

# Application of Rule-Based Expert Systems and Dynamic-Link Libraries to Enhance Hardware-in-The-Loop Simulation Results

Pedro Miguel Ortega-Cabezas<sup>1</sup>, Antonio Colmenar-Santos<sup>1\*</sup>, David Borge-Diez<sup>2</sup>, Jorge Juan Blanes-Peiró<sup>2</sup>

<sup>1</sup> Department of Electric, Electronic and Control Engineering, UNED, Juan del Rosal, 12, Ciudad Universitaria, 28040 Madrid, Spain.

<sup>2</sup> Department of Electrical and Control Engineering Universidad de León, Campus de Vegazana s/n, Escuela de Ingenierías, 24071 León, Spain.

\* Corresponding author. Tel: +34-913-987-788; e-mail: pedromiguel.ortegacabezas@actia.fr

Manuscript submitted February 24, 2019; accepted April 12, 2019.

doi: 10.17706/jsw.14.6. 265-292

---

**Abstract:** New and innovative techniques to validate software are needed to reduce cost and increase software quality.

This research focuses on the validation of engine electronic control unit software by using expert systems (EXs) and dynamic link libraries (dlls) with the aim of checking if this technique performs better than traditional ones.

To do this, a test-case database was built and run by using hardware-in-the-loop (HIL) simulations to validate a series of software modules (SMs) by using these techniques: the tester-in-the-loop, automation by using a Python script, the model-based testing and EXs combined with dlls with the aim of assessing several factors such as: productivity gain, bug detection skills, functional coverage assessment, ease to automate test-cases among others.

Dlls and EXs improve the HIL success rate by 4.8%, 6% and 20% at least, for simple, fairly-complex, and highly-complex SMs, respectively. Between 9 and 13 more bugs were found when using the EXs and dlls compared with other techniques. Two of the bugs would have required software not initially planned as they were linked to environmental policies. The proposed technique can be applied to any types of a SM, especially in those cases in which traditional validation techniques fail.

**Keywords:** Software validation, hardware-in-the-loop simulations enhance, expert system, dynamic-link library, performance and code bugs.

---

## 1. Introduction

### 1. 1. Engine ECU Software

Electronic control units (ECUs) have become essential for the correct operation of a vehicle [1]-[2]<sup>1</sup>. Software validation plays a key role and has two fundamental goals [3]. Firstly, the software must comply with the functional specifications set by the design team. Secondly, software validation ensures the integration of all software modules (SMs) into the hardware, simultaneously checking that all the elements

<sup>1</sup> It is recommended to read Appendix A which explains clearly how the engine ECU operates.

present in the network interact properly [4], [5]. The process of software validation of an ECU implies significant costs for the companies during a project because of the means necessary to carry out this activity [6], [7]. In addition, the cost of correcting bugs, once the software is marketed, is high and it can tarnish the brand's image [8], [9]. Consequently, a balance between costs, deadlines, and quality must be reached.

Powertrain<sup>2</sup> control is a system in charge of transforming the driver's will into an operating point of the powertrain according to the performance established for the product [10]. The key element of the control system is the engine ECU composed of complex hardware and software. The engine ECU (hardware and software) must be validated to assure that engine is properly controlled, the interaction with the rest of the ECUs is rightly performed and the passengers' safety is insured. Thus, one can deduce that the software validation process is complex and needs improvements with the aim of reducing costs, increasing productivity and reliability in the automotive sector [11], [12].

This research is focused on the engine ECU software validation and shows solutions to the main difficulties associated with traditional software validation techniques by using expert systems (EXs) and dynamic-link libraries (dlls) during the hardware-in-the-loop (HIL) simulation. The technique proposed in this research performs better than traditional techniques and allows improving: ease for automating test-cases, bug detection skills, functional coverage, difficulties to detect bugs linked to SMs that do many calculations and the difficulties to validate the software automatically among others. In addition, it shows that the HIL simulation can be automated in an easier way. All these topics are analyzed in-depth in this paper.

## 1.2. Related Works

The code and functional coverage is a real concern when validating a software. Research has been conducted on this topic to enhance this parameter [13], [17]. Therefore, test-case generation is a key issue. The black-box technique has been used for a long time in the automotive sector, as discussed by Conrad [18]. Despite its widespread use, it is true that it has some weak points as discussed by Chundur, Felt, and Adenmark [19]. In their dissertation, they consider that test-cases based on the engineers' experience usually imply gaps and test-redundancies. The model-based testing technique is an option to assess the code and functional coverage rate. The generation and execution of test-cases based on models have been proposed on several occasions. For instance, Skruch and Buchala (DELPHI supplier) proposed a study based on models [20]. The tool Automation Desk (dSpace®) was used. Raffaelli et al presented research focused on functional models by using the commercial software Matelo® [21], [22].

The HIL simulation should be carried out as quickly as possible and with the highest number of test cases executed to ensure the time-frame and quality of the project [23]. Test automation is essential to ensure a high code coverage and to improve reliability [24], [26]. There are many ways for automating HIL simulation in the market [27], [28]. The automation process is mainly based on black-box techniques such as exposed by Lemp, Köhl and Plöger: *"As a rule, the tests specified by the ECU departments are first performed as black box tests on the network system (know-how on software structures is not taken)"*.

The HIL simulation implies that a specific operating point is reached by the engine ECU. This can be extremely complicated, requiring a lot of manipulations on the HIL model due to software module (SM) interactions. There are three possible ways for executing a given test-case in an HIL simulation. Firstly, executing the test-case manually, that is, a technician performs all the necessary actions in the HIL simulation to reach the desired operating point. Secondly, the *"tester-on-the-loop"* concept can be used.

<sup>2</sup> Powertrain is composed of the clutch, gearbox, conical group and propeller shaft.

Petrenko, Nguena-Timo and Ramesh, reported the main problems and solutions associated with software validation in the automotive sector [29]. Their main conclusion was focused on the methodology known as “tester-in-the-loop”, in which the test engineer leads the system to a desired operation point, considered as a crucial operation point. Once the crucial point is reached, a series of automated actions are executed to reach the goals previously established in the test-case. Finally, test-cases can be fully automated. In this case, a script controls the whole execution process.

Some types of bugs are not detected by using some techniques such as the tester-in-the-loop or black-box, as shown in this research (Fig. 1). Fig. 1, depicts the obtained result for an output for a variable of a SM when executing the software in an HIL simulation (in red) and its expected value (in blue). As one can see, the results are different. This error represents an inaccuracy when it comes to calculating the gas speed in the exhaust pipe. This error impacts the amount of urea injected to treat NOx. Because this bug does not imply the presence of a functional bug, it is impossible to detect it by using the black-box technique. The detection of this type of bugs involves the checking and detailed analysis of the software code by running additional software.

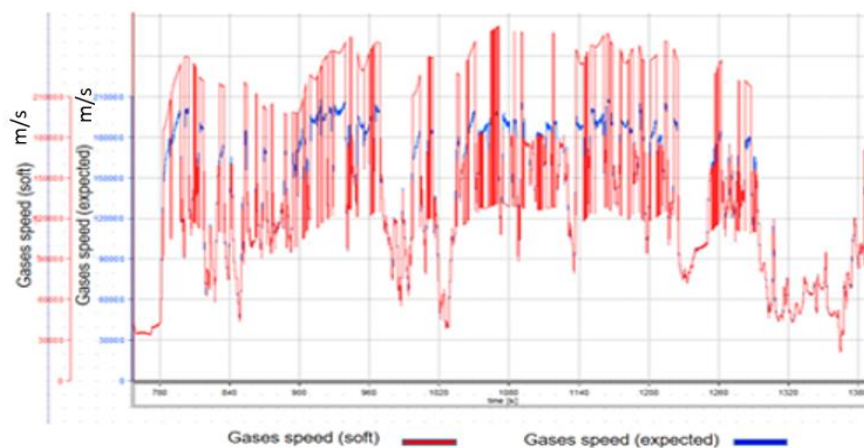


Fig. 1. Bug not detected when using traditional techniques.

The solution for validating no matter what type of software module (SM) is very far from achieving by employing a direct comparison between the HIL results and the expected outputs indicated in the test-cases. One can encounter some difficulties such as synchronization problems or difficulties to validate the software automatically, among others. Table 1 exposes the main issues.

Table 1. Potential Solutions for the Aforementioned Issues

| Consequences  | Reason  | Possible solutions  |
|---|---|---|
| Difficulties to validate the software automatically | When the values set in the test-case for the inputs are not reached due to SM interactions, then the output values set in the test-case may be no longer available. No automatic validation can be performed.   | Recalculate the output values that the software should provide for the specific input values reached after the HIL simulation. As a result, an automatic validation process can be carried out. Dlls can perform this task. |
|   | The test-engineer cannot establish the expected outputs before performing the test. In some cases, the output values are analog trends which depend on many factors (number of kilometers, number of regenerations of the diesel particulate filter, values of safety module counters, dilution oil rate, EEPROM properly initialized, etc). Consequently, the expected output can be set after |   |

|   |  |   |
|---|--|---|
|   | having performed the HIL simulation.   |   |
| Bug performance detection   | If input values are different from the ones established in the test-case, then the software performance <sup>3</sup> behavior is unknown   |   |
| Synchronization problems  | When a test-case is run, the process must compare the current state of the engine ECU and the expected outputs. It is not possible to read all variables involved in the test-case at the same time due to data acquisition software limitations combined with Python scripts. Consequently, a desynchronization problem occurs as some variables are read at t <sub>1</sub> , others at t <sub>2</sub> etc. | A data-acquisition can be done while the test-case is run. Then, when the process is ended, the data-acquisition is stopped, and the conformity of the results can be achieved comparing the HIL results with the dll results.            |
|   | The fact of having different values stored in EEPROM memories keeps the test-engineer from providing accurate screenshot and expected results.   | The EEPROM can be initialized when building the dll.  |
| Functional coverage unknown   | A functional code coverage could be established by analyzing the black-box test-cases before the HIL simulation. When reaching different values for the inputs after HIL simulations, then the use-cases <sup>4</sup> tested are different from the ones planned.  | Implementing a system that can assess whether the software performance is as expected or not. Considering the number of performance rules assessed, the functional coverage could be established. A performance EX can perform this task. |
| Difficulties to detect bugs linked to SMs that perform many calculations. | The calculations may be performed wrongly but they do not imply that the vehicle behaves in such a way that the client could detect any abnormality (Fig. 1)   | Implementing a system that can check if the software properly calculates all software outputs. Dlls can perform this task.  |

The present research proposes how to implement the possible solutions depicted in Table 1 thanks to the use of dlls for validating any types of SMs when automating a test-case through the HIL simulation, and especially all SMs that cannot be validated by employing traditional techniques. Thanks to dlls, SMs responsible for doing a great deal of internal calculations, can be validated. During the HIL simulation, it can be checked that all the calculations are properly carried out when the software and hardware are integrated. This feature allows finding bugs which cannot be found by traditional techniques. In addition, in case the desired operating point set in the test-case is not reached in an automated HIL simulation, owing to SM interactions, the dlls can determine the expected output that the software should provide. Thanks to rule-based EX, it is possible to verify whether the functional behavior of the software is correct for the outputs obtained after the HIL simulation. EXs can carry out a real-time performance validation when executing a test-case thanks to dlls.

This paper is organized as follows. Section 2 describes the method used in this research. Section 3

<sup>3</sup> It is essential to clarify the meaning of software performance. Usually, this concept is linked to the software behavior and the execution time among other factors. In this case-study, the execution time is not considered as the supplier in charge of coding the software must guarantee a CPU charge lower than 80%. When using this concept in this paper, the meaning is that the software is properly coded, but it does not behave well due to a design error (specification issue). In other words, for that specific case, the engine ECU does not control the vehicle correctly.

<sup>4</sup> In this research, the term use-case is employed in the automotive meaning way. It refers to a specific operating point that the driver makes the vehicle operate. Many bugs in the engine ECU software come from situations (operating points) not considered by the design team.

presents the results. Section 4 draws the main conclusions. The reader can find different appendices. Appendix A analyses the sensitivity of the results obtained in this research. A “threats to conclusion” validity section can be found in Appendix B.

## **2. Method**

### **2.1. Method Used**

#### **2.1.1. Description**

The engine specifications are composed of Simulink® models. Thus, the dll can be easily built considering that Matworks® has implemented different ways to build a dll from a Simulink® model[30].

The method used in this research are composed of different stages. Firstly, a series of test-cases are designed as analyzed in Section 2.1.2. Then, all test-cases are run by using the following techniques: manual execution by a technician, automation by employing Python scripts (with and without dlls), the tester-in-the-loop technique and fully automated process by using a performance EX combined with dlls. The EX compiles all rules (software requirements) related to the SM under validation. To conduct the test-cases, an HIL simulation is used. The HIL model belongs to the company subjected to this case-study and has been validated by its experts. The hypothesis to be proved by following this method is that all issues shown in table 1 can be solved thanks to this technique proposed by the authors.

Some aspects must be assured such as validity of dlls and the EX designed from the SM under validation. The process of doing measurements is standardized.

Several indicators are analyzed such as: evaluation of the success rate of the HIL simulation, main causes of failure and success for each of the methodologies when running test-cases, the functional coverage obtained, the productivity gain which may take place. The advantages and limitations of using dlls will be discussed. EXs will assess the software performance.

The dll can be implemented by following the steps indicated in many Mathwork® documentation available in their site. The only thing that the user really needs is the Simulink® model to be converted into a dll. In this research, this is not a problem as the specifications needed to code the engine ECU software, are composed of Simulink® models. The main difficulty is how to call the dll. To do this, as described in Matlab® documentation, different programming languages such as C or an m-file can be employed. In this research, C language has been chosen. It is important to describe how the HIL simulation is performed when using dlls to validate the software. Fig. 2 depicts the process when using an automation script. This description is valid for all techniques but the manual execution one (no automation process). A test-case is executed through a Python script coded by a test engineer. At this moment, the software Inca® [31], or any other software that can read the memory positions of the ECU, performs the data acquisition of all the software variables selected by the test engineer. The result of this process is to generate a data-acquisition file. During the HIL simulation the script is in charge of performing all the necessary manipulations on the driver-ECU interface of the HIL model automatically. If after a certain pre-established time, the values for the input set in the test case are not reached, the data acquisition process and the test-case execution are stopped by the Python script. Then, a data acquisition file containing all the software variables chosen by the test-engineer in the HIL simulation is obtained. A C-file is in charge of decoding the data acquisition file and sending, one by one, all the samples of the HIL simulation to the dll as exposed later. Every time a sample is sent by the C-file, the dll returns the theoretical value that the software should have delivered. Then, the Python scripts checks whether the software outputs are equal to dll outputs every time the dll returns a value. Two key topics must be reminded. Firstly, the outputs of the SM are also available in the asc-ii file. Secondly, the engine ECU software is an image of the Simulink® models of the SM under validation.

### 2.1.2. Functions used in the HIL simulation

The methodology proposed in this study has been tested in three types of functions or SMs chosen according to the number of calculations to be done as well as their complexity, number of inputs and/ outputs of the SM and the accuracy required for the output results (Table 2). They have been considered as representative for this case-study by the authors and the company subjected to this research.

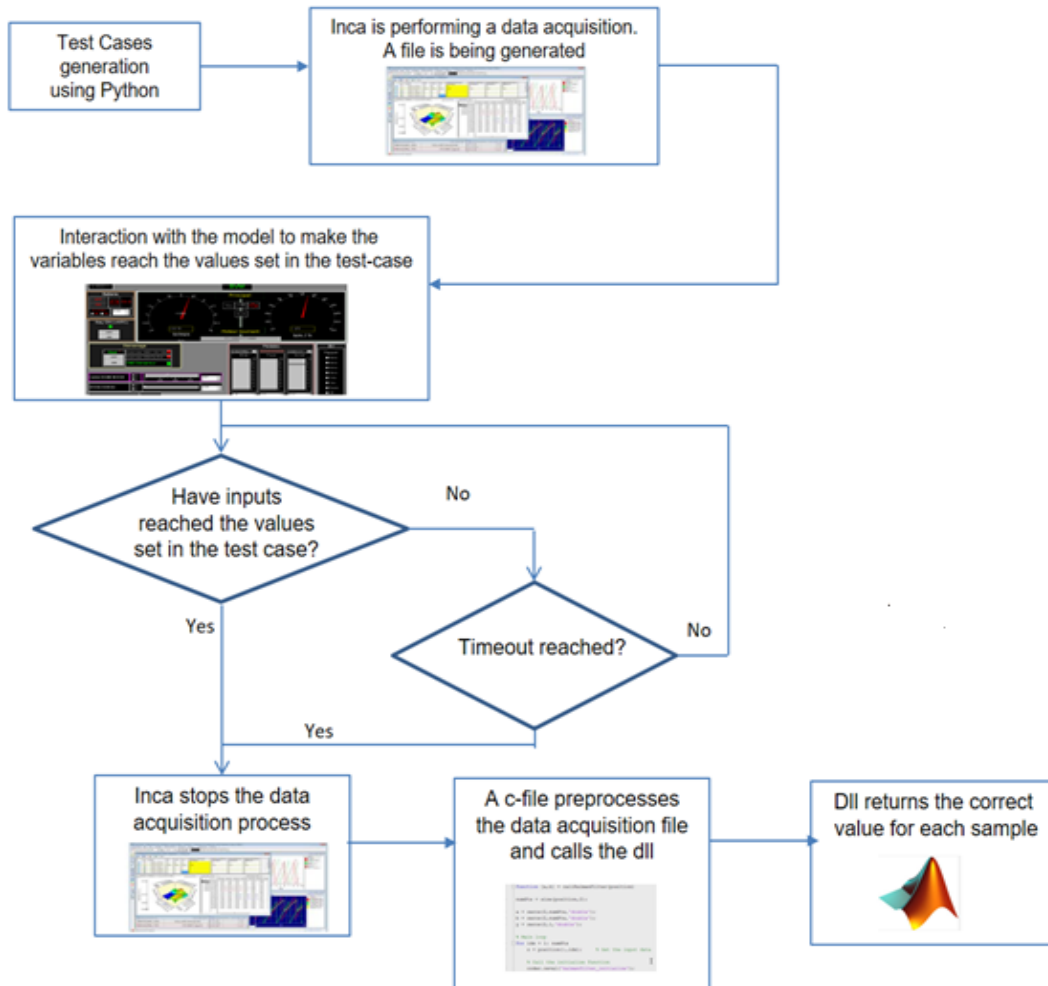


Fig. 2. Use of dlls in an HIL simulation when performing a test-case.

It is important to establish this classification because the validation requirements as well as the characteristics of the SM clearly influence the time required to carry out the validation process, as well as the additional difficulties that may arise. 5 SMs of each type were selected, based on different criteria such as test engineers' experience, the most problematic SMs in other projects, SMs that require systematic validations to ensure the vehicle safety, SMs that require frequent regression validations as well as those SMs that have never been implemented in previous projects and, in short, they are a novelty (see Table 2).

The company under this case-study has a database in which the staff document different bugs found throughout the engine project. The main advantage of this process is to guarantee easy mainstreaming between projects. Test engineers design test-cases based on different inputs such as this database, functional defects found during driving tests, specification requirements, as well as the defects found after the engine has been marketed. The goal is to keep the test-case libraries as complete as possible over time.

Table 3 shows the number of tests considered in this research according to the type of SM.



Table 2. Types of SM Presented in the ECU Software

| Type of SM     | Characteristics   | Validation requirements  | SM   |
|----------------|---|--|--|
| Simple         | a) A reduced number of input and output variables present in the SM and small number of calculations to be done. Furthermore, they are not complex.<br>b) High accuracy needed for calculations in some cases and easy to identify the main functional characteristics of the SM. | SMs require a few manipulations to make the engine ECU reach the desired operating point<br><br>For instance, the SM in charge of detecting whether the accelerator pedal is blocked. The engine ECU must check a few parameters | Such as:<br><br>Temperature estimators<br><br>Brake pedal monitoring             |
| Fairly complex | a) High number of input and output variables present in the module but moderate number of calculations to be performed.<br>b) Moderate accuracy needed for calculations. However, difficult to identify the main functional characteristics of the SM.                            | SMs require more manipulations to make the engine ECU reach the desired operating point.<br>For instance, SMs related with treatment of exhaust gases  | Such as:<br><br>Treatment of exhaust gases systems                               |
| Highly complex | a) High number of input and output variables and number of calculations.<br>b) Calculation not necessarily complex but high number of functional calculations but Moderate/low calculation accuracy   | SMs need weeks to reach the desired operating point For instance, the SM in charge of assessing the diesel dilution rate in the engine oil.  | Such as:<br><br>The SM in charge of controlling the oil rate diluted into diesel |

Table 3. Number of Tests Used in This Research

| Type of SM     | Number of test |
|----------------|----------------|
| Simple         | 250            |
| Fairly complex | 1250           |
| Highly complex | 100            |

Table 4 indicates the methods followed to generate test-cases for each technique.

Table 4. Methods to Generate Test-cases

| Technique  | Method |
|--|--------|
| Cause-effect technique                                   | A1     |
| Model-based testing                                      | A2     |
| One EX combined with dlls and Two EXs combined with dlls | A3     |

A1: A database in which the staff trace different bugs found throughout a project.

In addition, several test-cases come from the software requirements

A2: Pseudorandom values generated by Matelo® to cover a functional model

A3: Pseudorandom values generated by Python scripts

It is important to analyze what A2 and A3 mean. In A2, Matelo® can generate all necessary test-cases with the aim of covering the functional model. In A3, Python scripts also generate test-cases trying to cover the functional model. In addition, they generate pseudorandom values trying to reach functional states not

implemented in the model as exposed in section 2.2. A functional state not implemented in the model involves a use-case not considered by the design team. In other words, a design error. The fact of using fuzzy variables, as exposed later, allows increasing the combination of the inputs of the SM under validation. It must also be taken into account that the scripts in charge of generating pseudorandom values have to avoid impossible combinations such as a vehicle speed at 90 km/h and the first shift engaged.

Table 5 shows examples of test-cases which could be used to check some functionalities of the software by using different techniques. Fuzzy variables are used when using EXs combined with dlls by increasing the number of combinations of the inputs provided by the SM under validation.

Table 5. Examples of Test-Cases

| Feature to be checked                              | Actions to be done  | Expected results  | Technique                        |
|--|---|---|----------------------------------|
| Body Control Unit. Cyclic Redundancy Check invalid | Set a CRC invalid value of the frame BCM_A1   | Check the inhibition of adaptive cruise control               | Cause-effect Model-based testing |
| Diesel Particulate Filter regeneration             | 1. Var1_veh_started=TRUE → Start the vehicle.<br>2. Var2_temperature_exhaust_gas= 600°C. → Do a driving cycle and var3_vehicle_speed= 80km/h → Press the brake pedal to reach 40 km/h. Then Var4_particulate_filter = 40 g → Do not overpass 2000 rpm | When the RG is performing the variable var1_out is activated. | Model-based testing              |
| Diesel Particulate Filter regeneration             | 1. Var1_veh_started=TRUE → Start the vehicle and var2_temperature_exhaust_gas = High. → Do a driving cycle. Var3_vehicle_speed= High → Press the accelerator pedal to reach low speed   | When the RG is performing the variable var1_out is activated. | EXs combined with dlls           |

### 2.1.3. Equipment

The following equipment was used in this research.

- 1) An engine ECU software and hardware designed by the company subjected to this case-study.
- 2) The HIL bench used to conduct this research belongs to the manufacturer dSpace®, model dSpace® Simulator Full-size [32]. It is a versatile HIL simulator capable of emulating the dynamic vehicle behavior.
- 3) When it comes to building the model that serves as the driver's interface, ControlDesk® version 5.1 from dSpace® manufacturer is employed [33]. By using this software, it is possible to carry out all necessary data exchange between the HIL bench and the engine ECU. This model was designed by the company subjected to this case-study and it is validated by the Electronic Validation Powertrain and Hybrids service before using it.
- 4) Throughout this research, it is necessary to make measurements of different software variables stored in the engine ECU memory. To do this, it is imperative to use software that allows reading memory locations. In this research, version 7.1.9 of INCA® was used [31].
- 5) The automation process can be carried out in different ways: by using Python script or AutomationDesk® software [34]. In this research, the Python script was chosen because the staff's skill in AutomationDesk® in the service subjected to this case-study was low.
- 6) Matlab® R2013 and Microsoft Visual Studio 2015 were used to create the dlls used in this research.
- 7) Matelo®. Software used for validation purposes being able to generate test-cases



## 2.2. Validity of This Research

This section describes the validity of the different key elements involved in this research. In addition, readers can find Appendix B which contains a “threats to validity” section.

### 2.2.1. Expert system validation

In this research, the aim of the rule-based EXs is to check whether the software runs properly, carrying out an automatic analysis of the HIL simulation results. There is a knowledge base composed of rules coming from functional requirements set by experts and designers at the beginning of the project. These rules are the base of the expert knowledge. When it comes to the inference engine, it is composed of a functional model describing different states that the system can process when applying the rules presented in the knowledge base. When a test-case is analyzed by the expert systems, after having applied different rules, the inference engine determines in which state the system is. Therefore, the expert systems decide whether the outputs provided by the software are coherent for the test-case simulated. At this point, it is vital to verify in-depth the inference engine (Fig.3). As one can see, all functional states (S1, S2, S3, S4 and S5) are related to a state called S6. This one corresponds to an unexpected or unknown state, which represents a use-case not considered by the designers. By using this state, test-engineers can improve the expert systems if needed. The state 6 will be analyzed in the result section.

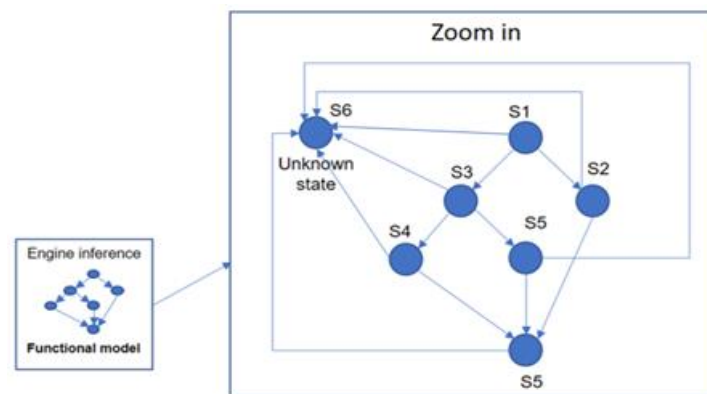


Fig. 3. Inference engine in detail.

### 2.2.2. Dynamic-link library validity

The reader may think that the fact of using dlls could keep the validation process from checking the SM interactions. This statement is not true for several reasons:

- 1) The Simulink® models are the transcription of the functional specifications of the engine ECU and must be met independently of the task scheduling, software–hardware integration, etc. Therefore, for a series of given inputs, the outputs provided by the Simulink® models must be equal to the ones provided by the engine ECU software when no bug is found. Otherwise, the functional specifications are not met.
- 2) The software and hardware integration are considered as the inputs processed to the dll are the consequence of an HIL simulation. The SM interactions are already considered in the data acquisition file. The software must provide the same output values as the Simulink® model (dll). Otherwise, the functional specifications are not met.

### 2.2.3. Measurement conditions

- The information provided by the probes must be equal in all cases (with and without dlls).

- The engine ECU memory must contain no errors before starting the HIL simulation.
- All test-case executions must be conducted on the same HIL bench.
- If a diagnosis defect appears when validating with dll and not when validating without dlls, or vice versa, then the test-case result is rejected.

## 2.3. Limitations

Simulink® models are not always available in an engine ECU project for all SMs. In this research, only approximately 7% of the SMs did not have a Simulink® model. A reader could consider how the conformity of dlls is performed. If there is an error design in a Simulink® model, it will also exist in the software. Therefore, when doing the validation, there will be no difference between the outputs provided by the Simulink® model and the software. EXs can detect this situation.

## 3. Results

### 3.1. Ease for Automation Test-Cases

#### 3.1.1. Simple software modules

Simple SMs, as indicated in the previous section, are characterized by handling a small number of variables. As a result, it is not difficult to reach the values established in the test-case. The problem associated with SM interactions appeared in all SMs considered in this research. For example, by analyzing the measurements obtained in the HIL simulation when validating a simple SM, by using MDA® [34], it was observed that, when actuating the brake pedal, multiple variables were affected and changed their values. When the brake pedal is actuated, the vehicle speed is reduced significantly, even without changing the accelerator pedal position. To decrease the vehicle speed, the engine ECU must control the engine combustion by modifying the air-diesel mixture rate. This phenomenon is regulated by other SMs which were not validated in this process. Therefore, one can conclude that to achieve the values set in the test-case, multiple SMs must be controlled simultaneously. This fact involves a great deal of complexity to code Python scripts.

One of the most important issues to be analyzed is the consequences of not reaching the values set in the test-case. Table 6 shows the results when validating the simple functions by using different techniques. As one can see, the tester-in-the-loop technique offers better results than the automated one without using dlls, because a technician makes the engine ECU reach a specific operating point during the test-case execution. When using dlls, the results are by 4.8% and 14.4% better than the tester-in-the-loop or automation results achieved by using a Python script only.

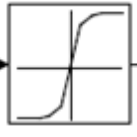
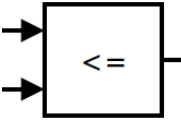
Table 6. Comparisons of Different Techniques for Validating Simple SMs

| <i>Methodology</i>   | <i>Number of cases in which the output value set in the test-case was no longer valid:</i> | <i>Error rate after 250 simulations:</i> | <i>Success rate</i> |
|--|--|--|---------------------|
| <i>Automated with a Python script but without using a dll. / Model-based testing</i> | 49   | 19.6%                                    | 80.4%               |
| <i>Tester-in-the-loop</i>  | 25   | 10%                                      | 90%                 |
| <i>Automated with a Python script and the use of dll.</i>                            | 13   | 5.2%                                     | 94.8%               |

The Simulink® blocks that, in most cases, prevent reaching the values set in the test-case in this research

are shown in Table 7.

Table 7. Most Problematic Simulink Blocks

|   |  |
|---|--|
|  | <p>Interpolator block. In this case, depending on the input values presented to the Simulink® block, an output value is provided by applying an algorithm or an interpolation method.</p>  |
|  | <p>Simulink® native comparator block. It has problems in all its versions (greater than, greater than or equal to, less than, less than or equal to). In engine ECU software, on many occasions the value of a certain physical magnitude (e.g., motor revolutions, vehicle speed) is compared with a calibration threshold.</p> |

It is important to analyze the root cause of the 5.2% failures. After the analysis of the 13 failures shown in table 6, it was verified that the dynamic model used for the HIL simulation failed. Analysis showed that this issue came from 2 SMs. These SMs needed a 10 ms-sample period. Owing to imperfections of the HIL model, latency times and hardware limitations of the HIL bench, in certain occasions this sample time was not respected.

### 3.1.2. Fairly-complex and highly-complex software modules

For fairly-complex and highly complex validation SMs, the number of variables increased up to 80. Therefore, the issue of SM interactions is even more present. Fig. 4 shows the total number and types of variables of a fairly-complex SM and the difficulty of manipulation to make the variables reach a specific value set in a test-case. The graph depicted in Fig. 4 shows that the Boolean variables were easier to be manipulated to reach the desired value, especially when they were related to variables directly linked to the driver's interface-model. If they were linked to analogical variables, it was not easy to reach the desired value. The triangle obtained for a fairly-complex SM was an isosceles whose height is focused on high difficulty. Therefore, the issue about SM interaction arises. On average, after having analyzed 5 SMs it was concluded that at least 40 variables were influenced between them. It is important to explain the nuance of "at least". The Boolean variables are simple to manipulate. Nevertheless, some of them have a direct impact on making the analogical ones reach the desired value established in the test-case. The HIL simulation results are shown in Table 8 in which one can see the number of times the expected output values specified in the test-cases are no longer valid when the SM inputs fail to reach the specific values set in the test-case. At the same time, the most problematic blocks present in the Simulink® models can also be observed (Table 9).

Table 8. Comparisons of Different Techniques for Validating Fairly-complex SMs.

| <i>Methodology</i>   | <i>Number of cases in which the output value set in the test-case was no longer valid:</i> | <i>Error rate after 1250 simulations:</i> | <i>Success rate</i> |
|--|--|---|---------------------|
| <i>Automated with a Python script but without using a dll/ model-based testing</i> | 480  | 38.4%                                     | 61.6%               |
| <i>Tester-in-the-loop</i>  | 200  | 16%                                       | 84%                 |
| <i>Automated with a Python script and the use of dll.</i>                          | 125  | 10%                                       | 90%                 |

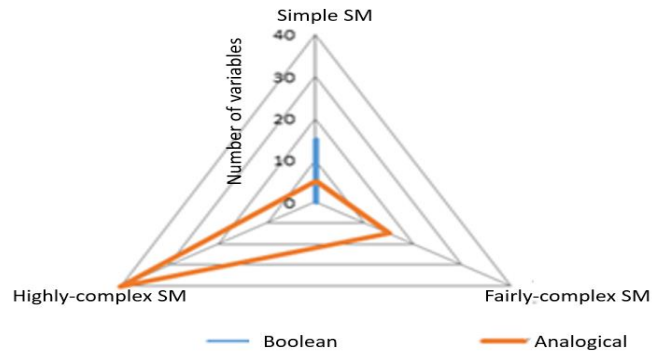


Fig. 4. Type of variables present in an average-complexity SM.

Table 9. Most Problematic Simulink Blocks for An Average SM

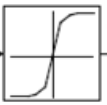
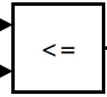
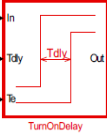
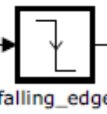
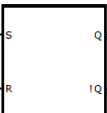
|   |   |
|---|---|
|    | Interpolator block. In this case, depending on the input values presented in the Simulink® block, an output value is provided by applying an algorithm or interpolation method.   |
|    | Simulink® native comparator block. It has problems in all its versions (greater than, greater than or equal to, less than, less than or equal to). In engine ECU software, on many occasions the value of a certain physical magnitude (e.g., motor revolutions, vehicle speed) is compared with a calibration threshold. |
|   | This block sets the output to TRUE while the input In remains TRUE for a certain calibratable time. Otherwise, the output is FALSE. As found in this research, when it comes to average and complex functions, it is more difficult than in simple functions to succeed by making the input In remain stable.             |
|  | This block provides a Boolean type TRUE when a falling edge is detected. Otherwise, it remains FALSE. In this case, when it comes to average and complex functions, it is difficult in certain cases (for example when validating exhaust gas treatment systems) to reach the conditions to generate a falling edge.      |
|  | This block works as a typical RS flip-flop. As in a falling edge block, when it comes to average and complex functions, it is difficult in certain cases (for example when validating exhaust gas treatment systems or oil adaptive maintenance function) to reach the conditions when the S-input could be activated.    |

Table 10. Comparisons of Different Techniques for Validating Highly Complex SMs

| Methodology   | Number of cases in which the output value set in the test-case was no longer valid: | Error rate after 100 simulations: | Success rate |
|---|---|-----------------------------------|--------------|
| Automated with a Python script but without using a dll/ Model-based testing | 61  | 61%                               | 39%          |
| Tester-in-the-loop  | 35  | 35%                               | 65%          |
| Automated with a Python script and the use of dll.                          | 15  | 15%                               | 85%          |

When it comes to a highly complex SM, the triangle obtained is closer to that of an isosceles one with a lower base. This characteristic indicates a greater presence of variables that are difficult to manipulate in a HIL simulation (Fig. 5). In this case, a total of 120 variables that influence the other variables had to be handled. The Simulink® blocks that pose the most problems were the same as those shown in Table 8. The results after the 100 HIL simulations are shown in Table 10.

In highly complex SMs, errors that prevent the HIL simulation from succeeding when using dlls were also detected. When validating a highly complex SM, a lower success-rate with dlls was obtained because these SMs require covering thousands of kilometers (close to 20,000 km in some cases). Thus, the probability of failure in the simulator increases. Considering the strong SM interaction, it is unlikely to reach the specific values set in the test-case. Thus, the tester-in-the-loop solution offers worse results than when using dlls.

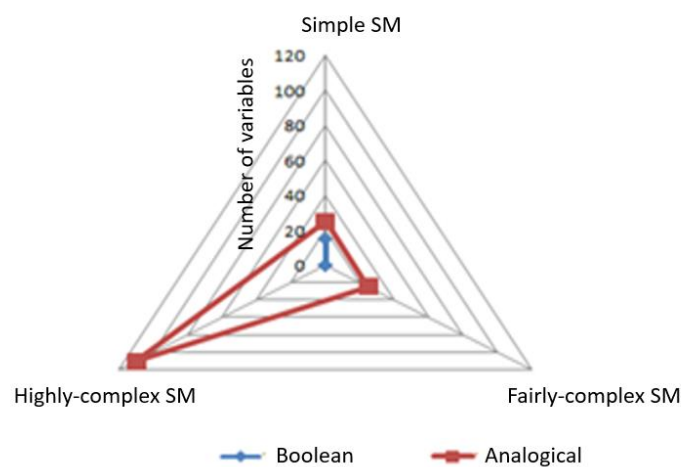


Fig. 5. Type of variables present in a high-complex SM.

In fairly and highly complex SMs, at any given time, it was observed that several variables were close to the values previously set in the test-case as long as other values were quite far. If some manipulations were performed to make all the variables closer to the values set in the test-case, then the ones which were far from the expected values started to get closer, and the remaining variables started to get further. Thus, it is unlikely to be able to reach the input values set in a test case owing to SM interactions in such complex software as in an HIL simulation, which was discussed in section 1. Fig. 6 shows how, by increasing the error tolerance against the value set in the test-case for the variables that constitute the test-case, the number of variables that remained within those tolerance margins increased. However, in any case, it was never possible to make all the variables remain within the established tolerance range. This fact happened when executing the test-cases manually or automatically. As a result, these results show the great difficulty of validating an engine ECU software version by using HIL simulation.

Fig. 7 summarizes the results obtained when using or not using dlls in an HIL simulation. As shown, dlls improve the HIL results in a significant way for all types of SMs, especially for simple and fairly-complex functions. It must be reminded that estimator SMs belong mainly to simple functions. That is why one can see such a huge difference when comparing the results obtained when activating or not activating dlls in an HIL simulation. In fairly and highly complex functions, it must also be noted that the SMs that require performing of many calculations belong to this category. Thus, there is also a significant difference when using dlls.

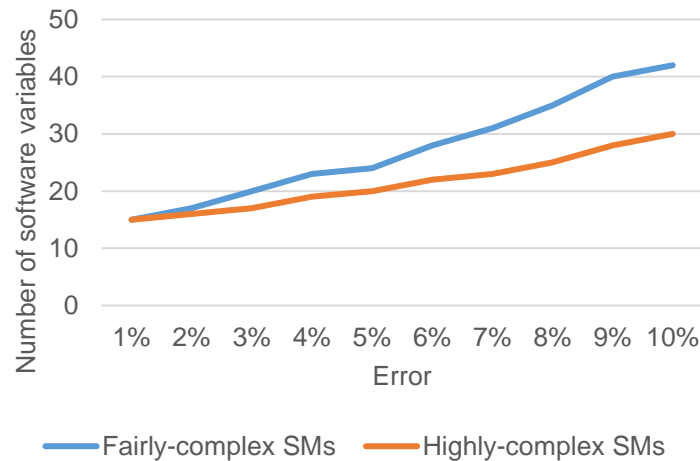


Fig. 6. Error trend depending on error tolerance of the SM inputs.

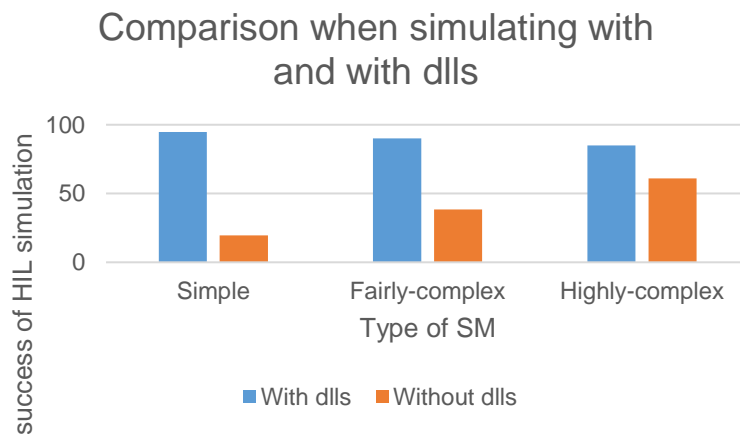


Fig. 7. Comparison of results obtained when using and not using dlls.

The reader may think that the automation process is not useful when validating the engine ECU software. This conclusion is false as there are some SMs, especially those related to electronics, which can be successfully automated such as CAN (Controller Area Network) and LIN (Local Interconnect network) bus or the basic functionalities of adaptive cruise control with the capacity to stop the vehicle (see Table 11). These statements have been proven in this research as shown in Table 12.

In this research, the SMs listed in Table 10 were used.

Table 11. List of SMs Used and Tested in This Research

| <i>Functions or SMs tested</i>                         | <i>Number of test-cases tested</i> |
|--|------------------------------------|
| <i>CAN Bus</i>   | 600                                |
| <i>Driving aid systems</i>                             | 140                                |
| <i>Pressure and Temperature carburant probe (SENT)</i> | 100                                |
| <i>LIN Bus</i>   | 50                                 |



Table 12. Comparisons of Different Techniques for Validating Functions Depicted in Table 10

| Methodology   | Number of cases in which the output value set in the test-case was no longer valid: | Error rate after 1000 simulations: | Success rate |
|---|---|------------------------------------|--------------|
| Automated with a Python script but without using a dll. | 12  | 1.2%                               | 98.8%        |
| Tester-in-the-loop                                      | 13  | 1.3%                               | 98.7%        |

Table 11 does not show the results for automation with dlls as most of the function did not have a Simulink® model as exposed in Section 2.

### 3.2. Functional Coverage

The functional coverage has been assessed by using equation (1) which is widely employed in the automotive sector. Table 13 shows the total number of functional requirements associated with the SMs validated in this research (section 2.1.1).

$$FC = \frac{\text{number of software requirements tested by a technique}}{\text{Number of software requirements indicated in table 15}} \times 100 \quad (1)$$

Table 13. Number of Total Functional Requirements

| Type of SM     | Number of requirements |
|----------------|------------------------|
| Simple         | 75                     |
| Fairly-complex | 400                    |
| Highly-complex | 510                    |

Table 14 depicts the results obtained for each technique in this research:

Table 14. Functional Coverage Obtained for Each Research

| Technique                         | Simple SM              |                         | Fairly-complex SM      |                         | Highly-complex SM      |                         |
|-----------------------------------|------------------------|-------------------------|------------------------|-------------------------|------------------------|-------------------------|
|                                   | Number of rules tested | Functional coverage (%) | Number of rules tested | Functional coverage (%) | Number of rules tested | Functional coverage (%) |
| Cause-effect                      | 64                     | 85.3                    | 312                    | 78                      | 357                    | 70                      |
| Model-based testing               | 64                     | 85.3                    | 312                    | 78                      | 357                    | 70                      |
| Tester-in-the-loop                | 64                     | 85.3                    | 312                    | 78                      | 357                    | 70                      |
| Performance EX combined with dlls | 68                     | 90.7                    | 348                    | 87                      | 445                    | 87.2                    |

#### 3.2.1. Cause-effect technique and tester-in-the-loop

All test-cases run in this research by using these techniques are similar to the ones depicted in table 5. It must be reminded that the test-cases can be run in a manual way or by employing Python scripts with the aim of automating the process.

The main limitation of the cause-effect technique is test-case redundancy. Many test cases run to validate the software were indeed linked to the same software requirements. The main reason behind this issue is

the lack of a functional model of the SM under validation. When a use-case is not considered initially in the software requirements, it cannot be found by the cause-effect technique. In addition, bugs linked to calculation errors cannot be detected.

### 3.2.2. Model-based testing

When using Matelo®, it is important to expose the problems found. If the test engineer let Matelo® generate test-cases, this software will assign specific values for each input of the SM under validation. As a consequence, the problems of SM interactions, are identified. The only way to overcome this issue is to use fuzzy values<sup>5</sup> combined with dlls. In this case, results are similar to the ones obtained when using a performance EX as long as dlls are used. Matelo® can be used also in such a way that Matelo® will not generate the test-case but it will control the automation process. In other words, the test engineer must code a Python script to generate the test -cases needed and then Matelo® will check the functional states covered as the automation is performed.

In the present research, the test engineer codes Python scripts with the aim of running the same test-cases as for the manual execution, the tester-in-the-loop and so on. These test-cases are present in the data base that is mentioned in Section 2.1.1. Consequently, the results shown in table 9 are the same for the cause-effect technique and the model based-testing one.

### 3.2.3. Performance expert system

The rule-based EX allows specifying the functional requirements of SMs. The method followed to validate the EX was described in section 2.2.1. Two phases are considered: a validation and a test one. On the one hand, the former consists of verifying a certain number of test-cases depending on the type of SMs to assess the EX performance to be sure that the EXs seem to work properly (Table 15). Table 16 shows the results obtained during the first phase in which a 83.3% success rate was obtained.

Table 15. Number of Test-cases Used to Validate the EXs

|                    | Number of test-cases used to test the EX during the verification process | Number of test-cases used to test the EX during the acceptance process |
|--------------------|--|--|
| Simple SMs         | 100  | 80   |
| Fairly complex SMs | 120  | 50   |
| Highly complex SMs | 5  | 2  |

Table 16. Errors Detected when Validating the EXs

| Type of error                                | Percentage (%) | Cases | Explanation   |
|--|----------------|-------|---|
| Wrong syntaxes                               | 8.8            | 20    | Because the rules used to design the EXs are extremely complex, the programmer made coding errors.                        |
| Incoherence between rules                    | 3.5            | 8     | In some cases of wrong performance of the EX, incoherence between rules was found.  |
| Misunderstanding of technical specifications | 3.1            | 7     | Because of innovative evolutions in some parts of the engine, some technical specifications were not understood properly. |
| Rules not coded or forgotten                 | 1.3            | 3     | This error is owing to the same misunderstanding of technical specifications.   |

Once the errors were corrected, the test phase was performed to assure that the EXs would assess the software behavior properly. If no error occurred the EX was accepted.

<sup>5</sup> Equivalence class is the concept described in Matelo® documentation to generate values similar to fuzzy values.

The main conclusion that can be drawn is all possible use-cases are not checked when no EX is used. When it comes to simple and medium-complexity SMs, the number of unchecked functional states is shown in Table 17. The number of untested rules in a medium or highly complex function is greater because of the large number of use cases involved in this type of SMs.

Table 17. Number of Rules or Functional States not Checked when an EX is not Used

| Type of SM     | Number of functional states not tested without using an EX |
|----------------|--|
| Simple         | 4  |
| Fairly-complex | 36   |
| Highly-complex | 88   |

These improvements are mainly based on two reasons:

- 1) DLLs allow controlling better the HIL simulation as it is possible to know at any time if the current state of the engine ECU is coherent or not as already exposed in this research.
- 2) EXs assess the functional coverage easily. The reader can think that a similar result could be obtained by using Matelo® combined with DLLs. Matelo® generates test-cases off-line. If after the HIL simulation, the inputs of SM under validation do not reach the desired operating point, Matelo® cannot calculate the expected value for the current state of the engine ECU in-real time. To do this, Matelo® must be used in another way as exposed in 3.2.2.
- 3) DLLs allow finding bugs linked to calculation errors.

### 3.3. Productivity Gain

It is essential to check if EXs implementation respects the timeframes of the project by analyzing several factors. As shown in Table 18, the gain is positive for fairly and highly-complex SMs when using an EX. This gain comes from the automation process which allows testing test-cases quicker. In addition, these test-cases can be always run thanks to DLLs. Consequently, an EX combined with DLLs performs better than the other techniques. For simple SMs, the result is different as the HIL simulation implies that very simple and quick manipulations are conducted on the driver's interface model. As a result, the time gain is negative and the timeframe of the project may not be respected. It must be reminded that several projects are being developed at the same time by car manufacturers: diesel or gasoline engines. Between these types of engines, one can find considerable differences when it comes to torque structure or after treatment of exhaust gas systems. However, when comparing engines of the same groups, they are remarkably similar. As a result, an EX designed for a project can be used for another one. Then, only the automation and validation phases will be performed. As one can see in these phases, this technique outperforms the other ones. The main conclusions which can be drawn is that the proposed technique always meets the project planning especially when there are several engines developing at the same time.

### 3.4. Bug Detection

Fig. 8 shows the bugs found by each technique when running the test-cases as described in section 2. The tester-in-the-loop offers a better performance than the automation process as it can make the system reach critical states that are not easy to reach when only using a Python script. There are not significant differences between manual and tester-in-the-loop techniques when it comes to bug detection as there is a technician who participates in the test-case execution, Python scripts detect fewer bugs than the rest of the techniques as test-cases are difficult to automate due to SM interactions. As a consequence, when the system reaches an operating point close to the one established in the test-cases, the outputs indicated in the

test-cases may be no longer valid. To solve these problems, fuzzy values for the SM inputs may be used as exposed later in this section.

Table 18. Time Needed to Design Test-cases and Rule-based EXs

|  |  | Simple functions | Fairly complex function | Complex function |
|--|--|------------------|-------------------------|------------------|
|  | Total time for designing test-cases (h)  | 8                | 80                      | 120              |
| Time for designing and coding  | Time for coding, design and validate EXs and Python script for the automation process(h) | 4                | 35                      | 70               |
|  | Time for preparing dlls (h)  | 2                | 6                       | 10               |
|  | Time for coding a Python script (h)  | 4                | 32                      | 50               |
|  | Time for coding when using the tester-in-the-loop (h)                                    | 2                | 25                      | 35               |
|  | Total time for designing and coding when using EXs (h)                                   | 14               | 121                     | 200              |
|  | Total time for designing and coding when using Python scripts (h)                        | 12               | 112                     | 170              |
|  | Total time for designing and coding when using the tester-in-the-loop (h)                | 10               | 105                     | 155              |
|  | Total time for designing when executing a test-case manually (h)                         | 8                | 80                      | 120              |
| Test-case execution  | Time for executing an automated test-case by using EXs (h)                               | 0.32             | 13                      | 73               |
|  | Time for executing an automated test-case (h)  | 0.25             | 12.5                    | 72               |
|  | Time for executing a test-case by using the tester-in-the-loop (h)                       | 0.46             | 62                      | 80               |
|  | Time for executing a test-case manually (h)  | 0.5              | 80                      | 170              |
| Validation   | Time for validating the results with automation (h) <sup>(1)</sup>                       | 0.00028          | 0.00347                 | 0.00044          |
|  | Time for validating the results without automation (h) <sup>(2)</sup>                    | 1.67             | 20.83                   | 2.33             |
| Total time   | Total time by using EXs (h)  | 14.32            | 134.00                  | 273.00           |
|  | Total time with automation by using Python scripts (h)                                   | 12.25            | 124.50                  | 242.00           |
|  | Total time when using the tester-in-the-loop (h)   | 10.46            | 167.00                  | 235.00           |
|  | Total time without automation (h)  | 10.17            | 180.83                  | 292.33           |
| (1) In this case, the following data have been considered: 50 test-cases for simple functions with an execution time of 0.02 s, 250 test-cases for fairly complex functions with an execution time of 0.05 s, and 50 test-cases for complex functions with an execution time of 0.08 s. The execution time was measured by using the Python function time clock. |  |                  |                         |                  |

The results obtained in this research show that EXs with dlls give better performance and can be used to test more functional states and detect more bugs than the other techniques. Basically, this statement is based on two main reasons:

- 1) The problems coming from the SM interactions are fixed due to dlls. Even though the operating point established in the test-case is not reached, dlls can provide the right values expected from the

software. Consequently, the test-cases can be successfully run and the automation process can validate the HIL simulation results automatically.

- 2) The functional coverage is improved due to the existence of the functional model. In addition, this model can be covered easily thanks to the automation success by using the dlls.

It is also important to establish the main types of bugs found for each technique (Table 19).

- 3) When the bug is linked to calculation errors (calculation faults)
- 4) When no code error occurred but there was unexpected performance software. This issue can come from an error design in the SM under validation (performance faults).
- 5) When there is a code bug. This means the programmer has made a mistake and coded differently from what was indicated in the specifications.

Table 19. Type of Bugs Detected

|                         | Calculation faults | Bugs | Performance faults |
|-------------------------|--------------------|------|--------------------|
| Manual validation       | 0                  | 12   | 2                  |
| Tester-in-the-loop      | 0                  | 14   | 1                  |
| Automation without dlls | 0                  | 10   | 0                  |
| Model-based testing     | 0                  | 10   | 5                  |
| EXs and dlls            | 5                  | 14   | 4                  |

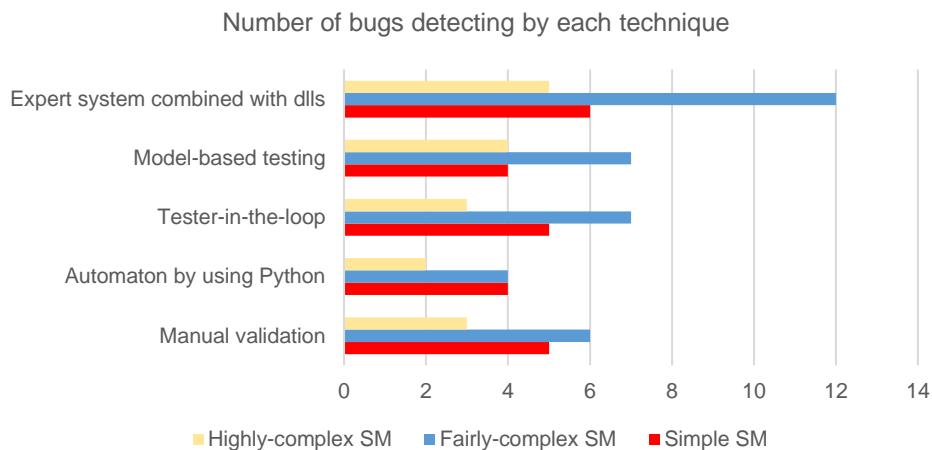


Fig. 8. Bugs found when using different techniques.

### 3.5. Costs

It is necessary to discuss costs. The first one is associated with the licenses needed to use a specific technique (already discussed). The other one is linked to software versions needed to correct bugs detected at the end of the project. This can be caused by two things. Firstly, certain SMs (especially those related to advanced driver assistance systems) cannot be tested at the beginning of the project. The validation of these functions needs very mature software of some ECUs present in the network (electronic stability program ECU, body control unit, radars, cameras, gearbox ECU in automatic cars). Secondly, some bugs appear when testing some use-cases that were not considered in the validation process. When these bugs are detected, the project team must decide whether the bug has a significant functional impact and therefore require correction of the software. Otherwise, the bug can be corrected in future engine projects and no correction

will be made<sup>6</sup>. Developing new software versions involves a high cost but also might imply updating the ECU of vehicles that have already been marketed. The results showed that EXs combined with dlls detected two bugs that would have required corrective software development. These bugs were not detected using the cause-effect technique, the model-based testing one, the manual execution or the model-based testing one.

The reader might think that, in case of bugs in the Simulink® model, the software will also contain these faults. As a result, no bug will be detected by using the method proposed in this research. This study has proven that this statement is true and that is why the performance EX must be used.

### 3.6. Comparison among other Methods

When performing an HIL simulation, it is not easy to reach the values indicated in the test-case due to SM interactions. Fig. 9 shows an example of a histogram displaying speed value. Depending on the value reached, the output can be 1 or 0. Consequently, if a test-case indicates that the speed must be 60 km/h, the accuracy is a critical factor and the expected output could be no longer valid.

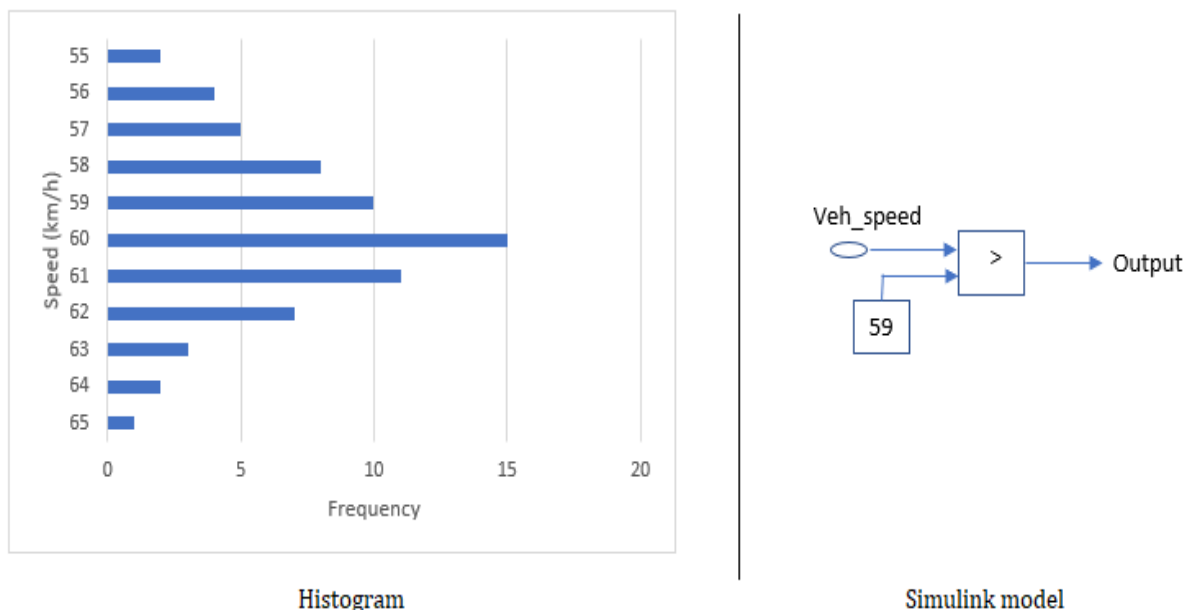


Fig. 9. Example of test-case.

A comparison among different techniques is shown in Table 20.

Table 20. Comparison among Techniques

|                        | Manual validation   | Automation without dll | Model-based testing | EXs and dlls   |
|------------------------|---|------------------------|---------------------|--|
| Validity of test-cases | As shown in Fig.9, it is necessary to reach the exact value indicated in the test-case. Otherwise, the validation process cannot be performed automatically |                        |                     | Even though the values indicated in the test-case are not reached, the validation can be performed automatically |

<sup>6</sup> There are several ways to classify bugs in the automotive sector. Sometimes they are broken into K1, K2, K3 or K4 depending on the importance. K1 and K2 must be always fixed before the vehicle is marketed. However, K3 and K4 will be fixed in future vehicles and the current engine can be marketed without any correction.



|                                    |   |   |
|------------------------------------|---|---|
| Accuracy needed                    | The test-case output may be no longer valid (Fig.9). The test-engineer should check the specification to confirm the expected output  | The test-case output may be no longer valid. However, dlls can check the expected output automatically                                |
| Complexity                         | As shown in Fig.9, it is highly complicated to reach the specific values indicated in the test-cases (see Table 6, Table 8 and Table 10) due to SM interactions                                     | Even though the HIL simulation does not reach the specific values indicated in the test-case, the validation process can be performed |
| Robustness in case of failure      | During the HIL simulation, the engine ECU can detect a failure (low rail pressure, turbo failure, etc.). In that case, the test-case output is no longer valid                                      | Even though the engine ECU detects a failure, the dll can detect the expected output in that case                                     |
| Reading ECU variables in real-time | INCA software does not allow reading in-real time variables by using Python while data acquisition is performed. The test-engineer has to analyze the data acquisition to check if a bug is present | The dll can do the validation process automatically when using a C-code at the same time.   |

#### 4. Conclusions

This research, conducted at the second most important European car manufacturer, is focused on the software validation of an engine ECU by using dynamic-link libraries (dlls) and an EX (ES). This combination allows the detection of software performance and coding bugs. As shown in this research, dlls and ES can detect bugs that other techniques such as the black-box or the tester-in-the-loop cannot, especially those in temperature estimator SMs and after-treatment of exhaust gases SMs, which require accurate calculations. The obtained results show how dlls and the EX can improve the HIL success rate compared with the tester-in-the-loop technique and can execute 4.8% of the test-cases in simple validation SMs, 6% of the test-cases of fairly complex SMs and 20% of the test-cases of highly complex SMs despite the presence of SM interactions. In comparison to the use of a Python script without using a dll, the dlls and the EX can improve the HIL and can execute 14.4% of the test-cases in simple validation SMs, 28.4% of the test-cases of fairly complex SMs and 46% of the test-cases of highly complex SMs. As a result, dlls can overcome the issue linked to SM interactions. In addition, between 9 and 13 more bugs were found when using the EX and dlls, six of which could not be detected by other techniques. Even though EXs and dlls require more time to be implemented, the timeframe of the project was respected.

#### Appendix A

The parameters shown in Table 21 were chosen to analyse the results obtained in this research.

Table 21. Parameters Considered in the Sensitivity Analysis

| Parameter  | Meaning   |
|--|---|
| % error associated with SM inputs                  | % Deviation of the SM inputs obtained once the HIL simulation is finished with respect to the input values set in the test-case.  |
| Needed time to code Python scripts                 | Trend of coding time according to the staff's training in Python. This parameter influences productivity gain thanks to the automation process by using dlls combined with EXs. |
| Gain trend versus productivity (measured in hours) | According to the SM type, the relationship between the number of test-cases and the productivity gain (measured in hours) versus the manual test-case execution                 |

Fig. A.I. Shows the results obtained after the execution of 120 HIL simulations involving the speed value.

Afterwards, the speed values obtained were classified depending on the speed error with respect to the value set in the test-case (60 km / h). As shown in Fig.A.I, as the error percentage becomes greater, so does the failure of the HIL simulation. In other words, the lower the error is, the greater the probability for the HIL simulation to fail without using dlls is, as the output value established in the test-case is no longer valid. Reaching errors close to 1% depends on the quality of the scripts and their capacity of controlling not only the SM under validation but also other SMs that can influence the final result.

The calculation performed for simple SMs is extraordinary difficult to perform for fairly and highly complex SMs owing to SM interactions. In practice, it is observed that an attempt to make a variable reach a specific value set in the test-case distances the other variables from the values established in the test-case. Therefore, perform the classifications based on the error (Fig. A.I) for functions containing more than 100 variables is highly complex.

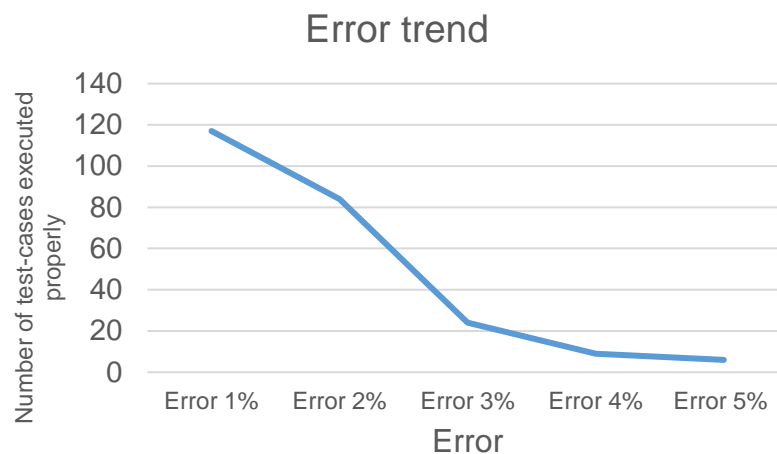


Fig A.I. HIL simulation success depending on the input errors with regard to the inputs values set in the test-case.

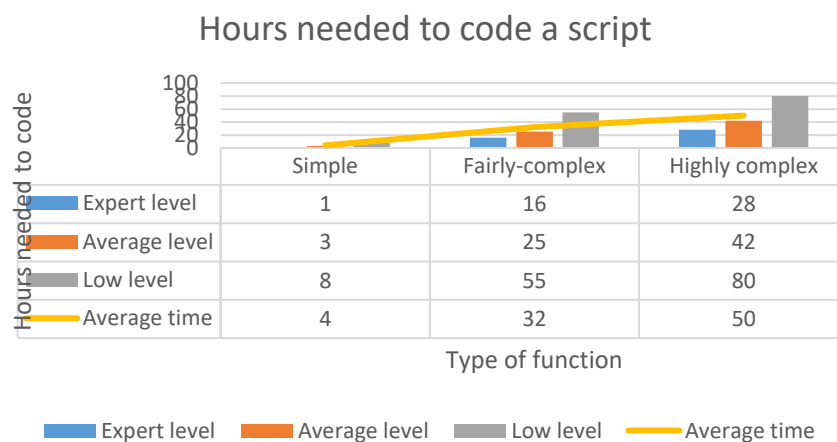


Fig.A.II Hours needed to code the Python scripts

The quality of the scripts depends on the staff's training in Python. Table 22 and Fig. A.II show how the time needed to code Python script evolves, depending on the staff's skills in Python as well as the type of SM to be validated.

When automating the HIL simulation, the productivity gain (in hours) by using Python, EXs and dlls depends on the number of test cases to be conducted. As shown in Fig. A.III, the gain for fairly and highly-complex SMs is always positive thanks to the improvements introduced by automation and automatic

verification of the results obtained in the HIL simulation. Simple SMs give positive gains of 143 test-cases. This result indicates that the only way to obtain a positive gain for this type of SMs is:

- a) Using the scripts in all possible engine projects being conducted in a powertrain validation service.
- b) Conducting regression tests every time software is delivered.

Table 22. Staff's Training in Python

| Group         | Experience in coding Python scripts | Number of members |
|---------------|-------------------------------------|-------------------|
| Expert level  | More than 2 years                   | 10                |
| Average level | Between 1 and 2 years               | 15                |
| Low level     | Less than 1 year                    | 15                |

Gain depending on the type of SM

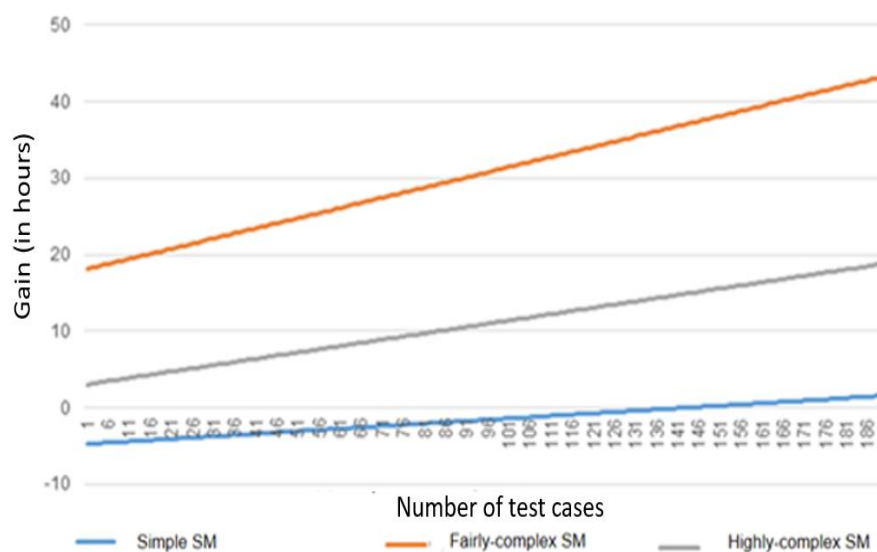


Fig. A.III Gain trend versus test-cases conducted.

## Appendix B

As exposed by Feldt and Magazinius, “a conclusion validity is focused on how sure we can be that the treatment we used in an experiment really is related to the actual outcome we observed” [35]. Many factors can threaten the results of a research. For the current paper, the factors depicted in Table 23 were chosen [35]. In this section, the actions taken to reduce these threats are exposed.

Table 23. Factors Chosen for the Conclusion Validity Assessment

| Id Factor | Factor                                    | Description  |
|-----------|---|--|
| 1         | Low Statistical Power                     | Studies with low power often result in the conclusion that no relationship exists when in fact a true relationship exists. Insufficient power most often results in using too few participants |
| 2         | Violated Assumptions of Statistical Tests | All statistical tests rely on assumptions. When the assumptions are violated, the researcher and consumer may be misled about the probabilities of making Type I and Type II errors.           |

|   |   |   |
|---|---|---|
| 3 | "Fishing" and Error-Rate Problems               | Conducting many statistical tests on a data set without stating specific hypotheses. This procedure inappropriately capitalizes on chance events and increases the probability of Type I error occurring. |
| 4 | Unreliability of Measures                       | Unreliable measures introduce error variance and obscure the true state of affairs.   |
| 5 | Restriction of Range                            | The restricted range usually occurs because the instrument measuring the variable is not sensitive enough to be measured at the upper limit (ceiling effects) or at the lower limit (floor effects).      |
| 6 | Unreliability of Treatment Implementation       | It is possible for treatments to be delivered or implemented in variety of ways.  |
| 7 | Extraneous Variance in the Experimental Setting | Any aspect of the experimental setting that leads to variability in responding will increase the error variance and obscure a true relationship.  |
| 8 | Heterogeneity of Units                          | Differences in experimental units can often lead to variability in responding.  |
| 9 | Inaccurate Effect Size Estimation               | There are instances when effects detected in studies will be inaccurately estimated. Outliers can dramatically affect correlations, for example.  |

#### Id factor 1

The number of people to be considered is limited to the members who composed this service. In this case, it was 40 people distributed as follows: 19 engineers and 21 technicians. The main question that may arise is if this number is enough to do this research. Considering the data that this company possessed at that time, its staff was similar to other manufacturers. In our criteria, the mix ratio between technicians and engineers was satisfactory. Of course, each group of experts can have different skills, but this fact was considered in the sensitivity analysis (Appendix B). Therefore, the mitigation strategy chosen was to perform a sensitivity analysis to be sure how the skills needed to develop the authors' proposal in a validation service may impact.

Table 24. Seniority in the Service

| Number of people | Seniority in the service |
|------------------|--------------------------|
| 7                | < one year               |
| 15               | Between 1 and 3 years    |
| 11               | Between 3 and 5 years    |
| 7                | More than 5 years        |

#### Id factor 2

One of the key aspects of this research is the staff's training. This obvious statement can be a threat to validity as if the results are given considering a staff with extensive knowledge and skills needed to implement the proposal of this paper, the assessment of productivity gain will not be accurate or even false for other scenarios. That is why for this research, technicians and engineers having different levels in coding Python or in engine operation knowledge were chosen. This is guaranteed as depicted in table 24. Then the

influence of all the aforementioned aspects, was analyzed in the sensitivity analysis which was considered as a mitigation strategy again.

#### Id factor 3

The software of an engine ECU is extremely complex as a whole, but it cannot be considered that all SMs are equal. It is not possible to draw exactly the same conclusions for a simple SM as for a highly complex one. To avoid Id factor 3, the authors decided to classify the SMs present in the engine ECU software as follows: simple, fairly-complex and highly complex SMs. This classification was validated by the staff of the company considering the feedback coming from other projects. As the reader can see in this research every conclusion is focused on a type of SMs and the authors do not generalize for the whole engine ECU software. Therefore, the mitigation strategy chosen divided the engine ECU software into several types of SMs considering a series of factors as depicted in table 2.

#### Id factor 4

The mitigation strategy chosen to avoid this factor was to assure that the measures were taken in the same conditions. To assure this, a procedure was written as the reader can see in section 2.5.2. which described when measures can be accepted and when must be rejected it. Therefore, the staff involved in this research agreed with this criterium and it was applied during the whole study. In addition, the decision of using the same HIL bench aimed to reduce the variance as all measures were implemented by using the same probes, wires and HIL hardware.

It is also important to consider how the conformity of EXs was assured. First, by meetings organized by the validation department and the expert of a specific SM with a large experience in design and engine projects. Second, by using data coming from many types of real driving tests. Third, by establishing a procedure described in section 2.5. All aforementioned constituted the mitigation strategy to assure the EXs conformity. In addition, driving tests were carried out at the same time by tuning teams. Their results did not contradict the ones obtained in this research.

#### Id factor 5

The probes installed in the HIL bench are designed to provide the measures in the range established by the design team. Then, the measure is sent by the CAN bus and record it by using the software INCA®. As a result, in this research, the methodology proposed consists of recording the information sent by other probes and the HIL bench through the CAN bus. The only factor to be assured is to generate test-cases which make the probes work in operating points close to their saturation limits. This conclusion was drawn in the test libraries and it was also established as a mitigation strategy for this research.

In addition, the EXs were validated as exposed for Id factor 4.

#### Id factor 6

The mitigation strategy chosen in this case was aimed to standardize the procedure to perform the HIL simulation with the aim of obtaining data which allowed to draw the conclusions of this research. The method is described in section 2.1.

#### Id factor 7

The company subjected to this case-study and the authors looked for a variance but under control with the aim of assessing the different factors which may impact the results shown in this research. It must be considered that the members of the staff of a validation service may change their positions in the company. As a result, the department may have more specialized people at a specific moment and vice versa in other occasions. Consequently, the company wanted to do this research considering different scenarios depending on the staff's training, for example. The authors proceeded to do a sensitivity analysis as a mitigation strategy to justify how the results might change.

#### Id factor 8

In this research, the heterogeneity is necessary to perform the sensitivity analysis. Different scenarios were established considering people very well trained, fairly trained and people who need to enhance their skills such as Python script coding. This factor was considered in the sensitivity analysis. As a result, this research can be useful for different companies which may have a different level of Python script or knowledge of how an engine works in depth. The rest of the data used in this research come from the software coded by Robert Bosch. Its units were chosen by the design team when developing the engine. Thus, the mitigation strategy was to assure how the results could change considering the heterogeneity present in the validation service which may also be present in other manufacturers.

#### Id factor 9

As exposed in Id factor 1, the population size was considered right. It is true that a validation department can have more or less staff. It must also be reminded that a validation department is of high cost for companies, so they try to limit the number of people who run the service. In addition, the size of this department is not linked to the number of vehicles manufactured every year by different manufacturers once the software is validated, it is embedded in the ECU and the supplier delivers the ECU the calibration and software loaded in. As a result, the authors and the company subjected to this case-study consider that by using a sensitivity analysis the trend between different factors is established in the sensitivity analysis and can be considered as a guide in case of reducing or increasing the staff members. Therefore, the mitigation strategy can be used to assure a heterogeneity in the population and perform a sensitivity analysis.

## References

- [1] Broy, M. (2006). Challenges in automotive software engineering. *Proceedings of the 28<sup>th</sup> International Conference on Software Engineering* (pp. 33-42). ACM.
- [2] Navet, N., & Simonot, F. (2008). *Automotive Embedded Systems Handbook* (1<sup>st</sup> ed.). CRC Press.
- [3] Roychoudhury, A. (2009). *Embedded Systems and Software Validation* (1<sup>st</sup> ed.). Morgan Kaufmann Publishers
- [4] Gajjar, M. J. (2017). *Sensor Validation and Hardware-Software Code Design. Mobile Sensors and Context-Aware Computing* (1<sup>st</sup> ed.). Morgan Kaufmann.
- [5] Rajan A., & Wahl T. (2013). *CESAR - Cost-Efficient Methods and Processes for Safety-Relevant Embedded Systems* (1<sup>st</sup> ed.). Springer.
- [6] Oshana, R. (2013). *Software Engineering for Embedded Systems: Methods, Practical Techniques and Applications* (1<sup>st</sup> ed.). Elsevier
- [7] Oberkamp, W. L., & Roy, C. J. (2010). *Verification and Validation in Scientific Computing* (1<sup>st</sup> ed.). Cambridge University Press.
- [8] Westland, J. C. (2002). The cost of errors in software development: evidence from industry. *Journal of Systems and Software*, 62(1), 1-9.
- [9] Zaman, N. (2015). *Automotive Electronics Design Fundamentals*. (1<sup>st</sup> ed.). Springer.
- [10] BOSCH, (2013). *BOSCH Automotive Electrics and Automotive Electronics* (1<sup>st</sup> ed.). Robert BOSCH.
- [11] Garousi, V., & Mäntylä, M. V. (2016). A systematic literature reviews in software testing. *Information and Software Technology*, 80, 195-216
- [12] Kasaju A., Petersen K., & Mantyla M. V. (2013). Analyzing an automotive testing process with evidence-based software engineering. *Information and Software Technology*, 55,1237-1259.
- [13] Jungui Z., Zhiyi Z., Peizhang X., & Jingyu W. (2015). A test data generation approach for automotive software. *Proceedings of the Conference IEEE*.
- [14] Hoffmann A., Quante J., & Woehrle M. (2016). Experience report: White box test-case generation for



automotive embedded software. *Proceedings of the IEEE Ninth International Conference on Software Testing*

- [15] Saglietti, F., Oster, N., & Pinte, F. (2008). White and grey-box verification approaches for safety and security critical software systems. *Information Security Technical Report*, 13(1), 10-16
- [16] Wernick, P., & Lehman, M. (1999). Software process white box modelling for FEAST/1. *Journal of System and Software*, 46(2-3), 193-201
- [17] Awedikian R., & Yannou, B. (2010). Design of a validation test process of an automotive software. *International Journal on Interactive and Manufacturing*, 4(4), 259-268
- [18] Conrad, M. (2005). Retrieved from: <http://drops.dagstuhl.de/opus/volltexte/2005/325/>
- [19] Chunduri, A. (2016). Retrieved from: <http://www.diva-portal.org/smash/get/diva2:945731/FULLTEXT02>
- [20] Skruch, P., & Buchala, G. (2014). Model-based real-time testing of embedded automotive systems. *SAE Int. J. Passeng. Cars – Electron. Electr. Sys.*, 7(2), 337-344
- [21] Raffaëlli, L., Vallée, F., Fayolle, G., Armines, A., Souza, P., Rouah, X., Pfeiffer, M., Géronimi, S., Pétrot, F., & Ahiad, S. (2016). *Proceedings of the Embedded Real Time Software and Systems Conference*.
- [22] All4Tec. (2017). Retrieved from: <http://www.all4tec.net/MaTeLo/homematelo.html>
- [23] Ilić, V., Popić, S., & Kovačić, M. (2016). Data flow in automated testing of the complex automotive electronic control units. *IEEE Instrumentation & Measurement Magazine*.
- [24] Keller, R., Alink, T., Pfeifer, C., Eckert, C. M., Clarkson, P. J., & Albers, A. (2007). *Proceedings of the International Conference on Engineering Design*.
- [25] Catelani, M., Ciani, L., Scarano, V. L., & Bacioccola, A. (2011). Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use. *Computer Standards & Interfaces*, 33(2), 152-8.
- [26] Köhl, S., Lemp, D., & Plöger, M. (2003). ECU network testing by hardware-in-the-loop simulation. *ATZ Worldwide*, 105(10), 10-12.
- [27] National Instrument. Retrieved from: <http://www.ni.com/white-paper/10343/en/> Accessed January 2019.
- [28] Winemantech. Retrieved from: <https://www.winemantech.com/services/hardware-in-the-loop-test-systems/>
- [29] Petrenko, A., Nguena, T., & Ramesh, S. (2015). Model-based testing of automotive software: some challenges and solutions. *Proceedings of the 52th Congress ACM/IEEE Design Automation Conference*.
- [30] Matlabcentral Fileexchange. Retrieved from: <https://www.mathworks.com/matlabcentral/fileexchange/9709-from-simulink-to-dll-a-tutorial>
- [31] NCA Software Product etas. Retrieved From: [https://www.etas.com/en/products/inca\\_software\\_products.php](https://www.etas.com/en/products/inca_software_products.php) Accessed January 2019
- [32] DSPACE. Simulator Hardware. Retrieved from: [https://www.dspace.com/en/inc/home/products/hw/simulator\\_hardware/dspace\\_simulator\\_full\\_size.cfm](https://www.dspace.com/en/inc/home/products/hw/simulator_hardware/dspace_simulator_full_size.cfm)
- [33] DSPACE Experimentandvisualization. Retrieved from: <https://www.dspace.com/en/inc/home/products/sw/experimentandvisualization/controldesk.cfm> Accessed
- [34] DSPACE. Test Automation Software. Retrieved from: [https://www.dspace.com/en/pub/home/products/sw/test\\_automation\\_software/automdesk.cfm](https://www.dspace.com/en/pub/home/products/sw/test_automation_software/automdesk.cfm)
- [35] ETAS Products. Retrieved from: <https://www.etas.com/en/products/mda.php>
- [36] Feldt, R., & Magazinius, A. (2009). Publications. publications

[http://www.robertfeldt.net/publications/feldt\\_2010\\_validity\\_threats\\_in\\_ese\\_initial\\_survey.pdf](http://www.robertfeldt.net/publications/feldt_2010_validity_threats_in_ese_initial_survey.pdf)



**Pedro-Miguel Ortega-Cabezas** has a degree in industrial engineering and a master in electrical, electronic and control engineering from UNED (Spanish University for Distance Education). Nowadays, he is doing his PhD in industrial engineering at UNED.

His field of specialization is focused on powertrains and the validation of the engine control unit software. He has participated in several engine design projects launched by such market leaders as PSA Peugeot Citroën and Renault in their design centers located in France. More specifically, his research is focused on how to use artificial intelligence when validating embedded software such as the engine control unit one



**Antonio Colemanar-Santos** has a PhD in industrial engineering and a master of science in industrial engineering, with a specialisation in electronics and automation engineering awarded both by the School of Industrial Engineering at National Distance Education University (UNED); and a bachelor of science in electrical engineering, with a specialisation in electronic instrumentation, regulation and control and industrial automation awarded by The School of Industrial Engineering at the University of Valladolid.



**David Borge Diez** has a doctoral degree and a master in industrial technologies research from UNED (Spanish University for Distance Education) and a bachelor's degree in industrial engineering with majors in energetic engineering from the University of Valladolid, Spain.

He is a specialist engineer in energy efficiency, energy economics and alternative energy sources and works as a professor in the Department of Electrical, Control and Automation Engineering in the University of León, Spain. He has been associated professor in the University of León and worked for energy companies in both public and private projects, including international R&D programs.



**Jorge Juan Blanes-Peiró** received the engineer degree from the “Universidad Politécnica de Valencia” (Spain) in 1990. In 1995 he obtained the Ph. D. degrees from the “Université Pierre et Marie Curie” (Paris VI-France) and from the “Universidad Politécnica de Valencia” (Spain). Currently he is researcher and teacher of the “Universidad de León” (Spain) and head of the Mining Engineering School of this University.