INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Continental

State of the Art

Software Testing for Automotive

February 24, 2021

Students

AMARI	Youssef
ARRONDELLE	Hugo
BAH	Thierno Amadou
BENSIALI	Adrien
CALMETTES	Pierre
CHAINE	Clément
FRAGONAS	Fabrice

Tutors

AURIOL	Guillaume
NICOMETTE	Vincent

Abstract

This paper discusses processes and methods for testing software. Firstly, different development processes are presented, the V-Model and the Agile methodology. It is then followed by an introduction to two testing approaches: White-Box and Black-Box testing. They include some examples of techniques related to theses concepts. Finally, the last part is a focus on embedded system testing with three major methods: Model-in-the-loop, Software-in-the-loop and Hardware-in-the-loop.

Keywords: Testing, Embedded, Model-in-the-loop, Software-in-the-loop, Hardware-in-the-loop

Contents

Introduction

Automotive industry has always been one of the most important witnesses of technological progress. Nowadays, the biggest challenge about the automotive industry seems to be the current work on autonomous cars and intelligent embedded devices. Such great projects need many systems to work together and exchange data. It is therefore a priority to realize in-depth tests in order to predict every possible behavior. In this report we will not only discuss a product designed for autonomous cars, but a more global tool, aimed to ease the testing process for the entire automotive industry.

Vitesco Technologies has recently developed the Jammer Box, a subsystem set up between the car engine and the calculator. This electronic device contains a micro-controller that receives and analyses two incoming signals from the Cam and Crank position sensors. The system can then inject some failures into the signals before sending them through the output. The aim is to generate specific anticipated failures so the calculator can deal with them. This process occurs without time interruption, which enables the car to remain a Real Time System despite the adding of new testing features.

The Jammer Box is software-driven by a computer using a USB cable. As a result, it allows the user to send requests and failures to inject from the User Interface. Two students' teams have been created at INSA Toulouse to help Vitesco develop this box. Some of us will work on the new sensors supposed to be used in the next generation of cars, and we are now going to study the global software testing process we could set up and build into the Jammer Box.

As the validity of the system cannot be verified during the coding phase, software testing is very important in the world of developers. Indeed, they have to deal with very complex software systems which can contain a very large number of lines of code. Bugs can appear and have varying consequences generally leading to failures.

Fortunately, tests can prevent faults. However, exhaustive testing is impossible even if we have unlimited resources. According to Dijkstra, program testing can be used to show the presence of bugs, but never to show their absence [2]. Thus, we must maximize the search for issues while minimizing the use of resources such as time or money. We need to know which components are more susceptible to contain bugs than others.

Then, we have to combine different testing strategies if we do not want to fall into the pesticide paradox: "Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual" [2]. A system context has to be taken into account to avoid these pitfalls. Therefore, the testing strategies for a mobile app will differ from those of a rocket guidance system.

Finally, a system's quality consists of code validation and code verification, that can be proven by answering respectively: Are we making the right software and Are we making the software in a good way.[2]

1 Testing processes

Development processes have massively evolved over the years, but tests are still an essential part of them. There are many kinds of tests suited for different levels of abstraction which happen at specific moments in an ongoing project.

1.1 V-Model

The V-Model is a *traditional* development process where a project is sliced into two phases : **Project Definition** and **Project Testing**.

The first step, going down the definition arrow (see figure 1), consists in specifying sub-systems incrementally to finally reach a level of abstraction where the sub-system cannot be divided anymore : that's the component level. Components will be implemented before going up the testing arrow. Each sub-system is tested within its level of abstraction[13].

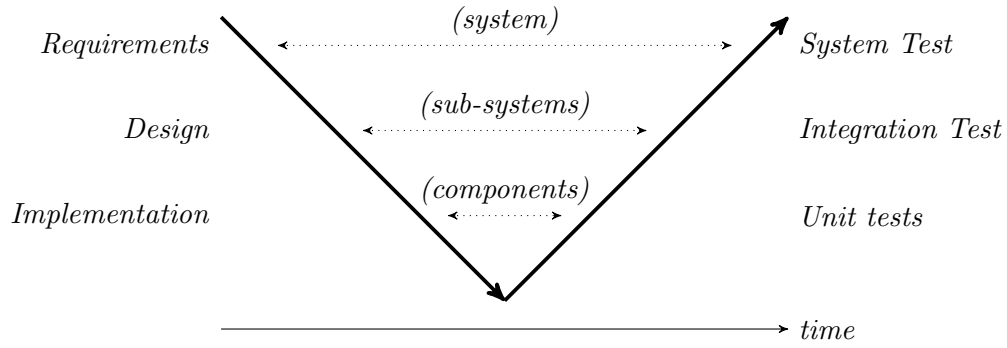

Figure 1: Traditional V-Model

There are three types of tests in the V-Model which correspond to the component, sub-system and system levels.

1.1.1 Unit tests

Tests for the components level are called *unit tests*, as they test small self-contained portions of code. This step is required to be able to confirm the validity of the implementation, by testing the code exhaustively[10].

The complexity of unit testing comes down to maximizing the exhaustiveness of the tests. Techniques such as *coverage testing* can be used to increase their robustness.

Unit tests are preferred compared to higher level tests for components as it doesn't generally involve too much code and makes the systematic analysis easier[8].

1.1.2 Integration tests

Once each component has been thoroughly tested separately, *integration tests* make sure that they work well together to be part of a larger sub-system.

Even though these types of test provide a way of testing the interaction between components to diagnose any potential problem, it doesn't help in identifying and understanding the issue[6].

Integration tests also tend to be more difficult to write than Unit tests, that's the reason why there is always less of them.

1.1.3 System tests

Making sure that each sub-system can perform together as intended to form the entire *system* can be achieved through *system tests*. They check the behaviour of the system to verify that it meets the previously written specification.

Although these tests are not very common due to their complexity, they can be automated using *Unified Modeling Language* requirement diagrams such as *use case* or *sequence* diagrams[9].

1.2 Agile software development

Ever since the Agile Manifesto and its underlying twelve principles were created, the model gained popularity in the industry and among the software engineering community.

As stated by its authors in the Manifesto of the Agile Alliance, listed below are the Agile values and principles:

We are uncovering better ways of developing software by doing it, and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Martin Fowler explained "The Essence of Agile Software Development" by saying "Agile Development is adaptive rather than predictive; people-oriented rather than process-oriented.". The Agile methodologies transform the software development process. It becomes more adaptable to change and therefore less costly and more time-efficient.

In Agile development, the testing process is directly integrated into the development phase to correct bugs upstream. It helps to identify problems as early as possible and speeds up product deployment[20].

Continuous integration is the practice of continuously integrating changes to a project and testing them at least once a day or more. Typically, each member of a team integrates his or her work at least once a day. Thus, every day, many integrations are made. The intended outcome is to identify any problems in the code and automatically notify the developer. This helps ensure that the code base is not broken any longer than necessary[20].

Continuous testing is the very important phase of the end-to-end application life cycle management process. It involves functional testing, performance testing, security testing, and so on. Selenium, Appium, Apache JMeter, and many other tools can be utilized for the same.

Above is an example of a possible specification for Agile testing [11]

1. **Define and execute "just-enough" acceptance tests:** it allows developers to clearly define a functionality and its expected behavior.

2. **Automate as close to 100% of the acceptance tests as possible:** Running all tests manually can slow down the development process when the application contains thousands of features to be tested. Automating tests saves time and facilitates the integration of new tests.

3. **Automate acceptance tests using a "subcutaneous" test approach with a xUnit (or an other framework) test framework:** this process makes it possible to ingest the tests in a versioned repository.

4. **Run all acceptance tests in the regression test suite with the build, daily (at a minimum).**

5. **Develop unit tests for all new code during a time period.**

6. **Run all unit tests with every build.**

7. **Run multiple builds per day**

2 The Box testing approach

The box testing approach is a way of classifying tests. There are two types of them: **White box** and **Black box** testing. The first one consists of tests which take into consideration the inner workings of a program. On the other hand, Black box testing is achieved by testing a program from its user's point of view. This candid approach can help identify a broader spectrum of issues.

2.1 White-box

2.1.1 Static and Dynamic code analysis

Static and Dynamic code analysis are two ways of automatically analysing computer programs. With static program analysis, the code is read but not executed and return information about correctness (partial and total), optimisation and statistics. There are two types of analysis : Data flow analysis which gathers values in the program and Symbolic analysis which us symbolic data tables. Static analysis can be achieved with linters, it is a static code analysis tool which analyses stylistic and semantic errors in the code. There is a command in Unix, `lint` (see Unix manual), which allows a user to statically analyze code in C. It detects various errors such as variable which were declared but not used.

Dynamic analysis allows the developer to get information about execution time, memory and power consumption by executing the code. Many tools are available like Address, Leak or Memory sanitizers from the `llvm` project. It can also detect data races and deadlocks with the ThreadSanitizers.[7]

2.1.2 Code coverage

In order to know if the amount of tests executed on a program is sufficient and every feature has been verified, we can proceed to do some code coverage measurements. They allow us to calculate the percentage of code executed in our program. It is based on analysing the variables, functions, and function returns to check if they are correct. Developers aim at getting the best code coverage in order to be sure most of the defects have been removed. So it takes time to create unit and integration tests to increase the percentage. However, a 100 percent code coverage does not mean a bug free program. It is important to have decent code coverage to avoid forgetting to test a major functionality. However, relying only on this is insufficient.[3]

2.1.3 Regression testing

As software evolves over time, new versions are released regularly. This involves adding new features and making modifications (improving the User interface for example) to satisfy consumers. Regression tests are used to check that the old functionalities are not affected by the new ones, that the modifications have not impacted the well functioning of the software and that there are no new bugs.[14]

2.2 Black-box

Black-box testing is a software test method that allows you to detects bugs in a program without knowing the internal structure of the elements tested. There are several types within the Black-box approach, as some bugs cannot be found using just one of them.

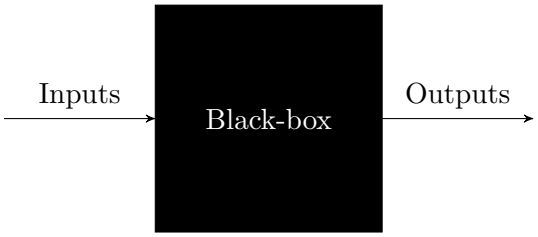

Figure 2: Black-box testing

2.2.1 Fuzz testing

The term “fuzz“ was first introduced in the 90s to test the reliability of computer programs. It only received worldwide attention a few years later. Fuzzing or fuzz testing is a software testing technique to reduce defects in software systems or given targets. These numerous faults can generate vulnerabilities exploited by malicious users to attack these systems by executing their own code. ”Fuzz testing is a process that sends malformed or semi-valid data to a program deliberately to detect vulnerabilities or errors in the target” [23].

Considering inputs are random, they are likely to reveal inappropriate and unexpected behaviour in the targeted program. The piece of software tested can crash during fuzzing if it does not correctly reject incorrect entries.

2.2.2 Decision Table Testing

The most rigorous functional testing methods are based on decision tables. The decision table test is built on its solid foundation. Since the early 1960s, decision tables have been used to analyze and represent complex logical

relationships. "They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions." [12]

A decision table is made up of four distinct parts:

- The stub portion
- The entry portion
- The condition portion
- The action portion

In decision table 1, when conditions `c1` and `c2` are True, and condition `c3` is False, actions `a1` and `a3` occur. For Rule 6, `c1` is wrong, `c2` is wrong and `c3`'s value does not have any impact. A decision table is actually a truth table if the conditions are binary, this guarantees us to take all possible combinations into account.

Stub	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	-	T	F	-
a1	x	x		x		
a2	x				x	
a3		x		x		
a4			x			x

Table 1: Decision table example

2.2.3 State Transition Testing

State transition testing is a Black-box testing technique. This technique is an improvement of the Decision table, where the output depends on the input conditions and the previous output. This allows developers to analyse more complex behavior such as finite-state machines.

"The two most common coverage criteria used in state transition testing are all states coverage and all transitions coverage. All states coverage criterion requires reaching all states, whereas all transitions coverage criterion requires exercising all transitions."[18]

There are 4 main components of the State Transition Model:

- State

- Transition

- Events

- Actions

We will take the example of a SIM card. If the PIN code is correct in any of the three attempts, the user will be accepted. If it fails after the third time, the user will be blocked. Also, if the user enters the wrong password the first and two times, we will ask them again.

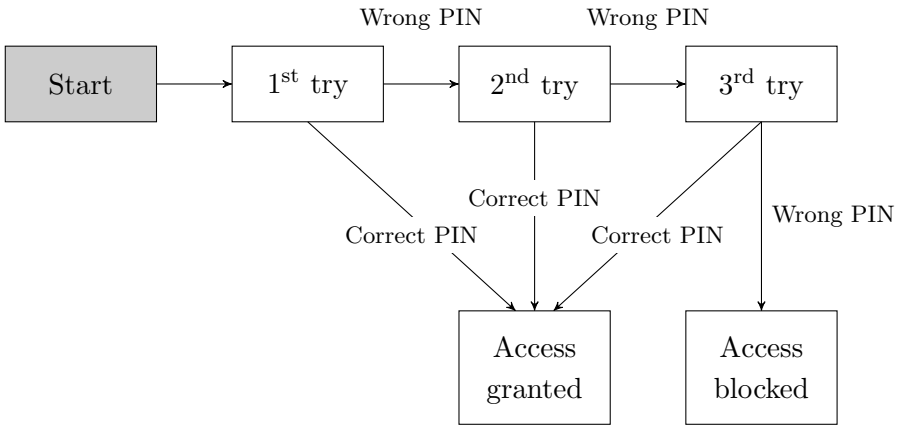

Figure 3: State Transition testing [18]

3 Embedded specific testing

Simulation techniques are used to test embedded systems. Model-in-the-loop, Software-in-the-loop and Hardware-in-the-loop simulations are the three testing techniques usually set up for verifying the correct functioning of an embedded system.

3.1 Model In the Loop

Model-in-the-loop is the first step of the validation process. During this test phase, the entire system is verified using modeling. It allows developers to use virtual simulation on a system via a real-time computer simulation. Thus, thanks to mathematical models, physical testing is not the only solution to testing a mechanical system. To design these models, tools such as MATLAB and Simulink are used.

This technique has many advantages, such as reducing the cost and the complexity of the physical test, and also improving team cohesion and communication, while facilitating the explanations and the agreement with the customer on the various features[1]. Moreover, component modeling is practical, a system can be tested even if some physical parts are not finished.

However, to work, the parts simulated with the computer need sensors to communicate with the rest of the system. These will not be present in the real system and can introduce errors. According to A. R. Plummer, "The difficulties associated with limited actuator response have been a major concern"[16], but various solutions have been proposed to counter these delays issues.

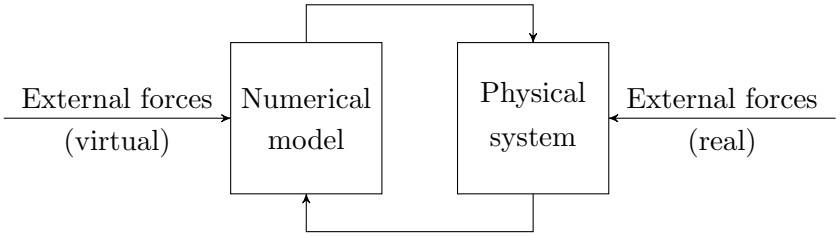

Figure 4: Model-in-the-loop concept[16]

The MIL method is therefore very useful to simulate a system at the start of a project. During the design phase, engineers can use it to properly define and understand the system. However, it can lack precision and it can not be completely in agreement with the final real system, because it is necessary to use tools to achieve it which can add inaccuracies.

3.2 Software In the Loop

SIL (Software-In-The-Loop) is a process for performing tests in order to develop and validate an algorithm while being based on a hardware representation of a given system [17]. More precisely, the tests of SIL are employed among manufacturers upstream of the development process [5].We can use the V-model to represent where SIL tests are supposed to be done.

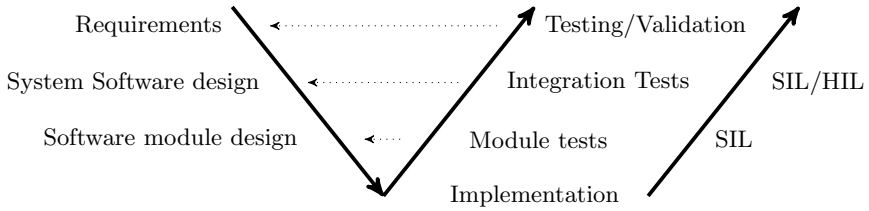

Figure 5: V-model in an embedded system.

We observe that SIL tests are used in Module tests of Software module design and also in integration tests of System Software design [15].

Software-in-the-loop provides several advantages: It includes the inexpensive costs and adaptability of a simulator and also the faithfulness of a hardware emulator. Indeed, the cost saving is explained by a high reuse of code and also a low exploitation of development resources [19].

It is the absence of hardware and also that tests are easy to parameter that justify the adaptability of SIL. The fact that we can test the system before the injection in hardware is a good advantage too. This induces a decrease in costs and time dedicated to software testing.

However, the notable drawback is the runtime performance that is not the same as in a real system [5].

Despite the fact that software-in-the-loop is an attractive solution for testing, it was not common until recently. Thanks to the appearance of APIs, this testing solution has become widely used [19].

3.3 Hardware In the Loop

Modern systems grow in complexity, particularly in software where thorough and reliable tests are necessary to verify and validate designs. HIL (Hardware-in-the-loop) is a technique where real signals from a controller are connected to a test system that emulates reality. The solution must provide comprehensive testing as though the real-world system is being used. With the HIL process, you can easily run through thousands of possible scenarios to properly exercise your controller without the cost and time associated with actual physical tests.[21]

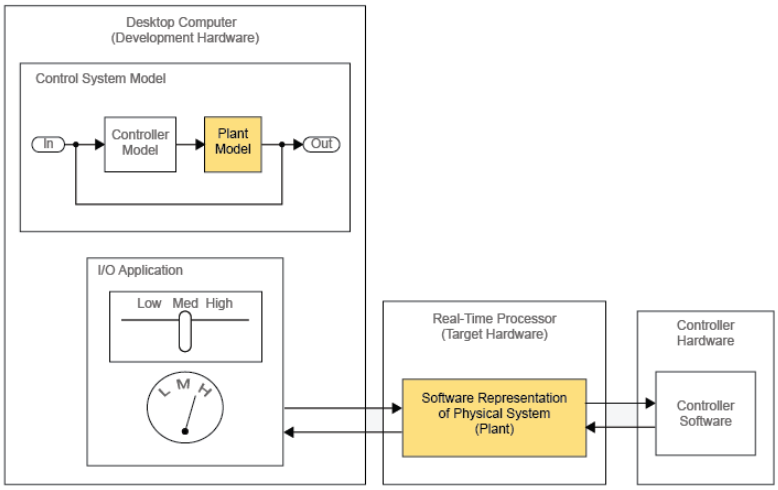

Figure 6: Hardware-in-the-Loop Simulation setup

The system model is composed of the Development Hardware, the Target Hardware and the Controller Hardware. The Development Hardware contains the model of the physical plant associated with an interface to control the virtual inputs. The Target Hardware contains the code that represents the environment of the tests. The Controller Hardware is associated with SIL.[22]

The HIL method has its limits. The performance of the test depends on the model implemented. It is complex to implement a precise physical system and a factor of uncertainty is present. To minimize the impact of this factor, it is important to test and validate the design of the controller. The test on the physical plant cannot be seen as the result but as a prediction of what could result. Further tests on the real-model are still important.[4]

However, HIL simulation offers cost savings over validation testing. It is possible to determine if the plant model is valid and perform design changes earlier in the project which leads to several benefits. You can identify design problems and make changes without the burden of using an assembled final product.[22]

Conclusion

In business, two development processes have been widely used: the V-Model and the Agile methods. The choice of process depends on the project. For instance, the Agile methodology allows for more flexibility, in case the client has not clearly defined the product he wants in the first place. This process will ease adding functionalities down the line. The major difference between these two processes lies in the tests. These are directly done alongside the development in the Agile methods, whereas the system is tested once developed for the V-Model. Tests can be classified into two categories according to their characteristics: white-box or black-box. In the development of embedded systems, simulation techniques such as MIL, SIL and HIL are favored to correct potential errors as they generally reduce costs and cut testing time.

Our mission will be to add functionalities to the Jammer box. We will take care of designing the software, with the development of a driver to manage the storage on an SD card. We will also improve the user interface to be able to easily access and use that data.

We will finally imagine new test scenarios, we could for example test the system against malicious injections. This is very interesting for modern cars where security is of paramount importance. Indeed, vehicles controlled by software need to be protected against external attacks. Developers should not only rely on SIL or HIL tests to track down known vulnerabilities. They must anticipate new problems that may arise concerning the safety of the vehicle.

References

[1] S. G.ABEYARATNE A. VIDANAPATHIRANA S. D. DEWASURENDRA. "Model in the loop testing of Complex Reactive Systems". In: (2013), pp. 30–35. DOI: 10.1109/ICIInfS.2013.6731950.

[2] M. Aniche. "Software Testing: From Theory to Practice". In: ().

[3] V. Antinyan and M. Staron. "Mythical Unit Test Coverage". In: (2019), pp. 267–268. DOI: 10.1109/ICSE-SEIP.2019.00038.

[4] M. Bacic. "On hardware-in-the-loop simulation". In: (2005), pp. 3194–3198. DOI: 10.1109/CDC.2005.1582653.

[5] Carsten Beyer, Jan Emmerich, and Uwe Werner. "Software in the loop — A window lifter model to guide students through the software development process". en. In: *2017 IEEE Global Engineering Education Conference (EDUCON)*. Athens, Greece: IEEE, Apr. 2017, pp. 1488–1493. ISBN: 978-1-5090-5467-1. DOI: 10.1109/EDUCON.2017.7943045. URL: http://ieeexplore.ieee.org/document/7943045/ (visited on 01/25/2021).

[6] H. K. Brar and P. J. Kaur. "Differentiating Integration Testing and unit testing". In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015, pp. 796–798.

[7] B. Cornelissen et al. "A Systematic Survey of Program Comprehension through Dynamic Analysis". In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 684–702. DOI: 10.1109/TSE.2009.28.

[8] R. Hamlet. "Unit testing for software assurance". In: *Proceedings of the Fourth Annual Conference on Computer Assurance, 'Systems Integrity, Software Safety and Process Security*. 1989, pp. 42–48. DOI: 10.1109/CMPASS.1989.76037.

[9] Jean Hartmann et al. "A UML-based approach to system testing". en. In: *Innovations in Systems and Software Engineering* 1.1 (Apr. 2005), pp. 12–24. ISSN: 1614-5054. DOI: 10.1007/s11334-005-0006-0. URL: https://doi.org/10.1007/s11334-005-0006-0 (visited on 01/28/2021).

[10] "IEEE Standard for Software Unit Testing". In: *ANSI/IEEE Std 1008-1987* (1986), pp. 1–28. DOI: 10.1109/IEEESTD.1986.81001.

[11] Shane Warden James Shore. *The Art of Agile Development, 2nd Edition*. en. 2021. URL: https://learning.oreilly.com/library/view/the-art-of/9781492080688/ (visited on 01/29/2021).

[12] Paul C. Jorgensen. *Decision Table–Based Testing*. eng. 2014.

[13] B. Liu, H. Zhang, and S. Zhu. "An Incremental V-Model Process for Automotive Development". In: (2016), pp. 225–232. DOI: 10.1109/APSEC.2016.040.

[14] Michael A. Long. “Software Regression Testing Success Story”. In: (1993), pp. 271–272. DOI: 10.1109/TEST.1993.470688.

[15] Dan Pitica Marius Muresan. “Software in the Loop environment reliability for testing embedded code”. In: (2012), pp. 325–328. DOI: 10.1109/SIITME.2012.6384402.

[16] A R Plummer. “Model-in-the-loop testing”. In: (2006), pp. 183–199. DOI: 10.1243/09596518JSCE207.

[17] D.H.C. Silva et al. “Design of Controllers Applied to Autonomous Unmanned Aerial Vehicles Using Software In The Loop”. en. In: *2019 20th International Carpathian Control Conference (ICCC)*. Krakow-Wieliczka, Poland: IEEE, May 2019, pp. 1–6. ISBN: 978-1-72810-702-8. DOI: 10.1109/CarpathianCC.2019.8766036. URL: https://ieeexplore.ieee.org/document/8766036/ (visited on 01/25/2021).

[18] Cassia de Souza Carvalho and Tatsuhiro Tsuchiya. “Coverage Criteria for State Transition Testing and Model Checker-Based Test Case Generation”. eng. In: IEEE, 2014, pp. 596–598. DOI: 10.1109/CANDAR.2014.111.

[19] Latha Kant Stephanie Demers Praveen Gopalakrishnan. “A Generic Solution to Software-in-the-Loop”. In: (2007), p. 6. DOI: 10.1109/MILCOM.2007.4455268.

[20] S. Stolberg. “Enabling Agile Testing through Continuous Integration”. In: *2009 Agile Conference*. Aug. 2009, pp. 369–374. DOI: 10.1109/AGILE.2009.16.

[21] *What Is Hardware-In-The-Loop Simulation?* URL: https://www.mathworks.com/help/physmod/simscape/ug/what-is-hardware-in-the-loop-simulation.html?w.mathworks.com (visited on 01/28/2021).

[22] *What Is Hardware-in-the-Loop?* en. URL: https://www.ni.com/fr-fr/innovations/white-papers/17/what-is-hardware-in-the-loop-.html (visited on 01/29/2021).

[23] Bin Zhang et al. “Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing”. eng. In: *IET software* 12.6 (2018), pp. 507–519. ISSN: 1751-8806. DOI: 10.1049/iet-sen.2017.0200.