when it is needed to be used.
- Then the camera is switched on.
- All the variables needed later by the program are declared.
- A text file to communicate with CAPL is opened and “PythonReady” is written, as a flag.
- Every captured frame of the Instrument Panel is saved in a variable (the frames are overwritten in the same variable at each iteration).
- Some changes are made to the saved image (e.g. black and white conversion) so that it can be used by the image recognition function.
- The function responsible for image recognition checks whether the template saved during the program configuration, is present inside the saved frame or not. It can do it by checking if the saved image (template) is contained, pixel by pixel, in the current frame, with a percentage higher than a threshold established by the programmer. The higher the threshold, the more precise the recognition of a specific template will be. In the case of positive feedback, a yellow rectangle is drawn around the template recognized in the IPC frame. The type of template matching operation used is the “TM_COEFF_NORMED”.

```
Result = cv2.matchTemplate(img_gray, template, cv2.TM_CCORR_NORMED)
loc = np.where(Result >= threshold)
for pt in zip(*loc[::-1]):
    cv2.rectangle(img_bgr, pt, (pt[0] + w, pt[1] + h), (0, 255, 255), 2)
```

Figure 35: Python function responsible for image recognition

- Some variables are used as flags to indicate which of the templates is recognized each time all the comparison operations on a single frame are performed. For example, if the “Template_1” is recognized, then the variable “Rec_Template_1” can be set to a specific value. Doing this, during the next comparison between the template and the captured frame, the “Template_1” will be ignored.
- For each recognized template, the same operations are executed:
 - A log file is opened in which the test results will be saved.
 - It is written: the data, the time, the outcome of the executed test, and the name of the recognized frame (e.g. frame_detected_2).
 - In a new folder are saved the images of the frames with their templates found (the whole captured frame of the IPC is saved, but there will be a yellow rectangle around the recognized template).
 - “PythonReady” on the communication file is deleted to have a clean text file at the next program execution.
 - The flag variable, used to avoid comparing the already recognized template during the next iteration, is declared.

The failed test recognition is managed as follow:

- Generic test failed:

If, after a defined time (e.g. 30 sec.), one or more templates are not recognized, the log file is opened, and the date and time of the executed test are added, together with the negative result (Failed). This always happens, whether there are one or more missing templates.

- Specific template missed:

If, after a defined time (e.g. 30sec.), a specific template is not recognized, in the log file are saved the date and time of the executed test, and the name of the unrecognized frame (e.g. Frame_not_detected_2).

4. Results

The following are the results obtained from the tests. For further clarity they are arranged in order:

- Results of automatic tests obtained by programming in CAPL.
- Results of image recognition software by programming in Python.

4.1 Automatic tests Results

After starting the test through the button “START” on the graphic panel, the results given by the execution of the CAPL code are:

- “start process” is written in the text file used to communicate with Python software:

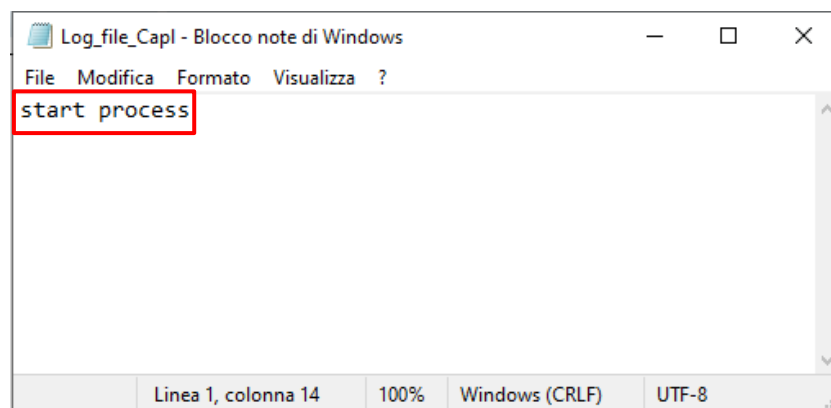


Figure 36: Text file used to communicate with Python the beginning of the execution of the automatic test

- The test is started and all the pop-up messages of the calling routine are shown, sequentially and after a set delay from each other, on the HMI of the IPC:

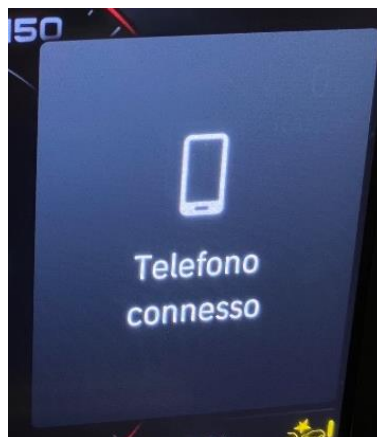


Figure 37: First popup. It simulates the connection of the mobile phone to the infotainment services of the vehicle. InfoCode = 010010. It consists of a single frame



Figure 38: Second popup. It simulates the outgoing call to the user "Enea". InfoCode = 010101. It consists of two frames



Figure 39: Third popup. It simulates the pending call to the user "Enea". InfoCode = 010111. It consists of two frames

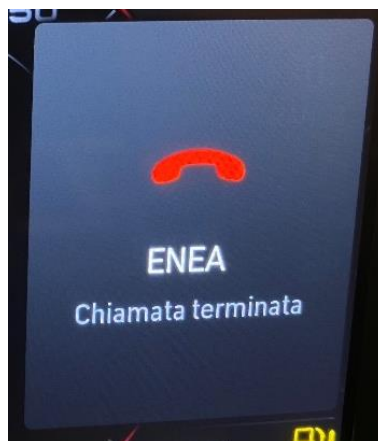


Figure 40: Fourth popup. It simulates the terminated call to the user "Enea". InfoCode = 011000. It consists of two frames



Figure 41: Fifth popup. It simulates the disconnection of the mobile phone to the infotainment services of the vehicle. InfoCode = 011000. It consists of a single frame

- At the end of the test, the graphic panel shows intuitively the result (past or failed), such that the user performing the test can receive feedback that does not require to be interpreted:

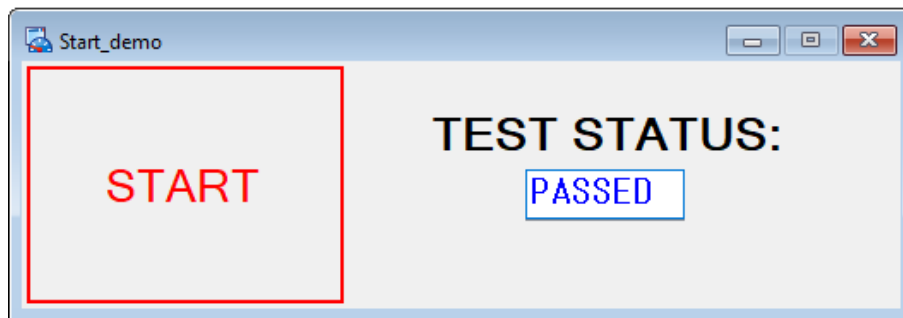


Figure 42: Passed test status on the graphic panel

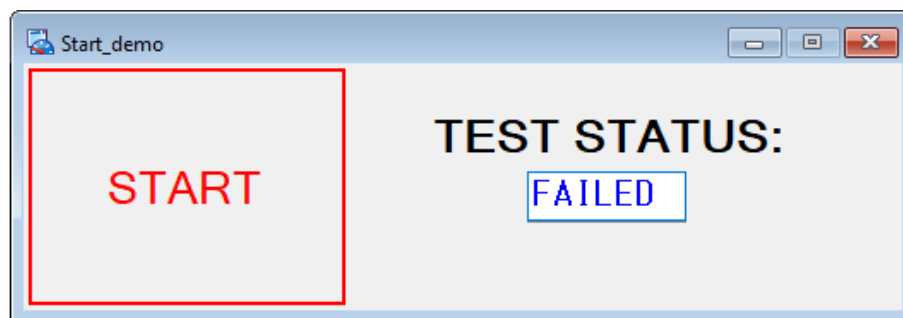


Figure 43: Failed test status on the graphic panel

4.2 Image Recognition Results

- “PythonReady” is written in the text file used to communicate with CAPL software:

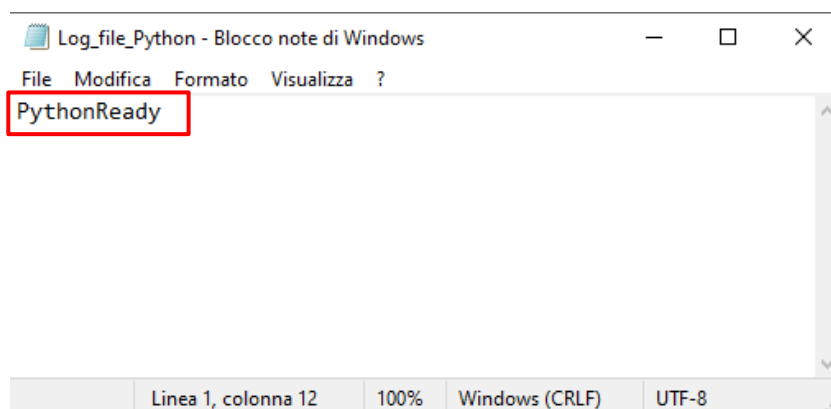


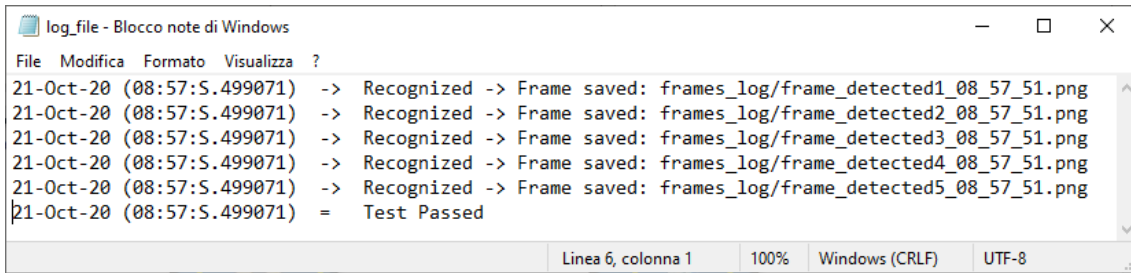
Figure 44: Text file used to communicate with CAPL the beginning of the execution of image recognition software

- During the execution of the Python software, anytime a template is recognized, the current frame captured by the camera is saved in a specific folder. The name of each image is formed by the number of the recognized template, plus the time (“hour_minutes_seconds”) in which the test is executed:



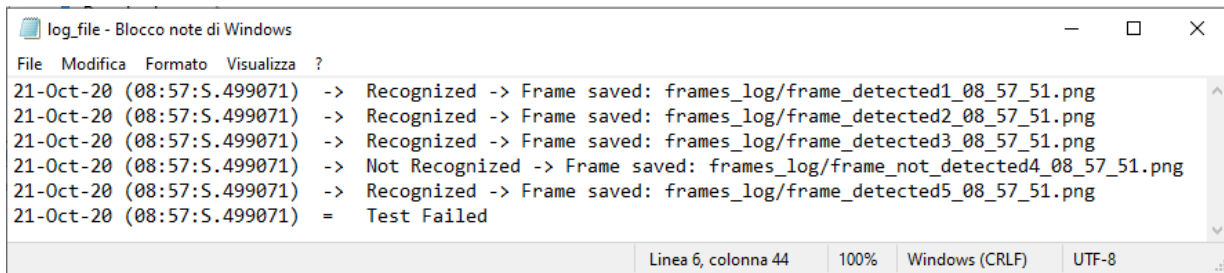
Figure 45: Detected templates in the frames captured by the camera

- A log file containing the date, time, and result of the executed test is created:



```
log_file - Blocco note di Windows
File Modifica Formato Visualizza ?
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected1_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected2_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected3_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected4_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected5_08_57_51.png
21-Oct-20 (08:57:S.499071) = Test Passed
Linea 6, colonna 1 100% Windows (CRLF) UTF-8
```

Figure 46: Example of a Passed Test recorded in the log file



```
log_file - Blocco note di Windows
File Modifica Formato Visualizza ?
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected1_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected2_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected3_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Not Recognized -> Frame saved: frames_log/frame_not_detected4_08_57_51.png
21-Oct-20 (08:57:S.499071) -> Recognized -> Frame saved: frames_log/frame_detected5_08_57_51.png
21-Oct-20 (08:57:S.499071) = Test Failed
Linea 6, colonna 44 100% Windows (CRLF) UTF-8
```

Figure 47: Example of a Failed Test recorded in the log file

The programmer can customize the content of the log file according to the needs of the user.

5. Conclusion

This chapter presents the goals achieved by the development of automatic testing with respect to the manual one, the limitations encountered, and the possible solutions to these limitations.

5.1 Goals

- Since, to automate a test, it is needed to use the CAPL programming language, the programmer decides how to customize the test automation. For example, he can decide that a test should be started simply by pressing a button from the graphic panel. He can also customize how many and which tests should run in succession. It is also possible to time the execution of a test, delaying the sending of certain inputs (this is made easier by the structure of the CAPL programming environment).
- As mentioned in the introduction, automatic testing is faster and more reliable than manual testing, especially when it comes to performing repetitive tasks that require a certain level of attention.
- During the automatic test, human resources may be used for other work. It is also possible to program the repetition of the same test to increase its robustness.
- Python is an open-source programming language, widely used and with the possibility to access many well-structured libraries and functions. Thanks to this, it is not difficult to customize a test report, easy to read, and more or less detailed.

5.2 Limits

Taking into account that everything has been realized with a limited budget, some of the following limits would be lost if you decided to modify the project having the possibility of drawing on greater availability of money.

1. A great limitation is given by the hardware available to perform the test. A low-quality image resolution and a wide-angle effect of the camera lead to templates not well defined and therefore difficult to use by the image recognition program. Its functionality is also dependent on the distance and angle between the camera and IPC, and on the surrounding brightness.

2. It was difficult to find a way for two programming environments so different from each other to communicate. To intensify the exchange of information between the CAPL software and the Python one, it would be uncomfortable and slow to continue using text messages to read and write from.
3. Imagining that it is needed to automate the testing of all the functions of an IPC in a vehicle, a lot of time will have to be spent creating a templates database.

5.3 Improvements

1. A higher resolution camera will allow us to have more defined templates and then raise the comparison threshold used by the image recognition software, to make it even more reliable. During the implementation of the project, I had the opportunity to test, for a few hours, the image recognition software on a DSLR Camera, the "CANON EOS 200D".



Figure 48: The DSLR Camera, CANON EOS 200D

With this level of detail in the image, and especially without the frames being distorted by a wide-angle lens (as in the case of the dash-cam), I was able to achieve much better results. In fact, this camera was able to discriminate more accurately the different templates in the captured frames and the software was able to complete the image recognition even if the IPC was not perfectly perpendicular to the camera.

Moreover, the construction of a solid support for the camera and the IPC will help us to obtain a faster and more precise initial configuration and to reduce any instrumental errors. Another possible solution is to improve the image recognition

software, making sure that templates are recognized also with different depths and angles from the original ones.

2. To improve the usability of the entire system, it could be possible to integrate the test automation and image recognition phase into a single programming environment. You would lose some specific features of Python or CAPL, but you would be able to improve the performance (you would no longer have to synchronize two different programs) and increase the potential of the whole software. The next step that we are working on, in the R&D team of the company, is to integrate the two different programming environments used for the thesis project, in that of C#. It is an object-oriented, C-like, programming language. Although it has not been specifically designed to perform simply and optimally the same functions of the CAPL and Python respectively for the configuration of an automatic test and the recognition of images, it lends itself to do both. Until today, with C#, we have been able to communicate with the IPC and automate the sending of some messages (even cyclical) for the activation of specific simple functions. We also imported some useful Python libraries and used the image recognition feature in combination with the camera management.
3. It would save time during the test preparation phase if the software could be more easily configured by a less experienced user. It means that it should no longer be needed to create a database of new template images whenever it is necessary to run the automatic test on a new device (e.g. a new version of the IPC). It should be created a test interface capable to simplify the test configuration, where a database of generic templates is already present and it is possible, through a few steps, to add the missing ones or modify those in disuse. Moreover, these operations should be done using an intuitive graphic interface and avoiding working directly on the C# code (it requires the intervention of a programmer).

References:

Bosch, 1991. *CAN Specification 2.0*. Stuttgart.

Dhananjayan, 2018. *embien*. [Online]

Available at: <https://www.embien.com/blog/working-automotive-can-protocol/>

[Accessed 12 2020].

Institute, I., 2019. *ALL About Circuits*. [Online]

Available at: <https://www.allaboutcircuits.com/technical-articles/introduction-to-can-controller-area-network/>

[Accessed 12 2020].

ISO, 1993. *ISO 11898: Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication*.

News, I., 2020. *COPPERHILL technologies*. [Online]

Available at: <https://copperhilltech.com/blog/can-bus-termination-adapter-suitable-for-can-bus-interfaces-with-dsub9-connector/>

[Accessed 12 2020].

Vector, 2004. *Programming with CAPL*. Michigan.

Vector. *VECTOR*. [Online]

Available at: <https://www.vector.com/int/en/products/products-a-z/hardware/network-interfaces/vn16xx/#c8929>

[Accessed 12 2020].

