



# 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

Pedro Miguel Ortega-Cabezas<sup>1</sup> · Antonio Colmenar-Santos<sup>1</sup> · David Borge-Diez<sup>2</sup> · Jorge Juan Blanes-Peiró<sup>2</sup> · Jorge Higuera-Pérez<sup>1</sup> · Eric Alcaide<sup>1</sup>

Accepted: 9 December 2021 / Published online: 15 January 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

## Abstract

The number of electronic control units (ECU) installed in vehicles is increasingly high. Manufacturers must improve the software quality and reduce cost by proposing innovative techniques. 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 (hardware-in-the-loop 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 hardware-in-the-loop (HIL) simulations and by using prototype vehicles. The second one tunes the inputs of the software module (SM) under validation to cover these parts of the Simulink® models. The usage of dynamic-linked libraries (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% compared to 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.

**Keywords** Engine control unit software testing · Genetic algorithms · Model-based testing · Black-box testing · Cause-effect technique

---

✉ Pedro Miguel Ortega-Cabezas  
pedro.ortegacabezas@gmail.com

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

<sup>2</sup> Department of Electrical and Control Engineering, Universidad de León, León, Spain

# 1 Introduction

## 1.1 Background

Both the number of electronic control units (ECUs) installed in vehicles and their complexity are increasing (Krûguer et al., 2016; Gajjar, 2017; Rajan & Wahl, 2013). Thus, manufacturers must assure software quality and reliability (Kasoju et al., 2013). The software and hardware validation of an engine ECU is performed by using the hardware-in-the-loop (HIL) simulation and prototype vehicles (Lockledge & Salustri, 2010). The HIL simulation has several advantages as no vehicle with all the ECUs updated with the latest software version is necessary. Secondly, the ECU behaviour in the network can be checked by analysing the frames transmitted and received when conducting a 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 that must have all ECUs properly updated: ESP (electronic stability program), ADAS (advanced driver-assistance system), ATCU (automatic transmission control unit), etc.

This research is focused on one of the most complex software installed in vehicles: the engine ECU software. It proposes the usage of genetic algorithms (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 software module (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.

## 1.2 Related works

The engine ECU software development comprises three phases (V-cycle development): implementing models based on Simulink® models to control the engine performance, generating C-code and checking the final integration of the software into the hardware. During the entire process, the engine software completes three levels of testing: model-in-the-loop (MIL), software-in-the-loop (SIL) and HIL simulations as stated by Raikwar et al. (2019). 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 (Plummer, 2006; Vivas et al., 2011; Zhan & Clark, 2008). During the HIL simulation, the integration between software and hardware is tested with a controller (i.e. the engine ECU and its software) which controls the system that imitates the engine behaviour (HIL simulator) (Martin et al., 2020; Haghightatkah et al., 2017; Hooshyar et al., 2015; National Instrument, 2020; Ortega-Cabezas et al., 2019a, b). In addition, prototype vehicles are used to test some functions which cannot be completely validated when using HIL simulations such as ADAS (Melo et al., 2019). 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 (Vandi et al., 2014).

Currently, software is tested based on software, architecture and system requirements (Raikwar et al., 2019). At this point, how to test software requirements is a key point discussed in some standards such as ASPICE (2020). Software testability depends on five factors such as requirements, built-in test capabilities, the test-case design, the test support environment and the software process in which testing is conducted (Garousi et al., 2018). 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 (Walia & Carver, 2009; Ågren et al., 2019). Dos Santos et al. (2019) carried out a detailed analysis about software requirements testing approaches such as requirement-driven testing (Abadeh, 2020).

Regarding the engine ECU, test-cases are derived from specification-based testing, regression tests and experience-based testing (Linderman et al., 2009; Raikwar et al., 2019; Roychoudhury, 2009; Sun et al., 2016; Yi et al., 2016). Focusing on the software and hardware integration by using HIL simulations, when a test-case is run, the obtained result is compared to the expected one to check whether the software has run properly. The black-box technique has been widely used in the automotive sector (Garousi & Mäntylä, 2016; Melo et al., 2019). In this technique, test-cases are based on engineers' experience, which usually involves gaps and test-redundancies. Chunduri (2016) proposes to work on three factors: enhancing the function requirements specification, establishing traceability across test levels and obtaining comprehensive function test-coverage information. Research has been focused on time reduction and code/functional coverage improvement. Zhou et al. (2015) proposed the optimised use of symbolic simulation aiming at reducing the time required to generate a test-case. Sopan-Barhate (2015) presented a theory about designing representative test-cases based on priorities by using orthogonal arrays testing.<sup>1</sup> Raffaëlli et al. (2016) presented research focused on the use of functional models to model a function combined with the commercial software Matelo® (2018) in order to assess the functional coverage accurately as all branches of the model could be tested. GAs have been used in software testing. Esfandyari and Rafe (2018) proposed a method based on a variable t-way minimal test suite generation approach. Sharma et al. (2013) did a survey on software testing techniques using GAs showing that GAs could be applied to white-box, grey-box and black-box testing, regression and model-based testing. The results showed that their performance and testing results were improved as confirmed by Sharma et al. (2016). Concolic testing has also been applied in many industries to test software. As stated by Kim et al. (2020), "*concolic testing combines dynamic concrete execution and static symbolic execution to explore all possible execution paths of a target program, which can achieve high code coverage*". Some constraints found by Kim et al. (2020) were based on the current automatic test generation techniques and tools not supporting symbolic settings for function pointers due to the limitations of solvers of concolic testing. To solve this issue, they proposed another technique which reduces the time needed for manual validation. In addition, the code coverage is increased. In all the techniques

---

<sup>1</sup> "*Orthogonal array testing is a black box testing technique that is a systematic, statistical way of software testing. It is used when the number of inputs to the system is relatively small, but too large to allow for exhaustive testing of every possible input to the systems. It is particularly effective in finding errors associated with faulty logic within computer software systems*" (Delius, 2004).

already described, the relationship between time needed to run test-cases and the most adequate means is not considered when generating test-cases.

The HIL automation process is mainly based on black-box techniques such as those reported by Köhl et al. (2003). Petrenko et al. (2015) reported the main problems and solutions associated with the software validation in the automotive sector. Their solution was based on the “tester-in-the-loop” concept, in which the test engineer leads the system to a desired operation point, considered as a crucial operation point aimed at assuring the correct execution of the test-case to check that the software runs as expected. Once the crucial point is reached, a series of automated actions is performed to finish the test-case execution. Tatar and Mauss (2014) proposed to use a virtual platform instead of HIL simulations. However, bugs linked to the software integration on the hardware are not found.

Concerning autonomous driving, ISO 26262 only covers functional safety when a failure occurs but not when there is no system failure. Therefore, the safety of the intended functionality (SOTIF) ISO 21448 came out (Feldhütter et al., 2018; ISO, 2019). Some key topics to validate the software are focused on 3D modelling and sensor buildings. The former aims to create a realistic environment while the latter consists of modelling and testing sensors among others (Utesch et al., 2020). Huang et al. (2016) detail in their research the main trends to validate software such as software testing, simulation testing, x-in-the-loop testing and driving test in real conditions. Riedmaier et al. (2020) describe an important method to test the software: the scenario-based approach which allows individual traffic situations to be tested by using virtual simulations. Other approaches such as formal verification, a function-based approach, real-world testing, shadow mode testing<sup>2</sup> 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 (dSpace, 2018).

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 (Möller & Hass, 2019). The factors which make automotive security more efficient include the integration of appropriate solutions such as firewalls, protecting communications, authenticating communications and encrypting data (El-Rewini et al., 2019; Vector, 2019). These topics are important to deliver performance such as on-the-air software update and V2X communication (Placho et al., 2020; Koegel & Wolf, 2018). As detailed by McAfee (2016), the scope of cybersecurity involves the distributed security architecture, hardware and software security and finally network security. Standards such as ISO/SAE 21,434 will help the automotive sector to implement solutions for effective compliance with cybersecurity requirements as today’s knowledge sharing is inadequate (Morris et al., 2020; ISO, 2020).

This paper is organised as follows. “Section 2” describes the method used to detail how engine ECU software models are developed as well as how GAs and test-cases are implemented. “Section 3” presents the results showing the different key performance indicator (KPI) values obtained when using different techniques to test the software. “Section 4”

---

<sup>2</sup> Wang and Winner present a method, in which the automated driving function is executed passively, in series production vehicles, which is sometimes known as the shadow mode (Riedmaier et al., 2020; Wang & Winner, 2019).

discusses the results obtained based on topics such as test-case formulation, means optimisation, test-case automation, among others. “Section 5” analyses the validity of this research by performing a sensitivity analysis. Finally, “Sect. 6” draws the main conclusions of GAs compared to other techniques such as the model-based testing and the black-box testing that are widely used in the automotive sector.

## 2 Methods

This section describes in detail the method used in this study.

### 2.1 Simulink models

The SMs are composed of multiple complex Simulink® models and subsystems. Figure 1 shows an example of the internal structure of the SM linked to the NO<sub>x</sub> 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 vapour. Afterwards, the NO<sub>x</sub> probes start to heat until reaching the required temperature to measure NO<sub>x</sub> ppm present in the exhaust gas

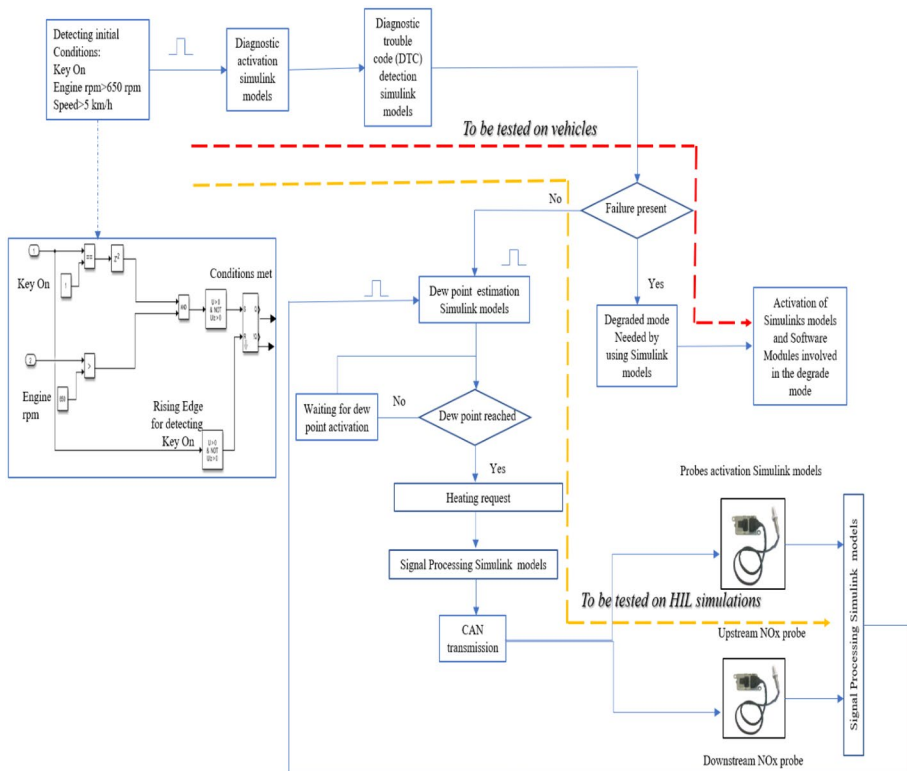
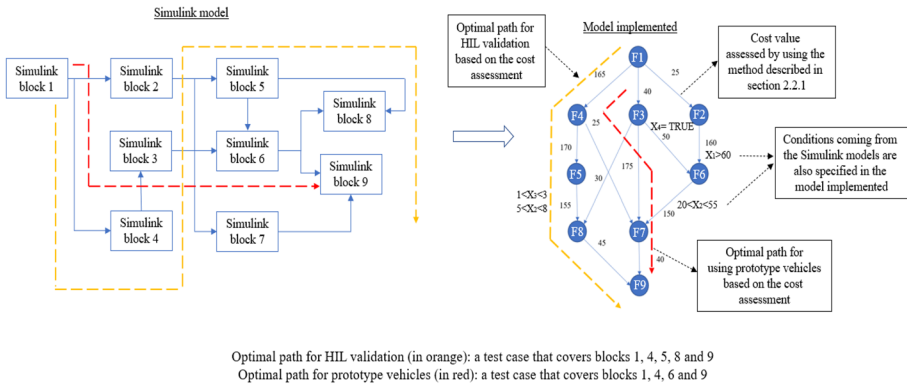


Fig. 1 Example of model and activation conditions



**Fig. 2** Example of model and activation conditions

pipe. In this study, all models were transformed into models based on nodes ( $F_x$ ) which represent different low-level system states<sup>3</sup> and relations between them (Fig. 2). These relations are established using transitions that are composed of two factors: costs and conditions. The costs are the effort needed to change states and the conditions come from the Simulink® models (Fig. 2). It is important to remark that only under certain values of the inputs of the SMs it is possible to change states. When designing test-cases, it must be determined which parts of the Simulink® model should be validated by using the HIL simulation or prototype vehicles, and how the inputs are tuned. “Section 2.2” describes in-detail how GAs work to do this.

## 2.2 How GAs work together

As described earlier, a complex Simulink® model is transformed into a functional model (Fig. 2). Each transition to go from one state to another has a cost set, as described later in “Sect. 2.2.1”. In addition to this, it is only possible to change states under certain conditions specified in the Simulink® models (Fig. 2), i.e. when the input variables of the SM under validation reach specific values. Consequently, the problem to be solved is to find values for the inputs of the SM under validation to find the optimal path<sup>4</sup> of the functional model that minimises the time needed to validate an SM when using the HIL simulation.<sup>5</sup> Figure 3 depicts the whole process.

- Phase 1. A first GA, called GA2, generates populations that consist of values for the inputs of the SM under validation, i.e. vectors, as shown in Fig. 3.
- Phase 2. A second GA, called GA1, generates populations for each vector based on functional states, but they must start always in  $F_1$  (initial state) and end in  $F_n$  (the final

<sup>3</sup> Low system states are functional states at low level. Consequently, the functional state cannot be detected by the driver.

<sup>4</sup> The optimal path is a part of the model (Fig. 2) that is composed of functional states. These states meet the conditions to change states and the path has the lowest cost.

<sup>5</sup> The reader can find a beta version of the code used in this research. See Appendix.

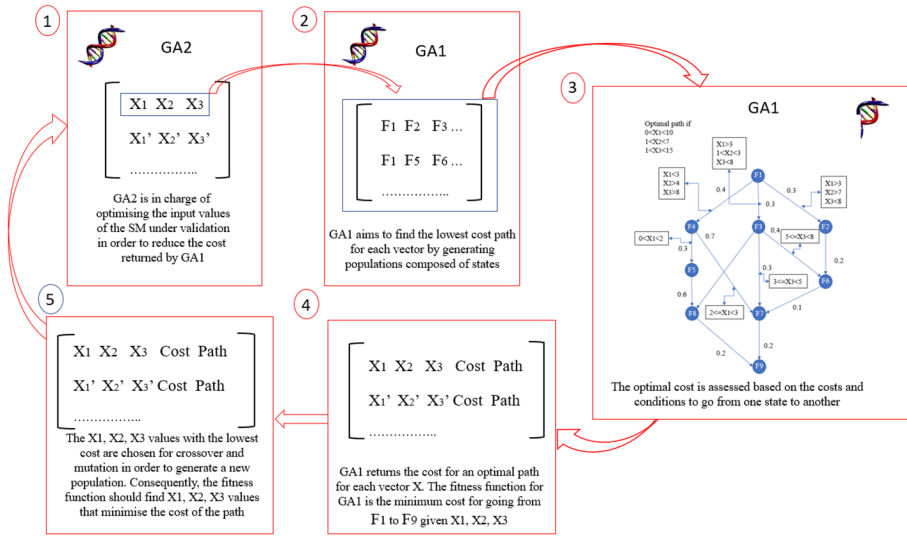


Fig. 3 How GAs work together

state). The aim of this GA1 is to find the paths with the lowest cost by considering the vectors provided by GA2. As stated earlier, only under certain values of the inputs of the SM is it possible to change states (Figs. 2 and 3).

- Phase 3. The optimal path for each vector provided by GA2 is found by using mutation and crossover operators, as described in the next sections. Only under certain values of the SM is it possible to go from one state to another one, as shown in Fig. 3 (phase 3).
- Phase 4. When GA1 finds the optimal path for each vector (if possible), it will return the cost and the states covered for the optimal paths.
- Phase 5. Only the vectors with the lowest cost will be retained by GA2, and they will be used to generate more populations by employing mutation and crossover operators, as described in the next sections. Consequently, GA2 becomes an optimiser of the input of the SM under validation by using the optimised paths provided by GA1.

Therefore, GA1 aims to minimise the cost for specific values of the inputs of the SM under test, and GA2 tries to find the values for the inputs of the SMs that minimise the cost of the model shown in phase 3 (Fig. 3). This implementation could have been done using one GA, but the authors decided to implement 2 GAs: one of them works with values for the inputs of the SM and the other one operates functional states.

### 2.2.1 GA1: path finder

When GA1 receives a vector, it starts to look for the optimal path. To do this, this GA must consider the cost of going from one state to another. This cost is set as follows. Several key factors must be considered when deciding the most adequate means to validate the software such as the ones shown in Table 1 (Banish, 2007; Haghighatkhah et al., 2017).

All factors shown in Table 1 are assessed when going through the states of the models to define the optimal means to validate the SM under validation. Consequently, the values

**Table 1** Factors considered to assess the fitness function

Factor	Description
<i>Tuning activities</i>	Some SMs must be tuned before their 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
<i>Time needed to go from one state to another one</i>	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
<i>Dependency on ECUs</i>	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
<i>Risk level</i>	The automotive safety integrity level (ASIL) is a system which classifies the potential risks in a vehicle when it is operated 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 levels are established. Regarding the engine ECU, the software must guarantee the passengers' and vehicle's safety in a dangerous situation. Depending on the ASIL values (A, B, C, D), the level of risk will be different
<i>Feedback from other projects</i>	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. Meanwhile, client complaints are also considered in such a way that if a project receives a client complaint, it is immediately transmitted to other projects, which could galvanise all the necessary actions

obtained are put into the model based on states (Fig. 2). This process is composed of two phases:

- Phase 1. A multidisciplinary team assesses these factors aiming at determining the cost of each path by using the process depicted in Fig. 4. As a result, a model with the whole cost set for each transition is obtained (Fig. 2).
- Phase 2. This GA chooses the most adequate means to validate the SM by assessing the cost function given by Eq. (1):

$$\text{Fitness function} = \sum_{i=1}^{i=n} F_i \quad (1)$$

where  $F_i$  is the cost of reaching the  $F_i$  state and  $\sum_{i=1}^{i=n} F_i$  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. This value was chosen as follows. As shown in Table 2, when



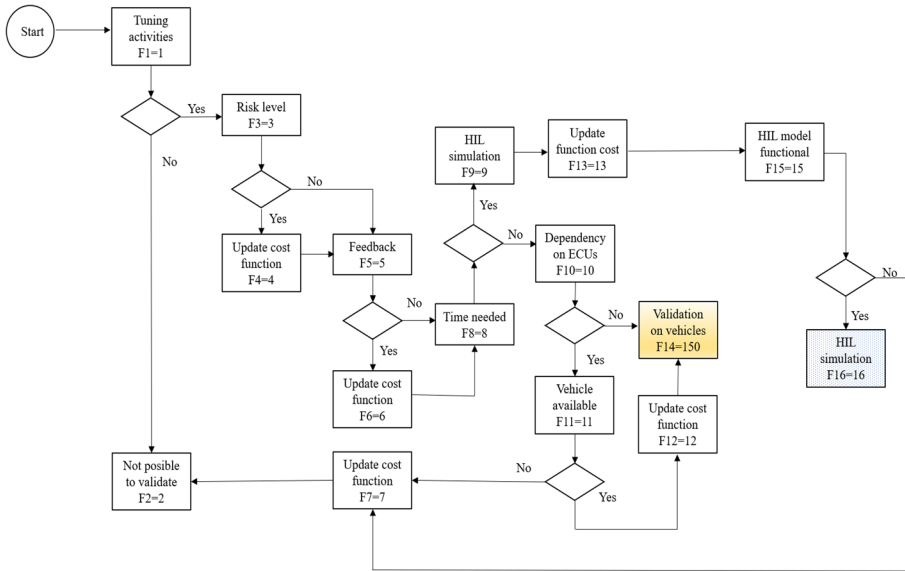


Fig. 4 Factors indicated in Eq. (1)

using the HIL simulation, the fitness function score ranges between 70 and 80 based on Eq. (1). When using prototype vehicles, the maximum and minimum values of the fitness function must be greater than 80. Otherwise, if the fitness function is 64 for example, it would be impossible to know which the optimal means to use during the validation are. Therefore,  $F_{14}$  should be higher than 53. In this study,  $F_{14}$  was 150 just in case new states should be added in the future. Like this, if the fitness function value is lower than 80, HIL simulations must be performed. If the fitness function value is higher than 80, prototype vehicles must be used.

Each path is composed of different states. The paths which contain state  $F_{16}$  more frequently are considered as the optimal ones to be validated by using an HIL simulation.

Table 2 Maximum and minimum values of the fitness function

HIL				Prototype vehicles			
State	Fitness function value. Max value	State	Fitness function value. Min value	State	Fitness function value. Max value	State	Fitness function value. Min value
$F_1$	1	$F_1$	1	$F_1$	1	$F_1$	1
$F_3$	4	$F_3$	4	$F_3$	4	$F_3$	4
$F_4$	8	$F_5$	9	$F_4$	8	$F_5$	9
$F_5$	13	$F_8$	17	$F_5$	13	$F_8$	17
$F_6$	19	$F_9$	26	$F_6$	19	$F_{10}$	27
$F_8$	27	$F_{13}$	39	$F_8$	27	$F_{14}$	$27 + F_{14}$
$F_9$	36	$F_{15}$	54	$F_{10}$	37		
$F_{13}$	49	$F_{16}$	70	$F_{11}$	48		
$F_{15}$	64			$F_{12}$	60		
$F_{16}$	80			$F_{14}$	$60 + F_{14}$		

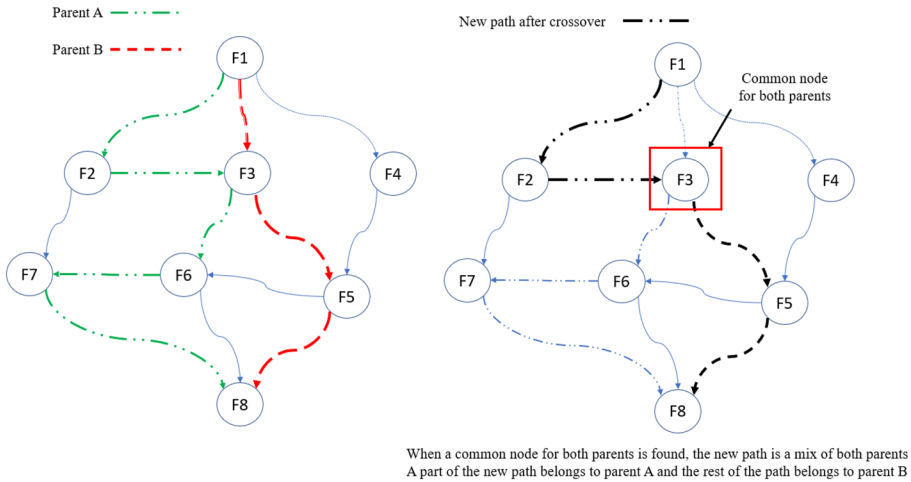


Fig. 5 Crossover operation

Otherwise, they should be validated by using prototype vehicles. The test-engineer can collect important information when analysing the states covered once the optimal path is assessed, such as dependency on other ECUs, feedback from other projects, etc.

To generate new populations, the previous populations with the lowest costs are chosen. Afterwards, the mutation and crossover operations are applied as follows:

- **Crossover.** The process of how GA1 determines the optimal path is depicted in Fig. 5. When a vector is received, several possible paths are generated, and the costs are assessed. The paths with the lowest cost are chosen. The crossover process consists of chosen paths two by two as parents with a common node, and a new possible path is

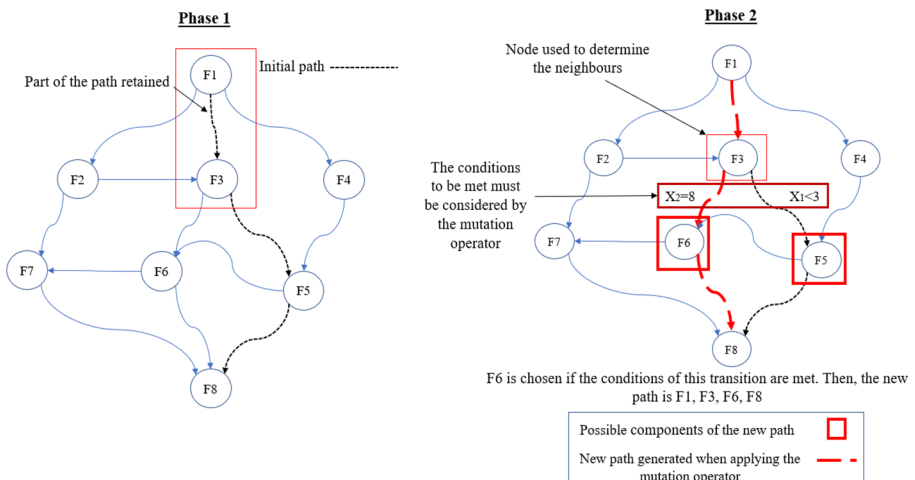


Fig. 6 Mutation operator

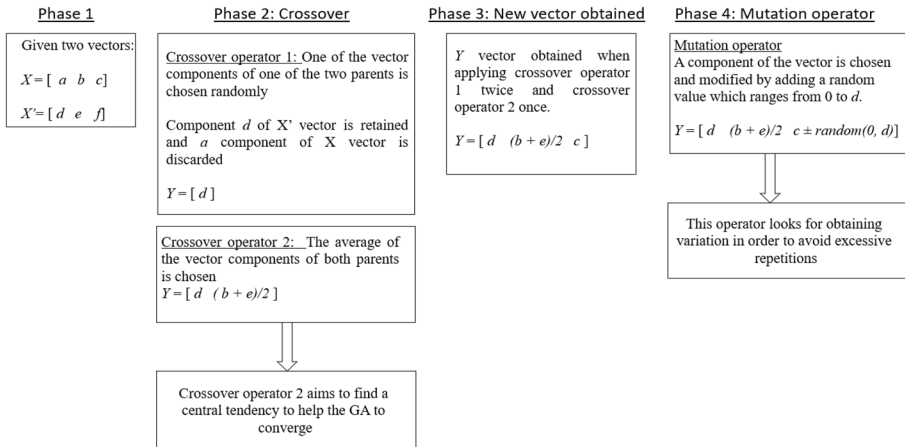


Fig. 7 Crossover and mutation operators for GA2

established. To do this, part of the path belonging to one of the parents from  $F_1$  to the common node is retained. The path belonging to the other parent from the common point to the final state ( $F_7$  in this example) is retained.

- **Mutation.** Once an optimal path is chosen, in some cases a mutation<sup>6</sup> is applied. As shown in Fig. 6, a path is a vector composed of functional states; it always contains  $F_1$  (initial state) and  $F_8$  (final state). Given a path, for example  $[F_1, F_3, F_5, F_8]$ , a randomly chosen part of the vector is retained:  $[F_1, F_3]$ . Then, taking  $F_3$ , the neighbours are established and one of them is randomly chosen if the values of the vector  $X$  allow this state to be reached (Figs. 2 and 3). Following this process, a new path is obtained. In some cases, there will be no solution. Therefore, the path will be discarded.

### 2.2.2 GA2: optimising the vector

GA2 optimises vectors in order to minimise the cost by using the cost values returned by GA1 (Fig. 3). When GA2 gets the cost and path for each vector, the vectors with the lowest cost are chosen. Then, the crossover and mutation operators are applied. As shown in Fig. 7, there are two crossover operators. Based on two parents (two vectors that contain the inputs of the SM under validation), crossover operator 1 randomly chooses the value of one of the parents and operator 2 calculates the average of both parents (Fig. 7, phase 1 and phase 2). When the new vector is obtained, in certain cases the mutation operator is applied. This operation consists of adding an aleatory value that ranges from 0 to  $d$  to a component of the final vector  $Y$ . This value  $d$  depends on the range of the inputs of the SM under validation.

<sup>6</sup> The number of vectors that will be mutated depends on the need. In this research, the authors have chosen 30% of the vectors to be mutated with good empirical results.

### 2.3 HIL simulations

Once the GAs are parametrized and a model is built as shown in Fig. 2, 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. 8) as the software variables have to reach the values specified in the test-case. Several ways can be used to set the conditions to pass from one state to another one. The first entails writing the equations directly in the code, which is limited to simple SMs as fairly complex and highly complex SMs involve many equations. The second option is to call the 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 dynamic-linked libraries (dlls) by following the steps described in the official Matlab® documentation. Figure 8 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 “Sects. 3 and 4”.

### 2.4 Network and software and hardware integration

This proposal validates the network and hardware and software integration by using the dlls as shown in Fig. 8. Once the software is coded, the software outputs must be equal or very close to the values provided by the Simulink® models despite the SM interactions. This point is checked by using the dlls that 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. 4 where the reader can find state  $F_{10}$  which assesses if the SM under validation has an impact on the network. 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 has also been considered in Fig. 4).

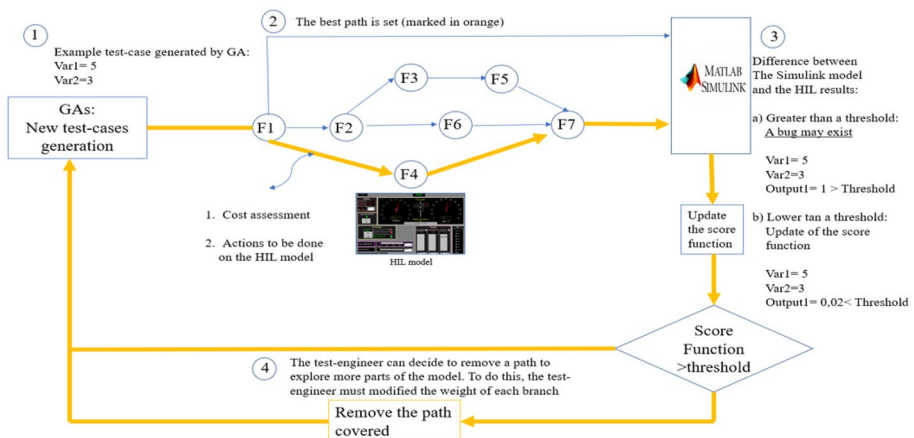


Fig. 8 HIL simulation process

## 2.5 Traditional techniques

The following techniques were used in this research.

### (a) Cause-effect technique

One of the most used techniques in the automotive sector is the black-box technique (Conrad et al., 2005). 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 behaviour. That is why this technique is also known as behavioural testing. When designing the test to be run, test engineers design test-cases and decide which means could be used according to their experience (Conrad et al., 2005; Garousi & Mäntylä, 2016; Kasoju et al., 2013). 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 behaviour. This behaviour is validated and assessed by using the outputs of the SM under validation. The means used to validate this behaviour are chosen by considering the test-engineers' experience when using this technique.

### (b) 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. Due 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 (Matelo® Software, 2018). 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 the variables used in a transition in such a way that it tries to cover all possible transitions and paths. Finally, each state can be a model, as is the case in this research, making the models extremely complex. Figure 9 sums up the aforementioned process. A test-case is generated and by using calls to Simulink® models, Matelo® determines which part of the model will be covered (Fig. 9 in orange). Many test-cases are generated to cover the whole model and to increase functional and code coverage.

## 2.6 Equipment

The following means were used in this research:

1. An engine ECU software and hardware designed by the company subjected to this research.
2. The HIL bench used to conduct this research belongs to the German manufacturer dSpace®, model dSpace® Simulator Full-size (dSpace Supplier, 2019a).
3. Regarding the HIL model which serves as the driver's interface, the ControlDesk®, version 5.1 from dSpace® manufacturer (dSpace Supplier, 2019b) was employed. By using this

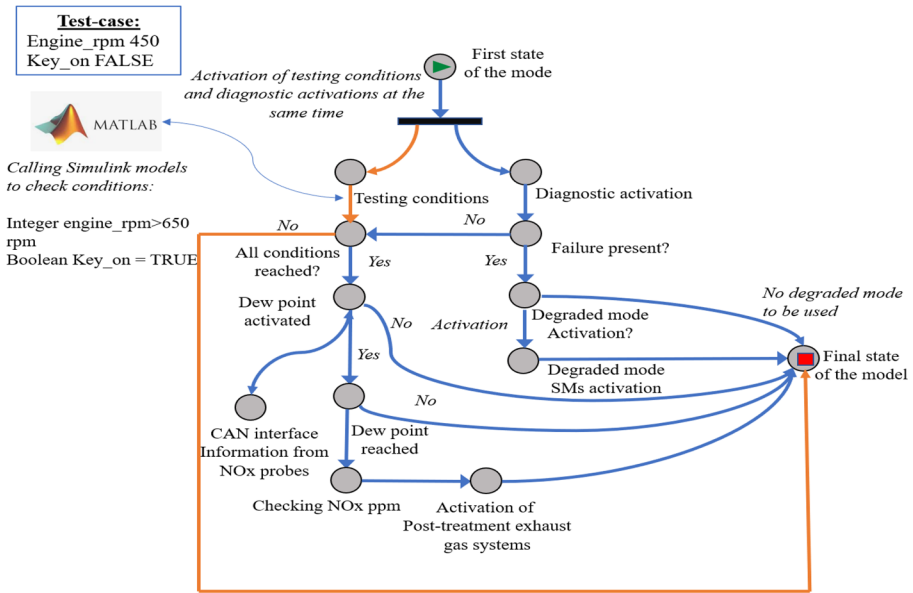


Fig. 9 Example of NO<sub>x</sub> activation model based on Matelo®

software, it is possible to carry out all necessary data exchange between the HIL bench and the engine ECU. All the data frames sent and received by the engine ECU through the vehicle networks can be modelled.

4. Throughout this research, measurements of different software variables stored in the engine ECU memory were made. Software that allows reading memory locations such as version 7.1.9 of INCA® was used (ETAS supplier, 2019). As a result, the necessary data acquisition could be made for each test-case run.
5. Python v. 3.7, Matlab® and Simulink® version 2015.
6. Matelo® software was used for the model-based testing technique.

Although ETAS® and dSpace® means were used, any manufacturer could have been chosen.

## 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 highly complex SMs (Table 3).

Table 4 shows the way of generating test-cases. All techniques used the software and system requirements traced in DOORs, feedback from other projects<sup>7</sup> and the Simulink®

<sup>7</sup> Feedback from other projects means bugs found in a project which could impact another project.

**Table 3** Types of SMs

Type of SM	Characteristics	Example of SMs	Validation requirements. Impact on validation time	Impact on means	Number of test-cases run	Reason to choose this number of test-cases
Simple	<ul style="list-style-type: none"> <li>oA reduced number of input and output variables present in the SM and a small number of calculations to be done. Furthermore, they are not complex</li> <li>oHigh accuracy needed for calculations in some cases</li> <li>oEasy to identify the main functional characteristics of the SM. They are also easy to test by using an HIL bench</li> </ul>	<p>Temperature estimators</p> <p>Brake pedal monitoring</p>	<p>Not time consuming. Some manipulations are required on the HIL model to make the engine ECU reach the desired operating point</p> <p>Example: When the fact that the accelerator pedal is blocked is detected, the engine ECU software must check a few parameters such as the break and accelerator pedal states and the vehicle speed</p>	<p>These types of SMs can be validated on HIL simulators as well as on prototype vehicles. However, some tests are better run in vehicles than through an HIL simulation</p>	250	<p>Considering it is easy to reach a specific functional state, almost all test-cases can be validated</p> <p>The number of use-cases is low due to SM complexity. Consequently, the number of test-cases chosen for this research is low compared to fairly complex SMs</p>
Fairly complex	<ul style="list-style-type: none"> <li>oHigh number of input and output variables present in the SM</li> <li>oModerate number of calculations to be performed and moderate accuracy needed for calculations</li> <li>oDifficult to identify the main functional characteristics of the SM</li> </ul>	<p>Torque engine limitation owing to the temperature of an engine component</p>	<p>They can be time-consuming depending on choosing the most adequate means to validate the SMs as well as the test-engineer's experience</p> <p>SMs require more manipulations to make the engine ECU reach the desired operating point</p> <p>Example: For instance, SMs related to the post-treatment of exhaust gases require that the NO<sub>x</sub> probes are heated before running whatever test-cases. This task can last between 3 and 7 min</p>	<p>Generally, these types of SMs can be validated on HIL simulators as well as on prototype vehicles. However, some specific use-cases must be validated by using vehicles</p>	1250	<p>Fairly complex SMs are widely used in the engine software. That is why a significant number of test-cases were chosen</p>

**Table 3** (continued)

Type of SM	Characteristics	Example of SMs	Validation requirements. Impact on validation time	Impact on means	Number of test-cases run	Reason to choose this number of test-cases
Highly complex	<p>oHigh number of input and output variables and number of calculations</p> <p>oCalculation not necessarily complex</p> <p>oHigh number of functional calculations and moderate/low calculation accuracy</p>	<p>The SM in charge of controlling the oil rate diluted into diesel</p>	<p>Time consuming. Running a test-case involves a lot of manipulations of the HIL simulator as well as time to reach the initial conditions</p> <p>Example: The SM in charge of assessing the amount of diesel diluted in the oil due to diesel particulate filter requires that the driver covers between 10,000 and 20,000 km for each functional state</p>	<p>To reduce the time needed to validate the SM, the test-cases are automated by using Python scripts through HIL simulations</p>	100	<p>As reaching the final conditions is time consuming, a lower number of test-cases was chosen for these SMs to meet the planning constraints</p>



**Table 4** 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	<ol style="list-style-type: none"> <li>1. Feedback from other projects</li> <li>2. Software requirements</li> <li>3. System requirements</li> <li>4. Simulink® specifications</li> </ol>	<p>DOORS Corporate database to trace bugs Excel® file which contains all information needed (initial conditions, actions to be done, etc.)</p>	<p>Manually, by interpreting: a)the software and system requirements b)the information of bugs traced in the corporate database</p>	None
Model-based testing	<ol style="list-style-type: none"> <li>1. Feedback from other projects</li> <li>2. Software requirements</li> <li>3. System requirements</li> <li>4. Simulink® specifications</li> </ol>	<p>Matelo® DOORS Corporate database to trace bugs</p>	<p>Automatically done by Matelo® by covering the model built by the test-engineer</p>	Functional model
Genetic algorithms	<ol style="list-style-type: none"> <li>1. Feedback from other projects</li> <li>2. Software requirements</li> <li>3. System requirements</li> <li>4. Simulink® specifications</li> </ol>	<p>Pseudorandom values generated by Python when coding GAs</p>	<p>Automatically done by genetic algorithms</p>	Low level model

specifications as inputs. By analysing all these input data, the test-engineers build models when using GAs and 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.

### 3 Results

This section compares the performance among GAs and traditional techniques by using the KPIs indicated in Table 5.

#### 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).

$$\sum_{j=1}^{j=m} |HIL_j - Simulink_j| \approx 0 \quad (2)$$

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  $Simulink_j$  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. (3) which relates to the number of Simulink® blocks tested versus the total number of blocks presented in the specifications of the SM under validation.

$$Code\ coverage = \frac{\text{number of Simulink® blocks tested}}{\text{number of Simulink® blocks present in the SM under validation}} \times 100 \quad (3)$$

Table 6 shows the number of blocks present in the SMs validated in this research, which is used to assess the code coverage (Table 7).

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

**Table 5** KPIs 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

**Table 6** Number of total Simulink® blocks

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

different branches of the Simulink® model (Fig. 10). Contrary to GAs, model-based testing does not allow tuning the inputs of the SM with the aim of choosing the best means (i.e. an HIL simulation or vehicles) to validate an SM. 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 as described in “Sect. 3.3”. 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 prototype vehicles. GAs perform better as they can cover more Simulink® blocks provided that the right means are used. This topic is discussed in depth in “Sect. 4”.

Code coverage should at least be 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.

### 3.2 Functional coverage

Table 8 shows the functional states linked to the Simulink® blocks present in the chosen SM. 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. 10).

Table 9 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 because of tuning, validation and test-driving activities.

### 3.3 Automation

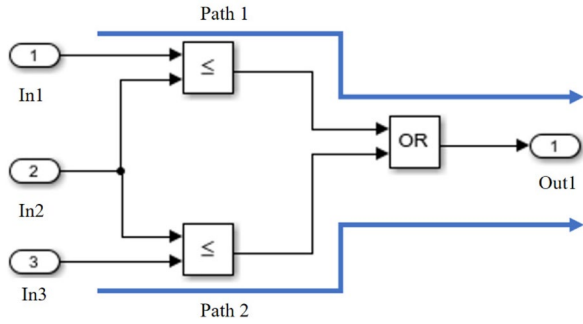
For several reasons, the automation process is difficult to be performed when it comes to engine ECU software due to SM interactions. Firstly, reaching the values for inputs of the SM under validation is difficult as the complexity of the SM increases. Secondly, if inputs do not reach the expected values, the values of the outputs of the SMs under validation will be no longer valid (Fig. 11).

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. 12. The importance of choosing the best technique to automate the process is discussed in “Sect. 4”.

**Table 7** Code coverage obtained when validating the 15 SMs

Technique	Simple SM		Fairly complex SM		Highly 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	

**Fig. 10** Example of different ways of activating an output



### 3.4 Time for implementation

Table 10 shows the time measured in this study with the aim of assessing the productivity gain of each technique. All data shown in this table were conducted empirically by using the test-engineer sample indicated in “Sect. 5”.

(a) GAs

GAs coded in this study use low state models which require time to be implemented (time for coding/updating, design and validating models). This time included: building the model and coding necessary conditions by going from one state to another. Then, the automation code to control the HIL model is created (Python script for the automation process per SM). Finally, the test-cases are run. Table 11 shows the results obtained for GAs.

(b) Cause-effect technique

In this case, test-engineers must check and understand the specifications in depth as they have to transform the Simulink® models into a test-case (*Time for designing test-cases*). Afterwards, the Python scripts are coded to automate the process. Before the approval of the validation procedure for a specific SM, the test-engineer who wrote the test-cases has to check whether initial conditions, expected results and the calibration used are coherent. The validation procedure of SMs cannot be delivered to the testing team without this verification. Table 12 shows the results obtained for the cause-effect technique.

(c) Model-based testing

In this case, test-engineers must create a functional model by using the Simulink® specifications (*Time for coding, design and validate models*). Afterwards, the python scripts are

**Table 8** Number of total functional requirements

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

**Table 9** Functional coverage obtained for each research

Technique	Simple SM		Fairly complex SM		Highly 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

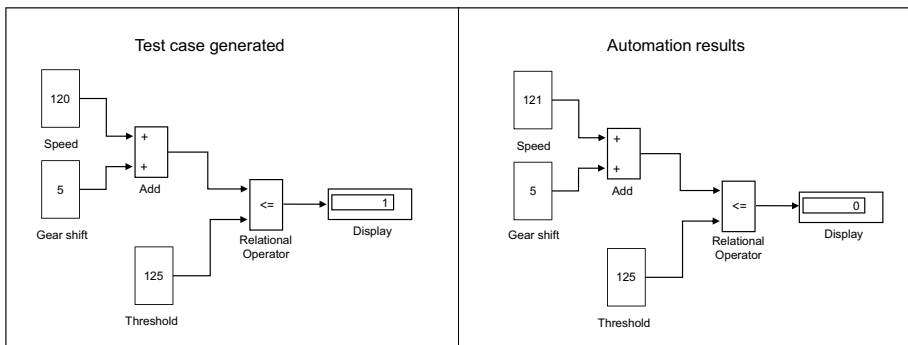
coded to automate the process. Table 13 shows the results obtained for the model-based testing. The time needed to code, design and validate models is lower in this case than with GAs. The main explanation is that low level models are used in GAs whereas in the model-based testing technique, functional models are implemented. As a result, low level models require more states and more time to be implemented.

Table 14 shows the total time needed for validating 15 SMs when using HIL simulations and vehicles. GAs performed better than other techniques since GAs determine what the best means to be used for each SM are. The result is that the average time for running test-cases is lower. Table 15 depicts the gain of GAs over other techniques. A comparison among all these techniques is drawn in “Sect. 4”.

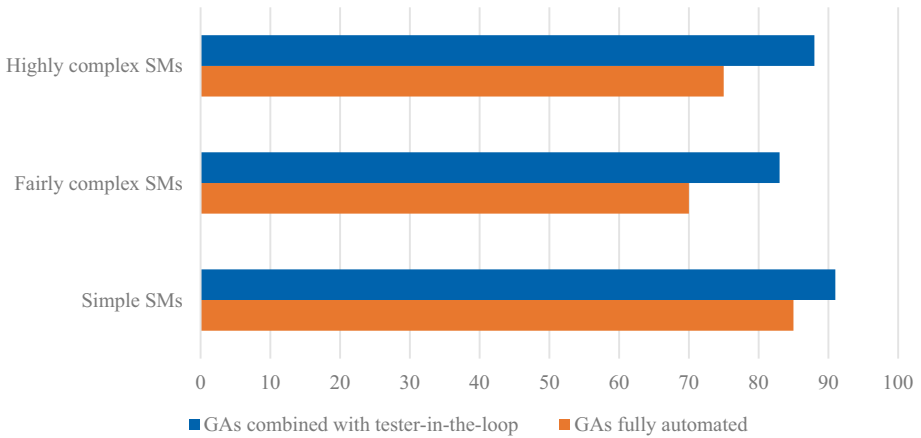
### 3.5 Bugs

#### 3.5.1 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 16 and Fig. 13. 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 14



**Fig. 11** Potential error when a test-case is automated



**Fig. 12** Success rate when automating the HIL simulation

depicts the obtained result for a software variable output of an 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<sub>x</sub>. 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. Only GAs combined with Simulink® model can detect it.

### 3.5.2 Number of bugs

The results are shown in Fig. 15. GAs overperform the rest of the techniques used in this paper because Simulink® blocks are used. 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 model is used. The result is that it is extremely difficult to establish both the code and functional coverage.

**Table 10** Factors considered to assess the productivity gain

Factors	Description
Time needed to design test-cases	Simulink® specifications and the process of how SMs operate must be understood to implement test-cases
Time necessary to implement the scripts and functional models	The automation process needs the implementation of Python scripts. In addition, some techniques require such models as GAs and the model-based testing
Time elapsed to run the test-cases	The time needed to run test-cases is different depending on the technique used as well as the type of SM under validation
Time needed to carry out the validation process	Time needed to check whether the software runs as expected after having conducted the test-case

**Table 11** Time needed to use GAs for validating SMs

	Simple SMs	Fairly complex SMs	Highly complex SMs
Time for designing and coding/ updating (h)	13	61	93
Time for coding, design and validating models per SM (h)	4	10	15
Python script for the automation process per SM	17	71	108
Total time for designing and coding (h) per SM	85	355	540
Total time for designing and coding (h) for 5 SMs	3.17	512.17	141
Time for executing an automated test-case (h)	0.0008	0.0267	0.08
Time for validating	88.17	867.2	681.08
Test-case execution (h)			
Validation time (h)			
Total time (h)			





**Table 13** Time needed to use the model-based testing

	Simple SMs	Fairly complex SMs	Highly complex SMs
Time for designing and coding (h)	12	55	75
Time for coding, designing and validating models per SM (h)	6	16	35
Python script for the automation process (h) per SM	18	71	110
Total time for designing and coding (h) for 5 SMs	90	355	550
Total time for designing and coding (h) for 5 SMs	3.5	641.08	147
Time for executing an automated test-case (h)	0.00028	0.00347	0.00044
Time for validating (h)	93.5	996.09	697
Total time when using the model-based testing			
Test-case execution (h)			
Validation time (h)			
Total time (h)			

**Table 14** Total time needed to validate SMs

		Number of test-cases		Time (h)		Total time (h)
		HIL	Vehicle	HIL	Vehicle	
GAs	Simple SMs	38	10	88.2	0.50	88.7
	Fairly complex SMs	878	350	867.2	87.50	954.7
	Highly complex SMs	94	3	681.1	3	684.1
Model-based testing	Simple SMs	42	8	93.5	0.67	94.2
	Fairly complex SMs	1099	151	996.1	88.08	1084.2
	Highly complex SMs	98	2	697.0	3	700.0
Cause-effect technique	Simple SMs	38	12	93.2	0.5	93.7
	Fairly complex SMs	1095	155	958.8	90.42	1049.2
	Highly complex SMs	99	1	698.5	2	700.5

### 3.5.3 Bugs found by other clients

Generally, bugs that not found are linked to fairly complex SMs. As detailed in the next section, it is difficult to test all possible combinations of variables considering all the possible values they can reach (see “Sect. 4.1”). All techniques offer good performance when validating highly complex and simple SMs (Fig. 16). The only occasion when a bug was not found while validating simple SMs was linked to a use-case that was not tested as the cause-effect technique does not employ models.

## 4 Discussion

### 4.1 Test-case formulation

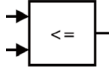

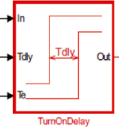

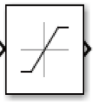


Several challenges must be considered when designing test-cases.

- (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 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.

**Table 15** Productivity gain of GAs vs other techniques

	Simple SMs	Fairly complex SMs	Highly complex SMs
GAs vs the model-based testing	5.8%	11.9%	2.3%
GAs vs the cause-effect technique	5.3%	5.4%	2.3%

**Table 16** Types of bugs found

	<p>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.</p>
	<p>This block allows choosing between two possible paths depending on B value (which is a Boolean variable). When B is TRUE, the output of this block provides A value. Otherwise, the value of C is provided. In many cases, A and C are also Boolean variables and they allow running different strategies to control a function or subfunction of the engine ECU software. During this research, many activations of the wrong strategy were detected as the value of B was not assessed properly due to calculation or coding errors in previous Simulink® blocks.</p>
	<p>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 fairly and highly complex SMs, it is more difficult to succeed by making the input In remain stable, than in simple SMs</p>
	<p>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. In this case, the bugs found were mainly linked to software performance<sup>8</sup> or coding errors as the interpolation was not coded or tuned properly. Consequently, the output provided by this block was wrong.</p>
	<p>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 is vital for limiting the value of an output variable. When running test-cases, the test engineer verifies whether the limits have been set properly or not. Due to this bug the consequences are diverse: calculation errors, activation of the wrong strategy to control a function of the engine ECU software, etc. The main root cause is a calibration issue that can be fixed quickly without releasing new software.</p>
	<p>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 to reach the conditions when the S-input could be activated in certain cases (for example, when validating exhaust gas treatment systems or oil adaptive maintenance functions).</p>
	<p>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 to reach the conditions to generate a falling edge in certain cases (for example, when validating exhaust gas treatment systems).</p>

- (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, a test can fail when a variable takes a value close to its upper limit, but if values are not close to this value, the test-case can provide the expected results. Therefore, the functional model must be covered with different test-cases which take different combinations, at least during the validation process.
- (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 the 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 may no longer be valid. Therefore, the traditional formulation of test-cases based on input and expected output values cannot be used in simulations. DLLs allow solving this technical issue as depicted in Fig. 8. As a result, it is always possible to assess Eq. (2) as DLLs can provide the output values for the input ones reached during the HIL simulation. Therefore, GAs can check if the software runs as expected by comparing the HIL results and the Simulink® models results.

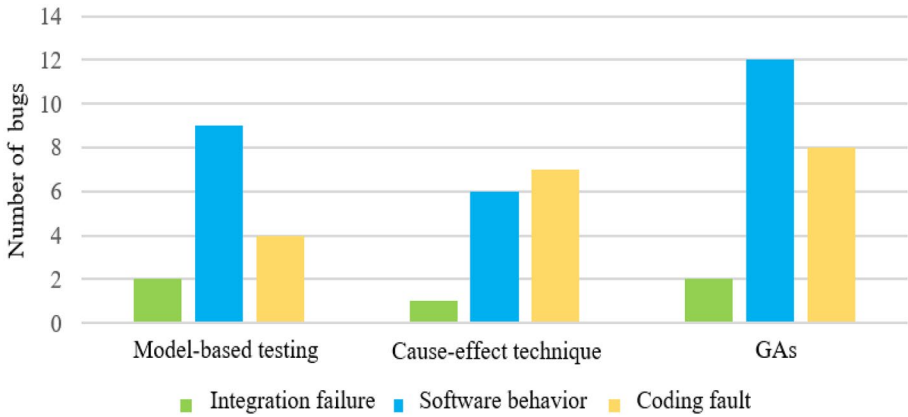


Fig. 13 Types of bugs found

### 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. 11). Regarding fairly complex SMs, as the number of variables present in SMs is high, it is recommended to use the tester-in-the-loop. Highly complex SMs have many functional states linked to the number of km covered (for 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, inject-

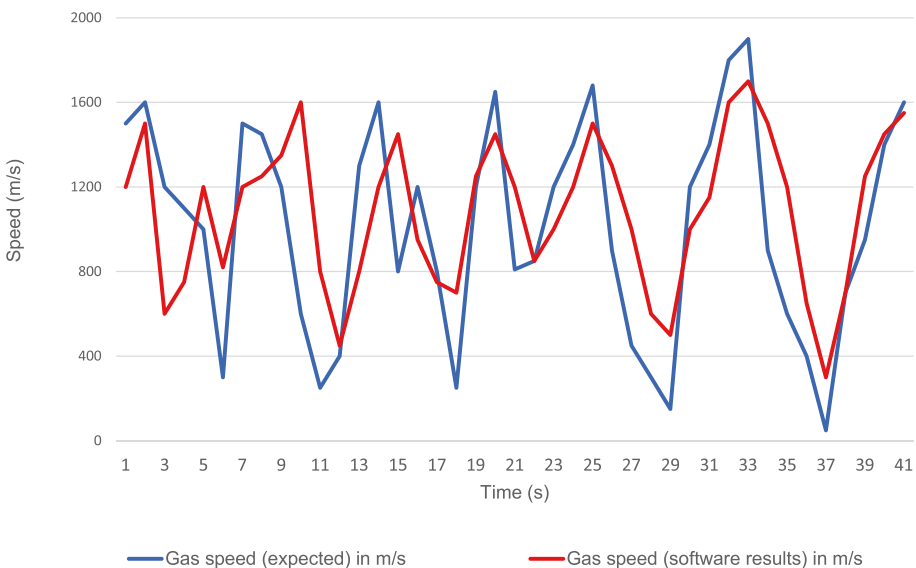
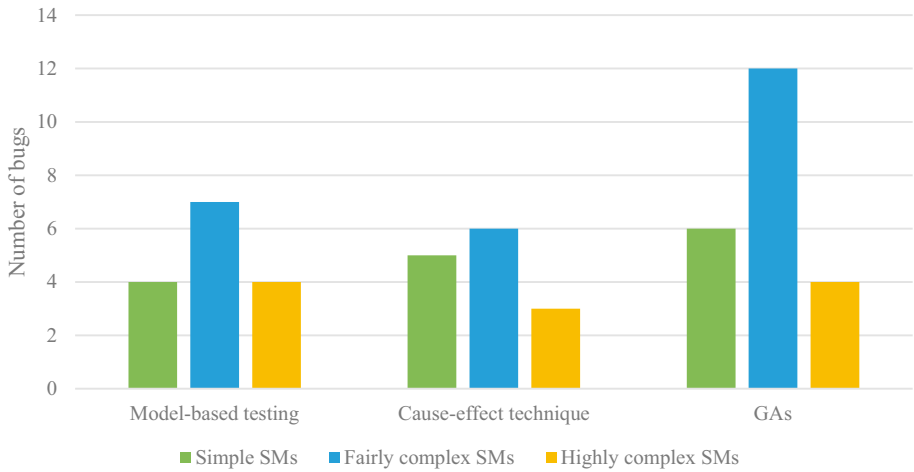


Fig. 14 Bug not detected unless GAs are used

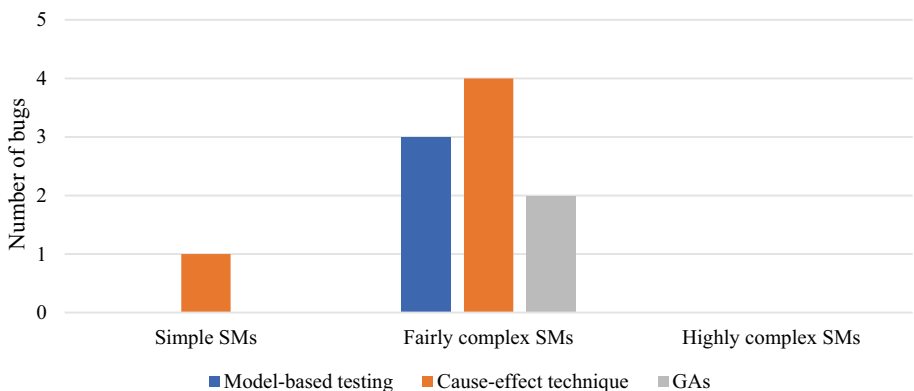


**Fig. 15** Number of bugs found by using each technique

ing the obtained data file 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 Controller Area Network (CAN). 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 modelled by using Simulink®. DLLs must be used if GAs are applied. Not all SMs of the engine ECU software have a specification based on the Simulink® models. Consequently, GAs cannot be applied to any SM. However, only 7% of the SMs did not have Simulink® models.

Certain highly complex SMs need to cover many kilometres 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



**Fig. 16** Bugs not detected

cases, the cause-effect technique is recommended to reduce the validation time. However, 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.

### 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 a prototype vehicle. If the wrong means is chosen, many attempts will be 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 due to GAs has an important impact on software quality. As shown in Fig. 17, if software version A is validated with some delay (weeks 17 and 18), after the specifications for software B are sent to the 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.

### 4.4 Means optimisation

Choosing the adequate means to validate the software and the increase in productivity contribute to a better establishment of the number of vehicles and HIL simulators needed. Consequently, the cost of the projects can be reduced.

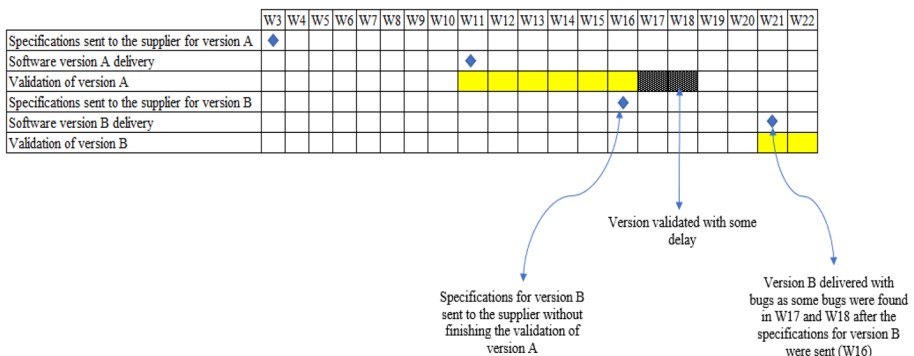


Fig. 17 Delays in software validation and impacts on software quality

## 4.5 Limitations of this proposal

Even if this research was focused on the engine ECU, this proposal can be applied to any ECU whose specifications are based on Simulink® models. When using these models, some parts of them should be validated by using HIL simulations while others by using prototype vehicles as it is in the case of the engine ECU. Therefore, GAs can be used to generate test-cases (the values for the inputs of the SM, the expected values and dlls) and to choose the most adequate means.

## 4.6 Future research

Future research should be focused on detecting faults in the performance of software. In some occasions, the software is coded properly but the software may not run as expected from a functional point of view, as the design team has not considered this use-case.

# 5 Validity of this research

## 5.1 Threats to validity

Several internal and external threats have been considered to conduct the threats to validity analysis. Table 17 shows the predictors (variables to be controlled) and their influence on the response variables (productivity gain, documentation quality, test-cases quality and bugs).<sup>8</sup>

Considering that one of the most important factors to be analysed is the number of bugs found when using GAs, it is vital to verify how these variables impact that. The authors have considered the staff's experience in the engine ECU as a key factor impacting quality. Consequently, 2 SMs of each type were validated by using test-procedures done by staff with 5 years of experience in the service and with less than 2 years. The results (Table 18) demonstrate that the choice of the best use-cases to test the software requirements is linked to the staff's experience. In addition, testability also includes the means used to run a test-case and that is where this research is focused on. Sometimes a bug is not found when using HIL simulation; however, it is when using vehicles. The consequence of having less experience is the wrong assessment of factors shown in Table 1.

Regarding external threats, it must be checked if the results can be generalised and applied to a larger group. To assure this, a time implementation validity study was conducted to validate the time obtained in "Sect. 3.4" ("Sect. 5.2").

## 5.2 Sensitivity analysis

The time needed for designing and coding test-cases for each technique, considering the staff's Python skills, is essential when analysing the results obtained. As shown in Table 19, the staff was divided into three groups according to their Python skills. It is vital to analyse the impact of knowledge of the engine ECU and Python skills on productivity. The staff

---

<sup>8</sup> Considering the complexity of this case-study, the number of variables used as predictors must be limited. Otherwise, it would be extremely complex to draw conclusions.



**Table 17** Factors considered

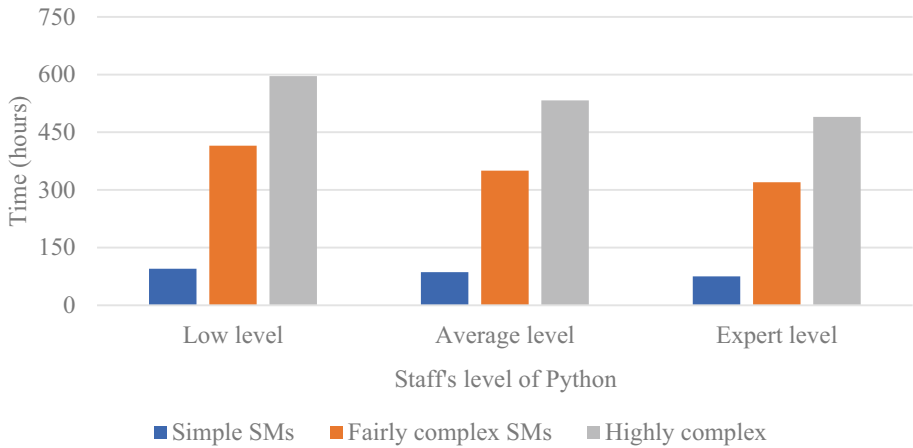
Id	Factor	Description
1	Sample used	This study was conducted in the software validation service of one of the most important manufacturers in Europe. The staff used in this research is composed of 40 people: 19 engineers and 21 technicians. Each person may have different skills, but this fact was considered in the sensitivity analysis in “Sect. 5.2”
2	Python skills	The more the validation department masters Python, the higher the productivity gain is. The more extensive knowledge of an engine operation the staff can acquire, the less time they require to write the tests. The influence of all these aspects was analysed in the sensitivity analysis (“Sect. 5.2”)
3	SM used	The SMs present in the engine ECU software have a different level of complexity. The conclusions differ depending on the SM under validation. The authors have divided the SMs into three groups to assess the productivity gain properly
4	Unreliability of measures	All measures were taken in the same conditions by using a procedure. In addition, all functional models used have been validated before using them. Otherwise, the conclusions could be wrong
5	Staff’s experience in the engine ECU software	The members of the staff of a validation service may change their positions in the company. As a result, the department may have more specialised people at a specific moment and vice versa on other occasions. This research was performed considering different scenarios depending on the staff’s experience (“Sect. 5.2”)

**Table 18** Bugs found depending on the staff's experience

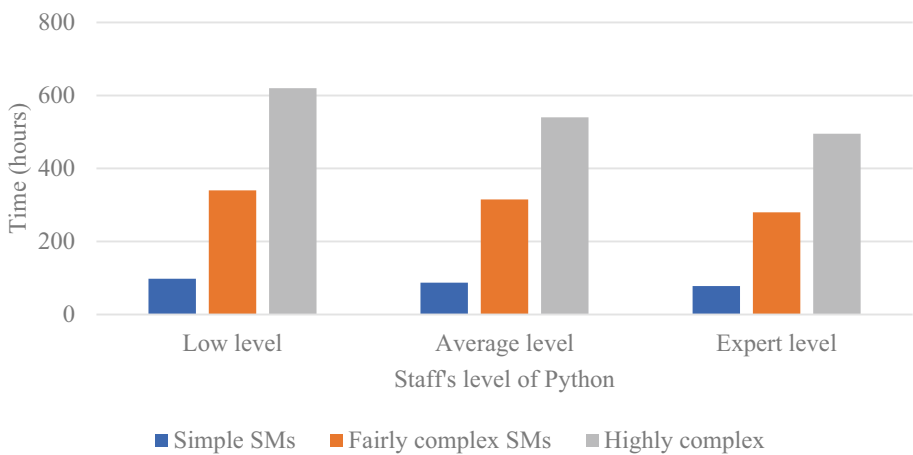
SM\technique	Model-based testing		Cause-effect		GAs	
	Experience > 5 years	Experience < 2 years	Experience > 5 years	Experience < 2 years	Experience > 5 years	Experience < 2 years
	Pedal accelerator blocked	1	1	1	1	1
Cruise control	2	1	2	1	5	2
Safety (torque control)	2	0	1	0	2	0

**Table 19** Staff’s training in Python

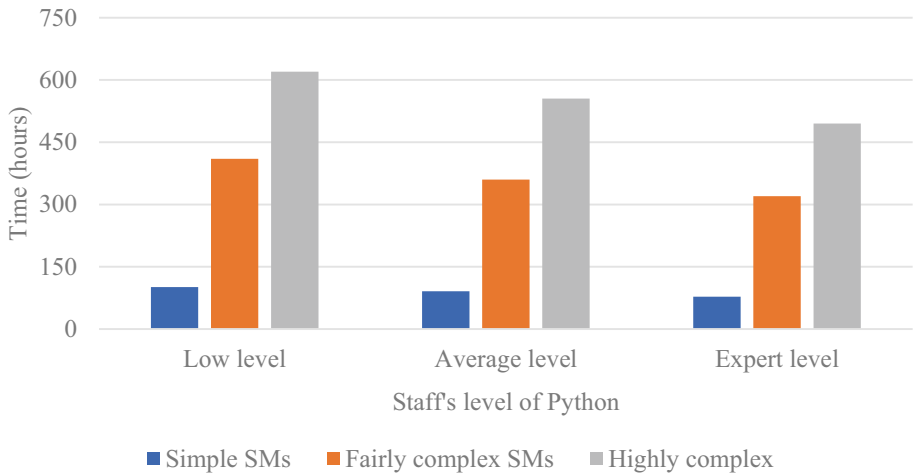
Group	Experience in coding Python scripts	Experience in engine ECU	Number of members
Expert level	More than 2 years	5 years	5
Average level	Between 1 and 2 years	2 years	7
Low level	Less than 1 year	Less than 1 year	3



**Fig. 18** Time for designing and coding vs staff’s Python skills for GAs



**Fig. 19** Time for designing and coding vs staff’s Python skills for the cause-effect technique



**Fig. 20** Time for designing and coding vs staff's Python skills for the model-based testing

of the validation service of the company subjected to this research has been classified as expert, average and low level, based on their experience in Python and in the engine ECU.

Figures 18, 19 and 20 show the time needed for designing test-cases and for coding Python scripts. In this research, all activities described in “Sect. 4” were performed by each group (expert, average and low level) to assess the performance difference between these groups.

## 6 Conclusions

Engine ECU software is one of the most complex software systems 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 (hardware-in-the-loop 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 due to the usage of Simulink® models 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 obtained 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 highly complex ones. However, when it comes to fairly complex ones, i.e. 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. The GAs performed better in terms of functional and code coverage. With respect to functional coverage, GAs improved up to 11% in fairly complex SMs and 8.4% in highly complex SMs, when using the cause-effect technique. The GAs improved up to 4% in fairly complex SMs and 3% in highly complex SMs in the model-based testing technique. In terms of code coverage, GAs improved up to 12.8% and 7% for fairly complex and highly complex SMs, respectively, under the

cause-effect technique; and up to 7.1% and 1.4% for fairly complex and highly complex SMs, respectively, with model-based testing.

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

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

## Appendix

For confidentiality reasons, only a beta version can be provided. It can be downloaded in the following link:

<https://github.com/pedroai1980/ga.git>

The version provided by the authors tries to solve the problem shown in Fig. 21. The reader could reuse it with some changes such as adding automation scripts to each

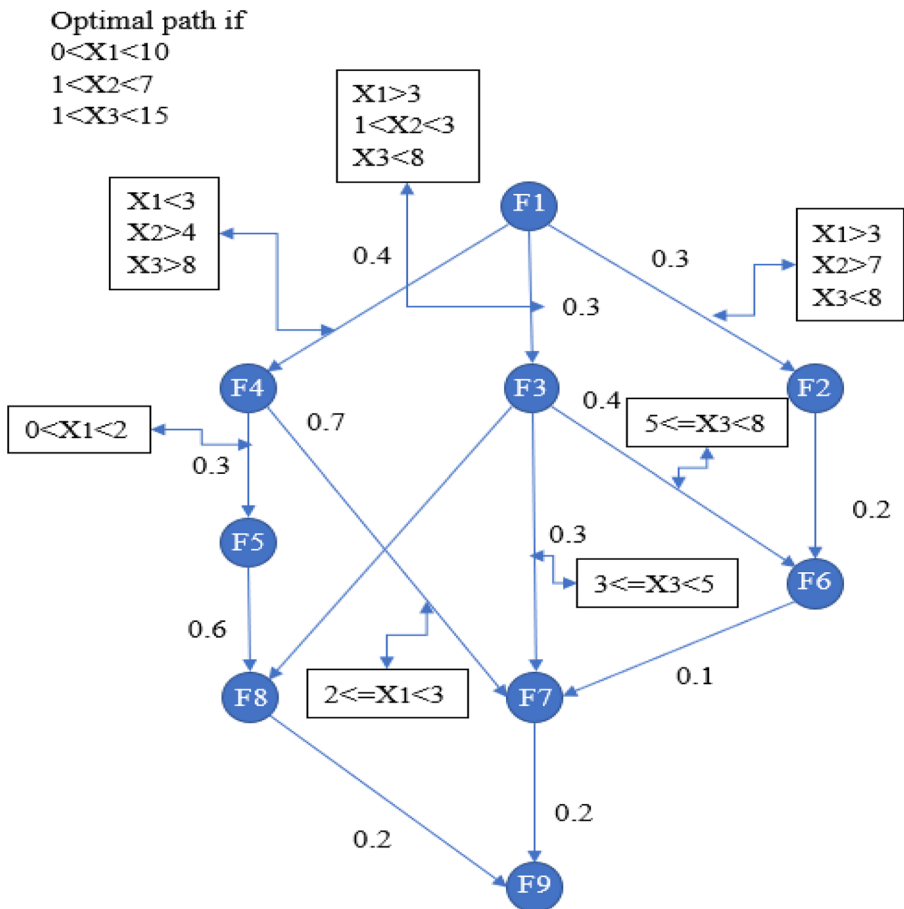


Fig. 21 Problem to be solved by using the code provided by the authors

transition between states. Adding calls to Simulink® models to assess conditions to go from one state to another one.

The code is composed of the following files:

- i. `utils.py`. This file defines the functions necessary to assess conditions for going from one state to another one. The reader can replace and add the functions they want or they can even add calls to Simulink® models.
- ii. `main_v2.py`. This file runs the code to solve the problem.
- iii. `g2_func.py` and `genetic_funcs.py` contain the code of the two genetic algorithms necessary to solve the problem.

## References

- Abadeh, M. N. (2020). Performance-driven software development: An incremental refinement approach for high-quality requirement engineering. *Requirements Engineering*, 25, 95–113.
- Ågren, S. M., Knauss, E., Heldal, R., Pelliccione, P., Malmqvist, G., & Bodén, J. (2019). The impact of requirements on systems development speed: A multiple-case study in automotive. *Requirements Engineering*, 24, 315–340.
- ASPIEC. (2020) ISO - ISO/IEC 33001:2015 - Information technology — Process assessment — Concepts and terminology. Accessed 30 January 2020.
- Banish, G. (2007). Engine management: Advanced tuning. Minnesota: Cartech.
- Chunduri, A. (2016). <http://www.diva-portal.org/smash/get/diva2:945731/FULLTEXT02>. Accessed 3 February 2020.
- Conrad, M., Fey, I., & Sadeghipour, S. (2005). systematic model-based testing of embedded automotive software. *Electronic Notes in Theoretical Computer Science*, 1111, 13–26.
- Delius, G. W. (2004). *Orthogonal Arrays (Taguchi Designs)*. University of York. <https://www.york.ac.uk/depts/maths/tables/orthogonal.htm>. Accessed 29 December 2021.
- Dos Santos, J., Martins, L. E. G., de Santiago Junior, V. A., Povoia, L. V., & dos Santos, L. B. R. (2019). Software requirements testing approaches: A systematic literature review. *Requirements Engineering*. <https://doi.org/10.1007/s00766-019-00325-w>
- dSpace. (2018). <https://www.dspace.com/en/inc/home.cfm>. Accessed 29 December 2021.
- dSpace Supplier. (2019a). [https://www.dSpace.com/en/inc/home/products/hw/simulator\\_hardware/dSpace\\_simulator\\_full\\_size.cfm](https://www.dSpace.com/en/inc/home/products/hw/simulator_hardware/dSpace_simulator_full_size.cfm). Accessed 10 December 2019.
- dSpace Supplier. (2019b). <https://www.dSpace.com/en/inc/home/products/sw/experimentandvisualization/controldesk.cfm>. Accessed 10 December 2019.
- El-Rewini, Z., Sadatsharan, K., Flor, D., Siby, S., Plathottam, J., & Ranganathana, P. (2019) Cybersecurity challenges in vehicular communications. *Vehicular Communications*, 23, 100214
- Esfandyari, S., & Rafe, V. (2018). A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy. *Information and Software Technology*, 94, 165–185.
- ETAS supplier. (2019). [https://www.etas.com/en/products/inca\\_software\\_products.php](https://www.etas.com/en/products/inca_software_products.php). Accessed 9 March 2020.
- Feldhütter, A., Segler, C., & Bengler, K. (2018). 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*, 597, 730–741.
- Gajjar, M. J. (2017). *Mobile sensors and context-aware computing*. Morgan Kaufmann Publishers.
- Garousi, V., Felderer, M., & Kilicaslan, F. N. (2018). A survey on software testability. Cornell University. <https://arxiv.org/abs/1801.02201>. Accessed 17 January 2020.
- Garousi, V., & Mäntylä, M. V. (2016). A systematic literature review of literature reviews in software testing. *Information and Software Technology*, 80, 195–216.
- Haghighatkah, A., Banijamali, A., Pekka Pakanen, O., Oivo, M., & Kuvaja, P. (2017). Automotive software engineering: A systematic mapping study. *Journal of Systems and Software*, 128, 25–55.
- 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*, 3, 36–51.

- Huang, W.L., Wang, K. Ly, Y., & Zhu, F. (2016). Autonomous vehicles testing methods review. In *IEEE 19th international conference on intelligent transportation systems (ITSC)* (pp. 163–168).
- ISO. (2019). Cybersecurity standard. <https://www.iso.org/standard/70939.html>. Accessed 20 September 2020.
- ISO. (2020). Autonomous driving safety standard. <https://www.iso.org/standard/70918.html>. Accessed 20 September 2020.
- Kasoju, A., Petersen, K., & Mäntylä, M. V. (2013). Analyzing an automotive testing process with evidence-based software engineering. *Information and Software Technology*, 55(7), 1237–1259.
- Kim, Y., Lee, D., Baek, J., & Kim, M. (2020). MAESTRO: Automated test generation framework for high test coverage and reduced human effort in automotive industry. *Information and Software Technology*, 123, 106221.
- Koegel, M., & Wolf, M. (2018). *Auto update – Safe and secure over-the-air (SOTA) software update for advanced driving assistance systems*. Springer.
- Köhl, S., Lemp, D., & Plöger, M. (2003). ECU network testing by hardware-in-the-loop simulation. *ATZ Worldwide*, 105(10), 10–12.
- Krügner, M., Straube, S., Middendorf, A., Hahn, D., Dobs, T., & Lang, K. D. (2016). Requirements for the application of ECUs in e-mobility originally qualified for gasoline cars. *Microelectronics Reliability*, 64, 140–144.
- Linderman, U., Maurer, M., & Braun, T. (2009). *Structural complexity management*. Springer.
- Lockledge, J. C., & Salustri, F. A. (2010). Defining the engine design process. *Journal of Engineering Design*, 10, 109–124. <https://doi.org/10.1080/095448299261344>
- Martin, H., Ma, Z., Schmittner, C., Winkler, B., & Kreiner, C. (2020). Combined automotive safety and security pattern engineering approach. *Reliability Engineering & System Safety*, 198, Article 106773.
- Matelo® Software. (2018). <https://www.all4tec.com/>. Accessed 7 February 2020.
- McAfee. (2016). <https://www.mcafee.com/enterprise/en-us/assets/white-papers/wp-automotive-security.pdf>. Accessed 7 September 2020.
- Melo, S. M., Carver, J. C., Souza, P. S. L., & Souza, S. R. S. (2019). Empirical research on concurrent software testing: A systematic mapping study. *Information and Software Technology*, 105, 226–251.
- Möller, D., & Haas, R. (2019). Guide to automotive connectivity and cybersecurity. Wiesbaden: Springer
- Morris, D., Madzudzo, G., & Garcia-Pereza, A. (2020). Cybersecurity threats in the auto industry: Tensions in the knowledge environment. *Technical Forecasting and Social Change*, 157, 120102.
- National Instrument. (2019). <https://www.ni.com/fr-fr/innovations/white-papers/17/what-is-hardware-in-the-loop-.html>. Accessed 3 March 2020.
- Ortega-Cabezas, P. M., Colmenar-Santos, A., Borge-Diez, D., & Blanes-Peiró, J. J. (2019a). Application of rule-based expert systems and dynamic-link libraries to enhance hardware-in-the-loop simulation results. *The Journal of Software*, 14(6), 265–292.
- Ortega-Cabezas, P. M., Colmenar-Santos, A., Borge-Diez, D., & Blanes-Peiró, J. J. (2019b). Application of rule-based expert systems in hardware-in-the-loop simulation case study: Software and performance validation of an engine electronic control unit. *Journal of Software: Evolution and Process*. <https://doi.org/10.1002/smr.2223>
- Petrenko, A., Nguena-Timo, T., & Ramesh, S. (2015). Model-based testing of automotive software: Some challenges and solutions. *52nd Congress ACM/IEEE Design Automation Conference*.
- Placho, T., Schmittner, C., Bonitz, A., & Wana, O. (2020). Management of automotive software updates. *Microprocessors and Microsystems*, 78, 103257.
- Plummer, A. R. (2006). Model-in-the-loop testing, proceedings of the institution of mechanical engineers part I. *Journal of Systems and Control Engineering*, 220(3), 183–199.
- Raffaëlli, L., Vallée, F., Fayolle, G., Armines, A., de Souza, P., Rouah, X., Pfeiffer, M., Géronimi, S., Pérot, F., & Ahia, S. (2016). *Embedded Real Time Software and Systems Conference*.
- Raikwar, S., Jiyabhau, L. W., Arun Kumar, S., & Sreenivasulu Rao, M. (2019). Hardware-in-the-loop test automation of embedded systems for agricultural tractors. *Measurement*, 133, 271–280.
- Rajan, A., & Wahl, T. (2013). *CESAR - Cost-efficient methods and processes for safety-relevant embedded systems*. Springer.
- Riedmaier, S., Ponn, T., Ludwig, B., Shick, F., & Diermeyer, F. (2020). Survey on scenario-based safety assessment of automated vehicles. *IEEE Access*, 8, 87456–87477.
- Roychoudhury, A. (2009). *Embedded systems and software validation*. Morgan Kaufmann Publishers.
- Sharma, C., Sabharwal, S., & Sibal, R. (2013). A survey on software testing techniques using genetic algorithm. *IJCSI International Journal of Computer Science Issues*, 20(1), 381–387.
- Sharma, A., Patani, R., & Aggarwal, A. (2016). Software testing using genetic algorithms. *International Journal of Computer Science & Engineering Survey (IJCSSES)*, 7(2), 21–33.
- Sopan-Barhate, S. (2015). Effective test strategy for testing automotive software. *International Congress of Electronic Instrumentation and Control*.

- Sun, W., Cai, X., & Meng, Q. (2016). Testing flight software on the ground: Introducing the hardware-in-the-loop simulation method to the alpha magnetic spectrometer on the International Space Station. *Nuclear Instruments and Methods in Physics Research Section a: Accelerators, Spectrometers, Detectors and Associated Equipment*, 815, 83–90.
- Tatar, M., & Mauss, J. (2014). Systematic test and validation of complex embedded systems. *Embedded Real Time Software and Systems*.
- Utesch, F., Brandies, A., Pekezou, P., Schiessl, F., & Schiessl, F. (2020). Towards behaviour based testing to understand the black box of autonomous cars. *European Transport Research Review*, 12, 48.
- Vandi, G., Nicolò, C., Corti, E., Mancini, G., Moro, D., Ponti, F., & Ravaglioli, V. (2014). Development of a software in the loop environment for automotive powertrain system. *Energy Procedia*, 45, 789–798.
- Vector. (2019). <https://www.vector.com/int/en/know-how/technologies/safety-security/automotive-cybersecurity/#c2941>. Accessed 10 September 2020.
- Vivas, J. L., Agudo, I., & Lopez, J. (2011). A methodology for security assurance-driven system development. *Requirements Engineering*, 16, 55–73.
- Walia, G. S., & Carver, J. C. (2009). A systematic literature review to identify and classify software requirement errors. *Information and Software Technology*, 51(7), 1087–1109.
- Wang, C., & Winner, H. (2019). Overcoming challenges of validation automated driving and identification of critical scenarios. *Proceeding IEEE Intelligent Transportation Systems Conference (ITSC)*, 2639–2644.
- Yi, L., He, H., & Peng, J. (2016). Hardware-in-loop simulation for the energy management system development of a plug-in hybrid electric bus. *Energy Procedia*, 88, 950–956.
- Zhan, Y., & Clark, J. A. (2008). A search-based framework for automatic testing of MATLAB/Simulink models. *Journal of Systems and Software*, 81(2), 262–285.
- Zhou, J., Zhang, Z., Xie, P., & Wang, J. (2015). A test data generation approach for automotive software. *IEEE International Conference on Software Quality, Reliability and Security*.

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Pedro-Miguel Ortega-Cabezas** has a degree in Industrial Engineering and a Master in Electrical, Electronic and Control Engineering from UNED (Spanish University for Distance Education). Nowadays, he is doing his PhD in Industrial Engineering at UNED.

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





**Antonio Colmenar-Santos** has a PhD in Industrial Engineering and a Master of Science in Industrial Engineering, with a specialization in Electronics and Automation Engineering awarded both by The School of Industrial Engineering at National Distance Education University (UNED); and a Bachelor of Science in Electrical Engineering, with a specialization in Electronic Instrumentation, Regulation and Control and Industrial Automation awarded by The School of Industrial Engineering at the University of Valladolid.

He has been part of the Spanish section of the International Solar Energy Society (ISES) and of the Association for the Advancement of Computing in Education (AACE), working in different projects related to renewable energies and multimedia systems applied to teaching. He has been a coordinator of both virtualization and telematic Services at ETSII-UNED, as well as deputy head teacher (administration) and Head of the Department of Electrical, Electronics and Control Engineering at UNED.



**David Borge-Diez** has a doctoral degree and a master in industrial technologies research from UNED (Spanish University for Distance Education) and a bachelor's degree in industrial engineering with majors in energetic engineering from the University of Valladolid, Spain. He is a specialist engineer in energy efficiency, energy economics and alternative energy sources and works as a Professor in the Department of Electrical, Control and Automation Engineering in the University of León, Spain. He has been Associated Professor in the University of León and worked for energy companies in both public and private projects, including international R&D programs.



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



**Jorge Higuera-Pérez** graduated in environmental sciences and agricultural engineer specializing in mechanics, his twenty years of professional experience in the automotive sector has been gained both in companies manufacturing automobiles such as PSA GROUP or companies manufacturing automotive components such as HUTCHINSON. During all this time he was able to perform different functions such as manufacturing quality and being a project manager.



**Eric Alcaide** Physics and Medicine undergraduate student. His main research interests include application computational methods to different areas such as engineering, biology and chemistry.