Chapter 2
Software Validation Techniques in the Automotive Sector

David Borge-Diez, Pedro-Miguel Ortega-Cabezas, Antonio Colmenar-Santos, and Jorge-Juan Blanes-Peiró

Abbreviations

ATCU	Automatic Transmission Control Unit
ADAS	Advanced Driver-Assistance System
dll	Dynamic-link libraries
CAN	Controller Area Network
ECU	Electronic Control Unit
ESP	Electronic Stability Program
EX	Expert system
GA	Genetic Algorithms
HIL	Hardware-in-the-loop
MIL	Model-in-the-loop
SIL	Software-in-the-loop
SM	Software Module

D. Borge-Diez · J.-J. Blanes-Peiró
Department of Electrical and Control Engineering, Universidad de León, Campus de Vegazana, s/n, 24071 León, Spain
e-mail: david.borge@unileon.es; dbord@unileon.es

J.-J. Blanes-Peiró
e-mail: jorge.blanes@unileon.es

P.-M. Ortega-Cabezas · A. Colmenar-Santos (✉)
Departamento de Ingeniería Eléctrica, Electrónica y de Control, UNED, Juan del Rosal, 12, Ciudad Universitaria, 28040 Madrid, Spain
e-mail: acolmenar@ieec.uned.es

P.-M. Ortega-Cabezas
e-mail: pedro-miguel.ortega-cabezas@valeo.com

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

A. Colmenar-Santos et al. (eds.), *Development and Testing of Vehicle Software and its Influence on Sustainable Transport*, https://doi.org/10.1007/978-3-031-47630-3_2

2.1 Introduction

2.1.1 Engine ECU Software

Electronic control units (ECUs) have become essential for the correct operation of a vehicle [1, 2]. 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 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 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 chapter 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.

2.1.2 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 et al. [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, 25]. There are many ways for automating HIL simulation in the market [26, 27]. The automation process is mainly based on black-box techniques such as stated 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 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. Petrenko, Nguena-Timo and Ramesh, reported the main problems and solutions associated with software validation in the automotive sector [28]. 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, Fig. 2.1, 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 NO_x. 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.

The solution for validating no matter what type of 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 2.1 describes the main issues.

The present chapter proposes how to implement the possible solutions depicted in Table 2.1 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

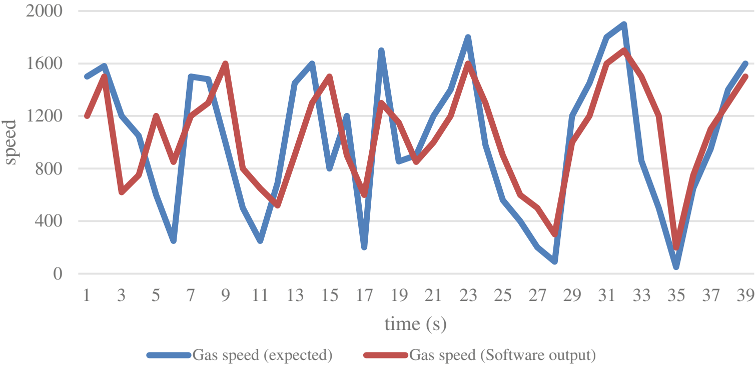


Fig. 2.1 Bug not detected when using traditional techniques

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.

2.2 Application of Rule-Based Expert Systems and Dynamic-Link Libraries to Enhance Hardware-In-The-Loop Simulation Results[1]

2.2.1 Introduction

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 EXs and dlls with the aim of checking if this technique performs better than traditional ones.

[1] Extract of the following paper published in *Journal of Software* “Application of Rule-Based Expert Systems and Dynamic-Link Libraries to Enhance Hardware-In-The-Loop Simulation Results” JSW 2019 Vol.14(6): 265–292. ISSN: 1796-217X. https://doi.org/10.17706/jsw.14.6.265-292. http://www.jsoftware.us/.

Table 2.1 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 So that 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, properly EEPROM initialized, etc.). Consequently, the expected output can be set after having performed the HIL simulation 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, properly EEPROM 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 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 t1, others at t2 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

(continued)

Table 2.1 (continued)

Consequences	Reason	Possible solutions
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 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. 2.1)	Implementing a system that can check if the software properly calculates all software outputs. Dlls can perform this task

To do this, a test-case database was built and run by using HIL simulations to validate a series of 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.

2.2.2 *Method*

2.2.2.1 Description

The engine specifications are composed of Simulink® models. Thus, the dll can be easily built considering that Mathworks® has implemented different ways to build a dll from a Simulink® model [29].

The method used in this research are composed of different stages. Firstly, a series of test-cases are designed. 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 2.1 can be solved thanks to this technique proposed by the authors. 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 Mathworks® 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 study, 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. Figure 2.2 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® [30], 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

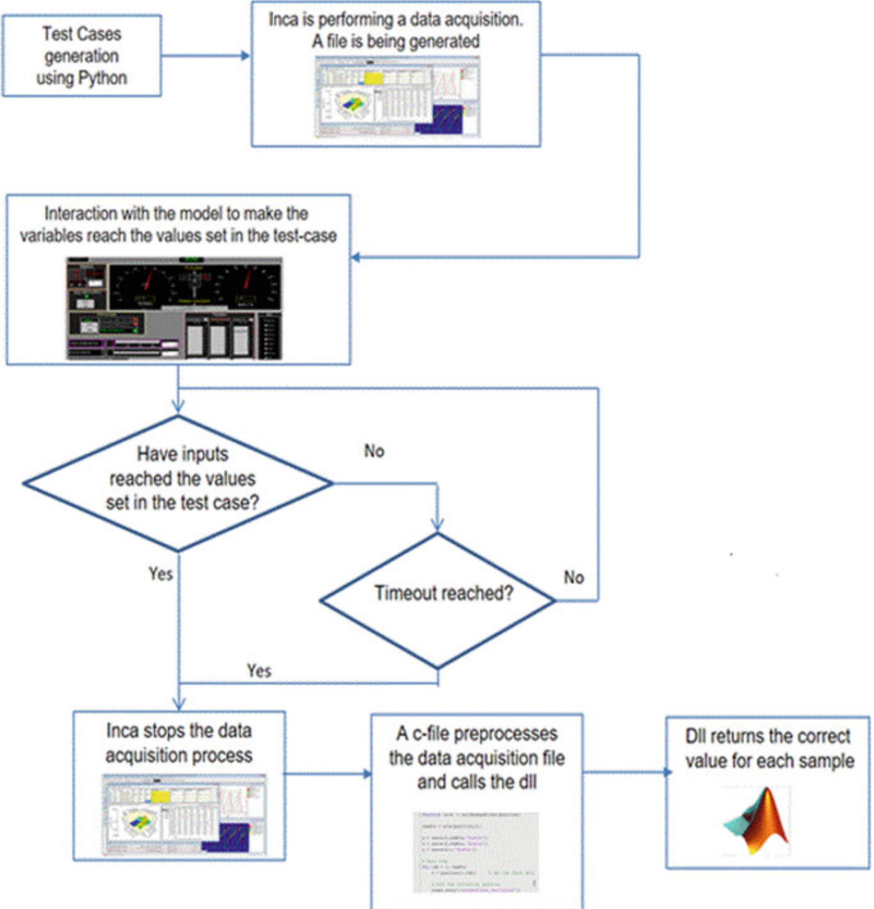


Fig. 2.2 Use of dlls in an HIL simulation when performing a test-case

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 asci-ii file. Secondly, the engine ECU software is an image of the Simulink® models of the SM under validation.

2.2.2.2 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 2.2). They have been considered as representative for this case-study by the authors and the company subjected to this research.

Table 2.2 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 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 For instance, the SM in charge of detecting whether the accelerator pedal is blocked. The engine ECU must check a few parameters	Such as: Temperature estimators 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 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 For instance, SMs related with treatment of exhaust gases	Such as: Treatment of exhaust gases systems
Highly complex	a. High number of input and output variables and number of calculations 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: The SM in charge of controlling the oil rate diluted into diesel

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 2.2).

Table 2.3 shows the number of tests considered in this research according to the type of SM.

Table 2.3 Number of tests used in this research

Type of SM	Number of test
Simple	250
Fairly complex	1,250
Highly complex	100

Table 2.4 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

Table 2.4 indicates the methods followed to generate test-cases for each technique.

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. 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 2.5 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.

2.2.2.3 Equipment

The following equipment was used in this research.

- An engine ECU software and hardware.
- The HIL bench used to conduct this research belongs to the manufacturer dSpace®, model dSpace® Simulator Full-size [31]. It is a versatile HIL simulator capable of emulating the dynamic vehicle behavior.
- When it comes to building the model that serves as the driver's interface, ControlDesk® version 5.1 from dSpace® manufacturer is employed [32]. 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.
- 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

Table 2.5 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 2. Var2_temperature_ exhaust_gas = 600ºC Do a driving cycle and var3_vehicle_speed = 80 km/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	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

use software that allows reading memory locations. In this research, version 7.1.9 of INCA® was used [30].

- The automation process can be carried out in different ways: by using Python script or AutomationDesk® software [33]. 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.
- Matlab® R2013 and Microsoft Visual Studio 2015 were used to create the dlls used in this research.
- Matelo®. Software used for validation purposes being able to generate test-cases.

2.2.3 Results

2.2.3.1 Ease for Automation Test-Cases

a. 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® [33], 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.

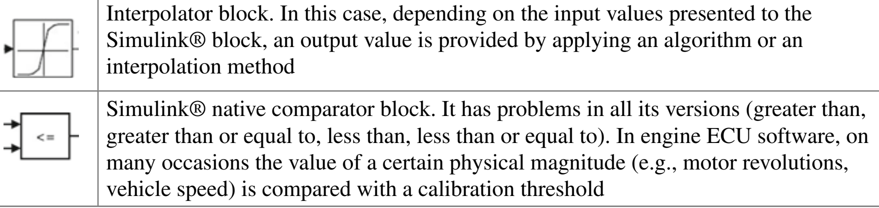
One of the most important issues to be analyzed is the consequences of not reaching the values set in the test-case. Table 2.6 shows the results when validating the simple functions by using different techniques. As one can see, the tester-in-the-loop technique offers better results than the automated 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.

The Simulink® blocks that, in most cases, prevent reaching the values set in the test-case in this research, are show in Table 2.7.

Table 2.6 Comparisons of different techniques for validating simple SMs

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

Table 2.7 Most problematic Simulink blocks



	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
	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

It is important to analyze the root cause of the 5.2% failures. After the analysis of the 13 failures shown in Table 2.6, 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.

b. 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. Figure 2.3 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. 2.3 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 2.8 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 2.9).

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. 2.4). 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 2.8. The results after the 100 HIL simulations are shown in Table 2.10.

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

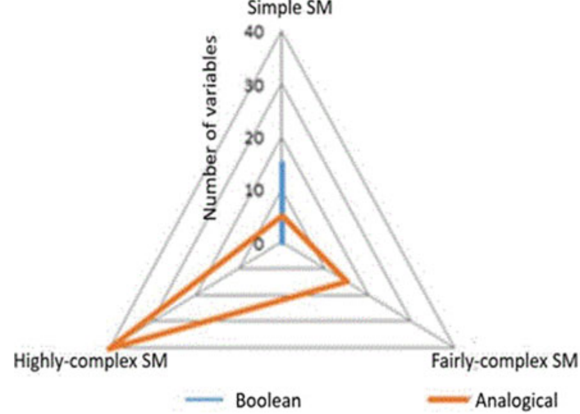


Fig. 2.3 Type of variables present in an average-complexity SM

Table 2.8 Comparisons of different techniques for validating fairly-complex SMs

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

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.

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. Figure 2.5 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

Table 2.9 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

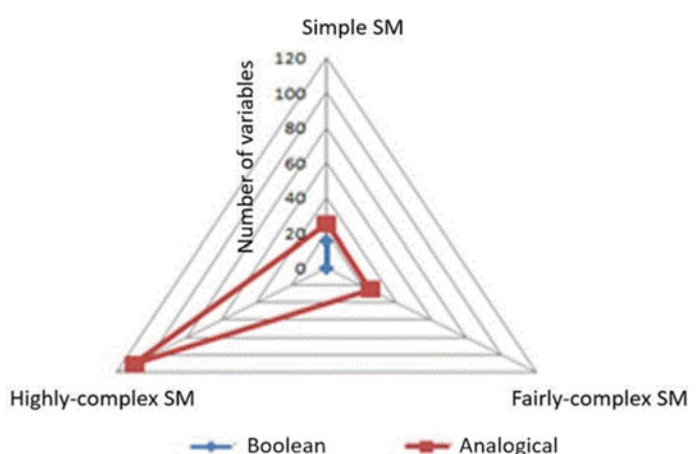


Fig. 2.4 Type of variables present in a high-complex SM

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.

Figure 2.6 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

Table 2.10 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

Fig. 2.5 Error trend depending on error tolerance of the SM inputs

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.

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 2.11). These statements have been proven in this research as shown in Table 2.12.

In this research, the SMs listed in Table 2.10 were used.

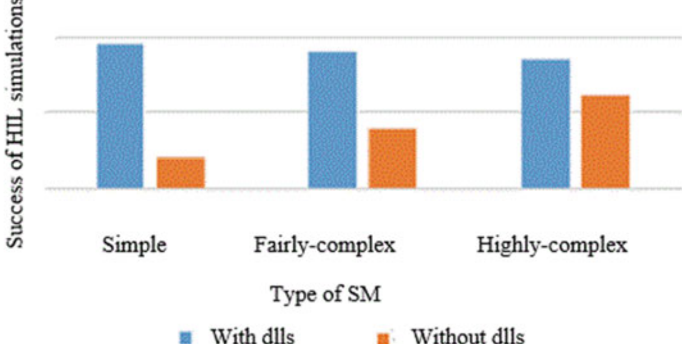


Fig. 2.6 Comparison of results obtained when using and not using dlls

Table 2.11 List of SMs used and tested in this research

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

Table 2.12 Comparisons of different techniques for validating functions depicted in Table 2.10

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 2.11 does not show the results for automation with dlls as most of the function did not have a Simulink® model.

2.2.3.2 Functional Coverage

The functional coverage has been assessed by using Eq. (2.1) which is widely employed in the automotive sector. Table 2.13 shows the total number of functional requirements associated with the SMs validated in this research.

Table 2.13 Number of total functional requirements

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

$$\text{FC} = \frac{\text{number of software requirements tested by a technique}}{\text{number of software requirements indicated in Table 15}} \quad (2.1)$$

Table 2.14 depicts the results obtained for each technique in this research.

a. 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 2.5. 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.

b. 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 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 order words, the test engineer must code a Python script to generate the test -cases

Table 2.14 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

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. Consequently, the results shown in Table 2.9 are the same for the cause-effect technique and the model based-testing one.

c. Performance expert system

The rule-based EX allows specifying the functional requirements of SMs. Two phases are considered when validating EXs: 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 2.15). Table 2.16 shows the results obtained during the first phase in which a 83.3% success rate was obtained.

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.

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 2.17. 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 2.15 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 2.16 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

Table 2.17 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.
3. Dlls allow finding bugs linked to calculation errors.

2.2.3.3 Productivity Gain

It is essential to check if EXs implementation respects the timeframes of the project by analyzing several factors. As shown in Table 2.18, 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.

Table 2.18 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)[a]	0.00028	0.00347	0.00044
	Time for validating the results without automation (h)	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

[a]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

2.2.3.4 Bug Detection

Figure 2.7 shows the bugs found by each technique when running the test-cases. 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.

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 2.19).
3. When the bug is linked to calculation errors (calculation faults).

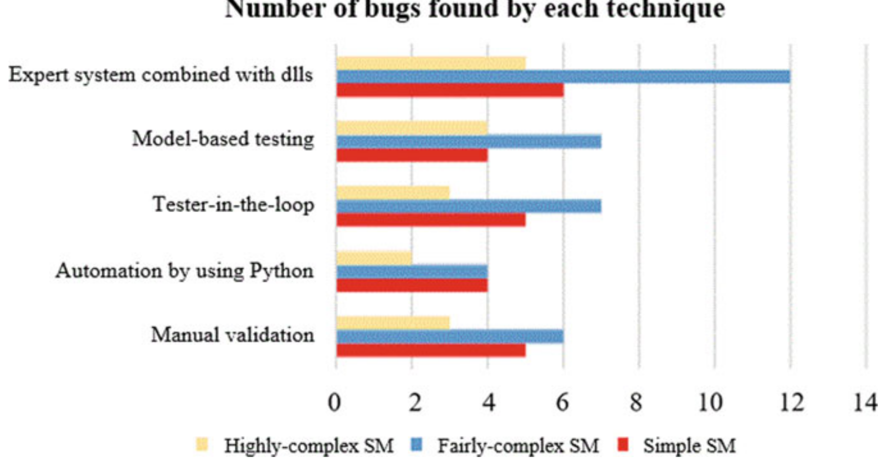


Fig. 2.7 Bugs found when using different techniques

Table 2.19 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

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.

2.2.3.5 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. 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.

2.2.3.6 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. Figure 2.8 shows an example of a histogram

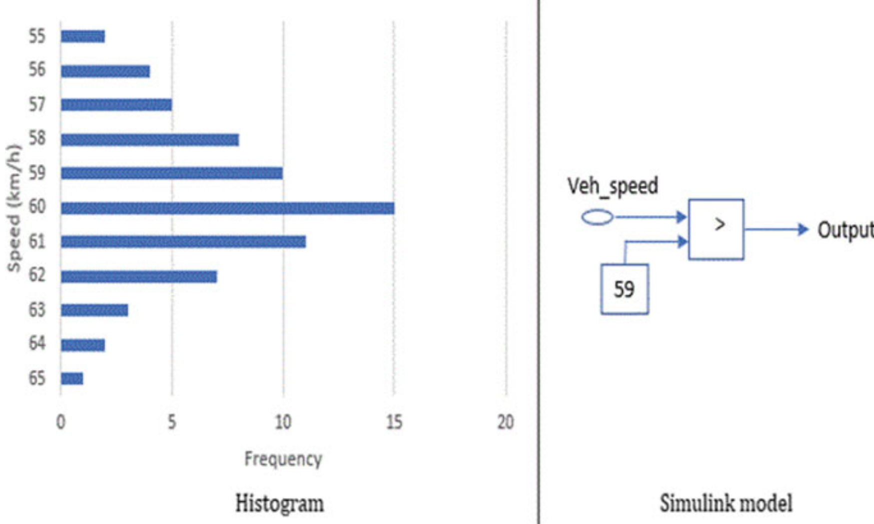


Fig. 2.8 Example of test-case

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.

A comparison among different techniques is shown in Table 2.20.

2.2.4 Conclusions

This research, conducted at the second most important European car manufacturer, is focused on the software validation of an engine ECU by using 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

Table 2.20 Comparison among different techniques

<table>
<tr><th></th><th>Manual validation</th><th>Automation without dll</th><th>Model-based testing</th><th>EXs and dlls</th></tr>
<tr><td>Validity of test-cases</td><td colspan="3">As shown in Fig. 2.9, it is necessary to reach the exact value indicated in the test-case. Otherwise, the validation process cannot be performed automatically</td><td>Even though the values indicated in the test-case are not reached, the validation can be performed automatically</td></tr>
<tr><td>Accuracy needed</td><td colspan="3">The test-case output may be no longer valid (Fig. 2.9). The test-engineer should check the specification to confirm the expected output</td><td>The test-case output may be no longer valid. However, dlls can check the expected output automatically</td></tr>
<tr><td>Complexity</td><td colspan="3">As shown in Fig. 2.9, it is highly complicated to reach the specific values indicated in the test-cases (see Tables 2.6, 2.8 and 2.10) due to SM interactions</td><td>Even though the HIL simulation does not reach the specific values indicated in the test-case, the validation process can be performed</td></tr>
<tr><td>Robustness in case of failure</td><td colspan="3">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</td><td>Even though the engine ECU detects a failure, the dll can detect the expected output in that case</td></tr>
<tr><td>Reading ECU variables in real-time</td><td colspan="3">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</td><td>The dll can do the validation process automatically when using a C-code at the same time</td></tr>
</table>

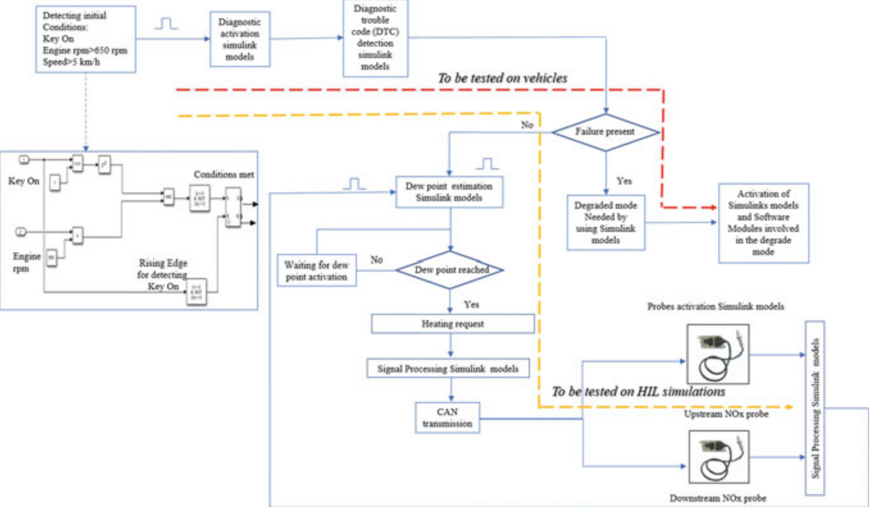


Fig. 2.9 Example of model and activation conditions

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.

2.3 Use of Genetic Algorithms to Reduce Costs of the Software Validation Process[2]

2.3.1 Introduction

The number of ECUs installed in vehicles is increasingly high. Manufacturers must improve the software quality. Innovative techniques must be proposed to reduce cost and increase software quality.

This research proposes a technique being able to generate not only test-cases in real time but to decide the best means to run them (HIL simulations or prototype vehicles) to reduce the cost and software testing time. It is focused on the engine ECU software which is one of the most complex software installed in vehicles. This software is coded by using Simulink® models. Two genetic algorithms (GAs) were coded. The first one is in charge of choosing which parts of the Simulink® models should be validated by using HIL simulations and which ones by using prototype vehicles. The second one tunes the inputs of the SM under validation to cover these parts of the Simulink® models. The usage of dlls is described to deal with the issues linked to SM interactions when running HIL simulations.

GAs found at least 7 more bugs than traditional techniques and improved the functional and code coverage by between 3% and 11% for functional coverage and by between 1.4% and 7% for code coverage depending on the SM complexity. The validation time is reduced by 11.9% regarding traditional techniques. GAs perform better than traditional techniques improving software quality and reducing costs and validation time. The usage of dlls allows testing the software in real time as described in this study.

Both the number of ECUs installed in vehicles and their complexity are increasing [5, 34, 35]. Thus, manufacturers must assure software quality and reliability [12]. The software and hardware validation of an engine ECU is performed by using the HIL simulation and prototype vehicles [36]. The HIL simulation has several advantages as no vehicle with all ECUs updated with the latest software version is necessary. Secondly, the ECU behavior in the network can be checked by analyzing the frames transmitted and received when conducting an HIL simulation. However, the real interactions between ECUs are not tested as all frames received are sent by a model and not by real ECUs. Regarding prototype vehicles, the engine ECU software is tested in real vehicles which must have all ECUs properly updated: ESP

[2] Extracted from Ortega-Cabezas, P.M., Colmenar-Santos, A., Borge-Diez, D. et al. Experience report on the application of genetic algorithms to reduce costs of the software validation process in the automotive sector during an engine control unit project. *Software Quality Journal* 30, 687–728 (2022). https://doi.org/10.1007/s11219-021-09582-x. https://www.springer.com/journal/11219.

(Electronic Stability Program), ADAS (Advanced Driver-Assistance System ECU), ATCU (Automatic Transmission Control Unit), etc.

This chapter is focused on one of the most complex software installed in vehicles: the engine ECU software. It proposes the usage of GAs aiming at choosing the most adequate means to be used for validation while generating test-cases automatically at the same time. The main goals are:

a. Choosing automatically the optimal means to reduce the validation time and costs.
b. Finding solutions to technical problems when using the HIL simulation due to SM interactions.
c. Assessing whether GAs perform better than other techniques such as the model-based testing and the black-box techniques.
d. Verifying whether GAs are able to find bugs when other techniques fail.
e. Assessing the staff skill impact on the validation process.

The engine ECU software development comprises three phases (V-cycle development): implementing models based on Simulink® software in order to control the engine performance, generating C-code and checking the final integration of the software into the hardware. During the whole process, the engine software completes three levels of testing: model-in-the-loop (MIL), software-in-the-loop (SIL) and HIL simulations [37]. Consequently, the software is tested to assure that it meets all requirements. During the MIL, a controller model is implemented and applied to the Simulink® model aiming at checking if the model behaves as expected [38–40]. During the HIL simulation, the integration between software and hardware is tested thanks to a controller (the engine ECU and its software) which controls the system that imitates the engine behavior (the HIL simulator) [41–45]. In addition, prototype vehicles are used to test some functions which cannot be completely validated when using HIL simulations such as ADAS [46]. Therefore, the most adequate means to validate the software must be chosen to reduce time and costs. Finally, SIL is employed to test an executable code within a modelling environment [47].

Currently, software is tested based on software, architecture and system requirements [37]. At this point, how to test software requirements is a key point discussed in some standards such as ASPICE (2020). Software testability depends on 5 factors such as: requirements, built-in test capabilities, the test-cases design, the test support environment, and the software process in which testing is conducted [48]. Regarding software requirements, the most significant cause of accidents due to software is linked to poorly created software requirements or requirements that are partially delivered to developers [49, 50]. Dos Santos et al. carried out a detailed analysis about software requirements testing approaches such as the requirement driven testing [51, 52].

Concerning autonomous driving, ISO 26262 only covers functional safety when a failure occurs but not when there is no system failure. That is why, the safety of the intended functionality (SOTIF) ISO 21448 came out [53, 54]. Some key topics to validate the software are focused on 3D Modeling and sensor buildings. The former aims to create a realistic environment while the latter consists of modeling

and testing sensors among others [55]. Huang et al. detail in their research the main tendencies to validate software such as software testing, simulation testing, x-in-the-loop testing and driving test in real conditions [56]. Riedmaier et al. describe an important method to test the software: the scenario-based approach which allows individual traffic situations to be tested by using virtual simulations [57]. Other approaches such as formal verification, a function-based approach, real-world testing, shadow mode testing and traffic-simulation-based approach are used to test SOTIF. The main difference among them is that in the scenario-based and function-based approaches, a microscopic statement about the safety of the system is first made to be transferred to a macroscopic statement. The rest of the methods result directly in a macroscopic statement. There are solutions in the market which allow rapid prototype, MIL/SIL simulations, HIL simulations and real test drives [58].

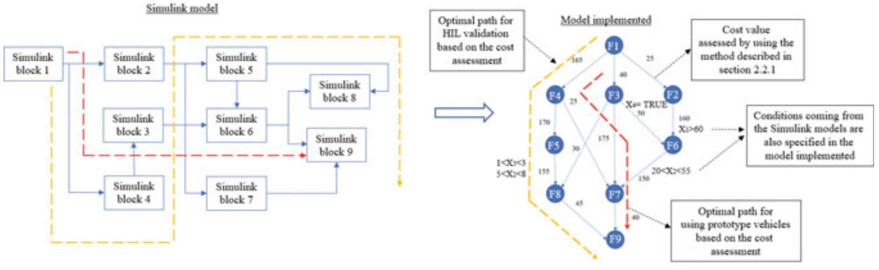
Cybersecurity in the automotive industry involves three main factors to be considered such as authentication and access control, protection from external attacks and detection and incident response [59]. The factors which make the automotive security more efficient include integration of right solutions such as firewalls, protecting communications, authenticating communications and encrypting data [60, 61]. These topics are important to offer performance such as on-the-air software update and V2X communication [62, 63]. As detailed by McAfee, the scope of cybersecurity involves the distributed security architecture, hardware and software security and finally network security [62]. Standards such as ISO/SAE 21434 will help the automotive sector to implement solutions for effective compliance with cybersecurity requirements as today's knowledge sharing is inadequate [64, 63] In this research, some topics linked to cybersecurity testing are analyzed.

2.3.2 Methods

2.3.2.1 Simulink Models

The SMs are composed of multiple complex Simulink® models and subsystems. Figure 2.9 shows an example of the internal structure of the SM linked to the NO_x heating probes installed in vehicles. When the initial conditions are reached (key on, the engine rpm more than 650 rpm and the vehicle speed higher than a threshold), the engine ECU software checks whether the dew point is reached. This point is the temperature to which air must be cooled to become saturated with water vapor. Afterwards, the NO_x probes start to heat until reaching the required temperature to measure NO_x ppm present in the exhaust gas pipe. In this study, all models were transformed into models based on nodes (S_x) which represent different low-level system states[3] and relations between them (Fig. 2.10). When designing test-cases, it must be determined which parts of the Simulink® model should be validated by

[3] Low system states are functional states at low level. Consequently, the functional state cannot be detected by the driver.



Optimal path for HIL validation (in orange): a test case that covers blocks 1, 4, 5, 8 and 9
Optimal path for prototype vehicles (in red): a test case that covers blocks 1, 4, 6 and 9

Fig. 2.10 Example of model and activation conditions

using the HIL simulation or prototype vehicles and how inputs are tuned. Next section describes in-detail how GAs work to do this.

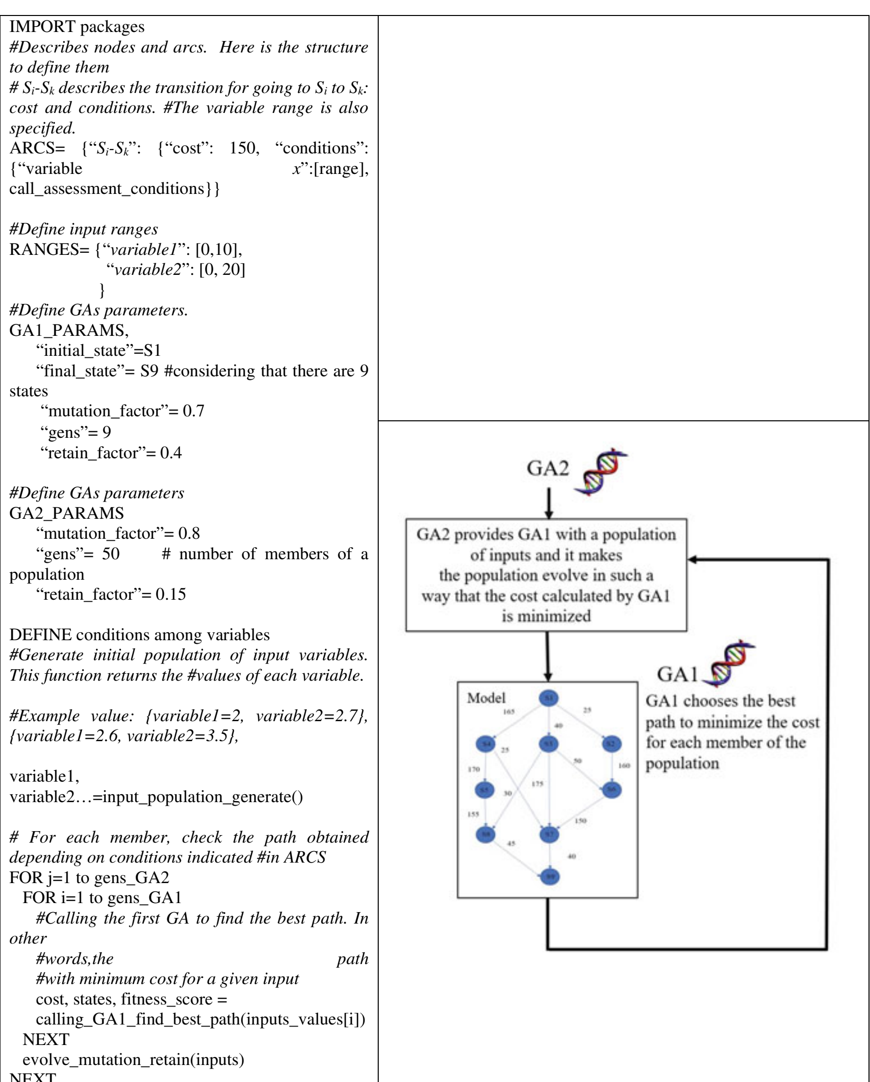
2.3.2.2 How GAs Work Together

Figure 2.11 depicts a pseudocode and a high-level description of the method. A model is implemented by using the Python code (Fig. 2.11) through the variable named *ARCS* which contains the cost and conditions to go from one state to another one. Next sections display how the conditions are specified. Once the model is made, the GAs are parametrized, and the range values of the input variables of the SM under validation and the constrained linked to the optimization problem are specified. The GA2 generates populations (inputs for the SM under validation). GA1 is used to assess and optimize the path with the lowest cost by doing operations such as mutation or crossover taking into account the population generated by GA2. The GA2 makes the population evolve in such a way that the cost calculated by GA1 is minimized.

2.3.2.3 GAs Description

a. GA in charge of tuning inputs

Once the model is implemented and set in the code, this generates populations. To do this, the range must be specified as well as the constrained among software variables linked to the optimization problem. The fitness function of this GA2 is the output of GA1 described later which was responsible for finding an optimal path given specific inputs. When the GAs are run, the results display the values of all inputs of the SM which cover the path requiring the minimum cost and the usage of HIL simulations.



```
IMPORT packages
#Describes nodes and arcs.  Here is the structure to define them
# Si-Sk describes the transition for going to Si to Sk: cost and conditions. #The variable range is also specified.
ARCS= {"Si-Sk": {"cost": 150, "conditions": {"variable x":[range], call_assessment_conditions}}

#Define input ranges
RANGES= {"variable1": [0,10],
         "variable2": [0, 20]
        }
#Define GAs parameters.
GA1_PARAMS,
    "initial_state"=S1
    "final_state"= S9 #considering that there are 9 states
     "mutation_factor"= 0.7
     "gens"= 9
     "retain_factor"= 0.4

#Define GAs parameters
GA2_PARAMS
    "mutation_factor"= 0.8
    "gens"= 50       # number of members of a population
    "retain_factor"= 0.15

DEFINE conditions among variables
#Generate initial population of input variables. This function returns the #values of each variable.

#Example value: {variable1=2, variable2=2.7}, {variable1=2.6, variable2=3.5},

variable1,
variable2…=input_population_generate()

# For each member, check the path obtained depending on conditions indicated #in ARCS
FOR j=1 to gens_GA2
  FOR i=1 to gens_GA1
    #Calling the first GA to find the best path. In other
    #words,the path
    #with minimum cost for a given input
    cost, states, fitness_score =
    calling_GA1_find_best_path(inputs_values[i])
  NEXT
  evolve_mutation_retain(inputs)
NEXT
```


Fig. 2.11 Pseudocode of how GAs work together

b. GA in charge of choosing the most adequate means

Several key factors must be considered when deciding the most adequate means to validate the software such as the ones shown in Table 2.21 [42, 65].

All factors shown in Table 2.21 are assessed when going among states of the models (see Fig. 2.10). This process is composed of two phases:

Table 2.21 Factors considered to assess the fitness function

Factor	Description
Tuning activities	Some SMs must be tuned before its validation such as combustion/injection SMs. In this case, the engine software can apply different cartographies to inject the optimal amount of diesel or gasoline. If one of them is not tuned, the engine may stall. Consequently, a dataset which guarantees a minimum functionality of the SM under validation must be available
Time needed to go from one state to another one	An estimate of the time needed to validate an SM when using a vehicle or an HIL model is made. There are two possibilities—either to perform the simulation by using an HIL model or a vehicle. The former implies that the HIL model must be robust. The latter implies that a vehicle should be used. Some use-cases are difficult to reach when using prototype vehicles. Test-engineers' experience is essential to assess properly this factor
Dependency on ECUs	When validating a certain SM by using a vehicle, all ECUs must be updated aiming at assuring that all frames are properly transmitted and received among other factors. Otherwise, the validation process such as the adaptive cruise control SM cannot be performed. In this case, at least the ADAS, ESP and engine ECU must be properly updated and tuned
Risk level	The automotive safety integrity level is a system which classifies potential risks posed in the vehicle when it is operated by using the ISO 26262. For this purpose, it uses three parameters such as: exposure, controllability and severity with the aim of establishing a score. By using this score, a series of automotive safety integrity level is established. Regarding the engine ECU, the software must guarantee the passengers' and vehicle safety in a dangerous situation. Depending on the ASIL values (A, B, C, D) the level of risk will be different
Feedback from other projects	It is common that several engine projects take place at the same time. Consequently, feedback from other projects is of paramount importance. Therefore, when a bug is found in a specific engine software version, it is immediately communicated to other project teams so that they can check if there is a bug in some other engine software applications. Meanwhile client complaints are also considered in such a way that if a project receives a client complaint, it is transmitted to other projects, which could galvanize all necessary actions

- *Phase 1*. A multidisciplinary team assesses these factors aiming at determining the cost of each path by using the process depicted in Fig. 2.12. As a result, a model with the whole cost set for each transition is obtained (Fig. 2.10).
- *Phase 2*. This GA chooses the most adequate means to validate the SM by assessing the cost function given by Eq. (2.2):

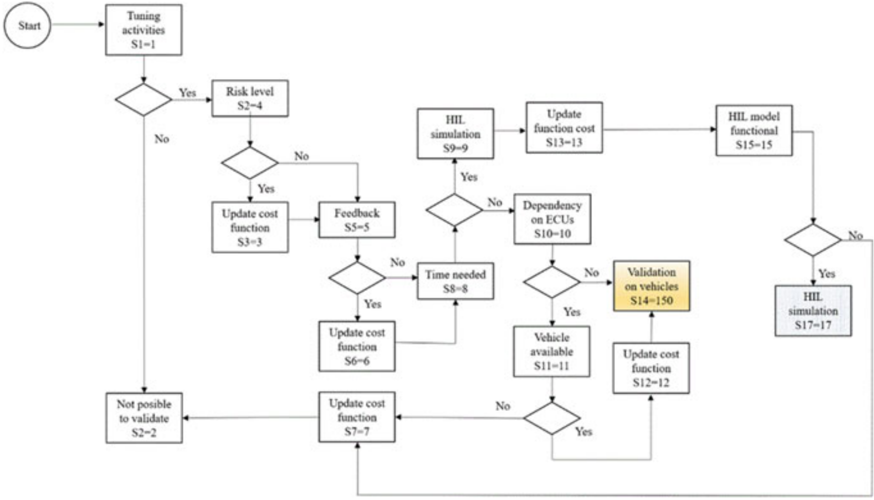


Fig. 2.12 Factors indicated in Eq. (2.1)

$$\text{Fitness function} = \sum_{i=1}^{i=n} S_i \tag{2.2}$$

where S_i is the cost of reaching the S_i state, $\sum_{i=1}^{i=n} S_i$ is the cost linked to all transitions of a specific path. When the HIL is chosen, the fitness function is always lower than 150. Otherwise, prototype vehicles are employed as, in this case, the fitness function reaches 150 or more. The reader can check this by adding all S_i values needed to reach S_{14}.

Each path is composed of different states. The paths which contain state 17 more frequently are considered as the optimal ones to be validated by using an HIL simulation. Otherwise, it should be validated by using prototype vehicles. The test-engineer can collect important information when analyzing the states covered once the optimal path is assessed (dependency on other ECUs, feedback from other projects, etc.).

2.3.2.4 HIL Simulations

Once the GAs are parametrized and a model is built as shown in Fig. 2.10, the HIL simulation can be conducted. In addition to the cost value, the actions to be conducted on the HIL model for each transition must be coded (Fig. 2.13) as the software variables have to reach the values specified in the test-case. Several ways to set the conditions to pass from one state to another one can be used. The first entails writing the equations directly in the code, which is limited to simple SMs as fairly complex and high complex SMs involve many equations. The second option is to call the

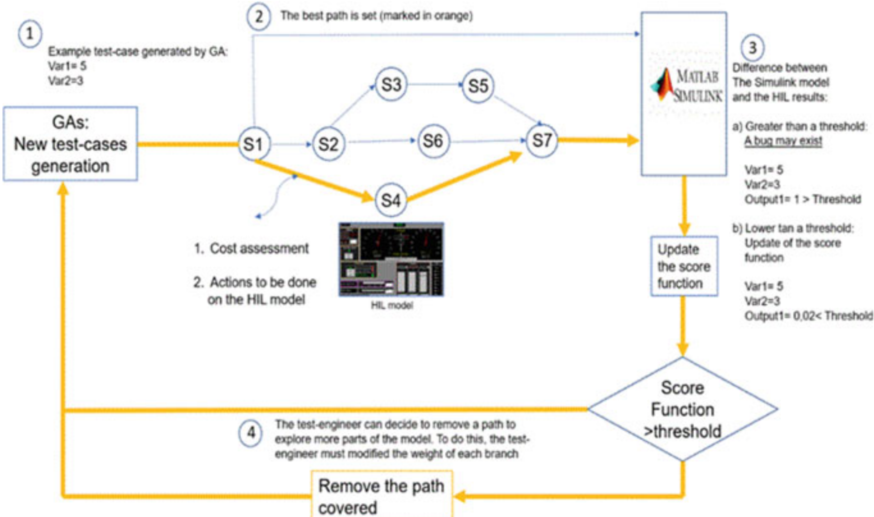


Fig. 2.13 HIL simulation process

Simulink® model by using the test-case inputs to make the Simulink® model return the expected output values. In this study, the Simulink® models were transformed into dlls by following the steps described in the official Matlab® documentation. Figure 2.13 depicts the usage of dlls. They are necessary to conduct the validation process to find bugs due to SM interactions as it will be shown in the results and discussion sections.

2.3.2.5 Network and Software and Hardware Integration

This proposal validates the network and hardware and software integration by using the dlls as shown in Fig. 2.13. Once the software is coded, the software outputs must be equal or very close (if the outputs are analogical) to the values provided by the Simulink® models despite the SM interactions. This point is checked by using the dlls which allow comparing the HIL results when running a test-case with the outputs provided by dlls. The same explanation can be used for vehicles as the data acquisition can be injected into the Simulink® models, and both results can be compared.

Regarding the network, it is tested when using prototype vehicles in real conditions. Not all SMs implemented in the software exchange information with other ECUs. All these aspects are considered in Fig. 2.12 where the reader can find state S_{10} which assesses if the SM under validation has an impact on the network. Anyway, if an SM must be validated and prototype vehicles with all ECUs properly updated are not available (specially at the beginning of the project), HIL simulations are used

considering that the frames are simulated by using a model (this situation is also considered in Fig. 2.12).

2.3.2.6 Traditional Techniques

The hereafter techniques were used in this research.

a. The cause-effect technique

One of the most used techniques in the automotive sector is the black-box technique [18]. The main idea behind this technique is to test software as a black box. In other words, the internal structure of the SM is not considered by the test-engineer who is focused on the software behavior. That is why this technique is also known as behavioral testing. When designing the test to be run, test engineers design test-cases and decide which means could be used according to their experience [18, 66, 67]. The cause-effect is a black-box technique widely employed in the automotive sector for several reasons (easy to automate among others). This technique is based on considering a series of conditions linked to inputs of the SM under validation, the test-engineer must check if the software runs as expected. To do this, the test-engineer performs a series of actions by using the means employed for validation (prototype vehicles or the HIL simulation) and, finally, verify the software behavior. This behavior is validated and assessed by using the outputs of the SM under validation. It must be reminded that the means used to validate it are chosen by considering the test-engineers' experience when using this technique.

b. The model-based testing

It is a software testing technique consisting of deriving test-cases from a functional model which describes the functional aspects and requirements of the SM under validation. Thanks to this model, it is easier to assess the functional coverage as the number of functional states covered when validating an SM is known. When implementing it, all functional states and the transition from one state to another are indicated. In this research, Matelo® software was used to generate the functional model of SMs [68]. This software allows implementing a model easily. Regarding the activation of each transition, the conditions are set. In this study, each transition calls a Simulink® model to check the next state to be activated. Matelo® allows generating test-cases by assigning values to all variables used in transition in such a way that it tries to cover all possible transitions and paths. Finally, each state can be a model as it is the case in this research making the models extremely complex. Figure 2.14 sums up all aforementioned process explained. A test-case is generated and by using calls to Simulink® models, Matelo® determines which part of the model will be covered (Fig. 2.14 in orange). Many test-cases are generated to cover the whole model and to increase functional and code coverage.

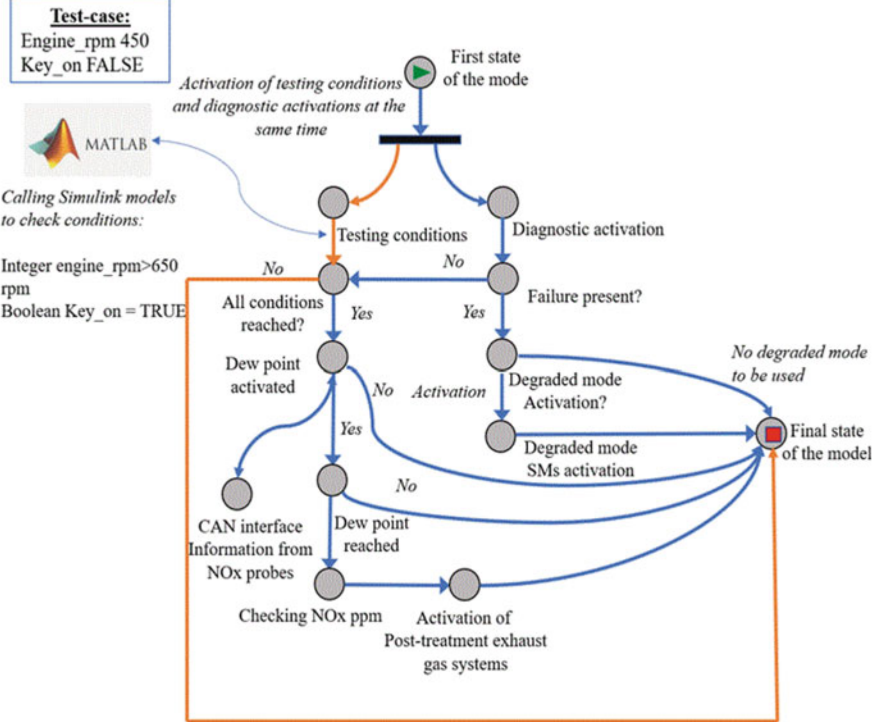


Fig. 2.14 Example of NO_x activation model based on Matelo®

2.3.2.7 Experimental Settings

The characteristics of SMs have an impact on three factors: the time needed to validate the software, the means used to run test-cases and the number of test-cases to be run considering the planning of the engine software development. According to the test-engineers' experience and the technical documentation used for coding the software, the SMs were classified as simple, fairly complex and high complex SMs (Table 2.2).

Table 2.22 shows the way of generating test-cases. All techniques used the software and system requirements traced in DOORs, feedback from other projects[4] and the Simulink® specifications as inputs. By analyzing all these input data, the test-engineers build models when using GAs and the model-based testing. Finally, test-cases are implemented automatically or manually. As described later, the test-engineers' skills have a significant impact on the time needed to implement test-cases and to obtain a productivity gain.

[4] Feedback from other projects means bugs found in a project which could impact another project.

Table 2.22 Test-cases run in this research

Technique	Inputs used for implementing test-cases	Software used	Way of implementing test-cases	Model used
Cause-effect technique	1. Feedback from other projects 2. Software requirements 3. System requirements 4. Simulink® specifications	1. DOORs 2. Corporate database to trace bugs 3. Excel® file which contains all information needed (initial conditions, actions to be done, etc.)	Manually by interpreting: a. the software and system requirements b. the information of bugs traced in the corporate database	None
Model-based testing	1. Feedback from other projects 2. Software requirements 3. System requirements 4. Simulink® specifications	1. Matelo® 2. DOORs 3. Corporate database to trace bugs	Automatically done by Matelo® by covering the model built by the test-engineer	Functional model
Genetic Algorithms	1. Feedback from other projects 2. Software requirements 3. System requirements 4. Simulink® specifications	Pseudorandom values generated by Python when coding GAs	Automatically done by genetic algorithms	Low level model

2.3.3 *Results*

This section compares the performance among GAs and traditional techniques by using the KPIs indicated in Table 2.23.

2.3.3.1 Code Coverage

During HIL simulations, a bug is detected if the difference between the HIL results and the outputs provided by the Simulink® models does not obey Eq. (2.3).

$$\sum_{j=1}^{j=m} \left| \mathrm{HIL}_j - \mathrm{Simulink}_j \right| \approx 0 \tag{2.3}$$

Table 2.23 KPI employed in this research

KPI	Description
Code coverage	It determines the number of Simulink® blocks successfully validated when running test-cases divided by the total number of Simulink® blocks considered
Functional coverage	It determines the number of functional states successfully tested when running test-cases divided by the total number of functional states considered
Validation software time	It describes the time needed to implement, run and validate an SM when running test-cases
Productivity gain	The time gain obtained when using a specific software validation technique
Bugs found and their types	Number of bugs and types found when using a specific software validation technique
Bugs found by other clients	Number of bugs found by other users of the engine ECU software such as ESP and ADAS validation staff

where HIL_j is the value for the output j of the SM under validation after having run a test-case by using an HIL simulation and $\mathrm{Simulink}_j$ is the value for the output j of the SM under validation after having run a test-case by using the Simulink® model.

The coverage is assessed by using Eq. (2.4) which relates to the number of Simulink® blocks tested versus the total number of blocks presented in the specifications of the SM under validation.

$$\text{Code coverage} = \frac{\text{number of Simulink}^{\circledR}\text{ blocks tested}}{\text{number of Simulink}^{\circledR}\text{ blocks present in the SM under validation}} \times 100 \tag{2.4}$$

Table 2.24 shows the number of blocks present in the SMs validated in this research, which is used to assess the code coverage (Table 2.25).

As the cause-effect technique does not use models, the code coverage is lower than the one obtained when using the model-based testing and GAs. Building a model in which each state is a Simulink® block allows testing the same functional state by following different branches of the Simulink® model (Fig. 2.15). The model-based testing does not allow tuning the inputs of the SM with the aim of choosing the best means (an HIL simulation or vehicles) to validate an SM contrary to GAs. In addition, this technique needs to define test-cases as inputs and expected outputs. In case of a problem with the automation process due to SM interactions, the expected outputs could be no longer valid. This problem is solved by GAs and dlls. Regarding GAs, the code coverage is the addition of the code coverage when using HIL simulation and

Table 2.24 Number of total Simulink® blocks

Type of SM	Number of Simulink® blocks
Simple	80
Fairly complex	350
High complex	530

Table 2.25 Code coverage obtained when validating the 15 SMs

Technique	Simple SM		Fairly complex SM		High complex SM	
	Number of Simulink® blocks	Code coverage (%)	Number of Simulink® blocks	Code coverage (%)	Number of Simulink® blocks	Code coverage (%)
Cause-effect	63	78.7	265	75.7	380	71.7
Model-based testing	68	85	285	81.4	410	77.3
GAs when using an HIL simulation	58	92.5	235	88.5	412	78.7
GAs when using prototype vehicles	16		75		5	

prototype vehicles. GAs perform better as they can cover more Simulink® blocks providing that the right means are used.

Code coverage should be at least 90% to meet standards. The validation process of an engine ECU is the combination of the software validation performed by the validation team (topic considered in this research), the tuning activities and the driving tests which consist of making 6 vehicles cover 20,000 km each to test the software in real conditions. The total code and functional coverage are assessed considering these three activities. No technique can reach 100% coverage due to several reasons such as project planning constraints. As proved later, validating by choosing the wrong means increases the validation time.

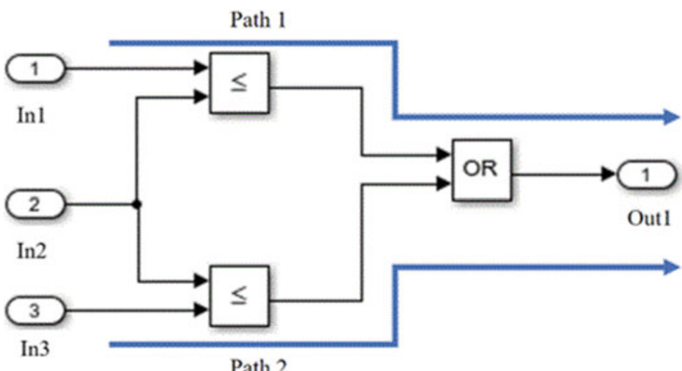


Fig. 2.15 Example of different ways of activating an output

Table 2.26 Number of total functional requirements

Type of SM	Number of requirements	Number of Simulink® blocks
Simple	75	80
Fairly complex	400	350
High complex	510	530

Table 2.27 Functional coverage obtained for each research

Technique	Simple SM		Fairly complex SM		High complex SM	
	Number of requirements tested	Functional coverage (%)	Number of requirements tested	Functional coverage (%)	Number of requirements tested	Functional coverage (%)
Cause-effect	60	80	302	75.5	357	70
Model-based testing	65	86.6	330	82.5	385	75.4
GAs	69	92	346	86.5	400	78.4

2.3.3.2 Functional Coverage

Table 2.26 shows the functional states linked to the Simulink® blocks present in the SM chosen. The number of functional states can be lower than the number of Simulink® blocks as some outputs of the SM can be activated by using several paths without any impacts on the functional state of the vehicle (Fig. 2.15).

Table 2.27 shows the results obtained for each technique. These results are logical as the higher the code coverage is, the higher the functional coverage is. The standard percentage of validation (90%) is reached thanks to tuning, validation and test-driving activities.

2.3.3.3 Automation

For several reasons, the automation process is difficult to be performed when it comes to engine ECU software due to SM interactions. Firstly, reaching the values for inputs of the SM under validation is difficult as the complexity of the SM increases. Secondly, if inputs do not reach the expected values, the values of the outputs of the SMs under validation will be no longer valid (Fig. 2.16).

Test-cases can be fully automated, partially automated or can be run manually. In this research, GAs were run by using the tester-in-the-loop and fully automated options. The success rate of reaching the values indicated in the test-case is shown in Fig. 2.17.

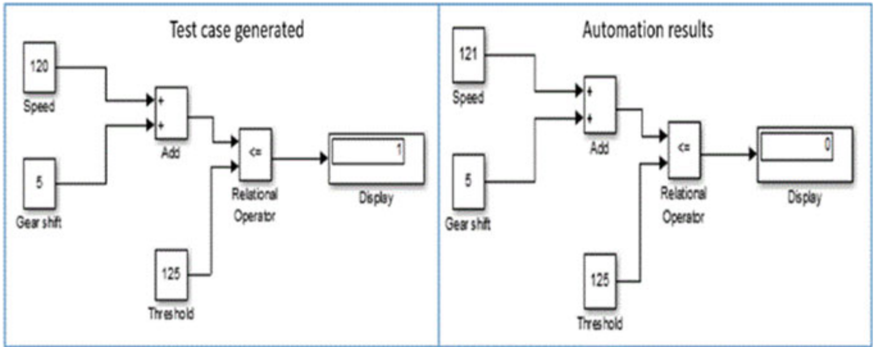


Fig. 2.16 Potential error when a test-case is automated

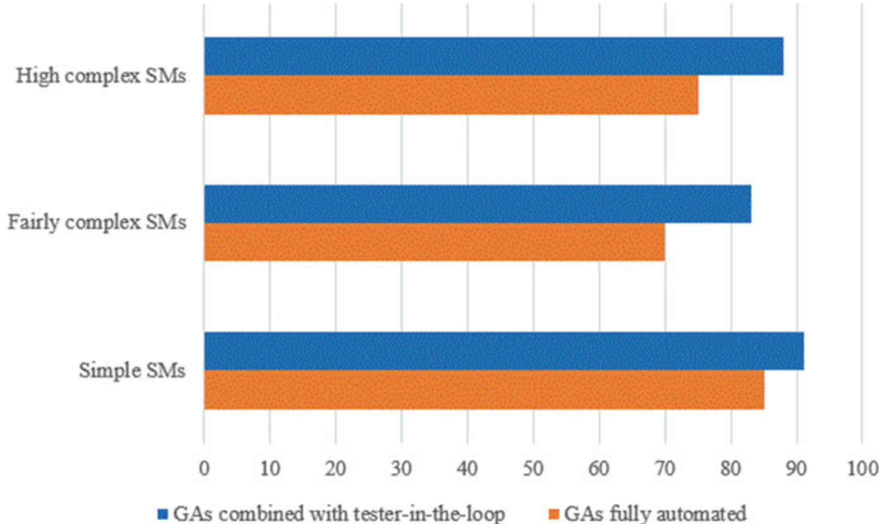


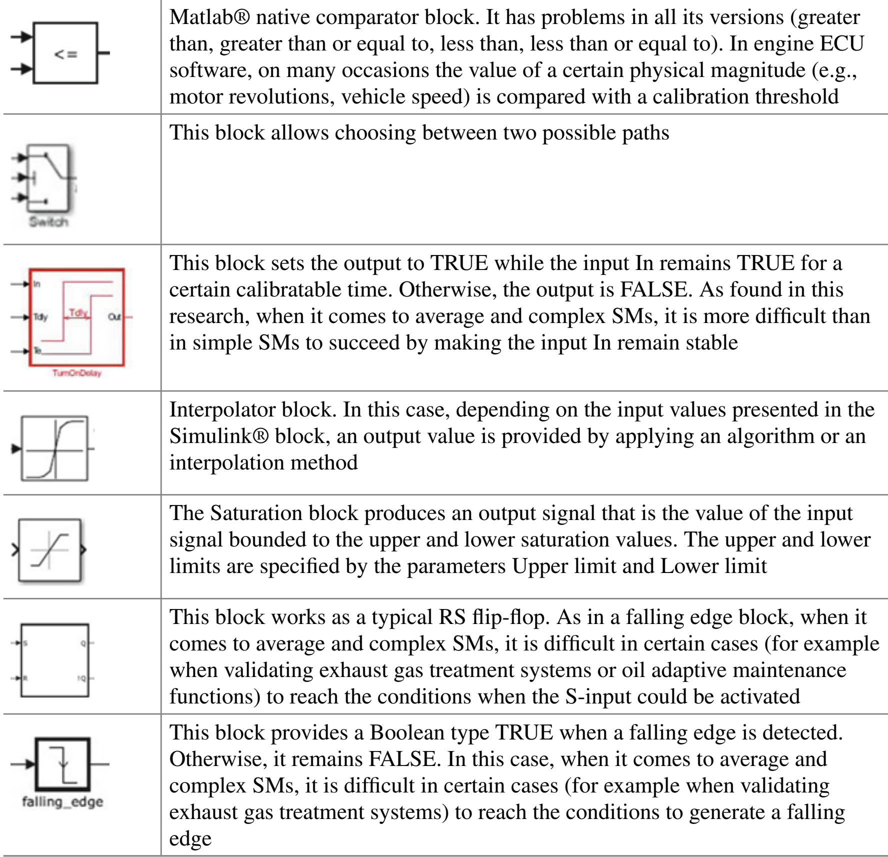
Fig. 2.17 Success rate when automating the HIL simulation

2.3.3.4 Bugs

Types of Bugs

Generally, all techniques detect the same types of bugs linked to Simulink® blocks. Some examples of Simulink® blocks where a bug was found are shown in Table 2.28 and Fig. 2.18. Some types of bugs linked to multiple calculations such as temperature or gas speed estimators can only be detected when using HIL simulations combined with dlls. Figure 2.19 depicts the obtained result for a software variable output of an

Table 2.28 Types of bugs found



<=	Matlab® 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
Switch	This block allows choosing between two possible paths
TurnOnDelay	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 SMs, it is more difficult than in simple SMs to succeed by making the input In remain stable
	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 an interpolation method
	The Saturation block produces an output signal that is the value of the input signal bounded to the upper and lower saturation values. The upper and lower limits are specified by the parameters Upper limit and Lower limit
	This block works as a typical RS flip-flop. As in a falling edge block, when it comes to average and complex SMs, it is difficult in certain cases (for example when validating exhaust gas treatment systems or oil adaptive maintenance functions) to reach the conditions when the S-input could be activated
falling_edge	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 SMs, it is difficult in certain cases (for example when validating exhaust gas treatment systems) to reach the conditions to generate a falling edge

SM when running the software by using an HIL simulation (in red) and its expected value (in blue). The error between the red and blue lines, represents an inaccuracy regarding the calculation of the gas speed in the exhaust pipe, which impacts the amount of urea injected to treat NO_x. Since this bug does not imply the presence of a functional bug unless it causes a malfunction detected by the driver, it is not detected by using the cause-effect technique or the model-based testing one. Only GAs combined with Simulink® model can detect it.

Number of Bugs

The results are shown in Fig. 2.20. GAs overperform the rest of the techniques used in this study because Simulink® blocks are used as shown in Fig. 2.13. Regarding the model-based testing, the fact of using models ensures better results than the cause-effect technique. Finally, the cause-effect technique performs least efficiently as no

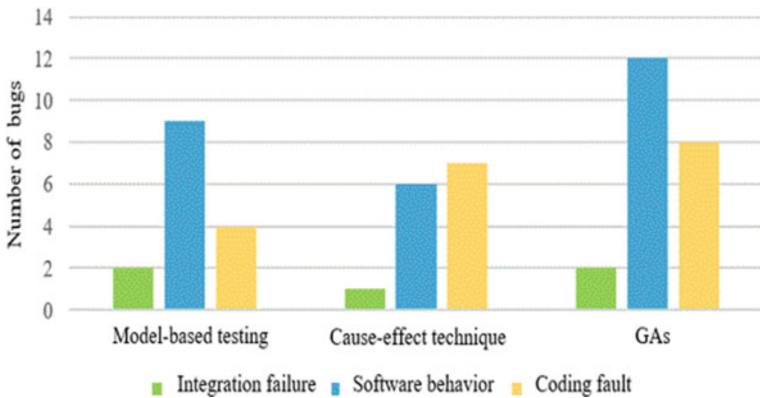


Fig. 2.18 Types of bugs found

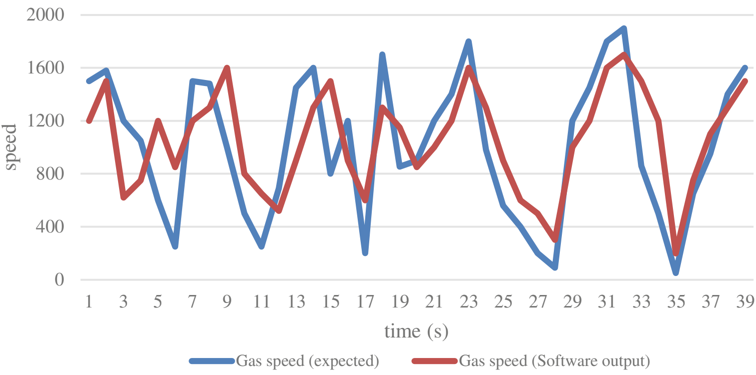


Fig. 2.19 Bug not detected unless GAs are used

model is used. The result is that it is extremely difficult to establish both the code and functional coverage.

2.3.4 Discussion

2.3.4.1 Test-Case Formulation

Several challenges must be considered when designing test-cases.

a. The engine ECU software consists of SMs composed of an important number of inputs and outputs which are usually analogical. Consequently, their values range between specific intervals. When running test-cases, it is difficult to reach

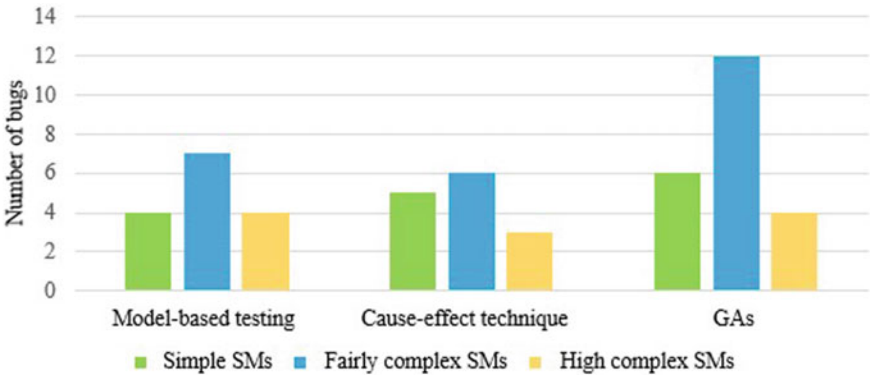


Fig. 2.20 Number of bugs found by using each technique

values contained in the variable range. For example, a variable representing the soot present in the diesel particulate filter can take a value of 40 g.

b. Considering the number of variables of SMs and their ranges, it is not possible to generate and run all test-cases which could cover the whole combination of the spectrum. In some occasions, when a variable takes a value close to its upper limit, the test-case can fail. However, if values are not close to this value, the test-case provides the expected results. That is why, at least during the validation process, the functional model must be covered with different test-cases which take different combinations.
c. Constraints must be considered to avoid generating uncoherent test-cases (for example speed = 100 km/h and first gear engaged).
d. When running a test-case by using automation processes, it is not possible to obtain the exact values indicated in the test-case due to SM interactions. Thus, the expected outputs specified in the test-case might be no longer valid. Therefore, the traditional formulation of test-cases based on input and expected output values cannot be used in simulations. Dlls allows solving this technical issue as depicted in Fig. 2.5. Thanks to them, it is always possible to assess Eq. (2.2) as they can provide the output values for the input ones reached during the HIL simulation. Therefore, GAs can check if software runs as expected by comparing the HIL results and the Simulink® models results.

2.3.4.2 Test-Cases Automation

Python scripts for automating the process must keep the inputs of the SM in a specific range. Otherwise, the expected output of the test-case may be no longer valid (Fig. 2.16). Regarding fairly complex SMs, as the number of variables present in SMs is high, it is recommended to use the tester-in-the-loop. High complex SMs have many functional states linked to the number of kms covered (example oil dilution

rate). Consequently, reaching a functional state is not difficult and test-cases can be fully automated.

GAs allow testing most of the SMs present in the engine ECU software except for:

a. *Estimators*. There are SMs responsible for predicting temperature and other magnitude trends of certain components, which involve many calculations. The easiest way to test these SMs is to perform data acquisition by using prototype vehicles and, then, the obtained.dat file is injected into the Simulink® model. The difference between the data acquisition and the Simulink® outputs is expected to be close to zero.
b. *Networks*. The most important network in cars is the CAN (Controller Area Network). In these cases, the testers have to verify if frames are transmitted and received properly, how the engine ECU reacts when receiving an invalid value or an absent frame, etc. This statement can be applied to other types of networks. It is easier to validate networks by using the HIL simulations than using prototype vehicles.
c. *SMs which are not modeled by using Simulink®*. Dlls must be used if GAs are applied. Not all SMs of the engine ECU software have a specification based on Simulink® model. Consequently, GAs cannot be applied to any SMs. However, only 7% of the SMs did not have Simulink® models.

Certain high complex SMs need to cover many kilometers to reach the specific operating point indicated in the test-case. When validating the software, GAs cannot be used as the number of generated populations is not compatible with the project planning. In these cases, the cause-effect technique is recommended to reduce the validation time. Anyway, these SMs can be validated by using GAs if the calibration dataset is modified in the same way as it is done in this study.

2.3.4.3 Means Used to Validate

Using the most adequate means to validate is an essential topic as:

a. The difficulty to reach an operation point depends on the means used to validate. It is easier to use test failures on a probe by using the HIL model than by using a prototype vehicle. If the wrong means is chosen, many attempts are required to run the test-case properly.
b. The chances to find more bugs than by using other techniques are increased as the validation time is reduced. Consequently, test-engineers have time to run more test-cases than other techniques. Thus, the code and functional coverage are increased. In addition, implementing a model by using the model-based testing and GAs reduces redundancies in test-cases.
c. The productivity gain obtained thanks to GAs has an important impact on software quality. As shown in Fig. 2.21, if software version A is validated with some delay (weeks 17 and 18), after the specifications for software B are sent to the

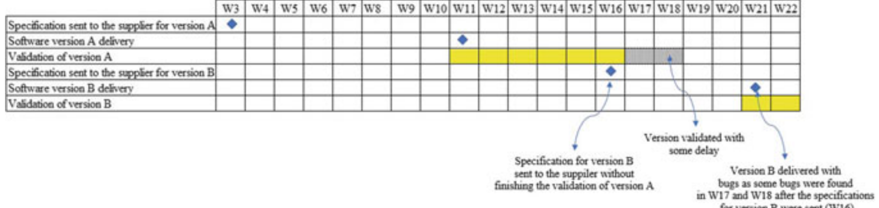


Fig. 2.21 Delays in software validation and impacts on software quality

supplier (week 16) in charge of coding the software, the software version B is delivered with bugs found in weeks 17 and 18, which may be blocking points. Therefore, the software version B could be not usable.

d. The test-engineers establish the best means according to their experience when using the model-based testing and the black-box technique. Regarding GAs, a multidisciplinary team sets the cost of the functional model.

2.3.5 Conclusions

Engine electronic control unit (ECU) software is one of the most complex software which is in charge of controlling the engine as well as other systems such as exhaust after-treatment systems. Among the main issues that test engineers can face is how to choose the best means to validate (HIL simulations or prototype vehicles) as well as design test-cases which are representative enough.

This research uses two GAs to establish the best means to validate SMs and to generate test-cases in which the expected outputs are no longer needed thanks to the usage of Simulink® models used to develop the engine ECU software with the aim of improving code and functional coverage, software bugs, test-case automation capacity and productivity. The obtained results were compared with the ones got by using traditional techniques such as the model-based testing or cause-effect ones.

The results obtained in this research show that GAs can find similar results for simple SMs and high complex ones. However, when it comes to fairly complex ones (the ones that are more present in the engine ECU software), GAs perform better than the other techniques as at least 7 more bugs were found. When it comes to functional and code coverage GAs perform better. When it comes to functional coverage, GAs improve up to 11% in fairly complex SMs and 8.4% for high complex SMs when using the cause-effect technique. When it comes to the model-based testing technique, GAs improve up to 4% in fairly complex SMs and 3% for high complex SMs. The code coverage is also improved by GAs reaching 12.8% and 7% for fairly complex and high complex SMs respectively when using the cause-effect technique. When using the model-based testing, GAs perform better up to 7.1% and 1.4% for fairly complex and high complex SMs respectively.

Another advantage of using GAs is that they can detect all types of bugs thanks to the usage of Simulink® models contrary to other techniques such as the model-based testing and the cause-effect ones.

The implementation time is compatible with an engine project planning as shown in this research.

2.4 Application of Rule-Based Expert Systems in Hardware-In-The-Loop Simulation. Case-Study: Software and Performance Validation of an Engine Control Unit[5]

2.4.1 Introduction

2.4.1.1 Background

Innovative techniques to validate software are needed to reduce cost and increase software quality.

This research aims to check if two rule-based EXs combined with dlls perform better than other techniques widely employed in the automotive sector when validating the engine control unit (ECU) software by using a HIL simulation.

To perform this research fifteen SMs of different complexity were chosen to be validated in an HIL simulation by using different techniques such as the manual execution, the tester-in-the-loop, the model-based testing, a rule-based EX and the combination of two EXs to establish the code and functional coverage, the productivity gain, the number of bugs found, potential limitations of each technique and the success rate of the HIL simulation. The test-cases used are described in-depth in the method section.

The enhancement, that dlls and EXs offer, depends on the number of states in the functional model used in the EXs and the number of subintervals in which the SM inputs can be divided. A range between 6 and 16 more bugs can be detected when using two EXs. The HIL enhancement can reach 6%, 16.8% and 18% depending on the SM complexity.

2.4.1.2 Engine ECU Software

The electronic architecture of today's vehicles is extremely complex. As a result, the number of ECUs present in vehicles is increasingly high [1, 2]. This trend will continue in the next years, thanks to driving assistance systems, which are essential for

[5] Extracted form *Journal of Software: Evolution and Process*. 2020, Volume 32, Issue 1. https://doi.org/10.1002/smr.2223, https://onlinelibrary.wiley.com/journal/20477481.

autonomous cars. ECUs are composed of hardware and software whose complexity depends on the function carried out in the network. Therefore there are multiple software running simultaneously and coexisting in a commercial car [5, 35]. This fact forces manufacturers to improve the software quality and the validation processes [3]. In addition, it is not difficult to find estimates that indicate that the total number of lines of code present in the software ECUs of a vehicle can reach up to more than 100 million. In the future, these figures will even grow significantly up to 200 or 300 million in autonomous vehicles.

Powertrain 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 (eg, consumption and emissions) [69]. The key element of the control system is the engine ECU composed of complex hardware and software. The hardware is responsible for getting information from sensors after a filtering process to reduce noise in signals. The software processes all data received and handles actuators to reach the operating point. In addition, when a vehicle is in motion, the engine ECU (hardware and software) interacts with other ECUs to ensure the proper functioning of the car. This implies that each ECU should receive the information at a specific time. Therefore, 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. Otherwise, some failures could occur and lead to the situation in which the vehicle stalls. This fact makes the most safety critical parts of the software a hard-real-time (HRT) system. In other words, the system is subjected to real-time constraints in which every critical task must be executed at a specific deadline to ensure the correct operation of the system. 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.

This chapter is focused on the engine ECU software validation (one of the most complex software present in a vehicle) and shows solutions to the main difficulties associated with traditional software validation techniques. The solution proposed is showing that two EXs working in cooperation and combined with dynamic-link libraries (dlls) perform better than traditional techniques such as the model-based testing or tester-in-the-loop among others.

2.4.1.3 Techniques Currently Used

The engine ECU software validation is based on HIL simulation, combined with different techniques for generating test cases. Three key stages must be considered when performing an HIL simulation: test-case generation, test-case execution, and validation of the execution results.

One can find different definitions for the black box concept such as "*the black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings*" [70, 71]. Among others, there are three types of techniques used when applying the black-box one:

a. Equivalence portioning

The inputs of the SM under validation are divided into partitions, and after having selected representative values for each partition, the test case is conducted. Then the software behavior is analyzed. The model-based testing can be defined as the automatic generation of software test procedures, using models of system requirements and behavior. To do this, a functional model must be implemented. This technique may be considered in this research as an equivalence portioning technique in the black-box testing. Because test cases are derived from functional models and not from the source code, the model-based testing is usually seen as one form of the black-box testing. The main advantage of this functional model is that all functional states and the transition from one state to another are indicated. Thanks to this, it is easier to assess the functional coverage as the number of states covered when validating an SM is known.

The EX combined with dlls consists of using an EX to assess if the software behaves as expected. The EX is built by using rules coming from the specifications and software requirements. The dlls are the Simulink model of the SM under validation that allows calculating the software outputs when performing the HIL simulation despite the SM interactions. This topic is analyzed in-depth in this research. The authors have considered this technique as an equivalence portioning one as it is exposed in this chapter.

b. Boundary value analysis

Boundary values for the SM inputs are determined and the test-case obtained is performed. Then the software behavior is analyzed.

c. Cause-effect technique

In the automotive sector, the test engineer usually has to validate cause-effect test cases that come from the software requirements. As a result, given a series of specific causes (conditions related to inputs), the validation process has to check the effect (software behavior). An example of a possible test case could be: "In case of an ESP frame is absent, the stop and start function must be inhibited." The tester-in-the-loop, the manual execution, or automated can be considered as cause-effect techniques in this research.

All techniques that may be used to validate the engine ECU software have to face several issues such as the SM interactions that prevent reaching the values established in the test case, the type of bugs that can be found, and the problem of enhancing the code and functional coverage. Considering that the engine ECU software has up to 70 complex SMs, the interaction between SMs is continuously present and disturbs the validation process such as electronic noise. Consequently, given a test case, it is almost impossible to make the inputs reach the desired value. The main consequence is that the expected output set in the test case could be no longer available.

Some types of bugs are extremely difficult to detect by using HIL simulation unless a technician uses a significant amount of time to analyze the data acquisition. Figure 2.22 shows an example, the obtained result for an output for a variable of

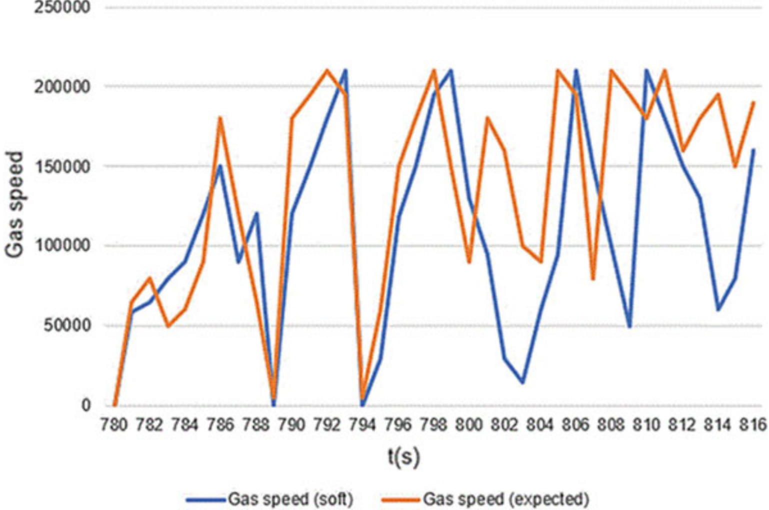


Fig. 2.22 Bug not detected when using black-box technique

an SM when running 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 could impact the amount of urea injected to treat NO_x. Because this bug is not linked to a functional bug, it is impossible to detect it by using the black-box technique. The detection of this type of bug involves checking and detailed analysis of the software code by running additional software.

Considering all aforementioned, the main limitations associated with these techniques currently used in the automotive sector when using the HIL simulation are depicted in Table 2.29. The aim of this research is to solve all these limitations by using two EXs working in cooperation combined with dlls. The fact of using two EXs allows improving the code and functional coverage and gaining a better control of the automation process, thanks to dlls. It also provides an opportunity to detect any type of bugs.

2.4.1.4 Related Works

The engine ECU software validation is based on HIL simulation. Several stages must be considered when performing an HIL simulation such as test-case generation and test-case execution.

A test-case consists of a set of inputs and their expected outputs that the software should provide when working properly. In an HIL simulation, a test case is run, and the obtained result is compared with the expected one to check whether the software has operated properly for this specific test case [72–74]. There are many different

Table 2.29 Problems analyzed in this research

Limitations	Reason	Possible solution
Difficult to validate the software automatically	When the values set in the test case for the inputs are not reached, then the output values set in the test case may be no longer available. Consequently, no automatic validation can be performed	Dlls can perform this task as shown in this research as they recalculate the output values that the SM under validation should provide for the specific input values reached after the HIL simulation. Therefore, an automatic validation process can be carried out
Possible bug performance detection improvement	If input values are different from the ones established in the test case, then the software performance behavior is unknown	
Functional coverage unknown	A functional code coverage could be established by analyzing the black-box test cases before the HIL simulation However, when reaching different values for the inputs after HIL simulations, then the use cases tested are different from the ones planned	A performance rule-based EX can assess the functional coverage as exposed in this research. An EX can assess whether the SM under validation performances as expected or not, thanks to the rules used for its implementation. Thus, performance bugs could be detected. Considering the number of performance rules assessed, the functional coverage could be established
Difficult 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	Dlls can perform this task as shown in this research as they can be used for checking whether the SM under validation calculates all SM outputs properly
Difficult to assess the code coverage accurately	There is no code model or something similar to use it for calculating the code coverage when using the black-box or similar techniques. It must be considered that there are many if-then structures in the software, which makes it extremely difficult to test all possible paths. However, the question is if the whole performance rules have been tested with a considerable number of software rules	A software and a performance rule-based EXs can assess the functional and code coverage as exposed in this research. It can be employed to establish the code and performance coverage

ways to generate a test case, such as assigning specific values to all inputs of the SMs under validation to cover a functional model, as exposed later in this research, or assessing the software performance when checking each software requirement [75–79]. The former is very difficult to implement owing to SM interactions, as it will be discussed in this paper. The aim of this method is to make the inputs reach specific values and check the outputs. The latter is widely used because the inputs of

SMs do not need to reach exact values but approximate ones to check the software performance. As a result, it is more flexible.

The black-box technique has been used for a long time in the automotive sector, as discussed by Conrad [80]. Despite its widespread use, it is true that it has some weak points, as discussed by Chunduri [81]. In their dissertation, they consider that test cases based on the engineers' experience usually imply gaps and test redundancies. Thus, they proposed a methodology to improve the black-box technique and the test-case generation. To do this, they proposed to work on three factors: enhancing function requirements specification, establishing traceability across test levels, and obtaining comprehensive function test-coverage information. In addition, it is essential to remark that the test-case execution must not be too time-consuming. Consequently, more test cases can be run, and the code/functional coverage is improved. Some research has also been focused on this topic. Zhou et al. proposed the optimized use of symbolic simulation with the aim of reducing the time required to generate a test case at the IEEE Conference [75]. As a result, given a model of a software function under validation, the time needed to cover the model will be reduced. Sopan-Barhate presented their theory about how to make the software validation process in the automotive sector more effective at the International Congress of Electronic Instrumentation and Control [76]. In their opinion, the main concerns linked to the software validation process are how to design representative test cases as well as how to prioritize the test-case execution based on priority levels, ensuring, at the same time, high code coverage rates. The solution proposed in their research is the use of orthogonal array testing.

Model-based testing is a good technique to test SMs, and it allows the assessment of the code/functional coverage in an easy manner. Raffaëlli et al. at the Embedded Real Time Software and Systems Conference, presented research focused on the usage of a functional model by running Matelo software [82, 83]. The aim of this research was to accurately assess the code coverage, as all branches of the model could be tested. The application in an HIL simulation for a more complex ECU, such as an engine ECU, was not shown. Perez et al conducted a review on the current state-of-the-art techniques used for the verification and validation of embedded systems, including software developed in the automotive sector [84]. Their main conclusion shows the need of further research concerning automatic validation, safety tests, and model validations. In short, these concepts are clearly linked to the test-case generation and improvement in automation processes. The aforementioned aspects are analyzed in-depth in this chapter.

There are many ways for automating HIL simulation in the market [85, 86]. The automation process is mainly based on black-box techniques such as those reported by Köhl et al: "*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)*" [86]. At the 52nd Congress of the ACM/IEEE Design Automation Conference, Petrenko and Nguena-Timo reported the main problems and solutions associated with software validation in the automotive sector, on the basis of the experience of General Motor Research and Development staff, powertrain software validation team of General Motors, and the Centre of Montreal [87]. Their main

conclusion was focused on the methodology known as the “tester-in-the-loop,” in which the test engineer leads the system to a desired operation point, considered as a crucial one, with the aim of assuring the correct execution of the test case in such a way that the software behavior can be assessed. Once the crucial point is reached, a series of automated actions are executed to reach the goals previously established in the test case. Tatar and Mauss proposed at the ERTS Congress: Embedded Real Time Software and Systems the possibility of not using HIL simulation. Instead, by using a virtual platform, engine ECU software could be validated, thanks to the interaction with a car model [88]. As a result, many points could be tested. All the possible issues or bugs linked to the software integration on the hardware would not be detected. Koopman and Wagner exposed the main future issues when it comes to software validation in the Society of Automotive Engineers Congress. One of the most important concepts introduced in their dissertation was the “driver-out-of-the-loop” concept. Currently, the ECUs are validated by considering the driver’s actions on the vehicle (accelerations, braking, etc.). If the vehicle is autonomous, these driver’s actions are not relevant, and some external factors such as traffic and pedestrians must be considered to validate the software. As a result, they consider machine learning techniques as a key aspect in the future.

2.4.2 Method

2.4.2.1 Description

The aim of this chapter is to validate the following hypothesis:

> Two EXs working in cooperation perform better than traditional techniques when validating an engine ECU software. In addition, two EXs can overcome the difficulties depicted in Table 2.29.

To do this, a series of test cases are run by using the following techniques: the cause-effect one the model-based testing one, one EX combined with dlls, and finally, two EXs combined with dlls by using the HIL simulation. Then the following parameters are measured for each technique to validate the hypothesis: code and functional coverage, productivity, bugs found, and automation process success.

2.4.2.2 Data Used in This Research

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 They have been considered as representative for this case study by the authors and the company subjected to this research. Considering the experience

of the company that is the subject of this case study, three types of SMs or functions can be distinguished as shown in Table 2.2.

When generating test-cases, three strategies were followed in this research:

1. Generating pseudorandom values for the SM inputs under validation in such a way that all paths of the models that belong to EXs are covered. For each combination of the inputs, the performance EX must assess the expected behavior of the vehicle (represented by an HIL bench) in cooperation with a software EX that will cover a software model to assure a high code coverage. The right outputs for all inputs generated in the test case are known by using the dlls. All aforementioned statements are exposed in this section. In this chapter, as exposed later, manual test cases were also generated in order to cover the functional and software models.
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. All data stored in this database are handled in meetings with the supplier responsible for coding the software and designing the hardware on a weekly basis. Test engineers design test cases on the basis of different inputs such as this database, functional defects found during driving tests, specifications requirements, as well as the defects found when the engine has been marketed. The goal is to keep the test-case libraries as complete as possible over time. When the test engineer has designed the test-case library for a specific SM, a validation process is carried out. The test engineer and the designer of the SM verify whether the use cases presented in the test-case library are representative enough. For each of the test cases presented in the database, it is possible to assign values to the SM inputs with the aim of checking the software rules.
3. Pseudorandom values are generated by Matelo software with the aim of covering the whole functional model. It must be reminded that this technique is an equivalence portioning one. As test cases are generated by Matelo, the functional model is covered. Matelo assesses the functional coverage automatically. Matelo could also be used to implement a software model. However, authors have not carried it out in this chapter.

Table 2.30 shows the number of tests considered in this research according to the type of SM.

The difference between the number of test-cases for each type of SMs is because the fairly-complex SM involves a greater number of use-cases.

Table 2.30 Number of tests used in this research

Type of SM	Number of test
Simple	250
Fairly-complex	1,250
Highly-complex	100

Table 2.31 Two EXs combined with dlls

Technique	Method
Cause-effect technique	A1
Model-based testing	A2
One EX combined with dlls	A3
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 and manual values generated by Python scripts

Table 2.31 indicates the methods followed to generate test-cases for each technique.

It is important to analyze what A2 and A3 mean. In A2, Matelo can generate off-line (before the HIL starts) all necessary test cases with the aim of covering the functional model. In A3, Python scripts also generate test cases trying to cover the software model. The Python scripts generate pseudorandom values trying to reach software states not implemented in the model. A software state not implemented in the model involves a use case not considered by the design team, in other words, a design error. In addition, a test engineer generates manually off-line test cases by establishing the most likely combination of variables by using fuzzy values to cover the functional and software states. This process consists of avoiding illogical situations such as engaging the fifth shift when the vehicle is at 5 km/h. These inconsistencies must also be taken into account when generating automatically test cases by using Python scripts. The fact of using fuzzy variables, as exposed later, allows increasing the combination of the inputs of the SM under validation. These test cases generated manually are run by using Python scripts.

For confidentiality reasons, the list of test cases cannot be published. However, It is important to remark that 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.

2.4.2.3 Equipment

The following means used in this research are shown in Table 2.32.

2.4.2.4 Methodology Proposed

In this section, the key elements used in this technique are presented (EXs and dlls). Then, the process how they collaborate to run a test case is described.

Table 2.32 Equipment used in this research

Item	Description	Phase where the item is used	Cost
HIL Bench	HIL bench manufacturer dSpace®, model dSpace® Simulator Full-size (dSpace, 2016a). Versatile HIL simulator capable of emulating the dynamic vehicle behavior	Every time a test-case is run. Necessary for the HIL simulation no matter which technique is used	Depending on the characteristics of the HIL bench, the price can vary. Estimation for this case-study: €100,000 each bench
INCA version 7.1.9 provided by ETAS® (BOSCH) (ETAS, 2017)	Software used to make measurements of different software variables stored in the engine ECU memory	Every time a test-case is run. Necessary for the HIL simulation no matter what technique is used	The price depends on the number of licenses. For a big car manufacturer, an estimation of €5,000 for each license can be made
Matlab® R2013 and Microsoft Visual Studio 2015	Software necessary to create dlls	Every time a test-case is run and the user wants to avoid the SM interaction problem	The price depends on the number of licenses. This information was not provided by the company subjected to this case-study
Matelo®	Software used for validation purposes being able to generate test-cases	Necessary to generate test-cases when using the model-based testing technique	The price depends on the number of licenses. Estimations of 20 licenses are €100,000
ControlDesk® version 5.1 from dSpace. (dSpace, 2016c)	This software is needed to build the HIL model which belongs to dSpace® manufacturer. The HIL model was built by the company subjected to this case-study	No matter which technique is considered	No information about cost was provided by the company subjected to this case-study

a. Expert systems

Two EXs are distinguished:

- Software EX

  Its aim is to establish the software rules which must be applied to assure the software operation, such as a sequence of updating variables to be followed when a failure occurs. A software rule is a Simulink® path to be followed to reach a specific operation point.

- Performance EX

  The second EX is responsible for checking whether the vehicle responds as expected for a specific use-case. The first EX only verifies if the software rule is applied. The other one is abstracted from the software and only focuses on the fact of verifying the correct behavior from vehicle performance point of view. Properly coded software may exhibit wrong behavior owing to design errors as some use-cases were not considered in the specifications used for coding the software.

b. dlls

As exposed earlier, it is highly unlikely to reach the operation point set in the test case because of SM interactions. This fact implies that the automation process is not easy to be performed. Figure 2.2 depicts the process to automate a test case by using Python scripts. During the HIL simulation, the script is in charge of performing all the necessary manipulations on the driver-ECU interface model automatically. During this process, a data acquisition is performed by employing the INCA software. If these values are not reached after time out elapsed, the data acquisition is stopped and the dll is called. The dll represents the Simulink model of the SM under validation, and it allows assessing and providing the expected values of the SM for a specific state of the ECU. Thus, by using dlls, it is always possible to obtain a result after an HIL simulation. Thanks to this data acquisition and a C-file, it is possible to call the dll.

c. EXs and dlls working in collaboration

Figure 2.23 describes the process.

- **Phase 1**. The software EX establishes the test-case to be run. It must be reminded that a rule corresponds to a Simulink® path of the model of the SM under validation. This rule is communicated to performance EX with the aim of establishing the performance rule to be applied during the HIL simulation.
- **Phase 2**. The HIL simulation is performed trying to reach the operation point established in the test-case.
- **Phase 3**. A test-case is composed of a series of input values and the expected outputs. If the specific operation point is not met after a specific time elapsed, then the expected output set in the test-case may not be longer valid. The dll of the SM under validation allows assessing the right output values for the current engine ECU state. The software EX collects this information and assesses the software rule that was tested after the HIL simulation.
- **Phase 4**. The software EX sends a message to the performance EX about the software rule tested in such a way that the performance EX can update (if needed) the expected software behavior.
- **Phase 5**. Both EXs checked the HIL simulation results and decide whether the software behavior is correct and meet the specifications.

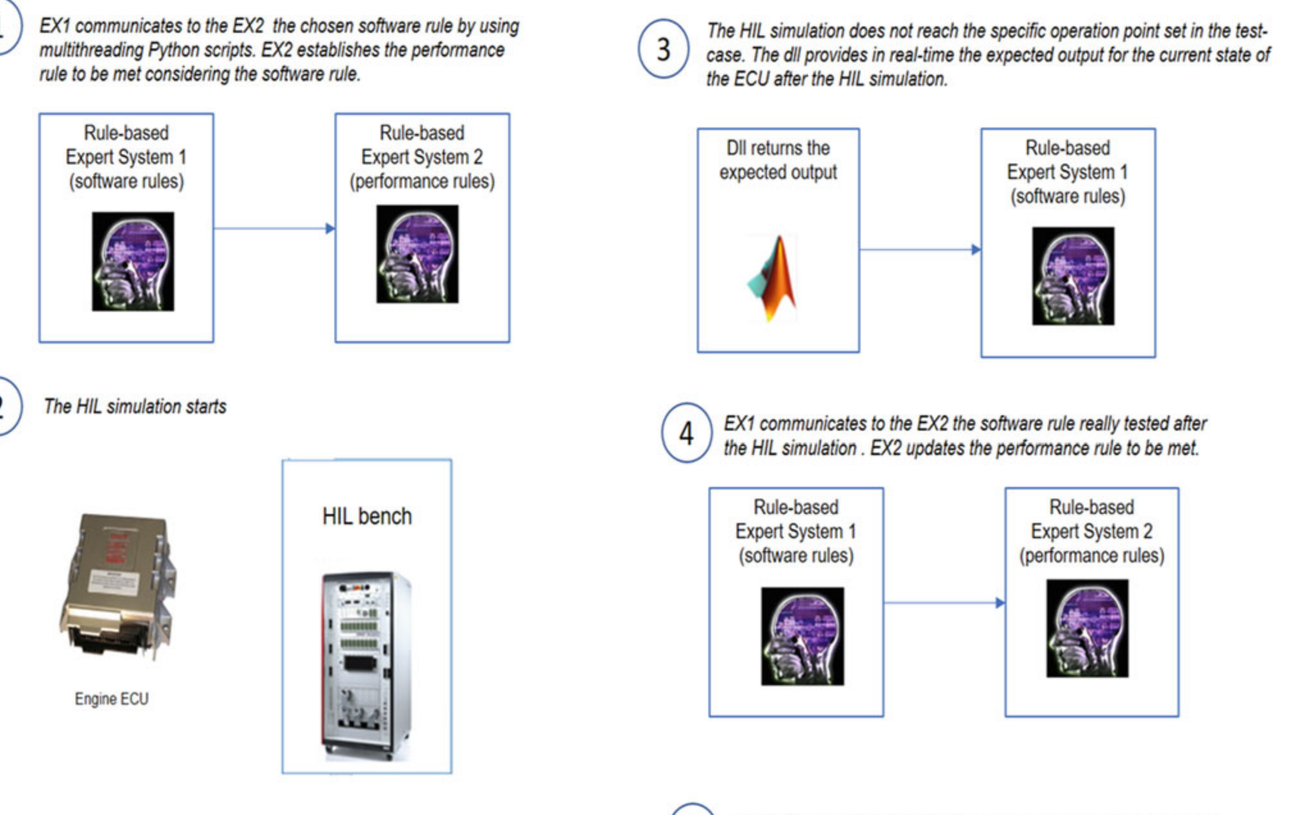


Fig. 2.23 EXs working in cooperation

2.4.3 *Validation of the Key Elements: EXs and Dlls*

This section describes the validity of the different key elements involved in this research.

2.4.3.1 Expert System Validation

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. The EX design is shown in Fig. 2.24. As shown, there is a knowledge base composed of rules coming from functional or software 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 or software models describing different states that the system can process when applying the rules presented in the knowledge base. It must be reminded that two EXs are designed for each SM under validation.

a. Software expert system

The aim of this EX is to check whether the software meets software specifications. To better understand this, Fig. 2.25 must be analyzed. One can see a software model of a given SM, where S1–S6 represent a state. In this case, the state represents a part of the Simulink model. The conditions to be met to pass from one state to another one come from the Simulink model used to code the software. As a result, depending on

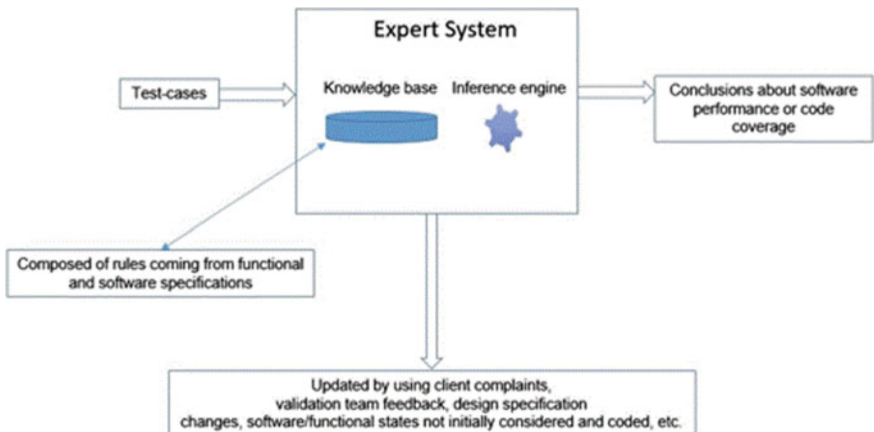


Fig. 2.24 Scheme of the EXs used in this chapter

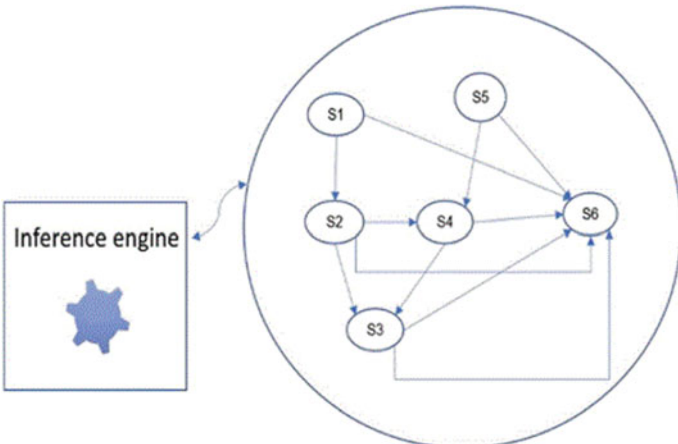


Fig. 2.25 Inference engine in detail

the HIL simulation, the values of the software variables of a given SM are analyzed in such a way that the final state is set. By checking different states covered after having executed a certain number of test cases, it is easy to have the first estimation of the code coverage. As exposed in the performance section, a test case could be run and the inference engine may not know in which software state the system is. This fact can occur, and it happens when a use case has not been considered by the design team. That is why all states in Fig. 2.25 are linked to state 6 as it represents an unknown software state.

To obtain an accurate code coverage, two key actions have been performed in this research:

- Generation of test cases in such a way that the range of possible values for a given variable is divided into intervals. In this way, the probability of covering all paths of the Simulink model is increased.
- Usage of as many states as necessary to describe the system.

b. Performance expert system

The performance EX is built by using functional states in which the vehicle can operate. Therefore, the model is not focused on part of the Simulink model of the SM under validation. The fact of covering the functional model allows assessing the functional coverage but not accurately as depicted in Fig. 2.26 when assessing the transition from S2 to S4; it is unknown if the value for Out1 was obtained following the path1 or the path2.

When a test case is analyzed by the performance EXs, after having applied different rules, the inference engine determines the state of the system. Therefore, the EXs 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.

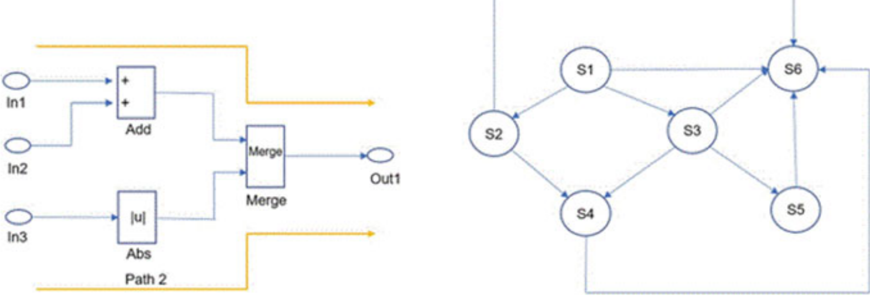


Fig. 2.26 Inference engine in detail

As shown, all functional states (S1, S2, S3, S4, and S5) are related to a state called S6. S6 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 EXs if needed. The S6 state will be analyzed later. In this research, the EX code is not provided as it belongs to the company's know-how and is confidential.

The validation process of both types of EXs is stimulated following these two phases:

- The established rules, used by the EX, are checked following a procedure consisting of a meeting between designers and testing engineers to assure the conformity of the EX. Then, the EX is implemented by using Python.
- The aim of the validation process is to check two key characteristics: firstly, to assure that the rules presented in the knowledge base are coherent and secondly, to verify that the EXs can assess the software performance properly. To do this, a set of data acquisitions, already analyzed by test engineers, is used for the aforementioned purposes.

2.4.3.2 Dynamic-Link Library Validity

Dlls are a key element of this research. 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:

- The effects due to inputs and outputs of SM interactions are collected in the data acquisition file as it is the result of the HIL simulation.
- It is essential to distinguish some important points when it comes to designing the engine ECU software. Before integrating the software into the hardware, there is a process of building prototypes with the aim of checking whether the Simulink models work properly. Once this is checked, the decision of integrating software and hardware is made. Afterwards, the design specifications are written, all the SMs are assembled, and finally, a software is coded and the validation

process starts. Therefore, the Simulink models are the transcription of the functional specifications of the engine ECU and must be met independently of the SM interactions, hardware design, task scheduling, software-hardware integration, etc. In addition, Simulink models are tested before sending the specifications to the supplier in charge of coding the software. 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 discovered. Otherwise, the functional specifications are not met.

- The fact of only considering one dll corresponding to the SMs under validation does not imply that software and hardware integration is considered as the inputs processed to the dll are the consequence of an HIL simulation. Therefore, 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.

2.4.3.3 Measurement Conditions

Before starting the HIL simulation, some conditions must be met. Otherwise, the result is rejected:

- The information provided by the probes must be equal in all cases (with and without dlls) when it comes to external factors such as air and pressure temperature and slope of the road.
- The engine ECU memory must contain no errors before starting the HIL simulation. If it does, then it must be erased by using the procedure established by the ECU supplier.
- All test-case executions must be conducted on the same HIL bench. This factor is important to assure that the same probes are being used during the whole research.

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 and it must be executed again as the HIL model could have failed.

2.4.4 Practical Implementation

A key issue in any project is costs. Therefore, costs must be reduced as much as possible. Therefore, in this research, it has been tried to implement software validation by using Python packages. Each test case is run by using Python scripts and C-code. Firstly, the test case is performed by using Python scripts that interact with the HIL model with the aim of reaching the values established in the test case. During this process, a data acquisition is completed in ascii format. Secondly, a C-code is used to call the dlls and to assess the software behavior.

2.4.4.1 Python Scripts When Using Two EXs

From a pseudocode point of view, a multithreading implementation was conducted. One can find the thread responsible for generating software rules that will be sent to two threads: the one in charge of automation control and the one that handles performance rules (Fig. 2.27). The process is as follows. A software rule is chosen, and consistent inputs values for the variables involved in such rule are generated. Then a message is sent to the EX 2 to set the rule to be applied according the one chosen by EX 1. Once done, the automation process can be conducted using an HIL simulation. EX 2 thread is waiting for the result. The automation thread sends a message indicating if the result was correct, that is to say, whether the system reached values close to the desired operating point. If so, the EX 1 communicates to EX 2 that the selected rule was correct. Otherwise, the EX 2 updates the performance rule to be applied according to the operation point that was reached in the HIL simulation.

The second thread is in charge of controlling the automation process (Fig. 2.28), which starts when the EX 1 thread establishes the software rule to be tested (Fig. 2.28 waiting_message_from_expert_system_1). Once the process starts, the automation thread tries to lead the system to the desired state set by the EX 1 thread. The automation process ends:

- when this operating point is reached. In this case, the software and performance rules for both EX must not be updated (Fig. 2.27 automation_OK)
- when a time out elapses as the operating point is not reached because of SM interactions. In this case, the software and performance rules initially chosen might be updated (Fig. 2.26 else).

```
Thread for controling expert system 1

rule= select_rule()
data_for_automation=generate_data_for_the_rule(rule)
send_message_expert_system_2()
wake_up_thread_automation()
waiting_for_automation_result()
If automation_OK then
    send_confirmation_message_expert_system_2()
else
    send_message_expert_system_2()
end
```

Fig. 2.27 Pseudocode of software EX thread

Fig. 2.28 Pseudocode of automation thread

```
Thread for controling the automation process

waiting_message_from_expert_system_1()
while time < time_out
    control_HIL_simulation()
end
sending_current_status_to_expert_system_1()
```

Finally, thread 3 is responsible for managing the performance EX (Fig. 2.29). Its practical implementation is extremely simple, as it only runs when it is allowed by the EX 1. This can take place in two distinct situations: firstly, when the thread is instructed to select the rule to be applied according to the one set by EX 1 and secondly, when it is indicated to proceed to update the rule depending on the final engine ECU state, once the process of the HIL simulation is completed.

To implement a cross-thread communication, a submodule event from the Python threading package was chosen. Its main advantage is its ease of use. Using the wait() and set() methods, it is possible to keep a thread waiting while another performs other tasks. When the latter ends, using the set method, an event occurs to wake up all paused threads. In this case, its use is essential for several reasons:

1. The automation thread and the EX 2 threads must not start calculations until EX 1 has been initialized.
2. The thread in charge of handling EX 1 must not continue its execution as long as the automation process is finished.

Fig. 2.29 Pseudocode of performance EX thread

```
Thread for controling the expert system 2

waiting_message_from_expert_system_1()
select_rule()
waiting_message_from_expert_system_1()
update_rule()
```

3. The EX 2 thread must not continue its execution as long as a confirmation about the current status of the ECU done by EX 1 is received. The main raison is that a rule updated could be necessary.

2.4.4.2 Dynamic-Linked Libraries

The implementation of dlls allows the use of the Simulink model on multiple computers without additional cost. The dll can be implemented by following the steps indicated in many Mathworks 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 case, these models are available as they are sent to the supplier to code the software. As described in Matlab documentation, the dll can be called by using different programming languages. In this research, C-language has been used. This process is depicted in Fig. 2.30. Firstly, when a test case is run, different software variables chosen by the user are recorded by using the INCA software. The result of this process is an ascii file that contains the variables (inputs and outputs of the SM under validation) and the specific time when each measurement was performed. Secondly, the ascii file is read by using a C-file in such a way that each line of the file is used for calling the dll (see phase 2, Fig. 2.30). The dll must return the expected output for the inputs used to call the dll. Finally, a comparison is performed as depicted in Fig. 2.30, phase 3. It must be reminded that the outputs of the SM are also available in the ascii file.

2.4.5 Results

2.4.5.1 Functional Coverage

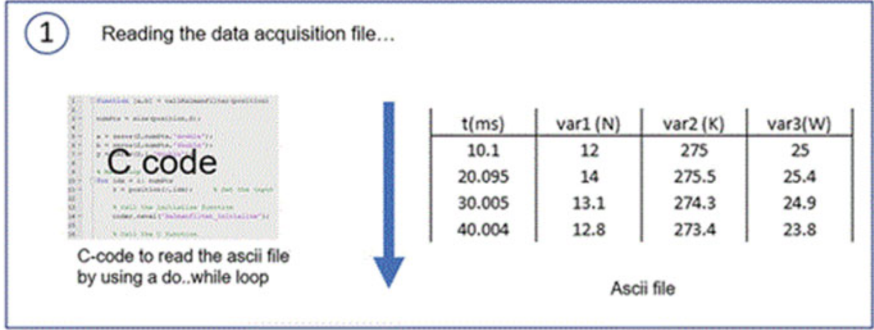
The functional coverage would be evaluated as Eq. (2.5). This equation is widely used in the automotive sector as it allows assessing the functional coverage in an easy way by using the software requirements. Table 2.33 depicts the total number of functional requirements linked to the SMs chosen for this research.

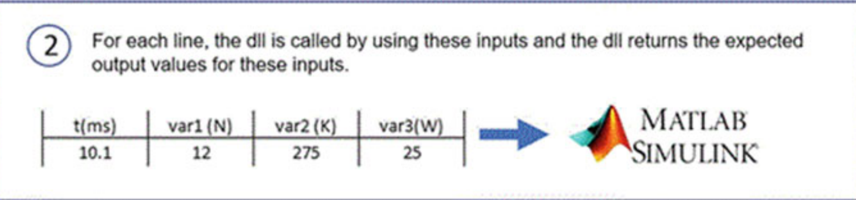
$$FC = \frac{\text{number of software requeriments tested by a technique}}{\text{number of software requirements indicated in Table 8}} \cdot 100 \qquad (2.5)$$

Table 2.34 shows the results obtained for each technique.

a. Cause-effect technique

The aim of the cause-effect technique is to check that the software requirements established at the beginning of the engine project are met. They come from a database in which the staff document different bugs found throughout the engine project. In other words, all test cases are based on the experience of the company subjected





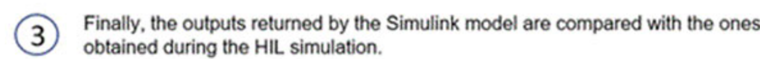


Fig. 2.30 Interactions between the C-code and the dll

Table 2.33 Number of total functional requirements

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

to this case study. These test cases can be run by using a manual execution or can be automated by employing Python scripts. The main limitation of the cause-effect technique is test-case redundancy [81]. This research confirms this statement. After having analyzed the test-cases run by using this technique, the authors found many of them which tested the same software requirements.

b. Model based-testing

As already exposed in this research, a functional model is built by employing Matelo software. In addition, this software is able to generate test cases with the aim of covering the whole functional model. The functional coverage can be calculated easily by using Eq. (2.5). Moreover, this technique allows detecting use cases not considered initially in the software requirements.

Table 2.34 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
Software EX and performance EX combined with dlls	71	94.6	360	90	465	91.2

When using Matelo (the model-based testing technique), it is important to expose the problems found during this chapter. 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. That is why this strategy could not be used. To face this issue, one can use dlls combined with Matelo. In this case, Matelo will not generate the test case, but it will control the automation process. In order 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 chapter, 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.

C. EXs Combined with dlls

The software performance is assured by using an EX capable of detecting whether the software behaves properly when a test case is conducted. As discussed earlier, the unexpected behavior can come from a coding fault or design error. In both cases, the performance EX can detect them. Therefore, the results obtained when validating the EX are analyzed in this section. As done in the previous case, a validation and a test phase were performed. The main problems obtained for the former phase are depicted in Table 2.35.

When the errors indicated in Table 2.36 were corrected, the EX was assessed during the validation phase. In this case, the same number of test cases used when validating the software EX was performed. The acceptance process was the same as reported in the software EX validation process (Table 2.36).

When using a performance EX, a certain number of test cases were conducted by assigning pseudorandom values to the inputs of the SMs: 25 for 20 simple SMs, 5 for fairly complex SMs, and 2 for highly complex SMs. Table 2.37 depicts the results obtained.

Table 2.35 Errors detected when validating the EXs

Type of error	Cases	Percentage	Explanation
Wrong syntaxes	6	5.5	Because the rules used to design the EXs are extremely complex, the programmer made coding errors
Incoherence among rules	2	1.8	In some cases of wrong performance of the EX, incoherence between rules was found
Misunderstanding of technical specifications	3	2.7	Because of innovative evolutions in some parts of the engine, some technical specifications were not understood properly
Rules not coded or forgotten	1	0.9	This type of error was made owing to the same misunderstanding of technical specifications

Table 2.36 Most important points checked during the validation meeting

Most important factors considered to validate the expert system
All safety concepts (ISO 26262) were modeled or considered in the EX
All diagnoses that may be detected by the engine ECU during the validation process were considered in the EX
The number of states is considered sufficient and representative enough by the project team
All use cases are modeled and considered in the EX (a priori)
The transitions among all the states considered in the EX are defined and modeled properly
The feedback of other projects was considered in the EX

Table 2.37 Code coverage when an EX is used

Type of SM	Number of rules	Number of functional states tested not checked when using an EX	Functional coverage (%)
Simple	75	7	90
Fairly complex	400	52	87
Highly complex	510	65	87.2

When both EXs are used together when performing an HIL simulation, the final results are enhanced, as more rules are checked as shown in this section (Table 2.38). The main reason behind this fact is that the higher the code coverage of the software EX, the higher the functional coverage obtained when carrying out HIL simulations. Therefore, it is essential that they work in cooperation. Another aspect that must be analyzed is why 100% functional coverage is reached when the software code coverage is not 100%. This fact can be easily explained as a specific variable can be activated by different software paths of the Simulink model. Figure 2.31 shows how output Out1 can be activated by two different paths. That is why the functional

Table 2.38 Number of rules or functional states not checked when an EX is not used

Type of SM	Number of rules	Number of functional states tested not checked when using both EXs	Software code coverage (%)	Functional coverage (%)
Simple	75	4	94.6	100
Fairly complex	400	40	90	95
Highly complex	510	45	91.2	94

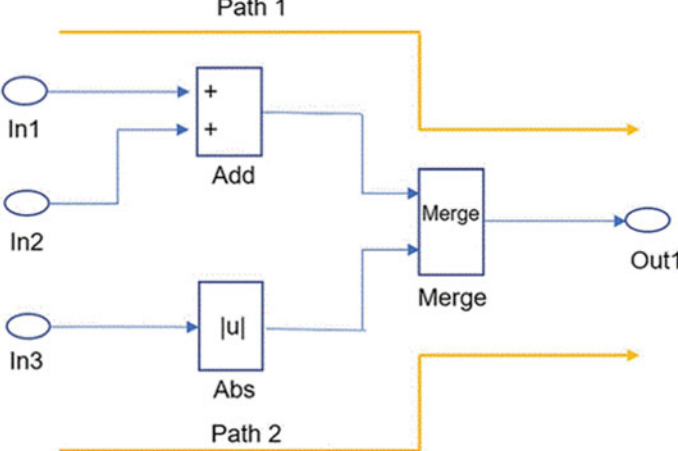


Fig. 2.31 Activation of a specific variable

coverage is 100% but not code coverage. This fact supports the conclusion that the number of subintervals is essential to get a high code coverage.

2.4.5.2 Code Coverage

The supplier responsible for coding the engine ECU software starts from the specifications composed of complex models which are provided by the car manufacturer. Thus, it is extremely difficult to reach a code coverage close to 100% as reported in previous research [81]. In order to assess the code coverage, the Eq. (2.6) was used which establishes the relation between the total number of Simulink® blocks to be tested (Table 2.39) and the total number of Simulink® blocks tested.

$$\mathrm{FC} = \frac{\text{number of Simulink® blocks tested by a technique}}{\text{number of Simulink® blocks indicated in Table 12}} \cdot 100 \tag{2.6}$$

Table 2.39 Number of total Simulink® blocks[a]

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

[a]When a state flow is present, each state is considered as a Simulink® block

The results obtained for each technique are shown in Table 2.40.

a. The cause-effect technique

When using the cause-effect technique, after having run all test-cases to assess the functional coverage the number of Simulink® blocks covered were calculated following the equation [2]. The cause-effect technique implies redundancies. Consequently, the code coverage is not high. The main limitation associated with this technique is that it is based on the software behavior and not on checking the code coverage and the number of Simulink® blocks covered.

b. The model-based testing

The model used for testing the SM under validation can be built from two points of view. The first one focuses on the functional software behavior. The other one focuses on the software structure, in other words, on the Simulink blocks without analyzing the purpose of each block. In this section, the second point of view is used. However, it faces the same problems already described when automating test cases because of the SM interactions.

c. EXs combined with dlls

Table 2.40 Code 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	63	78.7	265	75.6	410	77.3
Tester-in-the-loop	63	78.7	265	75.6	410	77.3
Model-based testing	63	78.7	265	75.6	410	77.3
Performance EX combined with dlls	74	92.5	295	84.3	435	82
Software EX and performance EX combined with dlls	76	95	313	89.6	425	80.2

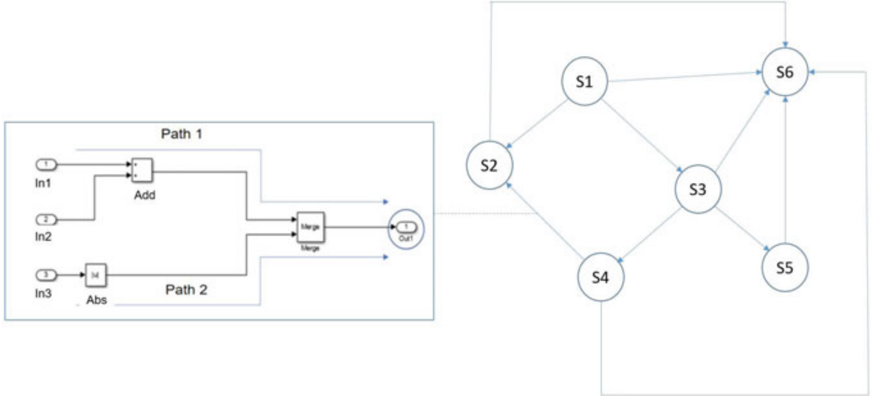


Fig. 2.32 Scheme of a software EX used in this research

A realistic way to assess the code coverage is to check whether all sub-blocks which composed a Simulink® model of a specific SM under validation, are verified after having run all the test-cases. In this research, two options were considered:

a. Division of the range of every software variable involved in the validation process into subintervals. The aim of this was to generate test-cases that allow covering as many paths of the Simulink® model as possible. This strategy is followed by commercial software such as Matelo®.
b. Number of states. This is a key factor as it allows modelling in detail the software behavior by using functional states. As depicted in Fig. 2.32, every path of a Simulink® model may be represented by a functional state.

By changing the value of these factors, the code coverage was assessed. To do this, it was checked how many functional states were covered when conducting all test-cases available to validate an SM following the strategies described earlier to generate test-cases. The obtained results are shown in Table 2.41. These figures show how the code coverage increases as the number of states goes up. This fact must be coherent with the functional coverage rate. This point will be analyzed in this section.

The code coverage could be calculated in a more accurate way. However, this implies that two main issues should be taken into account. Firstly, the number of test cases to be performed by using an HIL simulation increases, and the project time frame can be affected. In addition, some use cases are difficult to be simulated when using an HIL bench owing to the HIL model limitations, especially when it comes to SMs linked to advanced driver assistance systems. It must be reminded that these functions need a lot of information exchanged between different ECUs present in the CAN network. Secondly, the number of states should also be increased. However, it cannot be stated that the more states are used, the higher the code coverage is. As shown in Table 2.41, there is a limit at which the code coverage does not increase meaningfully (15 states for a simple function and 75 for a fairly and highly complex function). After analyzing the results, the conclusion was that many test cases were

Table 2.41 Code coverage trend depending on the number of the states (measured in %) (Subinterval = 3)

Number of states\type of SM	1	3	5	8	11	15	18	20	25	31	36	42	48	54	60	68	75	80
Simple	1.5	6.3	14	45	75	95	95.2	95.3	95.3	95.4	95.4	95.4	95.5	95.5	95.6	95.6	95.7	95.7
Fairly-complex	1.19	2.1	2.6	3.5	5.2	8.5	15	17	35.6	36.9	42.3	50	57.1	64.3	71.4	78.6	89.3	89.6
Highly-complex	1.1	1.8	2.2	3.1	4.7	6.7	13.5	15.8	29.8	33.2	38.2	43.2	53.2	58.5	68.7	72.5	80	80.2

redundant. As mentioned above, some states are difficult to reach when using an HIL simulation owing to HIL model limitations.

When it comes to subintervals breakdown, the obtained results are shown in Table 2.42. The main conclusion is the higher the number of subintervals, then the lower code coverage is, as redundancy in test cases occurs. In this research, the authors proceeded to use a fuzzy logic to establish the optimal number of subintervals. More specifically, the speed was considered as low, average, and high, the water cooling temperature low, average, or high, and so on.

Figures 2.33 and 2.34 depict the results in a more visual way.

Finally, it is essential to check the validity of the software EX. Two phases were considered: a validation and a test one. On the one hand, the former consists of verifying test-cases to assess the EX performance depending on the type of SMs under validation (60 for simple SMs, 40 for fairly-complex SMs and 10 for highly complex SMs). On the other hand, the later seeks its acceptance after having tested 30 for simple SMs, 20 for fairly-complex SMs and 5 for highly complex SMs. It is vital to remark that all the points, tested to validate the system, covered all the functional rules. Thus, the functional coverage rate was 100%. In the first phase, a 17.3% error was obtained. In the second one, 0%. As a result, the EX was validated. Table 2.43 shows the results obtained during the first phase.

Before using the EX, an acceptance process is performed, consisting mainly of a series of meetings in which some key factors are assessed. Table 2.44 depicts the most important ones. All the factors assessed cannot be indicated for confidentiality reasons. It is essential to remark that no bug or unexpected behavior of the EX was detected after its validation.

2.4.5.3 Bug Detection

When using one EX, the results obtained after executing the number of test-cases specified in Table 2.2 are shown in Fig. 2.27.

- The Cause-effect technique (automated or not) and the model-based testing one.

  The use of Python scripts is a less efficient technique because it is complicated to make the system reach a specific operating point, especially when dealing with certain SMs, such as those related to after treatment of exhaust gas systems. It must be reminded that these SMs perform multiple complex and accurate calculations. As a result, this technique faces the SM interaction problem. Despite this, a test-case can be executed by using an HIL simulation thanks to dlls. This statement is also true for model-based testing. The fact of reaching specific points remains difficult due to the SM interaction problem.
- The tester-in-the-loop technique and the manual execution one.

  The tester-in-the-loop technique offers better results as a technician or a test engineer can make the system reach a specific operating point. Then, a script is run to use all the necessary manipulations on the HIL model to end the test-case

Table 2.42 Code coverage trend depending on the subintervals and the number of states (measured in %)

Number of states\type of SM	1	3	5	8	11	15	18	20	25	31	36	42	48	54	60	68	75	80
Subinterval = 4																		
Simple	1	4	8	35	68	85	85.2	85.3	85.3	85.4	85.5	85.5	85.6	85.6	85.6	85.7	85.7	85.7
Fairly-complex	1.2	2.1	2.3	3	4.2	7	12	14	30.2	31.2	36	45	52.2	56.3	62.3	70	83.2	83.8
Highly-complex	1.1	1.2	2	2.5	4.1	5	11.6	12.5	18.2	23.5	29.2	34.5	40.2	43.5	48.5	53.1	58.2	58.8
Subinterval = 5																		
Simple	1	3.5	7.2	30	51	72	72.1	72.2	72.5	72.6	72.6	72.6	72.7	72.8	72.8	72.8	72.8	72.9
Fairly-complex	1.18	1.9	2.2	2.8	3.5	6.2	10.8	12.5	25.6	28.6	32	40	48.5	53.5	58	64.5	72.5	72.9
Highly-complex	1.1	1.8	1.9	2.2	3.5	4.2	7	8.5	12.7	16.5	20.1	24.2	27.6	32.5	37.5	42.8	50	50.6

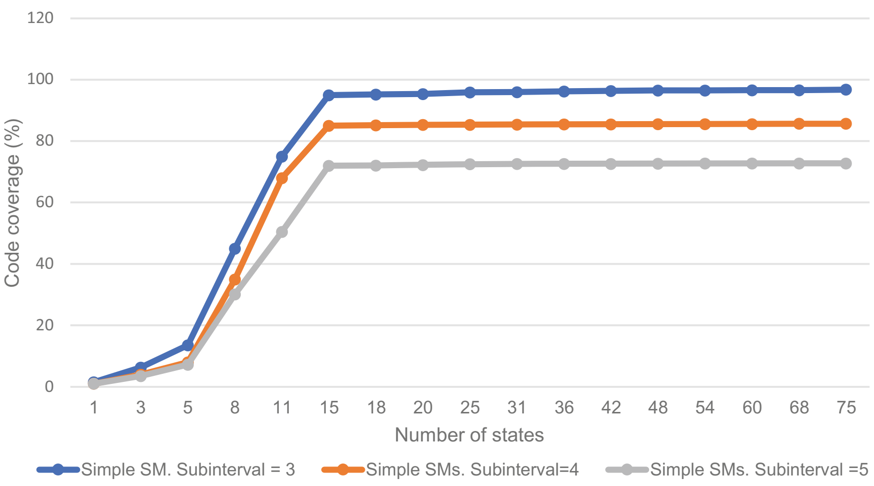


Fig. 2.33 Code coverage rate versus the number of subintervals considered when validating a simple SM

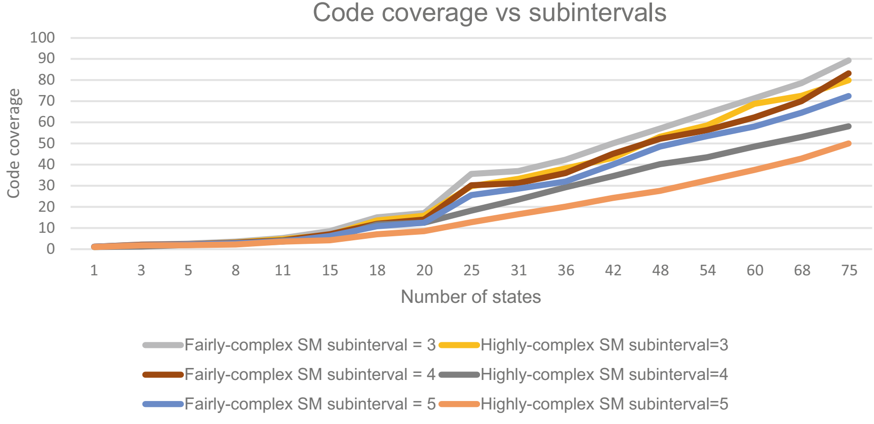


Fig. 2.34 Code coverage trend vs the number of sub-intervals chosen when validating a fairly and highly complex SMs

performance. This statement is also true for manual execution as a technician performs the whole test-case execution.

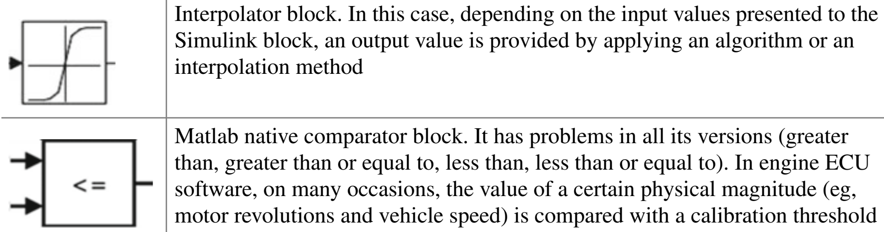
- Using EXs to validate the software

  EXs performance must be analyzed. In the previous research, which is under consideration for publication, the authors probed how the use of a performance EX introduced significant advantages such as the capacity of detecting more bugs than other techniques. The question that might arise is if the addition of a software EX introduces significant improvements, which would justify its implementation.

Table 2.43 Errors detected when validating the EXs

Type of error	Cases	Percentage	Explanation
Wrong syntaxes	10	9.1	Because the rules used to design the EXs are extremely complex, the programmer made coding errors
Incoherence between rules	6	5.5	In some cases of wrong performance of the EX, incoherence between rules was found
Rules not coded or forgotten	3	2.7	This error is due to the same misunderstanding of technical specifications

Table 2.44 Most problematic Simulink blocks



	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
<=	Matlab 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 (eg, motor revolutions and vehicle speed) is compared with a calibration threshold

As shown in Fig. 2.35, the answer is yes, as more six bugs were found. This fact supports the results shown in Table 2.38; the higher the code coverage, the more functional states are checked. Six bugs were detected by using two EXs. Figure 2.36 depicted a classification of these bugs. The term of strategy chosen showed in Fig. 2.15 refers to the ability of testing more paths of the Simulink models, thanks to the use of software EXs that allow to increase the code coverage rate. The rules not considered concept refers to functional states reached during HIL simulations that had not been considered by the design team. The value bugs term refers to certain bugs detected when a Simulink block did not perform some calculations properly (Table 2.44).

2.4.6 Dynamic-Link Libraries

The problem of SM interactions is resolved, thanks to the usage of dlls as proved in this research. It must be remarked that the obtained results are very similar no matter what technique is used provided that dlls are implemented as depicted in Tables 2.45, 2.46, and 2.47.

Several factors must be considered to better understand these results. Firstly, dlls are not needed when using the manual execution as the test engineer can control accurately the automation process. Secondly, the results for "Automated with a Python

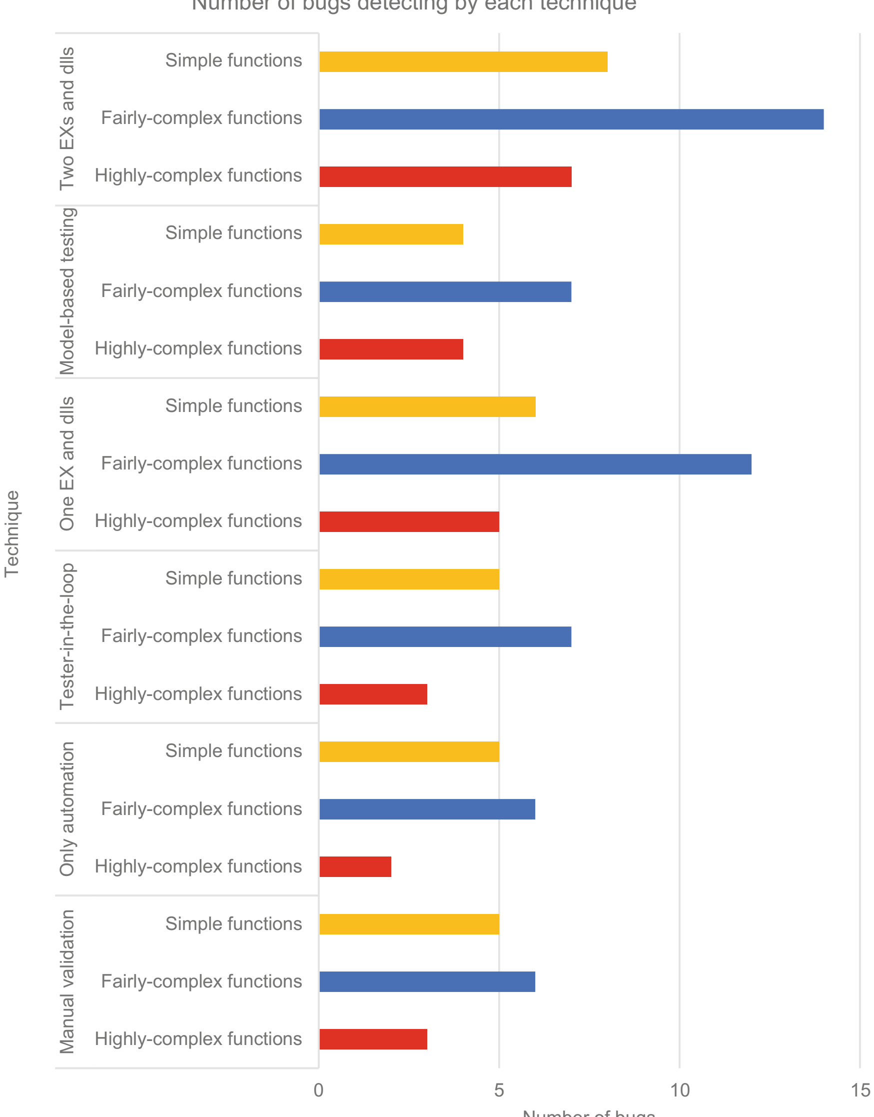


Fig. 2.35 Capacity of bug detection

script and the use of dlls" are representative for no matter what technique is used, which implies that a Python script is run to perform the automation process such as the model- based testing and EXs. Finally, when using dlls, a 100% success rate is not achieved because of HIL model inaccuracies. The HIL model, which represents the vehicle dynamic, is not perfect. Therefore, from time to time, the engine ECU

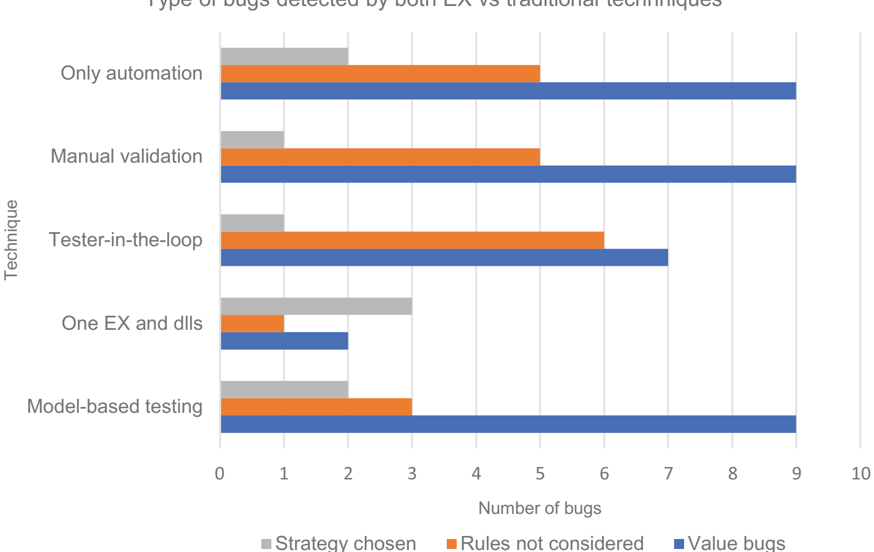


Fig. 2.36 Types of bugs found

Table 2.45 Comparisons of different techniques for validating simple functions

Methodology	Number of cases in which the output value set in the test case was no longer valid	Error rate after 250 simulations (%)	Success rate (%)
Only but without using a dlls	49	19.6	80.4
Tester-in-the-loop	5	10	90
Only automation and the use of dlls	13	5.2	94.8
One EX and dlls	12	4.8	95.2
Two EXs and dlls	13	5.2	94.8

can detect failures, which implies that the test case cannot be properly run despite the dlls usage.

2.4.7 *Limitations*

It is important to emphasize that the use of EXs does not allow the detection of any type of bugs. Indeed, the output provided by the software for a particular variable

Table 2.46 Comparisons of different techniques for validating fairly complex functions

Methodology	Number of cases in which the output value set in the test case was no longer valid	Error rate after 1250 simulations (%)	Success rate (%)
Only but without using a dlls	480	38.4	61.6
Tester-in-the-loop	350	28	72
Only automation and the use of dlls	125	10	90
One EX and dlls	126	10.1	89.9
Two EXs and dlls	124	9.9	90.1

Table 2.47 Comparisons of different techniques for validating highly complex functions

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 (%)
Only but without using a dlls	61	61	39
Tester-in-the-loop	35	35	65
Only automation and the use of dlls	15	15	85
One EX and dlls	15	15	85
Two EXs and dlls	14	14	86

differs from the one expected. However, if this fault does not introduce any serious malfunction, the EXs will not be able to detect it. That is why, the use of the dlls is essential in this methodology. This type of bugs may be present in SMs that perform many calculations.

The reader might think that, in case of bugs in the Simulink model, the software will also contain these errors. 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. In the engine ECU software, when some specific failures are detected, a software reset takes place. If, despite this, the failure still occurs, the ECU stops the car. Figure 2.37 shows a bug found during this research. The dll and the software did not increase a counter properly. The main consequence was that instead of counting until four software resets, they counted until two and the engine was not stopped. In this case, the dll and the software provided the same outputs. However, the EX detected this software bug.

Finally, the limitation associated with this methodology is no different to others that can be proposed as increasing the number of test cases to be conducted to ensure a code coverage of 100% is not compatible with the planning of an engine design project.

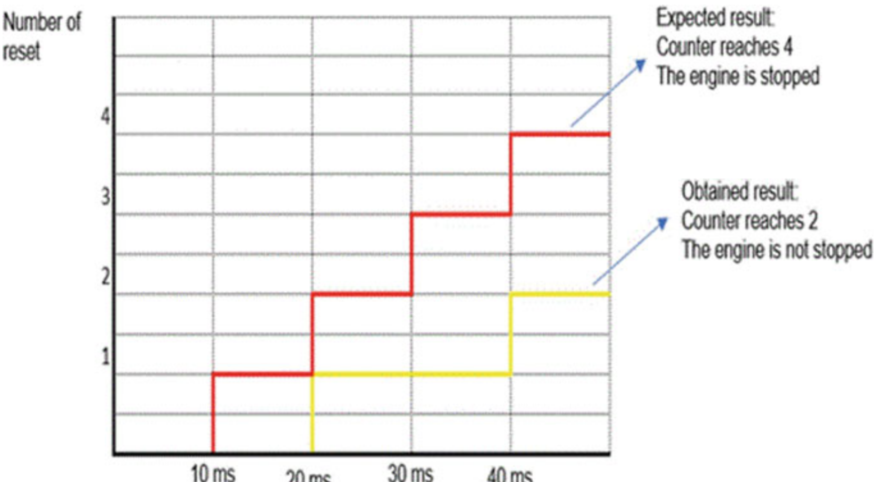


Fig. 2.37 An example of a software bug detected by the EX that could not be detected by using traditional techniques

2.4.8 Threats to Validity

Table 2.48 describes the main variables to be controlled (predictors) to check the influence on the response variables (productivity gain, documentation quality, and bugs). Among these predictors, one can distinguish the sample used in the study described in this chapter, the staff's skills in Python, the SM chosen to be validated, the staff's experience in how the engine ECU operates, the reliability of measures done during the validation, and finally, the quality of the documentation furnished to technician or engineers to validate the software (test description, python scripts, etc.).

All these factors are analyzed in the sensitivity analysis. The authors described in-depth how all these factors impact the time needed to code Python scripts and, therefore, productivity (internal threats). Considering that one of the most important factors to be analyzed in this research is the number of bugs found when using two EXs working in collaboration, it is essential to check how these variables impact this factor. Figure 2.17 shows that the less quality the documents have, the fewer bugs are detected, and therefore, the performance decreases. The quality depends on the sample used in this research, the training in Python, the staff's experience in the engine control unit ECU, and the number of people belonging to the staff. When it comes to external threats, it is of paramount importance to verify if the results can be generalized or if it is applicable to a larger group. Figure 2.38 shows that it can be applied as the quality depends on the number of members of the staff. This statement is based on the fact that the higher the staff is, the more hours can be devoted to improving the quality of documentation. Otherwise, the terms of the project will be prolonged.

Table 2.48 Factors to be controlled when validating the engine ECU

Factor	Description
Sample used in this research	This research was performed in a software validation service that belonged to one of the most important manufacturers in Europe. The staff used in this research is composed of 40 people: 19 engineers and 21 technicians. Each person may have different skills, but this fact was considered in the sensitivity analysis
Training in Python	The more a validation department masters Python, the more sufficient the productivity gain is or the more extensive knowledge of an engine operation the staff can acquire, the less time they require to write the tests. Technicians and engineers having different levels in coding Python or in engine operation knowledge were chosen. Then the influence of all the aforementioned aspects was analyzed in the sensitivity analysis
SM chosen for the research	Not all SMs present in the engine ECU software have the same complexity. It is not possible to draw exactly the same conclusions for a simple SM as for a highly complex one. The SMs were divided into three groups. The fact of not doing this implies that the productivity gain is not properly assessed
Unreliability of measures	All measures were taken in the same conditions. To assure this, a procedure was written, which describes when measures can be accepted and when they must be rejected it. In addition, EXs must be validated Otherwise, the conclusions could be completely random and wrong
Staff's experience in the engine ECU field	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. This research was performed considering different scenarios depending on the staff's training as shown in the sensitivity analysis
Quality of documentations provided to the technician to validate the software	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

2.4.9 Sensitivity Analysis

When automating a test case, it is necessary to make the vehicle reach specific operating conditions. To do this, there are two options: firstly, coding a high-quality script that can control all necessary parameters that could prevent the vehicle from achieving the desired operating point and secondly, the "tester-in-the-loop" concept can be applied. Thus, a technician makes the vehicle reach a desired operating point, and then an automation script performs all subsequent actions to run the test case completely. In this chapter, these SMs were automated in the company subjected to this case study by using Python scripts. The key to achieve this is to code libraries that can carry out specific interaction with the vehicle model interface, such as heating

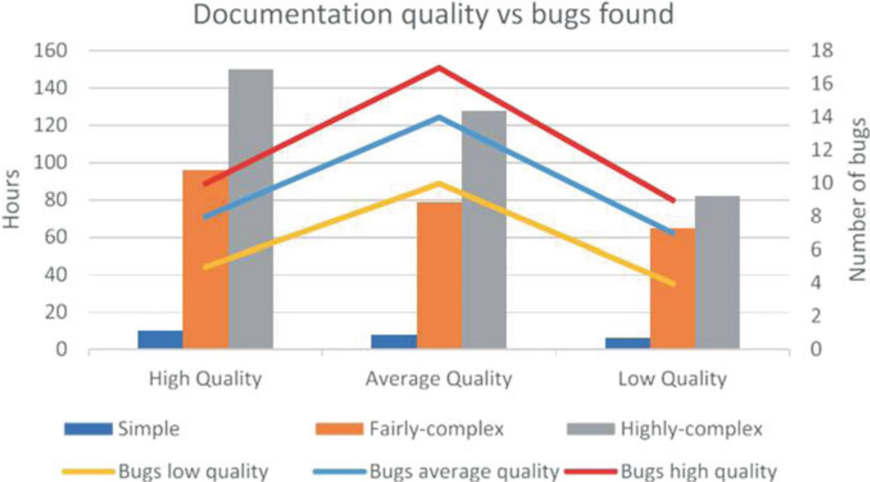


Fig. 2.38 Documentation quality vs bugs found when using Exs

the NO_x probes. Therefore, quick and robust scripts can be coded. However, the time needed to code Python scripts depends on the programmer's experience. As shown in Table 2.49, the staff of the validation software validation service of the company subjected to this case study has been classified as expert, average, and low level when it comes to their experience in Python.

Figure 2.39 depicts the obtained results.

Clearly, training in Python scripts is a key aspect to be taken into account to improve productivity when it comes to software validation.

However, training in Python is not the only key factor to improve the quality of software and time frame of the project. Knowledge about physical phenomena controlled by the SM under validation has a great influence on the time needed to design tests. For example, if a test engineer needs to design tests for validating the urea injection for the nitrogen oxide treatment, if he knows the physical foundation of the function, besides knowing the software architecture, the time needed to design a test case is reduced. To verify this, expert python test engineers were chosen to code python scripts to automate simple, average, and complex functions. However, these engineers had high, average, and low knowledge about the function to be automated. The obtained results are shown in Fig. 2.40. Consequently, in addition

Table 2.49 Staff's training in Python

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

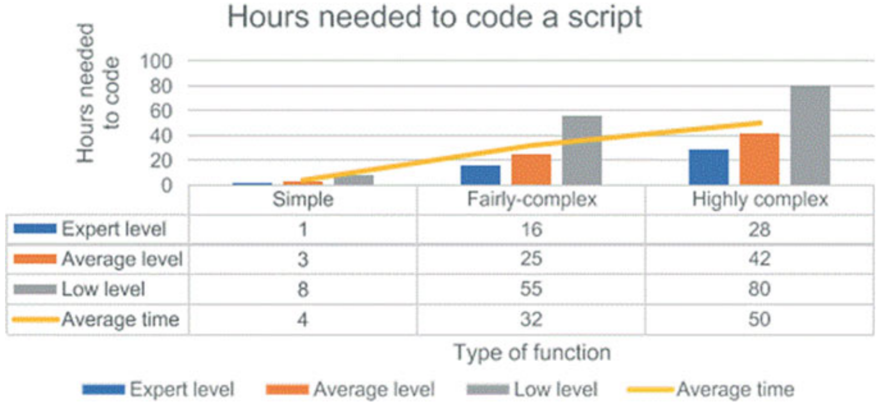


Fig. 2.39 Time needed to code a script depending on staff's training

to SM knowledge, another potential method of improvement is provided by the expertise in the physical phenomena linked to a combustion engine.

The number of engineering hours dedicated to design the tests used during the validation process depends on the final quality of the test documentation provided by the technician. If schedules, notes, and comments are attached, the cost increases. Figure 2.41 shows the total amount of engineering hours spent to design the tests depending on the final quality provided. In this research, the quality was measured by using a checklist built by the validation expert engineer of the powertrain software validation service.

Taking together Figs. 2.40 and 2.41, the total number of hours needed to design the test cases (test-case design and the time needed to code the Python scripts) is

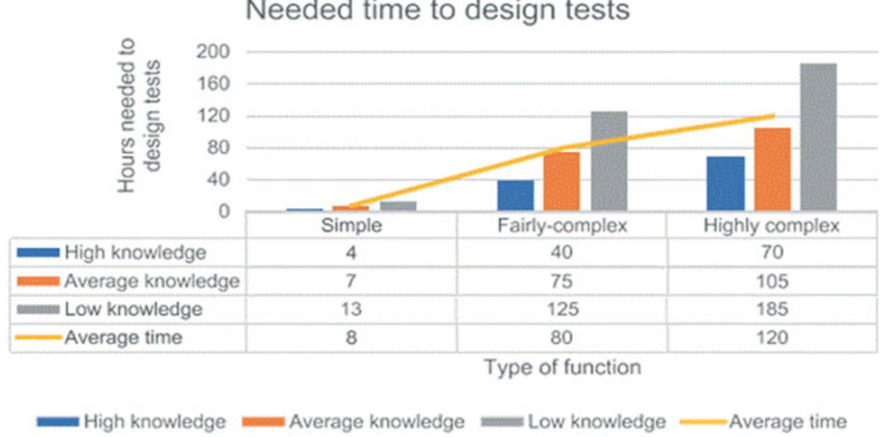


Fig. 2.40 Needed time for designing test cases vs functional and physical knowledge about a SM

Fig. 2.41 Engineering hours spent for test design depending on the type of SM to be validated in black-method

shown in Fig. 2.42. Significant productivity improvements when comparing with the black-box technique can be obtained when the training of the staff is improved: 13.5% for complex functions, 10.9% for fairly complexity functions, and 16.6% for simple functions considering the average knowledge case. These figures are based on the scenario of high Python skills as well as good knowledge of the SM under validation.

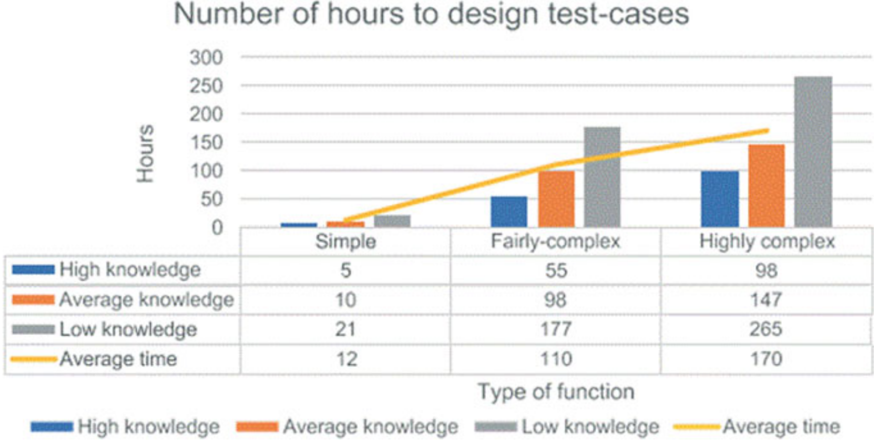


Fig. 2.42 Total number of hours to design the test cases (design and script coding time)

2.4.10 Conclusions

Several issues that the automotive sector must face when validating the electronic control unit (ECU) software are: how to design representative use-cases, how to properly automate the HIL simulation because of the interaction of SMs, and, how to be able to find coding and performance bugs when running a test-case.

This research, conducted at the second most important European car manufacturer, is focused on the software validation of an engine ECU by using dlls and two rule-based EXs, one for detecting performance bugs and the other for finding code bugs. This combination allows the detection of software performance and coding bugs. In this research, the use of dlls and two EXs were compared to other techniques such as the tester-in-the-loop, automation by using Python scripts and a performance EX and automation by using Python scripts without EXs. The results obtained show that dlls and two EXs are able to detect 6 bugs more than the use of dlls and a performance EX can, 14 bugs more than the tester-in-the-loop can, 16 bugs more than the automation by using Python scripts can,15 bugs more than a manual execution can and 14 bugs more than the model-based testing can. Dlls and EXs working in cooperation enhance the code coverage regarding the other techniques. This enhancement depends on the number of states in the functional model used in the EXs and the number of subintervals in which the SM inputs can be divided as shown in this research.

Dlls and Python scripts can be used combined with different techniques such as the using of a performance EXs or two EXs. The obtained results show that the methodology proposed in this research enhances the HIL success rate compared with the tester-in-the-loop technique by up to 6% for simple validation SMs, by 16.8% for fairly-complex SMs and by 18% for highly complex SMs despite the SM interactions. When it comes to automation without using dlls, the methodology proposed in this research enhances the HIL success rate up to 14.4% for simple validation SMs, by 27.4% for fairly-complex SMs and by 47% for highly complex SMs despite the SM interactions.

Even though ES and dlls require more time to be implemented for highly-complex and simple functions, the deadline of the project was met. When it comes to fairly-complex functions there is a productivity gain considering the number of SMs to be tested in an engine ECU software project versus the tester-in-the-loop and manual execution. In addition, the time needed to implement the model-based testing technique is similar to the one needed for two EXs. It must be reminded that the fairly-complex SMs are the majority in the engine ECU software.

References

1. Broy, M. Challenges in automotive software engineering. Proceedings of the 28th International Conference on Software Engineering (pp. 33–42). ACM. 2006
2. Navet, N., & Simonot, F. Automotive Embedded Systems Handbook (1st ed.). CRC Press. Florida, United States, 2008
3. Roychoudhury, A. Embedded Systems and Software Validation (1st ed.). Morgan Kaufmann Publishers, Massachusetts, United States, 2009
4. Gajjar, M. J. Sensor Validation and Hardware-Software Code Design. Mobile Sensors and Context-Aware Computing (1st ed.). Morgan Kaufmann, Massachusetts, United States, 2017
5. Rajan A., Wahl T. CESAR - Cost-Efficient Methods and Processes for Safety-Relevant Embedded Systems (1st ed.), Berlin, Germany, Springer, 2013
6. Oshana, R. Software Engineering for Embedded Systems: Methods, Practical Techniques and Applications (1st ed.), Amsterdam, Netherlands, Elsevier, 2013
7. Oberkampf, W. L., Roy, C. J. Verification and Validation in Scientific Computing (1st ed.). Cambridge University Press, Cambridge, United Kingdom, 2010
8. Westland, J. C. The cost of errors in software development: evidence from industry. *Journal of Systems and Software* 2002, 62(1), 1–9
9. Zaman, N. Automotive Electronics Design Fundamentals (1st ed.), Berlin, Germany, Springer, 2015
10. BOSCH, BOSCH Automotive Electrics and Automotive Electronics (1st ed.). Germany, Robert BOSCH, 2013
11. Garousi, V., & Mäntyla, M. V. A systematic literature reviews in software testing. *Information and Software Technology* 2016, 80, 195–216
12. Kasaju A., Petersen K., & Mantyla M. V. Analyzing an automotive testing process with evidence- based software engineering. *Information and Software Technology* 2013, 55, 1237–1259
13. Jungui Z., Zhiyi Z., Peizhang X., & Jingyu W. A test data generation approach for automotive software. Proceedings of the Conference IEEE 2015
14. Hoffmann A., Quante J., & Woehrle M. Experience report: White box test-case generation for automotive embedded software. Proceedings of the IEEE Ninth International Conference on Software Testing, 2016
15. Saglietti, F., Oster, N., & Pinte. F. White and grey-box verification approaches for safety and security critical software systems. *Information Security Technical Report* 2008, 13(1), 10–16
16. Wernick, P., & Lehman, M. Software process white box modelling for FEAST/1. *Journal of System and Software* 1999, 46(2–3), 193–201
17. Awedikian R., & Yannou, B. Design of a validation test process of an automotive software. *International Journal on Interactive and Manufacturing 2010*, 4(4), 259–268
18. Conrad, M. http://drops.dagstuhl.de/opus/volltexte/2005/325/, 2005
19. Chunduri, A. http://www.diva-portal.org/smash/get/diva2:945731/FULLTEXT02, 2005
20. Skruch, P., Buchala, G. Model-based real-time testing of embedded automotive systems. SAE *Int. J. Passeng. Cars – Electron. Electr. Sys* 2014. 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. Proceedings of the Embedded Real Time Software and Systems Conference, 2016
22. All4Tec, http://www.all4tec.net/MaTeLo/homematelo.html (last accessed on 10/03/2021)
23. Ilic, V., Popic, S., & Kovacic, M. Data flow in automated testing of the complex automotive electronic control units. *IEEE Instrumentation & Measurement Magazine* 2016
24. Keller, R., Alink, T., Pfeifer, C., Eckert, C. M., Clarkson, P. J., & Albers, A. Proceedings of the International Conference on Engineering Design, 2017
25. Köhl, S., Lemp. D., & Plöger, M. ECU network testing by hardware-in-the-loop simulation. *ATZ Worldwide* 2003*,* 105(10), 10–12
26. National Instrument, http://www.ni.com/white-study/10343/en/ (last accessed January 2020)

27. Winemantech, https://www.winemantech.com/services/hardware-in-the-loop-test-systems/ (last accessed January 2020)
28. Petrenko, A., Nguena, T., & Ramesh, S. Model-based testing of automotive software: some challenges and solutions. Proceedings of the 52th Congress ACM/IEEE Design Automation Conference, 2015
29. Matlabcentral File exchange. https://www.mathworks.com/matlabcentral/fileexchange/9709-from-simulink-to-dll-a-tutorial (last accessed January 2020)
30. NCA Software Product etas. https://www.Etas.Com/En/Products/Inca_Software_Products.Php (last accessed January 2019)
31. DSPACE. Simulator Hardware. https://www.Dspace.Com/En/Inc/Home/Products/Hw/Simulator_Hardware/Dspace_Simulator_Full_Size.Cfm (last accessed February 2022)
32. DSPACE Experiment and visualization. https://www.dSpace.com/en/inc/home/products/sw/experimentandvisualization/controldesk.cfm (last accessed February 2022)
33. DSPACE. Test Automation Software. https://www.dspace.com/en/pub/home/products/sw/test_automation_software/automdesk.cfm (last accessed February 2022)
34. Krûguer, M., Straube, S., Middendorf, A., Hahn, D., Dobs, T., Lang, K.D. Requirements for the application of ECUs in e-mobility originally qualified for gasoline cars. *Microelectronics Reliability* 2016, 64, 140–144
35. Gajjar, M.J. *Mobile Sensors and Context-Aware Computing*. Massachusetts: Morgan Kaufmann Publishers, Massachusetts, United States, 2017
36. Lockledge, J.C., Salustri, F.A. Defining the Engine Design Process. *Journal of Engineering design,* 10, 109–124. https://doi.org/10.1080/095448299261344 (last accessed March 2022)
37. Raikwar, S., Jijyabhau, L.W., Arun Kumar, S., Sreenivasulu Rao, M. Hardware-in-the-Loop test automation of embedded systems for agricultural tractors. *Measurement* 2019, 133: 271–280
38. Plummer, A.R. Model-in-the-loop testing, Proceedings of the Institution of Mechanical Engineers Part I *Journal of Systems and Control Engineering* 2006, 220 (3), 183–199
39. Zhan, Y., Clark, J.A. A search-based framework for automatic testing of MATLAB/Simulink models. *Journal of Systems and Software* 2008, 81(2), 262–285
40. Vivas, J.L., Agudo, I., Lopez, J. A methodology for security assurance-driven system development. *Requirements engineering* 2011, 16, 55–73
41. Martin, H., Ma, Z. , Schmittner, C. , Winkler, B., Kreiner, C. Combined automotive safety and security pattern engineering approach. *Reliability Engineering & System Safety* 2020, 198, Article 106773
42. Haghighatkhah, A., Banijamali, A., Pekka Pakanen, O., Oivo, M., Kuvaja, P. Automotive software engineering: A systematic mapping study. *Journal of Systems and Software* 2017, 128, 25–55
43. Hooshyar, H., Mahmood, F., Vanfretti, L., Baudette, M. (2015). Specification, implementation, and hardware-in-the-loop real-time simulation of an active distribution grid. *Sustainable Energy, Grids and Networks* 2015, 3, 36–51
44. National Instrument (2019). https://www.ni.com/fr-fr/innovations/white-studys/17/what-is-hardware-in-the-loop-.html (Accessed on 3 March 2020)
45. Ortega-Cabezas, P.M., Colmenar-Santos, A., Borge-Diez, D., Blanes-Peiró, J.J. Application of Rule-Based Expert Systems and Dynamic-Link Libraries to Enhance Hardware-in-The-Loop Simulation Results, *The Journal of Software 2019,* 14(6), 265–292
46. Meloa, S.M., Carver, J.C., Souza, P.S.L., Souza, S.R.S. Empirical research on concurrent software testing: A systematic mapping study. *Information and Software Technology,* 2019, 105, 226–251
47. Vandi, G. , Nicolò, C., Corti, E., Mancini, G. , Moro, D., Ponti, F., Ravaglioli, V. Development of a Software in the Loop Environment for Automotive Powertrain System. *Energy Procedia* 2014, 45, 789–798
48. Garousi, V., Felderer, M., Kilicaslan, F.N. (2009). A survey on software testability. Cornell University. https://arxiv.org/abs/1801.02201 (last accessed on 17 January 2020)
49. Walia, G.S., Carver, J. C. A systematic literature review to identify and classify software requirement errors. *Information and Software Technology* 2009, 51(7), 1087–1109

50. Ågren, S.M., Knauss, E., Heldal, R., Pelliccione, P., Malmqvist, G., Bodén, J. The impact of requirements on systems development speed: a multiple-case study in automotive, *Requirements Engineering* 2019*,* 24, 315–340
51. Dos Santos, J., Martins, L.E.G, de Santiago Junior, V.A, Povoa, L.V., dos Santos, L.B.R. Software requirements testing approaches: a systematic literature review, *Requirements Engineering* 2019*,* https://doi.org/10.1007/s00766-019-00325-w
52. Abadeh, M.N. Performance-driven software development: an incremental refinement approach for high-quality requirement engineering, *Requirements Engineering* 2020*,* 25, 95–113
53. Feldhütter, A., Segler, C., Bengler, K. Does Shifting Between Conditionally and Partially Automated Driving Lead to a Loss of Mode Awareness? In N. Stanton (Ed.), Advances in Human Aspects of Transportation. AHFE 2017. *Advances in Intelligent Systems and Computing* 2018, 597, pp. 730–741
54. ISO. Cybersecurity standard (2019). https://www.iso.org/standard/70939.html (last accessed on 20 September 2020)
55. Utesch, F., Brandies, A., Pekezou, P., Schiessl, F., Schiessl, F. Towards behaviour based testing to understand the black box of autonomous cars. *European Transport Research Review* 2020*,* 12, 48
56. Huang, W.L., Wang, K. Ly, Y., Zhu, F. Autonomous Vehicles Testing Methods Review. IEEE *19th International Conference on Intelligent Transportation Systems (ITSC)* 2016, pp. 163–168
57. Riedmaier, S., Ponn, T., Ludwig, B., Shick, F. Diermeyer, F. Survey on Scenario-Based Safety Assessment of Automated Vehicles. *IEEE Access* 2020*,* 8, 87456–87477
58. dSpace http://www.cokesen.com/resimler/1521204313_Dokuman1.pdf (last accessed 10 September 2020)
59. Möller, D., Haas, R. Guide to Automotive Connectivity and Cybersecurity. Berlin, Germany, Springer
60. El-Rewini, Z., Sadatsharan, K., Flor, D., Siby, S., Plathottam, J., Ranganathana, P. Cybersecurity challenges in vehicular communications. *Vehicular communications* 2020, 23, 100214
61. Vector. https://www.vector.com/int/en/know-how/technologies/safety-security/automotive-cybersecurity/#c2941(last (accessed on 10 September 2020)
62. Placho, T., Schmittner, C., Bonitz, A., Wana, O. Management of automotive software updates. *Microprocessors and microsystems* 2020, 78, 103257
63. Koegel, M., Wolf, M. (2018). Auto update – safe and secure over-the-air (SOTA) software update for advanced driving assistance systems. Berlin, Germany, Springer
64. ISO. Autonomous driving safety standard. https://www.iso.org/standard/70918.html (last accessed on 20 September 2020)
65. Banish, G. Engine Management: Advanced Tuning. Minnesota: Cartech, 2007
66. Garousi, V., Mäntylä, M.V. A systematic literature review of literature reviews in software testing. *Information and Software Technology* 2016*,* 80, 195–216
67. Kasoju, A., Petersen, K., Mäntylä, M.V. Analyzing an automotive testing process with evidence-based software engineering, *Information and Software Technology* 2013*,* 55(7), 1237–1259
68. Matelo® Software. https://www.all4tec.com/ (last accessed on 7 February 2020)
69. BOSCH. *BOSCH Automotive Electrics and Automotive Electronics* (1st ed.). Robert BOSCH. Germany. 2013
70. IEEE http://ieeexplore.ieee.org/document/4344112/ (last accessed June 2018)
71. Mariani L, Pezze M, Zuddas D. Recent advances in automatic black-box testing. *Adv. Comput*. 2015; 99: 157–193
72. Engström E, Runeson P, Skoglund M. A systematic review on regression test selection techniques. *Inf. Softw. Technol.* 2010; 52(1): 14–30
73. Linderman U, Maurer M, Braun T. Structural Complexity Management. 1st ed. Springer, Berlin, Germany. 2009
74. Yoo S, Harman M. Pareto efficient multi-objective test case selection. Proceedings of the ACM/SIGSOFT. *International Symposium on Software Testing and Analysis* 2007. ACM. 2, 140-150
75. Zhou J., Zhang Z., Xie P., Wang J. A test data generation approach for automotive software. IEEE International Conference on Software Quality, Reliability and Security. 2015

76. Sopan-Barhate, Effective test strategy for testing automotive software. International Congress of Electronic Instrumentation and Control. 2015
77. Xing Y, Gong Y, Wang Y, Zhang X. Intelligent test-case generation based on branch and bound. *The Journal of China Universities of Posts and Telecommunications*. 2014; 21(2): 91-97
78. Zhang W, Yang Y, Wang Q. Using Bayesian regression and EM algorithm with missing handling for software effort prediction. *Inf. Softw. Technol*. 2015; 58: 58-70
79. Zheng J. Predicting software reliability with neural network ensembles. *Expert Systems with Applications,* 2009; 36(2, Part 1): 2116-2122. 29
80. Conrad, M. http://drops.dagstuhl.de/opus/volltexte/2005/325/ (last accessed January 2017)
81. Chunduri, A. (2016) http://www.diva-portal.org/smash/get/diva2:945731/FULLTEXT02 (last accessed January 2018)
82. Raffaëlli L., Vallée F., Fayolle G., Armines A, de Souza P., Rouah X., Pfeiffer M. Géronimi S. Pétrot F. Ahiad S. Embedded Real Time Software and Systems Conference. 2016
83. All4Tec. http://www.all4tec.net/MaTeLo/homematelo.html (last accessed November 2017)
84. Perez, Y.M., Marin, H.A.P., Bedoya, A.E. http://www.revistaieeela.pea.usp.br/issues/vol14issue5May2016/14TLA5_41EspinosaBedoya.pdf (last accessed January 2018)
85. Mechanical Simulation. https://carsim.com/products/realtime/index.php (last accessed January 2018)
86. National Instrument http://www.ni.com/white-study/10343/en/ (last accessed November 2017)
87. Petrenko A. Nguena-Timo, Ramesh S. Model-based testing of automotive software: some challenges and solutions. 52th Congress ACM/IEEE Design Automation Conference. 2015
88. Tatar M, Mauss J. Systematic Test and Validation of Complex Embedded Systems. Toulouse, France: Embedded Real Time Software and Systems (ERTS 2014); 2014