Politecnico di Torino
Academic Year 2018/2019

Master Thesis

Master of Science in Mechatronic Engineering

A Virtual Testing Platform for Automotive Electronic Control Units

Thesis Advisor:
Prof. Luca Sterpone
Co-Advisors:
Prof. Boyang Du
PhD. Sarah Azimi

Candidate:
Simone Piattelli
242471

Turin, April 2019

Summary

In the last decades, the automotive industry faced a significant increase of electronic devices usage. In now-a-days vehicles, a plenty number of Electronic Control Units is used and mounted. In this sense, a modern car can be seen as a distributed intelligence.
These ECUs are used for several purposes, such as powertrain management and passengers comfort safety improvement. Due to mechanical and electronic integration, the vehicle development process has become much more complex. A huge number of requirements need to be fulfilled concurrently in order to provide a proper application item, indeed. Therefore, automotive applications are turning more and more software-dependent, since ECUs are taking care of crucial controlling purposes. The control algorithm needs to be developed and tested in a reliable and efficient manner.
This leads car manufactures to face the hard challenge of developing and testing items in a short time. The shorter the required time is, the less the cost of the item. Due to raising complexity, traditional methods are no longer able to cover test requirements. Thus, a lot of efforts are employed to find new ways for achieving this goal.
This Master Thesis is part of a relevant collaboration between "CAD and Reliability" group of Politecnico di Torino and "General Motor Powertrain" with the goal of development of software testing framework with virtual hardware prototyping tools
The main goal of this project is to provide a unique environment to test engineer for controlling the executions of tests across different validation platforms, such as Virtual Environment and Hardware-In-the-Loop (HiL). The considered ECU is the one responsible for engine management.
The environment consists in a Software Platform (Independent Python Platform). The implementation has been done using Python scripting language as an open source tool, which is compatible with GM developed platform. The key objective of the software testing platform is to achieve a unique environment to design and test the ECU using heterogeneous tools.
The working flow starts with the *Requirements Database*, which defines the configurations of the tests. In order to have a convenient method, *Javascript Object Notation (JSON)* file format has been proposed, which is an open source file format for transmitting data.

The platform is modular. The core module is the *TestManager.py* which is responsible for importing the requirements in the JSON file. Then, it will automatically generate the test commands required for running the HiL platform and executing the modules responsible for generating the expected executable requirements. Once the test execution terminates, the results are collected and stored in a certain file format and provided to the *Comparison Analysis* module for the post-process analysis. In particular, the analysis foresees the comparison among actual and expected results and checking whether they met or not.
As mentioned before, the *TestManager.py* loads the requirements. Then, it provides them to another module, called *Test.py*. This module is responsible for calling the procedures to generate the Input and Output Executable requirements, such as *Crank*, *Cam*, and *Injection Pulse* signals. Basically, the ECU uses the Crank and Cam signals as references to predict the pistons position and understand the whether to inject the fuel or not. For this reason accurate models of such signals are needed. The Master Thesis Project is focusing on Crank signal generation.

The Crank signal represents the angular position of the crankshaft. Basically, it is a square wave signal. Each tooth is identified by a rising and a falling edge. Each transition is considered as a reference to estimate the crankshaft relative rotation. Since the phonic or toothed wheel is considered to have 60 teeth, each tooth corresponds to 6°. Furthermore, the 59th and the 60th teeth are missed. This *gap* is needed for distinguishing between two different rotations. A single revolution equals to 720° swept by the crankshaft, indeed. Therefore, 120 teeth are placed inside a revolution or sample. It can be found under different conditions, such as: *Normal*, *Missing*, *Spurious*, and *Reference Transition Jitter*.
The *Normal Crank* represents the ideal signal, hence no error or misbehavior affects the signal. The corresponding model has been obtained considering several parameters embedded withing the JSON file as classes. The *NormalCrank.py* module is responsible for generating the Normal Crank Signal. It receives *General* and *Specific* parameters as arguments from the *Test.py* module. At first, a square wave is generate; then, according to the number of revolutions set, the gaps are inserted. The generated model is plotted and recorded by two more modules (*SignalPlot.py* and *SignalRecord.py*).
The *Missing Tooth Condition* affects the Crank signal by deleting one or more teeth within the same sample. In order to model this scenario, several parameters have been added inside the JSON file as subclass of the Crank Parameter class. The model is generated by the *CrankMissingTooth.py* module. It is launched by the *NormalCrank.py*. The required parameters are passed as arguments. Once the Missing Crank signal has been generated, the model is returned to the *NormalCrank.py* module and the execution keeps on.
The *Spurious Tooth Condition* occurs when extra pulses appear within two consecutive teeth of the Crank signal. In order to model such phenomenon, the *CrankSpu-*

riousTooth.py module has been implemented. Furthermore, several specific parameters have been embedded as subclass within the Crank parameter class of the JSON file. The module is invoked by the *NormalCrank.py.* The needed parameters are passed as arguments. The *CrankSpuriousTooth.py* module returns the Spurious Crank model, and the execution goes on.
The *Reference Transition Jitter Condition* affects one tooth of the Crank signal. In particular, it acts such a way to enlarge the transition period, causing time shifting to all the successive teeth. In order to model this event, the *CrankJitter.py* module has been implemented. Therefore, the specific parameters have been inserted as subclass within the Crank parameter class inside the JSON file. The module is called by *NormalCrank.py* passing the parameters as arguments. The *CrankJitter.py* module returns the model affected by such condition and the execution goes on.
Finally, the *Dynamic* system behavior has been modelled. In order to model such mode, two approaches have been used: *Time-based* and *Angle-based.*
The first approach considers a single section of an RPM-profile variation. It consists in dividing that specific section into several steps, defined by a fixed duration and RPM-increment. Several time vectors corresponding to each step are created. The same number of Crank signals are generated, each one is related to a certain time vector and a corresponding RPM. Then, the Crank signals are properly merged and the dynamical model is provided.
The second approach is based on the angle swept by the crankshaft during the speed transition. In particular, it foresees the evaluation of the angle to sweep for achieving a certain speed, considering a constant acceleration. Thus, the dynamical Crank model is generated tooth by tooth ($6°$), varying the corresponding period for each of them.
In conclusion, the purpose of this work is to provide suitable and usable models for testing an ECU responsible for engine-control. All these models are embedded in the Independent Python Platform as references for the automated testing procedures. Furthermore, thanks to them, the Platform is able to validate the system behavior under different conditions, and they can be used as reference for several heterogeneous tools used for testing (i.e. HiL, Virtual prototyping, FPGA-based).

Contents

1 Introduction **1**
1.1 Embedded Systems . . . 2
1.1.1 Hardware Platform Architecture . . . 2
1.1.2 Operating System Structure . . . 4
1.1.3 Real-Time Embedded Systems . . . 5
1.2 Automotive Electronic Control Unit . . . 6
1.2.1 Automotive Networks . . . 8
1.3 V-Shape Development Flow . . . 12
1.3.1 Model-in-the-Loop . . . 14
1.3.2 Software-in-the-Loop . . . 15
1.3.3 Process-in-the-Loop . . . 15
1.3.4 Hardware-in-the-Loop . . . 16

2 Related Works **19**
2.1 ISO 26262 . . . 19
2.1.1 Vocabulary . . . 20
2.1.2 Management of functional safety . . . 21
2.1.3 Safety Life Cycle . . . 21
2.2 Physical and Virtual Prototypes . . . 23
2.2.1 FPGA-based prototype . . . 24
2.2.2 Virtualization . . . 24

3 Virtual Independent Platform **26**
3.1 Independent Python Platform Architecture . . . 26
3.2 Requirement DataBase . . . 28
3.3 Test Manager . . . 33
3.4 Mathematical Models . . . 35
3.5 Test Commands . . . 38

4 Mathematical Models **39**
4.1 Introduction . . . 39
4.2 Crank Signal . . . 40

4.3 Normal Crank Signal Generation 41
4.4 Crank Missing Tooth Condition 44
4.5 Crank Spurious Tooth Condition 46
4.6 Crank Reference Transition Jitter 49
4.7 Dynamic RPM 52
4.7.1 Time-Based Approach 53
4.7.2 Angular-Based Approach 56

5 Results **63**
5.1 Normal Crank Signal 63
5.2 Missing Tooth Condition 65
5.3 Spurious Tooth Condition 67
5.4 Reference Transition Jitter Condition 69
5.5 Dynamic RPM in Time Domain 71
5.6 Dynamic RPM in Angle Domain 72

6 Conclusion **74**
6.1 Future Works 75

List of Figures

1.1 Microcontroller-based . . . 3
1.2 System-on-Chip . . . 3
1.3 Combustion Engine Timing Behavior . . . 7
1.4 General Network . . . 8
1.5 ISO/OSI protocol model . . . 9
1.6 V-shape model development flow . . . 13
1.7 Model-in-the-Loop. . . . 14
1.8 Software-in-the-Loop. . . . 15
1.9 Hardware-in-the-Loop. . . . 16
1.10 Hardware and Software ECU scheme. . . . 17

3.1 Test Environment Overview . . . 27
3.2 Process and Complex Product Requirements Verified simultaneously. 29
3.3 Graph Database Approach. . . . 30
3.4 Graph Representation of Complex Requirement for CrankCam stimuli. 31
3.5 Horizontal extension. . . . 32
3.6 Vertical extension. . . . 32
3.7 Test Manager work-flow. . . . 34
3.8 Crank to Cam Shift. . . . 36
3.9 Crank (red), Cam (green) and Injection Pulse Signals (blue). . . . 37

4.1 Forward and Backward Active Period . . . 42
4.2 Crank Signal generated with Crank Initial Position set to 0 . . . 43
4.3 Crank Signal affected by a single Missing tooth occurrence . . . 44
4.4 Missing tooth parameters embedded into .json file . . . 45
4.5 Example of Crank signal in Missing Tooth Condition . . . 46
4.6 Crank Signal affected by a single Spurious tooth occurrence . . . 47
4.7 Missing tooth parameters embedded into .json file . . . 48
4.8 Example of Crank signal in Spurious Tooth Condition . . . 49
4.9 Crank Signal affected by Reference Transition Jitter . . . 50
4.10 Reference Transition Jitter parameters embedded into .json file . . . 51
4.11 Example of Crank signal with Reference Transition Jitter . . . 52
4.12 Speed variation profile with constant acceleration . . . 53

4.13 Sampled speed variation profile with constant acceleration 54
4.14 Crank signal Variation example 55
4.15 Full Speed Profile. 58
4.16 Angular-based parameters in the .json file. 59
4.17 Example 1. 61
4.18 Example 2. 61

5.1 Normal Crank Signal: Example 1. 64
5.2 Normal Crank Signal: Example 2. 64
5.3 Missing Tooth Condition: Example 1. 65
5.4 Missing Tooth Condition: Example 2. 66
5.5 Missing Tooth Condition: Example 3. 66
5.6 Spurious Tooth Condition: Example 1. 67
5.7 Spurious Tooth Condition: Example 2. 68
5.8 Spurious Tooth Condition: Example 3. 68
5.9 Reference Transition Jitter Condition: Example 1. 69
5.10 Reference Transition Jitter Condition: Example 2. 70
5.11 Reference Transition Jitter Condition: Example 3. 70
5.12 Crank signal Variation example 71
5.13 Dynamic RPM Angle-Based: Example 1. 72
5.14 Dynamic RPM Angle-Based: Example 2. 73

List of Tables

4.1 Crank signal General and Specific Parameters. 43
4.2 Missing Tooth Specific Parameters. 45
4.3 Spurious Tooth Specific Parameters. 47
4.4 Reference Transition Jitter Specific Parameters. 50
4.5 Time-Based RPM Dynamic parameters. 55
4.6 Angle-Based RPM Dynamic parameters. 59

Chapter 1

Introduction

In the last years, the automotive industry has seen a significant increase of electronic devices usage. Indeed, in modern vehicles, a plenty number of *Electronic Control Units* is used for several control purposes. The main ones are the powertrain management and passengers comfort and safety improvement.
A vehicle development process has become extremely complex due to mechanical and electronic integration [1]. A huge number of requirements must be met in order to provide proper application item.
Furthermore, automotive applications are becoming more software-dependent; in this sense, a modern vehicle can be seen as a distributed intelligence network.
Another crucial aspect for automotive application is the control algorithm main feature. It is responsible for engine management; hence, real-time constrains must be respected and satisfied for providing, i.e., a proper torque to wheel.
In order to prevent human coding errors, *code generation* is used in mass-production development. Finding ways for testing and validating in a fast and efficient manner is becoming more and more important. Indeed, the key objective of such efforts is to obtain as more efficient and effective software as possible.
For achieving this goal, reliable and standard validation procedures are used for avoiding unwanted errors and faults. Hence, great efforts are put for guaranteeing quality and reliability for the item under development.
Currently, the used validation methods are based on *models* [8] or by using ad-hoc special purpose test requirements.
Due to raising complexity, traditional methods are becoming no longer able to cover the test requirements [5], which change constantly. This leads companies to face much higher costs for testing and validating.
For these reasons, the following Master Thesis work has been implemented. It is part of a bigger project which is meant for developing a *Virtual Testing Platform*. The platform under development will provide a unique, flexible and efficient framework to validate automotive applications. The platform will be able to be customized with different features for managing several test benches.

In the following chapter, a brief overview of the main aspects related to *Embedded Systems* and *Electronic Control Unit* are proposed.

1.1 Embedded Systems

An **embedded system** is a microprocessor-based electronic system designed to accomplish a specific task or sets of special applications (Special Purpose device). These systems can standalone or take place inside a further more complex product (i.e. Antilock Braking System) [3]. Since the purposes are a-priori well-known, its development (hardware and software combination) is meant for best meeting the requirements. Therefore, they are not re-programmable by user and mount an ad hoc hardware platform. The special hardware is meant for avoiding power consumption, lowering manufacturing costs and processing time.
Having a low processing time is fundamental for real-time software to best fit tasks deadlines. The **deterministic** behaviour of embedded systems is crucial in order to fulfill safety purposes.
The embedded systems are classified according to microprocessor, operating system and complexity of application.

1.1.1 Hardware Platform Architecture

The Embedded Systems family can be divided according to architecture in two categories:

- *Microcontroller-based implementation*: a single device contains most of the components (i.e. CPU, RAM memory, Flash memory), figure 1.1;
- *System-on-Chip implementation*: the components are discrete and different elements, figure 1.2.

The microcontroller-based systems are made up a single **monolithic** component running at quite low frequencies. It is able to be interconnected to others for adding functionality.The MCU is widely used for its cheapness and small size. Therefore, it is particularly favorite in *real-time* applications with no complex interface [4].
While, the SoC systems are more complex. Since networks and interconnections require a much higher effort, for adding more functionality it is necessary to mount other components on the board.
Furthermore, due to the sophisticated structure, they are more expensive than MCU systems. Anyhow, they can support higher frequency and complex user-interfaces (i.e. used for Smartphones).

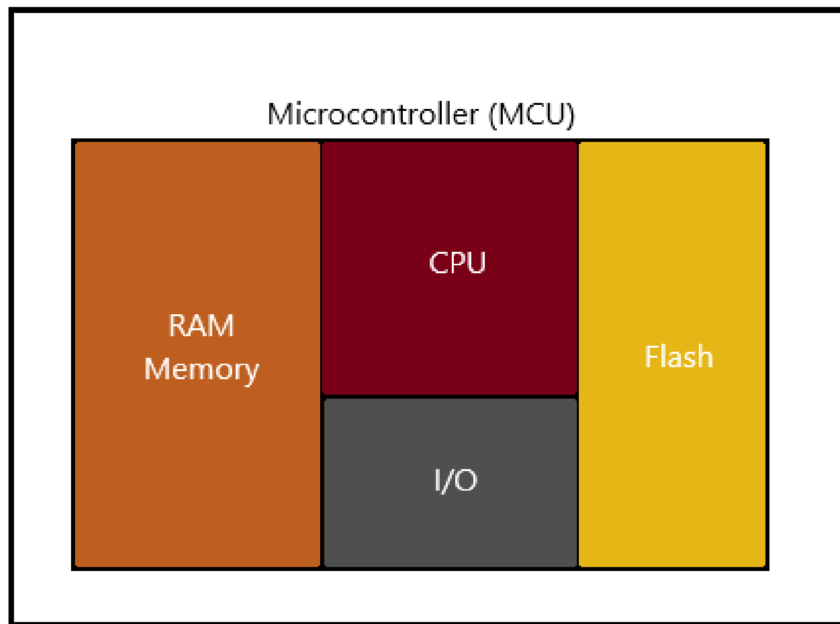

Figure 1.1: Microcontroller-based

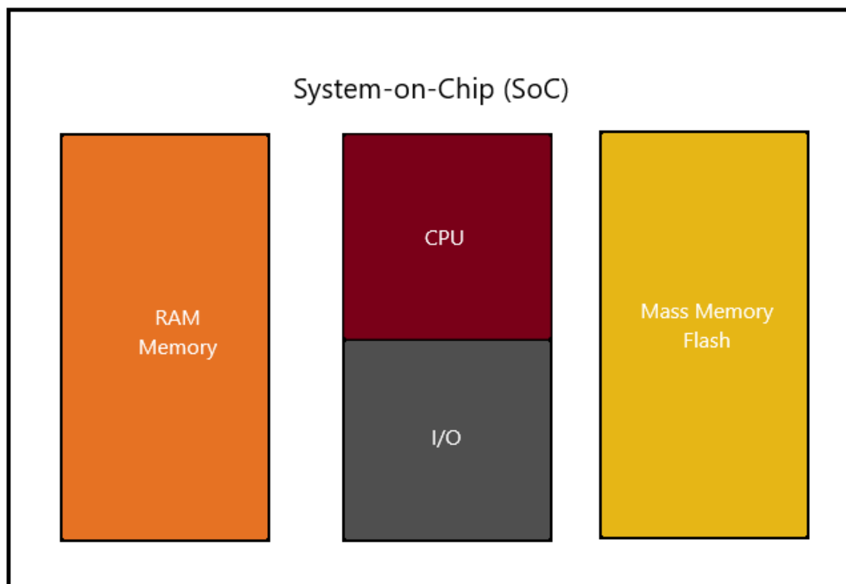

Figure 1.2: System-on-Chip

In many applications, the system is implemented as a printed circuit board (PCB). The MCU and all the application peripherals are mounted over the PCB.

1.1.2 Operating System Structure

The *Operating System* can be considered as an abstraction of the hardware resources. It is meant for simplifying and optimizing user access and usage to system components [10].
The anatomy of the basic software consists in two main layers: **Operating System** and **System-call interface**. The OS is the software which implements services using HW and is composed of:

- *CPU Manager*: it implements algorithm for handling the access to CPU resource and optimizing CPU time usage;
- *Memory Manager*: it takes care of satisfying applications memory demands;
- *File Manager*: it manages the access to the persistent memory;
- *Device Manager*: it defines protocols for providing communications between the I/O peripherals and the system;
- *HW-Specific Services*: it provides CPU-specific operations.

The System-call interface is the part exposed to the application software, which allows triggering the Operating System.
Many Operating Systems architectures have been proposed during the years. The most used ones are the *Flat Architecture*, the *Layered Architecture based on Monolithic Kernel* and the *Microkernel Architecture.*

Flat Architecture

This kind of architecture is meant for providing as much functionalities as possible using the least space available.
The Operating System is composed of a series of functions, which can be invoked by any application. Furthermore, there is no strictly separation regarding the memory access between OS and application; hence, no data discrimination inside the memory. This can lead to malfunctions propagation, which can hardly corrupt the OS. There is no data protection between OS and user application, indeed.
When a data corruption takes place inside the user address space of the RAM memory, it can spread to the Kernel address space. For this reason, this kind of Operating System is **not reliable**. Anyhow, it offers efficient communications.

Layered Architecture

This kind of architecture foresees the discrimination into layers of the different components of the Operating System. In this way, the services and the functions provided to each layer can only come from the lower layer.

Therefore, the upper application layer can communicates with the OS only through the System-call interface.
Furthermore, each layer has virtually assigned a specific address space set inside the RAM memory. All the services are provided by an executable called *Monolithic Kernel*.
The monolithic kernel can be updated by adding modules; there are two ways for inserting new drivers:

- the new driver is linked with the kernel and the new kernel must be rebooted;
- the modules are inserted into the kernel space during runtime.

Due to the address separation between Application and Basic software, malfunctions occurring on User layer cannot affect OS. This offers a *partial robustness*.
Therefore, OS faults can kill the OS itself (i.e. new buggy driver inserted in the kernel address space). No protection between operating system components, indeed.

Microkernel Architecture

In this architecture, every layer has its own dedicated and well-defined RAM address space, so memory location.
Therefore, the microkernel is easier to extend, it is more reliable and more secure. Even if there is a performance overhead of user space to kernel space communication. Malfunction can occur in wherever layer, it will not propagate to others. **Total separation among layers address spaces** leads to an extreme robustness.

1.1.3 Real-Time Embedded Systems

The CPU scheduler works on PCB (*Process Control Block*). Actually, a PCB contains all the information related to a specific process, such as: process state, ID number, copy of CPU register, address in the memory, and so on so forth.
In order to manage all the processes, the CPU scheduler needs to implement specific algorithms responsible for identifying the proper criteria to pick up a process among the ones in ready-status and make it running. These functions are called **Scheduling Algorithms**.
Depending on which kind of Scheduling Algorithm the OS implements (i.e. whether it is static priority-based or dynamic one), it is possible to discriminate between two types of systems [10]:

- *Non real-time systems*: the system has just to respond properly to an external event. No deadline specified for process to be run. Hence, delay between the begin and the end of a process can be whatever;

- *Real-time systems*: the system has to respond properly to an external stimulus within a certain deadline. Hence, the system must guarantee a **deterministic** behaviour in any possible operative conditions.

Therefore, in real-time systems the output shall be provided no later than a specific amount of time (deadline), otherwise it will no longer be valid.
There are many types of real-time systems:

- **Hard real-time**: if the deadline is missed, it may cause catastrophic consequences on the environment under control (i.e. control system to ensure that the temperature of a reactor does not go over a certain threshold for no longer than a certain time).
- **Firm real-time**: if the deadline is missed, the results are useless, but no serious damages occur. Data are useful if and only if produced within the deadline.
- **Soft real-time**: the meeting of the deadline is desirable for computing purposes; but missing it does not affect the system behavior. Indeed, after the deadline expires, there is a certain time within the output has a decreasing behavior but still useful. Obviously, the closer the output data are delivered to the deadline, the more they are effective (closer to maximum value).

In automotive applications, all the functional safety purposes systems are real-time (i.e. Air Bag).

1.2 Automotive Electronic Control Unit

In a now-days car we can find out a plenty number of **Electronic Control Units**. It is called ECU whichever real-time embedded system used for controlling mechanical and electronic devices inside the vehicle.
Their increasing number is leading to create of a sort of *distributed intelligence* within the car. Therefore, all the ECUs are interconnected by means of networks and cooperating to provide safety and efficient driving experience.
Inside a vehicle there are different types of ECUs; the main differences are related to the system to control:

- *Powertrain and Chassis control* ECUs are responsible for providing and distributing the torques and guaranteeing vehicle handling. For example, depending on the nature of the engine (whether it is combustion or hybrid), the transmission can be managed by several different ECUs. Hence, it is important to synchronize them for providing the proper required torque.
- *Body Electronics* ECUs fulfill comfort and safety purposes. Examples are instrument panel, window and lighting management, airbag, seat-belt.

- *Multimedia and Infotainment Applications* ECUs provides entertainment, information, localization and navigation functionalities to the vehicle occupants.
- *Integrated Services* ECUs performs specific features and functionalities activated by commands from other elements in order to achieve a certain goal (i.e. Electronic Stability Control, Autonomous Driving services).

The different nature of the systems to be controlled dictates specific requirements for each ECU; hence, different kind of networks.
Since ECUs are mainly real-time systems, data must be provided within a certain deadline: communication must be deterministic (on time) and reliable (safely).

Example: Timing Behaviour of Engine Management System

The figure below 1.3 shows the behavior of a piston of a combustion engine within a single cycle. Considering that the rotation speed is 600 RPM, the entire revolution (720°) will lasts for 20 ms.
Therefore, only few hundreds μs are available for injecting the fuel. Hence, the ECU has to provide the required data within that deadline.

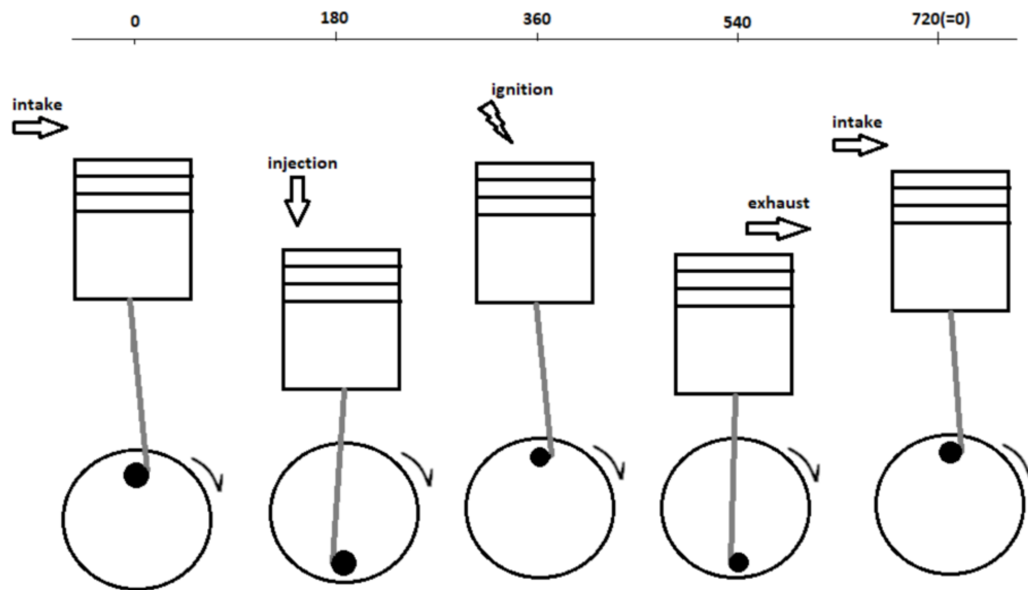

Figure 1.3: Combustion Engine Timing Behavior

Furthermore, the ECU takes into account several parameters for evaluating the quantity to be injected, such as the acceleration pedal position. Then, the data must be delivered through a network to the actuators in order to perform the required

operation.
Both computation and communication must be done in compliance with a certain deadline and properly, for guaranteeing **determinism** and **safety**.

1.2.1 Automotive Networks

The increasing number of ECUs inside vehicles have made necessary the definitions of several protocols for ruling and managing the inter and extra-vehicle communications. Thus, plenty numbers of specific automotive ISO/OSI models have been developed across the years for achieving these purposes. In this section, an overview of the most used standard is covered.

General Network Overview

A telecommunication network is a set of nodes and links connected with each other such as to guarantee communications among terminal nodes.

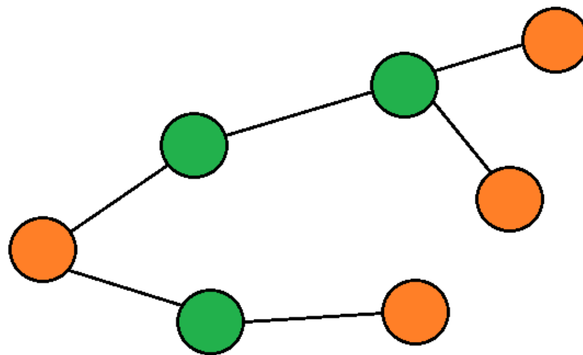

Figure 1.4: General Network

The figure above 1.4 shows a general network structure. The orange nodes are **terminals**, the green ones are the **intermediate** nodes (used for deliver the message to the target), and the bars which connect the different nodes are the **links** (responsible for exchanging messages between two successive nodes).
The terminal represents the producer or target of the communication. The **messages** are packed and ordered group of bits containing the information to deliver.
At first the messages are produced and sent by the terminal nodes. Then, they are exchanged by means of links to the consecutive intermediate nodes. The process is repeated as long as the target terminal nodes are not reached.
Each message is composed of two main parts: *payload* and *control*. The payload is the actual content of the message; while, the control is needed for communication

purpose only.
Furthermore, the communication is designed and implemented according to a well-fixed policy, called **protocol**. A protocol consists in a set of rules describing the type of networks, the communication handling, and the messages structure.
The ISO (International Standard Organization) defines the structure of each protocol, in order to facilitate the realization of inter-operable communication systems. Each structure represents an open model **OSI** (Open System Interconnection). An *ISO OSI* model is based on **7 layers**, each one specifying a certain aspect. Although, a layer is only able to interact with the layers right below and above and provides new service or improve functionality with respect to the previous one. The interaction takes place from the bottom to the top.

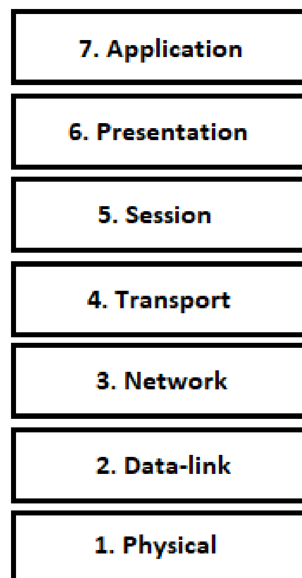

Figure 1.5: ISO/OSI protocol model

The figure 1.5 depicts the 7 layers taking part to the network communication process:

- *Physical layer*: electrical and physical specifications regarding the data connection. Hence, all the features related to the physical medium are specified in this layer. Therefore, the interconnection between two directly linked nodes and the conversion used for exchanging data (modulation) is described.

- *Data-link layer*: features and specifications for a reliable transmission between two consecutive linked nodes. Thus, the message structure is defined by identifying the control and payload fields. Notice, it is meant for making reliable the direct connection, not all the network.

- *Network layer*: extension of the communication to the several nodes composing the network. The concept of addressing is defined; therefore, the capability of the network for univocally distinguishing among the different nodes. In addition, messages are splitted into datagrams in order to facilitate the delivery of data.
- *Transport layer*: reliable delivery commitment of datagrams among terminals identified by a unique address.
- *Session layer*: features related to the dialogues among terminals. It defines the communication mode, whether it has to be full-duplex, half-duplex, or simplex.
- *Presentation layer*: adding functionalities regarding data modifications, such a way it is transparent to users. Hence, enabling the compression of data and it is performed by the user.
- *Application layer*: specific services regarding the final applications user are implemented. Thus, features for making the communication compliant with the application.

Anyhow, not all the layers are implemented by the several protocols (i.e. CAN implements only physical, data-link and application layers).

Controller Area Network

The **Controller Area Network** (**CAN**) is de facto the standard for data exchanging in vehicles. It is also used in production and industrial environment.
It is an *asynchronous serial bus.* Basically, the communications is initiated at any time by a sender for transferring data to N points. Furthermore, *two wires* are used for the transmission, one bit at time.
A CAN protocol implements only 3 layers of the ISO/OSI standard model: physical, data-link, and application. Since there is no intermediate layers between data-link and application layer, it is not possible to send data to a specific target. Thus, all the nodes share the same couple of lines. Then, due to the asynchronous behavior, any node can start the communication at any time.
The physical layer foresees two wires: CAN-H and CAN-L. These terminate with a resistance of 120Ω for reflection avoidance purpose. The voltage dynamic is from 0V to 5V. A logic 0 is implemented by setting CAN-H to 5V and CAN-L to 0V; while, both CAN-H and CAN-L are set to 2.5V to obtain a logic 1. It is important to underline that 2.5V is considered as *recessive voltage*; instead, the *dominant voltage* is assigned to 0V. The protocol implements a **multi-master** with **broadcast concept**. Thus, all the nodes belonging to the network are peers (masters) can start communicating independently at any given time. In addition, whichever message

sent over the network reaches indistinctly whichever node connected.
In order to avoid collisions, a policy for accessing the bus is needed: **Carrier Sense Multiple Access with Collision Detection**. It is meant for letting the several nodes willing to start communication to understand whether the bus is idle (recessive value) or busy (dominant value). This is done by sending one bit at time and then sense if the bus is set to the value just sent or not. If it is, the node will continue sending. Otherwise, it will stop.

Local Interconnected Network

The **LIN** protocol implements only the physical and data-link layer. It is intended for sub-networks, where the amount of data is low (i.e. managing of seat motors position). In fact, this protocol is not so performing, but the cabling complexity is very low.
A LIN protocol is an asynchronous serial bus, implementing a single master multi-slaves solution with broadcasting concept. The master node is the only one responsible for starting and deciding the slave allowed to communicate. So, collisions are intrinsically managed.
This protocol is based on the **token ring** concept, which is a distributed arbitration mechanism. At any given time, the master generates and sends over the line a specific group of bit (token or header), containing the address of the slave desired. This last recognizes its own address and replies with the required information.
In general, the master is part of a bigger network, such as CAN or Ethernet AVB.
The network is composed by a single wire connected to 1 master and up to 15 slaves nodes. Since only a line is used, a quite huge voltage gap between recessive (logic 0) and dominant (logic 1) values is needed for signal integrity and robustness purposes. Thus, the dominant value is 0V; while, the recessive value is equal to V_{bat} (8V-18V).

Automotive Ethernet

The Automotive Ethernet (or Ethernet AVB) is the reference protocol for high speed communications. It is meant for transporting huge amount of data in real-time applications.
Despite to general Ethernet, in order to guarantee functional safety and quality of signal, some sets of further protocols have been added. Even if the basic structure is almost the same.
The Ethernet AVB protocol is *full-duplex* with *bandwidth reservation*. This means that for delivering data from a transmitter to a receiver a certain preferential path is guaranteed. In this way, the time spent in the transmission is known a priori and a maximum boundary time is set (*deterministic*).
The network is composed of **end points** and **bridges**. The end points represent either the source and the target of the communication; while, the bridges allows the transmission among the several end points. Furthermore, the bridges are seen as

intelligent components able to route incoming packets to the proper destination.
A general Ethernet protocol implements two layers of the ISO/OSI model: physical and data-link layers. The physical layer consists in a certain number of medium lines used for the transmission with respect to the distance that the network must cover and cable.
The data-link layer is composed of two further layers:

- **Media Access Control** (**MAC**): bottom part, responsible for generating the frame according to certain network specifications. Hence, this layer is responsible for managing the access to the network, addressing the final target and packing the data to deliver in a logic manner. Furthermore, the MAC layers can be of different types according to the collision avoidance policy they implement. The policy is chosen in relation with the used architecture (i.e. CSMA-CD or token ring). Everything is implemented in hardware.

- **Logical Link Control** (**LLC**): top part, it provides additional services, such as: error and flow control, managing the interconnection between different kind of MACs. This is implemented as a software layer.

As previously mentioned, the Automotive Ethernet must guarantee a reliable and deterministic communication, due to real-time purposes. Special set of software protocol have been added in order to achieve this goal:

- **Best Master Clock Algorithm** (**BMCA**): needed for selecting and ensuring that all the end points and bridges of the network agrees on the same time base. Used for guaranteeing Quality of Service (QoS);

- **Precision Time Protocol** (**PTP**): it ensures that all the bridges and end points are permanently synchronous;

- **Stream Reservation Protocol** (**SRP**): it allows all the bridges to reserve the proper amount of bandwidth for the different connections to be managed;

- **Forwarding and Queuing Enhancement for Time Sensitive Streams** (**FQTSS**): it enables bridges to distinguish between time critical traffic flow and not ones. In this way, the priority is given to critical traffic, while best effort to non-critical traffic

- **Audio/Video Transport Protocol** (**AVTP**): it allows to streams audio/video traffic over the network.

1.3 V-Shape Development Flow

In automotive industry, due to market laws, ECUs software has to be developed and tested in a fast and fault-free manner. For this reason, model-based software design

is used as standard development. The de facto most popular design and testing working flow is the *V-shape* model. It is compliant with the **ISO 26262**, which rules the functional safety requirements for electronic and electrical devices inside the vehicles.

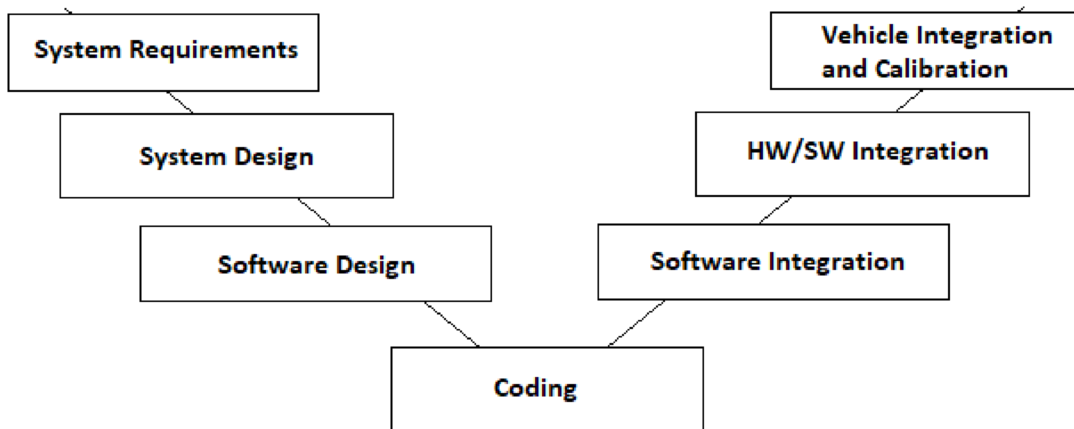

Figure 1.6: V-shape model development flow

The V-shape development flow foresees several steps, each one regarding a certain aspect of the item to be designed:

1. **System Requirements**: in this phase, item definition is defined. Hence, the functionalities to be provided in order to achieve functional safety.

2. **System Design**: partitioning of the functionalities into submodules that shall be designed and then implemented.

3. **Software Design**: definition of the functions to be used for implementing the subsystems functionalities.

4. **Coding**: implementation of the instructions of the previously defined functions.

5. **Software Integration**: merging of the different coded subsystems.

6. **Hardware and Software Integration**: integration of the whole software in the embedded hardware (execution platform) and testing the behavior.

7. **Vehicle Integration and Calibration**: joining the item into the vehicle and validating its functionalities. Furthermore, if the item is meant for being used by different vehicles, re-calibration is performed taking into account specific parameters.

With reference to the figure 1.6, the left descending branch is referred to phases 1-2-3 (design); while the right ascending branch is related to steps 5-6-7. It is possible to determine a correlation between all the opposite steps. Hence, whether an error is discovered in a certain phase of the ascending branch (i.e. 6), it is necessary to go back to the corresponding phase of the descending branch (i.e. 2). The later the error is detected, the more expensive the designing phase is.
In order to avoid unintended coding and requirement faults, *automatic code generation* and *model-based development* are used.
The automatic code generation is very efficient in terms of productivity, even if the obtained code needs to be optimized. Once the model has been tested, it is sent to tools (i.e. Simulink) for automatically code translation.
The requirement faults avoidance is performed by considering the **X-in-the-Loop** approach. This methodology allows to meet the requirements of the item by performing early phase development check **simulations**, hence virtually. This is very cost reduce; wrong implementations are detected as soon as possible.
The 4 main simulation methodologies are:

- **Model-in-the-Loop**, **MiL**
- **Software-in-the-Loop**, **SiL**
- **Process-in-the-Loop**, **PiL**
- **Hardware-in-the-Loop**, **HiL**

1.3.1 Model-in-the-Loop

Its purpose is to obtain a controller as much accurate as possible. The controller is the subsystem to be translated into code.
In this phase, both controller and plant are dynamically modelled. The goal is to test the controller by simulating the entire feedback control system, as shown in figure 1.7.

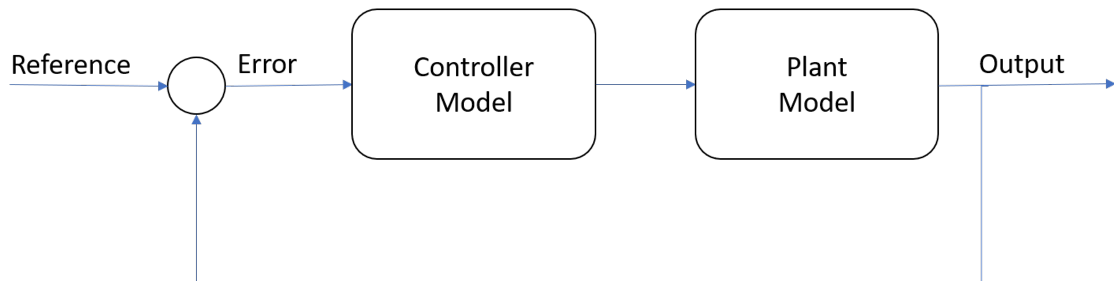

Figure 1.7: Model-in-the-Loop.

A reference signal is given as input to the feedback chain and the error signal is computed. This is set as input of the controller, which gives the control signal.
Hence, the plant is fed with the controller output and the response to control signal is evaluated.
The behaviour must be compliant with the expected results. If it is not, then the controller must be designed from the beginning.
Otherwise, it is possible to go further and perform the code generation of the controller.
It is necessary to underline that in MiL, both controller and plant models are simulated using the same environment (i.e. Simulink).

1.3.2 Software-in-the-Loop

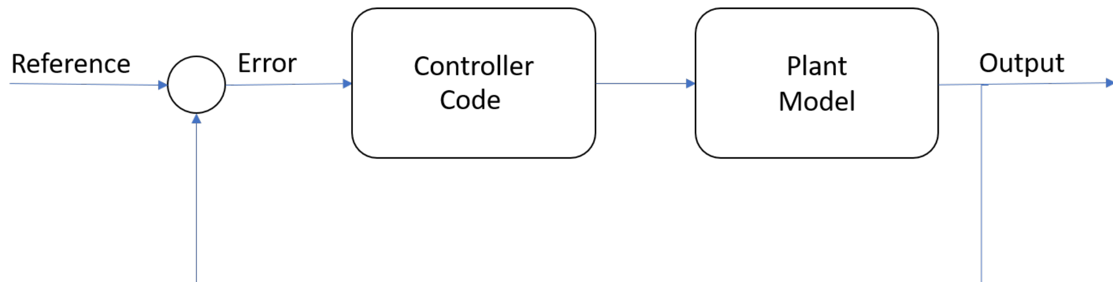

Figure 1.8: Software-in-the-Loop.

Once the code is generated, optimization and code generation are needed.
In this phase details related to the hardware platform are added (i.e. taking into account the processor to use). Indeed, the previous model was platform independent; while, in this phase, the software is forwarding to run over a specific platform. This means that it must be adapted for that target making the code less platform independent but more implementation friendly. This is done by optimization.
In the SiL process, the controller is translated into code and it is co-simulated with the plant model. This is needed for checking whether the code generation and optimization performed have affected the controller behavior. Therefore, the optimized code must be compliant with the model behavior. It is important to highlight that the simulated modules run over the same environment (laptop).

1.3.3 Process-in-the-Loop

In this part of the process, the software runs over a real embedded hardware, which will be used for the application. The plant is still simulated by a model (i.e. Simulink environment); anyhow, it interacts with the rapid prototyping hardware, or ECU, or an Evaluation Board (EVB).

Therefore, it is very close to the real implementation. This testing phase is meant for validating whether the platform-dependent code works properly in compliance with hardware requirements.
The board pins take incoming signals and discretize them by ADC and S&H. Hence, they are translated from analog to digital quantities. Obviously, software drivers for controlling hardware resources are needed. Once processed, the controller output is sent to the simulated plant. In order to do this, a simple import is performed.
Nevertheless, due to simulated plant, the time spent for running that simulation can be faster or lower than actual system. Thus, the control system is **not working in real-time**. In order to reach real-time validation, the ECU needs to be mounted over the vehicle.

1.3.4 Hardware-in-the-Loop

This is the latest phase of the validation process. Basically, the control algorithm runs over the ECU; while, the plant is emulated by an **emulation hardware**, figure 1.9.

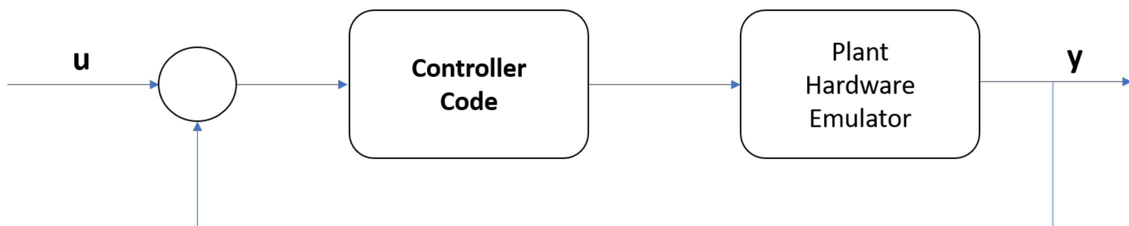

Figure 1.9: Hardware-in-the-Loop.

The emulation hardware is able to run in real-time. Hence, the controller is not able to distinguish from the real and the emulated plant. Therefore, the harness is the same expected in the final application.
It is necessary to highlight that 1 second in the emulator equals to 1 second in the real plant.

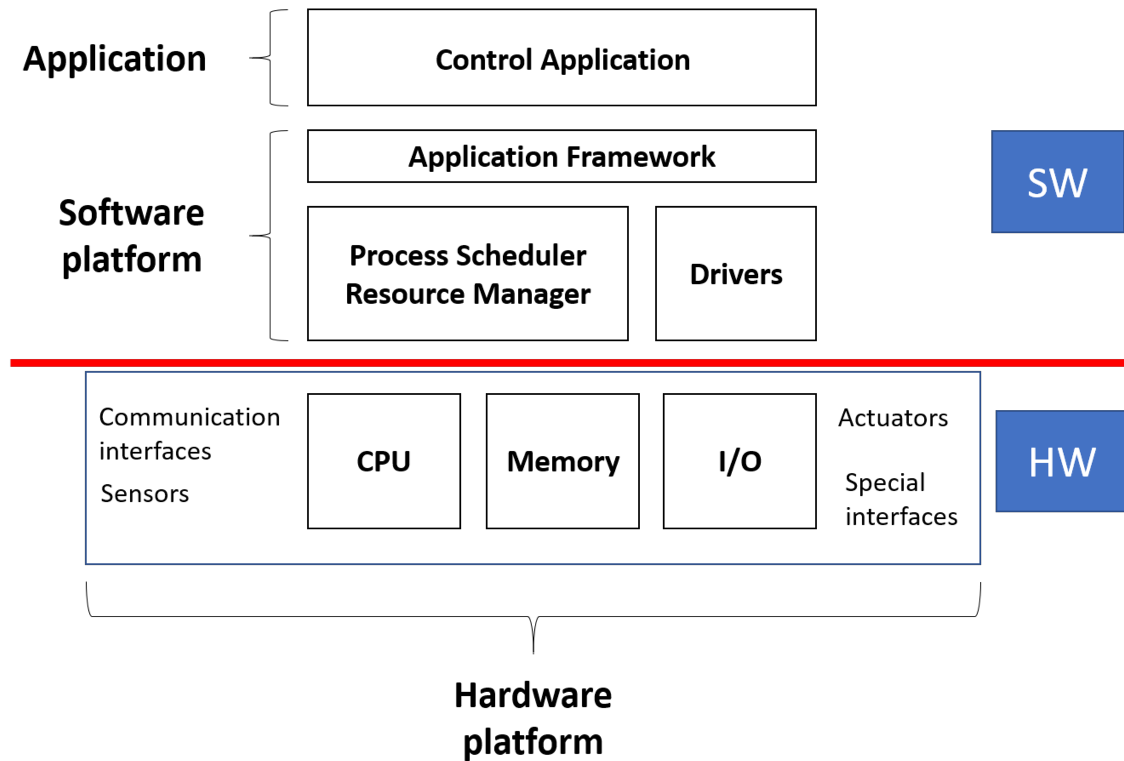

Figure 1.10: Hardware and Software ECU scheme.

The figure 1.10 represents schematically the ECU hardware and software components. The bottom part is made up of hardware platform. Basically, this is responsible for computation and I/O capabilities needed for deploying the application functionalities to real plant. Hence, this is the set of electronic components and printed circuit board that provides the infrastructure for running the software.
The top part is divided in two layers: Software platform and Application. The software platform represents the basic software; hence, the services needed for running the application. It is made up of the Operating System and the Call-service layer.
The OS is the software abstraction of the hardware below. It is meant for managing optimally the CPU and memory resources. Furthermore, it provides access to I/O peripherals through specific drivers and other user interfaces.
The Call-service layer acts like a mediator between the OS and the application, whether it is present.
The application is the controller code produced or item and to be validated.
Therefore, the HiL process is used for checking the **actual controller timing response**.
The HiL needs a lot of effort by test engineer, since it is one of the last step before commercializing the item. Once the HiL testing phase is performed, the item is ready to be mounted over the vehicle for the calibration and integration.

Chapter 2

Related Works

In the following chapter, related works associated to the item definition and validation are proposed.
In particular, in the first section the ISO 26262 is faced. This protocol is meant for outlining guide lines to determine functional safety for electrical and electronic items used in automotive. It is important for the ASIL definition. Indeed, item associated to high ASIL-level will be subjected to more rigid development and procedures, based on V-shape development flow.
The last section is related to validation and testing phases. More in detail, two developed prototyping methods will be briefly analyzed. One of them takes advantage of *physical* approach (**FPGA-based**); while the other uses a *virtual* approach (**Virtualization-based**). They are important since a post-process analysis can be performed by comparing the HiL results with the VP or FPGA-based ones. They are meant for reducing the time needed for validation, allowing cost reduction.

2.1 ISO 26262

The **ISO 26262** (*"Road vehicles - Functional safety"*) is the international standard by International Organization for Standardization (ISO) meant for functional safety for electrical and electronic items used in vehicles.
The main purposes of the ISO 26262 is to provide a **Safety life cycle** for the item to be designed. Basically, starting from the item definition, the **functional safety concept** is outlined. Hence, the additional functionalities to add in order to obtain an item able to satisfy functional safe requirements. Notice, these are further functionalities in addition to the basic user ones.
Then, the **production development phase** will be based on the previous steps, according to the V-shape model. Therefore, the ISO 26262 outlines the development phase intended for meeting functional safety requirements. The output of this is the integration of hardware and software; hence, **production**, **operation**, **service**, and **decommissioning**.

Each step of the development and production is well-defined. By means of standardization, errors and issues are easier to be discovered. This means benefit in terms of **time** and **costs**.
The ISO 26262 is made up of 9 normative chunks, such as:

1. *Vocabulary*
2. *Management of functional safety*
3. *Concept phase*
4. *Product development at the system level*
5. *Product development at the hardware level*
6. *Product development at the software level*
7. *Production and operation*
8. *Supporting processes*
9. *Automotive Safety Integrity Level (ASIL)*

2.1.1 Vocabulary

The ISO 26262 takes advantages of using a standard and specific *Vocabulary* [9]. This provides several definitions, such as:

- **Item**: It is referred to specific system or array of systems which implements a function inside the vehicle.
- **Element**: basic units which compose the item. An element may be a hardware or software component.
- **Fault**: irregular condition which causes the item or the element to faulty behavior.
- **Error**: dichotomy between the computed and observed value and the theoretically one.
- **Failure**: inability of the item to perform the required functionality.
- **Malfunctioning Behavior**: unintended behavior of the item, cause by a fault.
- **Hazard**: potential source of harm due to faulty item working.
- **Functional Safety**: absence of hazardous risks caused by faulty working condition of the item.

2.1.2 Management of functional safety

The standard outlines guidelines for functional safety management. Therefore, definitions of organizational safety management and standards for safety life cycle for the development and production of products are provided, such as:

- **Hazardous Event**: combination of hazard and operational situation which can cause accident if not controlled by driver.
- **Safety Goal**: safety requirements assigned to the item meant for reducing risk of hazardous events.
- **Automotive Safety Integrity Level**: it represents a risk-based classification of the safety goal. Different ASIL level corresponds to different validation and development methods to be applied.
- **Safety Requirement**: it defines the specific hardware and software component requirement to satisfy in order to reach safety goal.

2.1.3 Safety Life Cycle

It starts with the **Concept phase**. This has to purpose to describe the item in terms of functionalities and interaction with other items. Then, the *hazard analysis and risk assessment* is performed. It is needed for understanding which can be failure within the item and following implications.
The output is the **ASIL** label that can go from *QM* to *D*.
Based on previous definitions, **functional safety concept** is performed. Hence, whether it is required, based on item critical level, what to achieve in order to prevent injuries to persons.
Once item goals have been stated, the development can go ahead following the V-shape model.
In order to perform the ASIL label, the following steps are required:

1. Item definition
2. Hazard analysis and risk assessment
3. Functional safety concept

Item Definition

This phase is necessary for declaring and specifying the item **functionalities** and the **elements** responsible for delivering them.
Another aspect to outline is the **interaction**: which are the items and elements (i.e. sensors) that exchanges data with the underdevelopment one. Indeed, all these

components have to be taken into account during the analysis.
The required or provided functionalities are important for functional safety purposes. Considering that a misbehaviour of another item occurs, the item has to be able to inhibit the incoming information and send a feedback to the end-user. This means that specific mechanisms have to be implemented

Hazard Analysis and Risk Assessment

The step has the purpose to identify and categorize hazards from malfunctions of the item. Basically, this is meant for studying what will be the effects of failures on the component on the functionality to be delivered.
The standard does not define a golden procedure to address this topic (mathematical model), but *suggestions*:

- **Situation Analysis**: operational situations and operating modes are described. Basically, when the item is going to be used.

- **Hazard Identification**: the misbehavior of the fault-affected item is defined. Hence, taking advantages of appropriate techniques, the hazards are systematically determined. Furthermore, combing operational situations and hazards, hazardous events and their consequences are defined.

- **Hazard Classification**: for each hazardous event, several parameters are assigned in order to obtain the ASIL grade.

    1. *Severity (S)*: measurement of the extent of hard to an individual. Its ranking goes from S0 to S3. S0 stands for "no injuries", while S3 equals to "life-threatening injuries".
    2. *Controllability (C)*: capacity of the driver or persons involved to avoid harm or damage within a certain time reaction. The ranking goes from C0 to C3. C0 is "controllable", while C3 is "difficult to control" or "uncontrollable".
    3. *Exposure (E)*: measurement of the probability of being in a certain operational situation which can lead to a hazardous event if failure occurs. The ranking goes from E0 to E4; where, E0 is "incredible" and E4 is "High likely".

    Once the previous parameters have been defined, it is possible to assign the ASIL label by crossing the values by means of a table.

- **Safety Goal Determination**: determination of the safety goal for each hazardous event associated with an ASIL different from QM.
  Therefore, if the hazard in a certain operational situation is relevant (from A to D), then methods for avoiding and mitigating risks are required. Hence,

safety goal has to be specified. The safety goals are the top-level safety requirements for the item and are expressed in terms of functional objectives to be reached.

Functional Safety Concept

During this phase, the functionalities for reaching the specified safety goals are determined.
This means that, besides the applications functionalities of the item, in case of high level ASIL additional functionalities must be added for guaranteeing safety of persons involved. These functionalities are known as **functional safety concepts**. In order to develop them, specific architectures, components and design techniques must be used. Typically, the techniques fall in two main categories:

- *Active Redundancy*: the item recognizes the failure and provides the safety functionality by reporting it to the end-user. Then, this last is responsible for changing the faulty item.
- *Passive Redundancy*: The end-user never perceives the fault, since the item continues to provide the expected functionality.

2.2 Physical and Virtual Prototypes

Now-a-days automotive marketplace is very competitive. Thus, it is growing the necessity to find out methods for validating in a reliable manner the always more sophisticated items. Indeed, within a modern vehicle a plenty number of heterogeneous devices and architectures are set.
The automotive companies are asked to develop complex devices in the shortest time possible. For this reason, traditional methodologies are becoming obsoleted. In fact, in order to develop and test a software, developers have to wait for the hardware to be produced. Thus, obtaining a reliable and effective analysis is crucial.
Therefore, new approaches are emerging based on prototyping techniques.
It is possible to determine two kinds of prototyping:

- Physical Prototyping,
- Virtual Prototyping.

Both of them share the same purpose: reducing the time needed for validating and testing an item and avoiding the unwanted costs of re-spins during the V-shape development flow.
The **Physical Prototyping** takes advantages of physical model using dedicated hardware devices for the implementation (i.e. FPGA).
On contrary, the **Virtual Prototyping** uses numerical and mathematical models

which emulates the real hardware as an executable software running on the host laptop.

2.2.1 FPGA-based prototype

This kind of approach allows to run the application software in real-time. It is an important aspect, since the automotive ECUs works based on timer modules, responsible for synchronization with crankshaft and camshaft to provide the correct fuel injection.
The obtained platform results to be flexible and efficient and it is able to validate code meant for running over timer modules [5].
The platform is able to generate the reference Crank and Cam signal and acquiring the output signals coming from timer module under test.
In this way, it is possible to verify and validate the timer modules in different configurations by comparison. Indeed, the FPGA-based platform is able to generate reference signals under specific conditions.
Furthermore, the platform is easy to be expanded and modified according to the customer test requirements.
In this sense, FPGA-based prototyping is cycle-accurate, since the software runs in a real-time manner. In addition, the FPGA allows high execution performances and easy ways for connecting the platform to real environment by interfacing layers.

2.2.2 Virtualization

Traditional techniques foresee the serially hardware and software development. These are sources of many fails and bugs. Furthermore, companies have to wait for the hardware before starting developing the code. Hence, the societies have to spend a lot of effort and face higher costs for providing a proper suitable software in time.
Due to these reasons, Virtual Prototyping development methodology is getting ahead.
This kind of approach allows to simulate inside a unique environment a whole heterogeneous system, such as automotive ECU. Therefore, this technique allows to develop, debug, integrate and validate the software without the need of a physical hardware platform.
In this way, the software can be developed in concurrency with respect to hardware. Furthermore, it allows different work-team to cooperate over the same item, working on different aspect simultaneously. This environment is very fast and flexible. Indeed, it can be easily modified by changing the requirement to be applied during the test. Moreover, it can cover digital, analog and mixed signal component, ensuring a large fault tolerance coverage [2].
A Virtual Prototyping platform is able to embed the whole software platform of an embedded system, such as basic software, operating system, and a numerical model

of the hardware platform (core, memories, and so on). Hence, the application software will see no difference between the execution over Virtual Prototype and real hardware platform [11].
The Virtual Prototype uses the same debuggers of hardware. Anyhow, since the software is not flashed on an actual hardware, less debugging cycles are needed. In addition, the virtual platform can be controlled by means of scripts (i.e. Tcl or Python) and connected to simulation environment such as Simulink for simulating a plant model.
The main issue regards the low time accuracy. Indeed, by means of a Virtual Prototype it is not possible to have a proper real-time behaviour. This is a crucial problem in terms of hardware development; anyhow, it does not affect the software development. In fact, the time execution results constant and platform independent.

Chapter 3

Virtual Independent Platform

The continuous increasing of the complexity of electronic control systems inside vehicles is making traditional testing and validation techniques obsolete. Car manufactures are searching for new methodologies, indeed.
Therefore, the main issue regards time: the more time spent over development and testing an item, the more cost is required. Moreover, in order to test heterogeneous items, such as automotive ECUs, huge efforts are needed.
In this sense, the following **Virtual Independent Platform** is meant. Thus, it is aimed to reduce the workload and time for testing an ECU, in particular the engine-control one.
The project is willing to provide a unique testing framework for validating an ECU taking advantages of heterogeneous tools and prototyping techniques.
Basically, starting from a common requirement database file (i.e. JSON), the Independent Platform will be able to produce and generate automatically testing routines (*Test Commands*) for the chosen testing target environment (i.e. HiL, VP, FPGA). Concurrently, considering the same requirements, *Mathematical Models* are produced and taken as **Golden Output** or **Reference Signals**. Once the tests have been performed, a successive comparison analysis is performed and failed or passed labels are assigned to each test.
In this way, the test engineer has just to properly define the requirements for each test and launch the platform. Hence, the work and time spent for testing are offloaded. In fact, the platform is responsible for generating test command files and performing analysis. Then, it is up to the user to construe the results.
In the following sections, an overview of the main modules is outlined. All the platform has been implemented using Python3, since it is HiL friendly.

3.1 Independent Python Platform Architecture

The **Validation Framework** is based on *Heterogeneous Prototyping* tools. It is meant for testing UUT (Unit Under Test) in a continuous and automatic integra-

tion fashion.
The main purpose is to extend the functionality of the platform work-flow in order to be able to enable test execution among different target environments (different models of same UUT).
The Platform creates *Mathematical Model* used as Reference (**Executable Requirements**). These *Requirements* should be compliant with inputs and outputs of the Hardware-In-the-Loop platform or others environments (i.e. Virtual Prototyping, FPGA-based). The proposed Platform is meant for test an ECU responsible of the engine management (Engine Management System). Anyhow, the framework architecture is quite general.

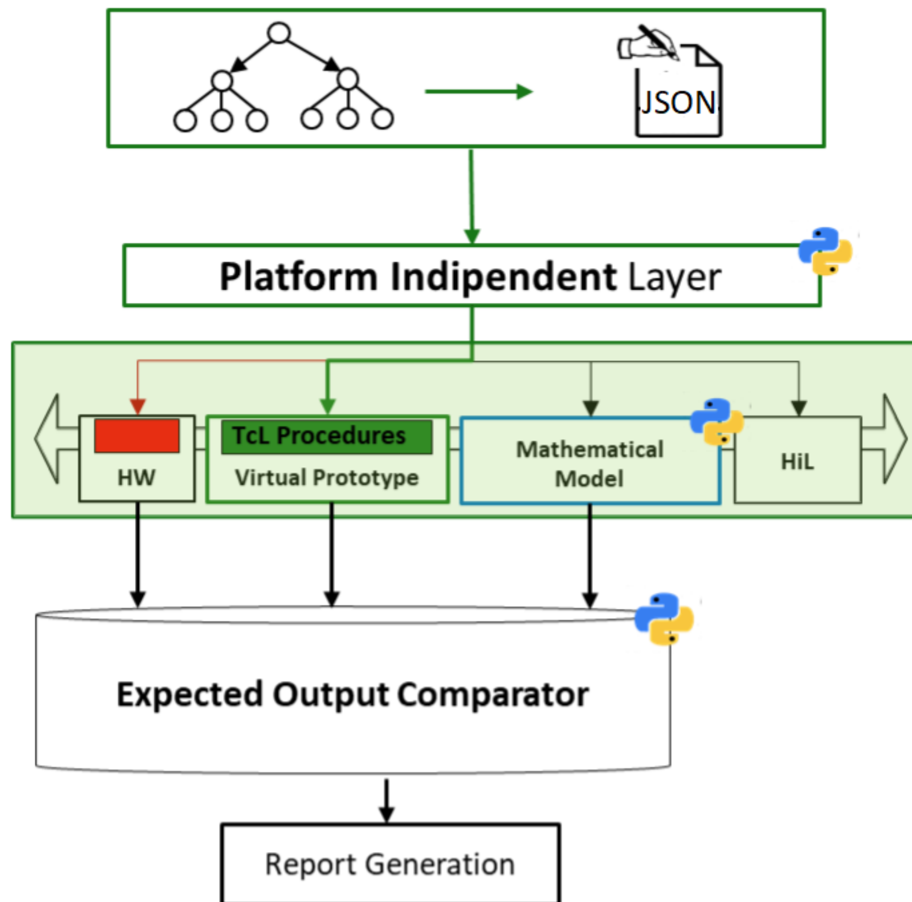

Figure 3.1: Test Environment Overview

As the figure 3.1 shows, the Virtual Platform receives configurations data from a database environment file, then it should make up the test procedure according to the target environment it is attached to.
Anyway, inside the Mathematical Model module, the Executable Output require-

ments are computed. Then, for further automatizing the process, a comparator compares all the platforms outputs and writes down a report.
The Platform has been developed as **modular** by using Python3. This modularity contemplates several and different python scripts which are responsible for generating, recording and plotting Inputs and Outputs requirements, based on a requirement database. Furthermore, it includes a module for producing Test Commands related to the specific model of the UUT attached to.
The main features of the Independent Python Platform are the following:

- **Requirement DataBase**, it defines the configurations of each test;
- **Test Manager**: based on the database, it is responsible for automatically managing the test execution;
- **Mathematical Models**: it is meant for generating inputs and outputs references.
- **Recording and Signal Plotting**: in charge for recording and storing signals for an eventually post-processing analysis;
- **Test Command Generator**: it generates parameters which are able to configure and initialize the tests over a specific environment.

The following chapter will show in details all these modules structures and functionalities.

3.2 Requirement DataBase

The *Requirement DataBase* defines the configurations of the test.
In order to achieve a complete description of all the used parameters a **Hierarchical** and **Graph-based** approach has been implemented.

Hierarchical description

Based on this approach, the Requirements are categorized in two classes:

- *Product Requirements* are the requirements that dealing with the system characteristics (i.e. HW frequency);
- *Process Requirements* are referred to the methodologies and the constrains for testing (i.e. Speed Management mode).

According to the Hierarchical Organization, the requirements are specified with name and values. It is possible to connect several requirements by means of logic operators in order to obtain more complex orders. The operators are inserted as

nodes. The connection within the same category can be both **and** and **or**. While, for linking different parameters categories, **and** operator has to be used.

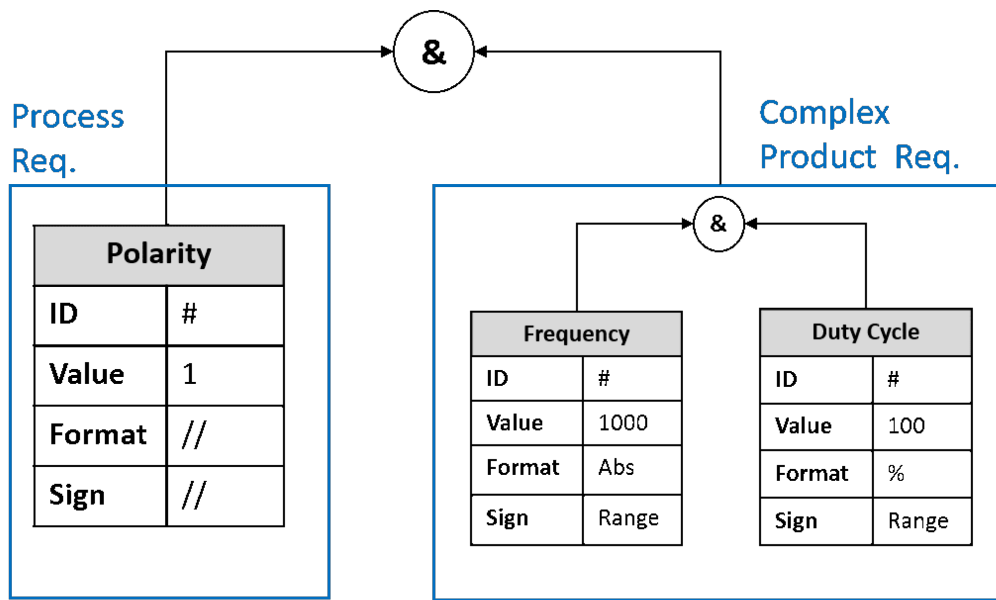

Figure 3.2: Process and Complex Product Requirements Verified simultaneously.

Graph-Based description

This approach leads to pack requirements on different layers. Each layer is identified by a set of nodes linked with the below ones.
The first node is called *Root* and defines the specific requirement.
From the root, some *branches* diffuse. At the end of each branch there is a *leaf*.
Leaves interconnected to the root by the same number of branches create a layer.
The leaves directly connected to root nodes are called **Macro-Parameters**; while, the others are referred to as **Basic-Parameters**.
With reference to figure 3.3, the *ID* node represents the root, hence the specific requirement. The layer below is where the Macro-Parameters are set and determines the name of each parameter. Going under, it is possible to find the Basic-Parameters layer, which specifies the name and value of each node belonging to the upper layer.

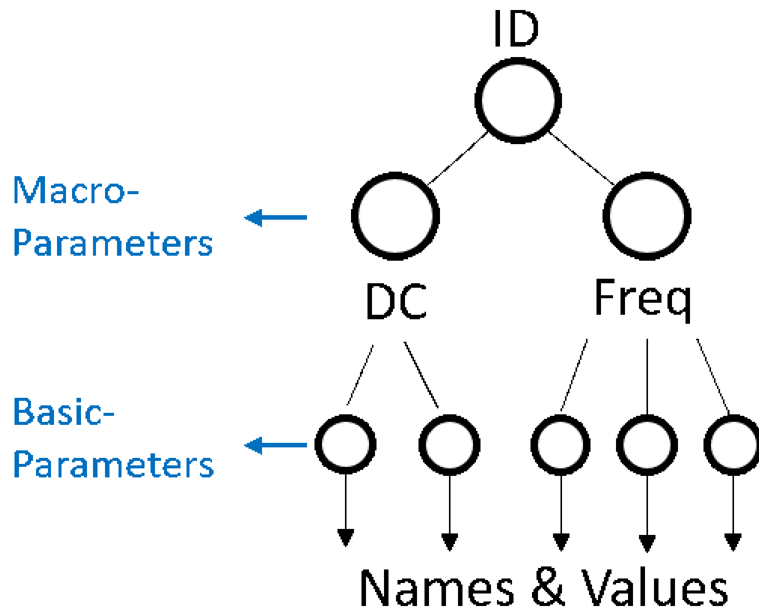

Figure 3.3: Graph Database Approach.

Notice that, in general, the parameters names are the same but are associated to different values according to the Macro-Parameter they are referred to.
Starting from this graphic description, it is possible to translate it into script readable file using meta-language (i.e. XML, JSON).

Input and Output Executable Requirement Classification

Every UUT has its own Input and Output requirements.
Inputs are related to signal requirements used for executing properly a Test Procedure; while, Outputs are needed for verifying and classifying whether the test is passed or failed.
Nevertheless, both Input and Output requirements need to be classified according to a certain classification method, due to their complexity and for verification purposes.
In order to achieve this, Hierarchical and Graph approaches are used. The following description will be focusing on Input Executable requirements, as it is possible to appreciate from figure 3.4.

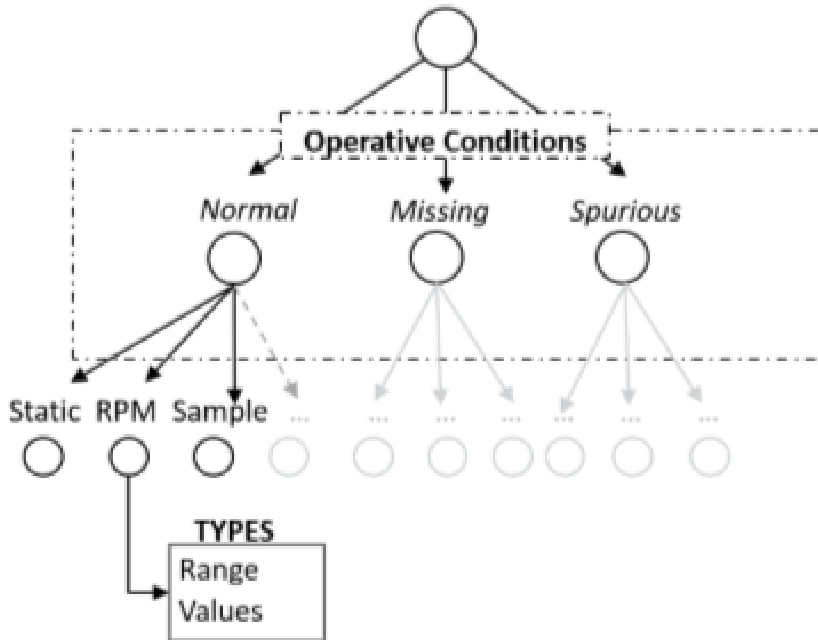

Figure 3.4: Graph Representation of Complex Requirement for CrankCam stimuli.

Starting form the top, the first root is corresponding to **CrankCam Requirements**.
Going to the layer right below, three different paths may be selected. These identify the **Operative Conditions** layer : *Normal*, *Missing*, *Spurious*.
An Operative Condition allows to select the proper operational mode for generating the corresponding Input Executable Requirement.
For each of them, a certain number of parameters has to be taken into account. The parameters are represented as leaves.
A parameter is specified by a *range* of possible values and by an alpha-numeric or Boolean *value*. In order to generate a specific test, it is needed to fix a value for each one of the parameters.
The benefit of such method for Requirement representation is that it is very easy to extend by adding more Roots and Leaves.
Therefore, the Graph method allows adding extensions in two different ways:

- *Horizontal*, which is used for adding furthermore roots if required (figure 3.5);

- *Vertical*, for extending the number of layers of a specific root (figure 3.6).

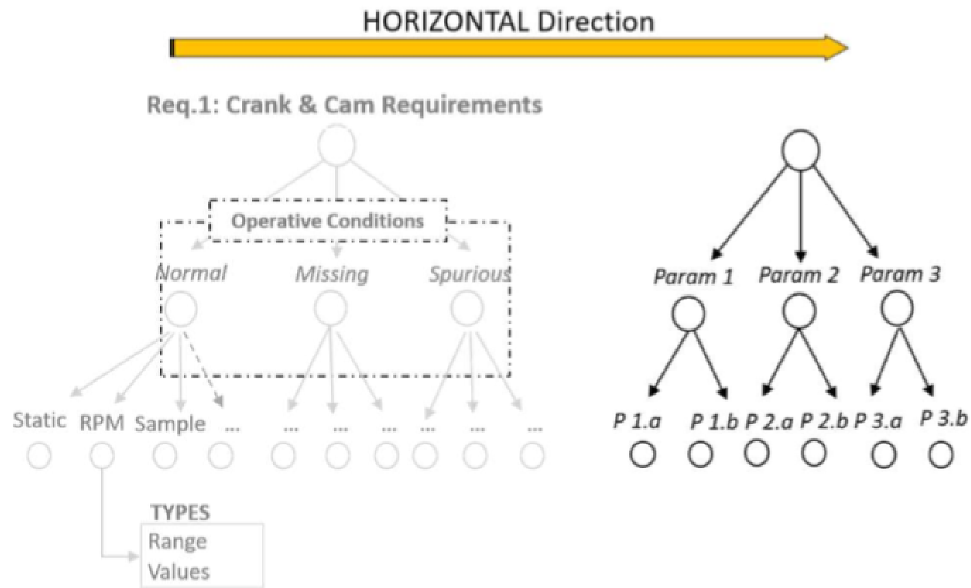

Figure 3.5: Horizontal extension.

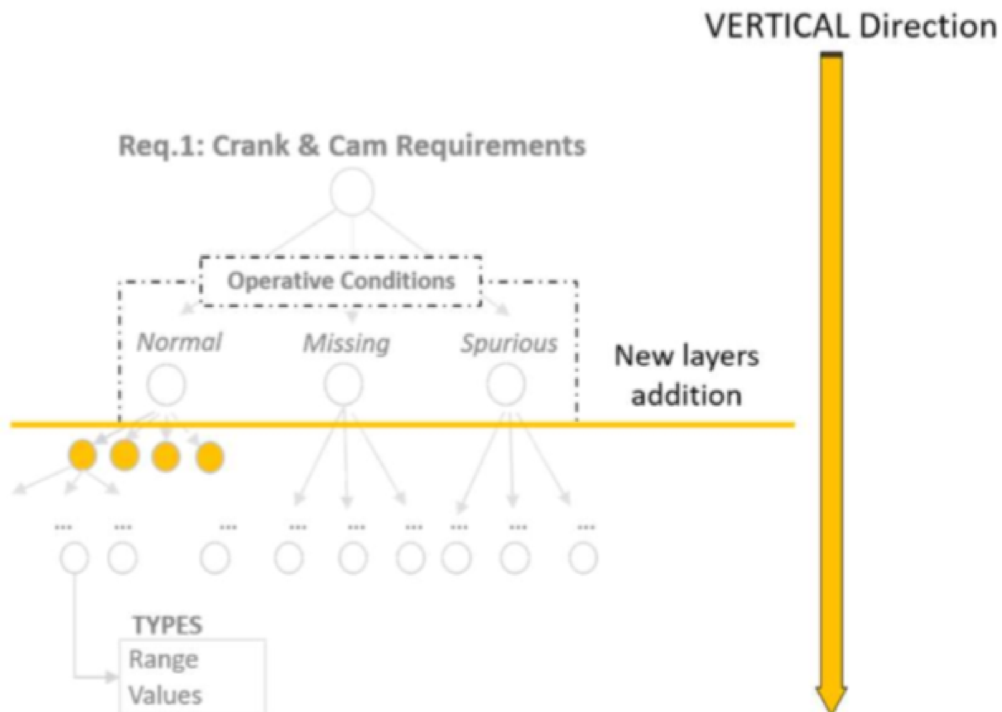

Figure 3.6: Vertical extension.

Requirement Database: JSON file

The Requirement Database defines the test configurations. In the Platform it is implemented using **JSON** file format.
The JSON (JavaScript Object Notation) is an open standard, used for data transmission. This file format is very flexible and fast; moreover, it is easy to be loaded within a python script.
By means of the JSON file, it is possible to provide multiple test orientation configurations, hence several different tests. In this way, it is easy to introduce as many tests as required. Indeed, by charging and changing the specific configurations of the parameters, different tests are able to be loaded.
Each test parameters can be divided into **four sections**:

- *General Parameters*: this class includes the general configurations of the test; so, whether the test is *Binary/Analog*, its *HW frequency*, the *Number of Revolutions* to be performed, the RPM mode of the test, and so on so forth.
- *Crank Parameters*: it introduces parameters related to the Crank signal generation, such as: *Crank Initial Position*, *Crank reference Transition Jitter*, and so on.
- *Cam Parameters*: it contains parameters related to Cam signal, like: *Number of Cam to generate*, *Crank to Cam Shift.*
- *Injection Pulse Parameters*: it defines the parameters for the Injection Pulse executable output. For each one, the *Number of Injection Pulse for each revolution* and the *Configuration of the pulses* have to be defined. It is also important to specify whether the Pulse is Time-based (*DWellPeriod, Width*) or Angular-based (*Start Angle "SOI", Width*).

If a new test has to be added, then it is necessary to define all the above parameters.

3.3 Test Manager

The final purpose of the Platform is to offer an unique environment for testing the ECU by means of heterogeneous tools. Hence, it has to be able to communicate correctly with different environments (i.e. HiL, VP, FPGA).
The main core of the software platform is the **Test Manager**. Basically, it is the middleware software and is responsible for elaborating test requirements and generating the corresponding sequences of executable requirements and test patterns.
The *Test Manager* has to provide several features, such as:

- Distinguishing reference values to the specific tagged requirements, performed by a direct mapping between the references values and the requirements property.

- Generation of a full test procedure for a given application software. The user should be able to access directly to test configurations; while, the test patterns should be automatically defined with respect to test configurations.
- Communicating the configuration requirements to infrastructure layers for executing the test over the desired environment.
- Generation of control commands for executing properly the test upon the desired environment. These commands include the reset and starting processes towards application software executed on the chosen environment tool.
- Control of the output report generation. The platform will receives back results from several environments and will compare them, taking care that all of them have the same type, format and organization.

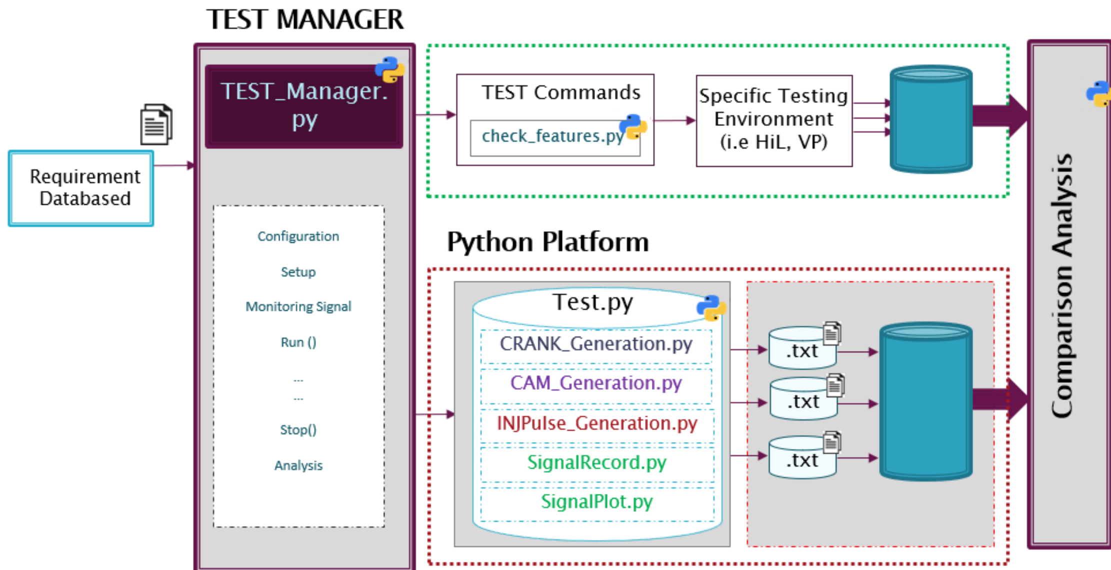

Figure 3.7: Test Manager work-flow.

Hence, basically, the Test Manager is responsible for two main duties:

- Automatic generation of test commands for running a different environment;
- Automatic execution of the python platform to generate the expected executable requirements.

As it is possible to see in the figure 3.7, this module is responsible for running and launching all the procedures related to the test management.
Once the Requirements database has been loaded, the Test Manager launches the

Test.py and the *check features.py*.
The first is responsible for generating and recording the Executable Requirements; while, the second produces the set of test commands for the target environment. Once the Executable requirements are collected, the corresponding .txt files are placed inside the folder associated to that test for post-processing analysis.

3.4 Mathematical Models

The Independent Python Platform is dedicated to the generation of input and output requirement signals (Crank, Cam, Injection Pulse). These are also called *Mathematical Models* or *Executable Input/Output Requirements* and are meant to mimic the behaviour of different functions.
The required stimuli and golden outputs are generated according to the test configurations embedded inside the JSON file, such as: RPM, HW frequency, number of revolutions, type of signal. Since the produced signals have to be compatible with other environments, they have to share the same basic resolution. Hence, the HW frequency has to be taken into account for evaluating it.
The executable input requirements can be in **Normal**, **Missing**, and **Spurious** conditions. Signals can be produced in **Binary** or **Analog** mode. For analog signal generation, it is convenient to consider *white noise* affecting the main signal. The module is implemented in such a way that the frequency of the analog signal is flexible.
Regarding the Executable Output Requirements, they are developed by libraries of functions which take the configuration parameters as input arguments. The mathematical model generation is modular-based; hence, every module is dedicated to a specific task. All this modules are called by the *Test.py*, which is responsible for managing the virtual generation of the executive requirements and the related recording and plotting.
With reference to figure 3.7, inside the Test.py module, it is possible to see 5 more modules:

- *Crank generator*
- *Cam generator*
- *Injection Pulse generator*
- *Signal Recording*
- *Signal Plotting*

Crank Module

The Crank signal is made up by teeth repetitions equally spaced. A tooth is considered as a reference transition.
Conventionally, one revolution is constituted by 120 teeth or transitions. It is also called **Sample** and is equal to two complete rotations (720°). At the end of each rotation (360°), there are two deleted teeth (the 59th and the 60th). The falling edge of each tooth is referred to *Falling Reference Transition*. While, the rising edge is called *Rising Reference Transition*.
The Crank module is dedicated to Crank signal generation. The software receives test configuration parameters both General and Crank specific ones, such as RPM, resolution, revolutions number, Crank initial position.
Their introduction is totally automatic and managed by Test Manager which parses the JSON file.

Cam Module

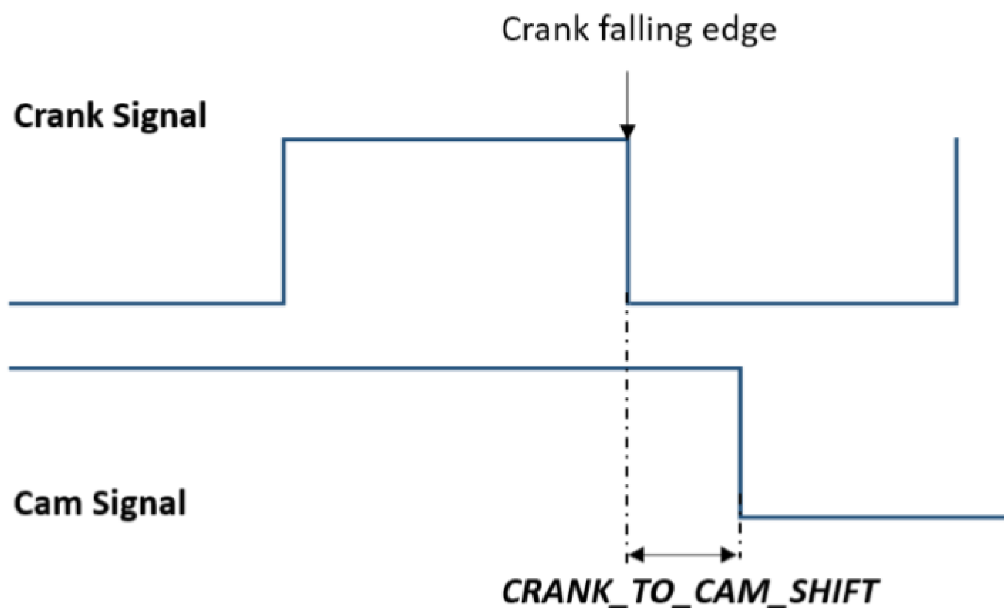

Figure 3.8: Crank to Cam Shift.

Despite to Crank, the Cam signal is made up of *not equally spaced* reference transitions. Inside a revolution (720°), **8 Cam events** take place.
As for Crank module, the configuration parameters are given as arguments by Test Manager.
Thanks to the parametric and feasible structure of the JSON file, it is possible to

easily modify the configuration and add several tests.
The main parameter of this function is **Crank To Cam Shift**. Basically, it is the value in degrees of the shifting between Crank and Cam signals (figure 3.8).

Injection Pulse Module

This module is responsible for the Injection Pulse generation. Inside one cycle, we have 8 Injection pulse events. Among them, 4 of these are **angular based**.
Therefore, considering the requirement *Start Angle*, the pulses are starting and continuing for the required *Width* of the pulse.
The other **temporal mode** pulses are generated based on *Dwell*. And, as for the previous ones, they are started and continuing for *Width* of the pulse.
Also in this case, the configuration values are introduced by parsing the JSON file with a parametric structure. This allows flexibility in modifications.

Signal Recording and Plotting modules

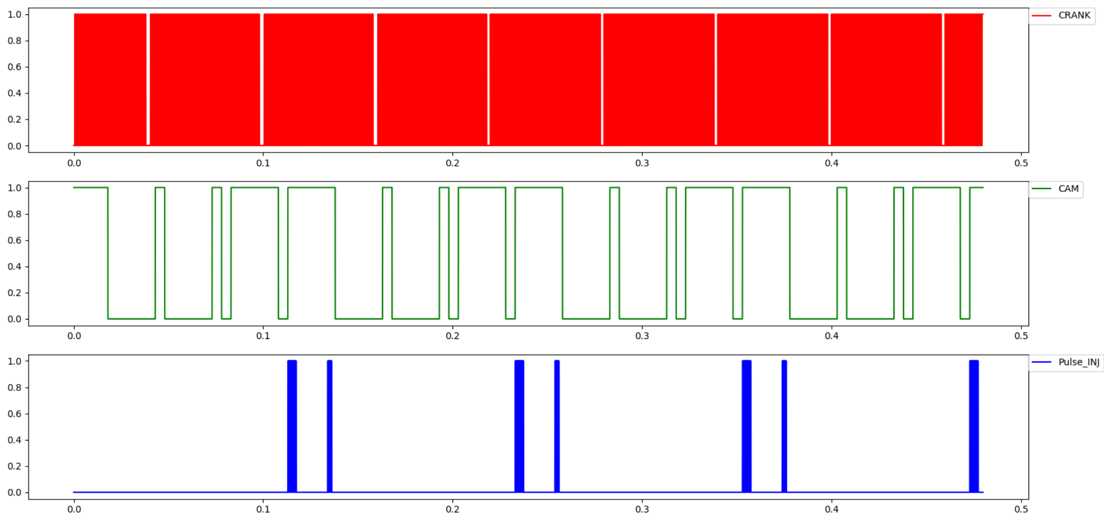

Figure 3.9: Crank (red), Cam (green) and Injection Pulse Signals (blue).

These modules are needed for plotting and storing the generated mathematical models. It is important to record Golden output for post-processing analysis and comparison with results coming from other environments.
All the records are stored as **.txt** files in a specific folder. Each folder contains results for one single test. Crank, Cam and Injection Pulses signals are recorded by

sensing reference transitions for each test. Therefore, once the reference transition is detected, the recording function is triggered and writes inside the file the corresponding signal and time values.
While, the plotting module simply represents the produced signals.

3.5 Test Commands

Once requirements are defined, it is possible to implement a Test Pattern Generation (Test Commands) policy. It can be managed in three ways:

- *Manual*: Test engineer manually writes environment file containing the value of each parameters.
- *Automatic*: the test pattern is generated without user contribution. I.e. python script parses the requirement scheme and chooses random values for every layer.
- *Mixed*: mash up of the previous methods. In this case, user can express preferences; the script generates the file taking and giving priority to user choices. All the other parameters are chosen in automatic way.

One goal to be accomplished by the Test Manager is to generate the **Test Commands** for running other environments according to the previous described policy, i.e. HiL.
Test Manager starts by receiving the requirements from the JSON file and passing them automatically to a python module responsible for generating Test Command (**CheckFeatures.py**).This script is necessary for executing the tests over the HiL platform. Each CheckFeatures script corresponds to a test introduced in JSON file. Hence, for each test introduced in the JSON file with different configuration, such a script is generated. The CheckFeatures contains parameters for the Test Environment. These are required for initialization phase, configuring the signal recorder and running the test.

As it has been shown, the Software Platform is meant for managing the entire test process. Anyhow, it is underdevelopment. In particular, the **Mathematical Models** related to *Crank Signal* have been implemented for this Master Thesis project. The implementations description and corresponding results are presented in the following chapters.

Chapter 4

Mathematical Models

4.1 Introduction

The ECU responsible for controlling the engine is the most important among the ones possible to find inside a modern vehicle. This has to manage the fuel injection in order to guarantee an efficient and powerful engine. Indeed, a good management of the injections helps to reduce fuel consumption and air pollution emissions. On other hand, it allows to maximize power conversion and to obtain a more efficient injection control.
The embedded systems devoted to this purpose takes advantages of timer modules. In this way, it is possible to generate real-time signals and synchronize fuel injection inside the different cylinders.
In general, two signals coming from the engine are taken as references: **Crank** and **Cam**. These signals are used to understand which is the engine position in a certain given time.
Basically, the automotive microcontroller performs task based on the engine angular position, in particular the cylinders position with respect to the crankshaft. In this sense, a complex model of the Crank signal is needed.
In order to have more complete Executable input requirement, the Platform Independent Layer has been updated.
Therefore, different conditions have been developed for having mathematical models closer to real system.
The attention has been focused upon **Crank Signal**. Starting from a Crank signal in Normal condition, other scenarios have been developed: Missing tooth, Spurious tooth, Reference Transition Jitter.
Furthermore, dynamic behavior has been faced following two different approaches: time-based and angle-based.
In the following sections, the implementations of models related to such signals are proposed.

4.2 Crank Signal

The Crank Signal represents the position of the crankshaft of a combustion engine (either petrol or diesel).
It is measured by a crank sensor and is used by Engine Management System for controlling fuel injection and ignition system timing.
Together with the Cam, these signals are used for evaluating and monitoring engine pistons and valves relationship. Thus, Crank and Cam sensors are the most important sensors inside modern engines.
Their main purpose is to synchronize the starting of four strokes engine. Indeed, the EMS is able to evaluate fuel injection.
Furthermore, Crank signal is used for measuring the engine speed as RPM. Basically, the Crank is a square wave signal. Each tooth is identified with a rising and falling edge transition. A falling edge is considered as a reference to estimate the crankshaft relative rotation.
The sensor responsible for detecting the crank signal is stationary and works in conjunction with a toothed ring or phonic wheel. The ring is in based reference condition with respect to the crankshaft. Actually, depending on the kind of sensor used, the crank signal may be generated in several ways.
The sensor creates a magnetic field in between the two edges of its sensitive terminals. By rotating, the teeth of the ring interrupt the magnetic field, producing a square wave as output [7]. Or the sensor generates a pulsing voltage signal, where each reference transition is linked to one tooth of the phonic wheel. It is important to underline that two teeth of the reluctor ring are missed, for synchronization purposes [6].
Considering that the crankshaft phonic wheel is made up 60 teeth, each tooth of the Crank signal corresponds to a $6°$ rotation. Furthermore, the 59th and 60th teeth are missed. This *gap* is needed for understanding the angular position of the engine.
Anyhow, it is not possible to determine whether the engine is in intake, compression, power or exhaust phases (**4-stroke engine**) basing on Crank signal only. Hence, Cam signal is required and consists on few pulses in synchronization with the crankshaft.
Furthermore, the crank signal can be affected by several misbehaviour. Indeed, since we are dealing with a complex heterogeneous system, there can be errors of different nature that can depend on the system its-self or by the noisy environment it works into. Hence, for being able to predict and meet functional safety requirements, models for testing and validating the item under faulty or **stressed conditions** are required.
In the following sections, crank signal models will be analyzed under several conditions, such as:

- *Normal Crank Signal*

- *Crank Signal in Missing Tooth Condition*
- *Crank Signal in Spurious Tooth Condition*
- *Crank Signal in Reference Transition Jitter Condition*

Then, *Dynamic RPM management* models will follow.

4.3 Normal Crank Signal Generation

The **Normal Crank Signal Generator** module allows to produce the Crank signal when Normal condition is set.
This function is considered as a setup function; hence, the other functions use these and their own parameters.
At first, the Test Manager takes the parameters from the JSON file and pass them to the *Test.py* module as arguments.
These parameters are used inside procedures in order to generate the signal. The *Test.py* module launches the *Normal Crank* module, sharing parameters both General and Specific.
The General configuration parameters used are:

- *Condition*: it defines whether the signal is **Binary** or **Analog**. If Analog is set, *white noise* affecting the main signal is considered for the generation.
- *Revolution Number*: it indicates the number of revolutions the crankshaft performs. It is important to underline that 1 revolution is equal to 720°.
- *Hardware Frequency*: it is considered for evaluating the **Resolution**. So, the minimum sample possible to discriminate.
  This parameter is important for having compliance with respect to other environments (i.e. Virtual Prototyping, FPGA-based, HIL).
- *Test RPM State*: it can be **Static** or **Dynamic**. Hence, it defines whether the crankshaft is subjected to acceleration/deceleration or it has a constant speed for all the test execution. Dynamic approach will be explained later on.
- *Test RPM Static*: it points out the **RPM** value when static condition is selected.
- *Test Case Execution Time*: it indicates the total amount of time to perform the simulation.

While the Specific parameters used are defined as:

- *Crank Initial Position*: it defines the index of the first tooth where the Crank signal starts from.

- *Crank Signal Is Directional*: this is a Boolean parameter which points out whether the Crank signal is managed as *mono* or *bi-directional*.
  If it set to **False**, the Crank signal is a square wave with 50% of *Duty Cycle*.
  While, if it set to **True**, the last two parameters have to be taken into account.

- *Crank Forward Rotation Active Period*: if this float parameter is different from 0, then it is used for evaluating the Duty Cycle as:

$$DutyCycle = \frac{CrankForwardRotationActivePeriod}{ToothPeriod} \tag{4.1}$$

.

- *Crank Backward Rotation Active Period*: if it is different from 0, then it used for evaluating the Duty Cycle as:

$$DutyCycle = 1 - \left(\frac{CrankBackwardRotationActivePeriod}{ToothPeriod}\right) \tag{4.2}$$

.

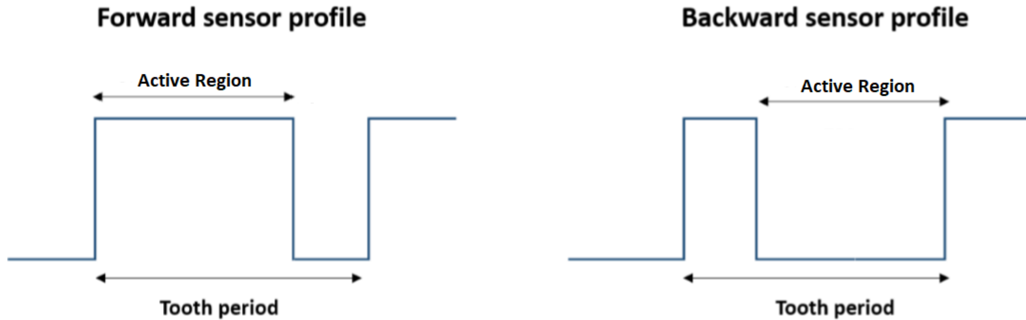

Figure 4.1: Forward and Backward Active Period

Once the parameters are loaded on *Test.py*, it launches the *Normal Crank Generator* procedure, transferring the required arguments.
At first, the Duty Cycle is evaluated (4.1) (4.2); then, considering the previously created time vector t, a square wave is generated as follows:

$$SquareWave = t\%Tooth_{period} > Tooth_{period} \times DutyCycle \tag{4.3}$$

The inequation (4.3) has as first argument a modulus operation. This divides each cell by the tooth period and returns an array containing values of the remainder of each operation.
The inequality comparison among the several cells and the float division returns a Boolean array. Within each cell a *True* or *False* is set depending on whether the

Parameter Name	**Range**	**Resolution**
Condition	Binary/Analog	
Revolution Number	>=1	1
Hardware Frequency	2000 Hz	
Test RPM State	Static/Dynamic	
Test RPM Static	>=1000	
Crank Initial Position	1,120	1 Tooth
Crank Signal Is Directional	True/False	
Crank Forward Rotation Active	0, Tooth_{Period}	1μs
Crank Backward Rotation Active	0, Tooth_{Period}	1μs

Table 4.1: Crank signal General and Specific Parameters.

condition is met or not. Then, a cast operation follows to turn it into an integer array.
In order to obtain a Crank signal, it is needed to insert *gap* in between every 360°. Hence, rising and falling edges are counted. When multiples of 59 and 60 are met, the corresponding teeth are set to 0.

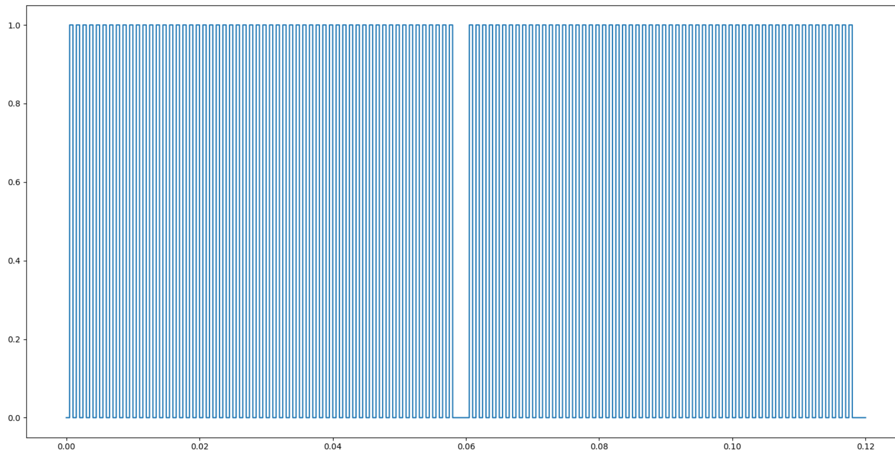

Figure 4.2: Crank Signal generated with Crank Initial Position set to 0

In the figure 4.2, it has been generated a Crank signal in Binary and Static mode, with RPM equal to 1000, Crank Initial position set to 0 and Duty Cycle 0.5. The figure represents 1 revolution cycle only.
Further graphical results may be found in the following chapter.

4.4 Crank Missing Tooth Condition

The Missing Tooth Condition occurs when one or more teeth of the Crank Signal are missed or deleted.

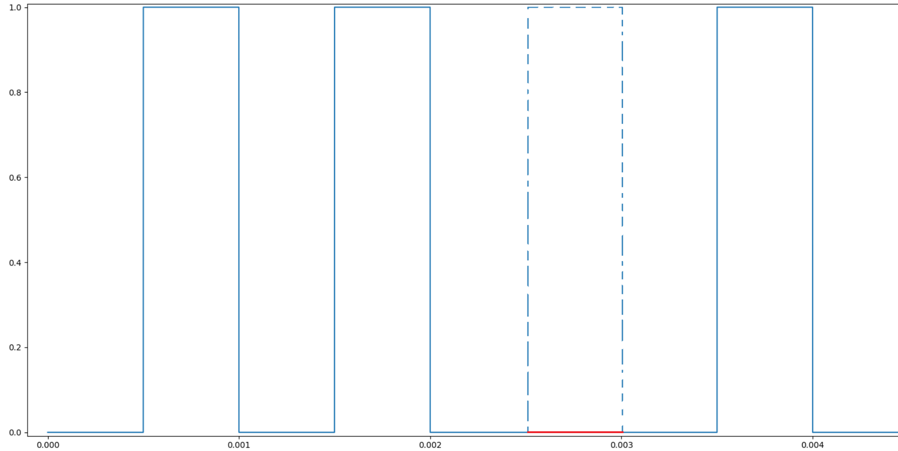

Figure 4.3: Crank Signal affected by a single Missing tooth occurrence

In order to provide a proper model of this event, a module has been developed, called *CrankMissingTooth.py*.
This module is integrated inside the main python platform and communicates with the requirement database JSON file. It uses General (i.e. Normal Crank) and Specific Parameters.
The Missing Tooth parameters are:

- *Missing Tooth ID*: it indicates the index of tooth to be deleted;
- *Missing Tooth Cycle*: it indicates the sample within the tooth has to be deleted;
- *Missing Tooth Repetition*: it indicates the number of consecutive occurrences of Missing tooth event.

Numerical details and features of Missing tooth parameters are shown in table 4.2
Also these parameters have been included inside the JSON requirement database by creating the subclass (**Missing tooth parameters**) inside the pre-existent Crank parameter class (figure 4.4).

Parameter Name	**Range**	**Resolution**
Missing Tooth ID	1,120	1 Tooth
Missing Tooth Cycle	>=2 , <= num of sample	1
Missing Tooth Repetition	1,120	1 Tooth

Table 4.2: Missing Tooth Specific Parameters.

```
"CRANK_parameters": {
    "CRANK_INITIAL_POSITION": 0,
    "CRANK_SIGNAL_IS_DIRECTIONAL": 0,
    "CRANK_SENSOR_DELAY": 0.5,
    "CRANK_FORWARD_ROTATION_ACTIVE_PERIOD": 26,
    "CRANK_BACKWARD_ROTATION_ACTIVE_PERIOD": 65,
    "Missing_tooth_active": "False",
    "Missing_tooth_parameters": {
        "Missing_tooth_ID": 3,
        "Missing_tooth_cycle": 2,
        "Missing_tooth_repetition": 3
    },
    "Spurious_tooth_active": "False",
    "Spurious_tooth_parameter": {
        "Spurious_tooth_ID": 4,
        "Spurious_tooth_cycle": 2,
        "Spurious_tooth_delay": 5.000000e-06,
        "Spurious_tooth_period": 5.000000e-06,
        "Spurious_tooth_repetition": 20
    },
    "CRANK_REFRENCE_TRANSITION_JITTER": 0.0005,
    "CRANK_REFRENCE_TRANSITION_JITTER_parameters": {
        "CRANK_REFRENCE_TRANSITION_JITTER_ID": 61,
        "CRANK_REFRENCE_TRANSITION_JITTER_cycle": 3
    }
},
```

Figure 4.4: Missing tooth parameters embedded into .json file

In order to generate a Missing Tooth Crank signal, the above figure is considered. Inside the Normal Crank Generation Module, the *Missing Tooth Active* Boolean variable is checked. When the if-condition is verified, the Crank Missing Tooth module is called. The input arguments of this module are as mentioned before General and Specific Parameters. An important aspect to underline is that both Normal and Missing Tooth Crank share the same resolution.
The Missing tooth module starts considering the first tooth to be deleted. Therefore, evaluating where the Missing tooth event has to begin and end, thus the first tooth. The Crank signal is scrolled and when the corresponding time vector fits the previous

computed range, the Crank signal is set to 0.
By iterating the process for *Missing Tooth Repetition* number of times, the Crank signal affected by Missing Tooth Condition is modelled.
Then, the module returns the new Crank signal to the Normal Crank Signal module for keeping the execution.

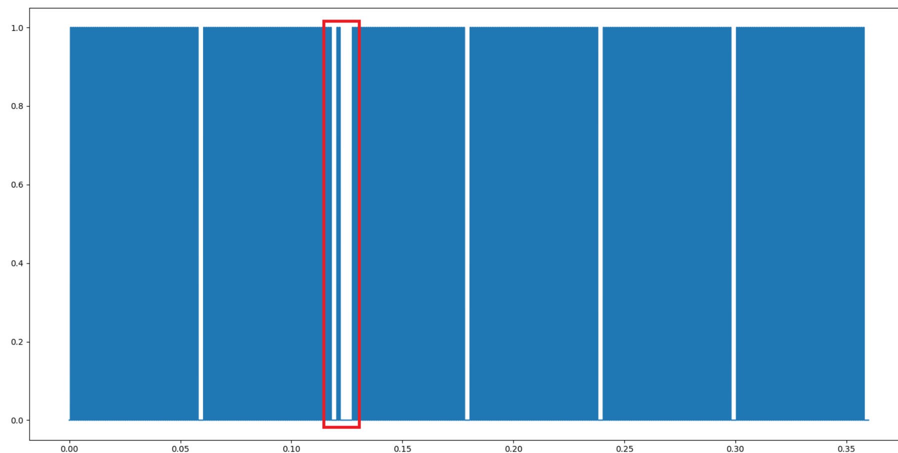

Figure 4.5: Example of Crank signal in Missing Tooth Condition

The figure 4.5 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 0; Missing Tooth ID = 3; Missing Tooth Cycle = 2; Missing Tooth Repetition = 5. Further graphical results may be found in the following chapter.

4.5 Crank Spurious Tooth Condition

The **Spurious Tooth Condition** happens when extra pulses appear within two consecutive Crank Teeth, without causing time-shifting over the signal.

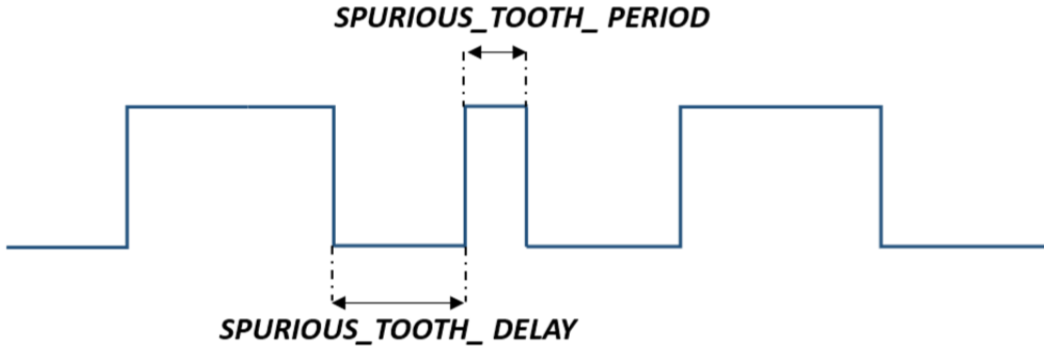

Figure 4.6: Crank Signal affected by a single Spurious tooth occurrence

For representing an adapt model of this phenomenon, a python module has been proposed (*CrankSpuriousTooth.py*) and integrated inside the main python platform. This module works with General and Specific parameters.
The .json file has been updated with the Spurious Tooth Parameters, by adding a new subclass to the Crank parameter class.
The Spurious Tooth Module takes advantages of the following parameters:

- *Spurious Tooth ID*: it represents the index of tooth which reference transition is used to insert the extra spurious pulse;
- *Spurious Tooth Cycle*: it indicates the sample within the spurious tooth condition occurs;
- *Spurious Tooth Delay*: it denotes the amount of time between the selected reference transition and the begin of spurious pulse;
- *Spurious Tooth Period*: it tells the duration of the spurious tooth;
- *Spurious Tooth Repetition*: it indicates the number of consecutive occurrences of Spurious tooth event.

Numerical details and features of Spurious tooth parameters are shown in table 4.3

Parameter Name	**Range**	**Resolution**
Spurious Tooth ID	1,120	1 Tooth
Spurious Tooth Cycle	$>=2$, $<=$ num of sample	1
Spurious Tooth Delay	0, Crank Period	1μs
Spurious Tooth Period	0, Crank Period	1μs
Spurious Tooth Repetition	1,120	1 Tooth

Table 4.3: Spurious Tooth Specific Parameters.

As mentioned above, all these parameters have been integrated inside the JSON file (figure 4.7).

```
"CRANK_parameters": {
    "CRANK_INITIAL_POSITION": 0,
    "CRANK_SIGNAL_IS_DIRECTIONAL": 0,
    "CRANK_SENSOR_DELAY": 0.5,
    "CRANK_FORWARD_ROTATION_ACTIVE_PERIOD": 26,
    "CRANK_BACKWARD_ROTATION_ACTIVE_PERIOD": 65,
    "Missing_tooth_active": "False",
    "Missing_tooth_parameters": {
        "Missing_tooth_ID": 3,
        "Missing_tooth_cycle": 2,
        "Missing_tooth_repetition": 3
    }
    "Spurious_tooth_active": "False",
    "Spurious_tooth_parameter": {
        "Spurious_tooth_ID": 4,
        "Spurious_tooth_cycle": 2,
        "Spurious_tooth_delay": 5.000000e-06,
        "Spurious_tooth_period": 5.000000e-06,
        "Spurious_tooth_repetition": 20
    },
    "CRANK_REFRENCE_TRANSITION_JITTER_parameters": {
        "CRANK_REFRENCE_TRANSITION_JITTER_ID": 61,
        "CRANK_REFRENCE_TRANSITION_JITTER_cycle": 3
    }
},
```

Figure 4.7: Missing tooth parameters embedded into .json file

In order to produce a proper Spurious tooth crank signal, the Boolean *Spurious Tooth Active* is checked inside the Normal Crank module.
When the if-condition is verified, the *Spurious Tooth Module* is called, taking as arguments several parameters (General and Specific).
It is important to highlight that both Normal and Spurious Tooth Crank share the same resolution.
The Spurious Tooth Module starts identifying where the first extra pulse is set inside the Normal Crank Signal. It evaluates also the Spurious tooth total duration.
At this point, the Crank signal is scrolled. When the corresponding time vector fits the interval previously computed, two scenarios occurs:

1. When the time vector is within the interval included between the *extra tooth begin* and the *extra tooth delay*, then the Crank Signal is set to 0;

2. When the time vector is inside the interval included between the *extra tooth delay* and *period*, the Crank signal is set to 1.

Whether the Spurious Tooth Repetition Number is greater than 1, the process is iterated for a number of times equal to the repetition value.
Then, the Spurious module returns the modelled Spurious Crank signal to the Normal Crank module and the execution continues.

Figure 4.8: Example of Crank signal in Spurious Tooth Condition

The figure 4.8 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Spurious Tooth ID = 2; Spurious Tooth Cycle = 2; Spurious Tooth Delay = 100μs; Spurious Tooth Period = 100μs; Spurious Tooth Repetition = 1. Further graphical results may be found in the following chapter.

4.6 Crank Reference Transition Jitter

The **Crank Reference Transition Jitter** occurs to one of the teeth of the Crank Signal, causing time-shifting to all the successive teeth.

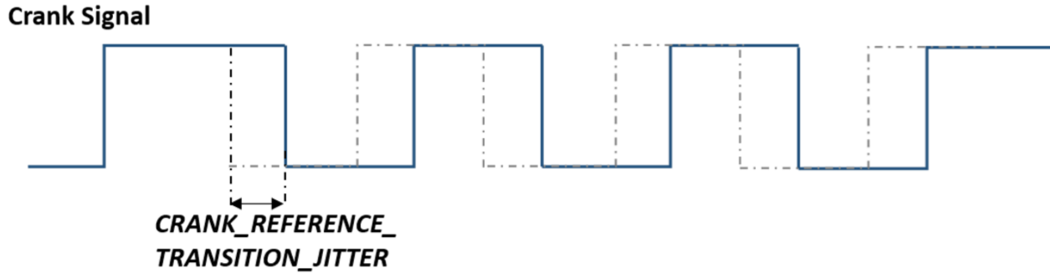

Figure 4.9: Crank Signal affected by Reference Transition Jitter

A module has been developed for providing this kind of effect. It is called **Crank Jitter** and is integrated inside the main python platform.
This module uses several parameters, both General and Specific ones. The *Crank Reference Transition Jitter parameters* have been embedded as a subclass inside the General Crank parameter class in the .json file.
The Reference Transition Jitter Module takes advantages of the following Specific parameters:

- *Crank Reference Transition Jitter*: it represents the amount of extra time added to the tooth affected by Jitter;
- *Crank Reference Transition Jitter ID*: it indicates the index of the tooth affected by Jitter;
- *Crank Reference Transition Jitter Cycle*: it denotes the revolution within the Jitter occurs.

Numerical details and features of Reference Transition Jitter parameters are shown in table 4.4

Parameter Name	**Range**	**Resolution**
Reference Transition Jitter	0, Tooth Period	1μs
Reference Transition Jitter ID	1, 120	1 Tooth
Reference Transition Jitter Cycle	>=0, <= num of sample	1

Table 4.4: Reference Transition Jitter Specific Parameters.

All these parameters have been inserted inside the JSON file (figure 4.10).

```
"CRANK_parameters": {
    "CRANK_INITIAL_POSITION": 0,
    "CRANK_SIGNAL_IS_DIRECTIONAL": 0,
    "CRANK_SENSOR_DELAY": 0.5,
    "CRANK_FORWARD_ROTATION_ACTIVE_PERIOD": 26,
    "CRANK_BACKWARD_ROTATION_ACTIVE_PERIOD": 65,
    "Missing_tooth_active": "False",
    "Missing_tooth_parameters": {
        "Missing_tooth_ID": 3,
        "Missing_tooth_cycle": 2,
        "Missing_tooth_repetition": 3
    },
    "Spurious_tooth_active": "False",
    "Spurious_tooth_parameter": {
        "Spurious_tooth_ID": 4,
        "Spurious_tooth_cycle": 2,
        "Spurious_tooth_delay": 5.000000e-06,
        "Spurious_tooth_period": 5.000000e-06,
        "Spurious_tooth_repetition": 20
    },
    "CRANK_REFRENCE_TRANSITION_JITTER": 0.0005,
    "CRANK_REFRENCE_TRANSITION_JITTER_parameters": {
        "CRANK_REFRENCE_TRANSITION_JITTER_ID": 61,
        "CRANK_REFRENCE_TRANSITION_JITTER_cycle": 3
    }
},
```


Figure 4.10: Reference Transition Jitter parameters embedded into .json file

During the execution of the Crank Normal Generator module, the *Reference Transition Jitter* parameter is checked. If it is different from 0, the *CrankJitter.py* module is launched. It receives General and Specific parameters. It is important to highlight that both Normal and Crank Jitter signal share the same resolution.
Starting from General parameters, the module first evaluates the timing characteristics for one normal tooth.
Then, the Reference Transition Jitter features are computed. In particular, its start and end times based on its length (*ReferenceTransitionJitter*), the affected tooth and the revolution within it takes place.
The time vector associated to the Normal Crank signal is scrolled and the Normal Crank signal is generated.
Concurrently, it is checked whether the time vector value is inside the range of the reference transition jitter start and end. If the condition is verified, the Crank signal value is set to 1.
Furthermore, the counter associated to the Crank signal timing features are increased of a time shift corresponding to the Crank Reference Transition Value. In

this way, all the successive teeth will be shifted. Then, the module returns the new Crank signal to the previous module.

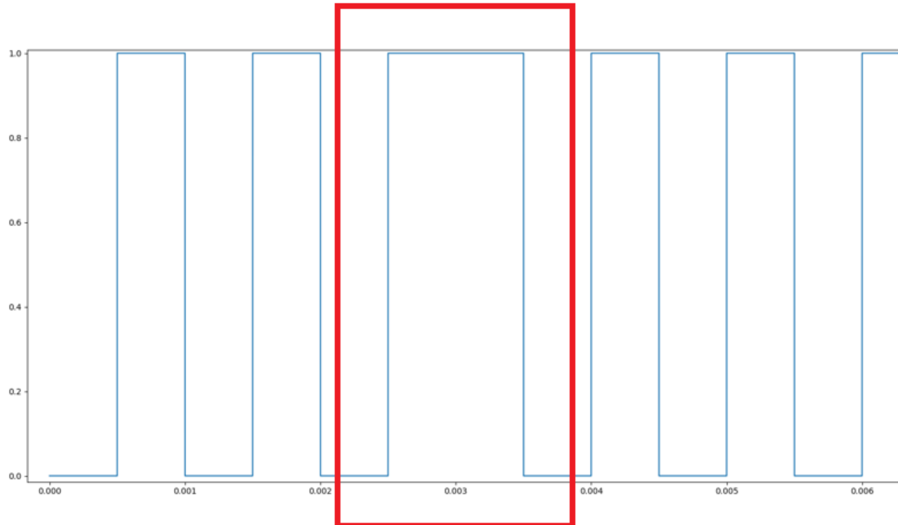

Figure 4.11: Example of Crank signal with Reference Transition Jitter

The figure 4.11 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Reference Transition Jitter = 0.5 ms; Reference Transition Jitter ID = 23; Reference Transition Jitter Cycle = 1.

4.7 Dynamic RPM

For generating a Crank Signal, a quite large set of parameters has to be considered. Up to now, the Crank generation has been performed in *Static* mode only. This means that for all the duration of the test, the RPM of the system does not change with time.
Anyway, in the real system, the angular speed of the Crankshaft can vary over time. Thus, it is important to cover this scenario.
First of all, a speed profile has to be defined: initial speed, acceleration, final speed. In order to obtain a dynamic Crank signal, two approaches have been developed: **time-based** and **angular-based**.
The time-based approach has been developed such a way it is compatible with FPGA-based validation platform. Instead, the angular-based one is more compliant with the actual system behaviour when subjected to accelerations or decelerations.

4.7.1 Time-Based Approach

Starting from basic equation of Classic Mechanics:

$$\omega(t) = \frac{d\theta}{dt},$$

$$\alpha(t) = \frac{d\omega}{dt} = \frac{d^2\theta}{dt^2},$$

$$\rightarrow \omega(t) = \int \alpha dt,$$

$$\rightarrow \theta(t) = \int \omega dt = \int \int \alpha dt^2.$$

Applying the *Uniform Accelerated Circular Motion* conditions:

$$\omega(t) = \int \alpha dt = \alpha t + \omega_0 \tag{4.4}$$

$$\theta(t) = \int \omega dt = \frac{1}{2}\alpha t^2 + \omega_0 t + \theta_0 \tag{4.5}$$

As we can see, the two equations (4.4) and (4.5) show a **time dependency**. Considering a constant angular acceleration α, the resulting RPM variation profile will be a straight line with **Slope** equal to the constant acceleration. Therefore, the scenario will appear as the one shown in the figure 4.12.

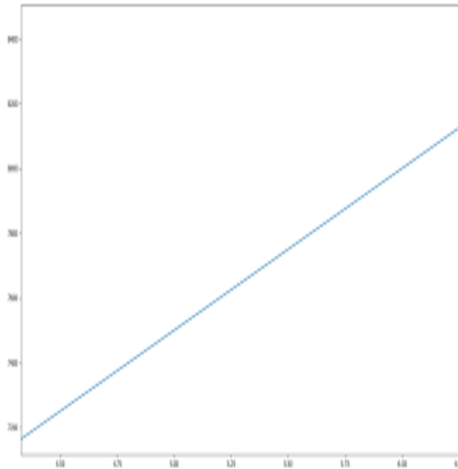

Figure 4.12: Speed variation profile with constant acceleration

In order to follow the RPM variation profile, it is possible to sample the straight line with a certain given granularity (number of samples).
Each sample is characterized by a certain duration Δ.
In this way, the sampled line will be a stair-like shaped line and where each step corresponds to a **finite variation** of the signal RPM (figure 4.13).

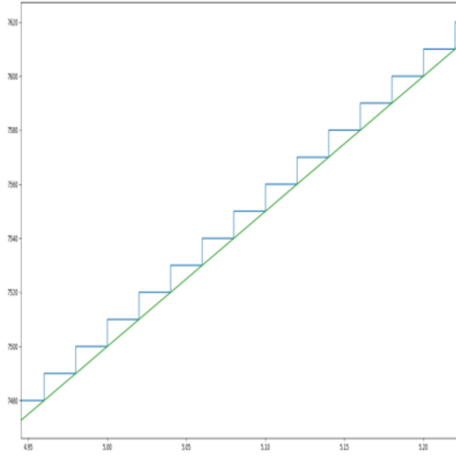

Figure 4.13: Sampled speed variation profile with constant acceleration

Furthermore, the higher the slope (higher acceleration), the bigger the number of steps needed. The problem is associated to the following formulas:

$$time_{dynamic} = \frac{(RPM_{final} - RPM_{initial})}{Slope} \tag{4.6}$$

$$\Delta = \frac{time_{dynamic}}{Sample} \tag{4.7}$$

$$increment_{RPM} = Slope \times \Delta \tag{4.8}$$

A python module has been developed for implementing this RPM dynamic approach.
The module has 3 main functions: *DynamicRPM*, *TimeVectors*, *SquareWave*.
First, the General and Specific requirements are received by .json file.
The Specific parameters of this module are the following:

- *RPM initial*: it represents the RPM at the begin of the test;
- *RPM final*: it indicates the RPM to be reached;
- *Slope*: it denotes acceleration;

- *Number of Steps*: it is the number of sections within the dynamic profile;
- Δ: it denotes the time duration of each sample.

Numerical details and features of time-based RPM dynamic parameters are shown in table 4.5

Parameter Name	**Range**	**Resolution**
RPM initial	-3000, +10000	1 RPM
RPM final	-3000,+10000	1 RPM
Slope	/	1 RPM/s
Number of Steps	/	1
Δ	>=1μs, <= Tooth Period	1μs

Table 4.5: Time-Based RPM Dynamic parameters.

The main function generates a time vector and the corresponding Crank signal for the RPM_initial.
Then, the *DynamicRPM* function is called and takes as input arguments RPM initial, final, Slope, number of samples and Crank resolution. It computes the step-RPM profile and returns it to the main function.
Now, for each step on the dynamic step profile, a Crank signal and the corresponding time vectors are computed by calling the *SquareWave* function.
By iterating the process is possible to merge together new Crank signal and time vectors with old ones.

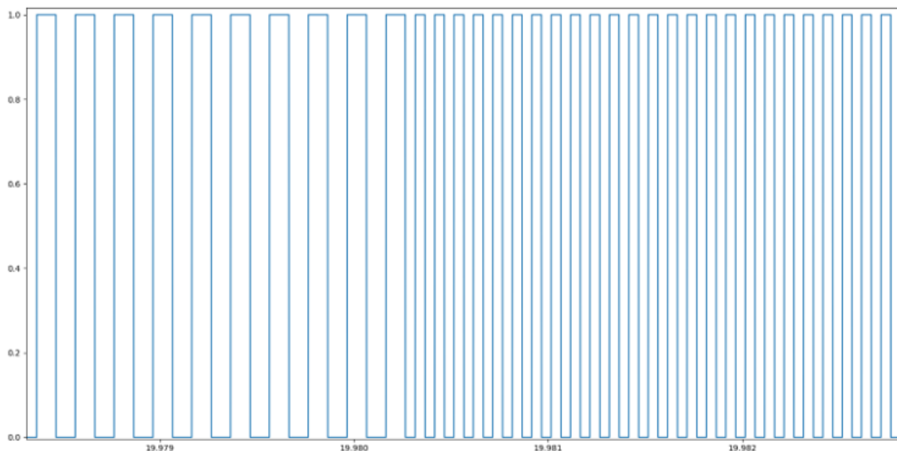

Figure 4.14: Crank signal Variation example

The figure 4.14 depicts a Crank Signal performing a RPM variation time-based. The RPM initial is set to 5000; while, the RPM final is equal to 10000. The Slope is 500 and it is sampled with 500 steps with Δ equal to 0.02s.
As it is possible to notice from the previous example 4.14, the signal dynamic transition does not mimic the natural behaviour of the system. Indeed, the linear speed profile has been sampled. This leads to **approximation**, since the lower bound for Δ is equal to the resolution of the platform (*HW frequency*).
Although, the Crank dynamic transient signals generation is based on the step-profile, by *rising* one step, the RPM increment is **finite**. So, it will suddenly change from a certain quantity to another one.
Moreover, if the increment occurs after every revolution or multiple, it does not gives problems. Otherwise, partial corrupted teeth appear.
A way for dealing with such teeth is to discard them, but this will affect the execution time (further approximation).

4.7.2 Angular-Based Approach

As mentioned before, the previous approach introduces approximations, so it is not possible to mimic the natural behaviour of the system. The time-based approach has an intrinsic limit: the littlest sample possible is equal to the HW frequency.
Even considering such a little sample, the sampled RPM dynamic will be always a step-profile (finite increment).
In order to avoid time dependency, it is possible to switch to an **angular-based domain**.
Considering equations (4.4) and (4.5) (*Uniform Accelerated* conditions applied):

$$(4.4) \rightarrow t = \frac{\omega(t) - \omega_0}{\alpha} \tag{4.9}$$

$$
\begin{aligned}
(4.5) \rightarrow \theta(t) &= \frac{1}{2}\alpha\left[\frac{\omega-\omega_0}{\alpha}\right]^2 + \omega_0\left(\frac{\omega-\omega_0}{\alpha}\right) + \theta_0 = \\
&= \frac{1}{2}\alpha\frac{(\omega-\omega_0)^2}{\alpha^2} + \frac{\omega_0}{\alpha}(\omega-\omega_0) + \theta_0 = \\
&= \frac{1}{2}\frac{(\omega-\omega_0)^2}{\alpha} + \frac{\omega_0}{\alpha}(\omega-\omega_0) + \theta_0 = \\
&= (\omega-\omega_0)\left[\frac{(\omega-\omega_0)}{2\alpha} + \frac{\omega_0}{\alpha}\right] + \theta_0 = \\
&= (\omega-\omega_0)\left[\frac{\omega-\omega_0+2\omega_0}{2\alpha}\right] + \theta_0 = \\
&= \frac{(\omega-\omega_0)}{2\alpha} + \theta_0 = \\
&= \frac{(\omega^2-\omega_0^2)}{2\alpha} + \theta_0, \forall\omega.
\end{aligned}
\tag{4.10}
$$

As we can see from equation (4.10), it is now possible to determine the angle θ for each value of the angular speed ω (parabolic relationship), *independently* by time. By making the inverse function of the equation (4.10), the **ω-profile speed** is found:

$$
\omega(\theta) = \sqrt{2\alpha\theta + \omega_0^2 - 2\alpha\theta_0}. \tag{4.11}
$$

Under this condition, the angular speed profile has a horizontal parabolic behaviour. So, it is important to evaluate its domain. It results:

$$
\theta \geq \frac{2\alpha\theta_0 - \theta_0^2}{2\alpha} \tag{4.12}
$$

Therefore, the profile speed function belongs to real class ($\subset \Re$), if and only if θ respects the constraint (4.12). Otherwise, it will be a subset of complex functions class ($\subset \mathbb{C}$).
When θ_0 does not respect the (4.12) condition ($\rightarrow \omega \subset \mathbb{C}$), the domain becomes complex $\mathbb{C}$. Hence, the resulting theta will have an **oscillatory** behavior.
In the proposed model, this case is not covered, since no experimental measurements on frictions or dumping factors have been provided. The way for managing this scenario is simply to keep the previous speed rate.
Considering that one tooth of the Crank signal is equal to 6° (1 Tooth $\equiv 6°{=}\theta_{tooth}$), it is possible to evaluate the angular velocity after that the crankshaft has swept 6°:

$$
\omega(\theta_{tooth}) = \sqrt{2\alpha\theta_{tooth} + \omega_0^2 - 2\alpha\theta_0}. \tag{4.13}
$$

Therefore, the period of the tooth results:

$$
period_{tooth} = \frac{\theta_{tooth}}{\omega(\theta_{tooth})}. \tag{4.14}
$$

Taking into account a certain given *DutyCycle*, the resulting active part of the tooth is:

$$active_{tooth} = period_{tooth} \times DutyCycle. \tag{4.15}$$

Going ahead, the corresponding tooth will be:

$$Crank(t) = \begin{cases} 1, & \text{if } active_{tooth} \leq t \leq period_{tooth} \\ 0, & \text{if } 0 \leq t \leq active_{period} \end{cases} \tag{4.16}$$

By computing the angle swept by the crankshaft during the speed profile section (θ_{swept}), it is possible to compute the number of iterations needed to cover all the velocities within:

$$\theta_{swept} = \frac{(\omega_1^2 - \omega_0^2)}{2 \times \alpha} + (\theta_0), \tag{4.17}$$

$$number_{iteration} = \frac{\theta_{swept} - \theta_0}{\theta_{tooth}}. \tag{4.18}$$

Furthermore, it is possible to evaluate the time spent for performing it as:

$$time_{swept} = \frac{(\omega_1 - \omega_0)}{\alpha}. \tag{4.19}$$

Since the procedure has been applied to one speed transition only, in case of multiple ones (**entire speed profile**), it will be repeated for each section composing the profile 4.15.

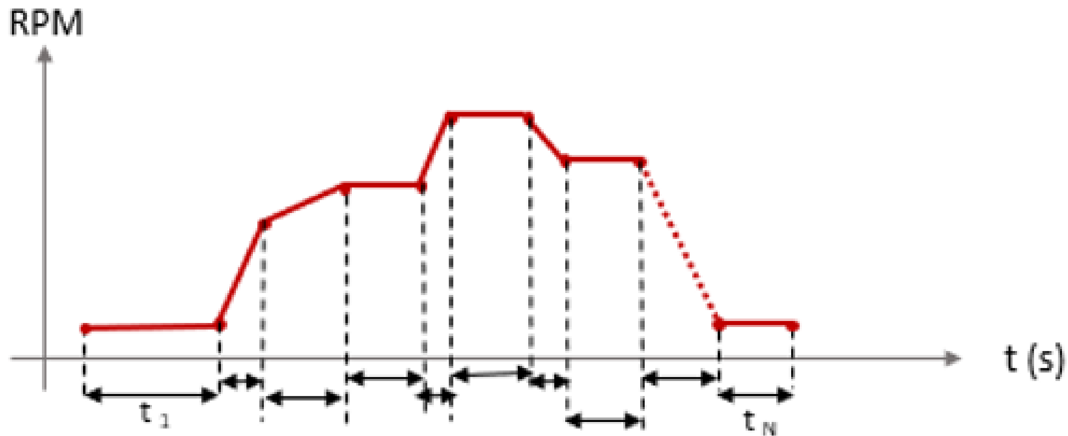

Figure 4.15: Full Speed Profile.

Proposed Code

The described approach has been implemented by developing a module (*Dynamic RPM tooth based*) and integrating in the main platform.
First, the requirement database has been updated with the parameters related to this approach.
In particular, the modules takes advantages of General and Specific parameters.
The used Specific parameters are the following:

- *Sections Number*: it represents the number of sections composing the RPM profile ;
- *RPM Dynamic*: array parameter that shows the RPM values at the edges of each section;
- *Slope Time*: array parameter that denotes the time taken for going through a single section;

Numerical details and features of angular-based RPM dynamic parameters are shown in table 4.6.

Parameter Name	**Range**	**Resolution**
Sections Number	1,20	1
RPM Dynamic	-3000,+10000	1 RPM
Slope Time	1 ms, 5000 ms	1 μs

Table 4.6: Angle-Based RPM Dynamic parameters.

```
"General_parameters": {
    "Condition": "Binary",
    "Revolution_NO": 1,
    "Sleep": 0.5,
    "HW_Frequency": 100,
    "TEST_RPM_state": "Static",
    "TEST_RPM_parameters": {
        "Sections_number":5,
        "RPM_dynamic":[3000, 10000, 5000, 7000, 4000, 3000],
        "Slope_time": [0.01, 0.01, 0.01,0.01, 0.01]
    },
```

Figure 4.16: Angular-based parameters in the .json file.

The platform communicates and loads the requirements from the JSON file (figure 4.16). In the *Test.py* module is checked a flag (*TestRPMState*), and if it is

verified ("Dynamic"), the RPM dynamic module is launched.
All the needed parameters are given to this module as arguments.
The module creates (or open) a *.txt* file inside the Test results folder, called *"Crank-Dynamic.txt"*, within which are the rising and falling edge of the Crank signal will be stored.
Then, information about θ_{tooth} and θ_0 are set:

$$\theta_{tooth} = 6^{\circ}, \theta_0 = Crank_{initialposition} \times \theta_{tooth}. \tag{4.20}$$

Then, a for-cycle is started and repeated Section Number time.
The RPM to degrees per seconds conversion is performed: $1^{\circ}/sec = 6RPM$. In particular, taken and converted the first two consecutive elements of *RPM Dynamic* array, they are assigned to ω_0 and ω_1, respectively.
Considering, the first cell of the *Slope Time* array, the **acceleration** is computed as the difference between the two speeds divided by the time spent for changing speed.
According to the acceleration values, two cases are managed: whether it is different from 0 (accelerating or decelerating) and whether it is equal to 0 (constant velocity).
These two different case leads to different ways for evaluating the angle swept in the section.
If the acceleration is not 0, then the previously used formulas are valid. Hence, the angle is evaluated by using the equation 4.17.
Otherwise, the *Uniform Circular Motion* laws have to be used:

$$\theta_{swept} = \omega_0 \times time_{section} + \theta_0. \tag{4.21}$$

Now, a **theta vect** is created, which goes from θ_0 to θ_{swept} with step of θ_{tooth}, and the *DutyCycle* updated.
When it is the first time that the module is called, the function responsible for writing inside the .txt file is run (*Generation*) and the first recording of the signal is written down: **"0, 0"**, in the format value/time.
At this point, the for-cycle responsible for running thought the same section is performed and the procedure previously described is done. So, from equation 4.13 to 4.16.
Within the Crank generation part, the rising edges are detected and the *Generation* function is called from printing Values and time inside the output file.
The Falling Edges are not detected, but simply printed at the time corresponding to each tooth end.
Once all the cycles have expired (all sections have been covered), the modules returns the new Crank signal and delivers the .txt file.

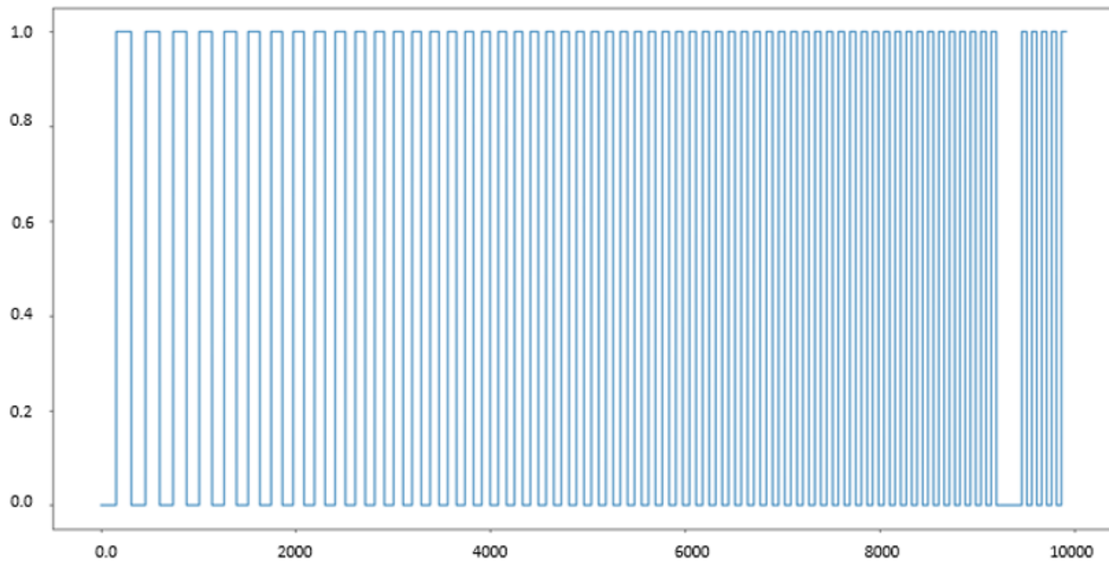

Figure 4.17: Example 1.

In the example 4.17, the crankshaft is subject to an acceleration for 0.01 second that forces the speed from $RPM_{initial}$=3000 to RPM_{final}=10000.

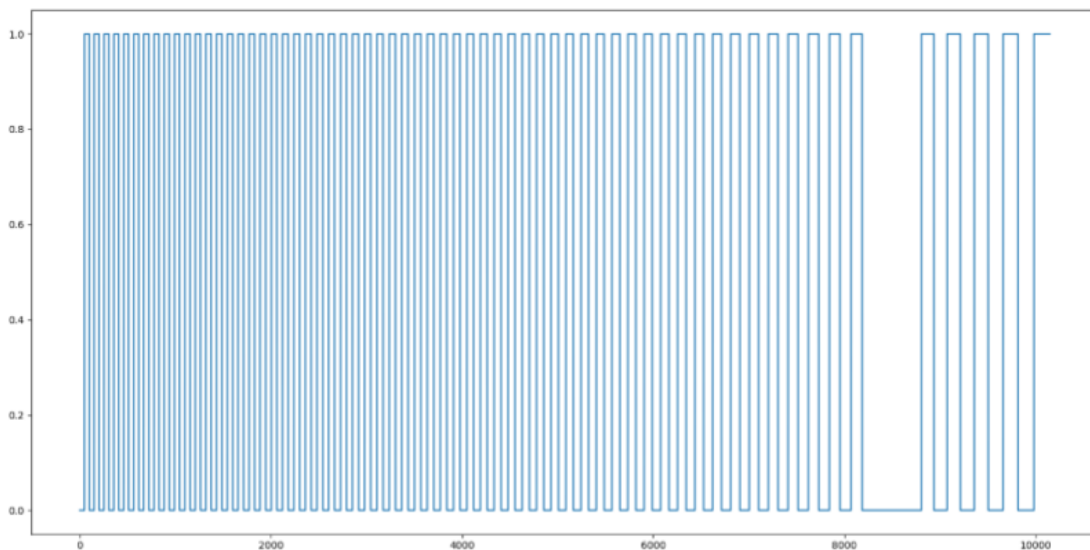

Figure 4.18: Example 2.

In the example 4.18, the crankshaft is subject to a deceleration for 0.01 second that forces the speed from $RPM_{initial}$=10000 to RPM_{final}=3000.
As it is possible to appreciate from the examples, the Angular-based approach offers

a model which behaviour is much more similar to the Actual System than the Time-based one.
Indeed, the speed variation is not discrete anymore, but continuous. This means that the variation happens every infinitesimal amount of space ($d\theta$).

Chapter 5

Results

The *Mathematical Models* are considered as Executable Requirements or Golden Outputs. Their main goal is to provide a reference during testing and validation phase. Hence, the output coming from a certain target environment is compared with such a Golden Output. Then, a report showing whether the test is passed or failed is produced.
In order to achieve this purpose, complex and more complete signal stimuli models have been implemented, as shown in previous chapter. Indeed, considering the main reference only, it is possible to generate the **Crank signal** in static or dynamic configurations and under several conditions (Missing, Spurious, Reference Transition Jitter). It is important to notice that the different conditions are available in static mode only.
In the following sections, graphical results of the Executable Requirements are depicted.

5.1 Normal Crank Signal

The Normal Crank signal is taken as a base model for all the further extensions. Considering a phonic wheel made up with 60 teeth, the resulting Crank signal consists in 60 pulses. Each one is equally spaced from the other, since the period (frequency) does not change with respect to time (Static).
Each tooth of signal corresponds to 6° and is characterized by two reference transitions: rising and falling.The last one is considered as timing reference.
Furthermore, the signal presents two missing teeth, in particular the 59th and 60th. They are used for synchronization purposes and for graphically distinguishing between two complete rotation.
When the crankshaft has swept 720°, then a complete revolution has been performed. This means that the corresponding Crank signal is shaped by 120 teeth.
The Independent Python Platform foresees the introduction of a specific parameter called $"CRANK_{InitialPosition}"$. This parameter is needed for setting the starting

initial position of the crankshaft.

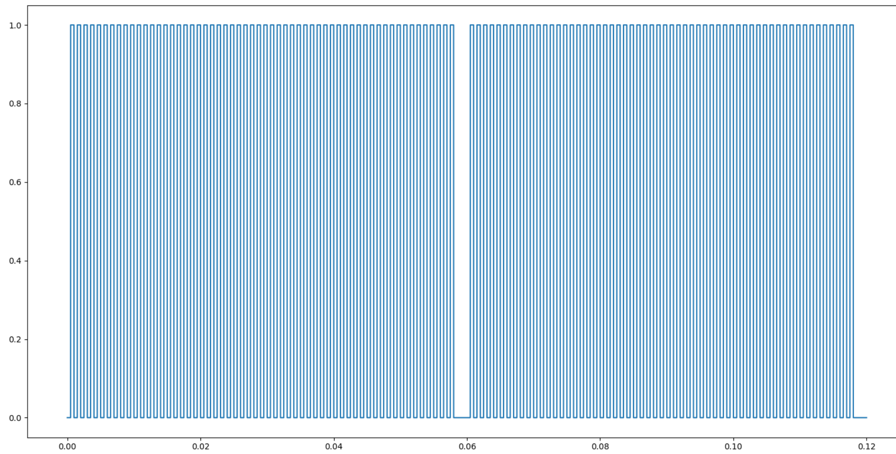

Figure 5.1: Normal Crank Signal: Example 1.

In the figure 5.1, it has been generated a Crank signal in Binary and Static mode, with RPM equal to 1000, Crank Initial position set to 0 and Duty Cycle 0.5. The figure represents 1 revolution cycle only. It is possible to appreciate the gap between the two successive half revolution (360°) and the one after the entire revolution, corresponding to the 119th and the 120th teeth.

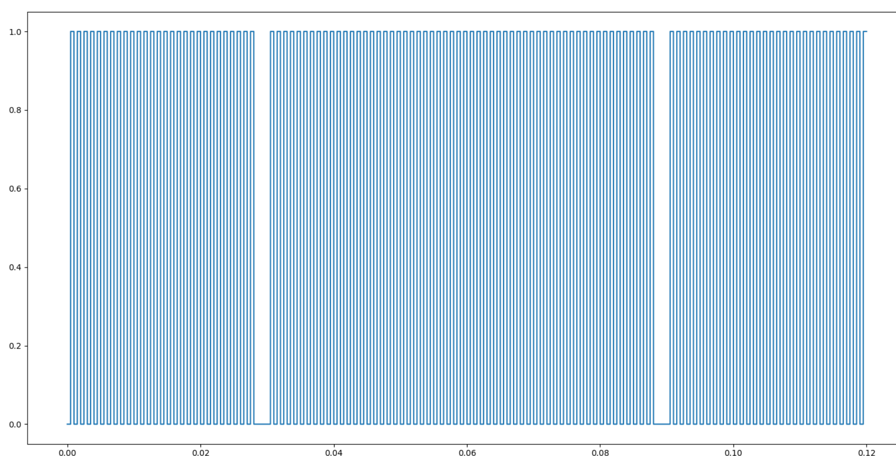

Figure 5.2: Normal Crank Signal: Example 2.

In the figure 5.2, it has been generated a Crank signal in Binary and Static mode, with RPM equal to 1000, Crank Initial position set to 30 and Duty Cycle 0.5. The figure represents 1 revolution cycle only. It is possible to appreciate that the gap occurs earlier than the previous scenario 5.1. Indeed, the crankshaft initial position is set to 30 (180°). Then, since a single revolution is performed (720°), the remaining part consists in 540°.
In the following sections, the resulting Crank signal in different conditions are shown. In particular in Missing, Spurious and Reference Transition Jitter conditions. Furthermore, the last sections will show some results related to Crank signal in Dynamic mode. The resulting Crank signals have been obtained by running the updated Independent Python Platform.

5.2 Missing Tooth Condition

The *Missing Tooth Condition* affects the Crank signal by deleting one or more teeth inside a specific sample. It is used for modelling, for example, whether the crankshaft sensor has output set to 0.
The following scenarios depict three examples of Crank signals in *Missing Tooth* condition generated by using the Independent Python Platform developed modules.

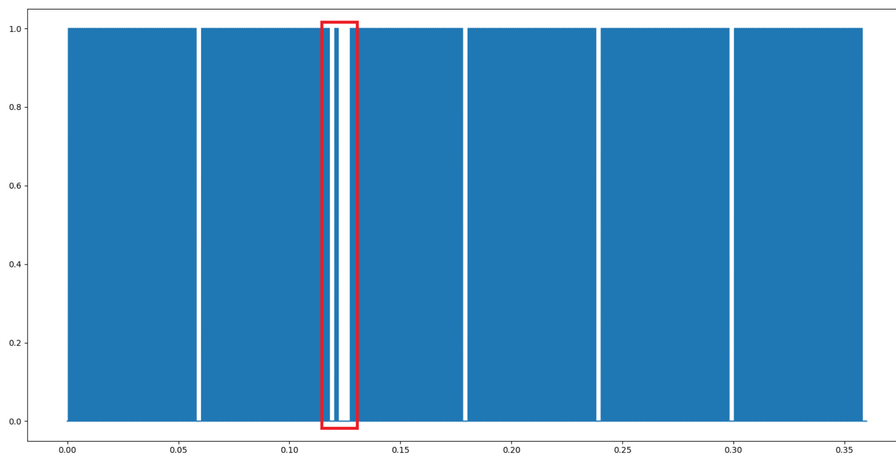

Figure 5.3: Missing Tooth Condition: Example 1.

The figure 5.3 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 0; Missing Tooth ID = 3; Missing Tooth Cycle = 2; Missing Tooth Repetition = 5. As it is possible to see, the 5 teeth of the Crank signal are missed within the 2 sample,

starting from the 3rd reference transition. The results completely matches with the set requirements.

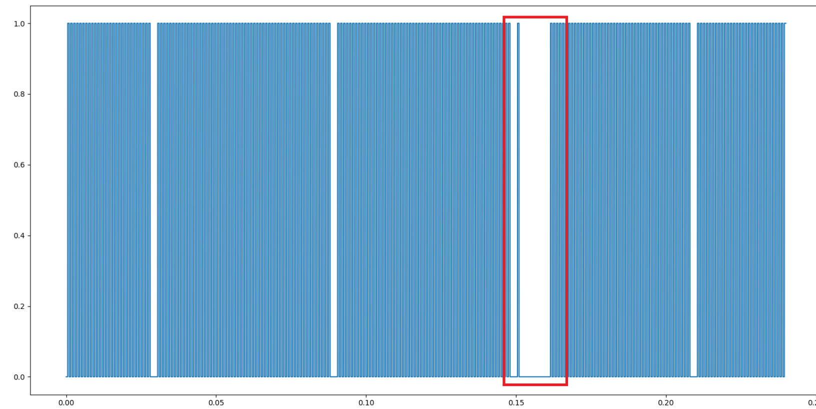

Figure 5.4: Missing Tooth Condition: Example 2.

The figure 5.4 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 2; Crank Initial Position = 30; Missing Tooth ID = 62; Missing Tooth Cycle = 2; Missing Tooth Repetition = 10.
In this second example, the resulting Crank signal has 10 missing teeth. The condition is applied starting from the 62nd tooth of the 2nd revolution. As expected, the result properly meets with the specifications.

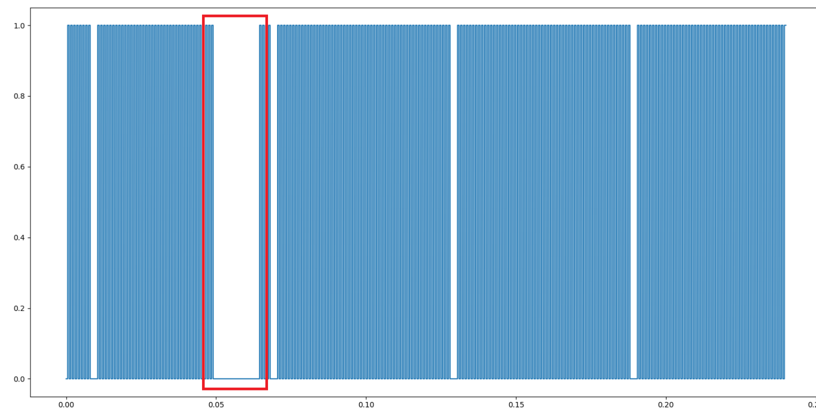

Figure 5.5: Missing Tooth Condition: Example 3.

The figure 5.5 shows a Crank signal generated with the following parameters:

RPM static = 1000; revolution number = 2; Crank Initial Position = 50; Missing Tooth ID = 100; Missing Tooth Cycle = 1; Missing Tooth Repetition = 15.
The third scenario foresees a Crank signal affected by missing tooth condition. In particular, starting from the 100th of the 1st cycle, the missing teeth are 15. As for the previous examples, also in this case, the requirements are totally met.

5.3 Spurious Tooth Condition

The *Spurious Tooth* is one of the condition that can affect the Crank signal. It foresees the presence of a certain number of extra pulses in between two contiguous falling and rising reference transitions. This type of condition can be occurred very often in noisy environment, such as a vehicle.
The following scenarios depict three examples of Crank signals in *Spurious Tooth* condition generated by using the Independent Python Platform developed modules. It is important to remember that all the consecutive extra pulses affecting the same sample are generated considering the same period and delay.

Figure 5.6: Spurious Tooth Condition: Example 1.

The figure 5.6 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Spurious Tooth ID = 2; Spurious Tooth Cycle = 2; Spurious Tooth Delay = 100μs; Spurious Tooth Period = 100μs; Spurious Tooth Repetition = 1. As it is possible to appreciate by the figure 5.6, the extra pulse starts after the 2nd tooth of the 2nd revolution and it is repeated only once. The model fits properly the specifications.

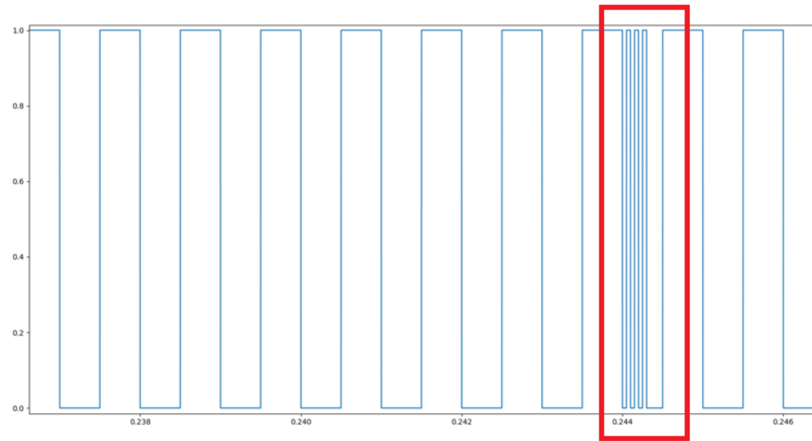

Figure 5.7: Spurious Tooth Condition: Example 2.

The figure 5.7 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Spurious Tooth ID = 24; Spurious Tooth Cycle = 3; Spurious Tooth Delay = 50μs; Spurious Tooth Period = 50μs; Spurious Tooth Repetition = 3. In this second example, the spurious tooth condition is applied after the 24th reference transition of the 3rd sample. It is repeated 3 times. Also in this case, the model respects the requirements.

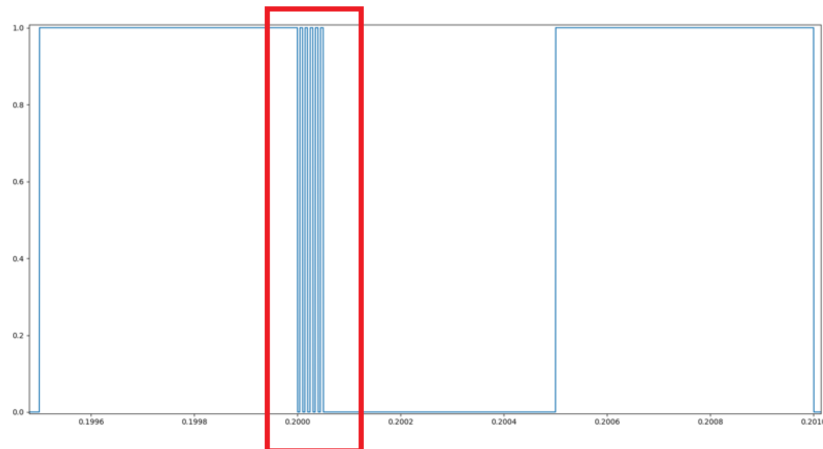

Figure 5.8: Spurious Tooth Condition: Example 3.

The figure 5.6 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Spurious Tooth ID = 100; Spurious Tooth Cycle = 2; Spurious Tooth Delay = 5μs; Spurious

Tooth Period = $5\mu s$; Spurious Tooth Repetition = 5. As it is possible to observe by the figure 5.8, the event occurs starting from the 100th tooth of the 2nd cycle. It is possible to count 5 extra pulses. The model is compliant with the expected one.

5.4 Reference Transition Jitter Condition

The *Reference Transition Jitter* condition affects one tooth of the Crank signal. It produces an extra period of the reference transition. Basically, the falling edge of the tooth is shifted. Due to this, the successive teeth are subject to a time shifting. It is important to model such a condition, since it affects very frequently the Crank signal. For example, it can be a sign of a sensor misbehavior or misplacement.

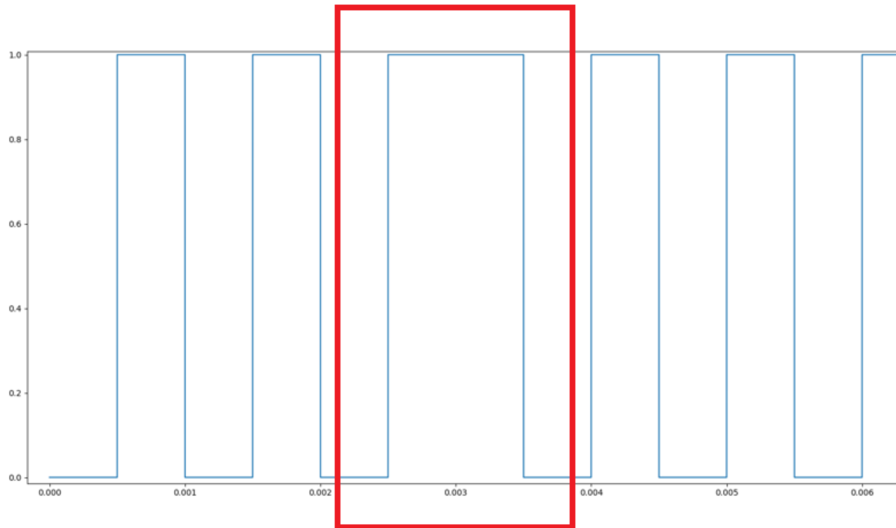

Figure 5.9: Reference Transition Jitter Condition: Example 1.

The figure 5.9 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Reference Transition Jitter = 0.5 ms; Reference Transition Jitter ID = 23; Reference Transition Jitter Cycle = 1. In this first scenario, the Reference Transition Jitter is applied to the 23rd of the 1st revolution. Furthermore, it is possible to appreciate that the affected tooth is enlarged of 0.5 ms with respect to others. In addition, the next teeth are shifted. The model is compliant with the set specifications.

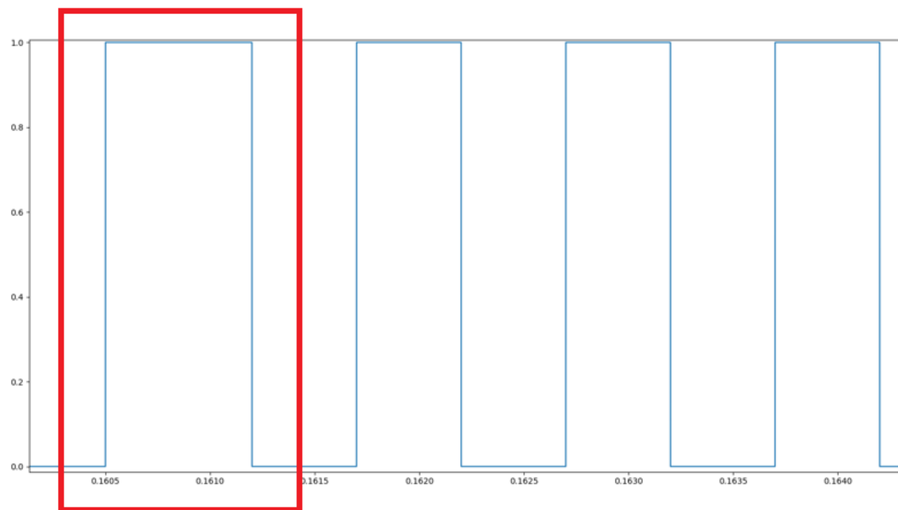

Figure 5.10: Reference Transition Jitter Condition: Example 2.

The figure 5.10 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Reference Transition Jitter = 0.2 ms; Reference Transition Jitter ID = 61; Reference Transition Jitter Cycle = 2. Considering this second set-up, the Reference Transition Jitter affects the 61th tooth inside the 2nd sample. In this case, the falling edge is shifted of 0.2 ms and all the successive teeth are also translated of such quantity. It is possible to appreciate that the graph is compliant with the requirements.

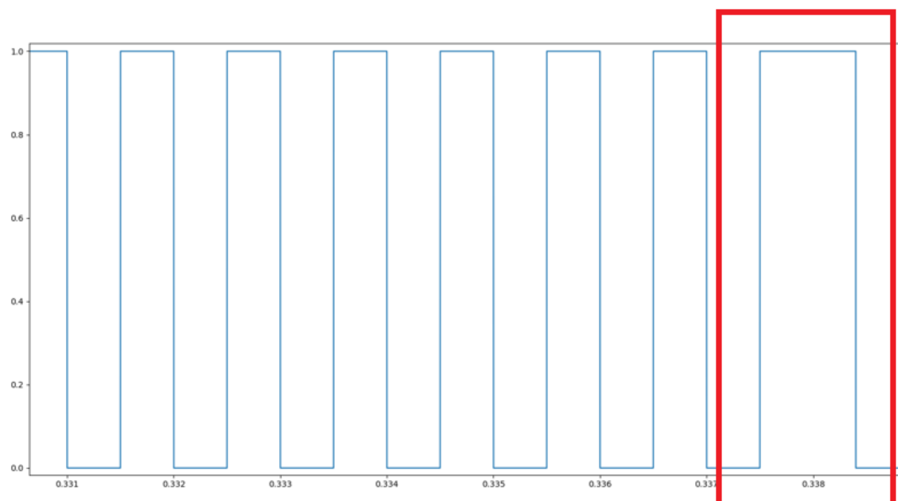

Figure 5.11: Reference Transition Jitter Condition: Example 3.

The figure 5.11 shows a Crank signal generated with the following parameters: RPM static = 1000; revolution number = 3; Crank Initial Position = 20; Reference Transition Jitter = 0.4 ms; Reference Transition Jitter ID = 118; Reference Transition Jitter Cycle = 3. In this last scenario, the Reference Transition Jitter affects the 118th tooth of the 3rd cycle. The tooth is enlarged by 0.4 ms and all the successive teeth are shifted. It is possible to see that the generated signal respects the given constraints.

5.5 Dynamic RPM in Time Domain

The Dynamic RPM management based on time foresees the sampling of a single speed profile. Indeed, considering one section only of the entire speed dynamic, it is divided into a certain number of samples.
Each step is defined by an RPM increment and by a certain duration, both fixed for every sample.
In this way, when the step expires, an instant discrete increment of RPM occurs, hence a variation of the tooth period. This lead to several mismatches with respect to the actual system. The main one is related to the appearance of partial teeth when switching from one step to the next one. For this reason, it has been needed to pass to an angle-domain.

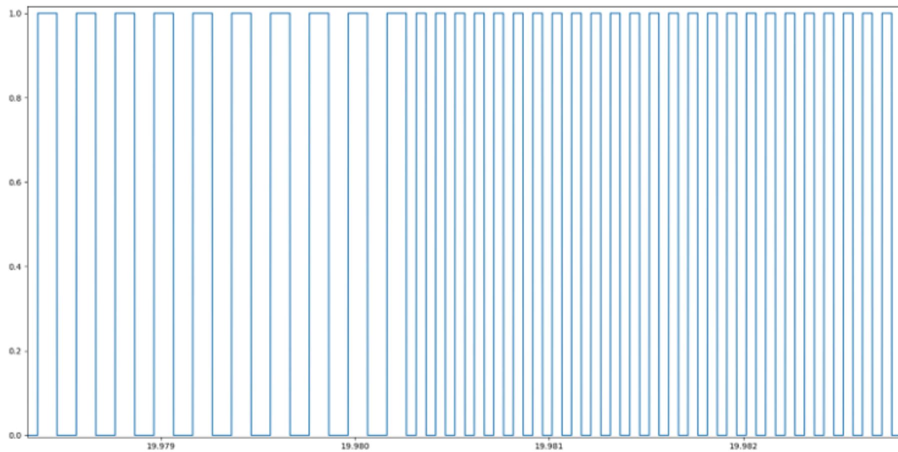

Figure 5.12: Crank signal Variation example

The figure 5.12 depicts a Crank Signal performing a RPM variation time-based. The RPM initial is set to 5000; while, the RPM final is equal to 10000. The Slope is 500 and it is sampled with 500 steps with Δ equal to 0.02s. It is possible to see

that the tooth period varies of a well-fixed quantity instantaneously. This does not meet the system behavior properly, as expected.

5.6 Dynamic RPM in Angle Domain

According to such an approach, the RPM management depends on angles. Basically, it foresees that the RPM increment or decrease is performed tooth by tooth. Hence, once 6° are swept, the corresponding ω is evaluated and set. Such method is applied to a single section of the entire RPM profile and repeated as long as it is not totally covered.

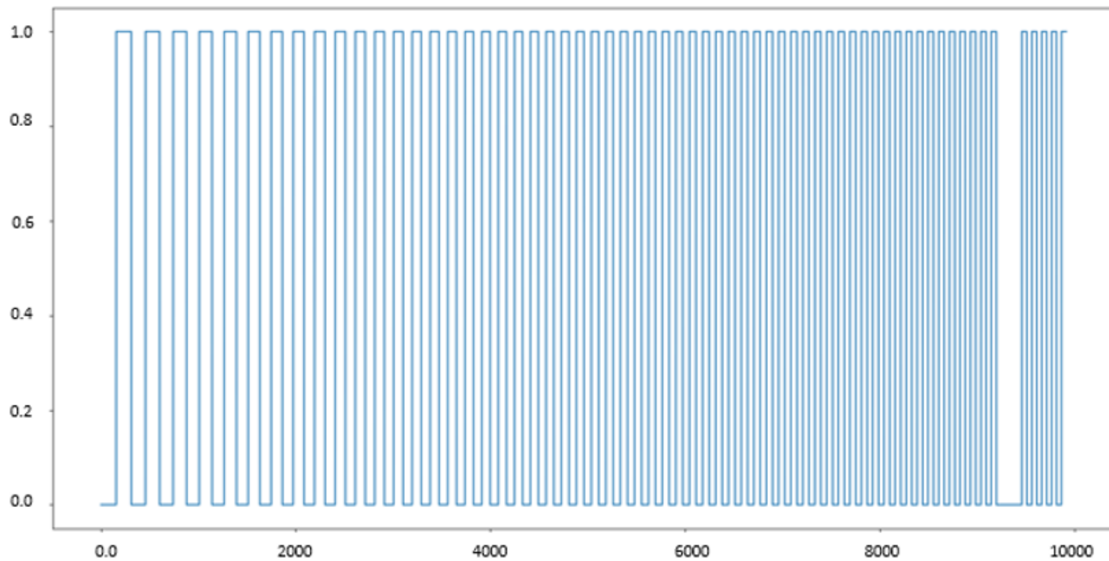

Figure 5.13: Dynamic RPM Angle-Based: Example 1.

In the example 5.13, the crankshaft is subject to an acceleration for 0.01 second that forces the speed from $RPM_{initial}$=3000 to RPM_{final}=10000.

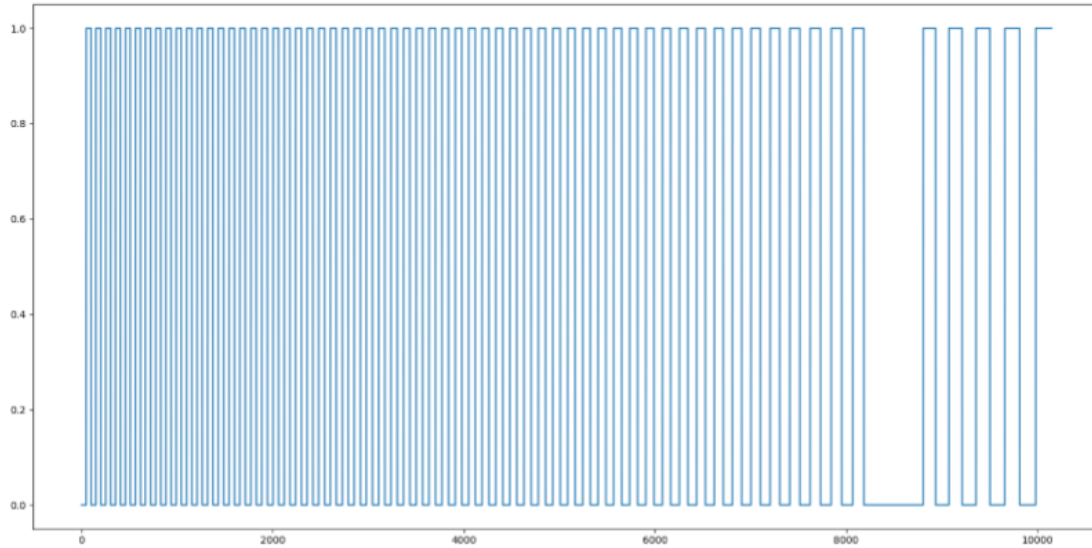

Figure 5.14: Dynamic RPM Angle-Based: Example 2.

In the example 5.14, the crankshaft is subject to a deceleration for 0.01 second that forces the speed from $RPM_{initial}$=10000 to RPM_{final}=3000.
As it is possible to appreciate from the previous results, the Angular-based approach offers a model which behaviour is much more similar to the actual system than the Time-based one.
Indeed, the speed variation is not discrete anymore, but continuous. This means that the variation happens every infinitesimal amount of space ($d\theta$).

Chapter 6

Conclusion

The continuous complexity increasing of the electronic systems in automotive field have led companies to face several challenges. The main one is related to develop and test an ECU system as fast as possible. Indeed, the less the time spent in designing and validating in a proper and systematic way, the less the amount of money spent. For these reasons carmakers have put a lot of effort in finding new strategies and techniques in order to achieve this goal.
Avoiding errors and reducing workloads for designers and test engineers is becoming more and more crucial. In this sense, the proposed virtual platform is projected.
The basic idea is to provide a unified environment for test engineers to control the executions of tests across different validation platforms and heterogeneous tools, such as HiL or Virtual Environment. Therefore, a middleware has to be implemented for communicating and interacting with different environments and providing a set of APIs to test engineers to manage and control the test features. In this way, the user will be able to launch tests considering different configurations, varying the test procedures and collect test results. Furthermore, once the test commands have been deployed and performed by the target, the framework has to be able to collect the results and compare them with Reference Models. In this way, it is possible to obtain a totally automatic test management. The considered Unit Under Test is the ECU responsible for engine control.
The Master Thesis work has been focused of the generation of the Mathematical Models to verify the feasibility of such approach. In particular, the main reference signal has been considered, known as the Crank signal.
The Crank signal is a square wave signal used together with the Cam one for synchronizing fuel injection within engine pistons. In order to cover the several scenarios possible to meet during the ECU life cycle, different conditions (Normal, Missing, Spurious, Reference Transition Jitter) and mode (Static, Dynamic) have been modelled.
The models have been implemented by using Python3 and generated according to real Unit Under Test profile and configurations. Hence, the obtained signals can be

used as actual reference for testing and validating the ECU.
As shown in the previous chapters, the Crank signal models are feasible and compliant with the expected behavior. Hence, the proposed models will be used in the Virtual Platform for generating the Test Commands and for analyzing the results coming from the environment chosen for the test.

6.1 Future Works

The future works will foresee several phases and levels.
First, the models can be improved in terms of accuracy and test cases. Indeed, the dynamic mode has to be extended with the Missing, Spurious and Reference Transition Jitter conditions. Then, different and more efficient approaches for the several conditions can be probed. In particular, a unified way for generating each signal in dynamic or static mode can be implemented.
Second, the actual implementation of the Automatic Test Pattern Generation shall be faced. The ATPG under exam will be able to assign to each parameter a certain value. Each parameter is expressed as a leaf of the variable tree-graph and defined by several quantities (range, values, Boolean, and so on). It is important to underline that the parameters are related both to Input Stimuli and Software Configuration (Driver).
In order to be able to select a certain value for a parameter, a well-defined policy is needed. This policy can be random or not. Anyhow, the ATPG will be able to cover all the use cases of interest (Test Coverage). The test coverage means producing a matrix which tells which are the generated tests (covered branches) and if they are passed or failed.
Once the test commands have been produced, the Python Platform will be able to launch them over the mathematical models and the target under exam (HiL). Then, a comparison analysis will be performed and the test coverage outlined.
Furthermore, since stress test analysis are needed, the ATPG shall generate tests considering out-range values for enlarging the test coverage and for identifying which are the actual working conditions of the System Under Test and the stressed conditions.

Bibliography

[1] E. Armengaud, A. Steininger, and M. Horauer. "Towards a Systematic Test for Embedded Automotive Communication Systems". In: *IEEE Transactions on Industrial Informatics* (2008).

[2] S. Azimi, L. Sterpone, and A. Moramarco. "Reliability Evaluation of Heterogeneous Systems-on-Chip for Automotive ECUs". In: (2016).

[3] M Barr. *Embedded Systems Glossary.* URL: https://barrgroup.com/Embedded-Systems/Glossary (visited on 02/09/2019).

[4] A. Berger. *Embedded Systems Design: An Introduction to Processes, Tools and Techniques*. CMP Books, 2001. ISBN: 1-57-820073-3.

[5] D. Boyang and L. Sterpone. "An FPGA-based Testing Platform for the Validation of Automotive Powertrain ECU". In: *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (2016).

[6] *Crankshaft position sensor: how it works, symptoms, problems, testing*. URL: https://www.samarins.com/glossary/crank_sensor.html (visited on 03/05/2019).

[7] *Error Code P0335 - Crankshaft Position Sensor A Circuit Malfunction*. URL: https://autoservicecosts.com/obd2-codes/p0335/ (visited on 03/05/2019).

[8] Q. Lui et al. "Modeling and Control of the Fuel Injection System for Rail Pressure Regulation in GDI Engine". In: *IEEE/ASME Transactions on Mechatronics* (2014).

[9] International Standardization Organization. *ISO 26262-1:2011 "Road vehicles - Functional Safety"*. 2011. URL: https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en.

[10] A. Silbershatz, P. Galvin, and G. Gagne. *Operating Systems*. Wiley, 2009. ISBN: 978-0-470-12872-5.

[11] Synopsys. "Virtual Prototypes: When, Where And How To Use Them". In: (2013).