# An Automated Continuous Integration Multitest Platform for Automotive Systems

Boyang Du, *Member, IEEE*, Sarah Azimi, *Member, IEEE*, Annarita Moramarco, Davide Sabena, Filippo Parisi, and Luca Sterpone, *Member, IEEE*

***Abstract*—Testing has always been a crucial part of application development. It involves different techniques for verifying and validating the features of the target systems. For a complicated and/or complex system, tests are preferred to be carried out in different stages of the development process and as early as possible to avoid extra costs due to the errors caught at later stages. With the increasing system complexity, the cost of testing is also increasing in terms of resources and time, which introduce further impact against development constraints such as time-to-market. On the other hand, more and more associated electronic components lead to an ever-increasing system complexity in high reliable applications such as automotive ones different from heterogeneous systems such as advanced driver assistance systems, sensor fusion systems, etc. In this article, we present a testing framework utilizing the continuous integration (CI) solution from software engineering, a commercial virtual platform, and a hardware field programmable gate array based verification platform focusing on the engine control unit to demonstrate the feasibility of the proposed method. The efficiency and viability of the CI method have been demonstrated on a real heterogeneous automotive system.**

***Index Terms*—Automotive electronics, reliability, system validation, system verification, testing.**

## I. Introduction

TESTING, as a crucial part of system development, has attracted a lot of attention over the years, especially when electronic devices are concerned. It involves steps at different stages of development for verifying and validating the features and functionalities of the target design.

Taking integrated circuit (IC) design as an example, different techniques have been developed for guaranteeing the correctness of the design at different development phases. Simulation-based techniques could be applied at different abstraction levels, from system behavioral level to register transfer level and down to post-synthesis/post-layout gate level, where the developer has fine observability and control over the internal behavior of the design. As the abstraction level goes down, the simulation results become more accurate w.r.t. the real system; however, the cost in term of simulation time and resources increase rapidly along the scale of the target design.

Meanwhile, emulations could be carried out for speeding up the verification process as low-level details are ignored. It is quite beneficial to have an emulation platform especially when both hardware and software developments are involved.

The emulation platform usually allows hardware and software co-design by providing software developers a base platform to start even without real hardware. Besides, software developers could also provide feedback to hardware developers at an early stage to tune the system regarding features, performances, or other constraints to avoid cost due to hardware modifications at a later stage.

As the complexity of the target design increase, testing cost including time and resources (e.g., power consumption) becomes one of the major constraints, besides the desired level of coverage. Automated test pattern generation (ATPG) is often used for generating test inputs for IC design test with the optimization of reducing the number of input vectors while maintaining an acceptable test coverage, in order to reduce the time cost and power consumption for executing test (considering the volume of manufacturing). Besides, when design complexity scales up, especially when it involves both hardware and software, it quickly becomes infeasible for manually generating test inputs (or at least the entirety of it, as expertise of the internal knowledge of the design, could ease the optimization effort of ATPG).

However, for some applications such as automotive, low-level testing techniques are not sufficient to guarantee the adoption of qualified commercial-off-the-shelf (COTS) components within an automotive system. In general, COTS components adopted in automotive systems undergo various kinds of tests, from the manufacturer to the final user (e.g., manufacturing and low-level input/output tests); however, these tests do not certify that functions that *Car Makers* are interested in are verified and, thus, properly pass all the functional requirements of the integrated automotive system.

Thus, regarding testing throughout application development, the focus is more on functional test and system integration test. Nevertheless, techniques commonly used for IC design testing (and software test) could be adopted in automotive application testing, especially considering the number of electronic components in modern vehicles is increasing rapidly with blooming Internet-of-Things technology and the rise of hybrid and electric cars.

Manuscript received October 19, 2020; revised February 2, 2021; accepted March 11, 2021. *(Corresponding author: Luca Sterpone.)*

Boyang Du, Sarah Azimi, and Luca Sterpone are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Torino, Italy (e-mail: boyang.du@polito.it; sarah.azimi@polito.it; luca.sterpone@polito.it).

Annarita Moramarco, Davide Sabena, and Filippo Parisi are with the PUNCH Torino S.p.A, 10129 Torino, Italy (e-mail: annarita.moramarco@punchtorino.com; davide.sabena@punchtorino.com; filippo.parisi@punchtorino.com).

Digital Object Identifier 10.1109/JSYST.2021.3069548

On one side, the growth of the number of electronic components in modern systems brings new challenges for testing as even though qualified COTS could be used. Moreover, they still need to be tested by the manufacturer either for an individual functional test or for system integration (or both) where hardware and software (and mechanical) components must be tested. Since the software is involved, the availability of a hardware emulation platform could allow the software development and test to be carried out in parallel with hardware (and mechanical) development, thus greatly reduce the cost in terms of time.

On the other side, automotive applications as safety-critical application have further constraints, such as the ones defined in standards [1], to satisfy besides the requirements for functional requirements. It is not enough to perform tests under normal conditions, not only because of the probability of failure of the COTS components used but also considering even at sea level electronic components could still be affected by radiation particles [2], which could lead to single event effects (SEEs) and propagate further causing system-level misbehavior. This phenomenon is becoming more and more serious and frequent as the IC technology progresses, as smaller node dimensions, higher clock frequency, and lower voltage make the components more susceptible to SEEs induced by radiation particles [3]. Different techniques could be applied to perform such tests. For example, a simulation-based fault injection campaign as one of the low-cost techniques could be used to mimic the SEEs in the components and investigate the design behaviors. Radiation test experiments using an accelerated particle beam to induce the SEEs could provide more accurate results related to radiation effects.

Thus, as per the focus of this article, the test has to be taken into consideration at an early stage of the design and with consideration of abnormal operating conditions, such as component misbehavior due to radiation effects. In this article, as the main contribution, a new test framework taking advantage of continuous integration (CI) solution and different test platforms for automotive applications with a unified API to allow automatic test cases generation across test platforms is presented. The rest of the article is organized as follows: Section II introduces state-of-the-art techniques regarding the CI solution and available test platforms for systems including automotive ones; Section III presents the proposed framework in detail; Section IV provides experimental results and analysis; Section V draws conclusions and discusses future work.

## II. State of the Art

Different techniques have been used to evaluate the reliability of hardware electronic systems, such as fault tree analysis and reliability block diagrams. When the integrated system embeds hardware and software modules, the analysis is increasingly difficult due to the heterogeneous characteristics of the system. Several kinds of approaches exist for evaluating the reliability of complex systems. They are grouped into two main categories: simulation- and emulation-based approaches. Simulation tools are based on a model of the system under evaluation and they may include fault injection capabilities in order to force the simulation of the system to unexpected behavior. On the other side, emulation approaches rely on a hardware prototype or on the effective hardware device used on the final manufacturing stage of the system, which allows the verification of the system functionality when hardware components are used. Fault injection methods for emulation platforms are also available but generally have different levels of applicability due to the intrusiveness of the approaches.

High-reliability systems integrate different electronic devices typically architected by heterogeneous system-on-chip designs. Due to the significant complexity, modern simulation approaches such as the work of Mishra *et al*. [4] rely on model-based solutions including hardware-level simulation engines that perform post-silicon validation. However, when the complexity of the system broads, these kinds of solutions became extremely time-consuming, therefore high-level model-based design and simulation approaches [5] are increasingly adopted to cope with the challenges of complex system simulation.

On the other hand, the evaluation of the reliability of complex systems does not only require efficient and accurate simulation tools, but it is necessary to adopt methods capable to inject faults into the system. The introduction of faults into a system with the specific purpose to evaluate the behavior and provide a measurement of the reliability is recommended by several safety standards including the ISO 26262 for automotive safety [6]. Several fault injection methods were developed aiming at inserting hardware fault models (e.g., bit-flip or stack-at) into the system, in order to represent real hardware faults such as Xception [7] or GOOFI [8]. The main innovation of these tools has been the possibility to insert hardware-based faults into the system, however, their capability to insert functional faults or software faults is limited.

In order to overcome this limitation, several hardware emulation platforms were also proposed during the last decade. The main goal of hardware emulation is the possibility to perform a quasi-real-time emulation of heterogeneous systems composed of electrical machines, controllers, drive systems, and protective devices. In order to achieve hardware platforms able to accurately modeling of the system under evaluation, several approaches rely on field programmable gate array (FPGA) devices [9]. Thanks to their reconfigurable hardwired architecture, these devices can embed a large amount of logic, as well as customized digital signal processing modules, thus they are able to satisfy the demand of high precision emulation [10].

The role of FPGA in today's hardware-in-the-loop (HIL) emulation is extremely growing. Several approaches are based on computational acceleration only, such as fixed-point computations [11] or floating-point hardware emulation used as computing estimators [12]. FPGA can be effectively used also to perform fault injection during the emulation of the system. In this case, the fault injection mechanism is based on the insertion of errors within the configuration memory of the device [13]. FPGA-oriented fault injectors are very popular since they allow us to perform the insertion of errors into a specific area of the device and to control accurately the precision of the process. Even if efficient and accurate, previously developed FPGA-based fault injection has limitations with the type of

emulated systems. Generally, the system under emulation is a stand-alone computational unit not including other heterogeneous components. Therefore, in order to perform the overall emulation or simulation of the entire system, it is necessary to create an ad hoc interface module between other platforms. This aspect limits the effective usage of FPGA accelerators in context where heterogeneous components are used. One key concept of our approach is acting this limitation. Thus, we create an agile method to perform interoperability between the emulated hardware systems, the overall HIL platform and the simulation engine.

The concept of agile methods has progressively taken its popularity in software engineering as well as in the area of embedded system design. The adoption of eXtreme Programming and CI could speed up the development process and tighten the communication among developers of different roles comparing to the traditional waterfall model [14] to allow more flexible and faster response to requirement change and customer feedback (which includes test report from test engineers as for designers). There are already plenty of research works and user reports of adopting agile methods in embedded system design, as described by Kaisti *et al*. [14]. Due to the diversity of different characteristics posed by various embedded system applications, such as real-time constraint and reliability requirements, certain practices in agile methods need to be tuned when applied in the embedded environment. For instance, Ronkainen and Abrahamsson [15] pointed out that the beginning phase of architectural design could not be avoided, and documentations and specifications have to be managed in a suitable way. When hardware is involved, Lima *et al*. [16] argue that at the beginning of the project constraining the availability of usable resources through early-stage software and hardware co-design experiments using prototyping techniques could help to avoid the later stages being affected by whole-project changes.

In the automotive field, component and/or system behavioral modeling for simulation or emulation has already been used for a long time to tackle problems such as vehicle emission control [17]–[20], energy optimization [20]–[22], etc. HIL could be used at a later stage when (part of) hardware is available to verify the result of simulation/emulation from an earlier stage and/or to validate the software implementation, for example, in [23]–[25]. Similarly, in the aerospace field, model-based system engineering is a widely adopted methodology for efficient system analysis and test rules insertion as proposed by Zhang *et al*. [26].

Meanwhile, as the development of a complex and complicated system usually involves a team consists of designers and engineers acting in different roles, tools, and services have been developed to manage the collaboration among the team members. One type of such tool is Version Control which allows multiple developers to work on the same or different parts of the design simultaneously. One of the most popular open-source version control systems (VCS), at the time of writing, is Git which supports branches, multiple workflows, etc. Based on Git, there are several options for repository hosting and more importantly the features for supporting CI, such as GitLab and GitHub. Also, there exist standalone automation servers such as Jenkins, where automation jobs could be configured to bind with the Git repository and configured using Jenkinsfile describing CI pipeline in different stages and steps as proposed by Liscouet-Hanke *et al*. [27].

There are also research works of applying them for applications in the area of embedded system design. Smart [28] presented the application adopting test-driven development and CI for the embedded system, in which with the test software developed in Ruby, they are able to interact with the real hardware to apply input stimuli and gather output data for testing.

The main contribution of the proposed article is the development of a unified testing infrastructure for the automatization of the CI test of complex systems. The unified testing infrastructure allows the application of identical input stimuli to the system under evaluation and to extract the behavior of the system at different implementation levels: software emulation level, hardware level, and HIL. The proposed platform improves the testing capabilities of previously developed approaches thanks to the flexibility of applying test algorithms at different implementation levels and to compare the system output responses.

Our approach is filling the gap by combining several testing platforms and techniques into a single integrated platform where it is possible to compare the tests and to obtain meaningful data. The proposed approach allows us to discriminate the source of the failure (software emulation, hardware emulation, or HIL) and to perform an investigation within the specific platform. Furthermore, the tests can be executed simultaneously since the developed platform adopts a CI approach, therefore providing an evident reduction of the overall requested testing and debugging time of the full system.

## III. Proposed Testing Infrastructure

In this article, we propose a new testing infrastructure that unifies different platforms targeting different stages of development, taking advantage of existing CI tools in software development. The utilized platforms include a reference model implemented in Python, an emulation-based validation environment (EVE), an FPGA-based validation platform (VP) as presented, and the final HIL testbench. The proposed testing infrastructure is able to generate test cases targeting different systems taking into account the availability of different features of each platform and gather test results to provide feedback to designers. The overall architecture is illustrated in Fig. 1.

The proposed framework is based on a unified interface for multiple test platform working at three different levels: software emulation level, hardware emulation, and HIL. The first level is supported by software emulation tools also known as Virtualizer tools. These kinds of tools provide an architectural simulation of the system under evaluation and they are generally programmed by the software application. The framework may be applied to any kind of electronic system characterized by heterogenous hardware modules, such as electronic systems with high dependability requirements. For the sake of this work, we rely on the Synopsys Virtual Development Kit tool which has a full model of the Renesas 850 processor core architecture and its analog and discrete components. This abstraction level is effective for

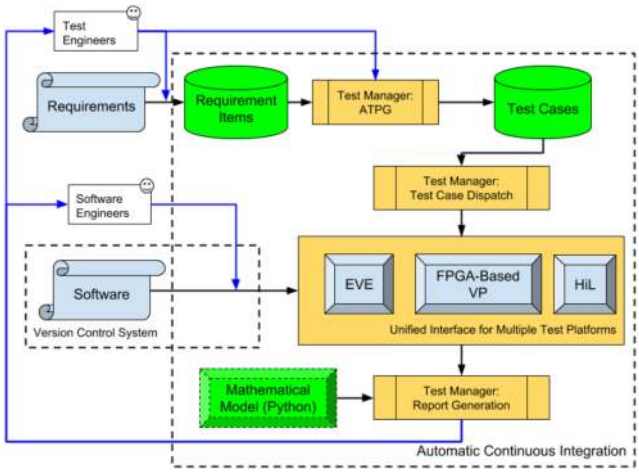


Fig. 1. Overall architecture of the proposed system test framework. Please note that the blue arrows identify test and software engineer's input and output controls, while the black arrows are related to an automatized flow.

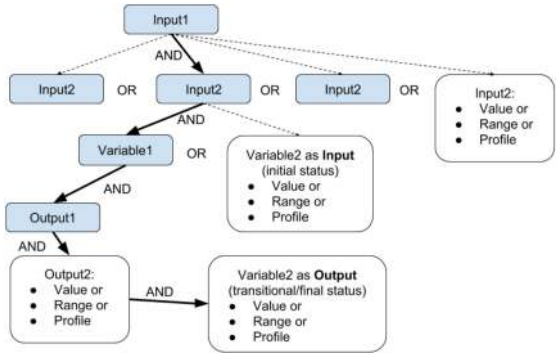


Fig. 2. Requirements represented in the tree structure, each path is a specific requirement item, each node can contain the value or range or profile of a specific input, output, or variable.

analyzing the behavior of software routines, but it does not provide any evidence of malfunctions at the hardware level. The second level is implemented by the FPGA VP. This platform is adopting a hardware description of the system that will be executed on an effective hardware device with tunable timing resolution capability. The main advantage of this platform is the possibility to observe the behavior of the portion of the system when the hardware is used, thus considering effective timing. Besides, this level has the possibility to perform hardware-based fault injection thus forcing through the FPGA platform localized and specific faults into the running circuit. Finally, the last level is the HIL which consists of the higher level implementation of the system using effective hardware (not the emulative device) and discrete modules. At this level, it is possible to observe the effective behavior of the system. All the three different abstraction levels are managed by the unified test infrastructure, this means that each platform will receive a similar input test pattern and will produce test results that can be directly compared or analyzed considering the effect of the faults and behavior on the different abstraction level platforms.

### A. Testing Requirements

The goal of testing is to verify and validate the implementation of the system against the requirement document that defines not only the features of the target system but also the expected correct responses from the system under different scenarios. Furthermore, regarding automotive applications, standards such as ISO-26262 [1] need to be complied with to guarantee road safety.

However, such requirement documents are usually handwritten by system architecture designers, which cannot be processed directly by software without the aid of natural language (NL) processing algorithms. Regarding this, there have been research specifically for requirement engineering for decades, which not only includes the methodologies for identification and definition of requirements but also for the communication of requirements across the development team [30], [31]. Several steps were introduced for structural analysis of requirements definition for system design by Huang *et al*. [23]. Ross and Schoman *et al*. [32] proposed a method to automatically convert requirement documentations to a formal language using NL methods. However, the automated conversion is not in the scope of this article, instead, the test engineers are in charge of such conversion, and in this way, the product of conversion can be guaranteed to be consistent with the original requirements and standard-compliant, for example, the traceability required by ISO-26262 [1].

To extract the test cases from the requirement documentation, the system is modeled as a box with inputs, outputs, and variables with different values that define the state of the system. As an application involving both hardware and software, the inputs include both the hardware signal and software input; the same goes to outputs; while variables are referring to all the internal data indicating the status of the system (or certain module of the system) including register in the concept of hardware or data in software (though eventually they are also stored in hardware structures).

We looked into the file formats used in nowadays software development. The JSON file format was determined to be used for storing processed requirements and test cases. The format is quite straightforward and could be used to express the requirements as a tree as shown in Fig. 2.

After the requirements are processed, test cases are generated to cover each requirement item and stored in the JSON file. The generation of the test cases is performed by the test manager Automatic Test Pattern Generator (ATPG). It elaborates the required input data file and it generates a requirement on the basis of coverage metrics on the tree structure. An example of the generated test case is the profile of a synchronization signal characterized by a duty cycle of 40%, a maximum voltage value of 2 V and a resolution of 150 $\mu$s. In our proposed framework, multiple test platforms could be utilized to allow tests in different abstraction levels which correspond to different stages of application development. However, one thing needs to keep in mind is that different abstraction level means different controllability and observability of the input, output, and variables in the target application. So, when the test cases are generated, depends on the specific requirement item, not all test platforms could be

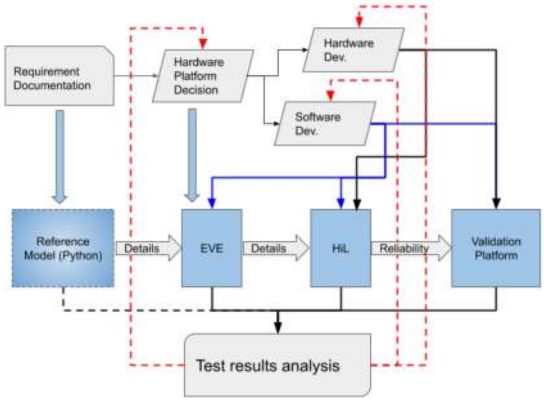


Fig. 3. Multiple platforms available at different stages of system development. Please note that the bold azure arrows report the user translation from documentations to computational models and tools control. The black and blue arrows are instead related to the automatic data flow for the hardware and software interface, respectively.

utilized for such test cases. Though for those test cases where multiple test platforms can be utilized, the comparison of the results brings in new and further insight into the behaviors of the target application.

### B. System Reference Model

Once the requirements are processed, a reference model could be built as shown in Fig. 3. The reference model in the case study of this article has been implemented in Python, considering only the inputs and outputs specified in the requirements to provide the golden reference data to be compared with test results from other platforms. The developed platform is applicable to any kind of system where the three description levels of the system, which are software emulation level, hardware emulation, and HIL, are available. However, the approach can be applied also when two of them are available, specifically the software emulation level and the HIL. In case the hardware description model is not available, it will not be possible to inject faults at the hardware level, however, it will be still possible to perform a fault injection campaign to simulate hardware faults at the software level. Besides, please note that HIL and FPGA-VP platforms can be replaced by each other or eventually excluded if a specific model of the system is not available (e.g., it is possible the VHDL or hardware description for FPGA is not available).

For the purpose of this article, we selected the engine control unit (ECU) as a case of study. For this purpose, we defined a reference system that in the case of the ECU includes the reference system for generating the input signals, such as Crankshaft and Camshaft signals, and the outputs including an injection pulse and a PWM signal. The model is using the following parameters.

1) *Revolutions per minute (RPM):* It indicates the engine speed. The current model supports static (single value) and dynamic (acceleration or deceleration speed profile) RPM as input.
2) *Crankshaft mode:* Normal, missing, and spurious tooth: the normal Crankshaft signal is a periodical pulse signal with a certain number of pulses (teeth) per revolution of the engine and a gap between each revolution. However, due to possible abnormal behavior of the sensor, the Crankshaft signal could be erroneous, with one or more tooth missing, noted as missing tooth. The spurious mode allows users to inject glitches in the Crankshaft signal.
3) *Missing tooth index:* This parameter can be settled in order to indicate which tooth is missing. It is used during the missing tooth functional model of the system.
4) *Spurious tooth index/duration:* It is a parameter used to model the glitch in the Crankshaft signal.
5) *Injection pulse mode:* Time-based or angular. It is a parameter used to select the operational model on a timing or angular reference.
6) *Injection pulse angle/time/duration:* It allows to control the different parameters related to the injection pulses. The controlled parameters include the multiple pulses that can be generated, the time or angular property, the pulse starting position, and the duration.

### C. Emulation-Based Virtual Environment

When the target system involves both hardware and software design, which is quite common in the embedded system world, an emulator for hardware platform is beneficial as it enables the software developers to start without being held back by the hardware availability and to be able to identify potential hardware or systemic problems at an early stage as the cost of re-design/re-manufacture hardware is much higher than the cost of software development.

There exist plenty of emulators commercially or as an open-source project. For example, QEMU is a widely-used open source machine emulator and virtualizer [33]; the gem5 simulator provides a modular platform for computer-system architecture research [34] which could be used in co-simulation with SystemC providing more flexibility and extendibility.

As shown in Fig. 3, the integration of EVE could be utilized once the hardware platform is determined to allow software developers to start immediately without waiting for the real hardware platform to become available. Furthermore, EVE usually provides more controllability and observability over HIL solutions, especially the features for software debugging. Thus, it provides a low-cost solution for preliminary stress tests. The functionality of the EVE platform is based on the virtual platform tool and the interface module with the other parts of the validation framework. A conceptual scheme of the virtual platform environment is illustrated in Fig. 4. The execution of virtual tools requires a system description in input which is transmitted as a set of resource locations. The resource locations are then used to perform the selective injection of faults into the emulation of the system. The control of the simulation is performed by a scripting language (e.g., typically *TCL* or *Python*) that is parametric and configured for each test case dispatch generated by the test manager. During the elaboration of each test, two logs are generated: one related to the system trace, which reports the value of the signals observed by the user; the second is a fault injection log, which reports the classification of each test with respect to the specific fault affecting

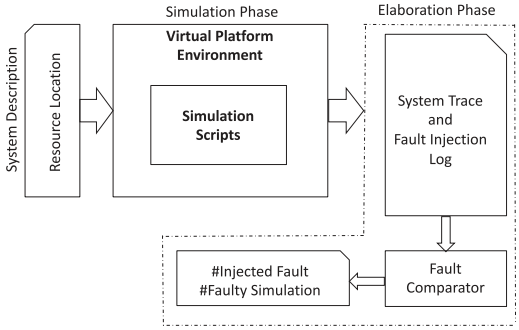


Fig. 4. Conceptual representation and interface of the virtual platform and the interface with the multiple platform environment.

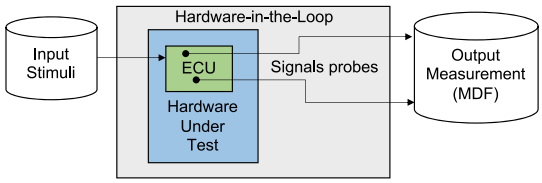


Fig. 5. HIL platform for input stimuli application and MDF signal probes extraction.

the system. In order to perform this mechanism, we developed a fault injection environment on the basis of [35] in order to evaluate the reliability of an automotive application regarding single event upset (SEU) in the memory. Though such injection is also possible with real hardware with the aid of a debugger, the EVE solution still provides an easier approach to enable the designer to have an early-stage assessment of the application regarding both hardware and software which could eventually affect the decision-making of the system architecture and reduce the cost by moving possible hardware/software architectural modification to an earlier stage of development.

### D. HIL Platform

When the hardware platform or at least part of it is available, HIL could be utilized along with software for testing to identify possible issues that have not been caught by EVE as certain aspects of the system, such as environment noises, process variations either mechanical or electrical, which are difficult to model accurately in the simulator or emulator.

In this article, the used HIL solution is able to put the device under test in a close loop, provide stimuli input and record the output (measurement) values in measurement data format (MDF). As mentioned before, the proposed framework includes a unified interface to apply test inputs, collect outputs, and perform test result analysis. The MDF files collected from HIL platform are converted to a common log format as in other test platforms to ease the effort for comparison later. In our case study based on the ECU, we considered input signals, the Camshaft and injection pulse signals, that were extracted adopting signal probes connected to the hardware under test relevant nodes within the ECU board, as illustrated in Fig. 5.

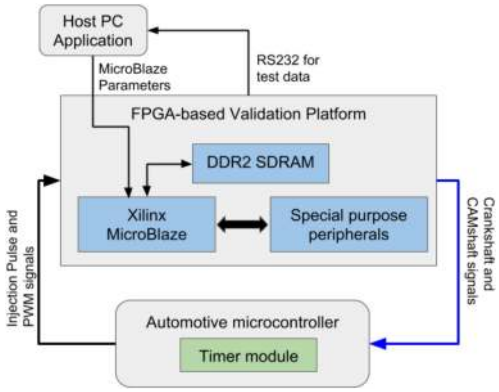


Fig. 6. FPGA-based VP for automotive applications.

### E. FPGA-Based VP

Comparing to commercially available HIL solutions, FPGA-based solution as proposed by Kim *et al*. [29], i.e., FPGA-based VP, has the advantage of flexibility and is fully controllable by the developer. Though the time cost for developing such a platform has also to be considered, once the base system is implemented, it is quite straightforward to add custom components for more features. The developed platform is illustrated in Fig. 6. Since the automotive microcontroller with the timer module is the target under test, the VP emulates input stimuli signals coming from Crankshaft and Camshaft position sensors to driver software and monitors the fuel injection pulse signals driven by the software. The platform can generate the signals under different scenarios, according to different profiles customized by the user, which include the normal static speed test, dynamic (acceleration/deceleration) speed test as well as faulty signals such as jitters, corruption, and missing signals. Similar to the reference model, the VP is implemented in such a way that the user could send parameters to the MicroBlaze processor in the design to configure the peripherals to generate Crankshaft and Camshaft signals in different conditions.

Since the automotive microcontroller with the timer module is the target under test, the VP emulates input stimuli signals coming from Crankshaft and Camshaft position sensors to driver software and monitors the fuel injection pulse signals driven by the software. The platform can generate the signals under different scenarios, according to different profiles customized by the user, which include the normal static speed test, dynamic (acceleration/deceleration) speed test as well as faulty signals such as jitters, corruption, and missing signals. Similar to the reference model, the VP is implemented in such a way that the user could send parameters to the MicroBlaze processor in the design to configure the peripherals to generate Crankshaft and Camshaft signals in different conditions.

Besides, a host PC application has been developed to retrieve test results from VP through UART (RS232) connection and MATLAB scripts for automatic test report generation. For example, one of the generated test reports could be directly used to verify if the injection pulse signal is still in the acceptable window when a faulty Crankshaft signal is generated.

TABLE I
EXAMPLE OF CAPABILITIES OF INPUTS, OUTPUTS, AND VARIABLES IN DIFFERENT TEST PLATFORMS

| Name | Platforms |
|---|---|
| I[1]:RPM | EVE, VP, HiL |
| I: Crankshaft Mode | EVE, VP |
| I: Missing Tooth | EVE, VP |
| … … | … … |
| V[2]: #Injection Miss | EVE |
| O[3]: Injection Pulse | EVE |

1.: Input, [2]: Variable, [3]: Output.

In the developed framework, VP could be used as part of the HIL solutions to provide support for abnormal test cases to verify the system reliability under different faulty conditions, i.e., fault injection capability. One of the possible faults when targeting the ECU is the missing tooth in the Crankshaft signal. In this case, the ECU is able to decode the exact position of the engine, i.e., the position of the piston of each cylinder. By the (mechanical) specification, fuel injection needs to be triggered within certain temporal or angular windows which are crucial to the overall performance of the engine. However, the Crankshaft sensor could behave erroneously either due to the sensor itself or environmental factors, e.g., vibration and radiation effects, leading to unstable output, such as missing teeth or even complete loss of signal.

Under such conditions, one of the important tests is to verify whether the ECU is still able to synchronize the engine position (with only the Camshaft signal) and generate the injection pulse within a reasonable error margin or not. With the proposed VP, such input conditions could be generated with the ability of static RPM or dynamic RPM profile. Please note that with EVE, such tests could also be performed as Zhang *et al*. [34], even with higher controllability and observability of the internal behavior of the microcontroller and software. However, the two platforms are utilized at different stages that the VP needs to be in a close loop with real hardware platforms to get high fidelity test results with software running at speed. This is exactly the purpose of the proposed framework: to allow certain tests to be executed as early as possible so that early test result analysis could tune the hardware platform and software development at an earlier stage to reduce overall cost, as shown in Fig. 3, while the more realistic and accurate result could be obtained later using real hardware.

### *F. Unified Test Platform Interface*

In different test platforms, it requires different implementations to control and observe inputs, outputs, and variables in the target design which are also subject to different accessibility (or availability). In the proposed framework, a unified interface is designed to generate test cases targeting multiple platforms if possible, thus increase test efficiency and allow tests as early as possible. The framework identifies which platform could be used by checking the compatibility with the inputs, outputs, and variables involved in the test case against each platform. Table I gives an example regarding the compatibility. When two test cases, as shown in Fig. 7, are to be executed, each of the involved inputs, outputs, and variables is checked against Table I, so that available test platforms could be determined. As in this article, only EVE is capable of logging the number of missed injection windows as such information is stored in a variable in the software under test; HIL does not have the capability for injecting faulty Crankshaft signal. Please note that in reality, such a table contains more detailed information for the same input, variable, and output, different platforms may have different levels of controllability and/or observability. One of such cases is the limitation of the clock frequency of the FPGA design, while in EVE, the time resolution could be much finer than that in the FPGA-based VP, this parameter can be tuned in relation to the kind of test.

However, one should keep in mind that along with development progress, the compatibility table will be changed as first, the inputs, outputs, and variables could change due to modification of hardware or software, or even the requirements; second, the controllability and observability of each test platform could change, for example, with aid of debugger and software instrumentation, the log of the injection miss number could be available in the VP and the HIL system modules. The unified interface provides a way to identify those test cases to be executed as early as possible, and a centralized database for the availability of each test platform, which should be continuously updated by designers and test engineers along with the application development. From the interface to each individual test platform, the middle-layer has to be developed as each has its own way of executing the test cases: providing input stimuli, monitoring the variables and outputs, test results collection, and postprocessing. For example, the Crankshaft signal is configured by a *tcl routine* in EVE while it is configured by sending corresponding parameters via UART connection in VP.

### *G. ATPG Module*

ATPG is widely used for complicated and complex circuits where the manual generation of test patterns is infeasible in terms of cost. Besides, the methodologies to achieve high fault coverage for well-structured design components have been well studied over the years, such as for the arithmetic logic unit and memory elements in microcontrollers. However, in IC testing, with the commonly used fault model such as stuck-at fault, the term fault coverage is well-defined, so ATPG is often implemented with the goal of maximizing fault coverage with minimal test patterns. As the functional test is concerned, the ATPG proposed in the framework is quite preliminary where it generates test cases according to the processed requirement document.

For the specific purpose of this article, the input conditions specified in the processed requirement items should contain condition coverage. For example, if a value range is provided, depending on whether there is also a resolution provided: in the positive case, multiple test cases will be generated to cover all the values in the range with step equals to the given resolution; otherwise, then depends on the exact input (and determined by test engineer), one or multiple test cases could be generated,

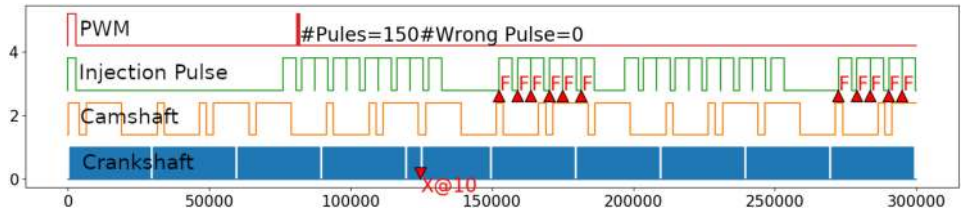


Fig. 7. Example of preliminary analysis of test results generated by report in the case a missing tooth is causing timing failure of the injection pulse signal.

for example, random value interpolation could be an effective solution. If a profile is provided, then one test case is to be generated with input following the exact profile. However, if the profile is parameterized, then multiple test cases are to be generated. For example, when acceleration is to be tested, requirement items could specify an acceleration profile (of RPM) with initial RPM not fixed (i.e., value range), then multiple test cases are generated with different initial RPM values but following exact acceleration profile.

### *H. Automated Continuous Integration Platform*

The proposed framework adopted the VCS and the automated continuous integration (ACI) modules. They are adopted together to optimize the efficiency for parallel work among team members and more importantly the utilization of the multiple test platforms. Designers with different roles could work on different aspects of the application: requirements, hardware, software, and test. Besides, developers could divide the application into modules; each could be implemented and tested without interlocking the others to make the highest usage of resources.

Due to operating system limitations at the beginning of this article, we chose Windows as the target platform which leads to the choice of VCS to be GitBlit [35] and ACI tool to be Jenkins [36]. The application repository was hosted in the GitBlit server, with a Jenkinsfile implementing the steps for building the software and launching test cases generated from processed requirement items (stored in a JSON file). The Jenkins server was used to host the CI jobs, which was bound to a specific branch of the application repository in the GitBlit server. The job in the Jenkins server has been configured to detect any commit in the application repository branch and then automatically launch the building and testing phases using the Jenkinsfile in the repository. The reason framework has been configured in the way that, in our case, the target application is the ECU, which has several different configurations with a similar code base with some modules shared. Therefore, different configurations have been organized into different branches to enable efficient code reusage. However, in this way, it means that for different configurations, requirements could be different so that the building and testing could be different. In this case, each branch will track its own test suite and Jenkinsfile for the ACI. An example of such a Jenkinsfile is shown in Listing 1. As could be seen that each step is implemented by a separate Python script.

1) Build launches building process that could happen in a dedicated remote server.
2) Flow generates the test cases through the unified interface and launches tests across different test platforms; for each platform, different initial steps have to be performed. For instance, when using the FPGA-based VP, the FPGA platform has to be initialized by downloading the bitstream file, and proper parameters have to be sent to the VP through a serial connection.
3) Report gathers data generated by the test platforms, performs preliminary result analysis, and generates test reports. An example of such preliminary analysis is shown in Fig. 6, regarding injection pulse timing verification. The red marks indicate erroneous behaviors in either input signal or output signals. In this case, there is a tooth missing from Crankshaft signal which causes timing failures in the injection pulse signal and missing PWM signals due to synchronization failure. Thus, the test case is marked as FAIL in the final summary report.

**Listing 1:** An example of the test management sequence.

```
Test_sequence {
 agent any
 stages {
 stage ('build') {
 steps {bat """python -u Build.py"""}
 }
 stage ('launch_test') {
 steps {bat """python -u Flow.py"""}
 }
 stage ('pack') {
 steps {
 bat """python -u Report.py"""
 archiveArtifacts artifacts: 'reports/**/*/*/',
 fingerprint:true
 }
 }
}
}
```

At the end of the Jenkins job, artifacts are collected as the result of the run. In our case, all the files in the reports folder as shown in Listing 1 includes not only the raw data gathered during the test but also the preliminary analysis reports as shown in Fig. 6. Artifacts then could be directly downloaded by team members to review and verify from the Jenkins server.

Another benefit of using ACI is to manage multiple (same) test platforms to further reducing time cost, as shown in Listing 1, in line 2 "agent any" is used to specify the target machine (agent) the job is to be executed could be anyone that is available. When there are multiple machines with multiple test platform setups

(EVE, VP, and HIL, etc.) available, this could be used to automatically boost resource utilization. Furthermore, parallelism is managed by the *Flow.py* to utilize different test platforms by issuing the execution of test cases as soon as the target test platform is available.

### *I. Report Generation*

For the tests generated and dispatched to different platforms, different test results data could be collected and should be processed accordingly. As in the examples mentioned above, besides FAIL or PASS of each test case, further test data, raw or pre-processed such as in Fig. 6, are also used by developers to better perform failure analysis. Besides some trivial verification procedures such as checking injection pulse location in Fig. 6 could be directly done through the unified interface using the output signal monitoring features, complicated analysis of test results requires the involvements of test engineers for generating automated procedure to be integrated into the ACI environment which is out of this article's scope.

## IV. Experimental Results

We performed the experimental analysis of the proposed test framework in order to evaluate efficiency, performance, and testing capabilities compared with traditional approaches. In this section, we will present some information for the tests we carried out using the proposed framework. In particular, with the EVE platform, we were able to execute fault injection campaigns regarding faults inside memory and I/O peripherals. The performance measurements of the proposed platform were done using a workstation Dell Alienware Aurora R8 equipped with an Intel Core i9, an NVIDIA GeForce RTX2080 GPGPU, and 32GB RAM. The VP and the HIL were executed at speed with specific settings for controllability and observability.

### *A. Case of Study: ECU RH850*

The developed multitest platform has been applied on an industrial reference as the automotive gearshift control system based on the Renesas Electronics RH850 microprocessor system. The system has been tested while executing the application consisting of the gearshift open-loop configuration. In order to emulate the realistic application execution, the gearshift open-loop application was tested using a round per minutes (RPM) signal configured as a triangular waveform input. The input was connected to the analog module of the D-space analog to digital converter (DS-ADC) in order to compare the results directly on the analog device modeled by the virtualization system.

The system is characterized by a 32-bit RISC processor core using 32 registers and working at 240 MHz. The main memory consists of 64 K cells of 32 bits. The tested application is mapped into the memory segments as reported in Table II. The application has been tested with a functionality test pattern having a real-time duration of 56 s.

TABLE II
GearShift Open-Loop Application Global Memory Segments

| Location | Size [byte] |
|---|---|
| Interrupts (Reserved) | 40,960 |
| Dynamic Data - Stack | 40,960 |
| Static Data | 98,040 |
| Executable Header | 256 |
| Code segment | 61,440 |
| Text segment | 4,096 |
| Interrupt Vectors (Reserved) | 16,384 |

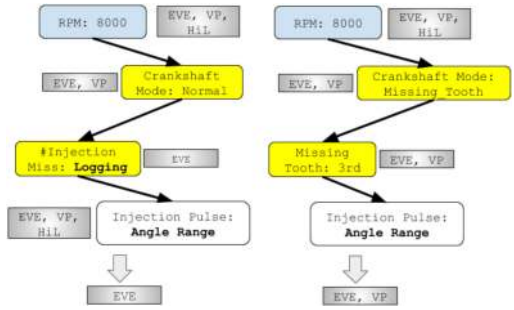


Fig. 8. Test cases generation for multiple test platforms: left test case could only be executed in EVE, while the right one could be executed in both EVE and VP.

### *B. Fault Injection Architecture*

The experimental analysis has been performed using the fault injection architecture illustrated in Fig. 8. The fault injection is executed in two phases: golden response and faulty execution. During the first phase, a fault-free execution is performed. A golden reference was generated by storing the voltage transition of the monitored signals into a golden result database. While in the case of the fault simulation, the execution phase is interleaved with a fault injection execution. The comparison with the system under injection has been performed by voltage transition comparison. In the case of the golden execution, the following phases were performed: initialization, execution, and data collection.

During the second phase, a fault is selected considering its time, location, type, and duration. The fault injection manager controls the characteristics of the injected fault and defines the time features of the fault injection campaigns such as the start, pause, and end of the emulation/simulation platform executions. In detail, the fault injection manager supports the synchronization between the input and the core functionalities and the simulation execution steps. The commands generated by the Fault Injection Manager are linked to the Renesas Core model through a proper hardware interface, in the case of the HIL and EVE testing environments, and with a software module when the VP platform is adopted. Please note that, even if the application software is synchronized with the fault injection manager, the fault injection process is not intrusive with respect to the behavior of the software under test since it affects exclusively its execution time.

The fault injection environment has been settled to perform injections of different types of faults at different levels. In particular, the VP platform is dedicated to the injections of physical

TABLE III
GLOBAL MEMORY FAULT INJECTION RESULTS

| Location | Stuck@0 [%] | Stuck@1 [%] |
|---|---|---|
| Interrupts (Reserved) | 28.0 | 32.0 |
| Dynamic Data - Stack | 14.2 | 10.4 |
| Static Data | 25.4 | 24.4 |
| Executable Header | 12.6 | 8.4 |
| Code segment | 6.9 | 14 |
| Text segment | 0 | 0 |
| Interrupt Vectors (Reserved) | 12.9 | 10.8 |

level permanent fault models such as stuck-at, coupling, and delay faults; and transient models such as SEUs or bit-flips. In detail, the tool can simulate hardware fault injection within the memory modules and the network infrastructure. Furthermore, depending on the availability of the system's processor cores the environment is able to inject faults also within registers and arithmetic cores.

For the purpose of this article, we injected faults within the user register and memory adopting two methods: access-based and time-based triggers. In the first case, the fault injection is triggered when the CPU is accessing a specific memory resource. While in the second case, the injection starts when the timer of the fault injection manager reaches the desired time. The EVE platform is instead dedicated to the injection of functional faults model such as *missing signals* when a rising or falling edge of a specific signal is not generated properly; *spurious signals*, when a glitch with a given amplitude and duration is added to the normal signal behavior and *jitter signals*, in case rising or falling edges undergo to a timing modification with respect to the original ones.

### C. Memory Modules Injection Results

The fault injection performed in the memory modules considers the RH850 Global RAM block that consists of 64 K 32-bits cells. The application under test uses around 80% of the data segment for executing the application. The overall Global RAM cells have been selected as the fault location for the fault injection campaigns. The model adopted for the first campaign is the stuck-at fault. For each run, a random cell and a random bit are selected and then changed in a stuck-at status for the whole duration of the application. The fault injection has been performed up to 1M times throughout the campaign.

The permanent fault injection results are shown in Table III, where we reported the percentage of faults generating errors for each specific location. A stuck-at fault is contributing to the wrong answers count if an error on the system output signals is generated during the application execution. The fault injection experiment required around 82 h to be completed. The monitored signals considered for the applications are the main ECU system output pins.

### D. System Register Injection Results

A second fault injection experiment has been performed injecting specifically into the ECU system registers. The injection has been performed on the user register and a signals mapping module. One register is selected randomly, and the fault injection is performed on a random bit within the selected register.

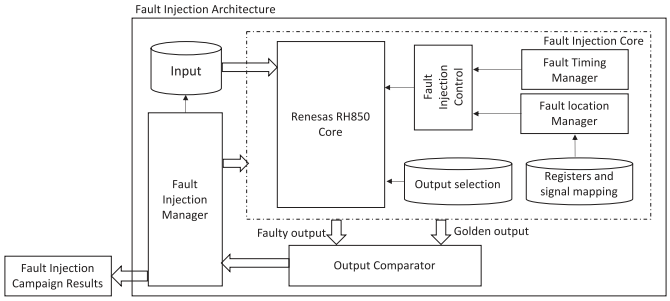


Fig. 9. Scheme of the fault injection architecture used to evaluate the performance of the developed multitest platform.

The fault injection experiment results are illustrated in Fig. 9 where we report the percentage of injected faults per register generating errors on the main ECU system output pins. The fault injection experiment required approximately 9 h to complete. Interestingly, register 2, which is related to the DS-ADC cylinder control register is critical for all the stuck-at fault injected within his values during the application execution time.

### E. System Architecture and Functional Injection Results

A third fault injection experiment has been performed to evaluate the system architecture robustness versus the injection of functional faults. For this purpose, the main control signals have been extracted using the EVE and VP environments. Furthermore, the fault injection campaigns have been performed evaluating the impact of the fault among the application time, thus faults have been injected every 30 $\mu$s. For the purpose of this fault injection campaign, we adopted a functional fault model able to inject missing signals, spurious signals, and jitters. In details, the missing signal has been configured in order to remove an entire tooth of the signal; the spurious signal condition has been configured in two cases, a 5-ns glitch and a 30-$\mu$s pulse; finally, the jitter has been configured as anticipated and postponed with respect to the rising edge of the original signal for a duration of 10 $\mu$s. All the cases were injected into the four ECU engine synchronization signals such as the PWM signal, the Fuel Injection signal, the Camshaft, and the Crankshaft. These signals are the ones generated directly by the engine sensors on the basis of the engine speed and revolutions. For the purpose of this experiment, we adopted an engine speed of 2000 RPM.

In the case of this fault injection experiment, the EVE platform required around 2 h to be completed. The monitored signals considered for the applications are the main ECU system output pins. The functional fault injection experimental results are illustrated in Table IV.

In order to evaluate the impact of the injected fault versus the engine speed and also to evaluate the capability of the environment to detect faults with the dynamic behavior of the system, we also evaluated the injection within the crankshaft signal of a specific engine cylinder (A) at the start-angle condition which determines the synchronization of the ECU with

TABLE IV
FUNCTIONAL FAULT INJECTION RESULTS

| Location | PWM [%] | Fuel Injection [%] | Camshaft [%] | Crankshaft [%] |
|---|---|---|---|---|
| Missing Tooth | 28.06 | 13.45 | 0.04 | 46.52 |
| Spurious – glitch | 0.40 | 0.05 | 0.00 | 2.45 |
| Spurious – tooth | 40.03 | 0.06 | 0.00 | 18.37 |
| Jitter – advance | 12.48 | 2.86 | 0.00 | 0.09 |
| Jitter – postponed | 11.42 | 2.44 | 0.00 | 0.13 |

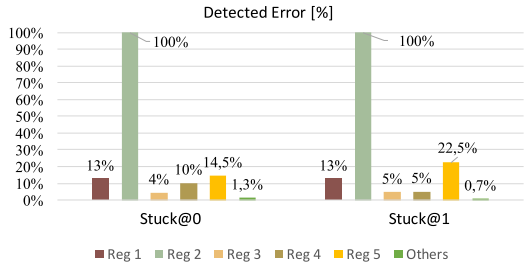


Fig. 10. Fault injection experiment results on the system registers.

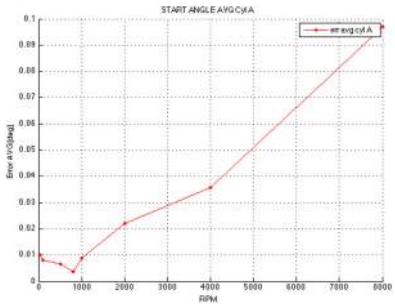


Fig. 11. Fault injection experiment results on the system registers.

respect to the position of the engine cylinders. The results are illustrated in Fig. 10 , where we reported the average error on the measured synchronization signals of cylinder A considering a normal behavior condition. As it is possible to notice, in normal condition, the measured average error is below 0.1%, please note that the evaluation has been performed at 80, 100, 400, 800, 1000, 2000, 4000, and 8000 RPMs.

### *F. Performance Analysis*

The performance of the developed multitest platform has been compared with respect to state-of-the-art tools adopted to simulate and emulate the behavior of a heterogeneous system. Please note that our development platform is, to the best of our knowledge, the first platform capable to integrate different tests at different levels but using a unified testing infrastructure.

The developed platform has been compared in timing performance on executing a functional system test and resolution of the monitored signals. In order to elaborate a fair comparison, we developed the functional test performed and described in Section IV-B with other four different platforms.

1) *Virtual platform*: The VP platform has been configured for generating, applying stimuli, and comparing the results using exclusively a virtual platform. For the purpose of this article, we adopted the Synopsys Virtual Platform tool.
2) *System C simulation*: A SystemC model of the ECU and peripheral has been described using the SystemC language. Due to the complexity of the case study, we only prepared a simplified version of the system using a MIPS core as ECU configured with a memory of 64 K words of 32-bits. Please consider that even in the case of a simplified architecture, SystemC simulation is drastically slower than the Virtual Platform tool.
3) *Emulation Debugging*: A hardware setup adopting a Lauterbach debugging cable and a system consisting of an Andorra STMicroelectronic microprocessor has been used to inject faults using the debugging port.
4) *DSpace HIL*: A HIL platform equivalent to the developed study case has been used for system test.

TABLE V
TESTING PLATFORM PERFORMANCE COMPARISON

| Location | Execution Time [h] | Execution Time Fault Injection [h] | | | | Resolution [ns] |
|---|---|---|---|---|---|---|
| | | EVE | FPGA | HiL | Total | |
| Developed Platform | 2.06 | 1.31 | 0.24 | 0.84 | 2.39 | 5 |
| Virtual Platform | 146.30 | - | - | - | 168.31 | n.a. |
| SystemC Simulation | 185.00 | - | - | - | 204.01 | n.a. |
| Emulation Debugging | 6.20 | - | - | - | 7.42 | 25 |
| DSpace HiL | 0.60 | - | - | - | 0.84 | 100 |

The obtained performance comparison results are described in Table V. We evaluated the execution time of the developed platform in case of execution of a test case without fault injection and in case of fault injection. Please consider that for the developed platform, the total execution time in case of fault injection is computed considering the average time related to the injection of an SEU into each individual platform. Clearly, the total execution time drastically varies between the different platforms since the tools have different natures. However, it is possible to notice, the proposed multitest environment has the best compromise between execution time and resolution. The HIL platform is the fastest way to test a system. However, it is necessary to take into account the elevated amount of time in order to prepare the test condition and the impossibility to reduce the resolution due to the provided hardware control mechanism. The resolution has not been provided for the Virtual Platform and SystemC since both the approaches adopted a simulation engine that cannot be compared with a system emulation. The proposed approach results in at least three times faster than traditional emulation and debugging approaches and at least five times better in resolution steps. Moreover, our approach has a resolution that is 20 times better than HIL platform. Therefore, it can be suitable to perform a fine detection of critical events and conditions that cannot be observed with HIL tests.

## V. CONCLUSION

In this article, we presented a new test platform framework for automotive applications, utilizing multiple test platforms with

different levels of observability and controllability at different stages of application development. Furthermore, a unified interface is introduced to distribute the test cases among different test platforms including EVE, FPGA-based VP, and HIL to be executed as earlier as possible. Together with the ACI solution adopted from software engineering, the proposed framework is able to automatically generate test cases from test requirement items and launch the test cases across different test platforms as soon as available. The case study presented in this article demonstrated that such a framework is feasible and could be integrated with further team collaboration tools such as VCS and cooperate with different test platforms to increase application development and test efficiency. Dedicated methodologies for automating requirement documentation parsing and ATPG for increasing test coverage, which, in turn, requires better definition in the context of requirement-driven tests in safety-critical applications, are under investigation.

## REFERENCES


[1] *ISO*, "Road vehicles-functional safety," ISO 26262-2, 2011.

[2] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[3] A. Dixit, and A. Wood, "The impact of new technology on soft error rates," in *Proc. Int. Rel. Phys. Symp.*, 2011, pp. 5B.4.1–5B.4.7.

[4] P. Mishra, R. Morad, A. Ziv, and S. Ray, "Post-silicon validation in the SoC era: A tutorial introduction," *IEEE Des. Test*, vol. 34, no. 3, pp. 68–92, Jun. 2017.

[5] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—Part I: Fault diagnosis with model-based and signal-based approaches," *IEEE Trans. Ind. Electron.*, vol. 62, no. 6, pp. 3757–3767, Jun. 2015.

[6] *Int'l Organization for Standardization*, "Product development: Software level," ISO 26262-6, 2011.

[7] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, Feb. 1998.

[8] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic object-oriented fault injection tool," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2001, pp. 83–88.

[9] H. F. Blanchette, T. Ould-Bachir, and -P. David, "A state-space modeling approach for the FPGA-based real-time simulation of high switching frequency power converters," *IEEE Trans. Ind. Electron.*, vol. 59, no. 12, pp. 4555–4567, Dec. 2012.

[10] I. Munteanu, A. I. Bratcu, S. Bacha, D. Roye, and J. Guiraud, "Hardware-in-the-loop-based simulator for a class of variable-speed wind energyconversion systems: Design and performance assessment," *IEEE Trans. Energy Convers.*, vol. 25, no. 2, pp. 564–576, Jun. 2010.

[11] G. G. Parma, and V. Dinavahi, "Real-time digital hardware simulation of power electronics and drivers," *IEEE Trans. Power Del.*, vol. 22, no. 2, pp. 235–1246, Apr. 2007.

[12] Y. Chen and V. Dinavahi, "Digital hardware emulation of universal machine and univrersal line models for real-time electromagnetic transient simulation," *IEEE Trans. Ind. Electron.*, vol. 59, no. 2, pp. 1300–11309, Feb. 2012.

[13] L. A. Aranda, A. Sánchez-Macián, and J. A. Maestro, "ACME: A tool to improve configuration memory fault injection in SRAM-Based FPGAs," *IEEE Access*, vol. 7, pp. 128153–128161, 2019.

[14] M. Kaisti *et al.*, "Agile methods for embedded systems development - a literature review and a mapping study," *EURASIP J. Embed. Syst.*, vol. 2013, pp. 1–16, 2013.

[15] J. Ronkainen and P. Abrahamsson, "Software development under stringent hardware constraints: Do agile methods have a chance?," *Lect. Notes Comput. Sci.*, pp. 73–779, 2003.

[16] G. L. B. Lima, G. A. L. Ferreira, O. Saotome, A. M. Da Cunha, and L. A. V. Dias, "Hardware development: Agile and co-design," in *Proc. 12th Int. Conf. Inf. Technol. – New Generations*, 2015, pp. 784–787.

[17] K. Ahn, H. Rakha, A. Trani, and M. Van Aerde, "Estimating vehicle fuel consumption and emissions based on instantaneous speed and acceleration levels," *J. Transp. Eng.*, vol. 128, no. 2, 2002.

[18] R. B. Noland and M. A. Quddus, "Flow improvements and vehicle emissions: Effects of trip generation and emission control technology," *Transp. Res. Part D Transp. Environ.*, vol. 11, no. 1, pp. 1–14, 2006.

[19] N. A. Kheir, M. A. Salman, and N. J. Schouten, "Emissions and fuel economy trade-off for hybrid vehicles using fuzzy logic," *Math. Comput. Simul.*, vol. 66, no. 2–3, pp. 155–172, 2004.

[20] Y. Kim, A. Salvi, A. G. Stefanopoulou, and T. Ersal, "Reducing soot emissions in a diesel series hybrid electric vehicle using a power rate constraint map," *IEEE Trans. Veh. Technol.*, vol. 64, no. 1, pp. 2–12, Jan. 2015.

[21] K. Çağatay Bayindir, M. A. Gözüküçük, and A. Teke, "A comprehensive overview of hybrid electric vehicle: Powertrain configurations, powertrain control techniques and electronic control units," *Energy Convers. Manage.*, vol. 52, no. 2, pp. 1305–1313, 2011.

[22] W. H. Lee, Y. C. Lai, and P. Y. Chen, "A study on energy saving and CO2 emission reduction on signal countdown extension by vehicular ad hoc networks," *IEEE Trans. Veh. Technol.*, vol. 54, no. 3, pp. 890–8900, Mar. 2015.

[23] X. Huang, Y. Tan, and X. He, "A torque control strategy with charge buffer for parallel hybrid electric vehicle," in *Proc. IEEE Veh. Technol. Conf.*, 2010, pp. 1–15.

[24] S. C. Oh, "Evaluation of motor characteristics for hybrid electric vehicles using the hardware-in-the-loop concept," *IEEE Trans. Veh. Technol.*, vol. 54, no. 3, pp. 817–8824, May 2005.

[25] E. Tara, S. Filizadeh, and E. Dirks, "Battery-in-the-loop simulation of a planetary-gear-based hybrid electric vehicle," *IEEE Trans. Veh. Technol.*, vol. 62, no. 2, pp. 573–581, Feb. 2013.

[26] H. Zhang, Y. Zhang, and C. Yin, "Hardware-in-the-Loop simulation of robust mode transition control for a series-parallel hybrid electric vehicle," *IEEE Trans. Veh. Technol.*, vol. 65, no. 3, pp. 1059–1069, Mar. 2016.

[27] S. Liscouet-Hanke, H. Jahanara, and J.-L. Bauduin, "A model-based systems engineering approach for the efficient specification of test rig architectures for flight control computers," *IEEE Syst. J.*, vol. 14, no. 4, pp. 5441–5450, Dec. 2020.

[28] J. F. Smart, *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. Newton, MA, USA: O'Reilly Media Inc., 2011.

[29] M. Karlesky, W. Bereza, G. Williams, and M. Fletcher, "Mocking the embedded world: Test-driven development, continuous integration, and design patterns," in *Proc. Embedded Syst. Conf. Silicon Valley*, 2007, pp. 1–15.

[30] B. Du and L. Sterpone, "An FPGA-based testing platform for the validation of automotive powertrain ECU," in *IFIP/IEEE Int. Conf. Very Large Scale Integration*, 2016, pp. 1–17.

[31] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," in *Proc. Conf. Future Softw. Eng.*, 2000, pp. 35–346.

[32] D. T. Ross and K. E. Schoman, "Structured analysis for requirements definition," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 1, pp. 6–15, Jan. 1977.

[33] B. S. Lee and B. R. Bryant, "Automated conversion from requirements documentation to an object-oriented formal specification language," in *Proc. ACM Symp. Appl. Comput.*, 2002, pp. 932–9936.

[34] F. Bellard, "qemu," 2017. [Online]. Available: git.qemu.org/qemu.git

[35] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, 2011.

[36] S. Azimi, A. Moramarco, and L. Sterpone, "Reliability evaluation of heterogeneous systems-on-chip for automotive ECUs," in *Proc. IEEE Int. Symp. Ind. Electron.*, 2017, pp. 1–16.

[37] "Gitblit," 2016. [Online]. Available: http://www.gitblit.com/index.html

[38] "Jenkins: Build great things at any scale," 2019. [Online]. Available: http://www.jenkins.io