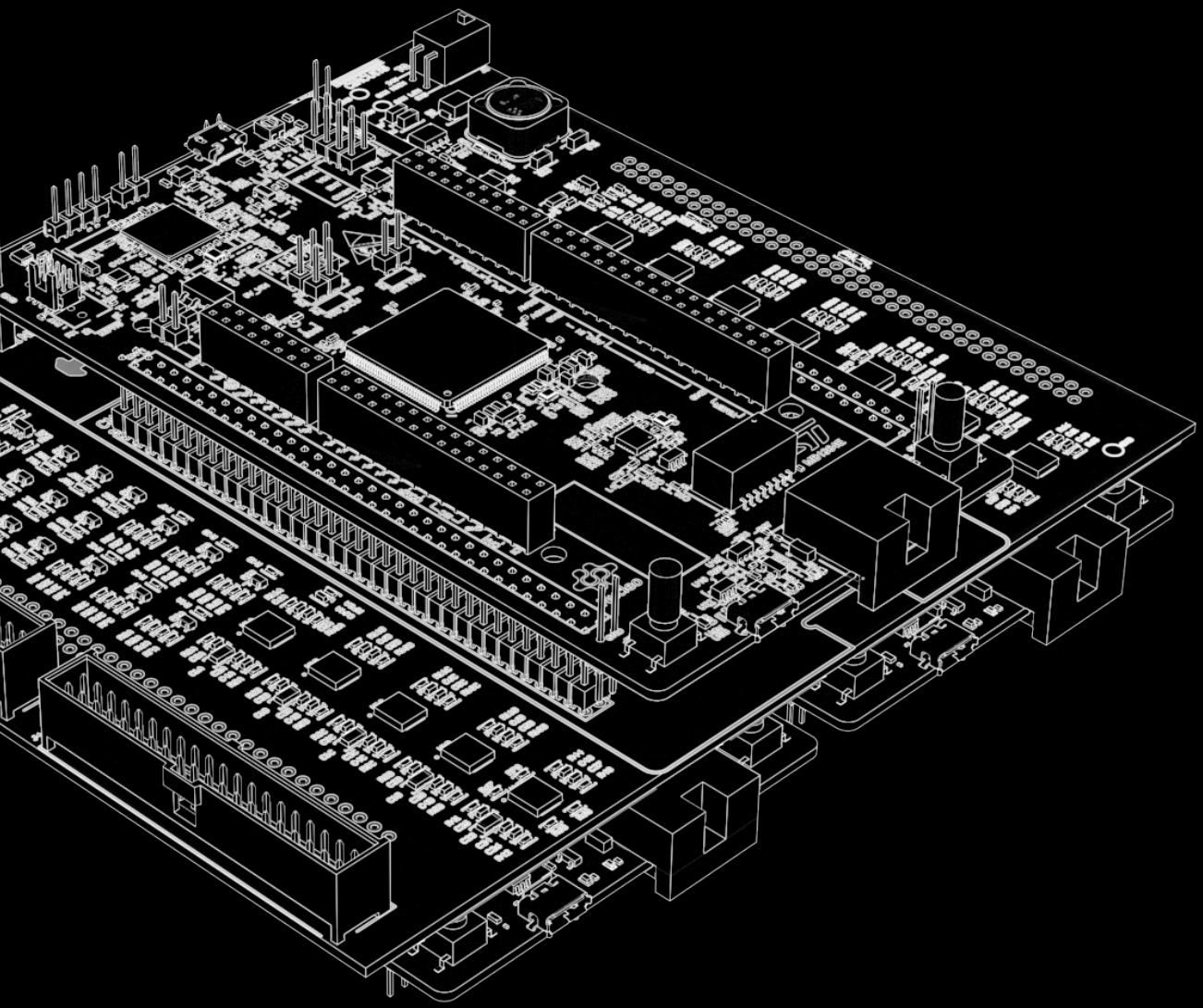
Full-Scale Research

Building a Sense & Control testing environment for the development of a hyperloop vehicle



HYPERLOOP **UPV**

Contents

List of Figures

List of Tables

- **R&D** Research and development
- **EHW** European Hyperloop Week
- **TIL** Testing-in-the-loop
- **MIL** Model-in-the-loop
- **MBD** Model-Based Design
- **SIL** Software-in-the-loop
- **TCU** Tube Control Unit
- **LCU** Levitation Control Unit
- **COG** Centre Of Gravity
- **VPS** Virtual Private Server
- **DOF** Degree Of Freedom
- **DUT** Device Under Test
- **PIL** Processor-in-the-loop
- **HIL** Hardware-in-the-loop
- **HW** Hardware
- **SW** Software
- **SHUTUP** Software & Hardware Universal Testing Unit Platform

Chapter 1

Introduction

1.1 Description of the Applicants

The Hyperloop UPV team comprises 48 members, three of which are part of the Direction of the team, six are part of the Management subsystem, and 39 are active members. The representative is Ricardo González-barranca Domingo. The research has been carried out by mainly four subsystems: Electromagnetics, Hardware, Firmware, and Software. A detailed list of team members and collaborators is shown in Table 1.1.

Table 1.1: Team members of involved subsystems.

Team board		
Lucía Zorroza	Ricardo González-barranca	Daniel González

Electromagnetics	Hardware	Software	Firmware
Raquel Murcia (lead)	Álvaro Rey (lead)	Sergio Moreno (lead)	Stefan Costea (lead)
Francisco Baixauli	David Ramón	Felipe Zaballa	Ricardo Chust
Marinao Andujar	Rubén Marz	Juan Martínez	Alejandro Gonzalvo
Álvaro Pérez	Estela Sánchez		Pablo González
Gema Acea	Daniel Montesinos		
Miquel Sitges			

1.2 General Concepts & Objectives

This section presents the foundations on which the research is based. Not only the characteristics of the verification processes in the current industry but also the environment in which the best verification strategy is going to be analyzed. Moreover, the objectives of the research will be stated.

1.2.1 Verification & Testing Fundamentals

Verification is the process of ensuring that a system or component meets its specifications and requirements. The goal of verification is to detect defects in the early development stages before the system is deployed.

Validation, on the other hand, is the process of evaluating a system or a component to determine if it meets the expectations of the stakeholders. It involves testing the system in a real-world context to ensure that it performs as expected and meets user needs.

At Hyperloop UPV, verification has the goal of checking if every component has been designed in the correct way while validation checks if they are working correctly. Verification is either a static or dynamic process while validation is always dynamic. The main difference between dynamic and static testing is that dynamic testing involves testing hyperloop by putting it to work while static testing involves analyzing the components of the vehicle and designs without functioning.

Different types of verification can be performed at different levels of abstraction and with different approaches and techniques. A complete verification strategy should be followed when developing a complex system. It ensures that the final design is reliable, safe, and meets its intended purpose identifying potential defects, errors or issues early in the development process saving time, costs and safety risks in the run.

There are mainly three types of verification: functional, performance and security.

- **Functional verification:** focuses on ensuring that the system or component operates correctly according to functional specifications and requirements. Some examples of functional requirements in the hyperloop context are acceleration, maximum speed, and braking.
- **Performance verification:** focuses on ensuring that the system or component operates correctly in terms of performance, such as speed, efficiency, ride quality, etc.
- **Security verification:** focuses on ensuring that the system or component is secure and does not have vulnerabilities that can be exploited. This includes verifying the software and systems are resistant to attacks and safe against unauthorized access or manipulation.

When designing a verification strategy three steps should be accomplished. Firstly, the identification of the verification cases, meaning identifying the requirements and specifications of the component or the system that need to be verified. Next, choosing the level of abstraction of the test bench i.e. to determine if the verification would be done at the system level, module level,

component level, etc. Finally, specifying the method of stimulus generation and data collection system –how the tests will be carried out and how the results will be collected.

For the identification of verification cases, the following techniques can be followed:

- **Static analysis:** examining the vehicle design, documentation, or software code to identify potential issues before the system is built or tested. It can include checking for compliance with industry standards, identifying design flaws, or analyzing code for potential bugs or vulnerabilities.
- **Dynamic analysis:** involves running the vehicle or component with different inputs, such as different scenarios, to identify potential issues. This can include testing the acceleration, braking, takeoff, or landing of the levitation to ensure they meet performance requirements.
- **Peer review:** get feedback and suggestions from other members of the development team or external experts.

Regarding the choice of abstraction level, the level of detail required for verification and the complexity of the system or component should be considered. The selected abstraction level may also depend on the verification goal. Some levels of abstraction in the hyperloop context are:

- **System-level testing:** Involves testing the hyperloop vehicle or system as a whole, without focusing on individual components or subsystems. This level of testing is useful for verifying the overall functionality, performance, and safety of the vehicle.
- **Subsystem-level testing:** Involves testing individual subsystems or components of the vehicle, such as the traction system, the levitation system, or the braking system. This level of testing is useful for verifying the performance and functionality of individual components.
- **Component-level testing:** Involves testing individual components of the vehicle, such as sensors, actuators, or control units. This level of testing is useful for verifying the performance and functionality of individual components and ensuring they meet design specifications.
- **Discipline-level testing:** Involves analyzing and testing the different disciplines that form each component of the vehicle, which can be tested in the following disciplines: mechanical, hardware, firmware, and software. This level of testing is useful for identifying potential implementation errors.

For the method of stimulus generation and data collection, different techniques can be used, such as:

- **Manual testing:** performs tests manually and records the results. The stimulus is generated manually by engineers while measuring and collecting the data of the process.
- **Automated testing:** uses automated testing tools to generate and execute test cases and record the results. The tests would be performed automatically when a trigger event occurs, generating a complete report of the results.

- **Simulation:** uses simulation tools to mimic the behavior of various vehicle subsystems, such as the traction or levitation systems. Performing virtual testing through simulations it is not needed for physical testing, allowing modifications and optimizations before physical prototypes are built.

Verification is a key part of developing a complex system, ensuring the final result will meet the requirements and specifications, and detecting defects in the early development process. The verification strategy should be carefully designed and adapted to the specific system or component being verified, considering the identification of the verification cases, the abstraction level, and the method of stimulus generation and data collection.

In Figure 2.1 map of the different features of the verification explained is shown.

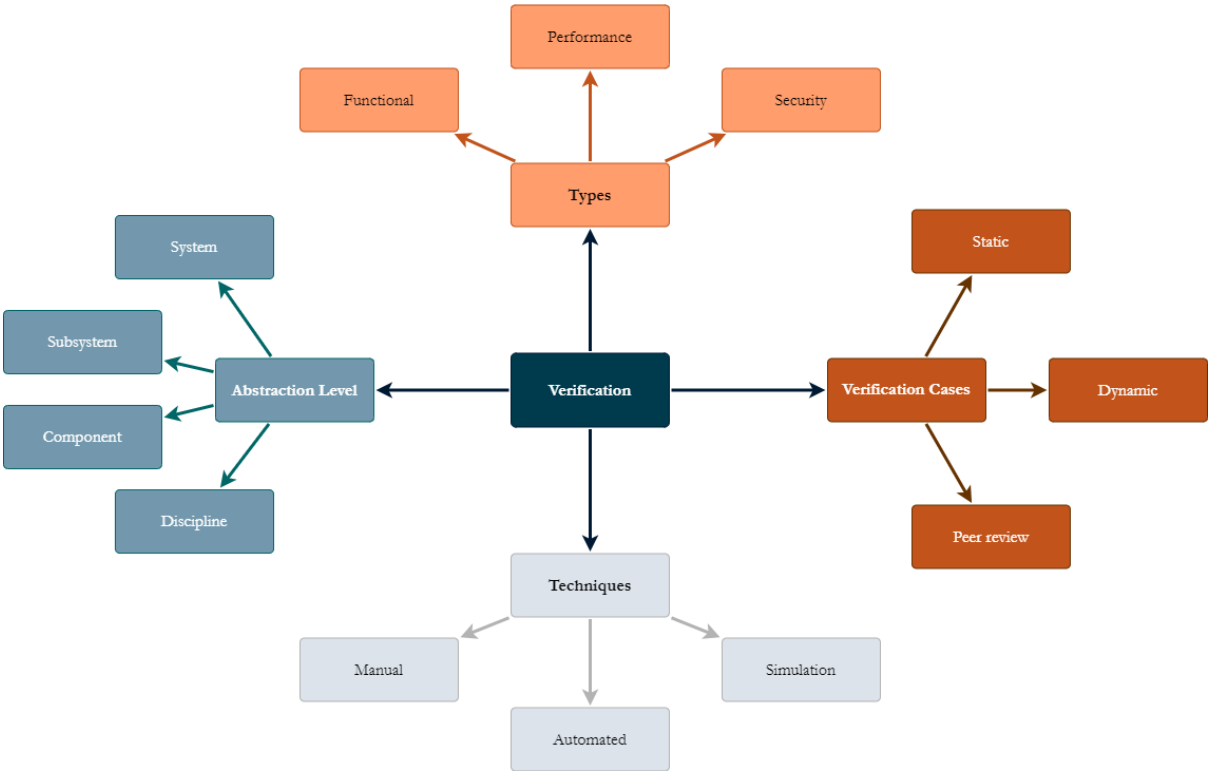


Figure 2.1: Verification features map.

1.2.2 Development Environment

This research is carried out within the framework of the European Hyperloop Week (EHW). This competition brings together students, researchers, companies, and enthusiasts from all over the world to show their advances in the field of hyperloop and compete in different challenges related to this transportation system. The main objective of the EHW is to encourage collaboration and the exchange of ideas between the participants, as well as to promote development and research around the hyperloop.

The EHW can be considered a Research and Development (R&D) environment where scientific research, technological development, and innovation are fostered and promoted. However, there are also some differences.

One difference could be the orientation towards demo and visibility. The EHW stands out for being a public event where hyperloop prototype demonstrations take place. The intention is to show the advances and potential of this technology to the general public, investors, and companies.

However, what primarily differentiates EHW from a conventional R&D environment is its deadlines and rushed nature. The projects presented in EHW are complex, multidisciplinary projects developed in a one-year spam by teams formed by students. Meanwhile, the product and research industry usually has time, money, and human resources orders of magnitude greater than the average team.

This complex and innovative character of the EHW mixed with very limited resources present in the teams creates an environment highly harsh and prone to errors. Therefore it is necessary to implement a verification system to guarantee the accuracy and reliability of the results, validate hypotheses and models, evaluate the effectiveness of the proposed solutions, detect and correct errors or biases, and establish trust and credibility in the project findings.

1.2.3 Objectives

Hyperloop has the potential to transform the way we think about transportation. To change the way people and goods travel around the world through disruptive technologies and changes. Such changes imply huge costs not only in research but also in infrastructure, time, and human resources.

This makes the feasibility of the hyperloop concept to highly depend on the efficiency of the R&D industry. Which can be radically improved through verification.

The questions that this research will try to ask are:

- What types of verification methodologies can be applied in an R&D development environment?
- What adaptations are necessary compared to industrial environments?
- What benefits does it provide? How much time does it save and consume? How much money does it save and consume?
- Is it worth applying verification methodologies in an environment such as EHW?

Ultimately, the findings of this research will contribute to a better understanding of the role that verification takes in the development of a safe and reliable hyperloop system.

1.3 Design Competition Award

Given that this paper aims to research testing methodologies in an R&D environment, such as the hyperloop context, this research is presented to the **Full-Scale Award**.

Chapter 2

Research

2.1 Abstract

The title of the research is 'Building a Sense & Control testing environment for the development of a hyperloop vehicle'. Sense & Control is the term used within the EHW to describe the resulting system that comes from integrating hardware, software, and control components. This system allows the vehicles to perform their function correctly as it is characterized by a great complexity both on its individual parts and on the integration points between them.

Control engineering is a field based on control theory that uses sensors and actuators to know the status of a system and act on it accordingly. This is applied in several critical environments, from the autopilot of planes and ships to medical instrumentation. In addition to that, control must be integrated with software and electronics systems, which are becoming –as shown in trends like autonomous vehicles, IoT, or smart shopping– more complex and innovative. This implies a massive necessity for testing and validation to obtain reliability.

Testing all these systems simultaneously in real vehicles is not only financially and humanly risky but it also increases the time necessary to detect and solve problems. Suppose a system has three critical errors within its software, four in its hardware, and two in the control design. All errors arise in different situations that sometimes overlap, creating different combinations of symptoms, 12 in this case -$3 \cdot 4 \cdot 2$- possibly leading to infinite testing times and complex and confusing debugging of the errors. The solution is to test all three systems separately in different stages but inside the same testing framework.

This research aims to study and develop a Hardware, Software, and Control Verification System. Studying if it is applicable to a changing and unstable industry like R&D and if cost can be lowered by building custom and domain-focused industries.

2.2 Model-Based Design & Testing-in-the-loop

Model-Based Design (MBD) is a design methodology where mathematical models are created in order to simulate the behaviour of each component the system is composed of. It is used to create systems with complex control, communication, or signal processing. In this way, MBD allows rapid prototyping and verification. MDB is the current most common approach in industrial, aerospace, and automotive products. Also, this allows the introduction of multiple validation stages along the implementation phases, allowing for *testing-in-the-loop* (TIL).

TIL consists of different stages which allow the continuous test of a system during the implementation of its different layers.

The first stage is *model-in-the-loop* (MIL). The models are tested under different conditions and they are verified based on the knowledge of the behaviour of different systems. In this stage, it is possible to make changes to the design in an early stage as well as to test different scenarios. In some products, this stage is also used as a proof-of-concept.

The second stage is *software-in-the-loop* (SIL). At this point, the software of the system is integrated into a simulation environment, replacing the ideal mathematical algorithms with code blocks while simulating inputs and outputs to them. SIL is used to evaluate performance and functionality in realistic conditions. It tests the correctness of the software which will later be executed on the hardware.

The next stage is *processor-in-the-loop* (PIL). The aforementioned code now runs on the actual processor or microcontroller being used. Tests are run using the software on the real processor, where the inputs are simulated –provided by the model– and the outputs are observed in the real target. The compatibility and performance of the software on the target processor are evaluated during this stage.

Finally, *hardware-in-the-loop* (HIL) is the last stage of the present methodology. The software runs on the actual hardware and is integrated into the simulator environment. Hence, tests are run using the software and hardware together to simulate real conditions. The interaction between hardware and software in realistic conditions is evaluated, thus verifying not only the functionality but also the efficiency and safety of the whole.

2.3 Methodology

Since R&D is a fast-changing and complex environment, the preferred levels of abstraction are the discipline or component levels. This ensures correctness in terms of basic functionality in the early stages of prototyping and design. Regarding the identification of cases, the best result is obtained in dynamic testing. It offers better guarantees than static or peer review as it avoids any possible personal bias of the designers involved and allows the system to be tested in a similar environment to the working conditions.

Regarding the types of verification, it is not necessary to carry out exhaustive performance or security tests in view of the fact that both are ensured if the system fulfills its expected functionality. The performance of the control algorithm is accurate if the levitation works as in the simulations. In consequence, the preferred option is functional testing. Finally, regarding the methods for generating stimuli, choosing the best option depends on the objective of the verification in each case.

As a result, TIL is the best option as it offers functional tests that progressively include more disciplines. Nonetheless, as for the code validation, it is strongly based around automatically generating all or almost all the software of the system. Code generation is extensively used in the industry as it presents the advantage of reducing human errors during implementation. However, generated code involves the sacrifice of a great quantity of memory, performance, and flexibility.

Of the aforementioned disadvantages, the loss of flexibility is the most critical. Due to the lack of experience, available resources, and tight deadlines present in the R&D and proposed use case, a large number of last-minute requirement changes and modifications happen during the testing process. When developing new technology, requirements are unstable due to the fact that unforeseen results keep taking place just as research progresses. The implementation of the needed changes requires a deep knowledge of the software implementation and must be developed quickly.

An easy access to iterate fast and make on-the-go changes to the control algorithms implementation should be provided in order to achieve an improvement in the productivity of the testing. As a result, the advantages of manually implementing the code overrun the ones code generation possesses. It is far more valuable to have team members with a deep knowledge of the produced code –which is something impossible for generated code. This choice allows the team to avoid exhaustive performance testing by making an overkill implementation that ensures high performance from the beginning.

Also, this allows the automation of all the software-related tests. Practices like 'continuous integration' –automatically testing the software on every change– and testing driven development –writing first the test for the software and then writing the software– are simplified if all the code base is written manually. As shown in Section 2.3.5.

Briefly, the testing workflow of Hyperloop UPV consists of:

- ***Model-in-the-loop.*** A model of the main subsystems that interact with hardware and software is developed: levitation and traction. With MIL, the aim is to validate the design of the control loops as well as to determine how the vehicle will react in corner cases such as the

sticking to the infrastructure, over-temperature, saturation of electrical variables, and extremely noisy sensors. This stage shows little difference from the industry.

- ***Software-in-the-loop:*** A custom C++ control library has been created and tested using Simulink. This library has a C++ class equivalence for each Simulink block. Simulink is also used for the testing of the integration between the blocks. This stage has no changes with respect to the industry, since everything is tested using a functionality of Simulink called equivalence testing.
- ***Processor-in-the-loop.*** Instead of verifying the system against the model once again, the software is tested using the SHUTUP platform developed by the team through unitary tests. These unitary tests –opposite to the usual unitary tests performed in software– include integration with the hardware. Hence, the difference between the industry practices resides in the fact that all tests are focused on verifying the software logic and its integration with the microprocessor. The reason why this happens is that PIL is usually performed with automatic code generation and is also used to test the performance of the control algorithms.
- ***Hardware-in-the-loop.*** Using the SHUTUP platform as well as Simulink Real-Time has enabled the running of a complete simulation by setting the vehicle PCBs in real conditions. The aimof this is to validate the performance of the complete system, including the interaction between real hardware and software.

Summing up, the main differences with a standard testing-in-the-loop workflow are:

- **Adding software verification.** Discipline-level testing has been done on the software. It allows us to focus not only on the system –SIL verification– but also on more low-level software, such as sensor readings, finite state machine cases, or protections. Moreover, to do this step, SHUTUP in validation code mode has been used, so the software has been run in the target hardware –commercial PCBs with the same microcontroller. For this reason, a combination of a *new verification stage* based on unitary tests and PIL has been used.
- **Removing PIL simulation.** In PIL verification, the generated source is run on the target hardware while communicating with the model straight from the microprocessor. In Hyperloop UPV context, it has been challenging to find an affordable cross-compiler that is compatible with the target microcontroller. Also, the most important information provided by PIL testing –performance– can be extracted from the Software verification stage mentioned earlier.
- **Building a custom HIL.** The main difference between an industrial application and the system developed in Hyperloop UPV is that –as explained in Section 2.3.5.5– a custom input/output board and Middleware server have been built to reduce costs and delivery/sponsorship times. The function of the middleware is to adapt the communication between Simulink and the I/O module –called SHUTUP. This allows both the software and HIL verification to be performed using the same platforms –the only changes are applied in the middleware.

Figure 3.1 visually displays the abovementioned changes and the application to the chosen use case. The project can be divided into three disciplines: control, software, and hardware.

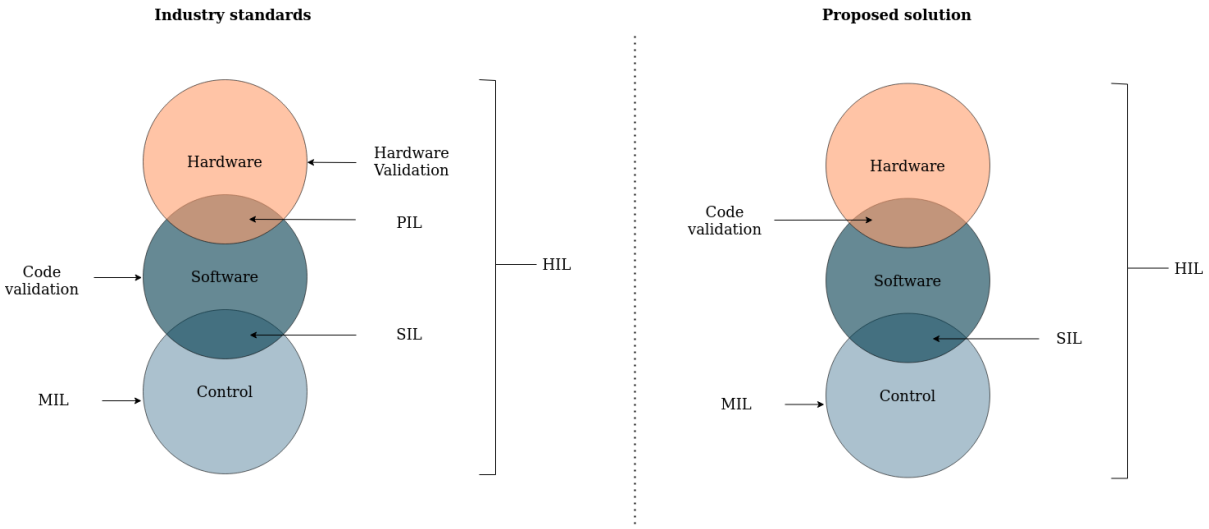


Figure 3.1: Proposed methodologies comparison.

To evaluate the proposed methodologies, two vehicle subsystems have been chosen as testers: the infrastructure monitoring and levitation subsystem. Therefore, the use cases for the tests are:

- **Infrastructure.** There is only one control board in the infrastructure, the Tube Control Unit (TCU). Which has no control algorithms. Thus, only code verification is necessary.
- **Levitation.** The levitation subsystem is made by two control boards, the Levitation Control Units (LCUs). LCUs run a cascade control algorithm that is distributed between both of them. Therefore, they need tests concerning MIL, SIL, HIL, and code verification to be verified. Also, the LCUs have to support two use cases –which affect the control and the tests: 1 degree of levitation ad 5 degrees of levitation. The model, control, setup, and tests differ for both use cases.

Figure 3.2 shows all the use cases stated above and the tests that concern each one. There are three groups of tests: 1 DOF, 5 DOF, and TCU. Levitation includes all tests relating to control –MIL, HIL, and SIL–, while infrastructure only has code verification.

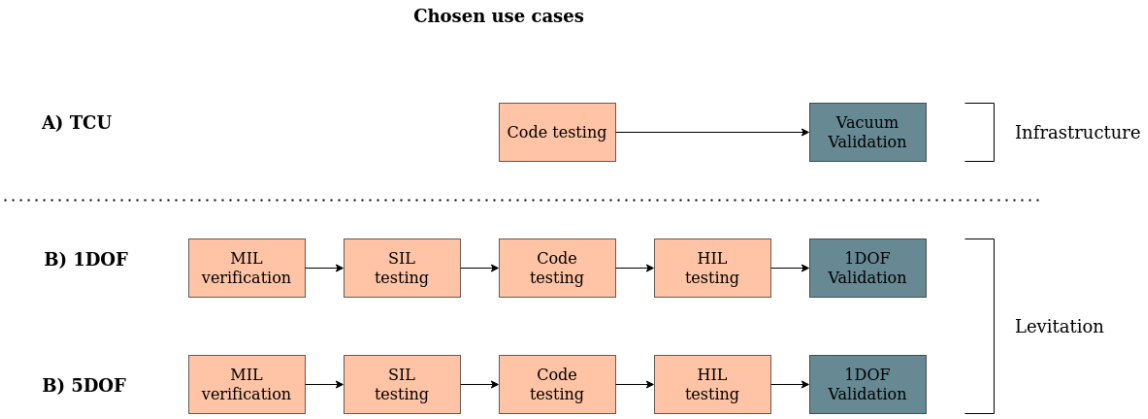


Figure 3.2: FSR use cases.

The approach used to evaluate the results of the applied methodology is a manual inspection and research of the control and code repositories of this and last year's development in the Hyperloop UPV team. As there is a repository for each control implementation, software library, and board code. The inspection is Based on the changes present in the commits, the documentation left, and dates. In this way, statistics concerning all errors produced in the development of different control boards can be produced.

2.3.1 Model-in-the-loop (MIL)

The development of a hyperloop system is a complex project that involves technical challenges such as levitation, propulsion, aerodynamics, and safety. In this context, MIL has been used to:

- **Design and test the control systems:** to ensure not only efficiency in operation but also safety. For example, testing the speed control before having the complete system in simulations will help in focusing on the algorithm and not on all the other systems involved, making it easier to debug the real problems in it. It decouples the different controls from the rest of the vehicle.
- **Develop and optimise the mathematical model:** it allows the simulation of different operating scenarios and conditions helping to identify design issues as well as to optimize system performance.
- **Testing in different environmental scenarios:** MIL can be used to validate the vehicle in different environmental conditions and operating situations such as high winds, changes in temperature, or mechanical misalignment increasing the reliability of the system.

In conclusion, MIL is used in an early design stage to simulate the behaviour of a complex system before the real implementation. It should not be confused with the digital twin concept.

Digital twins are used in the system operation stage to simulate system behaviour in real-time. It replicates the physical system by using the monitored parameters to analyze and predict its behaviour in different situations and operating conditions.

The relationship between both tools is that the model used in MIL can be used to create a digital twin of the system. The mathematical model is used as the basis for simulating the behavior of the system in the digital twin, which may include real-time data from sensors installed in the physical system.

The Model-in-the-loop testing in Hyperloop UPV starts by developing the model of the vehicle, Kénos. The approach followed is:

1. Divide the complete system into simple parts.
2. Classify the parts into different types: electromagnetic system, sensor, electronic device, or dynamic phenomenon.
3. Break down each type into the different physical phenomenon involved.
4. Specify the inputs, outputs, and functions of each part.
5. Develop the model of each physical phenomenon in the more modular way possible using as many parameters and masks as needed.
6. Check whether the specifications have been met.
7. Compose each type of system based on the previous blocks into sub-systems.
8. Customize each subsystem according to its characteristics.
9. Integrate each subsystem to form more general systems.
10. Put the systems together to create a complete complex system.

In each stage the designed part has been tested and reviewed by an expert in the topic, ensuring the correct implementation.

Figure 3.3 shows the **Kénos** model developed including each subsystem described and checked before.

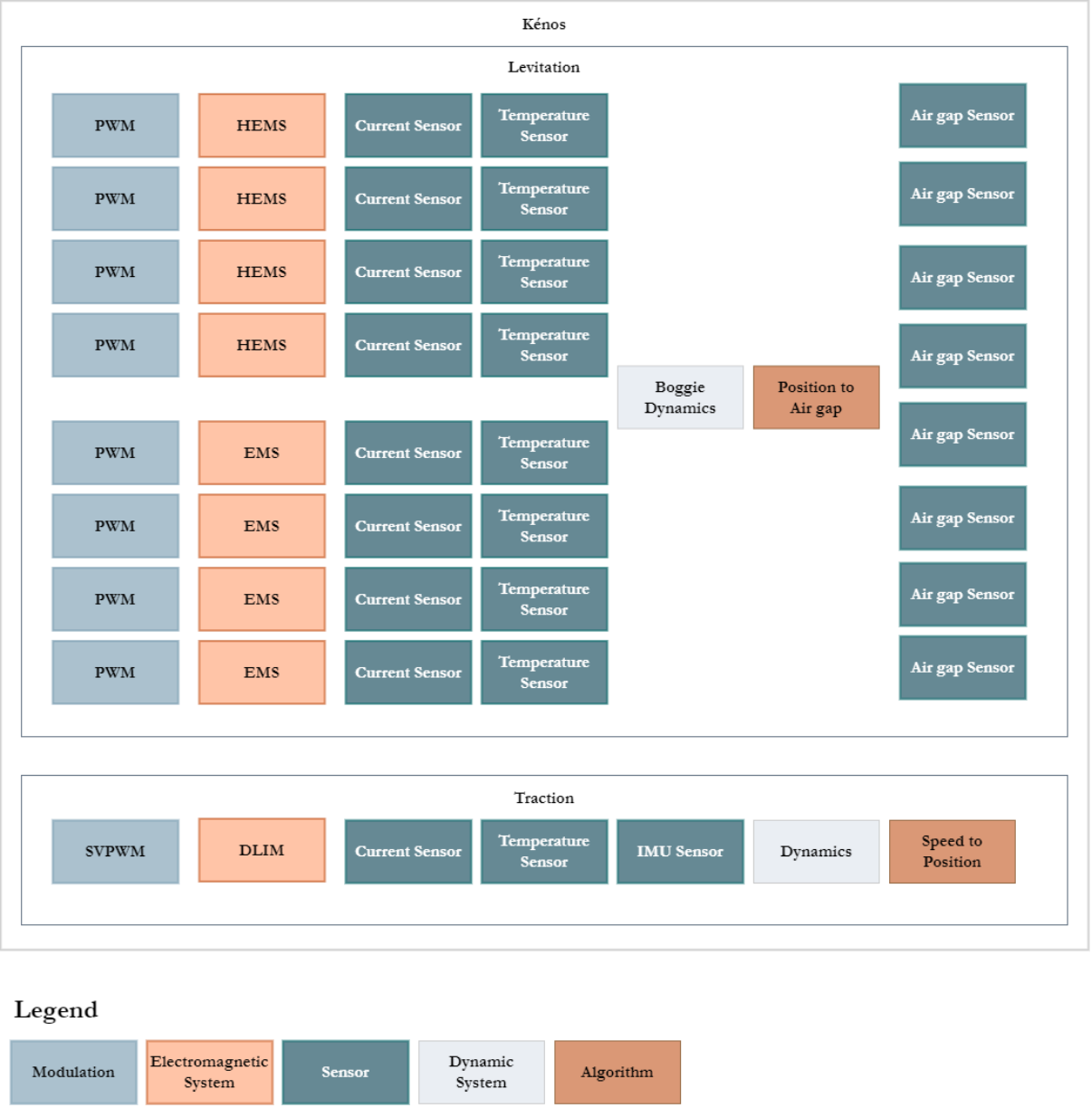


Figure 3.3: Complete description of **Kénos** model using subsystems.

When the model has been developed, the algorithm testing starts. In this stage, the main objectives are:

- Choose and adjust a control algorithm for speed control and levitation control.
- Analyse how imperfections in the real system against the model will affect, such as mass variation, center of gravity (COG) displacement, increasing sensors noise, mechanical misalignments, external perturbations…
- Ensure safety performance under different scenarios, such as temperature changes.

One example of using MIL in order to know how imperfections affect is shown In Figure 3.4. Where a sweep in the center of gravity (COG) position has been done in order of how much displacement is allowed without affecting the stability of the levitation control.

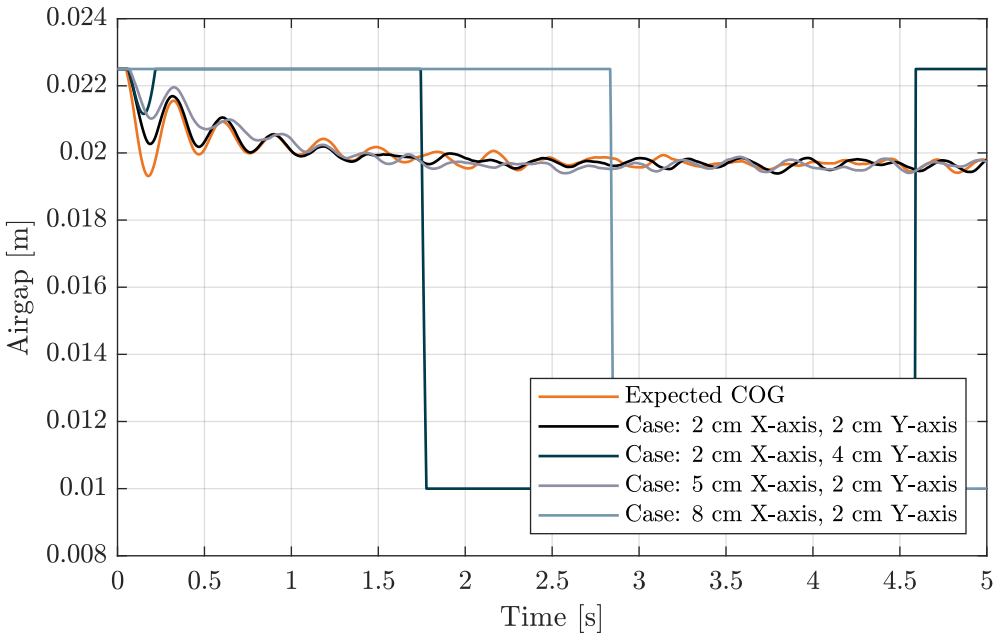


Figure 3.4: Using MIL to analyse how imperfections affect, sweep on COG displacement.

2.3.2 Software-in-the-loop (SIL)

In this stage, the goal is to check the correctness of the C++ code implementation for all controls individually. In order to do this, first all the blocks mentioned above are verified and then every control is verified separately. This is supported by the design of the control library.

As explained in Section 2.2, the tool used to model both the vehicle and the control algorithms is Simulink. To verify the different blocks, a Simulink equivalence test is performed. Simulink C++ code imports can be used to create a new model that includes both controls –one using Matlab blocks and other control that includes all the new blocks. Subtraction of both results is used to plot and verify the differences between both implementations.

Figure 3.5 shows the model used to compare both implementations of the control algorithm, while Figure 3.6 shows the difference between the Simulink output and the code-block.

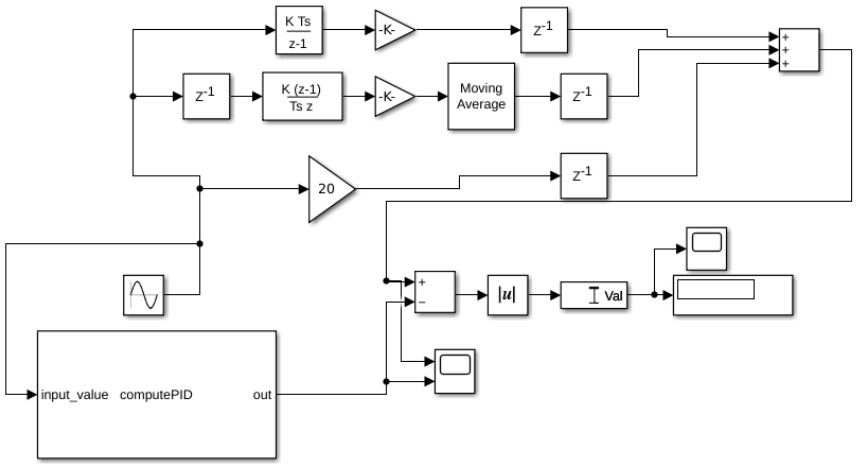


Figure 3.5: Model of the equivalence test mentioned previously.

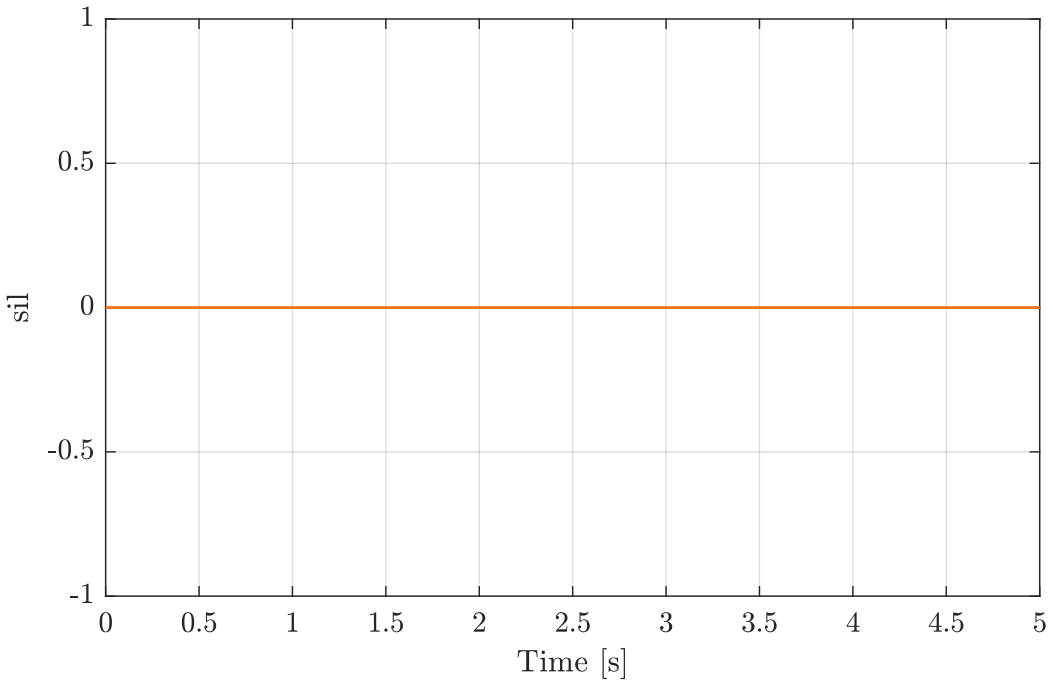


Figure 3.6: Results of an equivalence testing between the C++ and Matlab code in Simulink.

Therefore, the following tests have been carried out:

- Levitation
- Traction
- Control library

The use of the ST-LIB –explained in Section 2.3.5– allows the code to be validated in Simulink and then imported into the software without any modification.

2.3.3 Processor-in-the-loop (PIL)

Processor-in-the-loop (PIL) is the third stage in testing-in-the-loop workflow it cross-compiles source code and runs it on the target processor or an equivalent instruction set simulator. This technique is mainly used to:

- Perform execution profiling.
- Analyse how the rest of the code running on the hardware affects the operation of the code involved in said simulation, reducing the coverage reached with SIL testing.

Usually, PIL is performed by reading and writing the control inputs and outputs straight to the memory of the microcontroller. To do this, a cross-compiler for Simulink is needed. Most cross-compilers that perform this application are proprietary and expensive.

In the Hyperloop UPV context, finding a cheap and compatible cross-compiler with the target microcontroller is difficult. So, the SHUTUP platform in code validation has been used.

As explained before, SHUTUP has a software-validation mode in which the device under test is a Nucleo development board, therefore this is useful to validate the software. Also, to ease the development of tests, the tests are written in Python in a test server that communicates with the SHUTUP to perform the tests.

In this way, continuous integration can be included in the software development cycle. The test server polls for changes in the repositories, compiles, uploads, and tests every change in the code using SHUTUP.

For the research, two use cases have been implemented:

- **Verification of the Tube Control Unit (TCU)**: the TCU is the control board in charge of measuring ambient conditions inside the infrastructure and controlling the vacuum pumps according to the readings obtained. It also transmits all the data to a control station via ethernet.
- **Verification of the Levitation Control Unit (LCU)**: the LCU is a control board in charge of obtaining the data related to the position of the vehicle, calculating the control –previously validated using MIL and SIL–and controlling the vehicle position through the control signals that enter the Levitation Power Unit (LPU).

2.3.4 Hardware-in-the-loop (HIL)

The *hardware-in-the-loop* (HIL) technique is the final stage of the TIL and its goal is to run a simulation on a real-time computer with the plant model and a real version of the software running in the actual hardware. In Figure 3.7, a vision of the complete system is provided.

Figure 3.7: Overview of HIL system.

In the Hyperloop UPV context, a commercial system implies expenses and shipping times that are not suitable for the team. Therefore, a custom and simple solution has to be found. For this, as explained in Section 2.3.5.5, SHUTUP in HIL configuration has been used.

The real-time model has been obtained from the one used in previous simulations but with some adaptations. Those have been realised using the Simulink real-time application, being no time consumption or knowledge requirement.

This system has been applied to two use cases:

- **One Degree Of Freedom levitation (1 DOF)**: 1 DOF levitation is a validation step inside the development of the vehicle of Hyperloop UPV. Its goal is to validate the hardware and software systems involved in the levitation of the vehicle with a simpler use case. Therefore, in this test, a simplified version of the levitation in which the following elements are involved is done: a levitation test bench, a control board –a device that has to be also validated using HIL– and a power board.
- **Static Five Degrees of Freedom levitation (5 DOF)**: 5 DOF levitation is achieved in a 5 DOF test bench in which the vehicle is introduced completely and is tested in conditions really similar to the final infrastructure. The objective is to validate all the hardware and software involved in the levitation of the vehicle without having the final infrastructure ready.

The goal is to use this HIL system to speed up the tuning of the constants and validation of both tests, therefore, the HIL will be used with a 1 DOF and 5 DOF model. Meanwhile –for comparison– the control of the DLIM will be developed without any verification tool. It must be noted that the complexity of the 5 DOF and 1 DOF controls are way more significant than the propulsion control due to the fact that the levitation control is distributed between two boards, and also the amount of inputs and outputs is way higher on the levitation.

For the 1 DOF control, the following setup has been used:

- x1 SHUTUP at x0.1 as speed factor
- x1 Levitation Control unit (LCU) plugged into SHUTUP

- The Python middleware running in a workstation communicating Smulink with SHUTUP
- Simulink Real-Time desktop running in the same workstation and communicating with the middleware

For the 5 DOF control, the following setup has been used:

- x2 SHUTUP at x0.1 as speed factor
- x2 Levitation Control Unit (LCU) plugged into SHUTUP
- The Python middleware running in a workstation communicating Smulink with SHUTUP
- Simulink real-time desktop running in the same workstation and communicating with the middleware

2.3.5 Tools Employed

This section describes the different tools used and develop inside the Hyperloop UPV development ecosystem in order to follow the methodologies proposed below.

2.3.5.1 ST-LIB & Software

Software is one of the main aspects on which the reliability and robustness of a project depend. This is due to the fact that code is difficult to test and simulate. Also, the software is the last discipline to be fully integrated and tested in an engineering project and therefore is the one most affected by delays and deadlines.

Therefore, in the presented case of study –Hyperloop UPV– there is a strong focus on embedded code which has led to the development of a C++ library called ST-Library (ST-LIB). ST-LIB abstracts the behaviour common for all the boards in the vehicle, including the main blocks present in different control algorithms and basic behaviour like timers, communication protocols, sensors...

Figure 3.8 depicts the main blocks and layers of the library. As can be seen, its main goal is to abstract the HAL with a simpler and highly coupled Hyperloop UPV use case implementation.

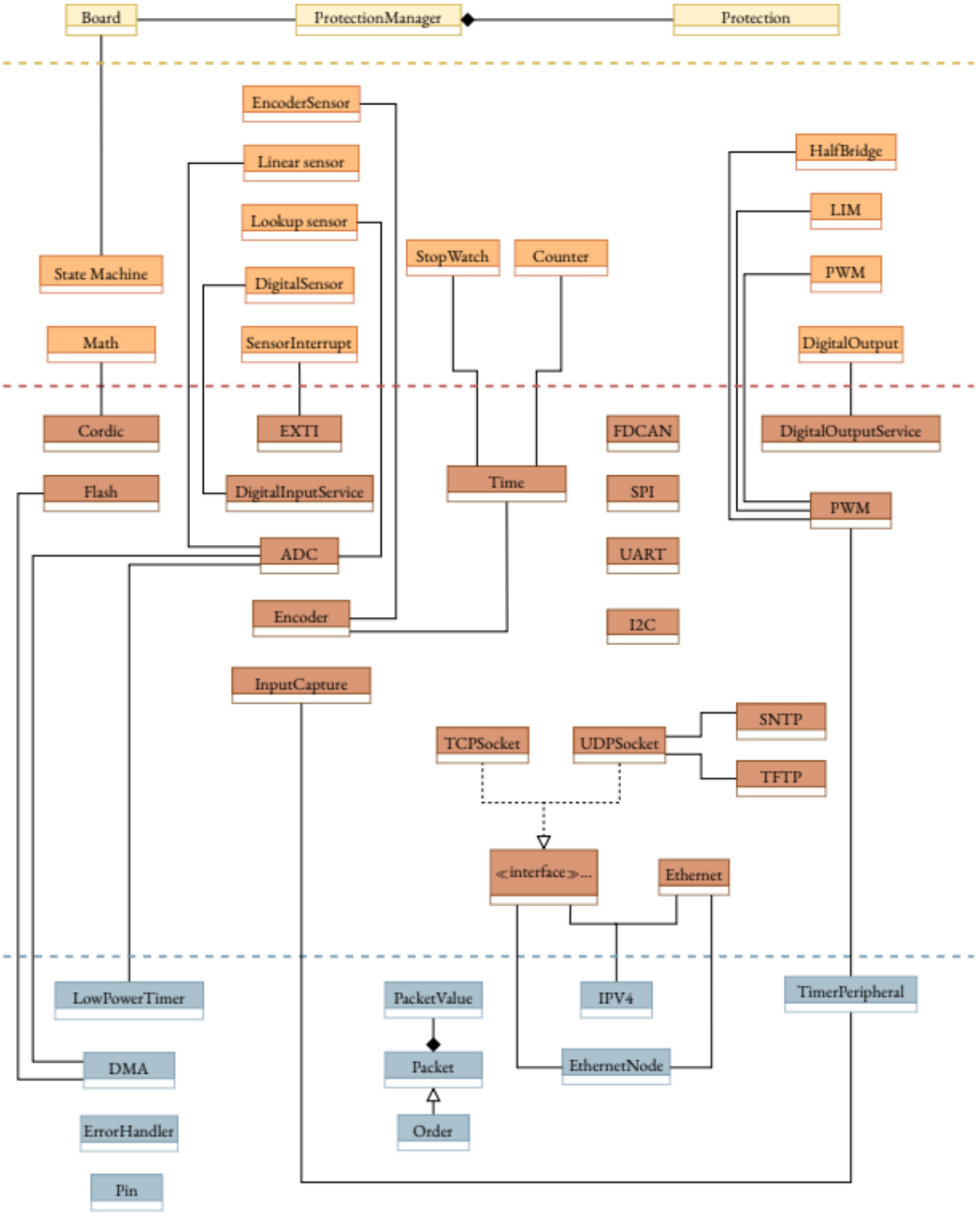


Figure 3.8: ST-LIB architecture.

Thus, validation of the library through the methodologies proposed would create a verified codebase that can be used to later write the code of the boards. Reducing the errors detected in later stages of development.

The code of the library is published on this ST-LIB repository and is publicly accessible and access to the repository is encouraged.

2.3.5.2 Simulink

Simulink is a MATLAB-based graphical programming environment for modeling, simulating, and analyzing multidomain dynamical systems. Its primary interface is a graphical block diagramming tool and a customisable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink is widely used in automatic control and digital signal processing for multidomain simulation and model-based design.

The methodologies proposed are used to model both the vehicle and the control algorithms. Also, Simulink has a C++ plug-in that allows the introduction of custom C++ code into blocks. This can be used to compare the C++ implementation of the blocks present in the ST-LIB to the Matlab original blocks of the control.

The Simulink model projects are published on the following Github repositories: traction repository and levitation repository. Both are public and access to the repository is encourage.

2.3.5.3 SHUTUP

Usually, in the industry testing platforms are built using proprietary or commercial components such as Speedgoat or CompactRio target machines –simulation and test processing units– and I/O modules –modules that translate the orders of the target machine to readings and writings of physical signals. However –in this and most use cases– this kind of product exceeds the budget and deadlines of the team, therefore a workstation running different custom servers will be used as a target machine

The Software and Hardware Unit Test Unified Platform –baptised as SHUTUP– is an I/O module that can be operated through TCP or CAN. SHUTUP can be used to read or write analogue and digital values from 0 to 5 V. It has support for the following signals:

To support software SHUTUP has to support :

- ×23 0 V to 3.3 V analogue outputs.
- ×3 0 V to 3.3 V analogue inputs.
- 8 0 V to 3.3 V input captures.

And to support the HIL mode SHUTUP has to support :

- ×18 0 V to 3.3 V analogue outputs
- ×6 0 V to 5 V analogue outputs
- ×4 4 mA to 20 A analogue outputs

- ×1 0 mA to 20 mA analogue input
- ×1 0 mA to 20 mA analogue input
- ×1 0 mA to 20 mA analogue input
- ×1 0 V to 3.3 V analogue input
- ×8 0 V to 5 V PWM input

Figure 3.9 shows an image of the designed board. As can be seen, it has 3 boards. two of them are testing unit devices (TUDs) which are in charge of having the logic that coordinates the writing and reading of the signals that SHUTUP handles. Also, there is another board –the one on top– that acts as a DUT for the software verification tests. As can be seen, the three control boards are commercial development boards from ST. This is to simplify the soldering and manufacturing of the board as well as to shorten the development time of the hardware.

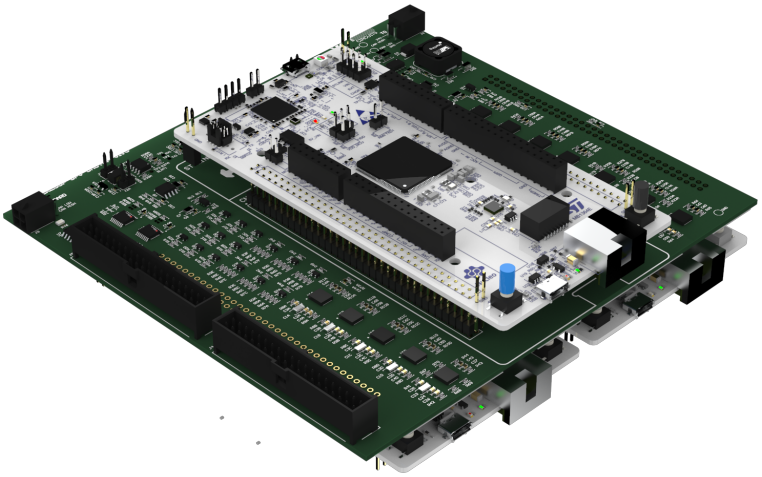


Figure 3.9: SHUTUP board.

Code for the testing unit devices is available in the following repository SHUTUP. The repository is public accessible and access to it is encouraged.

2.3.5.4 Test Server

A test server has been designed in order to implement continuous integration and automatic testing into the development of the software of the boards. Usually, to produce continuous integration the software is decoupled from the microcontroller. However, verification will always yield better results if the testing environment is as similar to real situations as possible. Therefore, continuous integration of embedded software makes no sense if the integration between hardware and software is not evaluated during the verification.

Consequently, in order to support correct software automatic testing, a test server with support for reading and writing digital and analogue signals must be designed.

Figure 3.10 depicts the architecture of the test server. As seen, it also uses SHUTUP as I/O module to handle the signals corresponding to the software tests. The test server has the same codebase as the middleware present in the HIL –as explained in Section 2.3.5.5. And a virtual private server (VPS) in the cloud is used to monitor changes in the GitHub repositories. Both the middleware, and cloud server are written in Python, and use the basic Github APIs are used to monitor changes in the code. When any change is pushed into any of the monitored repositories, an event is triggered in the online server which communicates with the on-site server. Then the on-site server uses the middleware to upload the code to the SHUTUP software DUT, execute the Python tests, and analyse the results. If any test fails, the push to the repository is canceled; otherwise, the merge is approved.

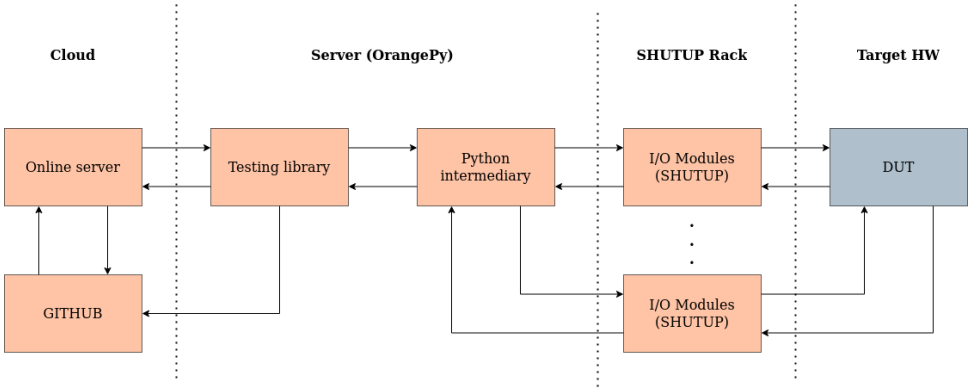


Figure 3.10: Test server.

The test server also contains a manual mode in which tests and binaries can be uploaded to the testing server locally. The test server executes the whole suite and returns the results through a simple http website hosted locally.

Figure 3.11 and 3.12 depicts the results of the software tests ran on the test Server.

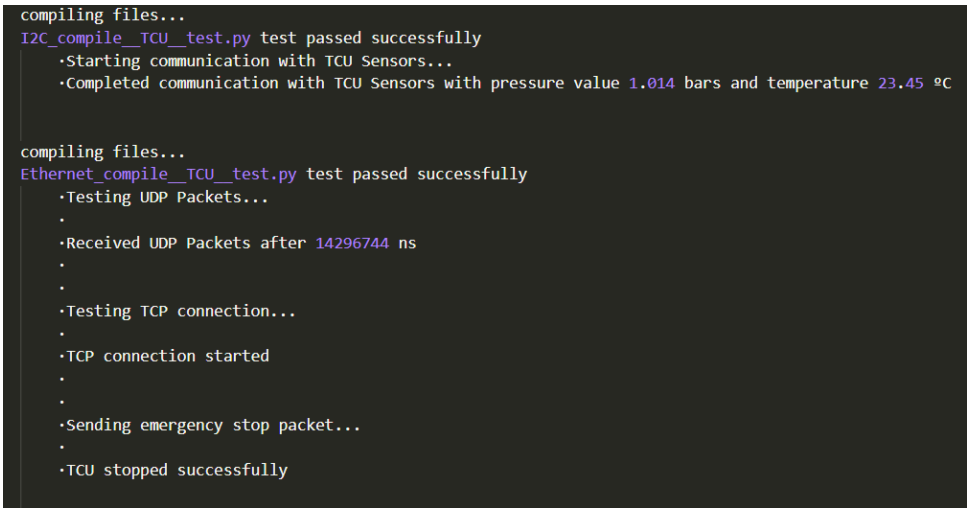


Figure 3.11: First part of results of code-validation on TCU.

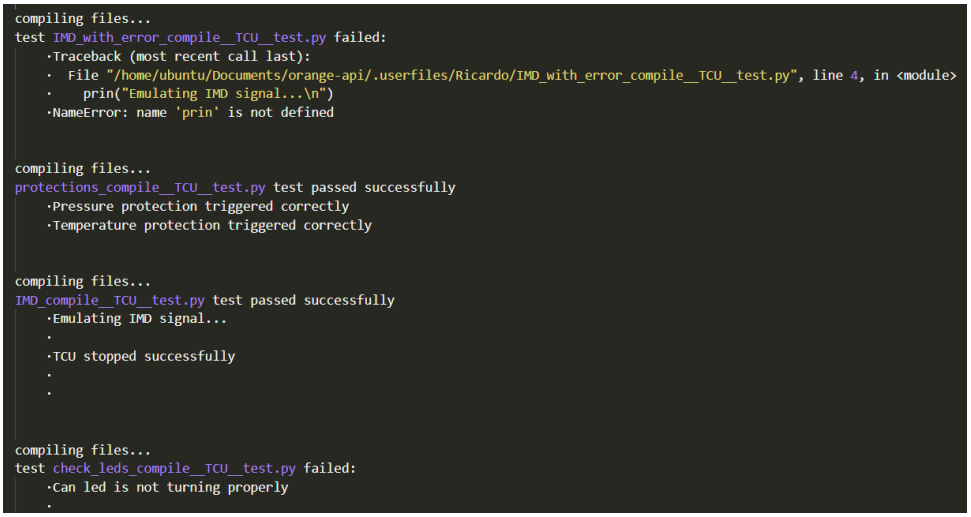


Figure 3.12: Second part of results of code-validation on TCU.

Figure 3.13 depicts the front-end page to upload manually tests and binaries to the test server.

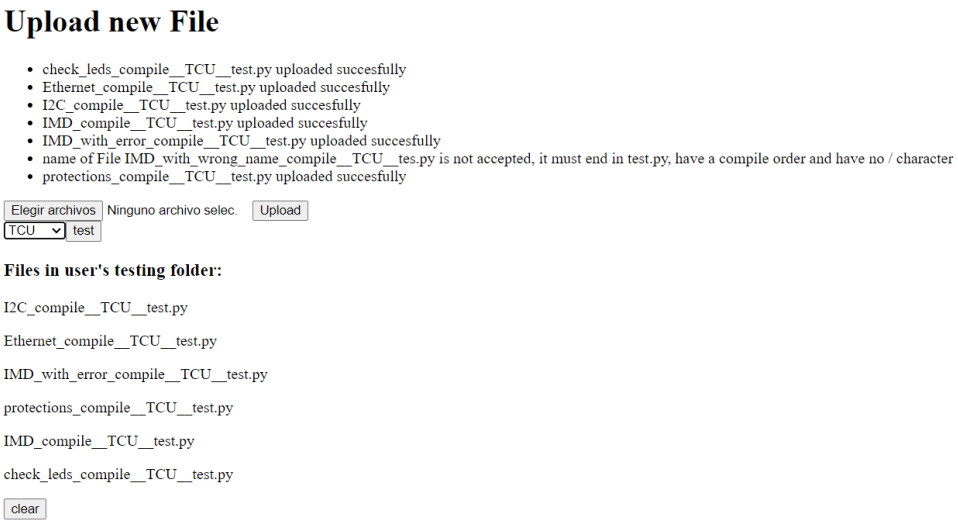


Figure 3.13: Test server front-end for uploading pages.

Code for the testing unit devices and the middleware is available in the following repository SHUTUP which is public and access to the repository is encouraged.

2.3.5.5 Custom HIL

Two possibilities have been exploded on how to run the real-time simulation. The requirement for the hardware is to have a CAN connection to communicate with SHUTUP, and to be able to run a complete 2 seconds simulation at 7.5 kHz. The execution frequency should be at least the reading frequency of the fastest sensor, which is the air gap one.

- On the one hand, it is the Speedgoat real-time target machine model 5929. When the model is deployed for the selected hardware and loaded is the time to select the running frequency. Because of memory limitations, only a 2 kHz frequency is reachable. This option is not usable for our application.
- On the other hand, there is an extension of Simulink, named Simulink Real-Time Desktop, which provides a real-time kernel for running Simulink models on a Windows or macOS desktop or laptop computer. Using this add-on it is possible to run the model with no frequency limitations. This is the chosen option.

The real-time simulation communicates with SHUTUP by TCP protocol. The sensor values are generated by the simulation and the SHUTUP generates the equivalent analogue value so the target HW/SW can read it.

When the analogue voltage value is read by the target HW/SW unit, it processes the measurement and generates the PWM signal that would be applied to the power unit and afterwards to the real electromagnetic system. Instead, the SHUTUP reads the signal and extracts the duty cycle applied.

This value is sent via TCP again to the model, which computes the voltage applied and actuates. The complete cycle is executed at 7,5 kHz periodic frequency.

Some of the inconveniences that have occurred are the following:

- A PCAN can be used to communicate a Linux/Windows system with the SHUTUP. However, Simulink does not have support for PCAN. Therefore, a Python Middleware has been developed to translate a TCP Simulink connection to CAN.
- To keep the timestamp invariable it has been necessary to synchronize the clocks between the SHUTUP and the real-time simulation. It has been done with a Simulink block named 'Real-time sync'.
- It is necessary to coordinate the state of the boards with Simulink –which has a rather complicated logic– therefore a library for controlling the boards from the middleware has been developed –emulating connections from other necessary boards, sending start levitation orders, etc.
- The proposed solution can have performance issues on high bandwidth peaks of messages. Also, debugging the control implementation is easier for the boards if the HIL is executed slower than reality –since performance issues on the board disappear. However, to do this, the control loop and data acquisition loops have to be slowed down by the same factor as the Simulink simulation. An API has been developed in the middleware to slow down the execution of the boards.
- STM32H7ZG microcontrollers have very few PWM inputs –reads the duty cycles by hardware– therefore the SHUTUP was designed using Input captures however, the precision is not enough for real-time simulations. Therefore two SHUTUPs have been soldered and included in the can-bus to perform the 5 DOF tests. For this, the API has been adapted to configure by hardware the ID of each SHUTUP, making a scalable system just by changing jumpers.
- STM32H7ZG microcontrollers do not have enough digital to analogue signals to perform the 5 DOF simulations. Therefore DACs have been implemented in SHUTUP using low-pass filters that filter any frequency higher than 10 kHz –higher than the control action frequency– and turns it into an analogue signal.

Figure 3.14 displays the overall architecture of the custom HIL. As stated before, Simulink communicates by TCP in a local host with a middleware that translates the communication to CAN to communicate with the bus of I/O modules. Then, the I/O modules are connected with the target hardware, which is the Device Under Test (DUT).

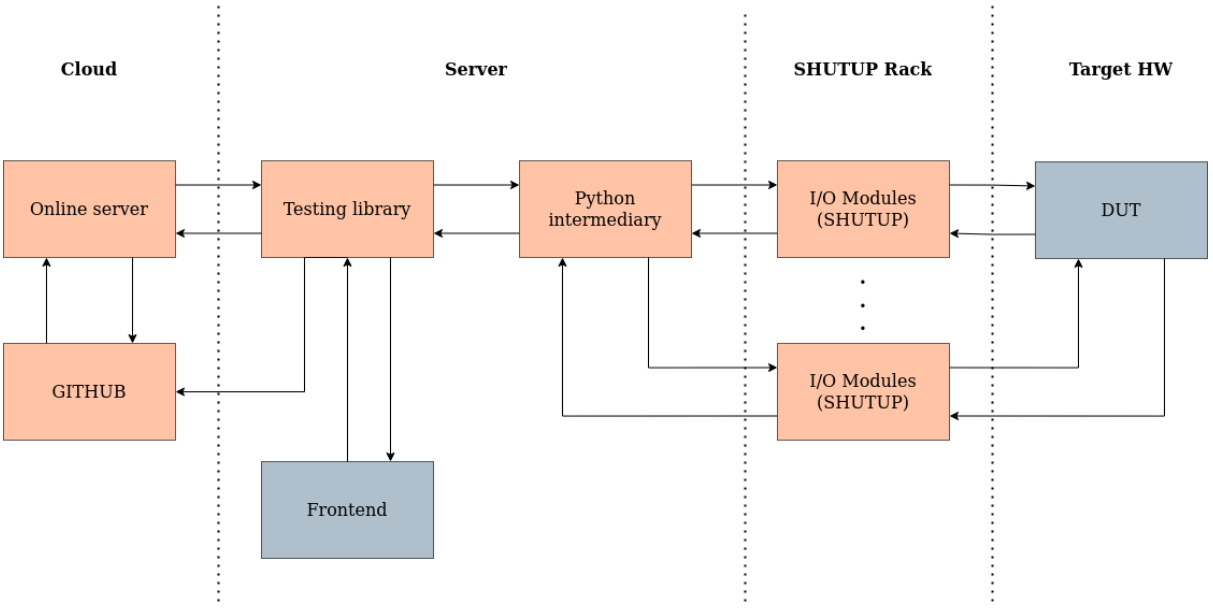


Figure 3.14: Custom HIL.

Also, as shown in Figure 3.14 the HIL has a front-end and back-end which processes the data from the test servers and sends it to a front-end. The purpose of the front end is to allow easy access to launching simulations and visualizing the results through 3D graphics. The back end is written in Goland and the front end is in typescript, running React.

Figure 3.15 shows a screenshot of the front end while simulating 5 DOF.

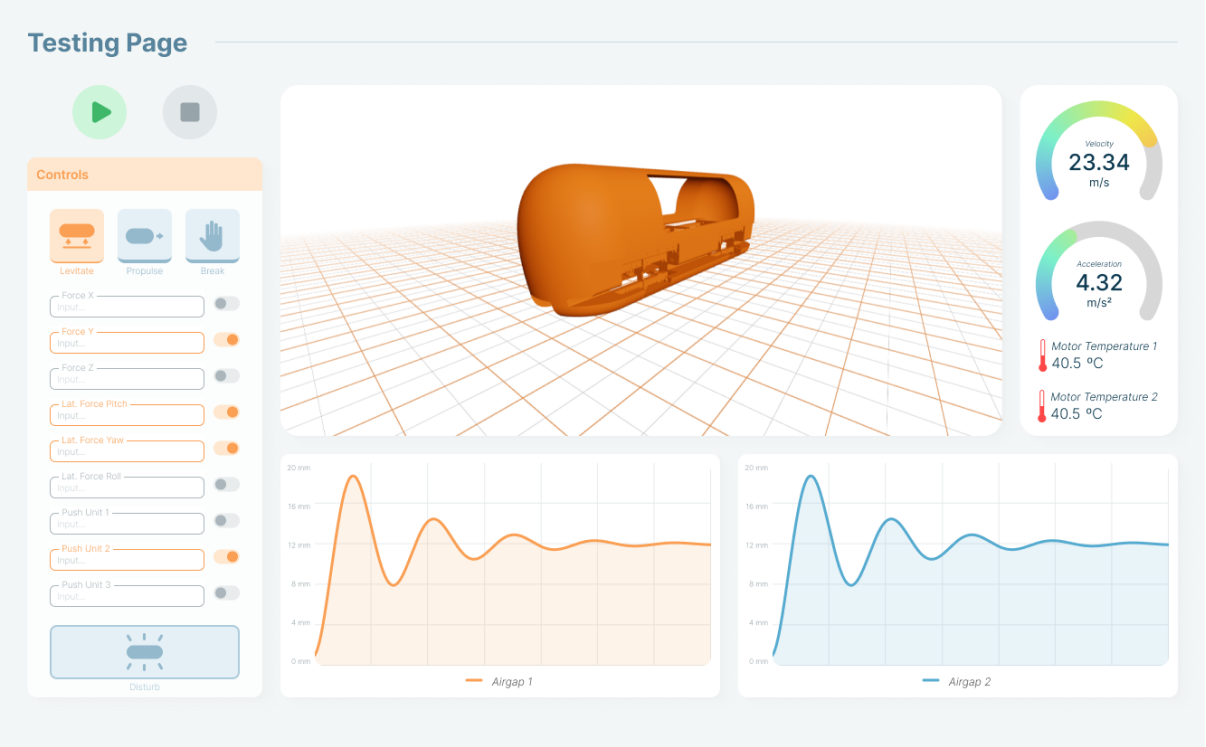


Figure 3.15: HIL front-end while simulating

Code for the system is in the following repositories: SHUTUP, HIL GUI and levitation model. Which is public and access to the repository is encouraged.

Chapter 3

Results & Discussion

3.1 Results

The results achieved on 1 DOF levitation control have been stunning. By adjusting the control in the HIL previously to the test bench, there have been much more less iterations. One hour of adjustment has been needed on the HIL, meaning around 15 trials. In the test bench –in real conditions– the constants given by the HIL worked on the first trial, while experiences from past years predicted weeks of working on tuning the constants in the test benches, demonstrating how the HIL increases flexibility and productivity.

Figure 1.1 shows a comparison between the Z-position evolution with the same constants in the HIL and the test bench. The results show that in reality, the setting time is higher as well as fewer oscillations are present. One possibility of the appearance of these discrepancies may be friction with the test bench guides, which are considered ideal in the model.

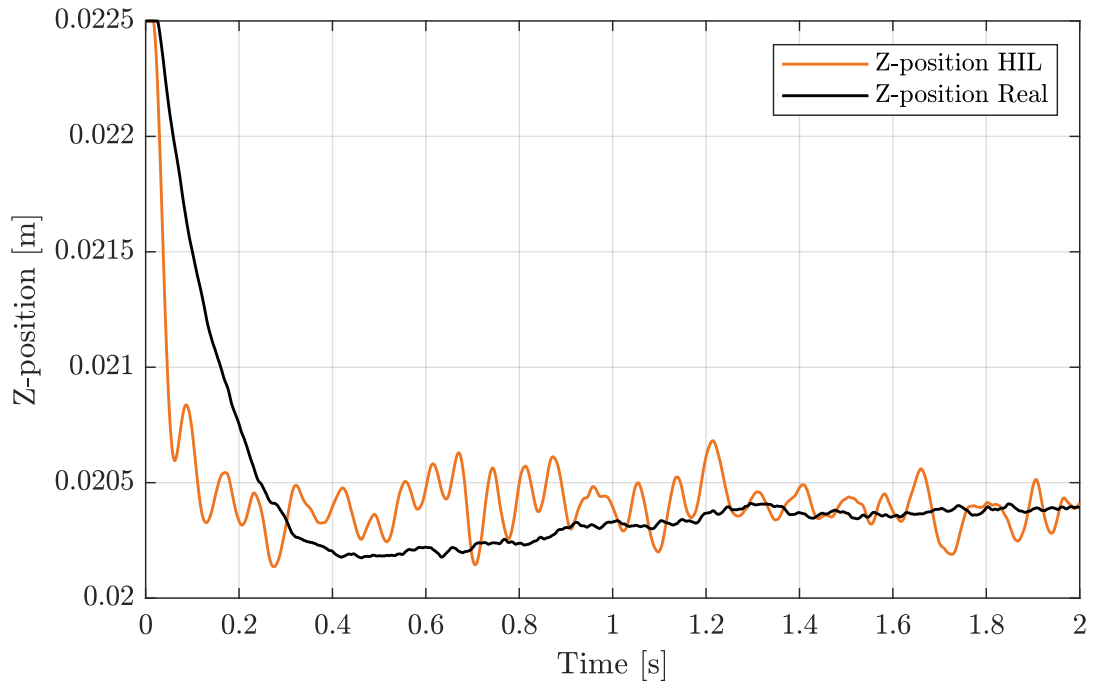


Figure 1.1: Comparison of the Z-axis position evolution in reality and the HIL.

Figure 1.2 shows the results achieved in the test bench with the first designed constants, the ones obtained by MIL. The difference in the results obtained thanks to the application of the methods is remarkable.

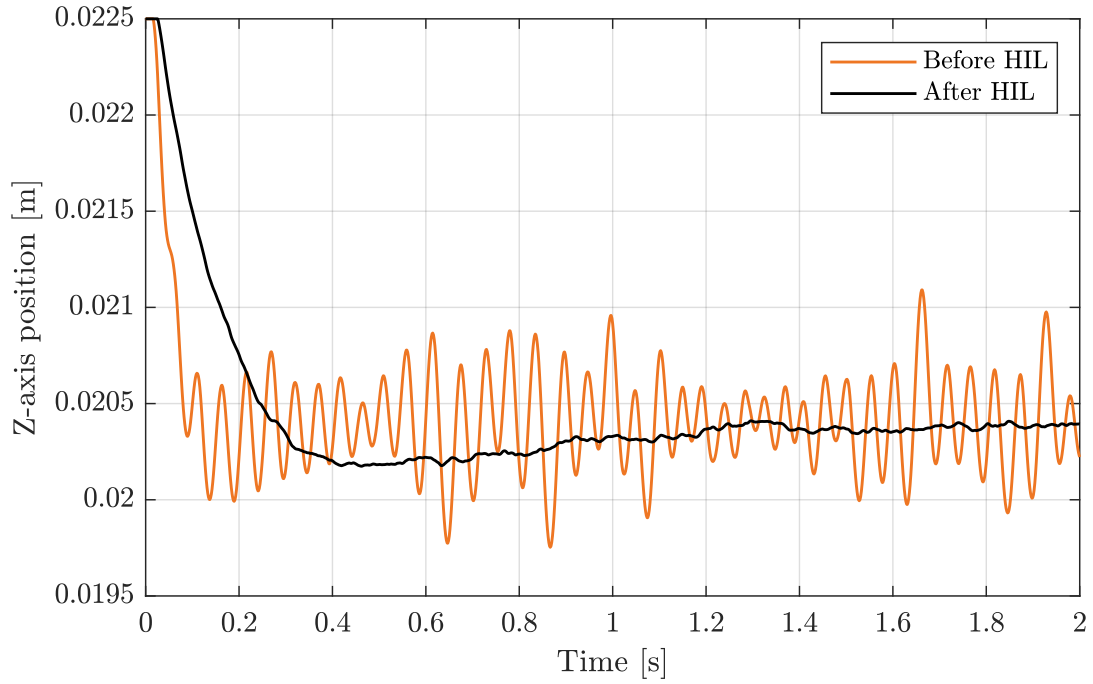


Figure 1.2: Comparison of 1 DOF levitation control in the test bench with before and after HIL constants.

The results of the 5 DOF control display similar stunning results as 1 DOF. A control iteration round took around 20 minutes using the vehicle in the 5 DOF test benches. Meanwhile, the HIL allows performing x10 5 seconds iterations of the 5 DOF control in under 5 minutes. Real conditions could be simulated in the HIL using real values as the sensor noise or the real center of mass of the vehicle in the HIL.

In Figure 1.3 it is shown a comparison between the Z-position axis with the constants adjusted in the MIL when using the HIL platform.

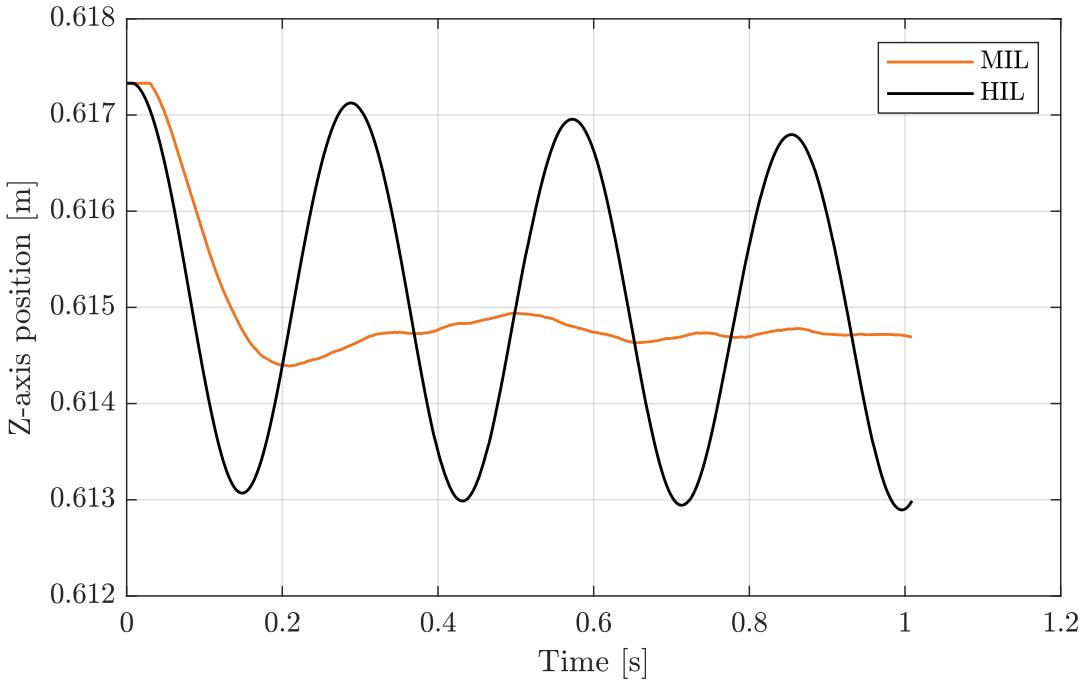


Figure 1.3: Comparison between MIL and HIL results of 5 DOF levitation control.

3.2 Error Detection Analysis

The results achieved by applying the verification plan described in previous sections, compared with those achieved in former years, when in the absence of a verification plan and having only validation, will be discussed in terms of error detection analysis.

The data assessed has been extracted by analysing the dates and comments of each commit of the Hyperloop UPV H8 repository and the Hyperloop UPV H7 repository.

The detected errors have been divided into the following types:

- **Implementation**: failures in code programming. They include syntax errors, incorrect logic, improper exception handling, and uninitialised variables.
- **Design**: errors in software architecture, poorly designed interfaces, inappropriate choice of algorithms, among others.
- **Interoperability**: the software fails to interact correctly with other system components, such as operating systems, libraries, or third-party applications. They can arise due to communication protocol incompatibility, data format, or configuration requirements.
- **Integration**: failures that occur when different modules or components of a system fail to combine correctly.
- **Performance**: failures in terms of slow response, excessive resource consumption, bottlenecks, or prolonged loading times.
- **Usability**: failures that affect the easy use of the software, such as confusing interfaces, lack of user feedback or complex workflows.
- **Maintenance**: errors that arise during software maintenance and updates. They may include introducing new errors when applying patches or updates and a lack of proper version control.

Figure 2.1 compares the number of errors of each type that appear in the validation stage, meaning that the verification strategy has been unable to find them.

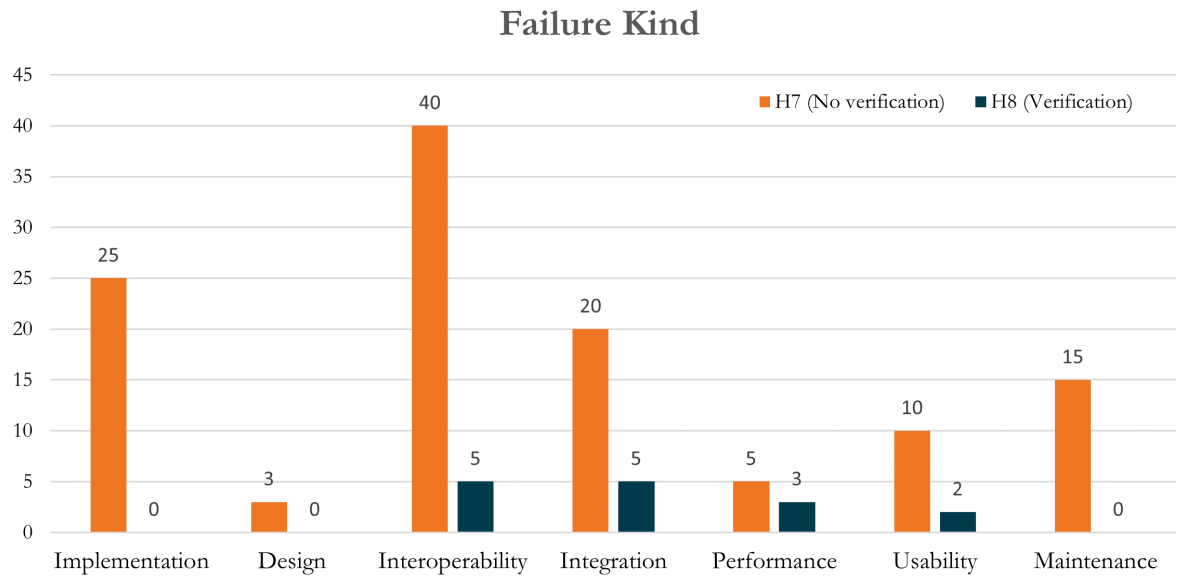


Figure 2.1: Comparison between the different varieties of errors that appeared in validation stages between the H7 generation of the team –without verification plan –and H8– with verification plan.

Therefore, including a verification strategy is the key to reducing errors and delays during development. Due to the nature of the project, most errors generally arise during integration between subsystem-subsystem implementation.

Table 2.1 shows the number of errors detected in each verification stage, meaning the verification strategy has been able to find them.

Failure Kind / Verification Stage Detection	MIL	SIL	SW Validation	HIL
Implementation	10	25	0	0
Design	20	2	0	0
Interoperability	0	0	0	5
Integration	0	0	0	5
Performance	0	0	2	2
Usability	0	5	2	0
Maintenance	0	0	6	0
Total	30	32	7	15

Table 2.1: Analysis of the errors detected in each verification stage.

As observed, having different verification stages has been helpful in detecting almost every error in the software part.

- Implementation and design errors disappear entirely in the first two stages.
- Usability errors are detected mainly in SIL and SW Validation stages because of the process of integrating the code under test in the platform.
- Performance errors are detected in the SW Validation and HIL stages since the code is running on the actual hardware.
- The maintenance errors are always detected in the SW Validation process since a new commit triggers the execution of the automated tests.
- The integration and interoperability detected are mainly connectivity errors between components inside the subsystem tested. The subsystem-level verification is the HIL one.

3.3 Timing Resources Analysis

When analysing timing resources, three main branches must be considered.

- Verification tools design, including time invested in developing SHUTUP hardware, SHUTUP programming, and unitary tests server creation.
- Time of using the verification tools.
- Validation time, meaning the final stage where **Kénos** is physically tested in **Atlas**.

Figure 3.1 shows the time invested in developing the verification tools. When the project is framed within a context such as the European Hyperloop Week, where the entire project must be ready within one year, allocating around two months and a half for development is not negligible.

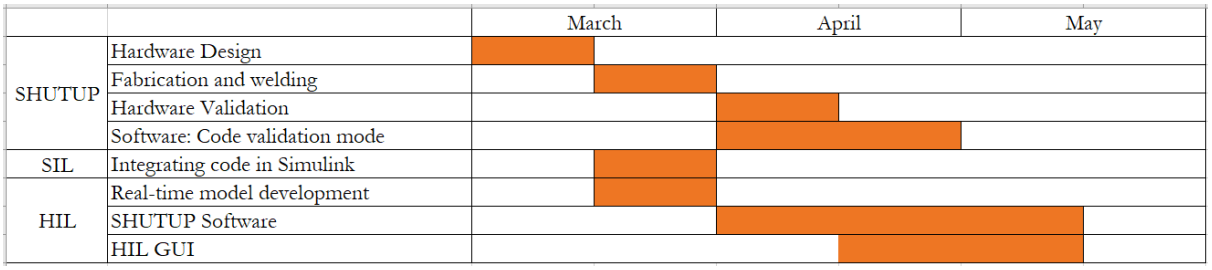


Figure 3.1: Timeline of verification tools design.

Figure 3.2 displays the timeline corresponding to the using the verification platforms. As shown, the period destined for development is noticeably more extended than the one for using the platforms, meaning that reusing these tools in future generations of the team will entail a significant advantage.

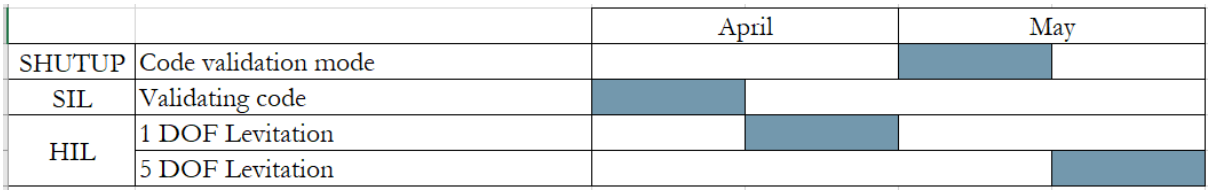


Figure 3.2: Timeline of verification tools usage.

Validation timelines of H7 and H8 generations are compared in Figure 3.3. It is observable that the validation time has been reduced from one month to one day in a 1 DOF levitation case, and it has been reduced to one-fourth in a 5 DOF levitation case.

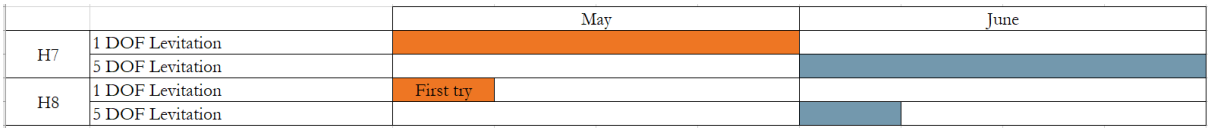


Figure 3.3: Timeline of the validation of the vehicle in test benches.

3.4 Economic Resources Analysis

The economic resources analysis compares the costs of developing our own verification platforms versus buying a complete HIL system that fills the requirements.

The quote has been asked for Speedgoat with the following requirements:

- Can bus 1.0 at 1 Mbps
- Ethernet port at 100 Mbps
- Analogue input x42 between 0 and 3.3 V with a resolution of 16 bits and a max reading frequency of 10 kHz.
- Digital input x9 between 0 and 3.3 v with a max reading frequency of 10 kHz.
- Digital output signals x12 between 0 and 3.3 V most of the PWMs with a maximum frequency of 40 kHz.

The resulting quote has the following modules:

- Performance real-time target machine
- Performance-P3-CPU-3.6GHz-8Core
- Performance-P3-RAM-32GB
- IO602-Performance
- IO110-Performance
- IO306-Performance
- IO306-Configuration Package

The final cost of this quote, without taking into account the ramp-up and installation of the system, is 26,139 €.

While analysing the cost of Hyperloop UPV verification platforms, the breakdown costs are: materials, software, and manpower. The values are shown in Tables 4.1, 4.2 and 4.3.

Material	Cost (€)
SHUTUP fabrication	300,00
SHUTUP components	150,00
SHUTUP nucleo boards	200,00

Table 4.1: Material cost breakdown.

Software	Cost (€)
Matlab	860,00
Simulink	1300,00
Simulink Real-Time	860,00
Python	0,00
Altium	1600,00
LTSpice	0,00

Table 4.2: Software cost breakdown.

Task	Hours	Cost/h	Cost (€)
SHUTUP hardware design	40	15	600
SHUTUP assembly	5	15	75
SHUTUP hardware validation	15	15	225
Software: code validating mode	30	15	450
Integrating code in Simulink	1	15	15
Real-Time model development	5	15	75
HIL GUI	40	15	600
Software: HIL SHUTUP	30	15	450

Table 4.3: Manpower cost breakdown.

The total costs of these verification platforms would be 7760 €. However, since all the software is sponsored and we are students that do not have a salary, the real cost is 650 €.

As shown, the cost of a commercial platform that fills the requirements of a complex project, such as R&D, may be unreachable in a reduced-budget environment. However, a custom verification platform costs 70 % cheaper without considering sponsorships.

3.5 Conclusions

It is commonly believed that verification systems are not necessary for RD precisely because of the complexity and changing nature of projects. Notwithstanding, this is exactly the reason why Hyperloop UPV believes it is essential to have a complete and robust verification system. And this system must enable the rapid identification of simple failures in order to let the truly innovative part of the project be the main priority.

It is no secret that the magnitude of the investment in resources entails the main hindrance to the implementation of verification methodologies. Whether the budget is generous enough as to make a commercial system affordable, the development time of a custom one can be devoted to different matters. Regardless, in limited-budget R&D environments –such as the one Hyperloop UPV belongs to– this is not a feasible option.

The team has been able to turn this into an opportunity. In the end, technological advances always impulse disruptive changes in the industry, and as modernising, automating, and simulating systems get more accessible every time, failures in the R&D industry start to concentrate on implementation and verification instead of concept errors.

The methodologies proposed in this paper are a 6-month student-made adaptation of the methodologies implemented by big software enterprises but made by students having resources that fall nowhere near theirs.

"They are doing things with $100 and 13B params that we struggle with at $10M and 540B. And they are doing so in weeks, not months." - Google

Chapter 4

Bibliography

Hyperloop UPV Repositories

Github Levitation-H8 repository

Github Firmware-tests repository

Github Action-tests repository

Github HIL GUI repository

Github SHUTUP repository

Github Traction-H8 repository

Consulted Resources

Pezzè, M., & Young, M. (2008). *Software Testing and Analysis: Process, Principles, and Techniques.* John Wiley & Sons.

Utting, M., & Legeard, B. (2012). *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann.

Binder, R. (2013). *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional.

Fraser, G., & Zeller, A. (2011). *Evolutionary testing and the road to systematic test case generation.* Communications of the ACM, 54(12), 66-75.

Arcaini, P., Gargantini, A., & Vavassori, P. (2014). *Combining model-based testing and runtime verification in the B method.* Software & Systems Modeling, 13(3), 907-930.

Utting, M., Pretschner, A., & Legeard, B. (2012). *A taxonomy of model-based testing approaches.* Software Testing, Verification and Reliability, 22(5), 297-312.

Online Tutorials

Speedgoat. *What is Hardware-in-the-Loop Simulation?* [En línea]. Disponible en: link.

Speedgoat. *What is Rapid Control Prototyping?* [En línea]. Disponible en: link.

Speedgoat. *Unit real-time target machine* [En línea]. Disponible en: link.

MathWorks. *Simulink Real-Time* [En línea]. Disponible en: link.

MathWorks. *Convertir el modelo en tiempo real: Create and Run Real-Time Application from Simulink Model* [En línea]. Disponible en: link.

MathWorks. *Ejecutar aplicación en tiempo real: Execute Real-Time Application* [En línea]. Disponible en: link.

MathWorks. *Target to Development Computer Communication by Using TCP* [En línea]. Disponible en: link.

MathWorks. *Target to Host Transmission by Using UDP* [En línea]. Disponible en: link.

HYPERLOOP **UPV**

Shaping the future.