Mondragon Unibertsitatea

DOCTORAL THESIS

AI-BASED PERFORMANCE ISSUE DETECTION IN CPS SOFTWARE UPDATES

AITOR GARTZIANDIA ALUSTIZA | Arrasate-Mondragón, 2023

AI-based Performance Issue Detection for Cyber-Physical System Software Updates

PhD thesis in Applied Engineering

Author:

Aitor Gartziandia Alustiza

Supervised by:

Dr. Goiuria Sagardui

Dr. Aitor Agirre

Computer and Electronics Department
Mondragon Goi Eskola Politeknikoa
Mondagon Unibertsitatea

Arrasate
September 2023

“*Amari.*

Acknowledgments

Now that this Ph.D. is reaching an end, I notice that a whole chapter of my life is about to close, so it's time for me to look back to the path that has brought me here.

On this path, I see in the beginning a kid who started his studies at the university with no idea what his place in the world was. Instead, in the end, I see a man who finishes a Ph.D. thesis and is anxious to start new adventures that will let him continue growing. This kind of personal evolution is of course something to be proud of oneself, but when I analyze all the steps of this journey, all peaks and valleys, I realize that I have not been walking alone.

Now that I see this journey with perspective, I understand that I have always had people around me who have always kept me on track, without whom I would never reached the point where I find myself now. I also feel ashamed for not having appreciated these people enough until now, but although it comes late, I would like to use this occasion to thank these people as they deserve. And as I write these lines, the faces of these people are coming to my mind incessantly, and I notice that the faces that appear the most, are the ones that sustain you when the times are darkest.

The face that appears the most is the one of my mother. She is not the one who most helped me during the thesis, but she is the one who suffered the most during a time in my life when I was lost. She knew how to guide me and trusted me to the point that I was encouraged to start a thesis. You are the major responsible for the point where I find myself now, so thank you a lot, and this thesis is for you!

Another person who deserves a special mention in these lines is my girlfriend, Elene. We have been through hard times during these years where we have both suffered, but you have always been by my side even in my worst days. I will never be able to pay you back for the support and loyalty you show me, so all I

can do is thank you every time I get the chance.

And then there are the people that make you forget you are screwed, the ones that make you laugh even when the day has gone downhill. These are my brother Imanol and my friends from "La Tramont". The simple fact of having a beer with these people makes you forget your problems and recover your energy. The beer also plays its role, but the most important thing is the company!

Finally, I want to thank Goiuria Sagardui and Aitor Agirre, my directors in this thesis, who have advised me and helped me to get to where I am now. I would like to extend these thanks to Ikerlan, who gave me the opportunity to start working as an intern and grow with them for 7 years. I hope this strong commitment to young talent continues for a long time!

Everyone mentioned here has made me a better engineer and person so I will be eternally grateful to all of you.

Eskerrik asko denoi bihotzez!

Declaration

Nik, Aitor Gartziandia Alustizak, aitortzen dut Doktore Tesi hau originala dela, nire lan pertsonalaren emaitza, eta ez dela aurkeztu aurretik beste titulu edota kalifikazio profesionalik lortzeko. Kanpoko iturrietatik hartutako ideiak, formulazioak, irudiak eta ilustrazioak behar bezala aipatu eta erreferentziatu dira.

Hereby I, Aitor Gartziandia Alustiza declare, that this Doctoral Thesis is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Yo Aitor Gartziandia Alustiza declaro que esta Tesis Doctoral es original, fruto de mi trabajo personal, y que no ha sido previamente presentado para obtener otro título o calificación profesional. Las ideas, formulaciones, imágenes, ilustraciones tomadas de fuentes ajenas han sido debidamente citadas y referenciadas.

Arrasate, September 2023

Aitor Gartziandia Alustiza

Abstract

Cyber-Physical Systems (CPSs) are systems that integrate computation and communication with the physical world. Software systems that are embedded in CPSs usually have a large life-cycle and are continuously evolving in order to incorporate new requirements, bug fixes, and to deal with hardware obsolescence. Despite the fact that the rise of new technologies provides an opportunity to improve CPS life-cycle management methods, there are still many complex issues that need to be addressed throughout its entire life cycle, from development to maintenance. In this sense, the increasing expansion of IoT and CPSs has highlighted the need for additional mechanisms related to the deployment and monitoring of these systems in operation, due to the challenge that represents the diversity of environments where these systems are deployed. The heterogeneity of the environments, along with the high configurability of CPSs, make testing the these systems under every possible condition impossible, increasing the possibility of errors appearing once the software is deployed in operation. This is especially concerning with performance errors, which are particularly difficult to detect in the lab.

In this context, this Ph.D. study proposes a mechanism to detect performance errors in Cyber-Physical Systems. The aim is to investigate different AI techniques to build a performance oracle, which predicts the expected performance of the system in operation and detects performance errors by comparing the predicted value with the actual monitored performance metrics. This prediction is obtained by training a performance model with data from the execution of previous versions of the software. The performance oracle is then encapsulated as a microservice, so that it can be seamlessly deployed along with other services to detect performance errors in operation. In order to make the performance oracle useful in multiple, heterogeneous environments, different training strategies have been investigated.

The method was evaluated by applying it to an industrial use case provided

by Orona, using its dispatching traffic algorithm for elevator systems. Results show that the used AI techniques can be used to predict CPS performance and detect errors, even in installations where data for training was not available.

Laburpena

Sistema ziberfisikoak (CPSak) konputazioa eta komunikazioa mundu fisikoarekin integratzen dituzten sistemak dira. CPSetan integratutako software sistemek bizi-ziklo luzea izaten dute eta etengabe eboluzionatzen ari dira baldintza berriak inplementatzeko, akatsak zuzentzeko edota hardwarearen zaharkitzeari aurre egiteko. Teknologia berrien gorakadak CPSen bizi-zikloaren kudeaketa metodoak hobetzeko aukera ematen duen arren, oraindik ere gai konplexu asko daude aztertzeko bizi-ziklo osoan zehar, garapenetik hasi eta mantenketa lanetararte. Ildo horretan, IoT eta CPSen hedapen gero eta handiagoak agerian utzi du martxan dauden sistemen kudeaketarekin lotutako mekanismo gehigarrien beharra, sistema horiek ezarrita dauden inguruneen aniztasunak dakarren erronka dela eta. Ingurune hauen heterogeneotasuna eta CPSen konfiguragarritasun handia direla eta, ezinezkoa da sistemak jasango dituen baldintza guztiak probatzea, softwarea martxan dagoenean akatsak agertzeko arriskua areagotuz. Horrek bereziki eragiten die errendimendu akatsei, laborategian detektatzeko bereziki zailak baitira.

Testuinguru horretan, doktorego tesi honek *sistema ziberfisikoetan* errendimendu akatsak detektatzeko mekanismo bat proposatzen du. Helburua Adimen Artifizialeko teknikak ikertuz errendimendu orakulu bat eraikitzea da, funtzionamenduan dagoen sistemen errendimendua iragartzeko eta errendimendu akatsak detektatzeko, aurreikusitako balioa monitorizatutako benetako errendimendu metrikekin alderatuz. Iragarpen hori errendimendu eredu baten bidez lortzen da, softwarearen aurreko bertsioen exekuziotik datozen datuekin entrenatu dena. Ondoren, errendimendu orakulua mikroszerbitzu gisa kapsulatu da, eta beste zerbitzu batzuekin batera zabaldu daiteke, gauzatzen ari diren errendimendu akatsak detektatzeko. Errendimendu orakulua ingurune anitz eta heterogeneoetan erabilgarria izan dadin, hainbat entrenamendu estrategia ikertu dira.

Metodoa Oronak eskainitako erabilera industrialeko kasu batean aplikatuz

ebaluatu da, igogailu sistemetarako duen trafiko algoritmoa erabilita. Emaitzek erakusten dutenez frogatutako Adimen Artifizialeko teknikak CPSen errendimendua aurreikusteko eta akatsak detektatzeko erabil daitezke, baita entrenamendurako daturik erabilgarri ez dagoen instalazioetan ere.

Resumen

Los *sistemas ciber-físicos* (CPSs) son sistemas que integran la computación y la comunicación con el mundo físico. Los sistemas software integrados en los CPSs suelen tener un ciclo de vida largo y están en continua evolución para incorporar nuevos requisitos, corregir errores y hacer frente a la obsolescencia del hardware. A pesar de que el auge de las nuevas tecnologías brinda la oportunidad de mejorar los métodos de gestión del ciclo de vida de los CPSs, sigue habiendo muchas cuestiones complejas que deben abordarse a lo largo de todo su ciclo de vida, desde el desarrollo hasta el mantenimiento. En este sentido, la creciente expansión del IoT y los CPSs ha puesto de manifiesto la necesidad de mecanismos adicionales relacionados con el despliegue y monitorización de estos sistemas en operación, debido al reto que representa la diversidad de entornos donde se despliegan estos sistemas. La heterogeneidad de los entornos, junto con la alta configurabilidad de los CPSs, hace que probar el sistema bajo todas las condiciones posibles sea imposible, aumentando las posibilidades de que aparezcan errores una vez desplegado el software en operación. Esto afecta especialmente a los errores de rendimiento, que son particularmente difíciles de detectar en laboratorio.

En este contexto, este estudio de doctorado propone un mecanismo para detectar errores de rendimiento en *sistemas ciber-físicos*. El objetivo es investigar en el uso de diferentes técnicas de IA para construir un oráculo de rendimiento que prediga el rendimiento esperado del sistema en ejecución y detectar errores de rendimiento comparando el valor predicho con las métricas de rendimiento reales monitorizadas. Esta predicción se obtiene mediante un modelo de rendimiento entrenado con datos procedentes de la ejecución de versiones anteriores del software. Después, el oráculo de rendimiento se ha encapsulado como un microservicio, que puede desplegarse de forma sencilla junto con otros servicios para detectar errores de rendimiento en ejecución. Para que el oráculo de rendimiento sea útil en entornos múltiples y heterogéneos, se han investigado diferentes estrategias de entrenamiento.

El método se evaluó aplicándolo a un caso de uso industrial proporcionado por Orona, utilizando su algoritmos de tráfico para sistemas de ascensores. Los resultados muestran que las técnicas de IA probadas pueden utilizarse para predecir el rendimiento de los CPSs y detectar errores, incluso en instalaciones en los que no se dispone de datos para el entrenamiento.

Contents

1 Introduction 1
1.1 Motivation and Scope . . . 1
1.2 Research Methodology . . . 2
1.3 Technical Contribution . . . 4
1.4 Publications . . . 5
1.4.1 Journal Articles . . . 5
1.4.2 International Conferences . . . 5
1.5 Document Structure . . . 6
2 Technical Background 7
2.1 Cyber Physical Systems life-cycle management . . . 7
2.1.1 Cyber-Physical Systems . . . 7
2.1.2 DevOps . . . 9
2.1.3 Taxonomy for Eliciting Design-Operation Continuum Requirements of CPSs . . . 9
2.2 Adeptness . . . 16
2.2.1 Microservice Architectures . . . 16
2.2.2 Adeptness architecture . . . 17
2.3 Performance Bugs . . . 22
2.4 AI Techniques . . . 23
2.4.1 Machine Learning . . . 24
2.4.2 Neural Networks . . . 25
2.4.3 Genetic Programming . . . 27
3 State of the Art 29
3.1 Literature Review Methodology . . . 29
3.1.1 Definition of Research Questions . . . 29
3.1.2 Search Process . . . 30
3.1.3 Inclusion Criteria . . . 31
3.1.4 Data Collection . . . 32

3.2 Software Deployment in CPSs . . . 32
3.3 Performance Testing on CPSs . . . 36
3.4 Performance Prediction . . . 37
3.5 Critical analysis of the State of the Art . . . 38
4 Theoretical Framework 41
4.1 Objectives . . . 41
4.2 Hypotheses . . . 42
4.3 Overview . . . 42
4.4 Case Study: Orona's Dispatching Algorithm . . . 44
5 The Performance Oracle 49
5.1 The Oracle as a Microservice . . . 50
5.1.1 Interfaces . . . 50
5.1.2 Sub-components . . . 52
5.2 The Oracle in the Adeptness Architecture . . . 53
5.2.1 Deployment . . . 54
5.2.2 Configuration . . . 54
5.2.3 Execution . . . 55
5.3 Requirements . . . 55
5.3.1 Mandatory Requirements . . . 56
5.3.2 Additional Requirements . . . 57
5.4 Evaluation . . . 58
5.5 Conclusion . . . 60
6 Performance Model 61
6.1 Training Data . . . 62
6.1.1 Monitoring . . . 62
6.1.2 Pre-processing . . . 64
6.2 AI Techniques . . . 65
6.2.1 Configuration . . . 65
6.2.2 Selection Criteria . . . 67
6.3 Evaluation . . . 68
6.3.1 Research Questions . . . 68
6.3.2 Experimental Setup . . . 68
6.3.3 Results . . . 81
6.3.4 Discussion . . . 85
6.3.5 Threats to Validity . . . 88
6.4 Conclusion and Future Work . . . 89
7 Arbiter 91
7.1 General Logic . . . 91
7.2 Parameters . . . 94
7.2.1 Time-span . . . 94
7.2.2 Thresholds . . . 95
7.3 Evaluation . . . 95

7.3.1 Research Questions . . . 95
7.3.2 Experimental Setup . . . 96
7.3.3 Results . . . 98
7.3.4 Discussion . . . 101
7.3.5 Threads to Validity . . . 102
7.4 Conclusion and Future Work . . . 103
8 Conclusion 105
8.1 Summary of the Contributions . . . 105
8.1.1 Hypotheses Validation . . . 106
8.1.2 Limitations of the Proposed Solution . . . 107
8.2 Lessons Learned . . . 108
8.3 Future Work . . . 109
Bibliografy 118
A Publications 119
A.1 Towards a Taxonomy for Eliciting Design-Operation Continuum Requirements of Cyber-Physical Systems . . . 119
A.2 Using Regression Learners to Predict Performance Problems on Software Updates: a Case Study on Elevators Dispatching Algorithms . . . 131
A.3 Microservices for Continuous Deployment, Monitoring and Validation in Cyber-Physical Systems: an Industrial Case Study for Elevators Systems . . . 142
A.4 Machine Learning-based Test Oracles for Performance Testing of Cyber-Physical Systems: An Industrial Case Study on Elevators Dispatching Algorithms . . . 151

List of Figures

1.1 Overview of the research methodology followed during this Ph.D. 3

2.1 Overview of the components of CPSs 8
2.2 DevOps methodology 9
2.3 Taxonomy of Design-Operation Continuum Requirements for CPSs 11
2.4 Difference between the monolith and microservices 17
2.5 Overview of the Adeptness architecture components 19
2.6 General overview of the main ML categories 24
2.7 Example of an architecture of a Neural Network 26

4.1 Overview of the method developed in the Ph.D. 43
4.2 Overview of the architecture of an elevator installation from Orona 45

5.1 Synchronous and asynchronous interfaces of the Performance Oracle 51
5.2 Overview of the sub-components of the Performance Oracle . 53
5.3 Overview of the deployment process of the Performance Oracle 54
5.4 Overall overview of the configuration of the Performance Oracle 55
5.5 Sequence diagram of the execution of the Performance Oracle . 56

6.1 Passengers activity of real installation and theoretical profiles obtained with Elevate 71
6.2 Execution time of the dispatching algorithm per active call in SiL 72
6.3 Execution time of the dispatching algorithm per active call in HiL 76
6.4 Execution time of the dispatching algorithm per active calls in HiL without outliers 78

7.1 The three reasons why a test can be catalogued as FAIL (blue signal refers to the reference valur and orange signal refers to the value obtained by the software version under test) 92

List of Tables

3.1 Overview of the characteristics of the available deployment tools 35

6.1 Summary of the experimental setup in SiL 69
6.2 Main characteristics of the used test cases during the experimental scenarios 71
6.3 Description of the designed testing scenarios 74
6.4 Summary of the experimental setup in HiL 75
6.5 Description of the features used to train the models 77
6.6 Summary of the experimental setup in multi-environment context 79
6.7 Characteristics of the installations used to train the multi-installation model 80
6.8 Description of the features used to train the models 80
6.9 Summary of the experimental setup 82
6.10 MAPE for the models trained with data from single installation SiL context 83
6.11 Footprint in KB for the models trained with data from single installation SiL context 83
6.12 Inference time in μs for the models trained with data from single installation SiL context 84
6.13 MAPE for the models trained with data from a single installation HiL context 84
6.14 Footprint in KB for the models trained with data from a ingle installation HiL context 85
6.15 Inference time in μs for the models trained with data from a single installation HiL context 85
6.16 MAPE for the models trained with data from multiple installations in HiL context 86

6.17 Footprint in KB for the models trained with data from multiple installations in HiL context 87
6.18 Inference time in μs for the models trained with data from multiple installations in HiL context 88

7.1 Results summary of the arbiter when tested with theoretical data from single installation SiL context 99
7.2 Results summary of the arbiter when tested with real data from single installation SiL context 100
7.3 Results summary of the arbiter when tested with theoretical data from single installation HiL context 101
7.4 Results summary of the arbiter when tested with real data from single installation HiL context 102
7.5 Results summary of the arbiter when tested with theoretical data from multiple installations in HiL context 103

Acronyms

AI Artificial Intelligence. 4, 23

AWT Average Waiting Time. 15, 45

CD Continuous Deployment. 12

CI Continuous Integration. 12

CM Continuous Monitoring. 13

CPS Cyber-Physical System. 1

DL Deep Learning. 63

FN False Negative. 97

FP False Positive. 97

GA Genetic Algorithms. 64

GP Genetic Programming. 24

HiL Hardware-in-the-Loop. 10

JT Journey Time. 45

kNN K-Nearest Neighbours. 25

MAPE Mean Absolute Percentage Error. 59, 86

MiL Model-in-the-Loop. 10

ML Machine Learning. 22, 24

NN Neural Network. 24

OS Operating System. 12

PiL Processor-in-the-Loop. 75

QoS Quality-of-Service. 22

ReLU Rectified Linear Unit. 66

RGP Regression Gaussian Process. 74

RQ Research Question. 29, 68

SGC Stochastic Gradient Descent. 66

SiL Software-in-the-Loop. 10

SOA Service Oriented Architecture. 16

SUT System Under Test. 14

SVM Support Vector Machines. 25, 74

SVR Support Vector Regression. 25

TN True Negative. 97

TP True Positive. 97

1 Introduction

This chapter introduces the main motivation and scope of the research conducted in this Ph.D. study in Section 1.1, as well as the used research methodology in Section 1.2. The main technical contributions are also summarized in Section 1.3 highlighting the publications where these contributions were presented in Section 1.4. Finally, the structure of the document is exposed in Section 1.5.

1.1 Motivation and Scope

Cyber-Physical Systems (CPSs) are gaining an increasing interest among companies and researchers driven by the growth of fields like Industry 4.0, autonomous vehicles, and smart cities [60]. The propagation of new CPS applications involves a growth in the complexity of these systems, while requirements become increasingly stringent. Hence, new practices and technologies are necessary to deal with the development and management of these systems in such scenarios.

CPS usually have a large life-cycle and its software is continuously evolving due to the incorporation of new requirements, bug fixes or hardware obsolescence [58]. Moreover, many factors such as high software-hardware coupling, real-time requirements, or high configurability (i.e., the capability to work with a wide range of different configurations) lead to a high complexity during the whole life-cycle, from design to maintenance.

Given the complexity of these systems, the use of Design-Operation Continuum Engineering methods such as DevOps [36] becomes relevant to guarantee the correct execution of the application [7]l. In contrast with other domains, such as web engineering, where the development and operation phases are tightly connected, the current process of CPS development and maintenance is very fragmented, which makes the life-cycle management tasks expensive [3]. Connecting the different development and operation methods among them can lead to an improvement in the CPS capabilities, reducing efforts and costs, so more

focus in this field is needed to take advantage of new technologies that make the adoption of these practices easierexecution of the application [7].

Within the DevOps pipeline, the deployment step in CPSs might have critical impacts unlike other less critical systems (e.g., web pages, mobile apps, etc.) [52]. Many of these systems may be located in environments where the resource availability can fluctuate considerably [51] and the system may be exposed to unforeseen situations. In addition, these systems interact closely with the physical world, including humans, which forces taking special care of the reliability and safety of the software deployments [12]. These software deployments may be frequent due to software updates caused by the implementation of new functionalities or bug detections. Hence, the continuous software updates require the adoption of an automated deployment strategy which would speed up the process and ensure its reliability.

As mentioned, environmental conditions can play a significant role in the execution of the system [25]. The remote location of software in multiple heterogeneous environments provokes these conditions to be very varied and dynamic. Besides, CPSs are very configurable systems which brings with it a limitation when testing the system in the lab, as it is unfeasible to recreate all the conditions and configurations the system will work on [24].

In this context, performance problems are likely to appear, as performance may be highly influenced by hardware, workload, connectivity, or other environmental conditions that can degrade the behavior of the system [14]. Performance error detection becomes difficult in such conditions, as establishing acceptance values for performance metrics under untested conditions becomes impossible and many causes that lead to the manifestation of these errors may appear due to the interaction of the system with the real environment [67].

1.2 Research Methodology

The research methodology to be followed in this Ph.D. is an iterative model named design and creation [64]. The methodology consists of five steps. Each step has an output that can be understood as the result of the activity related to the process step. Figure 1.1 shows an overview of the methodology.

The steps of the process are described below:

- **Awareness of Problem:** It is the first step, where a problem is detected from new developments in industry or literature review, and provides a proposal for solving the problem.

  The main problem this study aims to solve is the following: the detection of performance problems on software updates.

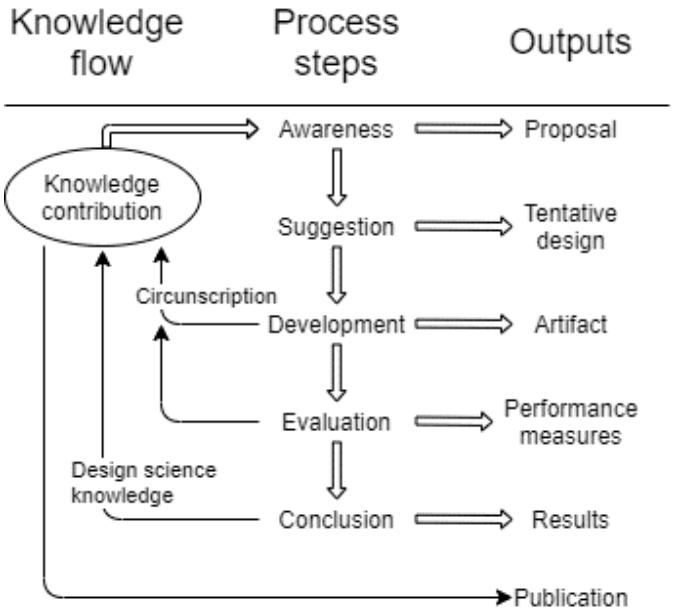


Figure 1.1: Overview of the research methodology followed during this Ph.D.

- **Suggestion:** This is a creative step where new functionalities are envisioned and a tentative design of an experiment to solve the problem is suggested.

  The suggested method to solve the problem is a performance error detection method based on microservices capable of predicting the performance of new releases, to compare the predicted and actual value and decide if an error exists or not.

- **Development:** In this step, the tentative design is developed and implemented, following different techniques depending on the artifact to be created, providing a novel artifact.

  The development of the Performance Oracle involves researching different aspects, that have represented different cycles. In each cycle, the awareness of a different problem is gained, and a solution is suggested and developed. The developed artifacts were evaluated, and conclusions were drawn to continue gaining knowledge.

  When the obtained knowledge is sufficiently relevant, a journal or conference paper has been published.

- **Evaluation:** This step evaluates the developed artifact according to certain evaluation criteria and the hypotheses are tested based on the measures.

  In the evaluation step, the developed methodology is evaluated by measuring its precision to detect performance errors and other features such as resource usage.

- **Conclusion:** It is the last step of a research cycle or effort, where the iterative model finishes and results are consolidated, detailing the obtained knowledge.

  In this step, the obtained results were observed to analyze the applicability of the methodology to real scenarios, the limitations it may present and the future work to be done in this field.

- **Publication:** When the knowledge contribution is considered relevant enough, the results are published in a conference or journal article.

  Specifically in this Ph.D. thesis, a number of papers and the present document have been written to explain the developed method and contributions made to the scientific community.

1.3 Technical Contribution

This section provides a summary of the main contributions of this thesis:

- A method to detect performance errors in operation for software updates in CPSs. The Performance Oracle is built as a microservice to be easily deployed in heterogeneous environments.
- A Performance Model, capable of predicting the performance of a software based on its input data. The use of different Artificial Intelligence (AI) techniques is investigated to build a model with high precision and low resource consumption.
- An analysis of the effect of the use of different data types in the prediction precision of the Performance Model. In this sense, The use of data monitored from real installations and theoretical data obtained from simulation tools has been investigated.
- Development of an Arbiter, which decides whether a performance error exists or not based on the prediction of the model and the real performance of the new software.
- Integration of the Performance Oracle microservice with the Adeptness architecture [1] to leverage the deployment, monitoring, and validation capabilities of this architecture.
- The evaluation was performed by using an industrial case study provided by Orona[1], one of the leading elevator companies in Europe. In the evaluation, the capabilities of the Performance Oracle to be used in practice for performance bug detection were evaluated.

[1] https://www.orona-group.com/

1.4 Publications

Different peer-reviewed publications were published in a journal and at conferences during the Ph.D. Notice that some of the conference publication papers are ranked by a ranking system supported by the Spanish Informatics Scientific Society (SCIE)[2]. The journal publications are scored with their current Journal Citation Report (JCR) quartile.

1.4.1 Journal Articles

During the Ph.D., a journal article was published.

- Gartziandia, A., Arrieta, A., Ayerdi, J., Illarramendi, M., Agirre, A., Sagardui, G., & Arratibel, M. (2022). Machine learning-based test oracles for performance testing of cyber-physical systems: An industrial case study on elevators dispatching algorithms. Journal of Software: Evolution and Process, 34(11), e2465. **JCR Ranking: Q3**

1.4.2 International Conferences

A total of three publications were achieved at international conferences, which are listed below:

- Ayerdi, J., Garciandia, A., Arrieta, A., Afzal, W., Enoiu, E., Agirre, A., Sagardui, G., Arratibel, M. & Sellin, O. (2020, August). Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In 2020 IEEE 28th International Requirements Engineering Conference (RE) (pp. 280-290). IEEE. **SCIE Ranking: A**
- Gartziandia, A., Ayerdi, J., Arrieta, A., Ali, S., Yue, T., Agirre, A., Sagardui, G. & Arratibel, M. (2021, March). Microservices for continuous deployment, monitoring and validation in cyber-physical systems: an industrial case study for elevators systems. In 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C) (pp. 46-53). IEEE. **SCIE Ranking: A**
- Gartziandia, A., Arrieta, A., Agirre, A., Sagardui, G. & Arratibel, M. (2021, March). Using regression learners to predict performance problems on software updates: a case study on elevators dispatching algorithms. In Proceedings of the 36th Annual ACM Symposium on Applied Computing (pp. 135-144). **SCIE Ranking: A**

[2] http://gii-grin-scie-rating.scie.es

1.5 Document Structure

The structure of the thesis is as follows. Chapter 1 provides an introduction to the thesis, outlining the motivation, the research methodology used, the contributions made, and the publications achieved. Chapter 2 exposes the necessary terminology and background concepts used throughout the document. An analysis of current studies and the most relevant research related to the topic is presented in Chapter 3. Chapter 4 presents the theoretical framework, discussing the research objectives, the hypotheses, a summary of the proposed solution, and the case study used.

In Chapter 5 the Performance Oracle is introduced, detailing its components, communication interfaces, the interaction with the Adeptness architecture, the requirements it must fulfill, and the methodology used for its evaluation. Chapter 6 describes the training process to build the Performance Model, detailing the data monitoring, the AI techniques used and its evaluation. Chapter 7 details the development and evaluation of the Arbiter used to raise the verdicts of the oracle.

Finally, in Chapter 8, the contributions of the thesis are summarized, the hypotheses are validated, the main limitations of the method are discussed and a set of lessons learned are provided. Furthermore, future research directions are proposed.

2 Technical Background

This chapter introduces some theoretical background about the main concepts within the scope of this thesis. First, CPS life-cycle management considerations are explained in Section 2.1, in Section 2.2 the H2020 Adeptness Project is introduced, which is closely related to this Ph.D. thesis. Then, the nature of performance bugs is explained in Section 2.3 and, finally, different AI techniques used in this Ph.D. are described in Section 2.4.

2.1 Cyber Physical Systems life-cycle management

In this section, the life-cycle management of CPSs is discussed, which is the main scope of this thesis. Firstly, the main characteristics of CPSs are describe, afterward, the DevOps concept is introduced, and, finally, a taxonomy for eliciting Desing-Operation continuum requirements in CPSs is presented.

2.1.1 Cyber-Physical Systems

The term Cyber-Physical System refers to types of systems that integrate global networking services and interaction with the physical world, by means of sensors and actuators, to provide coherent and intelligent services [53]. In Figure 2.1, the main components of a CPS can be seen, showing how CPSs make use of embedded systems, along with the physical world and the interaction with other systems or humans.

These systems currently include many different systems such as robotics, autonomous vehicles, elevators or trains, but they share certain common characteristics [59]:

- Resource-constrained embedded software: The software is located in embedded systems with limited resources such as computing, network bandwidth, etc., and stringent timing requirements.

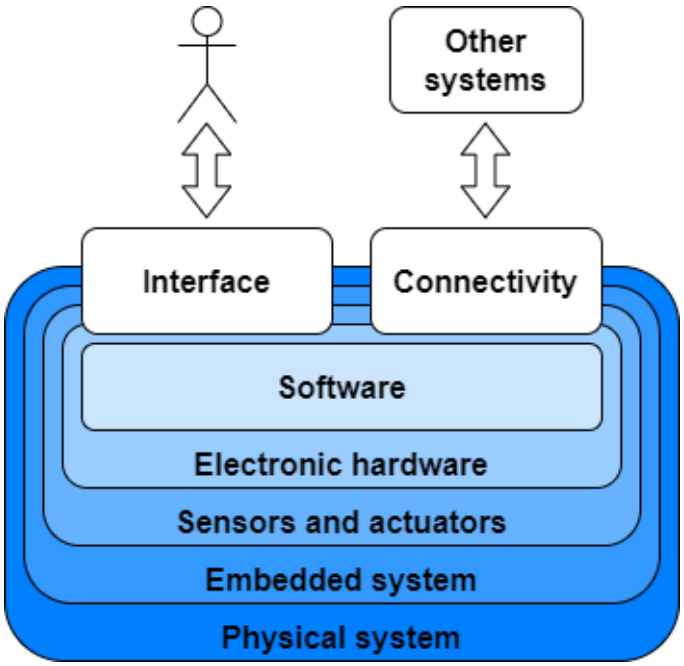


Figure 2.1: Overview of the components of CPSs

- Distributed and heterogeneous systems: CPSs are usually connected to many systems, which can be diverse, leading to a large-scale network of heterogeneous systems.
- High automation and re-configuration capabilities: The complexity of these systems requires automation and adaptive capabilities by means of human-machine interaction.
- Dependable operation: The system must guarantee high degrees of reliability, safety, and security.

These systems are inherently complex, and their life-cycle can last up to 30 years in sectors such as railway or elevation [7]. In these systems, an increasing trend is to implement most of the functionalities through software. During the life-cycle of these systems, the software continuously evolves due to many factors, such as hardware obsolescence, requirement changes, vulnerabilities, or bug corrections [58]. Consequently, this evolution requires reliable and automatic engineering methods for developing and operating CPSs.

In the last few years, there have been several improvements in terms of modeling and simulation techniques [6], [61], [56] to develop and validate complex CPSs from the early development stages. However, when the software is deployed in the CPS, the methods used during operation and maintenance do not have synergies with the methods used in development. In other contexts, such as web engineering, there are Design-Operation Continuum Engineering methods

such as DevOps that permit software development methods to be streamlined with methods for operation time.

2.1.2 DevOps

DevOps is an engineering paradigm that involves a set of practices to bring together development and operation activities by means of collaboration between the different actors (e.g., developers, testers, operation personnel, etc.). The aim is to shorten the lead time between a change request and the deployment in production using automation, agile software development, and continuous delivery pipelines. DevOps practices provide solutions to have a more efficient process which guarantees that (1) software updates are performed safely and quickly, (2) most faults are detected in the design phase and (3) problems that can emerge in operation can be reproduced in development in order to analyze and propose potential solutions.

In Figure 2.2, the typical DevOps pipeline is depicted, which specifies the tasks that must be undertaken to guarantee frequent and reliable software delivery in a continuous manner. The DevOps pipeline proposes a cyclic feedback process, where the operational data serves to development tasks as inputs to develop an improved version of applications.

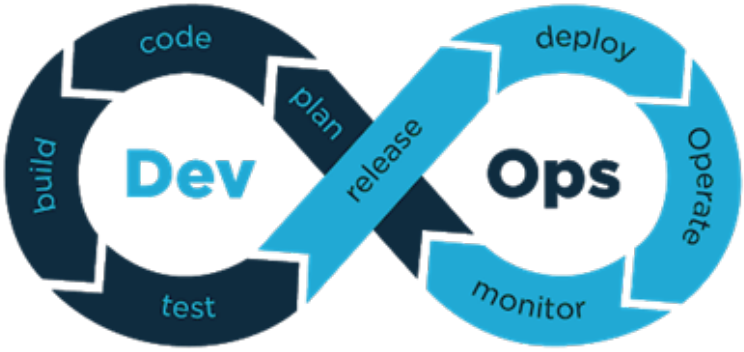


Figure 2.2: DevOps methodology

2.1.3 Taxonomy for Eliciting Design-Operation Continuum Requirements of CPSs

As stated, traditional Design-Operation Continuum Engineering methods require substantial changes in order to be dependable enough for CPSs. More specifically, Design-Operation Continuum methods must provide solutions to have a more efficient process that guarantees safety and security in software releases, early error detection, and operation monitoring to reproduce errors in the lab.

In Figure 2.3, a taxonomy of relevant concepts is shown to ease the understanding of the rather complex development and maintenance process of CPSs [7]. This taxonomy aims to assist requirements analysts with the identification and categorization of the requirements related to different aspects of the CPS Design-Operation Continuum Engineering. The main purpose of this classification is to support the elicitation of new requirements and the easier identification of problems such as omissions, ambiguity, vagueness, conflicts, or duplication in the requirements. Furthermore, this classification is also helpful for determining the organizational roles responsible for each requirement, as well as for the management and reuse of the elicited requirements in later stages of the development life-cycle. The taxonomy was built in collaboration with the Ph.D. studies of Jon Ayerdi, which focused more on the validation aspects, while this thesis focused more on deployment aspects.

The main aspects identified by this taxonomy are introduced below:

- **Life-cycle Stage:** This facet represents the X-in-the-loop system execution level, which is an aspect specific to CPS development processes. This taxonomy defines the four classes identified as relevant and more common for CPSs, which are Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Hardware-in-the-Loop (HiL), and Operation. At the MiL test level, the software that controls the physical part of the CPS is a model. At the SiL test level, this model is replaced by executable software. At the HiL test level, the software is integrated with the real-time infrastructure (e.g., real target processor and operating system) and the physical part emulated within a real-time test bench.

- **Scope:** The taxonomy distinguishes three different scope classes depending on the applicability of the requirement. However, depending on the strategy of the company, the categorization provided can be refined or extended. The main classes identified are Organisation, Product, and Release.

- **Domain:** This facet categorizes the requirement by the domain in which it belongs, divided into two categories: Infrastructure and Application.

- **Subsystem:** This facet classifies a requirement by the Design-Operation Continuum subsystem for which it is relevant. This taxonomy considers the subsystems of Deployment, Monitoring, Validation, and Integration.

Below, the three main subsystems concerning this thesis are further explained, which are the Deployment, Monitoring, and Validation subsystems.

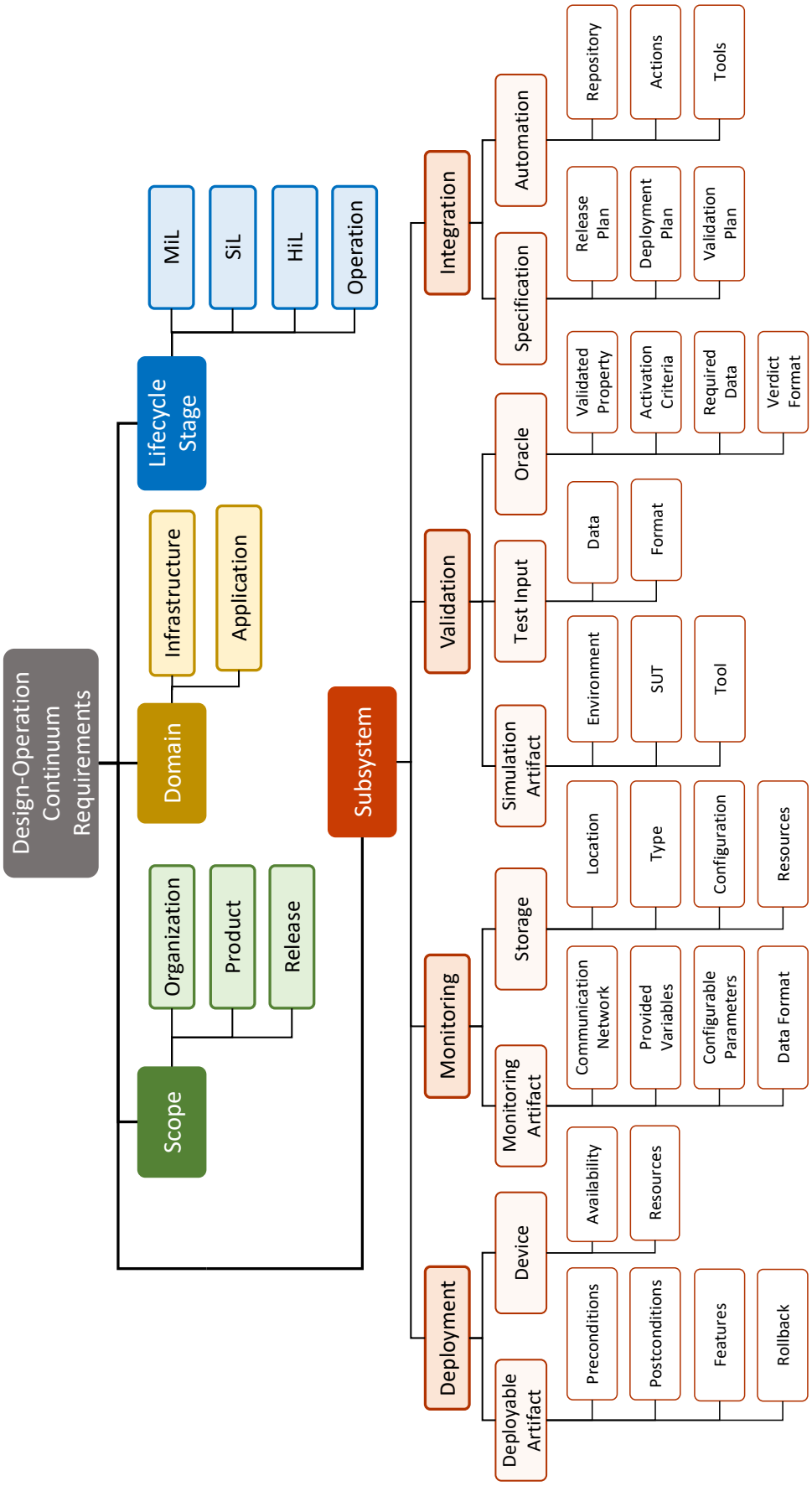


Figure 2.3: Taxonomy of Design-Operation Continuum Requirements for CPSs

2.1.3.1 **Deployment**

Automating deployment means providing the infrastructure that allows the automated Continuous Integration (CI) server to connect to the designated production/validation machine and upload executable and configuration files [46]. The Continuous Deployment (CD) subsystem allows the automatic deployment of a new software release in the virtual infrastructure for validation purposes. Afterward, the new release is deployed in the real CPS in Operation. In this subsystem, requirements that are necessary to deploy artifacts at the MiL/SiL/HiL/Operation of the system are specified. It is important to mention that in this category, the requirements for the Operation stage are the most demanding ones since aspects such as heterogeneous platforms or the status of the CPS before the deployment need to be considered. Examples of requirements in this category include "The deployment service shall provide support for ARMV7 boards", "The deployment service shall provide support for Linux and Windows Operating Systems (OSs)", "The downtime of the application during deployment shall be less than 15 sec" or "The system shall allow the deployment of artifacts by defining the allocation or by defining the memory requirements". Nowadays, releasing and deploying new software versions is a time-consuming and error-prone activity. Requirements in this category facilitate the automation of the CD for new releases.

Two subcategories have been defined for this subsystem: Devices and Deployable Artifacts.

- **Devices:** Automation of the deployment process in CPSs is highly complex due to the number of heterogeneous platforms, models, and interfaces necessary to deploy software releases. The goal of this subcategory is to collect requirements related to the variety and heterogeneity of hardware, software, and communications for which the deployment subsystem must provide support. These types of requirements have an influence on the deployment architecture that must be designed to provide support for all the devices in which an automatic deployment will be performed. This category also has an impact on the techniques and methods used for the deployment, e.g., container-based deployment mechanisms that are valid for Linux-based devices are not for embedded bare metal devices. There are different aspects to be specified: (1) *Resources* of the devices. Hardware, software (e.g., installed OS), and communication networks (e.g., CAN, Ethernet) that the deployment subsystem is going to deal with. (2) *Availability* of the device during deployment (e.g., maximum downtime of the device to perform the deploy).

- **Deployable Artifacts:** CPSs are composed of different software components distributed in heterogeneous devices. Deployable artifacts are "soft" components that are part of the CPS, such as software of new releases and

configuration files. When using Design-Operation Continuum methods, test oracles, monitors, etc. can also be considered deployable artifacts. This subcategory includes the specification of the features of the artifacts to be deployed. Requirements in this category define the deployment rules and are useful to ensure the pre- and post-deployment conditions and to design the rollback mechanisms. There are different aspects to be specified: (1) *Deployment conditions:* Pre-conditions specify criteria to be met before starting the deployment, e.g., "the CPS shall be out-of-service". Postconditions are verified after the deployment is completed, e.g., "the device reboots correctly" (2) *Features* of the artifacts: hardware requirements, e.g., minimum CPU or RAM requirements to execute the artifact, software requirements, e.g., supported OS, communication interface requirements, e.g., to be deployed in a device with access to CAN or/and the allocation of the artifact, e.g., in which device shall be deployed, (3) *Rollback* policy in case of deployment failure, e.g., "The system shall support the remote rollback to a previous version".

2.1.3.2 **Monitoring**

The goal of Continuous Monitoring (CM) is to extract data from a system so that it can be analyzed [35]. Monitoring in the deployment ensures that certain conditions are met before and after deploying. In the validation process, it provides data to the oracles so that they can provide a verdict. Besides, it can also be useful to observe and record the status of the infrastructure/application and later reproduce real scenarios in simulation. Examples of requirements in this category include *"Monitoring data from MiL/SiL/HiL test executions shall be available through logs", "Monitors shall provide connectors for CAN and Ethernet", "Monitoring data for the last day shall be persisted for further analysis"*.

This category facilitates gathering monitoring requirements at different lifecycle stages and levels of a CPS. Two subcategories have been defined:

- **Monitoring artifacts:** CM can be done (1) at the infrastructure level, e.g., to control the CPU or memory usage, or (2) at the application level, to monitor, for instance, the position and speed of an element. The goal of this subcategory is to collect the monitoring needs of both the infrastructure and the application. To gather requirements, an analysis of the application data life-cycle and the infrastructure features (e.g., CPU usage) shall be performed. Requirements in this subcategory have an impact on the design of the monitoring infrastructure. There are different aspects to be specified: (1) *Communication Network*, the source from which data must be collected, e.g., *"the monitor must gather the data from the CAN bus"*, (2) *Data fields* that are provided, e.g., *"the monitor must provide the elevator positions"*, (3) *Format*

in which the data is provided by the monitor, e.g., *"the monitor will provide the current elevator position periodically via MQTT"*, and (4) *Configurable parameters* for the monitor,e.g., *"the update period for the elevator positions may be configured with a value between 50 and 500 milliseconds"*.

- **Storage:** Storage of the monitored data is essential to analyze and reproduce scenarios in simulation. The storage strategy may be different depending on the data being monitored. Some data could be more critical and other may need more memory resources. These requirements might include, for example, dumping data on a local file, storing it on the edge of the network, or sending it to a cloud database. The goal of the Storage subcategory is to describe how the data shall be stored in order to be accessible from other services. To gather requirements, an analysis of the application data usage shall be performed. Requirements in this subcategory have an impact on the design of the storage strategy for the data that is being monitored. There are different aspects to be specified: (1) *Location* describes where the data is to be persisted, e.g., a shared folder on a NAS or a database endpoint, (2) *Type* relates to the database format, either a relational database, an object-oriented database or even text file based, (3) *Configuration* includes attributes such as duration of the saved data, backup replicas or even availability aspects), (4) *Resources* specifies the type of device used for persistence, as well as the disk space size.

2.1.3.3 **Validation**

Testing, verification, and validation activities are important in any kind of domain when developing software. In the case of CPSs, this is particularly important because most of the functionality of these systems is driven by software. Furthermore, this functionality is often safety or mission-critical, and a failure could lead to severe consequences. Many CPSs rely on simulation-based testing for the validation of software. This technique allows raising the level of abstraction of complex CPSs in which testing is performed [13]. It allows (1) executing larger test suites and (2) building test oracles that can automate verification and validation tasks [13]. Furthermore, simulation-based testing allows modeling the environment in which the CPS operates. Test, verification, and validation in Design-Operation Continuum methods for CPSs need to be practiced from MiL phases through the Operation. This is because failures that could not be observed in previous stages can be identified in Operation. To this end, oracles need to be re-used across all these test levels to allow full automation. Examples of requirements obtained by focusing on the industrial needs of the case studies in this category include *"The System Under Test (SUT) shall be the relevant version of the project-specific software", "The input to the test cases at the functional level shall be the stimuli triggering the execution of a defined functionality", "The oracles shall be*

activated by a test input or by identifying a precondition in operation". Note that the elicited requirements in this category shall provide the validation to be continuous and as automated as possible. To this end, three sub-categories were identified:

- **Simulation Artifact:** This category concerns the artifacts that are necessary in order to enable simulation-based testing, which are divided into three main categories: (1) *Environment* refers to the conditions under which the system runs, which are usually expressed in the form of simulator parameters (e.g., the number of floors in the building); (2) the *SUT* is the component of the system that is being tested, which must usually comply with certain interfaces in order to be usable for simulation-based testing; and (3) the *Tool* is the simulator used to execute the SUT. An example of a simulation artifact requirement for one of the industrial case studies is *"Test cases shall be executed by using the Elevate simulator"*.

- **Test Input:** In order to drive the execution of the selected test cases, test inputs must be injected into the SUTs before or during their execution. The requirements are divided for these test inputs into two main categories. (1) The *input data* itself, which is determined by the test cases that need to be executed (e.g., must test having multiple users at the same time), and (2) The *format* that is used to define the test inputs (e.g., test inputs must be provided in an XML file which follows a specific structure).

- **Oracle:** Test oracles are components in charge of emitting a verdict (e.g., PASS/FAIL) based on the conformance of the system towards a specified property. Note that although monitoring the system is required for the validation, monitoring is classified as a separate subsystem, since monitoring is often performed beyond the context of system validation. The purpose of the oracles is to determine whether the observed behavior of the system is correct or incorrect, which is usually done by verifying properties specified by a domain expert. An example of an elicited requirement for a test oracle is "test oracles shall be re-used across all levels (i.e., MiL, SiL, HiL, and Operation)", or "test oracles shall be capable of validating 100% of functional requirements". Four sub-categories were identified based on the industrial case studies. (1) *Validated properties* are requirements of the system themselves (e.g., Average Waiting Time (AWT) < 30 sec.); (2) *activation criteria* are pre-conditions that trigger a test oracle to validate a specific property; (3) *required data* refers to the monitoring data needed by the oracle in order this to be able to check certain property; (4) the *verdict format* refers to the semantics provided by the verdict (e.g., a quantitative value (e.g., from 0 to 1, with 1 meaning full compliance and the value becoming closer to 0 as the degree of compliance decreases).

2.2 Adeptness

As stated, with existing engineering practices for CPS, releasing and deploying new software versions is a time-consuming and error-prone activity. This is mainly due to the impossibility of thoroughly testing the software in a real environment. Furthermore, the deployment process itself is complex, as it is highly important to ensure that the CPS will be in a safe state when the software is updated. Besides, these systems often operate in dynamic and uncertain environments, which makes appropriate self-healing and recovery mechanisms necessary. These problems can be partially solved by implementing Design-Operation Continuum methods for the software development life-cycle, instead of relying on traditional software development methods (e.g., the V model). Nevertheless, to achieve this in the CPS domain, radically new solutions to overcome the limitations of today's CPS development processes need to be adopted.

As an alternative, in the context of the Adeptness H2020 project [1] a reference architecture based on microservices was proposed to enable Design-Operation Continuum activities in CPSs. By using this architecture, significant enhancements are expected in software development, reducing costs while increasing its quality. In this section, the main characteristics and benefits of microservice-based architectures are described, which the Adeptnes architecture leverages, and then the architecture itself is presented, which was designed based on the taxonomy presented in Section 2.1.3. The methodology developed in this thesis has been designed to take advantage of this architecture.

2.2.1 Microservice Architectures

The microservice architecture is an application development approach evolved from Service Oriented Architecture (SOA) which proposes structuring applications as a set of small independent services running in their own independent process and communicating with lightweight communication mechanisms [45]. Each microservice is responsible for specific functionality and may be written in different programming languages, can be deployed independently, and can be managed centrally, as opposed to the monolithic architecture, which pretends to build a big, united solution, as can be observed in Figure 2.4. These solutions are difficult to maintain when their size increases as they may get difficult to understand and modify, and do not easily scale, as making copies of the whole monolith is the only way to scale. These characteristics make continuous development and frequent software updates very difficult.

A microservice architecture offers a highly modular and decoupled architecture, which makes maintenance easier and leads to a higher flexibility and scalability of the system. Besides, the microservices can be deployed quickly to a production environment as they are independently testable and deployable [47].

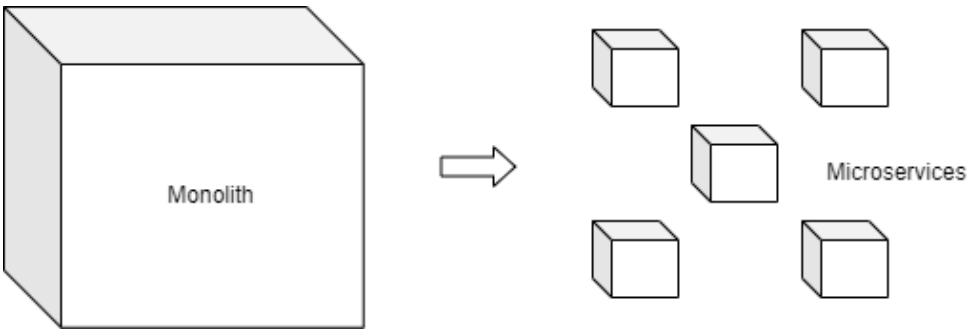


Figure 2.4: Difference between the monolith and microservices

The main characteristics of a microservice architecture are:

- Service isolation: Each microservice runs in its own process and has its own data store. This isolation ensures that failures in one microservice do not affect the entire application.
- Agility: Microservices are autonomous, which means they can be developed and deployed independently. This allows teams to work on different microservices simultaneously and deploy them independently, without interfering with each other.
- Scalability: Microservices can be scaled independently, which means that if increasing the capabilities of certain functionality of the application is necessary, more instances of the microservices involving this functionality can be deployed, without affecting the rest of the application.
- Flexibility: Microservices are designed to be modular, so that the architecture can adapt to specific use cases or the need for new functionalities by adding, removing or updating certain microservices.
- Improved performance: Microservices can be optimized for specific tasks, which can improve overall performance. For example, a microservice that handles image processing can be optimized for that specific task, resulting in faster processing times.

Microservice-based architectures have been mainly used in cloud/distributed application development but most recently started to be applied in edge platforms, mostly focused on the IoT domain providing services that connect CPSs to the cloud and adding analytic capabilities.

2.2.2 **Adeptness architecture**

In Figure 2.5 the general architecture of the Adeptness framework is presented. This architecture combines two paradigms: Cloud computing and Edge com-

puting. The architecture is divided into subsystems which can be composed by one or more microservices to provide all the necessary functionality. The microservices that compose a subsystem are distributed between both Cloud and Edge, depending on the goal of the microservice.

The main subsystems and the microservices are the following:

- **Automation server:** The Automation Server is in charge of the orchestration of the tasks to be performed by the different subsystems. It interacts with the source code repositories to monitor any changes in the deployment, monitoring, or validation plans. When a new plan is updated, the Automation Server performs the actions to generate the required artifacts, stores the generated Docker images in the Docker registry, and pushes the configurations or plans to each subsystem.
- **Deployment subsystem:** it is composed of the Deployment Orchestrator (Cloud) and the Deployment Agent (Edge). The orchestrator receives the deployment plan from the Automation Server and is responsible for deploying the necessary artifacts in each edge node.
- **Monitoring subsystem:** It is composed of the Monitoring Orchestrator (Clous) and the Monitoring Agent (Edge). It receives the monitoring plan from the Automation Server to configure the monitors so that the required data is obtained. The Monitoring Agents publish via MQTT the monitored data int the Logger so that the interaction with the subscribers is decoupled.
- **Validation subsystem:** composed by the Validation Orchestrator (Cloud) and the Validation Agent (Edge), the orchestrator receives the validation plan from the Automation Server to configure the Validation Agents. The oracles subscribe to the data they need to perform the validation, check the required conditions, and publish the verdicts of the tests in the Logger.
- **Logger subsystem:** The Logger subsystem receives both operational data coming from the Monitoring Agents and also validation results coming from the Validation Agents. Then it persists this information and provides services that are consumed by other microservices.
- **Test generation subsystem:** The test generation microservice generates test cases that are persisted in the Logger.
- **OSCL Bridge subsystem:** Composed of OSCL Tools and OSCL Bridge microservice. This subsystem oversees converting the test cases executing results to OSCL standard.
- **Recovery service:** Recovery microservice is in charge of proposing actions that can drive the recovery of the normal behavior of the system.

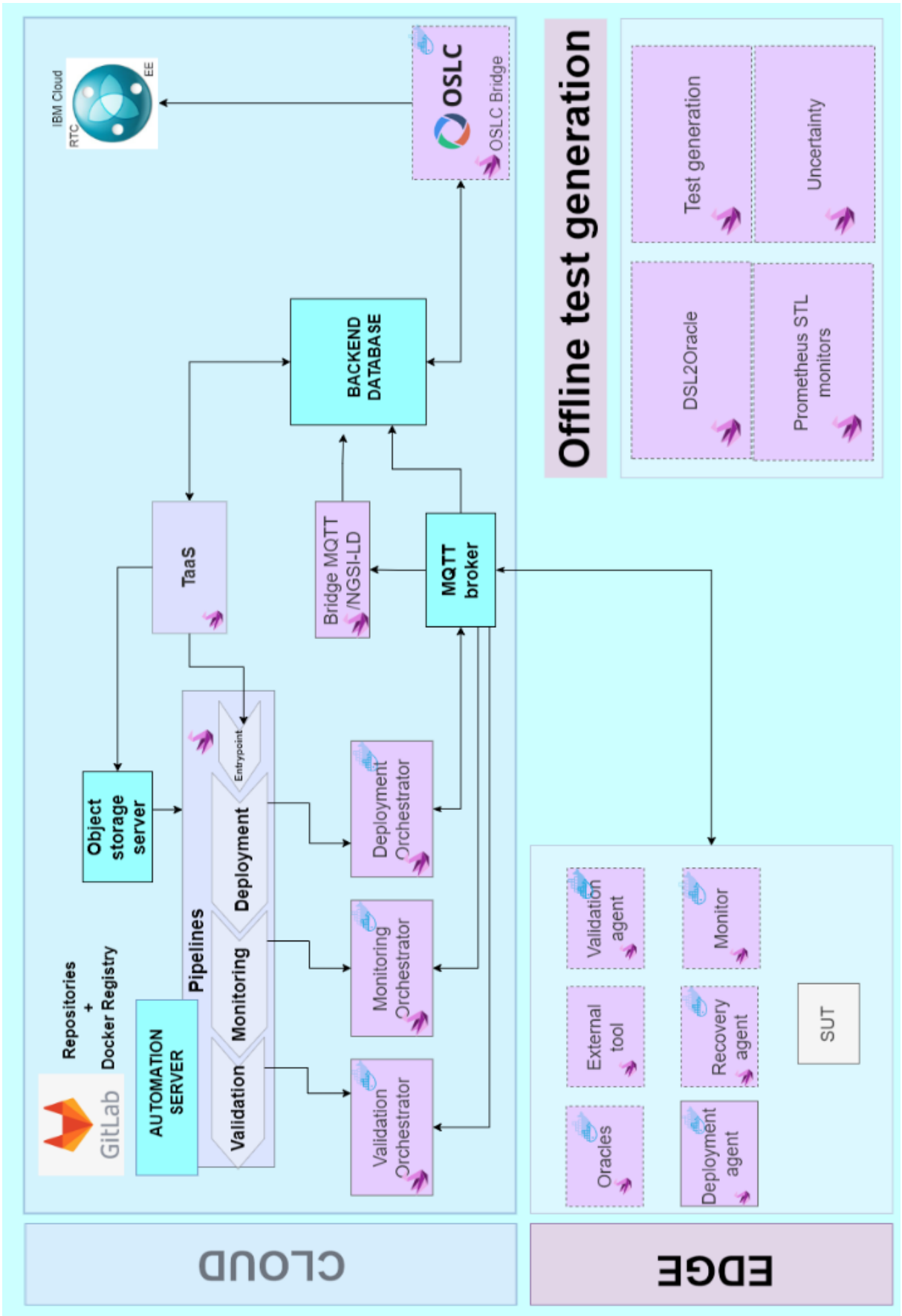


Figure 2.5: Overview of the Adeptness architecture components

As shown in the figure, the orchestrators, the microservices that coordinate and manage actions that take place on the devices are located in the Cloud. This is also the case of the Test Generator microservice, the Logger service, and the OSLC Bridge. The agent microservices, which perform actions on the devices, are located on the Edge, running on each device connected to the infrastructure. This is also the case of the uncertainty and recovery services. Now further detail on the core subsystems of the Adeptness architecture is given.

2.2.2.1 Deployment Subsystem

The Deployment Subsystem is responsible for downloading, and eventually decompressing and executing, the different microservices and artifacts needed to perform the validation in each of the targets or edge nodes. The Deployment subsystem executes a deployment plan and must be aware of the status of the deployment in each node. The plan contains the information regarding the components to be deployed, the repository where they are located in, and the node(s) where they should be deployed. The Deployment Subsystem is capable of deploying two different types of components, containerized microservices and generic files, and it is composed of two different microservices:

- **Deployment Orchestrator:** The Deployment Orchestrator receives the deployment plan from the Automation Server and parses the plan in order to execute it. There is only one instance of this microservice within the architecture and it is usually located in the Cloud. The orchestrator sends the deployment instructions to the Deployment Agents installed in each node by MQTT.
- **Deployment Agent:** The Deployment Agents must be installed in each edge node to perform the actual deployment of the necessary artifacts. Since there are two different types of components that may be deployed, two types of Deployment Agents have been designed: a docker-compose-based deployer to deploy docker containers, and a generic deployer to deploy any kind of file, e.g. an executable file, a library, or a .zip file. In the latter case, the deployer can perform the actual deployment of the zip file, decompress it, and execute the selected executable file.

2.2.2.2 Monitoring Subsystem

The monitoring subsystem supports the configuration of the monitors according to a monitoring plan. This plan specifies the source (i.e., physical interfaces, file system, ...) to obtain the data from as well as the value extraction mechanism. This subsystem provides access to telemetry data retrieved from different sources

so that other subsystems can subscribe to these data and use it to take decisions. This subsystem consists of two microservices:

- **Monitoring Orchestrator:** This microservice, deployed in the Cloud, handles the parsing of the monitoring plan sent from the Automation Server and configures all the Monitoring Agents indicated in the plan accordingly through their HTTP API. The plan specifies the parameters that each Monitoring Agent needs to specify the actual data source connection parameters (e.g., the CAN baud rate) and the variables to monitor.

- **Monitoring agents:** The Monitoring Agents, deployed at the edge nodes, are responsible for reading the operational variables from the different sources and publishing them asynchronously. The Monitoring Agents can be configured through a common HTTP API, which allows the specification of the variables to be monitored (i.e., name and needed parameters to obtain the data) and optionally, the configuration of subscriptions. The concept of subscription refers to a group of variables that are notified asynchronously as events, with the same publishing rate. In this sense, a service (e.g., an oracle) that needs to be notified about the changes of a set of variables can configure a subscription, specifying the publishing rate for those variables.

  There are specific Monitoring Agents for different data sources. Two examples of monitor agents are explained below:

  - *CAN monitor:* The CAN monitor allows the monitoring of the operational variables shared through a CAN field bus, which may be configured through an HTTP API. For each variable to be monitored, three parameters must be configured: (1) the name of the variable, (2) the identifier of the CAN frame where the variable is published, and (3) the mask to be applied to the frame to actually read the variable. When the monitor starts, it begins to publish the variables asynchronously through MQTT, following the standardized senML[1] payload format.
  - *Instrumented code monitor:* This is a special monitor type that supports the monitoring of variables that are not exported in any field bus but are needed by the oracles to raise a verdict. To do so, a library that publishes code variables through MQTT has been developed. The developer can use it to publish the internal code variables needed by the oracles into the MQTT broker, in the same senML format used by the rest of the monitors.

[1] https://tools.ietf.org/html/rfc8428

2.2.2.3 Validation Subsystem

The Validation Subsystem supports verification and validation activities at MiL, SiL, HiL, and Operation. The main microservices used for the validation subsystem are the following:

- **Validation Orchestrator** Located in the Cloud, the Validation Orchestrator manages the execution of a validation plan by communicating with the Validation Agents. A validation plan can require validations at different test levels.
- **Validation Agents:** These microservices launch validations at the SiL and HiL test environments, as well as in operational installations. For the execution of a validation, test oracles are activated and, in SiL and HiL environments, it also manages the additional tools required for the simulation. When an oracle provides a verdict, it notifies the Validation Orchestrator microservice.
- **Oracle microservice:** This microservice encompasses a set of test oracles that validate that the CPS behaves as expected. Many of these test oracles are based on domain-specific Quality-of-Service (QoS) measures that are collected from the monitoring microservices. Each of these test oracles provides a verdict that indicates to which extent the CPS behaves as expected. Different test oracles have been developed, such as those based on metamorphic relations for the SiL and HiL test levels [8], and some based on Machine Learning (ML).
- **Uncertainty detection microservice:** This microservice supports the automated detection of unforeseen situations in the different life-cycle stages of CPSs using data from both Operation and design time (e.g., test logs) with passive and active ML techniques. This service supports the validation microservice with uncertainty-related test oracles that are learned from data.
- **External tool:** This microservice allows launching domain-specific tools required to handle the execution of tests in SiL or HiL test phases.

2.3 Performance Bugs

Non-functional requirements refer to those which do not describe the functionality of the system, but the properties that the system must meet due to the environment where the system operates or other external constraints [16]. These requirements can include security, portability, safety, reliability, or performance of the system. In this thesis, the aim has been to detect performance bugs, which

consist of errors producing a behavior degradation of applications in terms of execution time, response time, CPU usage, memory usage, or energy consumption, without necessarily causing any fault on the expected results. With the growth of software complexity and resource-constrained applications, ensuring performance health is getting increasingly relevant. Identification of performance problems in the testing phase brings some limitations, such as the elicitation of proper performance requirements. In addition, many causes that lead to the manifestation of these problems may appear due to the system's interaction with the real environment [67]. At run-time, identifying performance problems may require a continuous monitoring of the execution, as problems may be revealed only under certain circumstances, for example, activation of specific modes or functionalities when the system has been running for a long period.

There exist a variety of root causes during implementation that can result in real-world performance problems. According to the classification conducted by Jin et al., [31], the authors identified (1) inefficient function calls, (2) skippable functions doing unnecessary work and (3) synchronization issues as main root causes for performance issues. Other less common causes are also mentioned, including wrong data structure usage, hardware architecture issues or high-level design errors. By analyzing how these bugs are introduced by developers, they concluded that these errors are usually introduced due to an API or workload misunderstanding.

From the testing perspective, testing performance requirements have been extended to many sub-fields, including Cloud Computing [37], distributed systems [18], or CPSs [62], among others. This practice is becoming paramount and the importance of integrating performance analysis inside the software life-cycle management process has proven to be relevant. In this sense, many challenges have been identified such as (1) the definition of a method to establish which performance tests to perform in each life-cycle stage, (2) enabling users to declaratively define performance objectives and (3) providing fast performance feedback to improve the system [20]. Regarding the test case design, some important aspects to consider are (1) the design of test generation and selection strategies and algorithms, (2) the definition of metrics to assess the effectiveness of performance testing strategies, and (3) the comparison of different hardware platforms for a given application [67].

2.4 AI Techniques

AI refers to the field that aims at providing intelligence to machines so that they become capable of performing tasks that typically require human intelligence. It involves the development of algorithms and systems that can process and analyze large amounts of data to recognize patterns so that they can then make decisions

in future problems. There exist many techniques to process the data to train an AI model so that it can extract useful knowledge from it. In this thesis three widespread techniques were investigated, such as ML, Neural Network (NN), and Genetic Programming (GP).

2.4.1 Machine Learning

ML is a sub-field of AI (AI) that provides machines with the ability to learn to do some job without being explicitly programmed to do so [41]. These techniques rely on learning from past experience or data to improve future results on the execution of a task in an automatic and autonomous manner. ML makes machines intelligent, giving them the ability to gain domain knowledge and make fast, data-driven decisions. There are several scenarios where ML can be particularly beneficial:

- Domains with a lack of human expertise.
- Dynamic environment scenarios (e.g., infrastructural changes, connectivity variations, etc.).
- Domains where rule-based programming gets extremely complex.

There exist many different learning algorithms that can be used in different contexts depending on the objective of each application [26]. The categories detailed below, which can be seen in Figure 2.6, are the main categories and the ones most used in IoT applications, which are supervised and unsupervised learning.

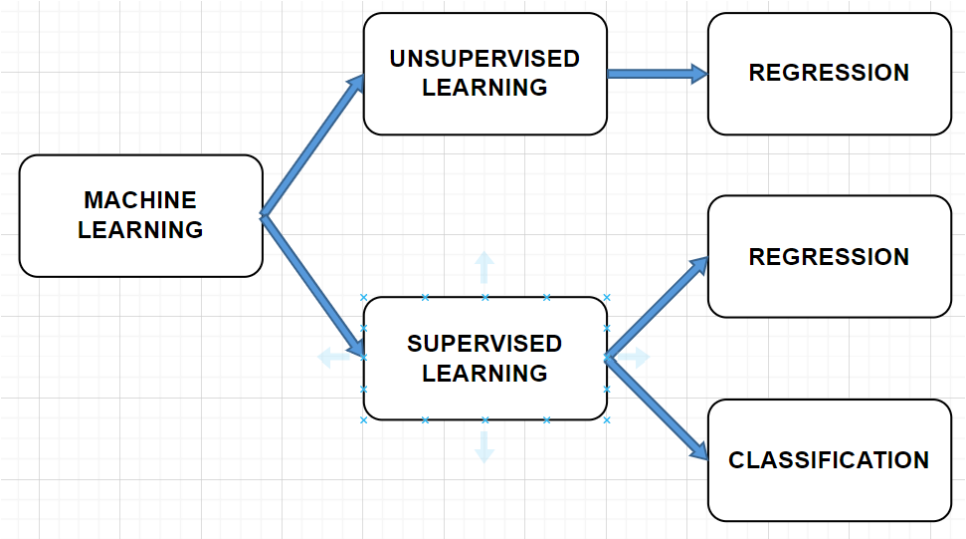


Figure 2.6: General overview of the main ML categories

2.4.1.1 Supervised Learning

In supervised learning, the training data consists of some input vectors with their respective labels, which indicate the output expected for each input vector. The objective of these algorithms is to learn to predict the corresponding output for the given inputs by modifying their internal values. Within this category, two subcategories can be distinguished: (1) classification algorithms, where the aim is to map input values to a finite number of categories, and (2) regression algorithms, where the output values consist of continuous variables. Some of the most common classification algorithms are K-Nearest Neighbours (kNN), Naïve Bayes, Random Forests, or Support Vector Machines (SVM), and regarding regression algorithms some extended algorithms are Linear Regression, Regression Trees or Support Vector Regression (SVR) [41].

2.4.1.2 Unsupervised learning

Unsupervised learning the objective consists of identifying similarities between the input samples to divide the samples into clusters, so the training data does not need any associated labels. Some common clustering algorithms are K-means or Density-Based Clustering.

2.4.2 Neural Networks

Neural networks are a type of ML algorithm that is designed to mimic the way the human brain works [2]. The basic building block of a neural network is the neuron. A neuron receives input signals from other neurons or from the environment and processes them to produce an output signal. In an artificial neural network, neurons are modeled as mathematical functions that take input values and produce an output value.

In Figure 2.7 an overall overview of the architecture of a neural network can be observed. A neural network consists of layers of neurons that are interconnected with each other. The first layer of the network is the input layer, which receives the initial data. The output layer produces the final result of the network. The layers in between the input and output layers are called hidden layers.

The process of training a neural network involves adjusting the parameters of the neurons and the connections between them so that the network produces accurate outputs for a given set of inputs. This process is called backpropagation, and it involves minimizing a cost function that measures the difference between the network's output and the desired output.

Neural networks permit recognition of more complex patterns than traditional ML algorithms, which make them useful in applications such as image recognition, speech recognition or natural language processing.

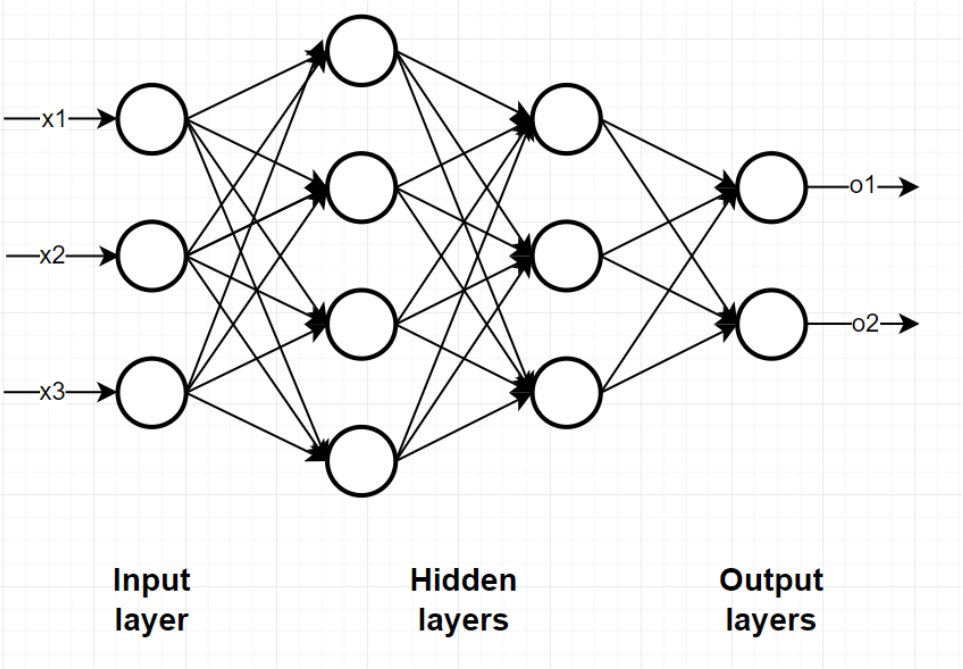


Figure 2.7: Example of an architecture of a Neural Network

There are many types of neural networks, each with its own architecture and purpose. These are some of the most commonly used types:

- Feedforward Neural Networks: These are the simplest and most common type of neural networks. They consist of layers of neurons that pass information forward from input to output, without any feedback loops. These are used for a wide range of applications, including image and speech recognition, classification tasks, and regression analysis.

- Convolutional Neural Networks (CNNs): These types of networks use a special type of layer called a convolutional layer which can identify patterns and features in images. They are primarily used for image recognition tasks, such as object detection, facial recognition, and image segmentation.

- Recurrent Neural Networks (RNNs): These networks have a feedback loop that allows them to use their own output as input for the next time step. These are used for tasks that involve sequential data, such as speech recognition, language translation, and time series analysis.

There are also many other types of neural networks, such as Long Short-Term Memory Networks (LSTMs), Autoencoders, and Generative Adversarial Networks (GANs), but the ones listed above are some of the most commonly used and well-known.

2.4.3 **Genetic Programming**

Genetic programming (GP) is a subfield of artificial intelligence that uses principles of evolution and genetics to automatically generate computer programs that can solve problems or perform tasks [4, 49]. GP works by generating an initial population of programs that are randomly created and evaluated to measure how well each program solves the problem or performs the task. Programs with higher scores are more likely to survive and reproduce, passing on their genetic material to the next generation of programs.

GP is useful in a wide range of applications, including data analysis, robotics, image recognition, and optimization problems.

The process of Genetic Programming (GP) involves the following steps:

- Initialization: The GP algorithm starts with an initial population of programs that are randomly generated. Each program in the population represents a potential solution to the problem or task at hand.
- Evaluation: The fitness of each program in the population is evaluated using a fitness function that measures how well the program solves the problem or performs the task. The fitness function can be based on a range of criteria, such as accuracy, speed, or efficiency.
- Selection: Programs with higher fitness scores are selected to move on to the next generation of programs. The selection process can be based on various selection methods, such as tournament selection or fitness proportionate selection.
- Genetic Operators: The selected programs are then subjected to genetic operators, such as crossover and mutation. Crossover involves swapping genetic material between two programs to create new programs, while mutation involves making small changes to the genetic material of a single program. The new programs created through these genetic operators form the next generation of programs.

This process is iterative, meaning that it involves the repeated application of these steps to generate and improve programs over multiple generations. The goal is to evolve programs that are efficient, effective, and optimized for the specific problem or task. The process continues for several generations until a stopping criterion is met. The stopping criterion can be based on various factors, such as a maximum number of generations or a target fitness score. Once the stopping criterion is met, the best program in the final generation is selected as the solution to the problem or task at hand.

3 State of the Art

In this chapter, the state-of-the-art on the topics related to this thesis are presented. As mentioned, the goal of this study is to solve current challenges in the deployment process of software in CPSs, focusing on the analysis of the performance of the system in the real operation environment. Thus, first the literature review methodology is introduced in Section 3.1, then, the current literature on software deployment in CPSs is analyzed in Section 3.2 and performance testing methods in CPSs in Section 3.3. Finally, the performance prediction approaches in the literature are analyzed in Section 3.4 and the state-of-the-art is critically analyzed in Section 3.5.

3.1 Literature Review Methodology

In this section, the methodology followed to develop the literature review on the aforementioned topics is explained.

3.1.1 Definition of Research Questions

To guide the search for relevant studies in the analyzed field it is necessary to identify clear and specific research questions to answer through the review. The Research Questions (RQs) established were the following:

3.1.1.1 Software Deployment

- Which tools permit modeling remote software deployment?
- Which aspects of the deployment process allow these tools to be modeled?
- Which are the current challenges of software deployment?

3.1.1.2 Performance Testing on CPSs

- Which performance metrics do the found works consider?
- Which techniques do they use to test system performance?

3.1.1.3 Performance Prediction

- Which metrics do the proposed approaches consider when predictiing performance?
- Which techniques and training strategies do these works use?

3.1.2 Search Process

To conduct a comprehensive search it is necessary to use a range of relevant databases, such as Scopus, Web of Science, and Google Scholar. The aim is to identify studies that address the RQs by using a combination of keywords and boolean operators to refine the search and identify relevant studies.

A set of search strings to be used on different scientific databases were defined with the aim of identifying conference proceedings and journal papers since 2015. The search engines used to find the papers were Scopus and Google Scholar. In the case of software deployment tools, besides published papers, commercial tools commonly used in industry were also studied. In addition, once the works considered relevant to this research were identified, some references of the selected studies and other studies citing these studies were also identified to find additional papers not emerged with the followed process.

The search string built to find the papers related to the topics of interest in this Ph.D. were the following:

3.1.2.1 Software Deployment

- (“Software” AND “Deployment”) AND (“Tool” OR “Challenges”)

3.1.2.2 Performance Testing on CPSs

- “Performance” AND (“testing” OR “Validation” OR “Verification”) AND “Cyber-Physical Systems”

3.1.2.3 Performance Prediction

- “Performance” AND “Prediction”

- "Environment" AND "Knowledge Extrapolation" AND "performance" AND "software"

3.1.3 Inclusion Criteria

After identifying the studies, it is necessary to screen them to determine if they meet the criteria established to include them. A first screen is done by reading the titles, a second screen is done by reading the abstracts, and finally, the full texts are read to determine their relevance.

3.1.3.1 General Criteria

- The publication should be written in English.
- The publication should be "journal", "conference proceeding" or "technical paper".
- The reader should clearly deduce by reading the abstract and the conclusions the topic of the paper.

3.1.3.2 Software Deployment

- The publication should present a novel tool to deploy software on IoT/Embedded devices.
- The publication should clearly detail the characteristics of the proposed tool.

3.1.3.3 Performance Testing on CPSs

- The approach is focused on CPS performance testing or evaluation.

3.1.3.4 Performance Prediction

- The work evaluates a method to develop a performance prediction model for CPS/Embedded systems.
- The study presents a novel technique to extrapolate knowledge through different environments.

3.1.4 Data Collection

Once a set of papers is included in the review, it is necessary to analyze the data from the studies and extract the needed information to answer the aforementioned RQs. Thus, it is possible to evaluate the quality of the studies by analyzing the methodology, sample size, and bias of the work to identify patterns, trends, and inconsistencies.

3.1.4.1 Software Deployment

- Full reference (authors, title, journal or conference and year).
- Institution or institutions belonged to the authors.
- Features provided of the proposed deployment tool.

3.1.4.2 Performance Testing on CPSs

- Full reference (authors, title, journal or conference and year).
- Institution or institutions belonged to the authors.
- Characteristics of the proposed performance testing technique.
- Case study (if proposed) or example (if available).

3.1.4.3 Performance Prediction

- Full reference (authors, title, journal or conference and year).
- Institution or institutions belonged to the authors.
- Techniques and training strategies used.
- Characteristics of the proposed knowledge extrapolation technique.

3.2 Software Deployment in CPSs

Software deployment is the moment in which a new software release is put into operation by loading it to its execution environment and running it. Automated deployment is beneficial for large distributed applications, such as multi-cloud platforms and Edge systems. However, CPSs lack platform homogeneity, making integration and automation of the development process complex. Furthermore, using commercial tools designed for different systems may not be viable in CPSoS due to safety and security standards.

On the one hand, there are a number of tools that permit automating the deployment of software such as CI/CD automation servers, which allow the integration of various tools and techniques. Software repositories, frameworks, build tools, testing environments, container-based virtualization, CI tools, domain-specific languages, or executable registry services are some of the tools that these automation servers allow to integrate. However, these capabilities are usually enabled by plugins or the implementation of specific code to take some actions. This requires developers to learn to use different tools or develop additional code, making the process complex. Among these tools, some of the more used are Jenkins[1], Circle CI[2], Travis CI[3] or Bamboo[4].

On the other hand, there are other tools that are more specifically meant for the deployment process on Edge platforms, which permit additional features for modeling deployment actions and offer greater usability for this purpose. In this category, there are widely used commercial tools and tools that are being newly developed by the research community. Table 3.1 shows a list of software deployment tools and the different aspects of deployment that can be modeled by each tool. Within the commercial tools, a set of widely used tools are listed, including XL Deploy[5], UrbanCode Deploy[6], Octopus[7], GoCD[8], AWS IoT Greengrass[9], Microsoft Azure IoT Hub[10], and Eclipse Hawkbit[11]. Within research tools, there are a variety of tools in scientific publications, but only the ones that provide enough details about their capabilities were included. These are GeneSIS [21], SMADA [50], URANO [57] and LE-DAnCE [51].

The aspects of interest to be modeled in software deployment have been extracted based on the taxonomy of Design-Operation Continuum requirements for CPSs presented in Section 2.1 and adding some other aspects of interest identified during the review of the characteristics of the different tools:

- Pre-conditions: checking some conditions are met before starting the deployment.
- Post-conditions: conditions to be verified after the deployment is completed to ensure the process succeeded.
- Features: hardware, software, and communication interface requirements or/and the allocation of the artifact.

[1] https://www.jenkins.io
[2] https://www.travis-ci.com
[3] https://www.travis-ci.com
[4] https://www.atlassian.com/software/bamboo
[5] https://legacydocs.xebialabs.com/xl-deploy
[6] https://www.ibm.com/cloud/urbancode/deploy
[7] https://octopus.com
[8] https://www.gocd.org
[9] https://aws.amazon.com/es/greengrass
[10] https://azure.microsoft.com/es-es/products/iot-hub
[11] https://www.eclipse.org/hawkbit

- Rollback: establishment of a rollback policy in case of deployment failure.
- Availability: Availability of the device during deployment (e.g., maximum downtime of the device to perform the deployment).
- Resources: Hardware, software (e.g., installed OS) and communication networks (e.g., CAN, Ethernet) that the deployment subsystem is going to deal with.
- Repository: requirements for the artifact storage system.
- Orchestration: the ability to automate the process.
- Configuration: configuration files to determine the behavior of configurable artifacts.
- Dependencies: establishment of dependency relationships between artifacts.
- Monitoring: specifying the metrics to be monitored during and after deployment.

Table 3.1 shows that the main gaps of the current deployment tools are the verification of the state of the infrastructure before the beginning of the deployment process (i.e., PRE-CONDITIONS), the establishment of requirements for the availability of the system during deployment (i.e., AVAILABILITY) and the validation of the system after the deployment process finishes (i.e., POST-CONDITIONS). In this sense, SMADA does have the capacity to establish some requirements to trigger changes in the system but does not focus on error detection but on adopting different deployment strategies to avoid bottlenecks. Therefore, it can be observed that current deployment tools do not focus on performance error detection after a software is deployed.

Other tools have addressed the challenge of detecting errors on the deployment process, mostly for Cloud environments.

Gandalf [38] is a service whose objective is to detect errors on cloud rollouts to stop them before they cause major failures. To this end, it continuously monitors a wide range of infrastructure data (performance data, failure signals, and update events) to detect anomalies. It uses time-series analysis techniques to predict expected performance and if a certain threshold is exceeded it is considered an anomaly. When any anomaly is detected it determines if the anomaly is caused by a rollout or not by means of correlations. Finally, it uses a Gaussian discriminant classifier to decide whether the impact caused by the deployment is significant enough to stop it.

FUNNEL [69] is a tool that collects performance metrics for each software change (i.e. software update or configuration change) to detect behavior changes.

Table 3.1: Overview of the characteristics of the available deployment tools

		PRE-CONDITIONS	POST-CONDITIONS	FEATURES	ROLLBACK	AVAILABILITY	RESOURCES	REPOSITORY	ORCHESTRATION	CONFIGURATION	DEPENDENCIES	MONITORING
Commercial	XL Deploy				X		X		X	X		
	UrbanCode				X			X				
	Octopus							X		X		
	GoCD							X	X			
	Hawkbit			X					X			
	Microsoft Azure			X					X	X		X
	AWS IoT Greengrass			X					X	X		X
Research	LE-DAnCE			X			X	X	X	X		X
	URANO			X			X			X		
	GeneSIS											X
	SMADA		X	X	X		X	X		X		X

It uses a Singular Spectrum Transform (SST) algorithm for the detection and a Difference-in-Difference (DiD) method, where it compares the relative performance of the treated group and a control group, to decide if the software change is the cause of the behavior change.

Liao et al. [39] proposed a method to detect performance errors of a software based on learning from its previous version. It uses different ML techniques and Neural Networks to build a model able to predict the CPU usage of the system based on its workload, and in case a certain threshold is exceeded, the version is considered faulty. The authors conclude that the traditional ML models outperform the deep neural networks (e.g., CNN and RNN) for modeling the performance of the studied systems and that the model is able to accurately predict the performance of the system under new unseen workloads.

SCWarn [71] is a system that focuses on using multi-modal anomaly detection for heterogeneous multi-source data. First, heterogeneous data is transformed into unified time series using log parsing. Then, anomalies are detected using a multi-modal LSTM model, capturing temporal dependencies and inter-correlations. When an anomaly score breaches the alerting rule, engineers are

notified with an analysis report. The action decision component identifies suspicious changes, allowing engineers to take proactive actions, such as rollbacks, to prevent service outages.

3.3 Performance Testing on CPSs

Performance testing consists of validating the behavior of the system in terms of performance metrics, such as energy consumption, response time or resource utilization. Performance testing has been more commonly used in cloud-based applications as they usually gather a large number of services and users, which contributes to the appearance of bottlenecks [40]. This field is already covered by commercial tools, such as NewRelic [12], AppDynamics [13] and Dynatrace [14]. However, as embedded applications increase in complexity resource-saving gets more important, and the performance of these applications gains attention among researchers.

A systematic literature review conducted by Muccini et al. on Self-Adaptive CPSs in 2016 [44] showed that the efficiency/performance of CPSs was the biggest concern to adapt a CPS. More recent reviews like the one performed by Oudina et al. on Cyber-Physical Production Systems Testing [48] show that lately the focus on CPS testing has shifted towards cyber-security [5, 54, 66], but performance remains one of the main concerns.

Performance evaluation is considered across all the life-cycle stages in Cyber-Physical Systems, from development to operation.

In the software development phase, early feedback and guidance on implementation decisions relevant to performance is valuable for developers to take design actions or to avoid errors to propagate.

Balasubramaniyan et al. [11] proposed a methodology to design and verify CPSs with the aim of optimizing their reliability and performance. They used an evolutionary algorithm to search the optimal configuration of the controller to enhance performance while maintaining the jitter within certain margin. The aim is to design the CPS to guarantee compliance with performance requirements in the presence of timing delays induced due to CPS components. The controller parameters are afterward validated by simulating the CPS for a given scheduling policy and network protocol. The conditions that fail can be then used to modify the design. After applying the proposed methodology in an industrial mine pump example, they conclude that the CPS design is a trade-off between jitter margin and achievable performance, and the best trade-off was obtained by modeling the design problem as a multi-objective optimization problem.

[12] https://newrelic.com
[13] https://www.appdynamics.com
[14] https://www.dynatrace.es

Song et al. [62] proposed a virtual testing environment to assess the performance of CPSs, which can simulate several configurations in parallel and identify the ones that lead to the best and worst performance. The authors make use of model-order reduction and distributed simulations along with the Particle Swarm Optimization method to find the optimal configurations to improve domain-specific Key Performance Indicators (KPIs) in a large configuration space in an efficient manner.

As stated in Section 2.3, performance bugs are difficult to detect in lab. Thus, once a CPS is running in the production environment, it is necessary to continue monitoring and validating its performance.

Reif et al. [55] proposed Δelta, a tool to analyze and correlate the runtime performance and energy demand of individual Cyber-Physical System components and provide useful information for the design, implementation, and evaluation of CPS networks. It can extract the impact of specific code or hardware configurations in the end-to-end latency and analyze the relation between energy demand and performance.

Markoska et al. [42] proposed a performance testing framework oriented to smart buildings, which is offered as a service in the cloud and considers performance metrics such as power consumption or timing constraints. The aim is to test the performance of the system once it is deployed, as performance may vary from expected due to environmental changes or software degradation. Offering the service in the cloud allows many systems to have access to the service so that they can subscribe to the services (tests) they may need and sending their instrumented metrics as input data for those tests. In a more recent work on the same topic, Jradi et al. presented ObepME [32], a method to detect performance degradations in smart buildings, which leverages a commercial tool called EnergyPlus[15] which provides building modeling capabilities to predict the expected energy consumption of the different elements in the building and compare it with the actual consumption.

3.4 Performance Prediction

Performance prediction consists of predicting the performance of a system in different environmental conditions to obtain a reference value of the expected performance of the system. This is an issue in CPSs due to their high configurability and the heterogeneity of their environments.

Zhang et al. [70] proposed a methodology to estimate the performance of embedded software for RISC-V processors using NNs. The authors propose generating and executing a set of softwares with different features to train a

[15]https://energyplus.net

performance model using ANN. The training data consists of the number of each type of processor instructions as features and the number of processor cycles as the label.

Ye et al. [68] used deep learning models to predict the energy consumption of a building using occupancy data collected from sensors. They built a hardware-based emulation platform to emulate the behavior of the building and extract the occupancy and energy consumption data to train an enhanced LSTM neural network.

As mentioned, CPSs operate in very heterogeneous systems, which prevents testing the system under every possible situation. Thus, it is necessary to find ways to predict the performance of the system in unseen environments. The following works focus on predicting the performance of software extrapolating knowledge from some environments to others.

Javidian et al. [30] performed different correlations between performance measurements made in different environments to detect which environmental values were linearly transferable. Jamshidi et al. [29] also proposed predicting software performance based on the learning made from a sample of configurations, but they include a method to learn which configuration options are most relevant to performance so that the learning process is more efficient.

Duttagupta et al. [19] propose a method to predict the performance of an application with high workloads based on the learning performed with small workloads. The proposed approach is able to predict the response time, throughput, and resource usage of applications as well as the bottlenecks or maximum load level affordable by the system. The system can extrapolate the throughput of applications until 6000 users from the throughput and resource usage for 50-400 users. The system uses linear regression and S-curve techniques and is evaluated with different configurations and workload scenarios.

3.5 Critical analysis of the State of the Art

In this section, the current state of the art in the presented topics is critically analyzed, with the aim of detecting potential research opportunities.

Regarding software deployment tools, in Table 3.1 was shown that current commercial deployment tools lack post-deployment monitoring and validation capabilities. Instead, newly developed research tools consider these features necessary and include some monitoring capabilities. However, only one tool permits taking some post-deployment actions based on these data, and the validation techniques are limited.

Besides, some works that aim at detecting performance errors after software deployments were mentioned, which are meant for cloud environments. The

mentioned Gandalf [38] and FUNNEL [69], are tools that need multiple components to be deployed in multiple nodes to correlate their metrics and decide whether the deployment is the cause of the anomalous behavior. This may not be feasible for CPSs, as the environment (e.g., workload and configuration) in which these CPSs are deployed are very heterogeneous, so performance metrics of different CPSs may differ significantly, so it may not be possible to correlate their data.

The work presented by Liao et al. [39] proposes building a performance model from previous version data to establish a baseline on the expected CPU consumption of the new version to decide whether the new version works correctly or not. This work infers the workload of the studied systems based on the generated logs. This approach is not feasible in a CPS context where each system may be running under different configurations or environmental conditions, which are not extracted from log files, and are modifying the performance behavior of the system. Furthermore, it does not make any considerations on the resource utilization of the generated models, which is paramount in this domain.

The last work in this category, developed by Zhao et al. [71] uses an unsupervised learning approach to model relations between multi-source data (e.g., performance, application logs, etc.). To relate different environmental conditions to the performance metrics a supervised approach may be necessary in this context.

Regarding performance testing in CPSs the works by Balasubramaniyan et al. [11] and Song et al. [62] were more focused on optimizing the configuration of the CPS in terms of performance in the design phase rather than detecting performance bugs. The works which focused on performance bug detection in operation, the works by Markoska et al. [42] and Jradi et al. [32] propose a performance testing approach to test the energy consumption of smart buildings. However, in the former, the thresholds to consider the behavior of the consumption of the building abnormal were established on a regulatory basis, not in an analysis of the expected performance of the building. On the latter, they used a tool to calculate the expected performance of the building but they do not give insights of the accuracy of this prediction to known data, and they use an arbitrarily selected threshold of 20% to consider an abnormal behavior.

Finally, in the field of performance prediction different works aimed at building performance models to obtain the expected performance of embedded/Cyber-Physical systems. Zhang et al. proposed predicting software performance based on the processor instructions generated for each software. This would imply constantly monitoring the processor instructions executed by the CPS, which implies a complex monitoring system for a remote device with limited resources.

The works by Jamshidi et al. [29] and Javidian et al. [30] focus on detecting which configuration options may be relevant to extrapolate performance from

one environment to another, rather than investigating different techniques to build the most accurate performance model. In this thesis, the collaboration of an industrial partner is available, who can give us detailed information about the configurations affecting the performance of the software. Thus, the focus can be on the research of different techniques to build the model.

Duttagupta et al. [19] proposed predicting the performance of software by extrapolating knowledge from low workload measurements to high workload contexts. This may not be useful in highly configurable contexts, where configuration changes or certain working modes may be activated when a high workload is detected, changing the behavior of the software.

In summary, there is a need for integrating post-deployment validation capabilities in deployment tools, specifically for performance validation, which is more widely considered for cloud applications than for CPS/embedded domains. When testing performance in CPSs, the main challenge is to precisely establish the expected performance of the system in heterogeneous environments to detect when a fault occurs. Many cited works do not consider the possibility of running the software in heterogeneous environments, which would be a limitation in this case. The works which consider extrapolating knowledge among environments, investigate ways to detect configurations affecting performance, but do not research on different techniques to build performance models and do not consider resource utilization considerations, which is necessary for a resource-constrained domain such as CPS.

4 Theoretical Framework

In this chapter, a general overview of the contribution of this thesis is provided. First, the research objectives of the work are defined in Section 4.1 as well as the hypotheses to be confirmed in Section 4.2. Then, an overall overview of the theoretical framework proposed to fulfill these objectives and hypotheses is given in Section 4.3, and finally, the case study that was used to validate the effectiveness of the proposed solution is explained in Section 4.4.

4.1 Objectives

This study aims at detecting performance issues in CPSs after a software is deployed. To do so, the aim is to obtain the expected performance of the system and compare this reference value with the actual performance of the system, so that it can decide if the system is working correctly or not. This deployment and validation mechanism was developed following a microservice architecture to ease the deployment of all the needed components and make the method flexible and scalable.

Within this global objective, the following sub-objectives were considered:

- ***Objective 1:*** Development of a performance prediction model able to predict software performance for multiple environments.
- ***Objective 2:*** Development of a performance prediction model with low resource consumption, so that its use is suitable in embedded systems with constrained resources.
- ***Objective 3:*** Development of an Arbiter able to detect performance bugs based on predicted and actual performance capable of detecting different types of errors.

- ***Objective 4:*** Integration of the Performance Oracle in a microservice-based CD mechanism to deploy the software and necessary tools to perform its post-deployment validation.

4.2 Hypotheses

The hypotheses to be confirmed during this study are the following:

- ***Hypothesis 1:*** It is possible to accurately predict the performance of a CPS in certain environment based on knowledge obtained from the execution of the previous release in the same environment.
- ***Hypothesis 2:*** It is possible to accurately predict the performance of a CPS on a certain environment based on knowledge obtained from the execution of the software on other environments.
- ***Hypothesis 3:*** Data obtained from the monitoring of real environments is more appropriate to train performance models than laboratory data.
- ***Hypothesis 4:*** It is possible to create a high-accuracy performance prediction model with low resource consumption.
- ***Hypothesis 5:*** It is possible to detect different types of performance bugs in a software based on different metrics.

4.3 Overview

Figure 4.1 shows a general overview of the methodology for detecting performance bugs on software updates in CPSs. The following phases summarize the whole process:

- **Monitoring:** The aim of the monitoring phase is to obtain the necessary data to train the Performance Model. This phase consists of two steps:
  - ***Input data:*** In this step the input data that is used to execute the studied software is obtained. In this Ph.D. the use of two types of data is considered: theoretical data and real data. To obtain the former, a domain-specific simulation tool is used, which generates simulated data to execute the software. For the latter, it is necessary to monitor a real operation environment and extract the data from its real execution.
  - ***Performance metrics:*** Once the input data for the software is available, in the performance metrics monitoring step the software is executed

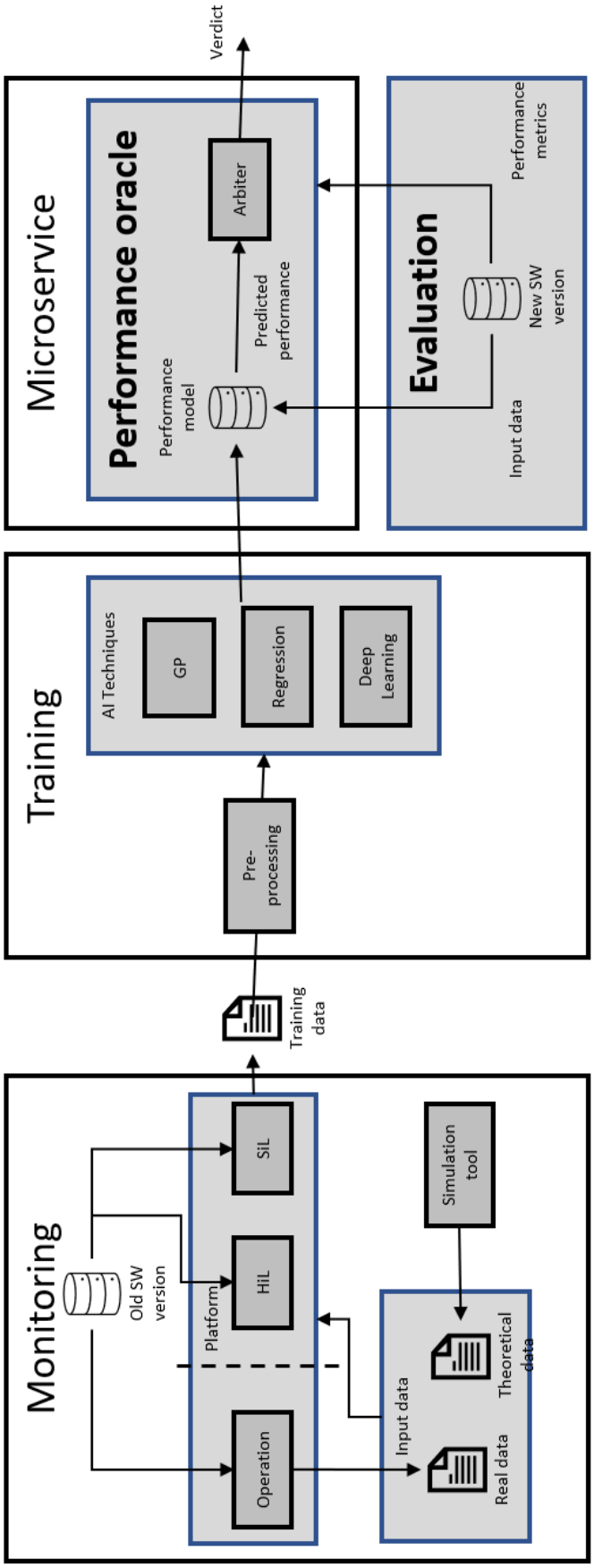


Figure 4.1: Overview of the method developed in the Ph.D.

with these data in different platforms to obtain its performance metrics. This study considers the execution time of the software as the performance metric to study, and also considers executing the software on two different platforms: SiL and HiL platform.

- **Training:** This phase aims at creating a Performance Model by means of the data obtained at the monitoring phase. This phase includes the following steps:
  - ***Pre-processing:*** In this step, the input data is prepared to be used for training the Performance Model. This includes transforming the data to obtain a set of features to train the model, cleaning the data to remove outliers, splitting the data into a training and testing set, or selecting the most relevant features.
  - ***AI techniques:*** In this step, the different AI techniques to build the most suitable Performance Model are investigated. These techniques include ML, NN, and GP. These techniques were used with different training data obtained from the monitoring phase (i.e., theoretical/real, SiL/HiL) to analyze the impact of the different techniques and data on the performance of the model.
- **Oracle microservice:** The Performance Oracle is built in the form of a microservice, composed of two components:
  - ***Performance Model:*** The Performance Model is built in the training phase and predicts the performance of the new software version based on its input data. It is built so that it can predict the performance of the software in different environments and considering its resource utilization, as it will be executed in a resource-constrained environment.
  - ***Arbiter:*** The Arbiter is the component that compares the prediction of the Performance Model to the actual performance of the new software version to decide if there is a performance error or not. It considers different metrics and applies different thresholds to detect different error types.

  The Performance Oracle is evaluated by executing faulty versions of the software and measuring its correct and incorrect verdicts.

4.4 Case Study: Orona's Dispatching Algorithm

The case study used to evaluate the proposed method has been the dispatching algorithm of Orona. Orona is a company dedicated to the design, manufacturing, installation, and maintenance of elevators, escalators, and moving ramps.

Elevators are complex CPSs composed of different subsystems that collaborate to transport passengers vertically in a building. In Figure 4.2 an elevator installation in a building can be seen.

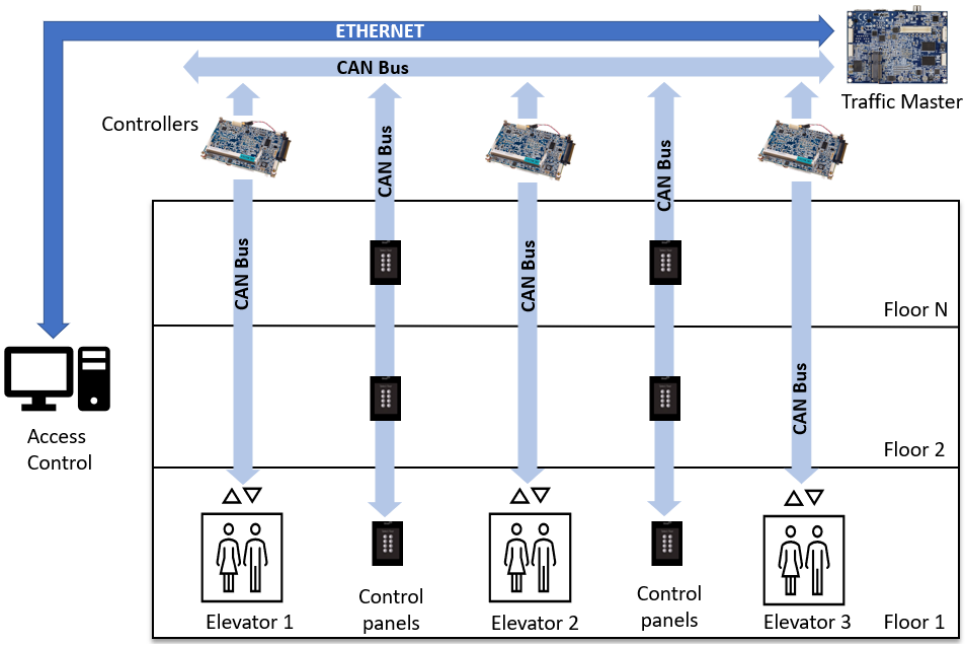


Figure 4.2: Overview of the architecture of an elevator installation from Orona

Controllers are in charge of managing both the vertical (from floor to floor) and the horizontal movements (doors opening and closing) of a single elevator. The traffic master is the software system in charge of the coordination of the controllers to serve the floor calls requested by passengers. The main responsibilities of the traffic master include the execution of the dispatching algorithm (i.e., the allocation of passenger calls to any of the available cars) and the overall system signaling (registration of the calls, information to the passenger), but it can also carry out additional functionalities such as access control or management of special operating modes.

The traffic dispatching algorithm is the software component that selects the optimal elevator to serve a specific landing call. Thus, this component is critical to ensure the QoS of the elevator installation because of many reasons. On the one hand, the assignment of the landing calls has a direct impact on the AWT and overall Journey Time (JT) of the passengers. On the other hand, it affects to the energy consumption or transport capacity of the elevators.

The traffic algorithm constantly evolves to be adapted to particular installations, improving the assignment process by including new rules or using new techniques such as artificial intelligence. Additionally, new criteria for the assignment such as the number of stops, load balancing or energy consumption are usually required for some installations.

The traffic dispatching algorithm is executed periodically to allocate all the active floor calls. Depending on the algorithm, already existing floor call allocations can be reallocated (to a different car) to optimize the overall cost function. This means that the allocation process is highly dynamic as far as it depends on the current system context. This fact, alongside other context situations such as highly demanding traffic profile or a large number of floors, can result in a high computing resource consumption on the allocation of landing calls in time. In this sense, the traffic algorithm should allocate the complete set of active landing calls in a limited time frame (execution time) to provide an acceptable QoS. Moreover, the traffic algorithm is executed within a task that shares computing resources with other tasks that provide the above-mentioned functionalities (signaling, access control, etc.).

Thus, system performance monitoring is crucial to ensure a proper system QoS. If such performance decreases, the overall system QoS degrades, and additional hardware resources should be considered. A special type of traffic algorithms in which performance is especially relevant is destination algorithms. Unlike conventional algorithms, in which many passengers share the same up-landing or down-landing call, in destination algorithms each passenger registers his own call. This "destination call" is composed of a landing call and a destination floor, and more importantly, it remains active until the destination is reached. Therefore, a destination algorithm requires managing each destination call individually. Additionally, this type of dispatcher provides extra functionalities such as access control, that could affect the execution time of the algorithm. Thus, in high-population buildings, the amount of active calls can be huge and can severely affect the system performance.

When a new version of the algorithm is released, performance under different traffic conditions and installations must be checked to identify issues related to poor implementation of new or improved functionalities. Usually, benchmark installations with different numbers of floors and elevators, and some theoretical passenger profiles are used to validate the new release. These profiles represent some traffic demands for different types of buildings (offices, residential, hotels, etc.) during different periods of the day. Most commonly used office profiles include (1) Morning UpPeak, representing the entrance to the office in the morning (2) LunchPeak, representing lunchtime, where there is a mix of passengers entering and exiting the building and (3) DownPeak, representing the exit of the office in the afternoon. There are also full-day profiles that represent the traffic in an office during a working day and therefore, include a morning UpPeak, LunchPeak at midday and Afternoon DownPeak complemented with some inter-floor traffic (passenger flow between different floors of the building) during the morning and the afternoon.

However, detecting performance problems that could compromise execution

time in a new release of the algorithm is a complex task due to the following factors: (1) some functionalities of the algorithm are only activated under certain traffic demands, keeping potential performance issues hidden. For example, parking of empty elevators to heavy floors is only activated when a big demand from that floor is given. (2) Performance is highly dependent on installation-specific factors: number of controllers, number of floors in the building, etc. (3) Performance is highly dependent on the passenger flow of each building. To summarize, poor performance can be exhibited in some installations or traffic conditions but not in others, which makes it difficult to validate the software system. Reproducing all the real scenarios to analyze the potential performance issues in the laboratory is unfeasible due to the effort it would mean. Moreover, there is often a lack of actual traffic profile information and thus, it is not easy to reproduce the operation conditions of the final installation in the laboratory.

Therefore, it is necessary to include monitoring and detection mechanisms in operation to facilitate the detection of potential performance problems in a particular installation. This way, when releasing a new version, potential performance issues can be detected automatically before they compromise execution time, and eventually, a rollback to a previous version could be performed.

5

The Performance Oracle

In this chapter, the Performance Oracle developed in this Ph.D. is introduced. A test oracle is a mechanism to check whether a certain software results in correct behavior or not by validating a set of conditions against certain inputs to raise verdicts that determine if the defined conditions are being met.

The oracle developed in this thesis is a performance oracle, which is an oracle focused on validating the behavior of a system in terms of performance, which may include degradation in execution time, CPU usage, memory usage or energy consumption. Specifically in this thesis, the Performance Oracle focuses on detecting deviations in the execution time of a software.

With the growth of software complexity and resource-constrained applications, ensuring performance health is getting increasingly relevant. Beyond the implications that performance errors may have in other less critical domains (e.g. web pages), in CPSs these errors may lead to a miss of deadlines in critical tasks that may be harmful to human lives. Besides, the identification of performance errors brings some additional limitations when compared to functional error detection. On the one hand, it is difficult to establish proper performance requirements, as determining the correct behavior of a software in terms of performance for some given conditions is not obvious. On the other hand, performance issues may be revealed only under very specific circumstances. This represents a big problem in the domain of CPSs because of their high configurability and the heterogeneity of the environments where they operate, which makes it unfeasible to test every possible situation in the lab.

As a response to these challenges, in this Ph.D., a Performance Oracle based on AI techniques was developed focused on detecting performance errors in CPSs. Different AI techniques have been studied to predict the expected execution time of a software to use it as a baseline to be compared with the actual execution time and look for deviations to give a verdict on the performance of the software. This oracle is then constructed as a microservice to be easily deployed in any

environment.

In Section 5.1 its internal sub-components and communication interfaces are detailed. In Section 5.2 its relationship with other components in the Adeptness architecture is described. The requirements it must fulfill to be usable in CPSs are listed in Section 5.3, and in Section 5.4 how these requirements have been validated is explained. Finally, in Section 5.5 the chapter is concluded by summarizing its content.

5.1 The Oracle as a Microservice

The HORIZON2020 Adeptness project [1] has proposed a microservice-based architecture that allows DevOps practices to be adopted in the context of CPSs. Microservices permit building a flexible architecture where services can be reused in different life-cycle stages and hardware, seamlessly deploying new services to all the installations and scaling the system.

Each microservice within the proposed architecture shall be responsible for a specific well-defined function in the life-cycle of a new software release and shall provide different lightweight communication mechanisms.

Thus, in this section, the sub-components composing the Performance Oracle are depicted as well as the interfaces it needs to interact with other microservices on the Adeptness architecture.

5.1.1 Interfaces

Each microservice provides both synchronous (i.e., HTTP) and asynchronous (i.e., MQTT) communication, offering a set of interfaces to interact with the rest of the microservices. These interfaces include (1) General Interfaces, which are common for every microservice within the system and permit configuring and checking the status of the oracle as a generic component, and (2) Oracle Specific Interfaces, which permit checking and configuring oracle-specific parameters. In Figure 5.1 an overview of the interfaces of the oracle developed in the Adeptness project is provided.

5.1.1.1 General Interfaces

All microservices within the architecture provide a set of basic asynchronous and synchronous communication endpoints, regardless of the role of the microservice. These endpoints offer basic information about the execution status, health, and performance. The synchronous interfaces allow other services to request microservices' health status, while asynchronous interfaces allow microservices

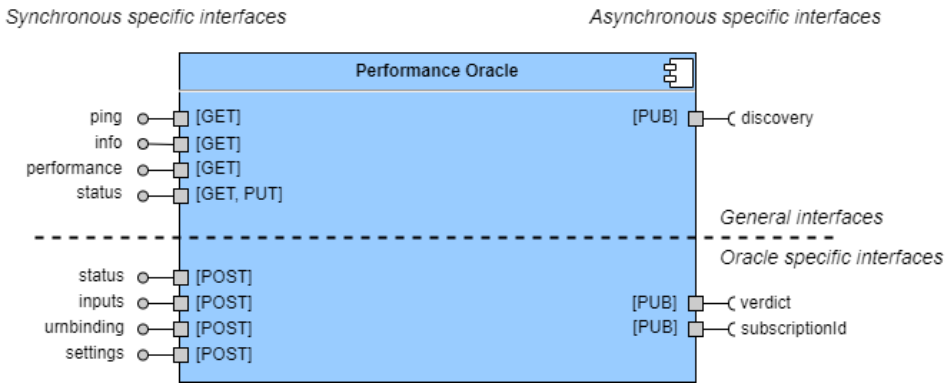


Figure 5.1: Synchronous and asynchronous interfaces of the Performance Oracle

to publish relevant data without knowing the receiver of the messages. The following interfaces are provided by the template developed within the HORIZON2020 Adeptness project [1].

5.1.1.1.1 Synchronous communication

- /adms/v2/ping [GET]: Ping service to check that the service is alive. Returns an empty 200 response if the microservice is working correctly.
- /adms/v2/info [GET]: Provides basic information about the microservice. It returns a JSON object containing the microservice ID and microservice role within the architecture.
- /adms/v2/performance [GET]: Provides CPU and memory usage metrics. It returns a JSON object containing the free and allocated memory and the CPU usage.
- /adms/v2/status [GET, PUT]: Permits getting or changing the execution status of the microservice. GET calls to this endpoint will return a JSON object containing the status of the microservice. Changes to the microservice status are performed by sending a JSON object with the desired state. The possible states for the microservice are "Ready" and "Running".

5.1.1.1.2 Asynchronous communication

- /adms/v2/discovery [PUB]: On microservice launch, the microservice publishes a hello message in this topic including the identifier, microservice role, and its MQTT and REST endpoints, defined as a JSON object.

5.1.1.2 Oracle Specific Interfaces

Other interfaces are specific to the Oracle microservices, which permits them to interact with the rest of the microservices to validate a software.

5.1.1.2.1 Synchronous communication

- /adms/v2/oracle/status [POST]: Get or change current oracle execution status. As in previous status messages, a JSON object containing the current or desired execution status is used.
- /adms/v2/oracle/inputs [POST]: Manage data inputs for the oracle. The inputs contain the subscription topic, input index, data type and monitoring rate, described as a JSON object.
- /adms/v2/oracle/urnbinding [POST]: Bind an evaluation function to the Oracle URN at runtime
- /adms/v2/oracle/settings [POST]: Send custom settings to the oracle

5.1.1.2.2 Asynchronous communication

- adms/v2/oracle/OracleGroupURN/OracleURN [PUB]: Raise verdict from the oracle conditions. A JSON object containing the verdict and its confidence is published. Additionally, it includes the evaluation start and stop times.
- /adms/v2/monitoring-agent/monitorId/subscriptionId: [SUB]: Get values from monitor subscriptions.

5.1.2 Sub-components

This oracle consists of two main elements: (1) the Performance Model, which yields the expected performance for the system, and (2) the Arbiter, which compares the performance obtained by the model with the actual performance to raise a verdict. In Figure 5.2 an overview of the components of the oracle and their relation are depicted.

5.1.2.1 Performance Model

The Performance Model is the component in charge of receiving the input data of the studied software and uses these data to make a prediction on the expected performance of the software in those conditions. This model can be obtained by

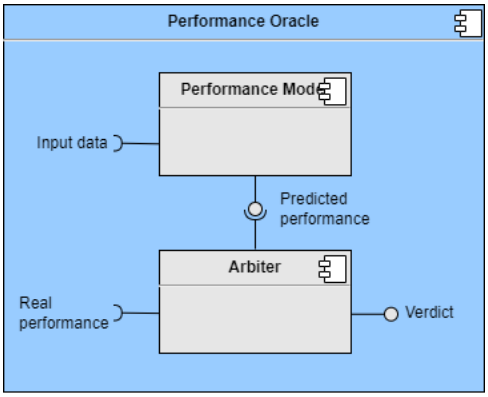


Figure 5.2: Overview of the sub-components of the Performance Oracle

means of different techniques to fit the use case in terms of prediction accuracy or other criteria such as resource consumption.

To train this Performance Model is necessary to have access to data from previous software version executions. This requires the data to be obtained from an error-free version of the software and the need to include performance data for several different conditions.

Once the model is trained, it can predict the performance of the new software version. The data monitored from the execution of this new version provides the input data for every cycle of the software to the trained performance model, which predicts the performance of the system so that the Arbiter can use these data to detect errors.

5.1.2.2 Arbiter

The Arbiter is the component that (1) receives the execution time predicted by the Performance Model and (2) the actual execution time of the studied software and computes different calculations to detect deviations in the performance of the software and raise a verdict. These calculations consider different metrics obtained from the performance data, as these errors may affect the system in different ways: some errors may be punctual errors manifested when specific situations arise, while other errors may degrade the performance of the system more persistently.

5.2 The Oracle in the Adeptness Architecture

This section describes how the Performance Oracle interacts with the other microservices in the Adeptness architecture (presented in Section 2.2) to perform the validation of a new software version. In this sense, three different steps are identified: the deployment of the oracle, its configuration, and its execution.

5.2.1 Deployment

In Figure 5.3 an overview of the deployment process of the Performance Oracle is shown. When the Automation Server detects that a new deployment plan is available in the repository, this deployment plan is provided to the Deployment Orchestrator. This deployment plan must specify the location where the new software and the oracle must be deployed. The Deployment Orchestrator communicates the specific deployment tasks to the Deployment Agent in the edge node where the oracle will be deployed so that it downloads the Oracle microservice and starts its execution. The oracle can be deployed as an executable or containerized, but containerization is recommended to gain in isolation of the microservice. Once the oracle is deployed, the Deployment Agent makes a 'ping' call to the Performance Oracle to check it is correctly deployed and running.

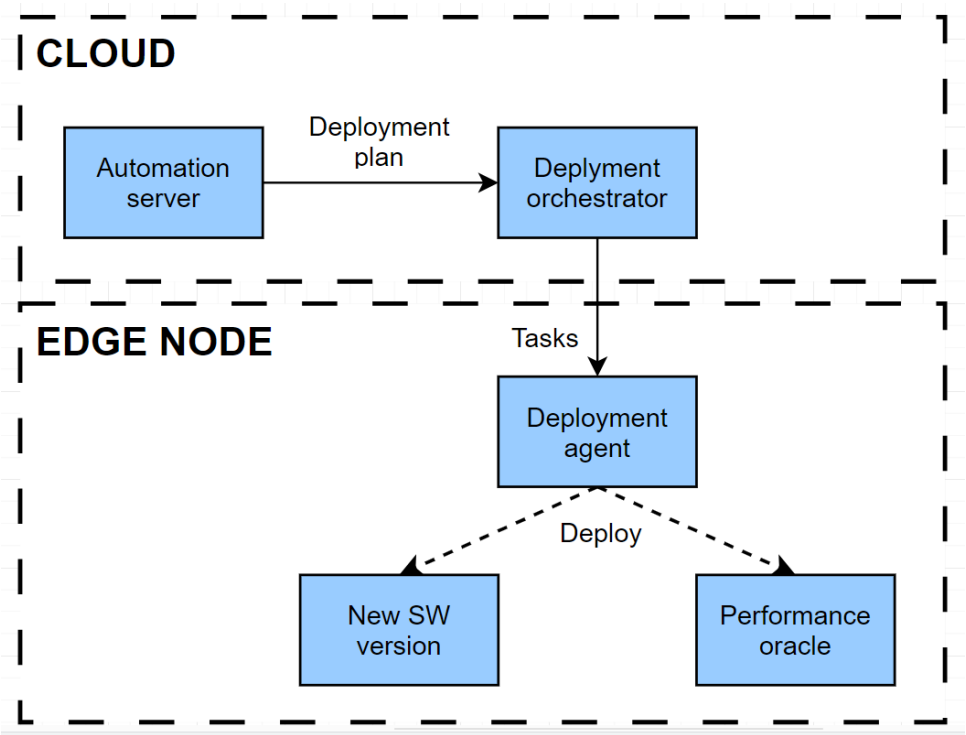


Figure 5.3: Overview of the deployment process of the Performance Oracle

5.2.2 Configuration

Once the oracle is deployed, it must be configured to perform the validation of the software. This configuration is established in the validation plan. In Figure 5.4 the configuration process of the oracle is depicted. The Automation Server provides the validation plan, where all the validation activities are specified, to the Validation Orchestrator. The Validation Orchestrator communicates the specific validation tasks to the Validation Agent in the edge node where the

oracle is deployed so that it can configure it with the appropriate parameters. The configuration needed by the oracle is the topic to which it must subscribe to receive the input data with which it can give a verdict.

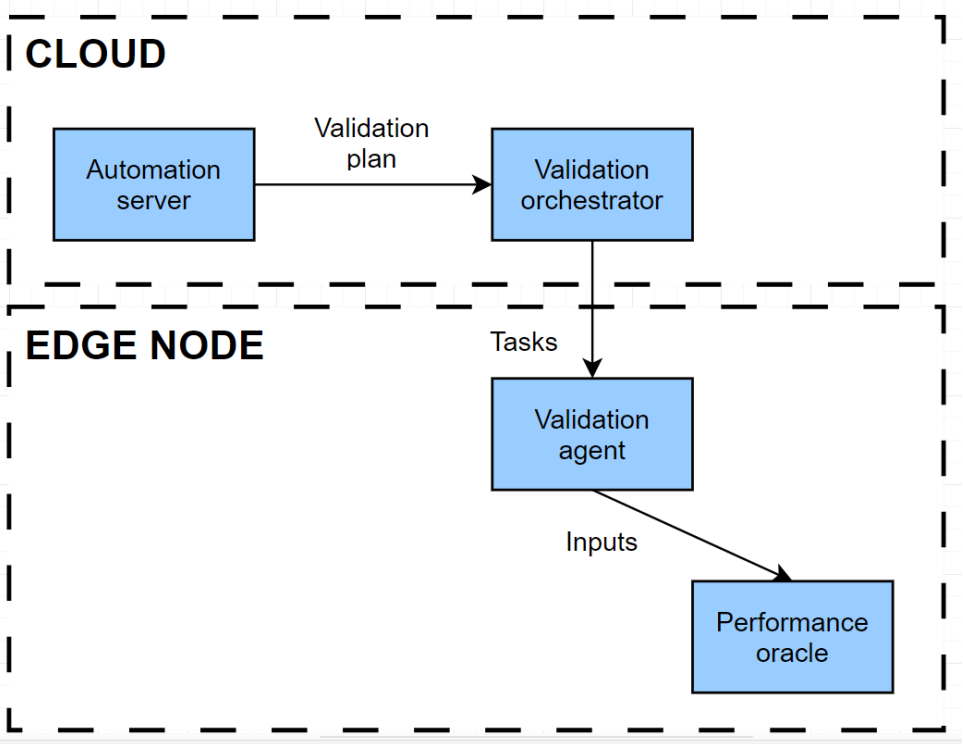


Figure 5.4: Overall overview of the configuration of the Performance Oracle

5.2.3 Execution

Once the Performance Oracle is deployed and configured, its execution starts with the subscription to the topic provided by the Validation Agent and the reception of its publications. For each publication received with input data of the software, a verdict is provided. In Figure 5.5 the whole message exchange of the oracle with the mentioned microservices is shown.

5.3 Requirements

In this section, the requirements that must be met by the Performance Oracle to be useful in the CPS context are described.

It is necessary for software projects that the effectiveness of the process used to test it is guaranteed, so the quality of the test oracle generated has to be measured and should fulfill a series of requirements. In this sense, two sets of requirements were identified: (1) mandatory requirements, which always need to be met in order to be useful in the context of CPSs, and (2) additional requirements, that

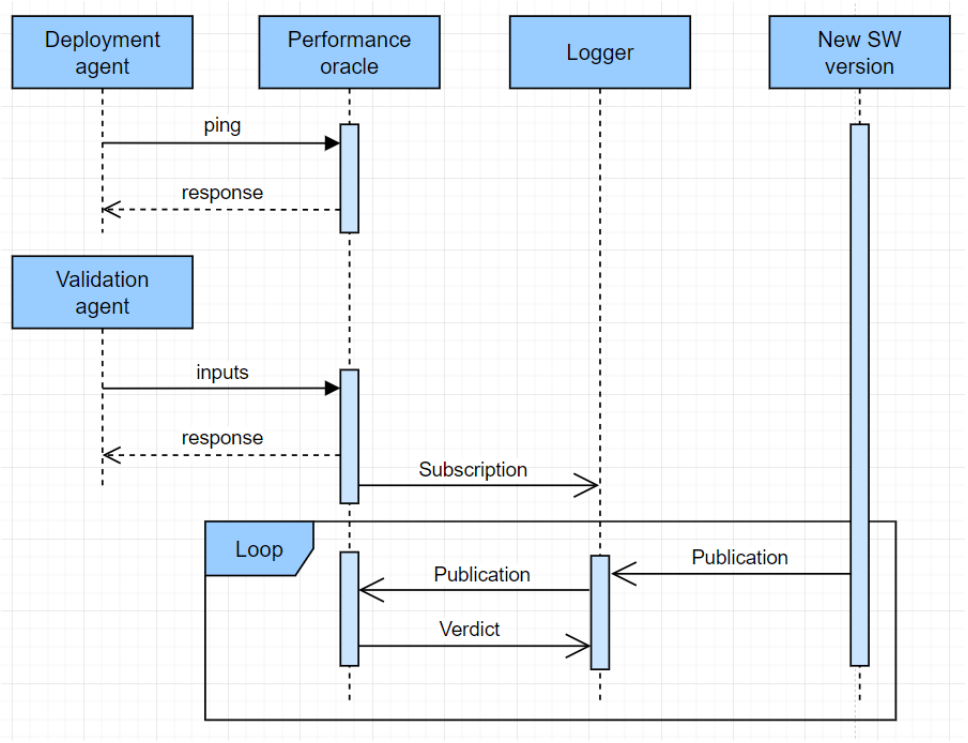


Figure 5.5: Sequence diagram of the execution of the Performance Oracle

even though they are not strictly mandatory, would make the oracle more usable and easier to implement.

5.3.1 Mandatory Requirements

This is the set of requirements that must necessarily be met by the Performance Oracle:

- **Usable in operation:** As previously mentioned, performance errors are likely to appear in operation, so it is necessary to continue testing the system once it is deployed in operation. Thus, the testing of new software should go through different test phases from simulation to validation in the real environment. Consequently, the Performance Oracle should be seamlessly deployed, configured, and executed in all test phases, which include (1) the SiL test phase, (2) the HiL test phase, and (3) Operation. The first one refers to the execution of the test in an environment where all the components of the plant are simulated. In the second, the software is deployed in the real hardware, and real elements of the plant are also included. Finally, the testing in operation occurs once the software is deployed in the real environment.
- **Prediction accuracy:** As an AI-based oracle, a relevant factor when detecting the performance error present in a software is the precision with

which the Performance Model is able to predict the expected performance of the software. Thus, the Performance Model is a crucial component in the oracle and its accuracy should be high enough so that decisions are taken with the most appropriate information, especially in the context of CPSs.

- **Lack of impact on the system:** As stated, CPSs operate in an environment where computational resources are limited and timing requirements can be highly stringent. If the tasks executed in this context are too resource-consuming, the system may crash or degrade its execution. Therefore, the oracle should be optimized in terms of resource consumption in this context. In this sense, the biggest influence on the resource consumption of the oracle is the Performance Model.
  - *Inference time:* If the execution time of the Performance Model is too long, the rest of the tasks executing on the system may miss their deadlines. Therefore, the Performance Model should be executed in a time span that allows the rest of the tasks of the system to continue working properly.
  - *Footprint:* The resource constraints also apply to the field of memory capabilities. Thus, the Performance Model should have a reduced footprint so that other tasks can perform correctly without a memory overflow.
- **Error detection rate:** In the final step of its execution, the Arbiter shall raise a verdict, declaring if the software gathers any performance error within it, by using the performance predicted by the model as a baseline. If a "FAIL" verdict is raised, the software is cataloged as faulty, and presumably, a rollback mechanism may be initiated and the engineers shall start looking for the bug causing this performance dysfunction. In case this verdict is incorrect, which means the software is error-free, this would mean that high efforts in human time and economic resources are being made for no reason. On the other hand, if the verdict raised by the oracle is "PASS", but the software is having an anomalous behavior that is not being detected, the faulty software will continue executing with the risk of degrading the behavior of the whole system.

5.3.2 Additional Requirements

Below the additional requirements that despite not being mandatory, make the oracle more usable in practice are listed:

- **Multi-environment:** CPSs are highly configurable systems that operate in many different environments. Each installation has its own characteris-

tics, with different setups of the system or different workloads. Therefore, it is recommended for the Performance Oracle to be able to maintain its capabilities across multiple environments, so that it is not necessary to build a new oracle for every installation.

- **Interpretability:** One of the reasons why AI is not being applied in industry as much as expected is that these AI models are commonly black boxes, which do not give any insight into the reason why they are making their decisions. This makes industrial actors cautious about giving these techniques control over decisions that may affect the business results of the company. As a consequence, the aim of making these models interpretable, so that they give some traces of the rationale of their decisions, has grown recently. Thus, the decisions of the Performance Model should be traceable so that some insights into the reasons why a ”PASS” or ”FAIL” verdict is raised can be obtained.

5.4 Evaluation

As previously mentioned, it is necessary to guarantee the quality of the test oracle by ensuring it fulfills certain requirements. In this section, how the requirements established in Section 5.3 must be evaluated is described.

In this evaluation, it is necessary to check the characteristics of the Performance Oracle in the testing process of real software. To do so, it is necessary to train the Performance Model with data from previous versions of this software considered to be error-free and domain experts need to configure the Arbiter with the proper parameters for the specific use case. Once, the oracle is built, a set of faulty versions of the software must be created with performance errors injected in them to measure the capacity of the oracle to detect them. These faulty versions go through the SiL and HiL test phases along with the Performance Oracle, and the effectiveness of the oracle throughout this process has been measured.

To follow this evaluation process and prove that the Performance Oracle is suitable for a real industrial case, the use case from Orona (Section 4.4) has been used. Specifically, data from a complex building installation that Orona typically uses to validate dispatching algorithms was used, which is related to a real installation named the Communication City, in Madrid. The building has a total of 10 floors and 6 elevators, each having a capacity of 1250 Kg weight and 16 passengers. Orona has relevant data obtained from this installation in operation, which can be used to build and evaluate the Performance Oracle.

The data to train the model was obtained by executing the dispatcher with data from this installation in the SiL and HiL platforms of Orona, and the Arbiter was developed by consulting the experts from the company.

The requirements established in the section 5.3 were evaluated as follows:

- **Usable in operation:** Evaluating the oracle in a real installation to check its suitability to this context would be too complex so the evaluation of the oracle included SiL and HiL test levels. However, real data obtained from the City of Communications has been used to test the system. Furthermore, the HiL test level is considered to be sufficiently close to the Operation stage to be representative of its behavior in this final step.

- **Prediction accuracy:** As stated, the Performance Oracle needs to obtain accurate predictions of the expected performance to be able to detect performance errors appropriately. Thus, it is necessary to monitor the predictions of the Performance Model and compare them with real performance metrics from the version of the dispatcher used to obtain the training data. The metric analyzed is the Mean Absolute Percentage Error (MAPE), which is easy to interpret and abstracts the developer from the scale of the data.

- **Lack of impact on the system:** As mentioned, it is necessary to limit the impact of the Performance Oracle in the system by minimizing its resource consumption in two terms, mostly focusing on the Performance Model:

  - *Inference time:* The Performance Model was monitored to check the time it requires to make its predictions. If this time is larger than what the system can handle, this model needs to be modified to reduce its impact. In the case of the dispatching algorithm, it is executed every 500 ms. As the oracle should give a verdict for every cycle of the dispatcher, the prediction time must be below this value.
  - *Footprint:* The footprint of the model can be checked before execution, and it is compared with the available memory in the testing platform to verify if it meets the memory requirements of the system.

- **Error detection rate:** To validate the error detection rate of the oracle the Performance Mutation Testing approach [17] was used, which was followed to systematically build faulty versions of the dispatcher with performance errors in it. These faulty versions are executed and the oracle checks this data to decide wether an error exists or not.

- **Multi-environment:** This requirement implies the oracle not to be installation specific so it can adapt to different installations. To verify this requirement, the Performance Model was trained with data from different installations other than the City of Communications, and then data from this installation was used to test the prediction accuracy of the model and the overall performance of the oracle.

- **Interpretability:** Depending on the type of technique used to build the Performance Model it gives more information on the causes of its predictions. For example, GP techniques output a function relating the inputs to the performance, which would help to understand the predictions of the model.

5.5 Conclusion

In this chapter, the characteristics of the Performance Oracle were detailed. First, the interfaces of the Performance Oracle and its internal sub-components were described, Then, how it interacts in the field with the Adeptness architecture was detailed. Afterward, the requirements it must meet to be usable in the CPS context were listed and finally, an evaluation procedure to test the capabilities of the oracle was established.

6

Performance Model

In this chapter, the process followed to build and evaluate the Performance Model of the oracle is described. To build the Performance Model so that it can accurately predict the performance of the system it is necessary to train it. The training of the Performance Model involves providing it with a set of input-output pairs, called the training data, which the model uses to learn patterns and relationships in the data. The process of training the model involves adjusting the parameters of the model so that it can accurately predict performance in new conditions. This is done by using optimization algorithms that minimize the difference between the predictions of the model and the true values in the training set. Once the model is trained, it can be used for predicting performance in new situations.

Building a prediction model is a complex process that requires the consideration of various factors. The main factors to consider when building a good prediction model include:

- **Data quality:** The data should be representative of the real-world situations the software may encounter and should be sufficient to train the model effectively. In this sense, the effectiveness of two types of data is explored: (1) data monitored in real environments and (2) theoretical data generated synthetically in the lab.
- **Features:** The selected features should be relevant to the performance of the system. Thus, it is necessary to explore the available data and combine it with domain knowledge to choose the most adequate features.
- **AI techniques:** Different AI techniques have different strengths and weaknesses and are better suited to different types of problems. Thus, it is necessary to explore different solutions for different use cases and choose the one best fitting the established requirements.
- **Evaluation criteria:** It is important to use appropriate evaluation metrics

that are relevant to the context and use them to compare different models and select the most appropriate one.

In Section 6.1 the process of monitoring and pre-processing the data to train the Performance Model is described, in Section 6.2 the different AI techniques used to build the model are detailed, and in Section 6.3 the characteristics of the model are evaluated. Finally, in Section 6.4 the chapter is concluded by summarizing its content and exposing the extracted conclusions.

6.1 Training Data

As mentioned, the data used for training can have a significant impact on the accuracy and robustness of the Performance Model, so the quality of the data must be guaranteed. In this sense, two main actions are considered regarding training data treatment, which are (1) the monitoring of the necessary data and (2) its pre-processing to maximize its effectiveness.

6.1.1 Monitoring

In the monitoring stage, it is necessary to obtain the performance metrics of a previous version of the software under different conditions so that the Performance Model can learn the patterns of its behavior. To execute the software, it is necessary to make a choice on what input data to use for the execution and the platform where it is going to be executed.

6.1.1.1 Input data

To execute the studied software in order to obtain its performance metrics, different input data can be used. In this sense, Two alternatives are considered:

- **Real data:** The first option is to use real data, which is actual data collected from real operational environments. This type of data can be more representative of the real-world scenarios in which the model will be used, and can lead to more accurate and robust models. However, real data may be more difficult to obtain and may require more pre-processing to make it usable for training. For example, in some cases, the data may be incomplete, noisy, or have missing values, which requires appropriate techniques to handle these issues. Additionally, real data may also contain bias and confounding variables, which should be handled carefully to avoid any negative impact on the model.

- **Theoretical data:** The second option is to use theoretical data, which is created artificially and not based on real-world observations. This type of data can be generated manually or by using simulation tools and can be useful for training and validating a model. Simulated data can be used to generate a large number of data points with less effort, which can be useful to train models requiring large amounts of data such as Deep Learning (DL) models. Moreover, as theoretical data is easier to obtain than real data, it allows training the model with very diverse data (i.e., data representing a wide range of different situations), which would require a big effort to obtain from real environments.

  However, it may not accurately represent the real-world scenarios in which the model will be used, and may not lead to a model that is robust and generalizable to new data.

Ultimately, the choice between using theoretical data or real data monitored from Operation depends on the specific use case and the availability of data.

6.1.1.2 Execution Platform

The performance of the software is obtained by monitoring its execution using the input data. The execution of the previous version of the software can be performed on different platforms, in order to have data to test the new versions in different test phases. In this thesis, two alternatives were considered, Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL).

- **Software-in-the-Loop:** SiL testing involves simulating the behavior of a control system or a software component within a virtual environment. In this type of testing, the software component is run on a computer, and its interaction with other components is simulated using software models. SIL testing allows for early testing and validation of the software component, which can be used to identify and fix issues before moving to more expensive and time-consuming testing methods.

- **Hardware-in-the-Loop:** HiL testing, on the other hand, involves testing the software component in conjunction with the physical hardware it is intended to control. In HIL testing, the software component is connected to a physical plant model, which simulates the behavior of the hardware. The software component is then tested against this simulated hardware to ensure that it behaves as expected. HIL testing can be used to identify issues with the software component that may only arise when it is integrated with the physical hardware.

- **Operation:** The execution of the software in real operational installations is necessary to obtain real data to train the model (Section 6.1.1.1). However, it is not necessary to extract the performance of the software in operation as long as the hardware used to execute the software is the same as the one used in real installations.

6.1.2 Pre-processing

Data pre-processing refers to the techniques used to prepare and clean the data before it is used to train a model. This step is important because the quality and structure of the data can have a significant impact on the performance of the model. Some common data pre-processing techniques include:

- **Data transformation:** This step involves converting data into a format that can be easily understood and used by AI techniques. This can involve changing the structure, format, or values of the data to make it more suitable for training a model.
- **Feature selection:** Feature selection is the process of choosing a subset of relevant features for use in the training of the model. The goal is to select a small number of features that have the strongest relationship with the performance of the system while excluding features that are less relevant or that may introduce noise into the Performance Model. This can improve the accuracy and interpretability of the model and can also decrease the computational cost of training the model. In cases where the dataset has a large number of features, or where the features are highly correlated, feature selection can help to improve the performance of the model by reducing overfitting and improving interpretability.

  To select the most relevant features to train a ML model, many techniques can be used:

  - *Correlation-based selection:* This method is based on the idea that features that are highly correlated with the target variable are likely to be important for the model.
  - *Wrapper methods:* These methods use a specific ML algorithm to evaluate the importance of each feature.
  - *Genetic Algorithms (GA)-based selection:* This method uses GAs to search for the optimal subset of features that maximizes the performance of a model.

  Additionally, domain knowledge in feature selection is crucial for building an accurate and reliable Performance Model. Domain knowledge can

help in identifying relevant features that are directly or indirectly related to the performance of the system. Moreover, domain knowledge can also help in identifying and excluding features that are unlikely to affect performance. This can help to reduce the dimensionality of the feature space and increase the effectiveness of the model by eliminating noise and irrelevant data. Additionally, domain knowledge can also be used to design appropriate data pre-processing and feature engineering steps that can help in transforming raw data into more meaningful features.

- **Data cleaning:** In this step the aim is to identify and remove errors, inconsistencies, and missing values in the data. This can involve techniques such as data imputation, data validation, and data verification.
- **Data splitting:** The step of dividing the data into training and testing sets. This is typically done to ensure that the model is tested on unseen data and to prevent overfitting.

These steps are common steps in data pre-processing, but they are not always necessary, neither the only steps involving pre-processing. The specific steps used depend on the dataset and the specific use case.

6.2 AI Techniques

In this section, the different AI techniques used to build the Performance Model are described.

6.2.1 Configuration

The accuracy of the models built with AI techniques can vary depending on the specific settings or values used to determine the behavior of the learning process. Each technique relies on different configuration parameters, also known as hyperparameters. Some insights into the hyperparameters of traditional ML algorithms, NNs, and GP are now given.

6.2.1.1 Machine Learning

In ML, the specific hyperparameters can vary from model to model, since the algorithms differ substantially from one another. For example, in Decision Trees the maximum depth of the tree or the minimum number of samples required to split an internal node can be configured [10], and in SVMs the penalty parameter of the error term (a.k.a., C) or the kernel type to be used in the algorithm (e.g., 'linear', 'poly', 'rbf', 'sigmoid') are configurable.

6.2.1.2 Neural Networks

NNs have a wide range of hyperparameters that can significantly impact their performance [34]. The most relevant of these include:

- **Number of Hidden Layers and Neurons Per Layer:** This is one of the most significant hyperparameters. It determines the complexity and capacity of the network. However, more layers and neurons can lead to overfitting. An approach to setting these parameters is to gradually increase them until the network begins to overfit.
- **Learning Rate:** It determines how much the weights in the network change with each update. If it is too large, the model could skip the optimal solution, but If it is too small, the model could need too many updates to converge.
- **Batch Size:** This is the number of training samples used in one iteration. A smaller batch size uses less memory and can train the model faster, but it also can have a regularizing effect and lead to a more general solution. However, a too-small batch size can lead to an unstable training process.
- **Epochs:** This refers to the number of times the learning algorithm works through the entire training dataset. Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model.
- **Activation Function:** Activation functions introduce non-linearity into the network, allowing it to learn from more complex datasets. Common activation functions include *Rectified Linear Unit (ReLU)*, *sigmoid*, and *tanh*.
- **Optimizer:** The choice of optimization algorithms includes *Stochastic Gradient Descent (SGC)* or *Adam*.
- **Regularization:** Techniques like *L1*, *L2*, and *dropout* can help prevent overfitting in the network by adding a penalty to the loss function or randomly disabling neurons.

6.2.1.3 Genetic Programming

In Genetic Programming there are several hyperparameters that need to be considered [4, 49]:

- Population Size: The population size refers to the number of individuals (programs) in each generation. Larger populations may provide more diversity, increasing the chance of finding a good solution, but they also require more computational resources.

- Generations: This is the number of times that new populations will be created, which is equivalent to the total number of iterations of the genetic programming run. More generations usually allow better solutions to be found, at the cost of increased computation.

- Selection Method: The selection method, such as tournament selection or roulette wheel selection, determines how individuals are chosen for reproduction.

- Crossover Rate: This controls the frequency of crossover (recombination) events. Crossover is a primary source of exploration in the search space, creating new, potentially better, combinations of existing program structures.

- Mutation Rate: This parameter controls the frequency of mutation events, which alter parts of the individuals in the population. Mutation provides diversity and can help to avoid local optima by introducing new elements into the current solutions.

- Max Depth of Programs: This hyperparameter sets a maximum limit on the depth of the programs (trees) that are evolved. This can prevent programs becoming excessively large and complex, but setting it too low can limit the expressiveness of the solutions.

- Fitness Function: Although not strictly a hyperparameter, the design of the fitness function is crucial. It must accurately reflect the problem's objectives, and the difference in fitness between solutions should reflect their relative quality.

6.2.2 **Selection Criteria**

The choice of using each technique depends on the specific requirements of each use case. Traditional ML has the advantage of being well understood, and thus easier to implement, debug and interpret. These algorithms are simple and computationally efficient. However, traditional ML is limited in its ability to handle complex non-linear relationships between inputs and outputs.

NNs, on the other hand, are a more complex type of technique that can handle complex patterns and representations in the data that traditional ML algorithms cannot identify. However, this complexity makes NNs often more computationally intensive and difficult to interpret. Furthermore, NNs require more data to be trained effectively, which also can lead to overfitting.

Finally, GP has the advantage of creating simple and very interpretable models, generating as output a function relating the input features with the

output values. However, it struggles to identify complex patterns and it can be challenging to ensure that the solutions are not overfitted to the training data.

6.3 Evaluation

In this section, the empirical evaluation of the training process of the Performance Model is reported.

As mentioned, the case study of the dispatcher algorithm of Orona has been used to evaluate this approach. As mentioned, the use case is the dispatching algorithm of Orona, and several AI techniques were used to train models with data obtained from the execution of this software in different platforms (i.e., SiL and HiL) and with different input data (i.e., theoretical and real) for a building called the City of Communications.

6.3.1 Research Questions

The evaluation aims to answer the following RQs:

- **RQ1:** How do the different performance models perform in the different test phases?
- **RQ2:** Are there differences in the prediction error when training the model with real and theoretical data?
- **RQ3:** What is the resource consumption of the performance models?
- **RQ4:** Is this performance maintained when the Performance Model is built for multiple installations?

6.3.2 Experimental Setup

A set of experiments were designed to answer the RQs. As stated, the aim is to build an oracle that is usable in Operation as well as in the previous test phases of CPSs. However, performing the evaluation in a real operational environment would be too complex. Thus, the evaluation has been done considering SiL and HiL contexts to answer RQ1. In these contexts, data from the mentioned City of Communications building was used, which was useful to answer RQ2. To answer RQ3, C code was generated for the models and their characteristics were measured. Furthermore, to answer RQ4, a third context was included, where data from different installations was used to train a multi-installation model in the HiL phase.

The framework used to train the performance models was MATLAB. This choice was made because of two main reasons: (1) it has multiple toolboxes

supporting a large corpus of ML algorithms, GP, and NNs, and (2) it allows automatic generation of C code, which can later be used to embed the model in the oracle and use it in the real target. However, an additional framework was used to ease the creation of NN models, called Autokeras[1]. AutoKeras is an open-source library that provides an easy-to-use interface for building NN models automating network architecture search and hyperparameter tuning to select the model that best fits the training data.

The performance metric used to model and test the performance of the dispatching algorithm in all the contexts was its execution time. However, each context required specific treatment of the training data as well as the use of different AI techniques.

6.3.2.1 SiL Context

The SiL platform consists of executing the algorithm in a simulation environment. In the context of dispatching algorithms a widely used simulation framework is Elevate[2]. Elevate is an elevator simulation framework that provides a platform to evaluate and test different elevator control strategies and to study the performance of elevator systems under different scenarios and traffic patterns. This tool simulates the operation of an elevator system by modeling the behavior of elevators, as well as the traffic flow and passenger demand within a building. Users can define the characteristics of the building, such as the number of floors, the number of elevators, and the traffic pattern, and then simulate the operation of the elevator system with different control systems, such as conventional or destination dispatchers.

In Table 6.1 the main characteristics of the SiL setup are summarized, which are afterward further extended.

Table 6.1: Summary of the experimental setup in SiL

Platform	Installations	Features	Profiles	AI Techniques
SiL	**1 installation** Levels: 10 Cars: 6	**1 feature** Active calls	**Theoretical & Real** Theoretical: 10 Real: 4	**Traditional ML** Regression Tree Ensemble RGP Stepwise SVM

[1]https://autokeras.com
[2]https://peters-research.com/index.php/elevate

6.3.2.1.1 Training Data

Now the process followed to monitor and pre-process the data to train the Performance Model is described.

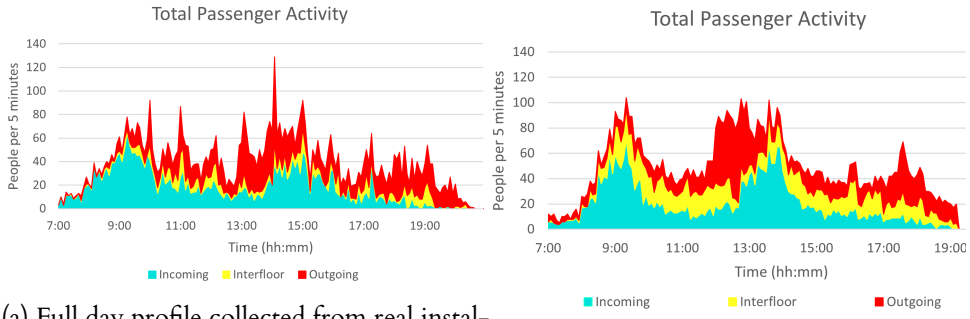
- **Monitoring:** The process of monitoring the execution time of the dispatching algorithm to obtain data to train the Performance Model is now described.

  - *Input Data*: Data obtained from the execution of the dispatcher with theoretical and real profiles for the Communication City building were used to train the models. Specifically, a total of 10 theoretical profiles were used, while the number of real profiles was 4. This is due to the lack of availability of more real profiles monitored from real installations. A passenger profile (either real or theoretical) consists of a .txt file including a list of all the passengers who made a call for an elevator. For each passenger, this file includes (1) the arrival time (i.e., when the passenger requests an elevator), (2) the arrival floor, (3) the destination floor, (4) weight of the passenger, (5) the capacity factor by mass, (6) the loading time, (7) the unloading time and (8) information related to the behavior of the passenger when not all elevators serve all floors. In Listing 6.1 an extract of a passenger traffic profile is provided.

    Listing 6.1: Snippet of a passenger traffic profile

    ```
    ...
    25420, 1, 5, 75, 80, 0.2, 0.2, 2
    25426, 1, 4, 75, 80, 0.2, 0.2, 2
    25427, 1, 3, 75, 80, 0.2, 0.2, 2
    25430, 1, 6, 75, 80, 0.2, 0.2, 2
    25455, 1, 9, 75, 80, 0.2, 0.2, 2
    25481, 1, 6, 75, 80, 0.2, 0.2, 2
    ...
    ```

    In Figure 6.1 the number of calls of real and theoretical passenger profiles can be observed. As can be seen, theoretical profiles present an overall higher number of calls during the whole day with long periods of high traffic, intended to stress the system. On the contrary, real profiles show a lower number of calls, with a more irregular distribution. One discrepancy that can be observed is that the inter-floor calls are not that frequent in the real passenger profiles when compared to the theoretical ones. In addition, at the lunch peak, a high peak of outgoing passengers can be seen at around 14:00 in



(a) Full day profile collected from real installation by Orona

(b) Siikonen Full day profile

Figure 6.1: Passengers activity of real installation and theoretical profiles obtained with Elevate

the real passenger data, which is a pattern that is not present in the theoretical data.

These differences may affect the effectiveness of the data to train the Performance Model, especially if the aim is to use it in a real installation. Table 6.2 summarizes the main characteristics of the traffic profiles used in these experiments, including the number of up calls, down calls, and the duration of the execution of the profile.

Table 6.2: Main characteristics of the used test cases during the experimental scenarios

Traffic profile	File Size (KB)	Data Points	Simulation time (h:min)	# of Up Calls	# of Down Calls
Real1			8:30	2756	1711
Real2			9:10	3086	2366
Real3			11:45	3438	3117
Real4			13:35	3508	3050
Theoretical1			12:55	3994	3377
Theoretical2			12:55	3950	3379
Theoretical3			12:55	3983	3379
Theoretical4			12:55	3989	3402
Theoretical5			12:55	3989	3387
Theoretical6			12:55	3964	3384
Theoretical7			12:55	3977	3386
Theoretical8			12:55	3919	3433
Theoretical9			12:55	3976	3354
Theoretical10			12:55	3945	3407

- *Performance metrics:* In this context, the execution of the dispatching algorithm is done in a PC running a Windows OS. This means that

the response time measurements may gather deviations caused by the interference of other processes. Therefore, two actions were taken to mitigate this limitation: (1) each passenger profile was executed 5 times to obtain more training data and (2) the data was aggregated so that each data point accumulated the number of active calls and execution times of all the cycles of the algorithm every minute, so that the error is reduced. A passenger call is considered to be "Active" from the moment the call is made until the passenger arrives at his destination. In Figure 6.2 the response time of the dispatching algorithm per number of active calls monitored in the SiL environment can be seen. These data include the measurements done by executing the algorithm with both theoretical and real profiles.

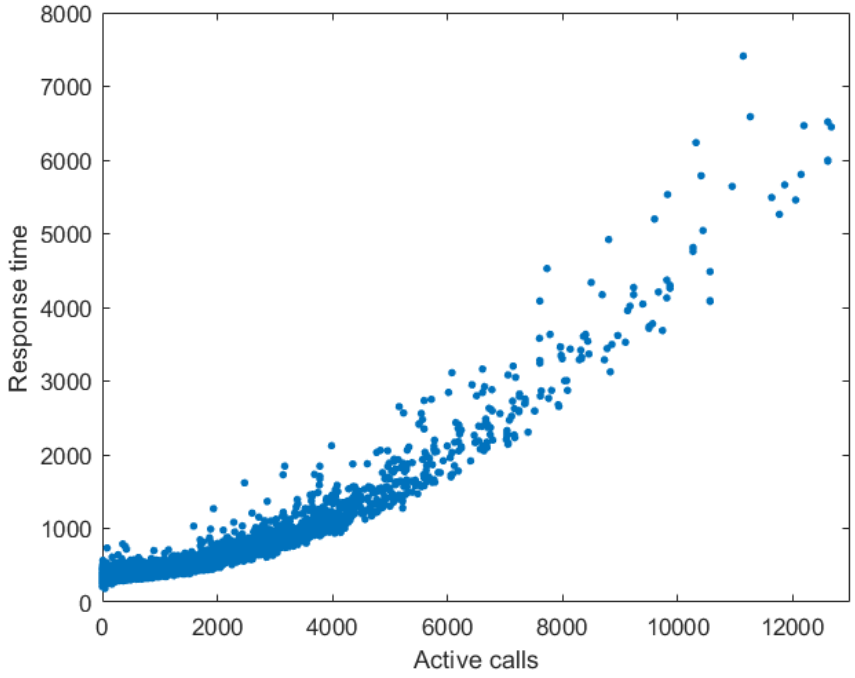


Figure 6.2: Execution time of the dispatching algorithm per active call in SiL

- **Pre-processing:** The process of treating the data to be useful to train the model is as follows:

  - *Data transformation:* As mentioned, the passenger profiles consist of a list of passengers calling for an elevator through time. But these calls remain active from the moment the passenger makes the call to the moment it leaves the elevator, passing through different phases. Elevate gathers the status of the building, including elevators and passenger call status, creating a set of parameters to pass as inputs to

the dispatching algorithm in every iteration. Some of these input arrays are listed in Listing 6.2:

Listing 6.2: Input arrays to the dispatching algorithm

```
...
call state = [1, 2, 0, ...]
origin_floor = [0, 1, 0, ...]
destination_floor = [4, 0, 7, ...]
car_state = [1, 0, 1, ...]
assigned_car = [3, 6, 1, ...]
...
```

From these input arrays, the number of active calls for every execution of the dispatching algorithm can be extracted, as well as the state of each call and its trajectory.

- *Feature extraction:* After analyzing the relation of the extracted features with the execution time of the algorithm and receiving feedback from domain experts, a set of features that could affect the execution time of the algorithm was defined. In this context, the most significant feature for the response time of the dispatching algorithm was selected to train the Performance Model, which is the number of active calls.
- *Data cleaning:* In this step a first overview of the data was taken to identify outliers that may distort the training of the model, affecting its accuracy. The data obtained in the SiL context, shown in Figure 6.2, presents a uniform pattern, without any data point that considerably differs from the general trend. This may be because of the fact that in the SiL environment, the data of all the cycles of the algorithm every minute was aggregated in each data point, to reduce the interference of other processes. Thus, it is not necessary to delete data points for SiL data.
- *Data splitting:* Two experimental scenarios were designed, where the aim is to analyze the effect of different training data on the results. The scenarios are described in Table 6.3 and the names of the proposed scenarios follow the pattern [*Training data type-Testing data type*].

6.3.2.1.2 **AI Techniques**

A number of AI techniques were evaluated to select the one best fitting the established requirements in this use case. In the SiL context traditional regression learning algorithms were used, discarding other techniques such as NNs or GP. This is due to the fact that in this context the model is fed only with one feature. This makes NNs too complex to be applied in this context, facilitating the

Table 6.3: Description of the designed testing scenarios

Scenarios	Description
Theoretical-Real	In this scenario, a different type of data was used for training and testing the model. Data obtained from the real installation was used for testing but theoretical data was used for training, 10 profiles specifically. This scenario would emulate how the theoretical passenger data performs when training the models to use the oracle in the real installations.
Real-Real	In this scenario, data obtained with data extracted from the real installation in Operation were used for training. In total, four passenger profiles were available for the building used in the evaluation. The 4-fold cross-validation was thus used to validate the models along with all the selected AI techniques.

appearance of overfitting. In the case of GP, it makes the search space too small. Thus, the training algorithms used for the SiL platform were: (1) Regression Tree, (2) Ensemble, (3) Regression Gaussian Process (RGP), (4) Stepwise, and (5) SVM Regression. The reasons why these algorithms were chosen were (1) availability within the MATLAB framework and (2) the selected algorithms are expected to have a fast prediction and training speed, as well as small memory usage, and require minimal tuning.

The C code for the models was generated by the Matlab Coder Toolkit[3].

6.3.2.2 **HiL Context**

In this context, the simulation environment is substituted by a hybrid environment where some hardware components are included so that the software can interact with them. This results in a more complex scenario where the software is integrated with the real-time infrastructure, including a Real-Time Operating System (RTOS) and the real target microprocessor and communications. Unlike at the SiL phase, the physical part of the system of elevators is emulated in real-time with appropriate HiL test benches. In practice, the dispatching algorithm would be executed within the real HiL test bench of Orona, but different adjustments had to be made to adapt the platform to the limitations found to evaluate the proposed approach.

- **1st approach:** Executing the algorithm in the real HiL test bench involves

[3]https://www.mathworks.com/products/matlab-coder.html

executing the algorithm in real-time, which made this solution unfeasible, as the experiments would take too long. As an alternative, a Processor-in-the-Loop (PiL) test bench was developed, in which the software is compiled and deployed on the ARM board, and the simulation of the installation and the passenger profiles are performed on the PC. Elevate communicates with the ARM board through HTTP requests to execute the dispatching algorithm, which returns the results to Elevate, which performs the simulation of the status of the building.

- **2nd approach:** After the first adjustment, the execution of the experiments is no longer in real time, but the communication between Elevate and the ARM board still kept the experimentation time too high. Therefore, the second adjustment was to execute the experiments in Elevate to record the inputs to the traffic dispatching algorithm for each cycle and then execute the dispatching algorithm in the ARM board with the recorded inputs. Note that the version of the dispatching algorithm used is deterministic.

In Table 6.4 the main characteristics of the HiL setup are summarized, which are afterward further extended.

Table 6.4: Summary of the experimental setup in HiL

Platform	Installations	Features	Profiles	AI Techniques
HiL	**1 installation** Levels: 10 Cars: 6	**6 features** Registered calls Assigned Calls Calls in travel Car calls Up calls Down Calls	**Theoretical & Real** Theoretical: 10 Real: 4	**Traditional ML** Regression Tree Ensemble RGP Stepwise SVM

6.3.2.2.1 Training Data

Now the process followed to monitor and pre-process the data to train the Performance Model is described.

- **Monitoring:** The process of monitoring the execution time of the dispatching algorithm to obtain data to train the Performance Model is now described.
  - *Input Data:* The input data used to execute the dispatching algorithm remained the same as in the SiL context, which was described in

Section 6.3.2.1.1. As mentioned, 10 theoretical and 4 real passenger profiles were used.

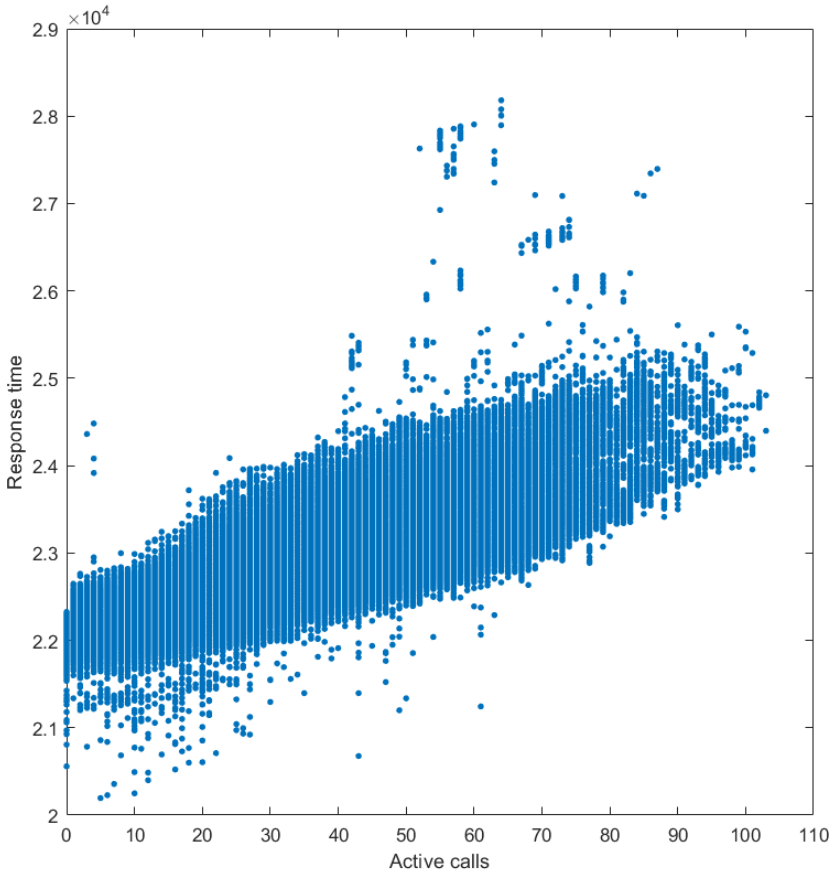


Figure 6.3: Execution time of the dispatching algorithm per active call in HiL

- *Performance metrics:* In this context, the traffic dispatching algorithm runs on top of a Linux OS with a real-time patch, which allows measuring the execution time of each task regardless of interference from other processes. Thus, a task running the dispatching algorithm was created to be executed in the ARM board and measure its execution time. In Figure 6.3 the execution time of the dispatching algorithm per number of active calls is shown. The data shows the measurements done from the execution of the dispatching algorithm in the HiL context with both real and theoretical profiles.

- **Pre-processing:** The pre-processing of the data had some similitudes and differences when compared to the SiL context. The data transformation and the design of the scenarios remained the same, but the data cleaning and feature selection changed.

– *Feature extraction:* In the HiL context, the measurements are much more precise, as the OS is a Linux distribution with a real-time patch, so accumulating minute aggregations can be avoided and establishing a data point for each execution of the algorithm is possible. Furthermore, by having more precise data the number of active calls can be expanded into more detailed features, which are shown in Table 6.5.

Table 6.5: Description of the features used to train the models

Features	Description
Registered calls	It refers to the number of calls made by passengers but not yet assigned to an elevator
Assigned calls	It refers to the number of calls made by passengers and assigned to an elevator but still unattended
Calls in travel	It refers to the number of calls made by passengers and being attended by the assigned elevators
Car calls	It refers to the number of calls made by passengers from inside the elevators
Up calls	It is the number of landing calls with an ascending trajectory
Down calls	It is the number of landing calls with a descending trajectory

– *Data cleaning:* In the HiL context, as can be seen in Figure 6.3, most execution times follow a certain pattern and remain between some limits. However, some data points are far from the general trend, caused presumably by an error in the measurements. Thus, these data points were deleted from the dataset, leading to a dataset represented in Figure 6.4.

6.3.2.2.2 **AI Techniques**

In the HiL context, the chosen algorithms continue to be the traditional ML algorithms mentioned for the SiL context. This was decided because the NN and GP approaches were expected not to give additional benefits in this context.

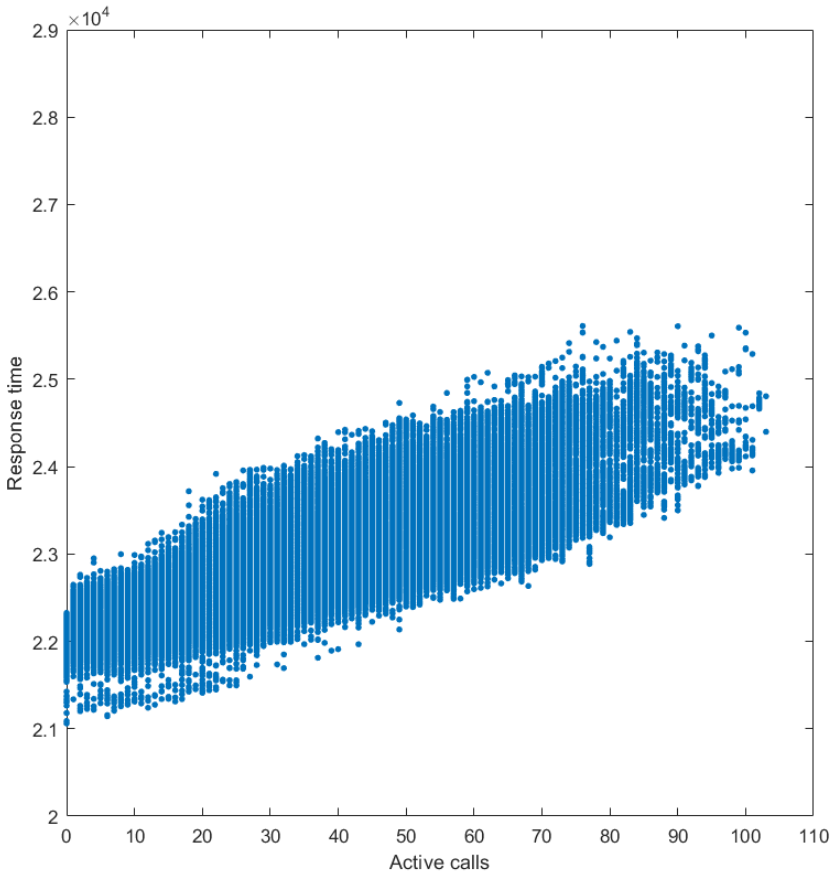


Figure 6.4: Execution time of the dispatching algorithm per active calls in HiL without outliers

6.3.2.3 Multi-environment Context

In this context, the evaluation platforms continue to be the HiL platform but some changes were made to build a model capable of predicting the execution time of the dispatching algorithm in installations with different characteristics. In Table 6.6 the main characteristics of the setup for this context are summarized, which are afterward further extended.

6.3.2.3.1 Training Data

Now the process followed to monitor and pre-process the data to train the Performance Model is described.

- **Monitoring:** The process of monitoring the execution time of the dispatching algorithm to obtain data to train the Performance Model is now described.

Table 6.6: Summary of the experimental setup in multi-environment context

Platform	Installations	Features	Profiles	AI Techniques
HiL multi-env	**8 installations** Levels: 6-20 Cars: 3-8	**8 features** Registered calls Assigned Calls Calls in travel Car calls Up calls Down Calls Levels Cars	**Theoretical** 1 per installation	**Traditional ML** Regression Tree Ensemble RGP Stepwise SVM **Neural Networks** **Genetic Programming**

  - *Input Data:* The Performance Model was trained with data monitored from the execution of the dispatcher with theoretical profiles of multiple installations. The lack of real data from multiple installations prevents us from training this model with real data. The installations executed to obtain the multi-installation data were:
  - *Performance metrics:* The platform where the dispatcher is executed is the same as described in Section 6.3.2.2 for the HiL context. The difference resides in the fact that the algorithm must be configured with the characteristics of each building to execute each passenger profile.

- **Pre-processing:** The treatment of the data in this context cchanges mostly on the selection of the features and the scenario setting.

  - *Feature extraction:* To prove the validity of the model for different installations, it is necessary to include features that model the characteristics of each installation. The features chosen to model the performance of the dispatching algorithms are show in Table 6.8.
  - *Data splitting:* In this context, there is no real data available for the proposed training installations, so only the *Theoretical-Real* scenario could be considered.

6.3.2.3.2 **AI Techniques**

This context becomes more complex allowing other techniques such as GP and NNs to contribute additional benefits.

Table 6.7: Characteristics of the installations used to train the multi-installation model

Installations	Levels	Cars
1	6	3
2	6	5
3	8	3
4	12	5
5	12	7
6	15	7
7	15	8
8	20	8

Table 6.8: Description of the features used to train the models

Features	Description
Levels	The number of levels of the building
Cars	The number of elevators of the building

- **Genetic Programming:** A GP approach was used to obtain a function that models the execution time of the dispatching algorithm by means of the MATLAB Genetic Programming Toolbox. According to the default values on the example provided by the toolbox and the recommendations done by Poli et al. [49], the GP algorithm was configured to run with the configuration listed in Listing 6.3:

  Listing 6.3: Genetic Programming Toolbox configuration

  ```
  Population: 500
  Generations: 50
  Crossover probability: 0.9
  Mutation probability: 0.4
  Selection type: Roulette wheel
  ```

- **Neural Networks:** The Autokeras library was used to perform a search among different NN architectures and hyperparameters to find the most optimal solution for this use case. The configuration of the library was left

as default in most of its parameters but the search space was limited to a maximum of 5 layers and 15 nodes per layer, as preliminary trials showed better results for small network architectures.

For the case, of the GP approach its output is a function relating the input features with the execution time of the algorithm, which can be manually included in the C code of the oracle, and for the NN technique, the generated model was converted to C code using the keras2c[4] library.

6.3.2.4 Summary

In Table 6.9 the experimental setup is summarized.

6.3.3 Results

The results obtained for the three contexts designed to answer the RQs are now presented. To measure the resource utilization of the models their C code had to be generated, but the used MATLAB Coder Toolkit only supports generating code for the Regression Tree and SVM models. Thus, the resource consumption of these models could only be measured. For the NN approach, the Autokeras library performed a search among different hyperparameters and architectures. In Listing 6.4, the layers of the network obtained as a result and their number of nodes is shown.

Listing 6.4: Input arrays to the dispatching algorithm

```
Input layer: 8
Hidden layer 1: 7
Hidden layer 2: 5
Hidden layer 3: 3
Output layer: 1
```

6.3.3.1 SiL Results

In Table 6.10 the MAPE metric for the performance models is shown. The results show how the lower errors are obtained for Regression Tree, Ensemble and RGP on the *Real-Real* scenario, around the 5% mean error, while Stepwise and SVM mean errors decay to 13.69% and 15.46% respectively. For the *Theoretical-Real* scenario, the results worsen for all the algorithms, rounding the 8% of mean error for Regression Tree, Ensemble, and RGP, while Stepwise and SVM techniques worsen their results to 32.44% and 362.12% respectively.

[4]https://github.com/f0uriest/keras2c

Table 6.9: Summary of the experimental setup

Platform	Installations	Features	Profiles	AI Techniques
SiL	**1 installation** Levels: 10 Cars: 6	**1 feature** Active calls	**Theoretical & Real** Theoretical: 10 Real: 4	**Traditional ML** Regression Tree Ensemble RGP Stepwise SVM
HiL	**1 installation** Levels: 10 Cars: 6	**6 features** Registered calls Assigned Calls Calls in travel Car calls Up calls Down Calls	**Theoretical & Real** Theoretical: 10 Real: 4	**Traditional ML** Regression Tree Ensemble RGP Stepwise SVM
HiL multi-env	**8 installations** Levels: 6-20 Cars: 3-8	**8 features** Registered calls Assigned Calls Calls in travel Car calls Up calls Down Calls Levels Cars	**Theoretical** 1 per installation	**Traditional ML** Regression Tree Ensemble RGP Stepwise SVM **Neural Networks** **Genetic Programming**

In Table 6.11 the size in KB of the performance models is shown. The results show that the Regression Tree models have a size of around 32KB, independent of the data used to train them, Regarding the SVM models, the size of the models is bigger and the data used to train it has an effect on the size of the model. Specifically, the model trained with real data has a size of 54,2 KB and the model trained with theoretical data has a size of 71,1 KB.

In Table 6.12 the mean inference time in μs of the performance models is shown. The results show that, as for the size, the SVM models have a larger inference time than the Regression Tree. However, the type of data used to train the models did not seem to have a deep impact on the inference time. Specifically, the mean inference time of Regression Tree models is 1.5 μs and 1.4 μs for theoretical and real training data respectively, and for SVM models the mean inference time is 7.7 μs and 7.9 μs. Note that these measurements were made in

Table 6.10: MAPE for the models trained with data from single installation SiL context

	Theoretical-Real	Real-Real
Regression Tree	8.63	5.06
Ensemble	8.66	5.57
RGP	8.27	4,76
Stepwise	32.44	13.69
SVM	362.12	15.46

Table 6.11: Footprint in KB for the models trained with data from single installation SiL context

	Theoretical-Real	Real-Real
Regression Tree	32.9	32.6
SVM	71.1	54.2

a Windows environment, where timing measurements have some interference.

6.3.3.2 HiL Results

In Table 6.13 the MAPE metric for the performance models is shown. The results show how all the algorithms remain around 0.50% of mean error when trained with theoretical data and that they all slightly improve when trained with real data to around 0.40%.

In Table 6.14 the the size in KB of the performance models is shown. The results show that the size of the Regression Tree models remain around 32 KB for both scenarios, 32,3 KB, exactly. However, the SVM models in the HiL context grow to 18 and 6 MB for the theoretical and real data, respectively.

Table 6.12: Inference time in μs for the models trained with data from single installation SiL context

	Theoretical-Real	Real-Real
Regression Tree	1.5	1.4
SVM	7.7	7.9

Table 6.13: MAPE for the models trained with data from a single installation HiL context

	Theoretical-Real	Real-Real
Regression Tree	0.51	0.42
Ensemble	0.45	0.40
RGP	0.51	0.40
Stepwise	0.47	0.38
SVM	0.50	0.38

In Table 6.15 the mean inference time in μs for the performance models is shown. The results show that, as for the SiL context, the inference time grows for bigger models, but in this case, a significant difference between models trained with theoretical and real data can be seen. The inference time of Regression Tree models is 10.7 μs and 9.8 μs for theoretical and real training data respectively, and for SVM models the inference time is 140,533.8 μs and 46,587.9 μs.

6.3.3.3 Multi-environment Results

In Table 6.16 the MAPE metric for the performance models is shown. The results show how the traditional ML techniques worsen from a mean error of around 0.50% to around 1%. Furthermore, it can be observed that the Genetic

Table 6.14: Footprint in KB for the models trained with data from a ingle installation HiL context

	Theoretical-Real	Real-Real
Regression Tree	32.3	32.3
SVM	18,090.8	6,763.75

Table 6.15: Inference time in μs for the models trained with data from a single installation HiL context

	Theoretical-Real	Real-Real
Regression Tree	10.7	9.8
SVM	140,533.8	46,587.9

Programming approach is not as precise as the ML techniques, with an error of 2.5%. Finally, the NN shows the best result with a 0.79% of mean error.

In Table 6.17 the size in KB of the performance models is shown. The results show that the Regression Tree model continues to have a size of 32 KB while the SVM model has a size of 39 MB. The GP results in a function relating the features with the execution time, so its size is of a few bytes. Finally, the NN model has a size of 81,5 KB.

In Table 6.18 the MAPE metric for the performance models is shown. The results show that THe Regression Tree, GP and NN models keep a low inference time 12.3 μs, 8.2 μs and 16.2 μs respectively, and the SVM model shows a much greater inference time with 276,012.9 μs.

6.3.4 Discussion

The evaluation aimed to assess the use of different data and AI techniques to build the most appropriate Performance Model for the oracle. To this end, the MAPE was evaluated as well as the resource consumption of each model. Overall, the obtained results are positive as the models obtained are able to predict execution time accurately and with low resource consumption.

Table 6.16: MAPE for the models trained with data from multiple installations in HiL context

	Theoretical-Real
Regression Tree	1.83
Ensemble	1.13
RGP	0.84
Stepwise	0.91
SVM	0.98
GP	2.55
NN	0.79

To measure the prediction accuracy, the MAPE metric was used. This decision was made because it is easy to interpret and use-case independent. An error metric that is not dependent on the baseline level may differ between use cases and may be more difficult to manage. In the SiL context, the MAPE metric is higher than in the HiL context for all the models. This is because the measurement of the execution time in the SiL context (i.e., Windows OS) can not be done with the same precision as in the HiL context (i.e., Real-Time Linux). As a consequence, the SiL models are not as precise as HiL models, but still, some algorithms maintain a low MAPE value. Thus, models trained with HiL data seem to be more appropriate to predict the performance accurately.

Furthermore, the source of the training data also has been shown to be relevant to the capabilities of the model. In both SiL and HiL contexts, the models trained with real data showed better results. This might be because the input values (i.e., the number of active calls) move in a similar range to the test data, contrary to theoretical data, which includes a higher number of calls to stress the system.

For the multi-installation models, the complexity of the models grows as installation-specific features must be added, so as expected, the MAPE value grows

Table 6.17: Footprint in KB for the models trained with data from multiple installations in HiL context

	Theoretical-Real
Regression Tree	32.3
SVM	39,759.2
GP	0
NN	81.5

for the traditional ML algorithms. However, in this context, the GP and the NN techniques were included to see if they could outperform the traditional ML algorithms. This did not occur with the GP technique, which did not perform as well as ML algorithms. This technique offers the advantage of creating models easy to interpret but did not show good results in this case. Instead, the NN approach did show a lower MAPE than the rest of the algorithms what indicates that a multi-installation model is to complex for the conventional ML techniques to capture its patterns.

Regarding resource consumption, only the models for which C code creation was possible were evaluated, which were Regression Tree, SVM, GP, and NN. The results show that overall, SVM has a bigger resource consumption than the rest of the approaches. Besides, it can be seen that as the complexity of the context grows from SiL to HiL and Multi-installation contexts, the resource utilization grows, except for the Regression Tree models, which continue around 32 KB. In these metrics, the training data also modified the results. The theoretical models are more resource-consuming, as more data were used to train them. This can not be correctly observed in the SiL model inference times, probably because of the measurement errors in the Windows OS.

Overall, the Regression Tree seems to be a good choice, as its MAPE value is relatively low in all the contexts and its resource utilization is very low. However, when using a more general model, usable for multiple installations, its results decay and the NN model seems to be the most accurate one, despite its resource usage is not as low as the Regression Tree. Thus, the choice of the most appropriate model may vary depending on the available training data or resource availability for each environment.

Table 6.18: Inference time in μs for the models trained with data from multiple installations in HiL context

	Theoretical-Real
Regression Tree	12.3
SVM	276,012.2
GP	8.2
NN	16.2

6.3.5 Threats to Validity

Now, the internal and external validity threats of the performed evaluation are discussed:

6.3.5.1 Internal Validity

A potential internal validity threat in this study might be related to the hyperparameters of the selected AI techniques. To reduce this threat the default parameters from the MATLAB framework were used for training the ML and GP algorithms. Instead, for the NN approach, the Autokeras framework was used, which enables the search of the best hyperparameters by trying different architectures and hyperparameters. Another internal validity threat relates to the usage of 4-fold cross-validation in the *Real-Real* scenario. This was due to the fact that those cases use data obtained from real buildings (i.e., operational data), and Orona did not have more operational data. However, it is important to note that each profile has several data points of full-day traffic profiles, meaning that there is sufficient data to train the algorithms. Finally, another threat is related to the measurements of the response time in the SiL context. As mentioned, this is a Windows environment, where these measurements suffer interferences from other processes, so are not completely precise. This limitation has been mitigated by repeating the measurements several times to obatin more data points.

6.3.5.2 External Validity

An external validity threat in this evaluation is related to using a single benchmark dataset based on test cases for testing dispatching algorithms. To reduce this threat, the dataset was obtained from actual test cases in Orona for testing dispatching algorithms. Furthermore, to avoid bias in the results, the same dataset was not used for training an algorithm and for testing it, using the appropriate k-fold cross-validation techniques in the *Real-Real* scenario. Another external validity threat relates to the used case study. Although only a single case study was used, it is important to note that it is a real industrial case study, which provides a high degree of complexity to the evaluation. Furthermore, the used dispatching algorithm is the most used one in the installations of Orona.

6.4 Conclusion and Future Work

In this chapter, the use of different data and AI techniques to train the Performance Model for the oracle were analyzed. On the one hand, the performance data to build the model was obtained from SiL and HiL context with the use of theoretical and real input data, On the other hand, the AI techniques used were traditional ML techniques, Genetic Programming, and Neural Networks. This model needs to be accurate in its predictions and needs to have a low resource usage, so the different techniques were evaluated to meet these requirements.

In the evaluation, where a dispatching algorithm from Orona was used, several performance models were built combining the mentioned approaches and evaluated their characteristics. This evaluation showed that the input data (i.e., real/theoretical data) can affect the precision of the models in both contexts (i.e., SiL/HiL), real data being the most appropriate choice. The use of HiL data has also been shown to improve the results over the SiL context due to the higher precision of the measurements and the higher complexity in terms of input features. Furthermore, the use of data from multiple installations led to models that worsened their results compared to the single installation context, but the use of a more advanced technique such as NNs proved to be able to outperform traditional ML techniques and build a high-precision model.

In terms of resource utilization, there was a great disparity among the different models. SVM proved to be much more resource-demanding than Regression Trees and NNs, being the former the most efficient. In this sense, the theoretical data led to bigger models, but this might be caused by the amount of data used, which is more abundant for theoretical data.

Overall, Regression Tree showed a good trade-off between prediction error and resource consumption, but NN demonstrated to be more precise in the most complex context (i.e., multi-environment).

As future work in this field, several aspects to continue working on are considered:

- The models were trained using data obtained from specific hardware, thus, these models are not capable of predicting the execution time in other hardware, as the execution time depends on hardware-specific conditions. In the future, it would be interesting to execute the software in different hardware to train the models with these data.
- The training has been done using real and theoretical input data, but the lack of availability of several input data profiles has limited the training with these data. This fact limits the conclusions that may be drawn from the comparison between the use of the different data types. Thus, in the future, obtaining more real data is expected to compare its use to theoretical data and extract more conclusions. Besides, the use of different theoretical traffic profiles will be further analyzed. In addition to the full-day profiles used in this work to evaluate the approach, the use of UpPeak, LunchPeak, DownPeak, and Inter-floor theoretical profiles will be further investigated to analyze whether they can be used to train a valid model with less effort.
- Different training strategies shall be also investigated to avoid the degradation of the model due to changes in the environment. This is known as Concept Drift [72] and, in this sense, two aspects must be taken into account: (1) the configuration of the installation may change over time due to regulatory aspects, maintenance or infrastructure changes, and (2) passenger flow may change depending on the day or season of the year, due to teleworking, vacation periods, etc.
- The tools used during the development have eased the process of building the models but had limited capabilities to build the code to execute the models in real environments. Thus, some models could not be evaluated in terms of resource usage, which is a relevant factor in the CPS context. Therefore, in the future, a different toolkit should be used to generate code for all the models and evaluate its capabilities for their usage in real environments.
- When building the multi-installation models NNs were included to evaluate if they improved the ML algorithms, as they are capable of capturing more complex patterns from data. However, this complexity can come along with higher resource consumption. In this sense, there are techniques that aim at reducing the size of the NN models so that they can fit in resource-constrained environments, such as Quantization [27].

7 Arbiter

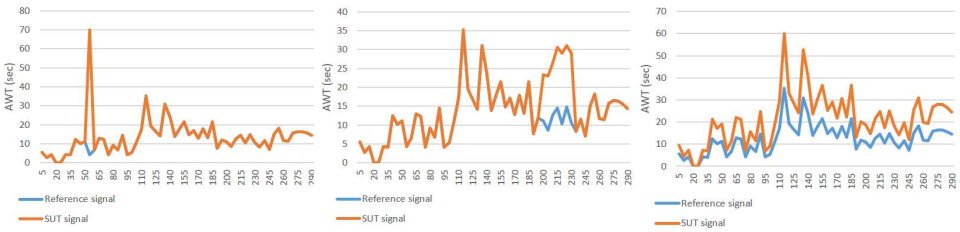
In this chapter, the process of building and evaluating the Arbiter of the Performance Oracle is discussed. As stated in Section 5.1.2.2 the arbiter is the component within the Performance Oracle that decides if the studied software shows a proper performance, or not based on the expected performance predicted by the Performance Model and the monitored actual performance. In Section 7.1, the general logic of the Arbiter is explained, and in Section 7.2 its specific parameters are established. In Section 7.3 the Arbiter is evaluated with a case study and in Section 7.4 the chapter is concluded summarizing its content and extracting conclusions.

7.1 General Logic

As stated in Section 5.1.2.2, the Arbiter must be able to detect errors that may affect the performance of the software at a specific point or in a more persistent way. Thus, three metrics were established to detect performance errors, which are illustrated in Figure 7.1.

The first metric aims to detect the software showing a high peak in the performance metric at a single execution, (Figure 7.1a). The second metric aims to detect the performance of the software exhibiting a value higher than expected for a specified time span (Figure 7.1b). The last metric detects a constant degradation of the performance throughout the execution of the software (Figure 7.1c). High peaks in the performance metrics in a single step or a short period of time may be due to a bug affecting functionalities activated in specific situations, while a more constant degradation may suggest a more global error.

The developed arbitration algorithm aims at detecting these three types of errors. To this end, firstly, the algorithm obtains the quantitative verdict for each execution of the software. This value is obatined by computing Equation 1. A negative value means that the studied software is performing worse than



(a) Failure due to a high peak out of threshold bounds (b) Failure due to a long time out of threshold bounds (c) Failure due to a constant degradation

Figure 7.1: The three reasons why a test can be catalogued as FAIL (blue signal refers to the reference valur and orange signal refers to the value obtained by the software version under test)

expected, whereas a positive value means that it is showing a better performance.

$$verdict(t) = \frac{referenceValue(t) - realValue(t)}{referenceSignal(t)} \tag{1}$$

To detect failures of the first case, a threshold is specified and the Arbiter checks whether this numerical verdict exceeds this threshold in every step of the execution of the software. A code example of this check is shown in Listing 7.1. This will be called as the Single-step Arbiter.

Listing 7.1: Single-step Arbiter

```
1  int get_single_step_verdict(int referenceValue, int realValue, int
       threshold1)
2  {
3      int quantitativeVerdict = (referenceValue - realValue)/
       referenceValue;
4      if (quantitativeVerdict < threshold1)
5          verdict = 0; //FAIL
6      else
7          verdict = 1; //PASS
8
9      return verdict;
10 }
```

This verification is performed for each execution of the studied software and *threshold1* is the threshold specified for the Single-step Arbiter.

To detect failures with the second metric, the Arbiter observes the execution steps for a specified time span and checks whether the mean deviation of the performance metric in this time span exceeds the specified threshold. Listing 7.2 exposes a code example for this verification, which will be called as the Multiple-step Arbiter.

Listing 7.2: Multiple-step Arbiter

```
int get_time_span_verdict(int referenceValue[], int realValue[],
    int threshold2, int numSamples)
{
    int quantitativeVerdict = 0;
    int i = 0;
    while(i < numSamples)
    {
        quantitativeVerdict += (referenceValue(i) - realValue(i))/
    referenceValue(i);
        i++;
    }
    if (quantitativeVerdict/numSamples < threshold2)
        verdict = 0; //FAIL
    else
        verdict = 1; //PASS

    return verdict;
}
```

where *numSamples* is the number of samples towards the past to be considered, and *threshold2* is the maximum deviation allowed defined for the Multiple-step Arbiter.

For the last case, the average value of the deviation over a whole day is obtained and compared against another threshold. Listing 7.3 shows a code example for the last metric, which will be called as the Full-day-Arbiter.

Listing 7.3: Full-day-arbiter

```
int get_day_verdict(int referenceValue[], int realValue[], int
    threshold)
{
    int quantitativeVerdict = 0;
    int i = 0;
    int length = sizeof(realValue)/sizeof(int);
    while(i < length)
    {
        quantitativeVerdict += (referenceValue(i) - realValue(i))/
    referenceValue(i);
        i++;
    }
    if (quantitativeVerdict/length < threshold)
        verdict = 0; //FAIL
    else
        verdict = 1; //PASS

    return verdict;
}
```

where *Threshold3* is the threshold specified for the Full-day-Arbiter.

Generally, the failure threshold for the arbiters is more tolerant for anomalies of shorter duration, since shorter-duration samples may be less representative of

the system. Therefore, the following will usually hold (note that threshold values are negative, and smaller values imply more tolerance):

$$threshold_{single\_step} < threshold_{multiple\_step} < threshold_{full_day} \quad (2)$$

7.2 Parameters

When building the Arbiter, it is necessary to establish the proper parameters for the Arbiter to be accurate. Expertise from domain engineers is necessary to establish the thresholds for the different metrics for each sub-arbiter (i.e., Single-step, Multiple-step, and Full-day arbiters) considering the characteristics of the evaluated software and the environment where it is executed. Furthermore, the time span considered when executing the Multiple-step Arbiter is also a relevant factor to consider in order to adapt the configuration of the oracle for different domains. Thus, domain engineers might consider establishing this parameter considering the expected duration of a performance error introduced on specific functionalities of the software.

7.2.1 Time-span

Specifying an appropriate time span for the multiple-step arbiter can depend on many factors that can be taken into account when setting this window.

- **Input data:** The behavior of the input data may affect the duration of a performance error. Certain conditions may activate certain functionalities of software that may expose performance errors. The duration of these conditions in the input data may establish the duration of the manifestation of the error.

- **Software:** Knowledge about the architecture of the software might give some insight into how much time an error may persist depending on what part of the code is introduced.

- **Actual data:** If data from real faulty software versions is available, its performance behavior can be analyzed to identify recurring patterns or cycles that can help to establish a time span.

Experience and domain knowledge are necessary to study these factors in order to take the most appropriate solution for each use case.

7.2.2 Thresholds

To establish these thresholds, there are some factors that may be considered:

- **Software complexity** The expected performance depends on the complexity of the software, and the effect of bugs in this performance might also do so. Thus, it is important to understand its complexity to determine how bugs may affect its behavior.
- **Input data** The expected performance depends on the specific input data used and varies depending on different input scenarios. Thus, the thresholds should be dependent on the expected value, not an absolute value.
- **Environment:** The actual performance also depends on the hardware and software environment in which the software is running, so the thresholds should change depending on these conditions.
- **Resource constraints:** Specific use cases may present different constraints that may establish hard limits that must be fulfilled in order to guarantee the correct behavior of the application.
- **Domain requirements:** Some domains may have regulatory standards that establish certain requirements that software running in those applications must fulfill.

7.3 Evaluation

As mentioned, the case study of the evaluation in this thesis is the dispatcher algorithm of Orona was used. In Chapter 6, the building process of the performance models was explained, using data from a real building that Orona uses to validate dispatching algorithms, named the Communication City, in Madrid. In this evaluation, the capabilities of the Arbiter to detect performance errors was evaluated by comparing the actual performance of new versions of the dispatching algorithms with the predictions made by the performance models.

7.3.1 Research Questions

The evaluation aims to answer the following RQs:

- RQ1: How does the Arbiter perform when detecting performance errors for new software versions?
- RQ2: How do the different performance models affect error detection?

7.3.2 Experimental Setup

The Arbiter was tested with the predictions of the performance models built as described in Section 6.3.2. Thus, the proposed scenarios remain the same.

7.3.2.1 Testing Method

To evaluate the effectiveness of the Arbiter to detect faulty versions of the dispatching algorithm Performance Mutation Testing [17] was used. Mutation testing aims to generate a set of versions from the original program adding a synthetic variation to its code. Then, these "mutants" are executed and when the outcomes of a mutant differ from the outcomes of the original version, it is considered that the mutant is killed. This technique has been found to be a good substitute for real faults [33]. The difference between traditional mutation testing and Performance Mutation Testing is that the mutation operators for the latter are focused on injecting performance errors while keeping the original functionality of the program, whereas the former focuses on changing the program functionality. In Listings 7.4 and 7.5 an example of a code snippet representing how a potential performance mutant affects the system is shown. The function in this example looks for a given number within an array until the number is found or the end of the array is reached. Instead, the performance mutant removes the second condition from the *while* loop, so forces the function to go through the whole array even if the number is already found. This keeps the functionality of the code but makes it inefficient and more time-consuming.

Listing 7.4: Code snippet of the original function

```
int array_contains(int num, int
    array[], int size)
{
    ret = 0;
    int i = 0;
    while(i < size && ret == 0)
    {
        if(array[i] == num{
            ret = 1;
        }
    }
    return ret;
}
```

Listing 7.5: Code snippet of a performance mutant

```
int array_contains(int num, int
    array[], int size)
{
    ret = 0;
    int i = 0;
    while(i < size)
    {
        if(array[i] == num{
            ret = 1;
        }
    }
    return ret;
}
```

A set of mutants have been systematically created that could affect performance based on the performance mutation operators proposed by Delgado-Perez et al. [17], which include: simulation of heavy operation in different parts of the algorithm, Move/Copy Statement into Loop, Removal of Stop Condition

in Loop and Unnecessary Calculation of values. These faults were introduced in a uniform manner throughout different sections of the source code that are relevant for passenger handling. A total of 30 performance mutants were generated that could affect the execution time of the dispatching algorithm. Seven of these performance mutants were finally discarded from the test executions for different reasons: two of them did not compile correctly, other two led to crashes at execution time, another one got caught in an infinite loop, and two took a too long time to execute, making its execution unfeasible. In all these cases, the manual detection of the performance bug was trivial, not requiring the use of a sophisticated oracle. Therefore, a total of 23 performance mutants were finally used in the evaluation.

7.3.2.2 **Evaluation Metrics**

The Arbiter compares the execution time of the mutants with the predicted execution times, cataloging the verdict either as “PASS” or “FAIL”. This is the Performance Oracle. Additionally, the original previous version was executed with the same passenger list and compared its execution time with the mutants by the Arbiter. This element is called the regression test oracle and also raises a ”PASS” or ”FAIL”, setting ground truth on the mutant functioning. Thus, similar to other works tackling the test oracle problem [15, 23], the verdict provided by the Performance Oracle was considered a True Negative (TN), a True Positive (TP), a False Negative (FN) or a False Positive (FP) as defined below:

- TN: Both the Performance Oracle and the regression test oracle returned a ”PASS” verdict.
- TP: Both the Performance Oracle and the regression test oracle returned a “FAIL” verdict.
- FN: The Performance Oracle returned a “PASS” and the regression test oracle returned a “FAIL”.
- FP: The Performance Oracle returned a “FAIL” and the regression test oracle returned a “PASS”.

Based on a similar work [23], four measures were selected to evaluate the quality of test oracles: Precision (Equation 3), Recall (Equation 4), Accuracy (Equation 5) and F1 (Equation 6). The Specificity measure was also included (Equation 7). In this context, classifying faults well is as important as classifying correct behavior as correct. Therefore, it is necessary to consider metrics involving both TP rates (Precision and Recall) and TN rates (Accuracy and

Specificity).

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5}$$

$$F1 = \frac{2 \times (Precision \times Recall)}{Precision + Recall} \tag{6}$$

$$Specificity = \frac{TN}{TN + FP} \tag{7}$$

7.3.2.3 **Parameters**

To establish the most appropriate parameters of the Arbiter, they were discussed with domain experts from Orona. The time span of the Multi-step Arbiter was set to 5 minutes, based on typical passenger flow patterns. Regarding, threshold values, they were set to values that minimized the appearance of FPs. FPs were considered more problematic than FNs, as the apparition of FPs would make engineers start debugging and testing the software looking for an error that does not exist. Instead, FNs can make faulty versions of the software continue executing in the installation, but these faulty versions are not expected to be dangerous, as the most damaging versions are expected to be easily detected.

For the SiL use case a reduced version of the Arbiter had to be implemented, as the limitations on the execution time measurement in this context forced the data to be compiled for every minute, not for every execution. Thus, the SiL Arbiter only gives verdicts for every minute.

7.3.3 **Results**

Now the results obtained from the scenarios designed to answer the proposed RQs are presented.

7.3.3.1 SiL Results

Table 7.1 shows the metrics obtained for the Arbiter with the *Theoretical-Real* scenario models. The results show that Regression Tree, Ensemble, and RGP are the algorithms showing the best results, with a Precision of around 1, indicating nearly 0 FPs. SVM and Stepwise showed better Recall measures (i.e., 1) but lower Precision, Accuracy, and F1 measures, which means that there are no FNs but FPs appear in this case.

Table 7.1: Results summary of the arbiter when tested with theoretical data from single installation SiL context

	Precision	**Recall**	**Accuracy**	**F-1**
Regression Tree	0.99	0.70	0.83	0.82
Ensemble	1.00	0.65	0.81	0.79
RGP	1.00	0.67	0.82	0.81
Stepwise	0.60	1.00	0.60	0.75
SVM	0.61	1.00	0.61	0.76

Table 7.2 summarizes the obtained results for the ML algorithms for the *Real-Real* scenario. Regression Tree, Ensemble, and RGP showed the strongest results since they showed a Precision of around 0.95, a Recall of around 0.93, an Accuracy of around 0.93, and an F1 measure of around 0.94. Conversely, SVM and Stepwise algorithms showed overall worse results. Despite having a high Recall, their Precision, Accuracy, and F1 were significantly lower than the rest of the algorithms. This means that there was a high number of FPs.

It is noteworthy, however, that for the best algorithms (i.e., Regression Tree, RGP and Ensemble), the Recall, Accuracy and F1 measures improved with respect to the previous scenario, although the Precision decreased. This means that when training with theoretical data, the number of FNs decreased, whereas the number of FPs slightly increased.

Table 7.2: Results summary of the arbiter when tested with real data from single installation SiL context

	Precision	Recall	Accuracy	F-1
Regression Tree	0.95	0.94	0.94	0.95
Ensemble	0.96	0.92	0.94	0.94
RGP	0.95	0.94	0.94	0.95
Stepwise	0.62	1.00	0.65	0.77
SVM	0.58	1.00	0.58	0.74

7.3.3.2 HiL Results

Table 7.3 shows the metrics obtained for the Arbiter with the *Theoretical-Real* scenario models. The results for Regression Tree, RGP, SVM, and Stepwise showed good performance in terms of Precision and Specificity (between 0.93 and 0.94 for the former and 0.95 for the latter), but their Recall measure was low (0.57 for SVM, Regression Tree and Stepwise, and 0.67 for RGP) indicating the appearance of FNs. Their Accuracy and F1 measure remained relatively high (i.e., between 0.75 and 0.79 for the Accuracy metric and between 0.71 and 0.79 for the F1). Conversely, the Ensemble algorithm obtained different results. While its Precision and Specificity were lower, but still competitive (0.90 for both), the rest of the metrics were higher: a Recall of 0.80, an Accuracy of 0.85, and an F1 measure of 0.85.

Table 7.4 show the metrics obtained for the Arbiter with the *Real-Real* scenario models. Within this scenario, all the algorithms showed similar results. In fact, all of them showed the same Precision and Specificity of 0.95, which is highly competitive. In terms of Recall, Accuracy, and F1 measure, the Ensemble showed slightly better performance than the rest, although the differences were minimal. All these values were over 0.80 for the Recall, and over 0.87 for Accuracy and F1. This means that all the algorithms are highly effective within this scenario.

Table 7.3: Results summary of the arbiter when tested with theoretical data from single installation HiL context

	Precision	Recall	Accuracy	F-1	Specificity
Regression Tree	0.93	0.57	0.75	0.71	0.95
Ensemble	0.90	0.80	0.85	0.85	0.90
RGP	0.94	0.67	0.79	0.79	0.95
Stepwise	0.93	0.57	0.75	0.71	0.95
SVM	0.93	0.57	0.75	0.71	0.95

7.3.3.3 Multi-environment Results

The results in Table 7.5 show that traditional ML algorithms keep performing well, with Precision and Specificity values above 0.90, Recall values above 0.80, and an Accuracy of around 0.90. On the contrary, SVM drops the Precision and Specificity metrics to 0.65 and 0.18 respectively, which indicates the appearance of FPs and the Recall value rises to 0.97 which indicates few FNs. Regarding GP, it shows similar results to SVM with very high Recall (0.99), low Precision and Accuracy values (0.62 and 0.65, respectively), and very low Specificity values (0.17). Finally, the NN approach shows a value of 1 in Precision and Specificity, indicating no FPs, a Recall of 0.85, an F1 measure of 0.92, and an Accuracy of 0.90.

7.3.4 Discussion

The evaluation aimed to assess whether the Performance Oracle can be applicable in the context of performance testing of CPSs. To this end, an empirical evaluation was carried out using the study by Delgado-Perez et al., [17] for the creation of performance mutants.

Overall, the results are considered positive, indicating that this approach can be applicable in the CPS context. In the SiL context, some ML techniques showed good results, but this context was less exigent, and fewer mutants were detected. In the HiL context, the results remain similar, with an improvement of SVM and Stepwise, but the scenario is more realistic and similar to the Operation environment, and more mutants are detected. Besides, the relevance of training

Table 7.4: Results summary of the arbiter when tested with real data from single installation HiL context

	Precision	Recall	Accuracy	F-1	Specificity
Regression Tree	0.95	0.80	0.87	0.87	0.95
Ensemble	0.95	0.82	0.88	0.88	0.95
RGP	0.95	0.80	0.87	0.87	0.95
Stepwise	0.95	0.80	0.87	0.87	0.95
SVM	0.95	0.80	0.87	0.87	0.95

the algorithms with real data instead of theoretical data was identified to improve the results.

Based on the performed evaluation, the results of this approach are believed to be good enough to be applicable by Orona engineers when performing long full-day tests.

Similarly to other works in the field of the test oracle problem [23], the Precision, Recall, Accuracy, and F1 metrics were employed, alongside Specificity, which are commonly used by the ML community. Nevertheless, a recently performed systematic literature review [22] revealed that many studies used the mutation score metric, which aims at assessing the percentage of mutants killed. In this case, the mutation score mostly depends on the type of test inputs used, rather than the test oracle itself. In fact, the same criterion as the regression test oracle was used, currently used in the context of Orona for determining whether a mutant is killed or not by a test case. It is important to note that test oracles are prone to false verdicts [9, 63], and therefore, the right classification done by the Arbiter has been prioritized rather than the number of mutants detected.

7.3.5 Threads to Validity

The internal and external validity threats of the performed evaluation are now discussed:

Table 7.5: Results summary of the arbiter when tested with theoretical data from multiple installations in HiL context

	Precision	Recall	Accuracy	F-1	Specificity
Regression Tree	0.96	0.81	0.86	0.88	0.94
Ensemble	0.95	0.86	0.90	0.92	0.97
RGP	0.95	0.86	0.90	0.92	0.97
Stepwise	1	0.83	0.89	0.91	1
SVM	0.65	0.97	0.65	0.78	0.18
GP	0.62	0.99	0.65	0.76	0.17
NN	1	0.85	0.90	0.92	1

7.3.5.1 Internal Validity

A potential internal validity threat in this study might be related to the parameters of the Arbiter, which are configurable. To reduce this threat, the parameters were with domain experts to see which thresholds could be appropriate to consider a test as ”PASS” or ”FAIL”.

7.3.5.2 External Validity

As in the Performance Model evaluation cases, the external validity threats for the evaluation of the Arbiter are related to using a single benchmark dataset and the use of a single use case, which have already been addressed in Section 6.3.5.2

7.4 Conclusion and Future Work

In this chapter, an Arbiter has been proposed, which relies on the predictions of a Performance Model (forming the Performance Oracle) to automatically test new versions of software for CPSs, . This arbiter is composed of three sub-arbiters, which aim at detecting different types of performance errors.

In the evaluation, where an industrial dispatching algorithm from Orona was used, the type of training data used impacted these results, as well as the different AI techniques used. These results are believed to be competent enough to transfer the tool to practitioners, although further investigation is required to enhance these results.

8 Conclusion

This chapter concludes the thesis. A summary of the contributions is presented in Section 8.1, where the validation of the hypotheses is discussed and the main limitations of the proposed solutions are highlighted. 8.2 discusses a set of lessons learned extracted from the experiments conducted during the thesis. Finally, future work is exposed in Section 8.3.

8.1 Summary of the Contributions

In this thesis, the aim was to build a method to detect performance errors in CPS software updates based on AI techniques. This method consists of a Performance Model that predicts the expected performance of the system and an Arbiter that compares this prediction to the monitored actual performance to decide whether an error exists or not.

To build the Performance Model the use of different training data and AI techniques have been investigated. On the one hand, the execution of the software on different platforms (i.e., SiL and HiL) with the use of different input data types (i.e., theoretical and real) was performed to obtain the training data, On the other hand, the AI techniques used were traditional ML techniques, Genetic Programming, and Neural Networks. The aim was to build the most accurate and resource-efficient model to be applicable in the CPS domain. These models were then used to feed the Arbiter with their predictions to verify its ability to detect performance errors. This Arbiter aimed at detecting different types of errors and was evaluated by using the Performance Mutation Testing method. Finally, the mechanism is encapsulated as a microservice, forming the Performance Oracle.

The method developed in this thesis was evaluated using an industrial use case provided by Orona, a company from the elevation domain, which offered its traffic dispatching algorithm.

8.1.1 Hypotheses Validation

In this Ph.D., five research hypotheses were stated in Section 4.2. This section analyses each of the contributions and argues whether the stated hypotheses can be validated.

Hypothesis 1: ***"It is possible to accurately predict the performance of a CPS in certain environment based on knowledge obtained from the execution of the previous release in the same environment."***

To validate this hypothesis, executing an old error-free version of the software was proposed in order to monitor its performance and obtain the training data for the Performance Model. The execution of the old version of the software was done considering specific environmental values and the model was tested under that same environment. The evaluation of the performance models was done using the case study of the dispatching algorithm of Orona, where a real installation of the company was taken as a reference to train and test the model. Different models were built by means of different ML algorithms and the results show that most models show low prediction errors.

Hypothesis 2: ***"It is possible to accurately predict the performance of a CPS in a certain environment based on knowledge obtained from the execution of the software on other environments.***

To validate this hypothesis, a similar approach as for validating the first one was used, but to train the Performance model data from different environments was used and it was tested in another environment. The evaluation of the performance models was also done using the case study of Orona, where a number of simulated installations were used to train the models, and a real installation of the company was used to test them. In this case, Genetic Programming and Neural Networks were used besides traditional ML techniques and the results showed that the models worsened their results compared to the single installation training but the NN model outperformed traditional ML techniques.

Hypothesis 3: ***"Data obtained from the monitoring of real environments is more appropriate to train performance models than laboratory data."***

To validate this hypothesis, the mentioned performance models were trained with data monitored from real operational environments and theoretical data obtained from simulation environments. To evaluate the hypothesis with the use case of Orona, data monitored from a real installation of the company and data extracted from a simulation tool for elevation systems called Elevate were used. The prediction accuracy of the models trained with both types of data were compared and the results revealed that real data training shows better results compared to theoretical data.

Hypothesis 4: ***"It is possible to create a high-accuracy performance prediction***

model with low resource consumption."

To validate this hypothesis, the resource consumption of the performance models (i.e., inference time and footprint) was measured. This was only possible for the models that were supported by the used C code generation tools. The results showed that Regression Tree models are very resource-efficient maintaining high prediction accuracy, while SVM is much more resource consuming and its prediction accuracy falls in some contexts. Furthermore, the more GP was the most lightweight technique but its prediction capabilites were not that good.

Hypothesis 5: ***"It is possible to detect different types of performance bugs in a software based on different metrics."***

To validate this hypothesis, an Arbiter that compares the predictions of the performance models with the actual performance of the system was developed, which based on different metrics decides if an error exists or not. The Arbiter was evaluated by generating a set of faulty versions of the software called Performance Mutants evaluating the capabilities of the Arbiter when detecting these mutants. The results showed that the Arbiter is able to detect mutants precisely, minimizing the appearance of False Positives, and it maintained its capabilities across different environments.

8.1.2 Limitations of the Proposed Solution

This section discusses the limitations of the proposed solutions to be applied in practice.

- The tool used to train the models was MATLAB, which was chosen because it supported a large corpus of ML algorithms and it allowed the automatic generation of C code. However, C code generation was not supported for all the algorithms, so these models could not be tested in the evaluation.
- As stated, the performance models were trained using theoretical and real data. The latter was obtained from real installations of the use case provider, which is Orona. This behavior was reproduced in SiL and HiL platforms for a single installation, but it could not be tested in the multi-installation context because of a lack of real data. Furthermore, the amount of real data available for the used installations was small, so more data should be extracted to obtain more complete conclusions.
- The performance data was obtained from specific hardware for the SiL and HiL environments. This means that if the hardware of the system is different in any installation, the models must be re-trained, as execution time is hardware-dependent.

- The measurements of the execution time of the studied software were made in a context where only one core of the CPU was enabled, as multi-core functioning distorted the measurements. Thus, the proposed approach is currently valid for single-core mode.

- At the moment, the Performance Oracle gives a verdict on the performance of the system but does not give any information about the reason why this error appeared or the conditions that led to that error, so engineers need to debug the software without any insight on the cause of the error.

8.2 Lessons Learned

Based on the results obtained from the evaluation section and the discussion with domain experts about the applicability of the approach, the following lessons were extracted:

Lesson 1 – Training data: The use of the right data for training the AI models plays a critical role in the accuracy of the Performance Oracle. The results obtained in scenario *Real-Real* are much better than the ones obtained in scenario *Theoretical-Real*. According to the precision of the performance models, there was not a huge difference between the two scenarios, but there was when detecting the errors, This might be because despite both having a similar average error, the models trained with real data learn more precisely the patterns of the performance of the software. Therefore, in order for the oracle to be accurate enough in real operational environments, the training data should be also obtained from these environments.

In other contexts, such as web engineering, technologies like DevOps permit the use of data from operation at design time to enhance software engineering processes (e.g., testing). The good performance of the proposed approach with field test data shows the importance of researching on adapting design-operation continuum techniques (e.g., DevOps) in the context of CPSs and in domains like elevation. The simulation at design-time of situations seen only in operation is of great advantage for engineers. For instance, this permits the detection of unforeseen situations that can only be seen when the system is in operation.

Lesson 2 – AI techniques: The empirical evaluation suggested that some AI techniques performed better than others depending on the context to detect performance errors. In the SiL context Regression Tree, Ensemble and RGP showed good results, contrary to SVM and Stepwise, In the HiL context, all the algorithms performed similarly, with good results for real data training and worse for theoretical data training, except Ensemble, which showed good results also for the latter case. Finally, in the multi-environment context, NN outperformed traditional ML techniques and GP did not show good results.

Regarding resource consumption, Regression Tree was shown to be the most efficient algorithm. For the application of the tool in industry, further investigation is required before recommending one of the algorithms for detecting performance bugs.

Lesson 3 – Consequences of mistaken oracle: An oracle may be mistaken when providing a test verdict. That is, the test oracle is subject to False Positives and False negatives. On the one hand, a False Positive means that a test was cataloged as "FAIL" when it should have been provided a "PASS". This results in, probably, the need for the developer to debug where the potential fault is located. Since there is no fault behind it, this may result in time spent by an engineer debugging a fault that does not exist. This issue could be mitigated by confirming the test verdict by using a regression test oracle.

On the other hand, a False Negative means that a test was cataloged as a "PASS" when it should have it done as a "FAIL". If this happens at design time (i.e., at the SiL or at the HiL phases), the bug might be missed and shipped to production. These faults are very minor errors that do not degrade the system, but this consideration is use-case-specific.

8.3 Future Work

In this section, the further research needed to complement this work is summarized. On the one hand, some of these considerations are related to the training of the performance models, already addressed in Section 6.4. On the other hand, there are considerations related to the development of the Arbiter, which were addressed in Section 7.4. Finally, some future work is required related to the structural limitations mentioned in Section 8.1.2 and general considerations.

- Regarding the training data, the options used were theoretical and real data monitored from real operational installation. The latter was limited by the amount of data available from the use case provider. Thus, more data should be obtained in the future to improve the training with this type of data. The training with theoretical data was proven to be less effective than the real data, but only one type of theoretical data was tried. The use of different types of theoretical data can be investigated to test their effectiveness against real data.

- The usage patterns or the configuration of the CPS may vary over time and this can degrade the precision of the Performance Model. This is known as Concept Drift [72] and different strategies should be investigated to detect it and activate a re-training process to adapt the model to the new conditions.

- The performance data was obtained from specific hardware so the model is hardware-dependent. This makes it necessary to build a new model for each hardware platform. Thus, in the future the data monitoring should be extended to multiple hardware and the models should be trained to be able to predict the performance for each hardware platform.

- As mentioned, resource consumption is a relevant consideration when using the Performance Oracle in the CPS context. Thus, additional may be considered to optimize the resource consumption of the models. For instance, Quantization [27] is a method to reduce the size of NNs that could be applied.

- The oracle relies on the predictions of the Performance Model and the monitored performance to give a verdict on the functioning of the system. However, these measurements may suffer distortions caused by the hostile nature of the operational environment, provoking bad verdicts. Therefore, additional mechanisms are needed to detect whether the deviations are caused by software bugs or environmental causes.

- The Performance Oracle gives a verdict on the performance of the system without giving any additional information to help solve the bugs. In the future, it would be interesting for the oracle to give some information about the conditions that led to that verdict so that engineers have some insight into the cause of the error before debugging the software. For instance, different thresholds can be established to differentiate severe and minor errors.

- At the moment, the Performance Oracle only detects performance errors considering the execution time, but the performance of the system may suffer degradation as well in terms of memory usage, energy consumption, or other performance metrics. Thus, new performance metrics may be considered in the future.

- As mentioned, the tool used to train the models was MATLAB due to its large support of ML algorithms and its automatic code generation toolkit. However, some algorithms were not supported and this limited the evaluation. Thus, in the future, new tools for building the models shall be used that permit the C code generation of a wider range of ML models.

- The proposed method was evaluated with the use case of Orona. However, performance metrics can be found in many other CPS applications, including those from the automotive [28, 65] or aerospace domains [43]. Another line of research could be the application of the Performance Oracle in other CPS domains.

Bibliography

[1] Adeptness project webpage: https://www.adeptness.eu/.

[2] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11), 2018.

[3] Pekka Abrahamsson, Goetz Botterweck, Hadi Ghanbari, Martin Gilje Jaatun, Petri Kettunen, Tommi J. Mikkonen, Anila Mjeda, Jürgen Münch, Anh Nguyen Duc, Barbara Russo, and Xiaofeng Wang. Towards a Secure DevOps Approach for Cyber-Physical Systems. *International Journal of Systems and Software Security and Protection*, 11(2):38–57, 2020.

[4] Milad Taleby Ahvanooey, Qianmu Li, Ming Wu, and Shuo Wang. A survey of genetic programming and its applications. *KSII Trans. Internet Inf. Syst.*, 13(4):1765–1794, 2019.

[5] Liwei An and Guang-Hong Yang. Attack detectability and stealthiness in distributed optimal coordination of cyber-physical systems. 66(9), 2023.

[6] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A surveyon testing for cyber physical system. *In IFIP International Conferenceon Testing Software and Systems*, 2015.

[7] Jon Ayerdi, Aitor Garciandia, Aitor Arrieta, Wasif Afzal, Eduard Enoiu, Aitor Agirre, Goiuria Sagardui, Maite Arratibel, and Ola Sellin. Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 280–290. IEEE, 2020.

[8] Jon Ayerdi, Sergio Segura, Aitor Arrieta, Goiuria SagarduiMaite Arratibel, and Maite Arratibel. Qos-aware metamorphic testing: An elevation case

study. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 104–114. IEEE, 2020.

[9] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1264–1274, 2021.

[10] Mohammad Azad, Igor Chikalov, Shahid Hussain, and Mikhail Moshkov. Minimizing depth of decision trees with hypotheses. In *Rough Sets: International Joint Conference, IJCRS 2021, Bratislava, Slovakia, September 19–24, 2021, Proceedings*, pages 123–133. Springer, 2021.

[11] Sreram Balasubramaniyan, Seshadhri Srinivasan, Furio Buonopane, B. Subathra, Jüri Vain, and Srini Ramaswamy. Design and verification of Cyber-Physical Systems using TrueTime, evolutionary optimization and UPPAAL. *Microprocessors and Microsystems*, 42(2016):37–48, 2016.

[12] A. Banerjee, K. K. Venkatasubramanian, T. Mukherjee, and S. K. S. Gupta. Ensuring safety, security, and sustainability of mission-critical cyber–physical systems. *Proceedings of the IEEE*, 100(1):283–299, 2012.

[13] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th international conference on software engineering companion*, pages 789–792, 2016.

[14] Andreas Brunnert, Andre van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, Anne Koziolek, Johannes Kroß, Simon Spinner, Christian Vögele, Jürgen Walter, and Alexander Wert. Performance-oriented DevOps: A Research Agenda. 2015.

[15] Wing Kwong Chan, Jeffrey CF Ho, and TH Tse. Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, 20(2):89–120, 2010.

[16] Lawrence Chung and Julio Cesar Sampaio Do Prado Leite. On non-functional requirements in software engineering. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5600 LNCS:363–379, 2009.

[17] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. Performance mutation testing. *Software Testing Verification and Reliability*, 2020.

[18] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. *Proceedings of the Fourth International Workshop on Software and Performance, WOSP'04*, pages 94–103, 2004.

[19] Subhasri Duttagupta and Manoj Nambiar. Performance extrapolation using load testing results. *Proceeding of International Journal of Simulation Systems, Science & Technology*, pages 66–74, 2008.

[20] Vincenzo Ferme and Cesare Pautasso. Towards holistic continuous software performance assessment. *ICPE 2017 - Companion of the 2017 ACM/SPEC International Conference on Performance Engineering*, pages 159–164, 2017.

[21] Nicolas Ferry and Phu H. Nguyen. Towards model-based continuous deployment of secure IoT systems. In *Proceedings - 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2019*, pages 613–618. Institute of Electrical and Electronics Engineers Inc., sep 2019.

[22] Afonso Fontes and Gregory Gay. Using machine learning to generate test oracles: a systematic literature review. In *Proceedings of the 1st International Workshop on Test Oracles*, pages 1–10, 2021.

[23] Ahmet Esat Genç, Hasan Sözer, M Furkan Kıraç, and Barış Aktemur. Advisor: An adjustable framework for test oracle automation of visual output systems. *IEEE Transactions on Reliability*, 2019.

[24] Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. Enabling model testing of cyber-physical systems. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 176–186, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Charles Hartsell, Abhishek Dubey, Nagabhushan Mahadevan, Theodore Bapty, Shreyas Ramakrishna, and Gabor Karsai. Demo abstract: A CPS toolchain for learning-based systems. *ICCPS 2019 - Proceedings of the 2019 ACM/IEEE International Conference on Cyber-Physical Systems*, pages 342–343, 2019.

[26] Priyanka S Helode, Dr. K. H. Walse, and Karande M.U. An Online Secure Social Networking with Friend Discovery System. *International Journal of Innovative Research in Computer and Communication Engineering*, 5(4):8198–8205, 2017.

[27] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[28] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. Quality metrics and oracles for autonomous vehicles testing. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 194–204. IEEE, 2021.

[29] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 71–82, 2018.

[30] Mohammad Ali Javidian, Pooyan Jamshidi, and Marco Valtorta. Transfer Learning for Performance Modeling of Configurable Systems: A Causal Analysis. pages 497–508, 2019.

[31] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–87, 2012.

[32] Muhyiddine Jradi, Krzysztof Arendt, Fisayo Caleb Sangogboye, Claudio Giovanni Mattera, Elena Markoska, Mikkel Baun Kjærgaard, Christian T Veje, and Bo Nørregaard Jørgensen. Obepme: An online building energy performance monitoring and evaluation tool to reduce energy performance gaps. *Energy and Buildings*, 166:196–209, 2018.

[33] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[34] Alexios Koutsoukas, Keith J Monaghan, Xiaoli Li, and Jun Huan. Deep-learning: investigating deep neural networks hyper-parameters and comparison of performance to shallow methods for modeling bioactivity data. *Journal of cheminformatics*, 9(1):1–13, 2017.

[35] Richard Lai, S Mahmood, R Lai, and Y S Kim. Survey of component-based software development. *The Institution of Engineering and Technology*, 3(May 2007):58–64, 2014.

[36] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.

[37] Rakesh Kumar Lenka, Pranali Bhanse, and Utkalika Satapathy. Load Performance Testing on Cloud Platform. *Proceedings - IEEE 2018 International Conference on Advances in Computing, Communication Control and Networking, ICACCCN 2018*, pages 414–419, 2018.

[38] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Microsoft Azure, Peng Huang, Johns Hopkins University, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Microsoft Research, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[39] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Using black-box performance models to detect performance regressions under varying workloads: an empirical study. *Empirical Software Engineering*, 25:4130–4160, 2020.

[40] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. *FOREPOST: finding performance problems automatically with feedback-directed learning software testing*, volume 22. 2017.

[41] Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P. Sheth. Machine learning for internet of things data analysis: a survey. *Digital Communications and Networks*, 4(3):161–175, 2018.

[42] Elena Markoska and Sanja Lazarova-Molnar. Towards smart buildings performance testing as a service. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 277–282. IEEE, 2018.

[43] Claudio Menghi, Enrico Viganò, Domenico Bianculli, and Lionel C Briand. Trace-checking cps properties: Bridging the cyber-physical gap. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 847–859. IEEE, 2021.

[44] Henry Muccini, Mohammad Sharaf, and Danny Weyns. Self-adaptation for cyber-physical systems: a systematic literature review. In *Proceedings of the 11th international symposium on software engineering for adaptive and self-managing systems*, pages 75–81, 2016.

[45] Dmitry Namiot and Manfred Sneps-Sneppe.

[46] Phu H. Nguyen, Nicolas Ferry, Gencer Erdogan, Hui Song, Stéphane Lavirotte, Jean Yves Tigli, and Arnor Solberg. A systematic mapping study of deployment and orchestration approaches for IoT. In *IoTBDS 2019 - Proceedings of the 4th International Conference on Internet of Things, Big Data and Security*, pages 69–82. SciTePress, 2019.

[47] Rory V. O'Connor, Peter Elger, and Paul M. Clarke. Continuous software engineering—A microservices architecture perspective. *Journal of Software: Evolution and Process*, 29(11), 2017.

[48] Zina Oudina, Makhlouf Derdour, and Mohammed Mounir Bouhamed. Testing cyber-physical production system: Test methods categorization and dataset. In *2022 4th International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–8. IEEE, 2022.

[49] Michael O'Neill. Riccardo poli, william b. langdon, nicholas f. mcphee: A field guide to genetic programming: Lulu. com, 2008, 250 pp, isbn 978-1-4092-0073-4, 2009.

[50] Nenad Petrovic and Milorad Tosic. SMADA-Fog: Semantic model driven approach to deployment and adaptivity in fog computing. *Simulation Modelling Practice and Theory*, 2019.

[51] Subhav Pradhan, William R. Otte, Abhishek Dubey, Aniruddha Gokhale, and Gabor Karsai. Towards a resilient deployment and configuration infrastructure for fractionated spacecraft. *ACM SIGBED Review*, 10(4):29–32, 2013.

[52] Antoine Proulx, Francis Raymond, Bruno Roy, and Fabio Petrillo. Problems and Solutions of Continuous Deployment: A Systematic Review. 2018.

[53] Sampath Kumar Veera Ragavan and Madhavan Shanmugavel. Engineering cyber-physical systems-Mechatronics wine in new bottles? *2016 IEEE International Conference on Computational Intelligence and Computing Research, ICCIC 2016*, (1), 2017.

[54] A. Ramachandran, K. Gayathri, Ahmed Alkhayyat, and Rami Q. Malik. Aquila optimization with machine learning-based anomaly detection technique in cyber-physical systems. *Computer Systems Science and Engineering*, 46(2):2177 – 2194, 2023.

[55] Stefan Reif, Andreas Schmidt, Timo Hönig, Thorsten Herfet, and Wolfgang Schröder-Preikschat. δelta: differential energy-efficiency, latency, and timing analysis for real-time networks. *ACM SIGBED Review*, 16(1):33–38, 2019.

[56] Reza Matinnejad, Shiva Nejati, Lionel C Briand and Thomas Bruck-mann. Test generation and test prioritization for simulink modelswith dynamic behavior. *IEEE Transactions on Software Engineering*, pages 919–944, 2018.

[57] Luis F Rivera, Norha M Villegas, Gabriel Tamura, Miguel Jiménez, and Hausi A Müller. UML-driven Automated Software Deployment. *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 257–268, 2018.

[58] K. Sampigethaya and R. Poovendran. Aviation cyber–physical systems: Foundations for future aircraft and air transport. *Proceedings of the IEEE*, 101(8):1834–1855, 2013.

[59] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of Cyber-Physical Systems. *2011 International Conference on Wireless Communications and Signal Processing, WCSP 2011*, (November 2011), 2011.

[60] Chi-Sheng Shih, Jyun-Jhe Chou, Niels Reijers, and Tei-Wei Kuo. Designing CPS/IoT applications for smart buildings and cities. *IET Cyber-Physical Systems: Theory & Applications*, 1(1):3–12, 2016.

[61] Siddhartha Kumar Khaitan and James D McCalley. Design techniquesand applications of cyberphysical systems: A survey. *IEEE SystemsJournal*, pages 350–365, 2014.

[62] Zhen Song, Philippe Labalette, Robin Burger, Wolfram Klein, Sudev Nair, Suhas Suresh, Ling Shen, and Arquimedes Canedo. Model-based cyber-physical system integration in the process industry. *IEEE International Conference on Automation Science and Engineering*, 2015-Octob(September 2016):1012–1017, 2015.

[63] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1178–1189, 2020.

[64] Vijay Vaishnavi, Bill Kuechler, and Stacie Petter. Design Science Research in Information Systems. 2004.

[65] Pablo Valle. Metamorphic testing of autonomous vehicles: a case study on simulink. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 105–107. IEEE, 2021.

[66] Zhuping Wang, Haoyu Shen, Hao Zhang, Sheng Gao, and Huaicheng Yan. Optimal dos attack strategy for cyber-physical systems: A stackelberg game-theoretical approach. *Information Sciences*, 642, 2023.

[67] Elaine J. Weyuker. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26:1147–1156, 2000.

[68] Zhijing Ye, Zheng O'neill, and Fei Hu. Hardware-based emulator with deep learning model for building energy control and prediction based on occupancy sensors' data. *Information (Switzerland)*, 12(12), 2021.

[69] Shenglin Zhang, Ying Liu, Dan Pei, Yu Chen, Xianping Qu, Shimin Tao, Zhi Zang, Xiaowei Jing, and Mei Feng. FUNNEL: Assessing Software Changes in Web-Based Services. *IEEE Transactions on Services Computing*, 11(1):34–48, 2018.

[70] Weiyan Zhang, Mehran Goli, Alireza Mahzoon, and Rolf Drechsler. Ann-based performance estimation of embedded software for risc-v processors. volume 2022-October, page 22 – 28, 2022.

[71] Nengwen Zhao, Junjie Chen, Zhaoyang Yu, Honglin Wang, Jiesong Li, Bin Qiu, Hongyu Xu, Wenchi Zhang, Kaixin Sui, and Dan Pei. Identifying bad software changes via multimodal anomaly detection for online service systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 527–539, 2021.

[72] Indrė Žliobaitė, Mykola Pechenizkiy, and Joao Gama. An overview of concept drift applications. *Big data analysis: new algorithms for a new society*, pages 91–114, 2016.

A

Publications

A.1 Towards a Taxonomy for Eliciting Design-Operation Continuum Requirements of Cyber-Physical Systems

This paper was presented at the IEEE International Requirements Engineering Conference (RE) in 2020 and then published in the conference proceedings. The full citation:

Ayerdi, J., Garciandia, A., Arrieta, A., Afzal, W., Enoiu, E., Agirre, A., Sagardui, G., Arratibel, M. & Sellin, O. (2020, August). Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In 2020 IEEE 28th International Requirements Engineering Conference (RE) (pp. 280-290). IEEE.

Towards a Taxonomy for Eliciting Design-Operation Continuum Requirements of Cyber-Physical Systems

Jon Ayerdi*, Aitor Garciandia†, Aitor Arrieta*, Wasif Afzal‡, Eduard Enoiu‡, Aitor Agirre†,
Goiuria Sagardui*, Maite Arratibel§ and Ola Sellin¶
University of Mondragon*, Ikerlan †, Mälardalen University‡, Orona§, Bombardier Transportation¶,
*{jayerdi,aarrieta,gsagardui}@mondragon.edu, †{agarciandia, aagirre}ikerlan.es, ‡{wasif.afzal,
eduard.paul.enoiu}@mdh.se, §marratibel@orona-group.com, ¶ola.sellin@rail.bombardier.com

***Abstract*—Software systems that are embedded in autonomous Cyber-Physical Systems (CPSs) usually have a large life-cycle, both during its development and in maintenance. This software evolves during its life-cycle in order to incorporate new requirements, bug fixes, and to deal with hardware obsolescence. The current process for developing and maintaining this software is very fragmented, which makes developing new software versions and deploying them in the CPSs extremely expensive. In other domains, such as web engineering, the phases of development and operation are tightly connected, making it possible to easily perform software updates of the system, and to obtain operational data that can be analyzed by engineers at development time. However, in spite of the rise of new communication technologies (e.g., 5G) providing an opportunity to acquire Design-Operation Continuum Engineering methods in the context of CPSs, there are still many complex issues that need to be addressed, such as the ones related with hardware-software co-design. Therefore, the process of Design-Operation Continuum Engineering for CPSs requires substantial changes with respect to the current fragmented software development process. In this paper, we build a taxonomy for Design-Operation Continuum Engineering of CPSs based on case studies from two different industrial domains involving CPSs (elevation and railway). This taxonomy is later used to elicit requirements from these two case studies in order to present a blueprint on adopting Design-Operation Continuum Engineering in any organization developing CPSs.**

***Index Terms*—DevOps, Design-Operation, Requirements Elicitation, Cyber-Physical Systems**

I. Introduction

Cyber-Physical Systems (CPSs) integrate computation with physical processes whose behavior is defined by both physical and software parts of the system [1]. While the cyber-physical controller consists of discrete software, the physical layer is composed of parallel physical processes running in continuous time. The cyber layer is composed of computational platforms and networks that are in charge of monitoring and controlling physical processes [31]. These systems are part of many products we use in our daily life, such as vehicles, airplanes, elevators and trains. As the lifecycle of these systems is rather long, all their components require maintenance, including their software components. Given that the software of these systems is usually extremely complex, the software constantly evolves during the CPS lifecycle based on several aspects [18], such as (1) new functional and non-functional requirements, (2) hardware obsolescence and/or system degradation, and (3) correction of bugs detected while the system is operating.

In the last few years, there have been several improvements in terms of modeling and simulation techniques [2], [15], [20] to develop and validate complex CPSs from the early development stages. However, when the software is deployed in the CPS, the methods used during operation and maintenance do not have synergies with the methods used in development. In other contexts, such as web-engineering, there are design-operation continuum engineering methods such as DevOps that permit software development methods to be streamlined with methods for operation time. DevOps practices efficiently integrate development and operations, aiming at shortening the lead time between a change request and the deployment in production using automation, agile software development and continuous delivery (CD) pipelines. Yet, for CPSs, traditional design-operation continuum engineering methods require substantial changes in order to be dependable enough. More specifically, design-operation continuum methods must provide solutions in order to have a more efficient process which guarantees that (1) software updates are performed safely and securely, (2) most of the faults are detected in the design phase before the software is deployed in the CPS and (3) problems that can emerge in operation can be reproduced in development in order to analyse and propose potential solutions.

In order to start developing design-operation continuum engineering methods supported by the appropriate tools for CPSs, a taxonomy of relevant concepts is expected to ease the understanding of their rather complex development and maintenance process. In this paper, we build and instantiate such a taxonomy in order to assist requirements analysts with the identification and categorization of the requirements related to different aspects of the CPS Design-Operation Continuum Engineering. The main purpose of this classification is supporting the elicitation of new requirements and the easier identification of problems such as omissions, ambiguity, vagueness, conflicts or duplication in the requirements. Furthermore, this classification is also helpful for determining the organisational roles responsible for each requirement, as well as for the management and reuse of the elicited requirements in later stages of the development lifecycle. This taxonomy is inspired by case

studies from two different industrial domains: the elevation domain and the railway domain. Both case studies are provided by companies that are leaders in their sectors. By analyzing the data provided by these companies through their documentation (e.g., internal technical documents, repositories, code, etc), as well as through interviews of their engineers, we were able to develop a general purpose taxonomy for design-operation continuum engineering methods for CPSs. With our taxonomy, organizations can instantiate their domain-specific categorization and classification of requirements in order to adopt design-operation continuum methods for the development of CPSs.

The remainder of this paper is structured as follows: Section II describes the taxonomy development process. Section III presents the two industrial case studies that inspired this work, and are also the first systems where this taxonomy is used to elicit requirements for their design-operation continuum. Section IV describes the developed taxonomy in detail. Section V describes the process of requirements elicitation using the taxonomy. Section VI reviews related work and Section VII concludes the paper.

II. Taxonomy Definition Method

To develop the taxonomy, we followed the guidelines proposed by Ralph [24], which provide a set of steps and certain options for each of the steps. The first step refers to *choosing the strategy*. We opted for using the "grounded theory and interpretative case study" approach as our main strategy, in addition to personal experiences acquired by the long-term collaboration between the industrial and academic partners involved in this paper. In this case, we analyzed two case studies from different domains, which permitted us to identify both commonalities and differences between them. We do not expect that our taxonomy can be generalized to all CPSs by using only two industrial case studies as a basis, but at least, we believe that it provides fundamental evidence that it could be adopted for many complex and industry-relevant cases. On the other hand, we unavoidably made use of personal experience to an extent in order to develop this taxonomy, so a certain degree of bias can be expected.

The second step is the *site selection*. Two sites were selected for developing the taxonomy, which are two organizations of CPS developers: Orona (from elevation domain) and Bombardier (from railway transportation domain). We chose these sites due to several reasons. Firstly, there is a long-standing collaboration between the researchers and practitioners that work for these organizations. Secondly, both sites are developers of complex CPSs. Thirdly, the domains are sufficiently different (i.e., elevation and railway) to ensure a minimum degree of heterogeneity for the development of a general taxonomy. Lastly, and most importantly, both sites are relevant target users for the taxonomy, they produce rich data and detailed explanations for their requirements, and are accessible to the authors of this paper.

As for the *data collection*, which is the third step proposed in the guidelines [24], two processes were followed. On the one hand, the direct observation methodology [24] was employed. To this end, we had access to internal documentation provided by both Orona and Bombardier. On the other hand, we interviewed participants from the companies involved. To this end, we prepared a set of relevant questions carefully selected by the researchers. We later interviewed practitioners from Orona that would directly benefit from adopting design-operation continuum methods into their development processes. These developers had various positions and experience levels. As we were creating a taxonomy from scratch, similar to [13], the interview questions had to be as generic and open-ended as possible. Therefore, we opted for semi-structured interviews [28], which combine open-ended questions with specific questions. Thus, the interviewer had to improvise new questions based on the interviewee's response. Additionally, we used internal documentation from both industrial companies, including test plans, comments from the code repository, standards, etc. More information related to the data collection from each of the case studies is given in Section III.

The fourth step is related to the *data analysis* [24]. We took notes based on (1) the interviews to engineers working in Orona, and (2) by accessing internal documentation at Orona and Bombardier. We then used an iterative approach to code our notes and build the taxonomy. We initially developed a first structure of the taxonomy by having reviewed the state-of-the-art on design-operation continuum methods. We later evolved this initial taxonomy with the information extracted from the interviews as well as the internal documentation of the case companies.

The last step refers to the *conceptual evaluation*. Similar to [13], in order to ensure that the final taxonomy was comprehensive and representative, we validated the taxonomy by involving both researchers and industrial participants. These were different from those involved in the interviews. The participants were asked to (1) identify potential weaknesses of the theory, as suggested by Ralph [24], and (2) provide evaluation criteria based on the credibility and transferability of the taxonomy.

III. Case Studies

In this section, we describe the two industrial case studies used to extract the taxonomy. One of the case studies is from the vertical transport (elevation) domain, whereas the other one is from the railway domain. Both companies that provide the case studies are leaders in their domain, and therefore, the technology that they use is cutting-edge. In this section, we explain the subsystems considered to build the taxonomy of the paper and its current software development process, the specific methodology followed in each of the case studies for developing it, and how both companies expect to improve their software development process by adopting design-operation continuum methods.

A. Case study from the elevation domain

Orona's activities are focused on the design, manufacturing, installation, maintenance, and modernisation of elevators,

escalators, and moving ramps and walkways. An elevator installation is a complex CPS composed by a set of elevators that interact to provide service to passengers with the goal of minimising the Average Waiting Time (AWT) and, more recently, also taking into account other criteria such as energy consumption, transport capacity, or overall transit time. Nowadays, over 250.000 elevators worldwide use Orona's technology. As most of the new functionality in elevators' installations is provided by software, Orona has a systematic and well established process for the development and release of new software versions.

The traffic master manages the passenger flow. It is composed by several software modules such as the dispatcher, which executes the traffic algorithm to allocate calls to elevators, the signalling to guide passengers (e.g., by communicating the assigned elevator), or the access control which disables specific floors for unauthorized passengers. The traffic master is constantly evolving in order to improve the service by including new functionalities or adapting to the building requirements. In conclusion, this system is a good candidate for the adoption of design-operation continuum methods.

Interviews to the dispatcher manager, two software engineers and a system validation engineer were carried out by the researchers. The dispatcher manager defines new functionalities and analyzes poor performance in installations. The software engineers develop and validate the software. Lastly, the validation engineer tests the dispatcher in the Elevator.

Figure 1 illustrates the process (current tasks, roles and tools) extracted from the analysis of the data collected during the interviews. Within the requirement elicitation of a new release (1), a rigorous validation plan is defined comprising three main validation phases. The *Software-in-the-Loop (SiL)* phase usually encompasses most of the development work for a new functionality. The software produced in this step (2) (depicted as system under test (SUT)) is validated (3) in a purely virtual environment using a domain specific simulator (i.e., Elevate™).[1] At the SiL phase, tests ensure the quality of service requirements (e.g., AWT over time). The *Hardware-in-the-Loop (HiL)* phase (4-5-6) follows the previous SiL phase. In this phase, both virtual and real components are mixed together to compose an integrated scenario that is very close to the real one. Simulators that are used at the SiL phase are substituted by real hardware (e.g., elevator controllers) and communication networks, enabling integration tests of the entire system. Nevertheless, some parts may still be simulated (e.g., elevator shaft simulator, passenger demand, etc.). At the HiL phase, test cases check the functional correctness of the release. Finally, the software is deployed into the real system, *operational phase* (7-8), and eventually monitored by maintenance staff (9). Some installations require a deep analysis in order to understand the perception of the customers (10). This analysis is performed by trying to reproduce the situations observed in reality in simulations at the SiL phase.

[1]https://www.peters-research.com/index.php/elevate

Executing the validation plan both at SiL and HiL phases follows a similar sequence: (a) deploy to validation infrastructure; (b) configure the context (i.e., the type of building, number of floors, etc.); (c) define and configure data to monitor; (d) execute the test cases; (e) analyze the data; (f) decide whether the version is ready for the next phase. Configuration, deployment, analysis and decision are mainly manual steps and the execution of test cases at HiL is semi-manual. Therefore, these tasks are error-prone, require significant effort and are dependent on specific knowledge and skills. This is especially exacerbated at the HiL phase, where a configuration of the validation infrastructure requires especial knowledge and can take hours to ensure a proper configuration and deployment. Besides, test cases are executed in real-time, and thus, on top of the test execution times being potentially long, the availability of an engineer is required during this process.

Once in operation, feedback for new requirements or bug fixes is received from (a) customers, (b) monitors of the CPS and (c) regulation changes. Checking customers' feeling of poor performance about the installation (i.e., the feeling that some passengers are waiting too long) by reproducing the scenario in the domain-specific simulator is always a time consuming and cumbersome task. Sometimes, an in-situ monitoring process of the installation is required for a limited period of time, which is extremely costly. Besides, it is not always possible to reproduce the situation, and therefore, a deep analysis to identify differences has to be performed.

Design-operation continuum methods could automate several tasks of a software release, from SiL to operation and from operation to SiL. The ultimate goal of Orona can be summarized in the following points: (1) Automatically configure validation test benches at the SiL and HiL phases, reducing the number of errors and the time to configure a validation context; (2) Automate the deployment of new software releases to the SiL, HiL and real elevators worldwide, considerably reducing the costs; (3) Automate the execution and evaluation of test cases at the HiL phase; (4) Automate the validation of software releases by using streamlined test oracles that can be re-used across all levels (i.e., SiL, HiL and Operation); (5) Automatically collect data during operation, which will enable reproducing real-life scenarios at design-time (i.e., SiL and HiL).

Achieving these objectives would result in an improvement of the software development practice and a reduction of the overall cost of releasing a new software version. The software quality would be improved by using real data from operation to identify realistic situations at design-time and to detect potential bottlenecks by monitoring the quality of service across the different software releases. In addition, the quality would also benefit from increasing the likelihood and frequency of detecting bugs both at design-time and at operation-time by using streamlined test oracles at all the levels.

B. Case study from the railway domain

Bombardier Transportation (BT) is one of the leading companies in the rail industry, providing rolling stock and

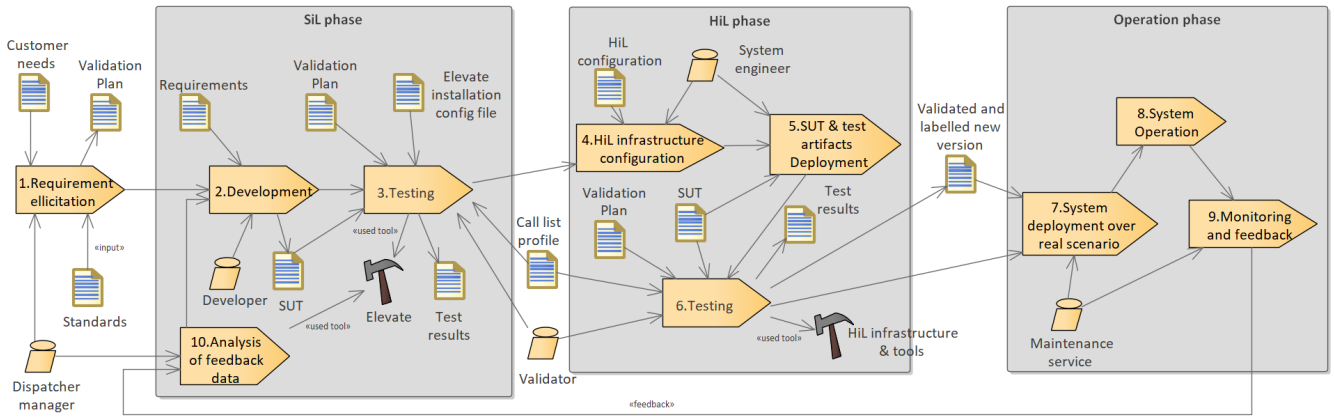


Fig. 1. Current software development process at Orona

associated services of system maintenance, signalling, fleet management and asset life management. It has a broad and innovative product portfolio in rolling stock, consisting of e.g., light rail vehicles, metros, commuter and regional trains. For the definition of design-operation continuum requirements, we have selected BT's Train Control and Management System (TCMS). This system is the centre of the distributed system that controls the train. It is involved in almost all train functions, either in a controlling or a supervisory capacity. Examples of train functions controlled and managed by the TCMS are collecting line voltage, controlling the train engines, opening and closing the train doors and upload of diagnostic data. For collecting data on BT's current status of the development of TCMS as well as finding out opportunities of design-operation continuum methods for testing and deployment, we made use of archival data as one of our first steps. We had access to internal documentation of a relevant BT project. Documentation such as test plans were read and analyzed. We used document analysis [4] as a systematic procedure for reviewing or evaluating these documents. Two experienced researchers, with extensive experience in research projects with BT, were involved in the analysis.

At BT, a test plan documents the scope, approach, resources and schedule of the testing activities per project. The deployment plan is also partly reflected in the test plan. The test plan covers the detailed planning regarding the three levels of tests for TCMS: software component test, function test and system test. The test plan also mentions the PASS/FAIL criteria for the three levels of tests. The plan touches upon the deployment in terms of environmental/infrastructure needs. The three levels of tests are performed in MiL, SiL and HiL simulators. Component tests, functional tests are typically performed in the domain-specific MiL/SiL simulator while system tests are typically performed in the HiL simulator. Fig. 2 shows the development process at TCMS with corresponding simulation levels. The execution of test plan at MiL, SiL and HiL follows a somewhat similar sequence: 1) prepare software component test/function test/system test infrastructure; 2) develop software components/features; 3) develop component/function/system test cases; 4) build and deploy on test bench; 5) execute tests; 6) record defects (if any); 7) generate test report; 8) release software when no critical defects remaining. This is shown in Fig. 3. Many activities in this process require manual interventions such as setting up of test environment and activities around testing at different simulation levels.

In testing TCMS, the engineering processes of software development (including requirement engineering and testing) are performed according to safety standards and regulations [6]. Specification-based testing is mandated by the EN 50128 standard to be used to design test cases. Each test case should contribute to the demonstration that a specified requirement has indeed been covered and satisfied. Executing test cases on TCMS is supported by a test framework that includes the comparison between the expected output with the actual outcome. Testing at the functional level is done against TCMS design requirements. The created test suites are based on functional specifications expressed in a natural language. Stimulation and responses are checked at the interfaces at the functional level. The test cases are composed by test steps, which define a stimulation and the expected output. Test case design is done in a scripting language and in a test management tool. As long as the test management tool is not fully operational, alternatively the test case design can be done in word documents or by comments in the test scripts.

A streamlined process following the design-operation continuum methods is expected to bring optimizations in the TCMS development process at BT from vehicle validation and operation to MiL testing. This work shall focus on (1) efficient management of product variants throughout the testing process, (2) use of requirements at different levels of abstractions for validating the obtained designs. For example, the software architecture specification can provide a solid foundation for developing a plan for testing at this level and testers can use architecture-based test models, criteria, and techniques tailored to the railway domain. Architectural (integration)-based tests, developed using these functional models, can be used to assess the architecture itself or to test the software component design conformance with the architecture. This methodology aims at detecting defects earlier in the development lifecycle

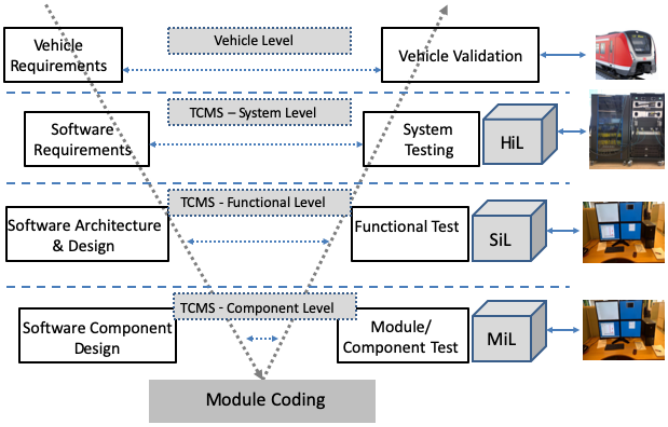


Fig. 2. Development process for TCMS at BT.

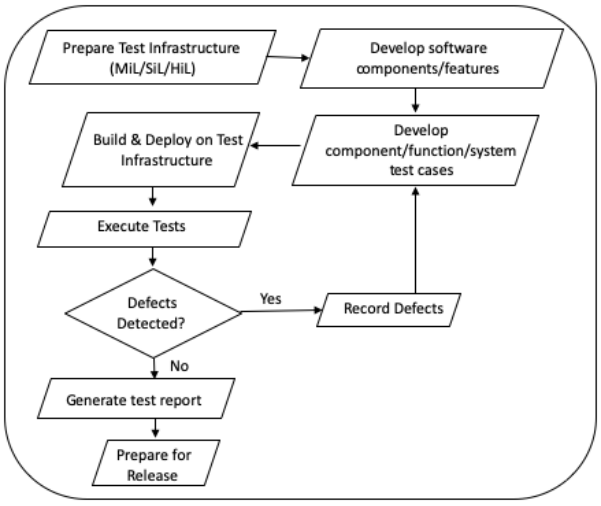


Fig. 3. Execution of test plan at MiL/SiL/HiL levels at BT.

and reducing testing costs at vehicle, system, functional and component levels. Each of the abstraction levels has a specific role in the testing process. By modelling these requirements, we can provide means to generate test cases using the whole lifecycle information.

IV. Taxonomy for Design-Operation Continuum Methods of CPSs

Taxonomies can be organized following one of the two main classification approaches: enumerative or faceted [27]. *Enumerative approaches* utilize a fixed set of predefined classes, which results in simple and easy to apply classification schemes, but may not be appropriate for unexplored domains where the knowledge base is still unstable or incomplete. *Faceted approaches*, on the other hand, allow the classification of entities based on multiple perspectives, which provides a flexibility that is useful when developing taxonomies for immature domains [32]. In this work, we followed a faceted classification structure, since the design-operation continuum, particularly when applied to CPSs, is an emerging domain that has not been fully defined yet. The final taxonomy contains four facets which are orthogonal to each other, three of which contain a single level of categories and the last of which contains several levels of sub-categories. The full taxonomy is shown in Figure 4.

A. Lifecycle stage

This facet represents the X-in-the-loop system execution level, which is an aspect specific to CPSs development processes. Requirements may be applicable to one or more of these classes. In this work, we define only the four classes which we identified as relevant for both of the analyzed case studies. For CPSs, although we believe that this classification is general, there could be cases where other levels might need to be defined. For instance, to the knowledge of the authors, there are companies where the MiL phase is split into several sub-phases with different fidelity levels. A few years ago, there was a common step between SiL and HiL, named as Processor-in-the-Loop phase, which had as a main goal to detect potential inconsistencies that the compiler could introduce [29]. However, to the best of our knowledge, this step is not commonly used any longer and could be challenging to apply in the context of CPSs, as several processors are involved. Furthermore, as this phase was not used in the industrial case studies used to build our taxonomy, we did not include it.

At the **MiL** stage, the CPS, including the software, hardware and environment, is executed as a model by a model execution software. This setup allows the early and easy detection of failures in a controlled environment, but there are several types of errors that cannot be observed at this level, e.g., communication errors.

At the **SiL** stage, the CPS software is run on a simulated environment. The use of the real software allows the detection of several errors that could not be detected in MiL, such as those related with arithmetic precision. Nevertheless, not all the software errors can be observed yet, since the processor where the SiL runs is often different from the processor in the real hardware.

At the **HiL** stage, the CPS software is deployed on the real hardware, but within a controlled environment, such as a test bench. Since physical hardware is involved, the CPS execution must be real-time, which makes it much more costly than the previous stages. This execution level allows the detection of many new classes of errors, such as timing or hardware interaction issues, which were not observable in previous stages.

The **Operation** is the stage where systems are deployed in real environments, possibly in production. We distinguish this stage from HiL because usually intrusive testing can no longer be performed at this level, since the CPS is already running in real scenarios. Nevertheless, some non-intrusive validation techniques such as Runtime Verification can still be performed.

B. Scope

We distinguish three different scope classes depending on the applicability of the requirement. The significance of this

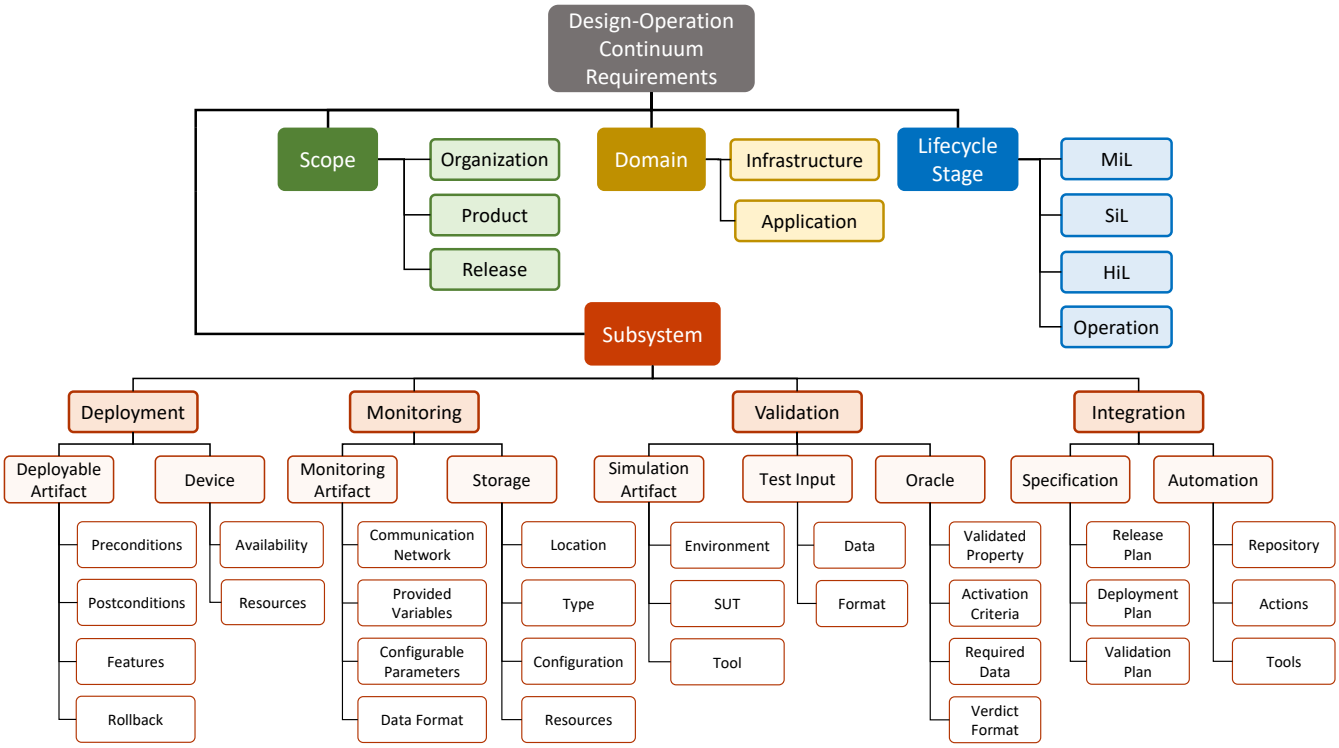


Fig. 4. Taxonomy of Design-Operation Continuum Requirements for CPSs

facet is that it enables the reuse of the requirements throughout their applicable scope. Depending on the product strategy of a company, the categorization we provide can be refined or extended beyond the three classes we define. For example, we could have a family of products that share some common requirements, in which case a more fine-grained classification, such as a feature-level scope, would be useful.

Organization refers to the requirements that will be reused for the Design-Operation Continuum of all the products of the organization. For example, this category may include requirements such as the deployment subsystem being able to copy files to a target device running an SSH server.

Product refers to the requirements that are specific of a particular product, which can therefore be reused across all of its releases. For instance, this would include being able to automatically launch the simulators used for the test execution of an elevator dispatcher.

Release refers to the requirements specific to a particular software release of a product, which we differentiate from requirements applicable to all its releases. For instance, the verification of an optional feature of a product belongs to this category, as it is only applicable to some specific releases.

C. Domain

This facet categorizes the requirement by the domain in which it belongs, which we divide into two sub-categories. This categorization facilitates the assignment of the requirements within the organization between two different roles, usually IT department and development team.

- **Infrastructure**. On the one hand, we identify the requirements related with the Design-Operation Continuum infrastructure itself, which address concerns such as the monitoring of the infrastructure elements (e.g., deployment progress, status of the deployed components, etc.). Within the organization, the roles responsible for these requirements may not be directly related with the development of CPS products, since they only concern the development infrastructure.
- **Application**. On the other hand, we consider the requirements related with the particular applications being developed (i.e., the CPS), such as the monitoring of the application itself (e.g., tracking the status of the elevators based on CAN messages, etc.). These requirements must be managed by the organizational roles working directly on the CPS products.

D. Subsystem

This facet classifies a requirement by the Design-Operation Continuum subsystem for which it is relevant. Our taxonomy considers the subsystems of Deployment, Monitoring, Validation and Integration.

1) Deployment: Automating deployment means providing the infrastructure that allows automated CI server to connect to the designated production/validation machine and upload executable and configuration files [21]. The continuous deployment subsystem allows the automatic deployment of a new software release in the virtual infrastructure for validation purposes. Afterwards, the new release is deployed in the real

CPS in operation. In this subsystem, requirements that are necessary to deploy artifacts at the MiL-SiL-HiL-Operation of the system are specified. It is important to mention that in this category, requirements for the Operation stage are the most demanding ones, since other aspects such as heterogeneous platforms or the status of the CPS before the deployment need to be considered. Examples of requirements obtained by our industrial case studies in this category include *"The deployment service shall provide support for ARMV7 boards", "The deployment service shall provide support for Linux and Windows", "The dispatcher down time during deployment shall be less than 15 sec" or "The system shall allow the deployment of artifacts by defining the allocation or by defining the memory requirements"*. Nowadays, releasing and deploying new software versions is a time-consuming and error-prone activity. Requirements in this category facilitate the automation of the continuous deployment for new releases. Two subcategories have been defined: Devices and Deployable Artifacts. Note that this category is closely related to the Continuous Integration category.

- *Devices*. **Description.** A CPS is composed by heterogeneous platforms. Automation of the deployment process in CPSs is highly complex due to the number of heterogeneous platforms, models and interfaces necessary to deploy software releases. The goal of this subcategory is to collect requirements related to the variety and heterogeneity of hardware, software and communications for which the deployment subsystem must provide support. To gather requirements, an analysis of the platforms of the CPSs shall be performed. These type of requirements will have influence on the deployment architecture that must be designed to provide support for all the devices in which an automatic deploy will be performed. This category also has impact on the techniques and methods used for the deployment, e.g., container based deployment mechanisms that are valid for Linux based devices are not for embedded bare metal devices. **Subcategories.** There are different aspects to be specified: (1) *Resources* of the devices. Hardware, software (e.g., installed OS) and communication networks (e.g., CAN, Ethernet) that the deployment subsystem is going to deal with. (2) *Availability* of the device during deployment (e.g., maximum downtime of the device to perform the deploy).
- *Deployable Artifacts*. **Description.** A CPS is composed of different software components distributed in heterogeneous devices. Deployable artifacts are "soft" components which are part of the CPS, such as software of new releases and configuration files. When using design-operation continuum methods, test oracles, monitors, etc. can also be considered deployable artifacts. This subcategory includes the specification of the features of the artifacts to be deployed. To gather requirements, an analysis of the software elements of the CPS shall be performed. Requirements in this category define the deployment rules, and are useful to ensure the pre and post deployment conditions and to design the rollback mechanisms. **Subcategories.** There are different aspects to be specified: (1) *Deployment conditions*: Pre-conditions specify criteria to be met before starting the deployment, e.g., *"the Elevator shall be out-of-service"*. Post-conditions are verified after the deployment is completed, e.g., *"the device reboots correctly"* (2) *Features of the artifacts*: hardware requirements, e.g., minimum CPU or RAM requirements to execute the artefact, software requirements, e.g., supported OS, communication interface requirements, e.g., to be deployed in a device with access to CAN or/and the allocation of the artifact, e.g., in which device shall be deployed, (3) *Rollback policy* in case of deployment failure, e.g., *"The system shall support the remote rollback to a previous version"*.

2) Monitoring: Continuous monitoring is a key process in design-operation continuum. The goal of this process is to extract data from a system so that it can be analyzed [17]. Monitoring in the deployment ensures that certain conditions are met before and after deploying. In the validation process, it provides data to the oracles so that they can provide a verdict. Besides, it can also be useful to observe and record the status of the infrastructure/application and later reproduce real scenarios in simulation. Examples of requirements in this category include *"Monitoring data from MiL/SiL/HiL test executions shall be available through logs", "Monitors shall provide connectors for CAN and Ethernet", "Monitoring data for the last day shall be persisted for further analysis"*.

This category facilitates gathering monitoring requirements at different lifecycle stages and levels of a CPS. Two subcategories have been defined:

- *Monitoring artifacts*. **Description.** Continuous monitoring can be done (1) at the infrastructure level, e.g., to control the CPU or memory usage or (2) at the application level, to monitor, for instance, the position and speed of an elevator. The goal of this subcategory is to collect monitoring needs of both the infrastructure and the application. To gather requirements, an analysis of the application data lifecycle and the infrastructure features (e.g., CPU usage) shall be performed. Requirements in this subcategory have impact on the design of the monitoring infrastructure. **Subcategories.** There are different aspects to be specified: (1) *Communication Network*, the source from which data must be collected, e.g., *"the monitor must gather the data from the CAN bus"*, (2) *Data fields* that will be provided, e.g., *"the monitor must provide the elevator positions"*, (3) *Format* in which the data will be provided by the monitor, e.g., *"the monitor will provide the current elevator position periodically via MQTT"*, and (4) *Configurable parameters* for the monitor,e.g., *"the update period for the elevator positions may be configured with a value between 50 and 500 milliseconds"*.
- *Storage*. **Description.** Storage of the monitored data is essential to analyze and reproduce scenarios in simula-

tion. The storage strategy may be different depending on the data being monitored. Some data could be more critical and other may need more memory resources. These requirements might include, for example, dumping data on a local file, storing it on the edge of the network, or sending it to a cloud database. The goal of Storage subcategory is to describe how the data shall be stored in order to be accessible from other services. To gather requirements, an analysis of the application data usage shall be performed. Requirements in this subcategory have impact on the design of the storage strategy for the data that is being monitored. **Subcategories.** There are different aspects to be specified: (1) *Location* describes where the data is to be persisted, e.g., a shared folder on a NAS or a database endpoint, (2) *Type* relates to the database format, either a relational database, an object oriented database or even text file based, (3) *Configuration* includes attributes such as duration of the saved data, backup replicas or even availability aspects), (4) *Resources* specifies the type of device used for persistence, as well as the disk space size.

3) Validation: Testing, verification and validation activities are important in any kind of domain when developing software. In the case of CPSs, this is particularly important because most of the functionality of these systems is driven by software. Furthermore, this functionality is often safety or mission-critical, and a failure could lead to severe consequences. Both industrial case studies used to build the taxonomy rely on simulation-based testing for the validation of software. This technique allows raising the level of abstraction of complex CPSs in which testing is performed [5]. It allows (1) executing larger test suites and (2) building test oracles that can automate verification and validation tasks [5]. Furthermore, simulation-based testing allows modelling the environment in which the CPS operates (e.g., in the case of the elevators, the interaction of the elevators with passengers). Test, verification and validation in design-operation continuum methods for CPSs needs to be practised from MiL phases through the Operation. This is because failures that could not be observed in previous stages can be identified in Operation. To this end, oracles need to be re-used across all these test levels to allow full automation. Examples of requirements obtained by focusing on the industrial needs of the case studies in this category include *"The SUT shall be the relevant version of the project-specific TCMS software", "The input to the test cases at the functional level shall be the stimuli triggering the execution of a defined functionality", "The oracles shall be activated by a test input or by identifying a precondition in operation"*. Note that the elicited requirements in this category shall provide the validation to be continuous and as automated as possible. To this end, three sub-categories were identified:

- *Simulation Artifact*. **Description.** This category concerns the artifacts that are necessary in order to enable simulation-based testing, which we divide in three main categories: **Subcategories.** (1) *Environment* refers to the conditions under which the system runs, which are usually expressed in the form of simulator parameters (e.g., the number of floors in the building); (2) the *SUT* is the component of the system that is being tested, which must usually comply with certain interfaces in order to be usable for simulation-based testing; and (3) the *Tool* is the simulator used to execute the SUT (e.g., Simulink). An example of a simulation artifact requirement for one of the industrial case studies is *"Test cases shall be executed by using the Elevate simulator"*.
- *Test Input*. **Description.** In order to drive the execution of the selected test cases, test inputs must be injected into the SUTs before or during their execution. We divide the requirements for these test inputs into two main categories. **Subcategories.** (1) The input data itself, which is determined by the test cases that need to be executed (e.g., must test having multiple passenger calls at the same time), and (2) The format that is used to define the test inputs (e.g., test inputs must be provided in a XML file which follows a specific structure).
- *Oracle*. **Description.** Test oracles are components in charge of emmitting a verdict (e.g., PASS/FAIL) based on the conformance of the system towards a specified property (e.g., for the Orona's dispatching system, the daily average waiting time per passenger shall be less than 30 seconds). Note that although monitoring the system is required for the validation, we classify monitoring as a separate subsystem, since monitoring is often performed beyond the context of system validation. The purpose of the oracles is to determine whether the observed behaviour of the system is correct or incorrect, which is usually done by verifying properties specified by a domain expert. An example of an elicited requirement for a test oracle is "test oracles shall be re-used across all levels (i.e., MiL, SiL, HiL and Operation)", or "test oracles shall be capable of validating 100% of functional requirements". **Subcategories.** Four sub-categories were identified based on the industrial case studies. (1) Validated properties are system's requirements themselves (e.g., $AWT < 30sec.$); (2) activation criteria are preconditions that trigger a test oracle to validate a specific property; (3) required data refers to the monitoring data needed by the oracle in order this to be able to check certain property; (4) the verdict format refers to the semantics provided by the verdict (e.g.,a quantitative value (e.g., a quantitative value from 0 to 1, with 1 meaning full compliance and the value becoming closer to 0 as the degree of compliance decreases).

4) Integration: Continuous Integration encompasses the subsystems to automate the pipeline from the development environment to the continuous deployment, monitoring, and validation subsystems. Automation is achieved by chaining different tasks together. The process involves software repositories, usage of adequate build tools, automated testing environments and testing tools, and deployment to operation.

Examples of requirements in this category include *"The deployment specification shall provide support to link an artifact to a device", "A validation specification shall allow to specify test cases at SiL and HiL level", "The source code shall be available from outside the company"*. This category facilitates gathering requirements related with the integration of all of the subsystems (deployment, validation and monitoring) into an automated pipeline. Two sub-categories have been defined:

- *Specification*. **Description.** A pipeline is a sequence of actions that have to be performed from the initial build of the project to the deployment of the real system. For this, different aspects must be specified, such a as the validations to be performed. In this category, requirements related to the specification of the sequence to be automated for the CI/CD scenario are provided. These requirements will be used to select or develop the CI/CD tool and the languages to specify the pipeline. **Subcategories.** There are different aspects to be specified: (1) *Release plan*: requirements for the language to specify the configuration and build process of the software. (2) *Deployment plan*: requirements for a language to specify all the steps related to the deployment. (3) *Validation plan*. Requirements for the language to specify the validation of the CPS, which is a critical step in the continuous integration process. This plan will describe the configuration, coordination and management of all the verification artifacts. For our case studies, for instance, we identified validation plan requirements such as being able to execute multiple instances of a system concurrently in order to compare their behaviour. All of these plans could be integrated into a single CI/CD plan. However, we have decided to classify these specification languages separately because different roles might be involved in the requirements elicitation for each of them.
- *Automation*. **Description.** Implementation of continuous integration or continuous deployment mechanisms depends on a series of tools that facilitate functions necessary to achieve fully automated operation. In this category, requirements that should be considered for automating the pipeline are defined.These requirements will be used to select the CI/CD tool and define the pipeline. **Subcategories.** There are different aspects to be specified: (1) *Repositories*: requirements for the artifact storage, which may be, for instance, a Git repository hosted on the cloud. The deployment subsystem will pull the artifacts from this repository when a deployment is executed. Availability, security, storage capacity, etc are defined in this category. (2) *Actions*: Actions that must be executed in the CI/CD pipeline, such as automatically initiating a deployment plan when a new commit is pushed to the master branch of the repository. (3) *Tools*: there might be requirements for using specific tools for some tasks of the pipeline. For instance, and organization might decide that the continuous integration process will be automated with Jenkins.

V. Requirements Elicitation Process

The requirements elicitation process we propose is iterative and and is summarized in Figure 5. In a first phase, requirements are elicited by considering the four categories. Each of the sub-levels of each category needed to have at least one requirement. These elicited requirements are general to any kind of CPS (or at least, any CPS that includes the categories that we identified in the taxonomy). Each of the requirement belongs to at least one of the sub-categories of each of the four facets we extracted in the taxonomy (i.e., Subsystem, Scope, Domain and Lifecycle). It is important to note that one requirement might affect more than one of the sub-levels of each category (e.g., both MiL and SiL).

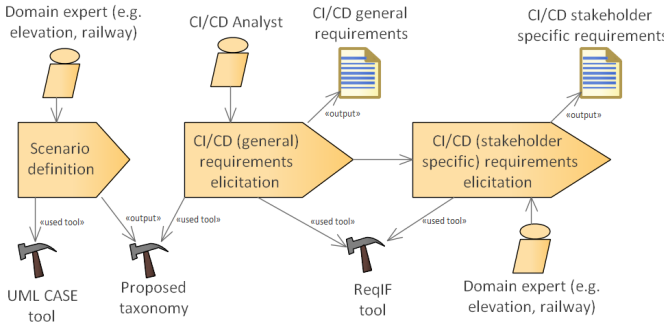


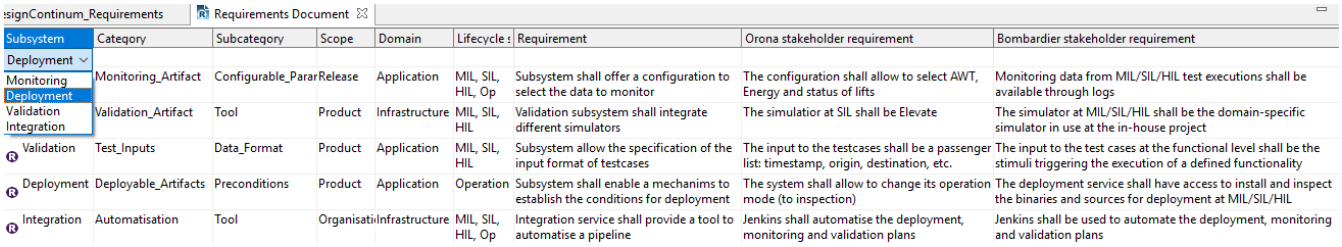
Fig. 5. Requirements elicitation process overview

In a second phase, for each of the elicited requirements, the industrial companies (i.e., Orona and Bombardier) instantiated the proposed requirement to their domain and application (e.g., in the Orona's use-case, there are some specific requirements for adapting the design-operation continuum methods for the dispatcher, for instance, the simulator used for SiL). The tool used to document requirements was ReqIf studio, because it is was freely available and both companies, as well as the researchers involved, have access to it.[2] A template with the facets of the taxonomy has been developed in order to classify each of the requirements, which includes (1) the general requirement, (2) how the requirement is instantiated for the case of the Dispatching algorithm in Orona, and (3) how the requirement is instantiated for the TCMS of BT. A screenshot of the developed template along with five elicited requirements is shown in Figure 6.

VI. Related Work

White and Edwards [33] proposed a taxonomy (RE-Views) to classify system views, subviews and their interdependencies. The requirements are classified into operational environment, system capabilities, system constraints, development requirements, verification & validation requirements and specification of system growth and change. The authors also mention a classification of requirements specification approaches, ranging from informal (natural language) to formal (mathematical) approaches. A more thorough classification of requirements specification techniques was given by

[2]https://reqif.academy/software/reqif-studio/



esignContinum_Requirements | Requirements Document

Subsystem	Category	Subcategory	Scope	Domain	Lifecycle s	Requirement	Orona stakeholder requirement	Bombardier stakeholder requirement
Deployment								
Monitoring	Monitoring_Artifact	Configurable_Parar	Release	Application	MIL, SIL, HIL, Op	Subsystem shall offer a configuration to select the data to monitor	The configuration shall allow to select AWT, Energy and status of lifts	Monitoring data from MIL/SIL/HIL test executions shall be available through logs
Deployment	Validation_Artifact	Tool	Product	Infrastructure	MIL, SIL, HIL	Validation subsystem shall integrate different simulators	The simulatior at SIL shall be Elevate	The simulator at MIL/SIL/HIL shall be the domain-specific simulator in use at the in-house project
Validation	Test_Inputs	Data_Format	Product	Application	MIL, SIL, HIL	Subsystem allow the specification of the input format of testcases	The input to the testcases shall be a passenger list: timestamp, origin, destination, etc.	The input to the test cases at the functional level shall be the stimuli triggering the execution of a defined functionality
Integration	Deployable_Artifacts	Preconditions	Product	Application	Operation	Subsystem shall enable a mechanims to establish the conditions for deployment	The system shall allow to change its operation mode (to inspection)	The deployment service shall have access to install and inspect the binaries and sources for deployment at MIL/SIL/HIL
	Automatisation	Tool	Organisati	Infrastructure	MIL, SIL, HIL, Op	Integration service shall provide a tool to automatise a pipeline	Jenkins shall automatise the deployment, monitoring and validation plans	Jenkins shall be used to automate the deployment, monitoring and validation plans

Fig. 6. Screenshot of a set of elicited requirements using our taxonomy

Roman [26] in terms of formal foundation, scope, level of formality, degree of specialization, specialization area and development method. More recently, Hasan et al. [11] provide a classification of specification approaches for non-functional requirements. Hughes et al. [12] provide a two dimensional taxonomy for requirements analysis; one dimension corresponds to the set of viewpoints of different stakeholders (*concerns*) and the second dimension (*frames*) represents the views of technical specialists. Examples of a concern and a frame are functional requirements and behavioral model respectively. Jarke et al. [14] describe an ontology of requirements engineering of an information system by dividing it into three 'worlds': *subject world* to represent properties such as timeliness, accuracy; *usage world* to represent user interface and functions; *development world* to represent development time, cost and consistency with standardized procedures. Nuseibeh et al. [22] describe a viewpoint interaction model to represent heterogeneity in requirements of software systems. A viewpoint is composed of five *slots*: style (to show representation knowledge), work plan (to show development process knowledge), domain (to show area of system under development), specification (to show system description) and work record (to show development history).

Several different domain-specific requirement taxonomies are also found in literature, e.g., for: safety requirements [7], security requirements [8], trust-related requirements [30], mobility-related requirements [10], usability requirements [3] and web-based enterprise systems [9]. Recently, automatic requirements categorization techniques have also been proposed. Knauss et al. [16] present a tool-supported approach based on Bayesian classifiers to identify security-relevant requirements. Ott [23] uses a similar approach to automatically classify and extract requirements with related information using text classification algorithms.

We were also able to find some fragmented evidence on requirements elicitation approaches for cyber-physical systems. Reza et al. [25] generated a set of quality attribute scenarios using pre-defined templates to document key non-functional requirements of a small spacecraft (CubeSat). The templates had the following fields: source, stimulus, environment, artifact, response, and response measure. Wiesner et al. [34] present a gamified approach for eliciting stakeholder requirements for a cabin video surveillance system of an aircraft. Though lacking in details, the approach works with storytelling and mutual agreement on requirements from different stakeholders. Loucopoulus et al. [19] report on the e-CORE (early Capability Oriented Requirements Engineering) approach that utilizes modeling for enterprise capabilities, goals, actors and information objects. This model-driven approach suggests different models such as capability model, goal model, actor-dependency model and information model.

Differently to all these studies, the taxonomy that we propose is related to design-operation continuum engineering methods in the context of CPSs. CPS is an emerging domain, and there is no clear path for adapting design-operation continuum practices to it, because the challenges are inherently different from the domains discussed in the existing literature, such as web development. This taxonomy is used to elicit the requirements for design-operation continuum engineering methods from two different industrial domains developing CPSs. This sets an example for instantiating a taxonomy for any other organization developing CPSs.

VII. Conclusions, Limitations and Future Work

This paper proposes a taxonomy for Design-Operation Continuum methods applied to the context of CPSs, which has been systematically developed by following the guidelines proposed by Ralph [24]. To this end, two industrial case studies have been used, interviews have been performed with industrial experts, and we have been provided access to internal documents from both companies. The last phase of the taxonomy has been the validation with CPSs engineering experts that were not involved in the development of the taxonomy. By using this taxonomy, requirements can be elicited in two steps: Firstly, generic requirements are derived, which can be applied to any CPSs. Secondly, these generic requirements are instantiated for specific systems.

While the proposed taxonomy is applied to two different case studies, the applicability of it needs further examination, both for similar and different contexts. The taxonomy is based on fairly general categories, but we nevertheless foresee revisions, particularly to cater for the domain-specific requirements of other types of CPSs. In the future, we would like to perform a more comprehensive taxonomy considering (1) other sources from a systematic literature review and (2) other industrial CPSs. On the other hand, the taxonomy could also be extended to cover Design-Operation Continuum tasks beyond the ones identified for our two case studies, such as fault recovery automation and unforeseen situations detection.

REFERENCES

[1] Rajeev Alur. *Principles of cyber-physical systems*. MIT Press, 2015.

[2] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. In *IFIP International Conference on Testing Software and Systems*, pages 194–207. Springer, 2015.

[3] H. Belani. Towards a usability requirements taxonomy for mobile aac services. In *Proceedings of the First International Workshop on Usability and Accessibility Focused Requirements Engineering*. IEEE Press, 2012.

[4] G. A. Bowen. Document analysis as a qualitative research method. *Qualitative research journal*, 9(2):27, 2009.

[5] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th international conference on software engineering companion*, pages 789–792, 2016.

[6] CENELEC. 50128: Railway Application: Communications, Signaling and Processing Systems, Software For Railway Control and Protection Systems. In *Standard Official Document*. European Committee for Electrotechnical Standardization, 2001.

[7] D. Firesmith. A taxonomy of safety-related requirements. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=29419, 2004. Online; accessed 27 Feb 2020.

[8] D. Firesmith. A taxonomy of security-related requirements. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30108, 2005. Online; accessed 27 Feb 2020.

[9] A. Ghazarian. Characterization of functional software requirements space: The law of requirements taxonomic growth. In *2012 20th IEEE International Requirements Engineering Conference (RE)*, 2012.

[10] S. Gopalakrishnan and G. Sindre. A revised taxonomy of mobility-related requirements. In *2009 International Conference on Ultra Modern Telecommunications Workshops*, 2009.

[11] M. M. Hasan, P. Loucopoulos, and M. Nikolaidou. Classification and qualitative analysis of non-functional requirements approaches. In I. Bider, K. Gaaloul, J. Krogstie, S. Nurcan, H. A. Proper, R. Schmidt, and P. Soffer, editors, *Enterprise, Business-Process and Information Systems Modeling*, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[12] K. J. Hughes, R. M. Rankin, and C. T. Sennett. Taxonomy for requirements analysis. In *Proceedings of IEEE International Conference on Requirements Engineering*, 1994.

[13] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. *International Conference on Software Engineering (ICSE)*, 2020.

[14] M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, and Y. Vassilou. Theories underlying requirements engineering: an overview of nature at genesis. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, 1993.

[15] Siddhartha Kumar Khaitan and James D McCalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):350–365, 2014.

[16] E. Knauss, S. Houmb, K. Schneider, S. Islam, and J. Jürjens. Supporting requirements engineers in recognising security issues. In D. Berry and X. Franch, editors, *Requirements Engineering: Foundation for Software Quality*, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[17] Richard Lai, S Mahmood, R Lai, and Y S Kim. Survey of component-based software development. *The Institution of Engineering and Technology*, 3(May 2007):58–64, 2014.

[18] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[19] P. Loucopoulos, E. Kavakli, and N. Chechina. Requirements engineering for cyber physical production systems. In P. Giorgini and B. Weber, editors, *Advanced Information Systems Engineering*, pages 276–291, Cham, 2019. Springer International Publishing.

[20] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 45(9):919–944, 2018.

[21] Phu H. Nguyen, Nicolas Ferry, Gencer Erdogan, Hui Song, Stéphane Lavirotte, Jean Yves Tigli, and Arnor Solberg. A systematic mapping study of deployment and orchestration approaches for iot. In *IoTBDS 2019 - Proceedings of the 4th International Conference on Internet of Things, Big Data and Security*, pages 69–82. SciTePress, 2019.

[22] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[23] D. Ott. Automatic requirement categorization of large natural language specifications at mercedes-benz for review improvements. In *Proceedings of the 19th International Conference on Requirements Engineering: Foundation for Software Quality*, Berlin, Heidelberg, 2013. Springer-Verlag.

[24] P. Ralph. Toward methodological guidelines for process theories and taxonomies in software engineering. *IEEE Transactions on Software Engineering*, 45(7):712–735, 2018.

[25] H. Reza, C. Korvald, J. Straub, J. Hubber, N. Alexander, and A. Chawla. Toward requirements engineering of cyber-physical systems: Modeling cubesat. In *2016 IEEE Aerospace Conference*, 2016.

[26] G. C. Roman. A taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–23, 1985.

[27] J. Rowley and R. Hartley. *Organizing knowledge: an introduction to managing access to information*. Routledge, 2017.

[28] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999.

[29] Hesham Shokry and Mike Hinchey. Model-based verification of embedded software. 2009.

[30] G. Sindre. Trust-related requirements: A taxonomy. In W. Wojtkowski, W. G. Wojtkowski, J. Zupancic, G. Magyar, and G. Knapp, editors, *Advances in Information Systems Development*, Boston, MA, 2007. Springer US.

[31] Martin Törngren and Paul T Grogan. How to deal with the complexity of future cyber-physical systems? *Designs*, 2(4):40, 2018.

[32] M. Usman, R. Britto, J. Börstler, and E. Mendes. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology*, 85:43–59, 2017.

[33] S. White and M. Edwards. A requirements taxonomy for specifying complex systems. In *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems*, 1995.

[34] S. Wiesner, J. B. Hauge, F. Haase, and K-D. Thoben. Supporting the requirements elicitation process for cyber-physical product-service systems through a gamified approach. In I. Nääs, O. Vendrametto, J. Mendes R., R. F. Gonçalves, M. T. Silva, G. von Cieminski, and D. Kiritsis, editors, *Advances in Production Management Systems. Initiatives for a Sustainable World*, Cham, 2016. Springer International Publishing.

A.2 Using Regression Learners to Predict Performance Problems on Software Updates: a Case Study on Elevators Dispatching Algorithms

This paper was presented at the ACM/SIGAPP Symposium On Applied Computing (SAC) in 2021 and then published in the conference proceedings. The full citation:

Gartziandia, A., Arrieta, A., Agirre, A., Sagardui, G. & Arratibel, M. (2021, March). Using regression learners to predict performance problems on software updates: a case study on elevators dispatching algorithms. In Proceedings of the 36th Annual ACM Symposium on Applied Computing (pp. 135–144).

Using Regression Learners to Predict Performance Problems on Software Updates: a Case Study on Elevators Dispatching Algorithms

Aitor Gartziandia
Ikerlan
agarciandia@ikerlan.es

Aitor Arrieta
Mondragon University
aarrieta@mondragon.edu

Aitor Agirre
Ikerlan
aagirre@ikerlan.es

Goiuria Sagardui
Mondragon University
gsagardui@mondragon.edu

Maite Arratibel
Orona
marratibel@orona-group.com

ABSTRACT

Remote software deployment and updating has long been commonplace in many different fields, but now, the increasing expansion of IoT and CPSoS (Cyber-Physcal System of Systems) has highlighted the need for additional mechanisms in these systems, to ensure the correct behaviour of the deployed software version after deployment. In this sense, this paper investigates the use of Machine Learning algorithms to predict acceptable behaviour in system performance of a new software release. By monitoring the real performance, eventual unexpected problems can be identified. Based on previous knowledge and actual run-time information, the proposed approach predicts the response time that can be considered acceptable for the new software release, and this information is used to identify problematic releases. The mechanism has been applied to the post-deployment monitoring of traffic algorithms in elevator systems. To evaluate the approach, we have used performance mutation testing, obtaining good results. This paper makes two contributions. First, it proposes several regression learners that have been trained with different types of traffic profiles to efficiently predict response time of the traffic dispatching algorithm. This prediction is then compared with the actual response time of the new algorithm release, and provides a verdict about its performance. Secondly, a comparison of the different learners is performed.

KEYWORDS

Machine learning, Performance bugs, Cyber-physical systems

ACM Reference Format:
Aitor Gartziandia, Aitor Arrieta, Aitor Agirre, Goiuria Sagardui, and Maite Arratibel. 2021. Using Regression Learners to Predict Performance Problems on Software Updates: a Case Study on Elevators Dispatching Algorithms. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3441894

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3441894

1 INTRODUCTION

Elevators systems are among the most widely used transportation systems. An elevator system is composed of several subsystems that, interacting among them, have as a goal to transport passengers safely and by considering certain Quality-of-Service (QoS) measures [2]. To this end, new functionalities of elevators are increasingly implemented through software [1]. A key component to maintain QoS measures within acceptable values in a system of elevators refers to the dispatching algorithm. These algorithms are in charge of assigning an elevator to each passenger. Being a critical component to ensure the correct operation of a system of elevators, they must be continuously maintained and evolved, addressing issues like bug-fixes, new functionalities, adaptation to legislation changes, etc.

New communication technologies (e.g., 5G) allow for updating new software versions remotely. This facilitates and shortens the release time of new versions. Subsequently, updates can be performed frequently, every time a new functionality is included or a bug is fixed, as manual intervention is minimized. As any other type of software, the dispatching algorithms are not exempt of bugs, either functional or non-functional. Specifically, non-functional bugs are hard to detect [22], and often infeasible until they are deployed on the final target (i.e., the embedded processor). Thus, run-time monitoring techniques are paramount to enable the detection of such faults in operation.

In this context, we propose a performance problem detection approach for software updates, leveraged by Machine Learning (ML). Our approach predicts the performance of a new software version, based on what it learns about the behaviour of the old version. Then, it compares its prediction to the actual behaviour of the system and gives a verdict, indicating whether the behaviour of the new release is correct or a performance problem exists. To evaluate our approach, we have followed a performance mutation testing strategy, where different mutants with performance bugs are analyzed. Different Machine Learning algorithms were tested to identify which ones best fit our purpose.

This paper reports on the experience of applying our machine learning-based performance problem detection in an industrial use-case provided by Orona. The main contributions of this paper can be summarized as follows:

- We propose an approach based on regression learning algorithms that are trained with different types of passenger

traffic profiles to predict the response time of the traffic dispatching algorithm of elevators. This prediction is later compared with the actual response time given by traffic dispatching algorithm in order to detect any possible inconsistency related to a performance bug. A trustworthy approach is employed to provide a verdict (i.e., PASS or FAIL) at runtime, which indicates whether a bug has been detected or not.

- The proposed approach is evaluated with an industrial case study provided by Orona. Specifically, performance mutation testing [8] is employed to seed performance bugs through the software program. We compare a total of five state-of-the-art Machine Learning algorithms to predict performance problems on elevators dispatching algorithms. We showed that three of them are competent enough in order to be employed in practice.

The rest of the paper is organized as follows. Section 2 provides background and related work, where we position our approach with the current state-of-the-art. Section 3 provides an overview of the performance requirements in the context of elevator dispatching algorithms. In Section 4 we present our method to predict performance problems in traffic dispatching algorithms. The evaluation of the approach is carried out in Section 5, where we used performance mutation testing to assess how different Machine Learning algorithm perform to detect potential non-functional inconsistencies. We conclude our paper in Section 6, summarizing the future work we envision.

2 BACKGROUND AND RELATED WORK

Performance issues consist on errors producing a behaviour degradation of applications in terms of execution time, response time, CPU usage, memory usage or energy consumption, without necessarily causing any fault on the expected results [8]. With the growth of the software complexity, ensuring the performance health in resource constrained applications is getting increasingly relevant. Identification of performance problems in the testing phase brings some limitations, such as the elicitation of proper performance requirements. In addition, many causes that lead to the manifestation of these problems may appear due to the system's interaction with the real environment [25]. At run-time, identifying performance problems may require a continuous monitoring of the execution, as problems may only be revealed under certain circumstances, for example, activation of specific modes or functionalities, when the system has been running for a long period, etc.

There exist a variety of root causes during implementation that can result in real-world performance problems. Jin et al. [14] identified (1) inefficient function calls, (2) skippable functions doing unnecessary work and (3) synchronization issues as main root causes for performance problems. Other less common causes are also mentioned, including wrong data structure usage, hardware architecture issues or high-level design errors [14]. By analyzing how these bugs are introduced by developers, they concluded that these errors are usually introduced due to an API or workload misunderstanding [14]. Zhang et al. [27] studied the different synchronization errors that can appear in distributed systems. According to their study, the main factors causing synchronization issues are time-consuming operation, nested loops, recursive calls and high frequency locks. A recent study by Delgado et al. [8] identified some common causes of performance problems and generated mutants based on them in order to conduct performance mutant testing.

From the testing perspective, testing non-functional properties is becoming paramount. Ferme et al. [10] emphasize the importance of integrating performance analysis inside the software life-cycle management process. Their main identified challenges encompass (1) definition of a method to establish which performance tests to perform in each life-cycle stage, (2) enabling users to declaratively define performance objectives and (3) providing fast performance feedback to improve the system. In order to design test cases, Weyuker et al. [25] summarize important aspect to be considered by practitioners. This includes (1) the design of test generation and selection strategies and algorithms, (2) definition of metrics to assess the effectiveness of performance testing strategies and (3) comparison of different hardware platforms for a given application. Besides testing, other approaches have focused on static analysis approaches to detect performance issues [18, 20]. Testing performance of software systems have been extended to many subfields, including cloud computing [16], distributed systems [9] or CPSs [24]. Unlike all these approaches, which focus on testing approaches before the software version is deployed in production, our contribution aims at detecting performance issues at run-time, once the software system has been deployed. We opted by this approach because according to domain experts, most of the performance problems are exhibited in operation because the software is highly configurable. In the case of Orona's software, it needs to be configured for each building by considering several parameters, such as the number of floors, the number of elevators and other building specific parameters.

In this sense, we have identified some tools that address the challenge of detecting errors on the deployment process. Gandalf is a service whose objective is to detect errors on cloud rollouts to stop them before they cause major failures [17]. To this end, it continuously monitors a wide range of infrastructure data to detect anomalies and when any is detected it determines if the anomaly is caused by a rollout or not by means of correlations. Finally, it uses a Gaussian discriminant classifier to decide whether the impact caused by the deployment is significant enough to stop it. FUNNEL is a tool that collects performance metrics for each software change (i.e., software update or configuration change) to detect behaviour changes [28]. It uses a Singular Spectrum Transform (SST) algorithm for the detection and a Difference-in-Difference (DiD) method, where it compares the relative performance of the treated group and a control group to decide if the software change is the cause of the behaviour change.

Our approach aims at detecting performance issues in CPSs, which might have critical impacts as compared to other less critical systems (e.g., web applications, mobile apps, etc.). Balasubramaniyan et al. proposed a methodology to design and verify CPSs with the aim of optimizing their reliability and performance mostly focused on timing issues [3]. Song et al. proposed a virtual testing environment to assess performance of CPSs [24]. The tool can simulate several configurations in parallel and identify the ones that lead to best and worst performance based on user-defined performance indicators. Markoska et al. proposed a performance testing

framework oriented to smart buildings, which is offered as a service in the cloud [19]. Among the performance metrics they consider, it includes power consumption or timing constraints. Unlike all these approaches, our approach is based on Machine Learning to identify bugs at operation-time when the software version has been deployed on the real target.

Machine learning has already been used to detect performance bugs in other contexts. Gulenko et al. evaluated a total of 13 classification algorithms used for anomaly detection on a cloud environment running Virtualized Network Function services [12]. The performance metrics monitored are processing related (e.g., CPU usage), memory related (Disk I/O) and network related (Network IO). This work concludes that Machine Learning algorithms were able to predict anomalies with high precision and recall values, with an average F1 score of 92%. Sauvanaud et al. proposed an anomaly detection system for virtualized cloud services [21] by monitoring both system metrics as well as virtual machine specific performance metrics. Unlike these approaches, which used classification Machine Learning algorithms, we propose using regression learning algorithms.

Hu et al. proposed a Machine Learning based resource usage prediction for grid computing environments [13]. Specifically, multiple Machine Learning techniques are evaluated and compared to predict metrics related to CPU, memory, disk, and network for resource allocation and load balancing. Wieder et al. proposed a QoS prediction approach where run-time monitoring and Machine Learning techniques are used to predict performance problems in the cloud [26]. In addition to infrastructure metrics as memory or CPU usage, the system proposed also monitors application-level metrics such as response time or number of logins. This work proposes combining Machine Learning classification methods (Random forest, decision tree and SVM) with time series analysis methods (AR, ARIMA and ETS) to improve the prediction capacity of the system. The main difference between these studies and ours is that their approach aims to predict future inconsistencies and adapt the system to withstand those conditions. Conversely, in our case, the approach is solely focused on identifying the performance issue and notifying to engineers in order them to take the appropriate correct action. Furthermore, we assess our approach by using an industrial case study along with performance mutation testing [8].

3 CASE STUDY: PERFORMANCE REQUIREMENTS IN ELEVATOR DISPATCHING ALGORITHMS

Elevators are complex CPSs composed of different subsystems that collaborate to transport passengers vertically in a building. Controllers are in charge of managing both the vertical (from floor to floor) and the horizontal movements (doors opening and closing) of a single elevator. The traffic master is the software system in charge of the coordination of the controllers to serve the floor calls requested by passengers. The main responsibilities of the traffic master include the execution of the dispatching algorithm (i.e., the allocation of passenger calls to any of the available cars) and the overall system signalling (registration of the calls, information to the passenger), but it can also carry out additional functionalities such as access control (i.e. permission for the passengers to access certain floors) or management of special operating modes.

The traffic dispatching algorithm is the software component that selects the optimal elevator to serve a specific landing call. Thus, this component is critical to ensure the Quality of Service (QoS) of the elevator installation because of many reasons. Firstly, the assignment of the landing calls has a direct impact on the average waiting time and overall journey time of the passengers. Secondly, it affects to the energy consumption or transport capacity of the elevators. The traffic algorithm constantly evolves to be adapted to particular installations, improving the assignment process by including new rules or using new techniques such as artificial intelligence. Additionally, new criteria for the assignment such as the number of stops, load balancing or energy consumption are usually required for some installations.

The traffic dispatching algorithm is executed periodically to allocate all the active floor calls. Depending on the algorithm, already existing floor call allocations can be reallocated (to a different car) to optimize the overall cost function. This means that the allocation process is highly dynamic as it depends on the current system context. This fact, alongside with other context situations such as highly demanding traffic profile or a big number of floors can derive in a high consumption for the limited resources available on the allocation of landing calls in time. In this sense, the traffic algorithm should allocate the complete set of active landing calls in a limited time frame (response time) to provide an acceptable QoS. Moreover, the traffic algorithm is executed within a task that shares computing resources with other tasks that provide above mentioned functionalities (signalling, access control, etc.). Thus, system performance monitoring is crucial to ensure a proper system QoS. If such performance decreases, the overall system QoS degrades, and additional hardware resources should be considered [4].

A special type of traffic algorithms in which performance is specially relevant is destination algorithms. Unlike conventional algorithms, in which many passengers share the same uplanding or downlanding call, in destination algorithms each passenger registers his own call. This "destination call" is composed by a landing call and a destination floor, and more importantly, it remains active until the destination is reached. Therefore, a destination algorithm requires managing each destination call individually. Additionally, this type of dispatchers provides extra functionalities such as access control, that could affect the response time of the algorithm. Thus, in high-population buildings, the amount of active calls can be huge and can severely affect the system performance.

When a new version of the algorithm is released, performance under different traffic conditions and installations must be checked to identify issues related to a poor implementation of new or improved functionalities. Usually, benchmark installations with different number of floors and elevators, and some theoretical passenger profiles are used to validate the new release. These profiles represent some traffic demands for different types of buildings (offices, residential, hotels, etc.) during different periods of the day. Most commonly used office profiles include, (1) Morning UpPeak, representing the entrance to office in the morning (2) LunchPeak, representing the lunchtime, where there is a mix of passengers entering and exiting the building and (3) DownPeak, representing the exit of the office in the afternoon. There are also full day profiles that represent

the traffic in an office during a working day and therefore, include a morning UpPeak, LunchPeak at midday and Afternoon DownPeak complemented with some inter-floor traffic (passenger flow between different floors of the building) during the morning and the afternoon [4].

However, detecting performance problems that could compromise response time in a new release of the algorithm is a complex task due to the following factors. Firstly, some functionalities of the algorithm are only activated under certain traffic demands, keeping potential performance issues hidden. For example, parking of empty elevators to heavy floors is only activated when a big demand from that floor is given. Secondly, performance is highly dependent on installation specific factors: number of controllers, number of floors in the building, etc. Lastly, performance is highly dependent on the passenger flow of each building. In summary, poor performance can be exhibited in some installations or traffic conditions but not in others, which makes it difficult to validate the software system.

Reproducing all the real scenarios to analyse the potential performance issues in the laboratory is unfeasible due to the effort it would mean. Moreover, there is often a lack of actual traffic profile information and thus, it is not easy to reproduce the operation conditions of the final installation in the laboratory. Therefore, it is necessary to include monitoring and detection mechanisms in operation to facilitate the detection of potential performance problems in a particular installation. This way, when releasing a new version, potential performance issues can be detected automatically before they compromise response time, and eventually, a rollback to a previous version could be performed.

4 PERFORMANCE PROBLEMS PREDICTION METHOD

Figure 1 shows the overall architecture of the proposed approach. The proposed solution is divided into two main phases: (1) the training phase and (2) the performance bugs identification phase.

The idea of the training phase is to train a regression learning algorithm by using performance data collected from operation. During the training phase (a) the current software version remains in operation and (b) the data generated is used to adapt the Machine Learning algorithm's internal parameters, so that its performance improves on future unseen input data. To train the Machine Learning algorithm, the data is categorized in two groups: the features, including the inputs of the studied piece of software and the label, with the measures of the performance metric we want to predict.

When the regression learning algorithm is trained, it yields a trained regression model, which is used in the performance problems identification phase to identify performance issues after a new release has been deployed remotely. This phase has three steps: (a) execution of the new release to collect the input data and performance metrics, (b) prediction by the regression model, which yields the expected performance bases on the input values, and (c) the arbitration process, which compares the expected performance obtained by the regression algorithm with the actual metrics collected during execution.

With this method, we aim at detecting performance issues that depend on the inputs of a software. We now explain the methodology and the developed implementation more in detail.

4.1 Performance metrics

The response time is the time required by the algorithm to assign a set of landing calls and communicate them to the elevators. The algorithm is invoked periodically and must provide a valid assignment every cycle. In every cycle, the algorithm receives all the information about the status of the elevators (e.g., position, velocity, doors status, etc.) and a set of car and landing calls. Car calls are assigned to a particular car and represent the destination of the passengers. In Orona's algorithm, car calls are not assigned by the algorithm, they are input information to decide the assignment of landing calls. Landing calls represent the origin of the passengers, and therefore, it is the algorithm's responsibility to decide which car is the most suitable to attend the landing calls.

There are two types of landing calls: conventional calls and destination calls. In destination algorithms, a call is registered by each passenger. Within an installation, the number of active calls impacts the response time of the algorithm. The lunch time, for example, is one of the most demanding periods for the algorithm in office buildings. During this period, there is a mix of passengers entering and exiting the building. During LunchPeak and, for example, during the morning UpPeak, the processing load of the algorithm is high. In addition to the processing of a high amount of calls, some other mechanisms such as parking or control of long waiting times are activated, which also increases the processing time.

Specially in destination algorithms, where the attendance to the passengers is individualized, the number of active calls influences the response time. Active calls can be in different states: (1) Registered: first cycle in which the algorithm receives a call and the assignment remains pending. (2) Assigned: the algorithm has already selected an elevator to attend the call but the elevator has not still arrived to answer the call. (3) Answered: the elevator has arrived at the origin of the passenger and the passenger is travelling to his destination. Finally, once the passenger arrives to his destination, the call is cancelled.

In this context, the performance metric that has been monitored is the response time of the periodic task that executes the dispatching algorithm. That is, the time frame in which the algorithm dispatches (allocates) the active calls to the available cars. This response time is directly related to the existing traffic: an increase in passenger traffic causes a rise in the number of active calls and this results in a higher demand of computing resources to allocate these calls. As a consequence, the response time of the task executing the algorithm increases and, eventually, a deadline loss can occur. Thus, we have chosen the number of active calls as the value to predict the response time of the algorithm. To measure the response time of the task, the algorithm code has been instrumented with high precision clocks provided by the operating system.

4.2 Regression learning algorithms

Machine Learning (ML) algorithms are a subset of Artificial Intelligence algorithms. They provide systems the ability to automatically learn and improve from past experiences without being explicitly programmed to. ML algorithms can be catalogued into two categories: supervised and unsupervised learning. While supervised learning algorithms aim to map input objects and output variables from labelled training data, unsupervised learning algorithms are

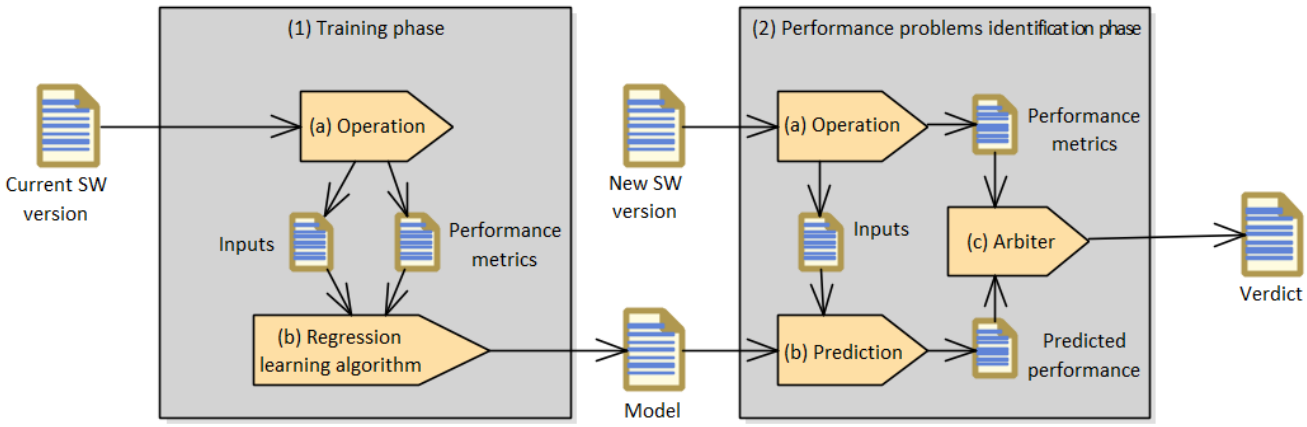


Figure 1: Architecture of the performance problem prediction method

trained based on input data only, acting as a clustering technique [7]. Supervised learning techniques can be categorized both in regression and classification techniques. Regression algorithms aim to map inputs to real-valued outputs (i.e., a number), whereas classification aims to map inputs into categorical outcomes [7]. Our approach uses regression learning algorithms because it is necessary to predict a quantitative value to detect a performance issue. The regression algorithms evaluated in this paper have been TRGP, Regression Tree, Ensemble, Stepwiselm and SVM. The latter is commonly used as a classifier but has also proven to be useful as a regression algorithm [6].

4.3 Training phase

In this phase, the input feature will be the number of active calls, labelled with the response time for each cycle of the algorithm every minute. To do so, we developed a script that automatically extracts this data from a database. Once, the data is extracted, the script launches the training phase by using the MATLAB Machine Learning toolbox. The regression learning algorithm yields a trained regression model, which can later be used in the testing phase to predict the response time of the algorithm based on the number of active calls.

4.4 Prediction phase

When a new version of the algorithm is released to operation, data of active calls and response time of the algorithm are collected every minute. Response time has to be less than the deadline of the task that executes the traffic algorithm. This is checked with a rule. However, other performance problems that can be indicators of bad implementations are more difficult to detect due to the dependence of the response time to the characteristics of the installation, the functionalities activated in the algorithm and the traffic flow. Therefore, active calls and response time measurements are collected to compare them with predicted data.

The execution of the new release provides the number of active calls every minute to the trained regression model. This model, estimates the response time over time based on the training produced during the training phase.

The arbitration process is in charge of detecting performance problems in a new release of the software. By comparing the data that is being collected from the execution of the new release and the prediction of the regression model, this module provides a verdict. The arbitration process has been designed considering three criteria: (1) average response time (2) maximum response time and (3) variance in response time. These criteria have been analyzed in two time-frames: (1) daily, with the response times of the algorithm in a whole day traffic flow and (2) every minute, with the traffic of the previous 5 minutes. Experts from Orona have set the tolerable variations in the variance, average and maximum response times that could be considered normal in the execution of the algorithm. In addition the variation percentages that should be classified as performance problems within the two time-frames were specified by the same domain experts. The arbitration process inputs the real response time of the new release and compares it to the prediction of the regression model. When an abnormal variation is detected in any of the criteria, it must be analyzed, and having the minute by minute verdict facilitates the detection of the performance problem, as it allows analyzing under which traffic conditions is the problem shown.

5 EMPIRICAL EVALUATION

This section evaluates the proposed approach by using an industrial case study. To this end, we aimed at answering the following Research Questions (RQs):

- RQ1: How does each of the selected regression learning algorithms perform for predicting performance problems on software updates?
- RQ2: Are there any differences when training the machine-learning algorithms with real field data or theoretical data?

5.1 Experimental setup

This subsection explains the proposed experimental design to answer the aforementioned RQs.

5.1.1 Case study. As a case study, we used the dispatching algorithm of Orona developed in C/C++. This algorithm is the component inside an elevator in charge of assigning a specific elevator to each call. Performance on this algorithm is critical, as it must provide a response in a limited time frame. Therefore, it is important that when software updates are performed in the dispatching algorithm of Orona, these are free of performance issues. The type of algorithm used for the evaluation is a destination algorithm and, therefore, there is one call per each passenger. For the evaluation, two types of passenger profiles have been used: full day theoretical profile, an actual traffic profile from an office building in Paris obtained from literature [23], and full day profile collected from a real building of Orona. The real building is a 10-floor office building with six elevators in it and the entrance in the ground floor.

5.1.2 Evaluation metrics. We used performance mutation testing to evaluate the approach [8]. To this end, a total of 45 performance mutants were generated by following the performance mutation operators proposed by Delgado et al. [8]. Mutation testing aims to generate a set of versions from the original program and adding a synthetic variation on it. When the outcomes of a test differ from the mutant to those of the original version, it is considered that the mutant is killed. This technique has been found to be a good substitute of real faults [15]. The difference between traditional mutation testing and performance mutation testing is that the mutation operators for the latter are focused on injecting performance problems and keeping the original functionality of the program [8], whereas the former focuses on changing the program functionality. To consider a mutant killed in performance mutation testing, performance service metrics are considered, such as execution time or memory usage [8]. We have systematically created mutants that could affect performance based on the performance mutation operators proposed in [8], which include: method call, loop perturbation and conditional execution. Specifically, we have created mutants by: simulating heavy operation in different parts of the algorithm, Move/Copy Statement into Loop, Removal of Stop Condition in Loop and Unnecessary Calculation of values. In this paper, we focused on the response time of the dispatching system as a measure for detecting performance problems.

Similar to related studies [5, 11], we used the precision, recall, accuracy and F-measure metrics to measure the performance of the proposed regression learning algorithms. We also developed a regression test oracle, which involves the original dispatching algorithm. When executing the tests, we obtained the running times over time of both the original dispatching algorithm as well as the one from the mutants. The running time from the dispatching was provided as input to the arbiter, forming this way the regression test oracle to classify the test as pass or fail without the prediction part, which is the core of this study.[1] This classification was catalogued as a True Positive (TP), True Negative (TN), False Negative (FN) or False Positive (FP) as defined below:

- TN: Both our approach and the regression test oracle returned a "PASS" verdict.
- TP: Both our approach and the regression test oracle returned a "FAIL" verdict.
- FN: Our approach returned a "PASS" and the regression test oracle returned a "FAIL".
- FP: Our approach returned a "FAIL" and the regression test oracle returned a "PASS".

When mutants were catalogued as TN, TP, FN or FP, the precision, recall, F1 and accuracy were obtained following Equations 1, 2, 3 and 4.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F1 = \frac{2 \times (precision \times recall)}{precision + recall} \tag{3}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

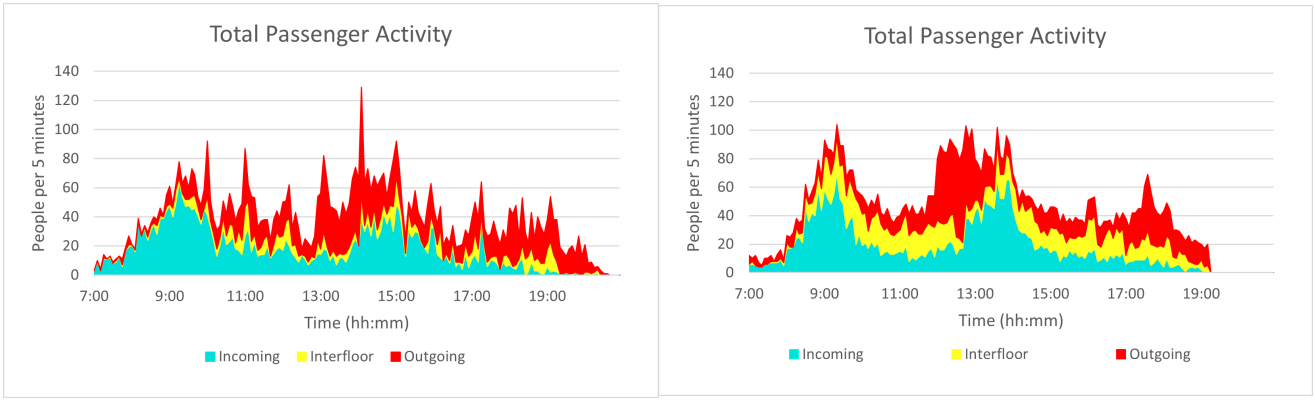
5.1.3 Training strategy. To perform the training of the model we have established three different scenarios. In the first scenario, the training was performed with four test cases, which were obtained from field data in a real office installation maintained by Orona (the same real office installation used in our evaluation). We had access to real passenger flows for 4 different days in an office building with 6 elevators and 10 floors. Therefore, we have been able to use this data to execute the algorithm and obtain the response time of these days. We have executed the passenger profiles 5 times as the execution time varies slightly depending on the other tasks of the dispatcher. We have used a destination dispatcher that requires an individualized treatment of each active call. Figure 2a illustrates the traffic collected in a working day in the office building. During the morning, most passengers go from the ground floor to the upper floors of the building, from 13:00 to 15:00 there is a mix of traffic going from and to the ground floor, and at the end of the working day, most passengers' destination is the ground floor.

In the second scenario, the training was performed with four test cases, which were obtained from Elevate™[2], a passenger traffic simulation tool. The used profile was Siikonen full day traffic profile that is based on a sample multi-tenant office building in Paris [23]. Each test case was executed 5 times as response time can differ slightly in different executions. Figure 2b illustrates the traffic flow of the siikonen profile.

In the third scenario, both of the training sets from the previous scenarios have been used. Therefore, four profiles collected from the field and four profiles collected from the theoretical profiles where used to train the model. Each of them was executed 5 times. The analysis of these three scenarios provided us information about how important is to obtain real field data to predict performance.

[1] Notice that in practice, in operation, we lack this regression test oracle, and subsequently, it is only used for our evaluation

[2] https://www.peters-research.com/index.php/elevate



(a) Full day profile collected from real installation by Orona

(b) Siikonen Full day profile

Figure 2: Total passenger activity of real installation and theoretical profiles obtained with Elevate

5.1.4 Performance problems identification. As the goal is to identify performance problems in operation, we have used the field profiles as input for the prediction model. In the cases were field data was used to train the algorithms, to avoid bias, we used the k-fold cross validation. This means that we did not use the same dataset for training as well as for testing.

5.2 Analysis of the Results

Table 1 summarizes the obtained results of the selected regression algorithms for the three considered scenarios. The first scenario aimed at comparing the performance of the selected algorithms when trained with field data. For the five regression learning techniques, TRGP, Regression Tree and Ensemble showed the strongest results. These techniques had a precision of around 0.95, a recall of around 0.93, an accuracy of around 0.93 and an F1 measure of around 0.94. Conversely, SVM and Stepwiselm algorithms showed overall worse results. Despite having a high recall, their precision, accuracy and F1 were significantly lower than the rest of the algorithms.

Since field data is not always available, the second scenario aimed at investigating how the different algorithms performed with theoretical test data. Similar to the previous scenario, TRGP, Regression Tree and Ensemble were the algorithms showing the best results. SVM and Stepwiselm showed better recall measures but lower precision, accuracy and F1 measure, similar to the first scenario. It is noteworthy, however, that for the best algorithms (i.e., TRGP, Regression Tree and Ensemble), the recall, accuracy and F1 measures were dropped with respect to the previous scenario, although the precision increased. This means that when training with theoretical data, the number of false negatives slightly increased, whereas the number of false positives decreased.

The third scenario aimed at investigating how the different algorithms performed with a mixture field and theoretical test data. Similar to the previous scenarios, TRGP, Regression Tree and Ensemble were the algorithms showing best results. However, as in the second scenario, they showed lower recall, accuracy and F1 measures when compared to those results from the first scenario (i.e., when the algorithms were trained with field data). Nevertheless, their values were slightly higher than those obtained when trained solely with theoretical data (i.e., second scenario). Again, for this scenario, SVM and Stepwiselm showed low precision, accuracy and F1 values.

5.3 Discussion

From the results, it can be seen that Regression Tree, Ensemble and TRGP are overall algorithms showing best results. On the downside, SVM and Stepwiselm showed the worst precision, accuracy and F1 values for the three scenarios, despite having a higher recall value. This means that there were a high number of false positives, which results in a high number of mutants catalogued as failing when they should have been catalogued as non-failing. An insight behind the obtained results is that the problem to solve is non-trivial, as the SVM, which is commonly taken as a baseline algorithm to determine the difficulty of the problem, yields bad results. We can thus answer the first RQ as follows:

> Overall, Regression Tree, Ensemble and TRGP are the best algorithms for predicting performance problems in software updates in elevator dispatching algorithms, while SVM and Stepwiselm are not valid.

Results for TRGP, Regression Tree and Ensemble seem promising when trained with field data, although the use of theoretical test data for training them seems to distort the recall, accuracy and F1 measures of the algorithms. When having a close view on the training sets, we figured out that the training data in theoretical profiles are significantly different to the data used in the real installation. Specifically, this led not to train certain areas that covered scenarios appearing in all real installations. Instead, the theoretical profiles focused on stressing the system, increasing significantly the number of active calls at certain times during the full day. Conversely, the real traffic profiles had a more constant passenger flow signals, without strong peaks, having an average lower number of active calls per minute, as can be seen in Figure 2. This is specially remarkable during morning UpPeak and midday LunchPeak.

Table 1: Results summary for the different scenarios of our approach [Scenario 1, Scenario 2, Scenario 3]

	TRGP	REGRESSION TREE	ENSEMBLE	SVM	STEPWISELM
PRECISION	[0.95, 1.00, 1.00]	[0.95, 0.99, 0.97]	[0.96, 1.00, 1.00]	[0.58, 0.61, 0.58]	[0.62, 0.60, 0.59]
RECALL	[0.94, 0.67, 0.75]	[0.94, 0.70, 0.73]	[0.92, 0.65, 0.73]	[1.00, 1.00, 1.00]	[1.00, 1.00, 1.00]
ACCURACY	[0.94, 0.82, 0.87]	[0.94, 0.83, 0.84]	[0.94, 0.81, 0.85]	[0.58, 0.61, 0.58]	[0.65, 0.60, 0.59]
F1	[0.95, 0.81, 0.86]	[0.95, 0.82, 0.84]	[0.94, 0.79, 0.85]	[0.74, 0.76, 0.74]	[0.77, 0.75, 0.74]

In our evaluation we show again the differences between theory and practice, instantiated in a real-life example in a widely used domain, i.e., the vertical transport domain. Many assumptions are made in theory to explain the phenomenon and concepts, yet, in real-life, assumptions and conditions do not always hold and are not unique. In areas like web or mobile engineering, Design-Operation Continuum methods (e.g., DevOps) are widely used, bringing advantages like testing "on-the-fly" or taking data from operation to development-time for a more extensive analysis. In our study, we show that this concept is required to be extended to wider engineering areas, such as the ones related to embedded and Cyber-Physical Systems (CPS).

Having discussed this, we can answer the RQ2 as follows:

> Results significantly differ depending on the training type used. Training them with field data is the best option, whereas the use of theoretical test data shall be further analyzed.

5.4 Lessons learned

The results show that collecting information from the operation of the traffic algorithm can be used to detect performance problems when a new version is released. Usually, changes in the software of the algorithm consists on (1) including new strategies to improve the QoS of the algorithm, (2) adapting existing functionalities to special installations (e.g., multi entrance floors building), (3) extending the algorithm to consider new assignment rules or (4) adapting new legislative changes. Engineers in charge of the updates and extensions are not always original developers of the algorithm, therefore, misunderstanding about the usage of data structures and control structures is not uncommon. In addition, some changes must be performed quickly to support operational installations.

The algorithm includes complex control structures to manage the floors, controllers and calls and an inefficient implementation of these structures can result in performance problems in some installations or under some kind of traffic flow. To check the functional behaviour of a new version, a simulator named Elevate is used. In simulation, several validations can be performed semi-automatically. This provides a good confidence level that the updates will behave as expected. However, an exhaustive checking of the performance of the new release is usually not feasible. The performance is dependent on the features of the installations and the traffic profile, and requires real time validation, therefore, the number of executions that can be run in the laboratory are reduced. Consequently, usually static performance analysis is used to guarantee the response time under the worst scenario.

The method presented in this paper uses previous information collected in the installations to detect performance problems that can be exhibited during operation. The results show that regression learning algorithms can be used to train a model to predict the response time of an installation. This information can be used to identify performance problems on the release of a new version by using rules provided by the domain experts. The number of active calls has proven to be a good input factor for the model. By using this method in installations of Orona, an overview of the performance problems detected in different installations can be obtained. By analyzing this information, implementation inefficiencies that have a negative impact in performance can be detected. From the experiment carried out on the first scenario we obtain that Regression Tree results in 5 false positive out of 180. The type of mutants that result in false positives are two mutants simulating heavy operation in parts of the algorithm that are activated conditionally, two related to the Removal of Stop Condition in Loop and one of Unnecessary Calculation of values. Regarding false negatives, 6 out of 180 were detected. Overall, 94% of the tests provided a good results.

We highly recommend using this method systematically especially in office buildings and hospitals where the traffic is more demanding than in the residential sector and performance problems have a higher probability of being exhibited. It is important to remark that the analysis shows that the training should be specific for a particular installation.

This would imply: (1) continuously collecting the number of active calls and response time during the normal operation of the dispatcher in the installations, (2) training the model with this information and (3) integrating in the dispatcher the model and the rules to identify performance problems. To continuously monitor the active calls and response time, a persistence mechanism shall be provided. Due to performance reasons, we think that the training phase shall be performed at the laboratory, not in the installations. Therefore, data collected in the installation shall be sent to the laboratory for analysis.

Figure 3 depicts the physical infrastructure needed to accomplish the proposed approach, i.e., the deployment of the architecture proposed in Fig.1. Oronas cloud could be used as a bridge both for deployment and telemetry purposes. The software developed in the lab would be remotely deployed to the real installation using a microservice based approach. Due to security issues, the real installation equipment should not have open ports and thus, a notification-based schema based on MQTT is proposed to alert the edge gateway that a new version is available in Oronas docker registry. This way, the edge gateway downloads and installs the new software in the traffic master. Besides that, in operation, the

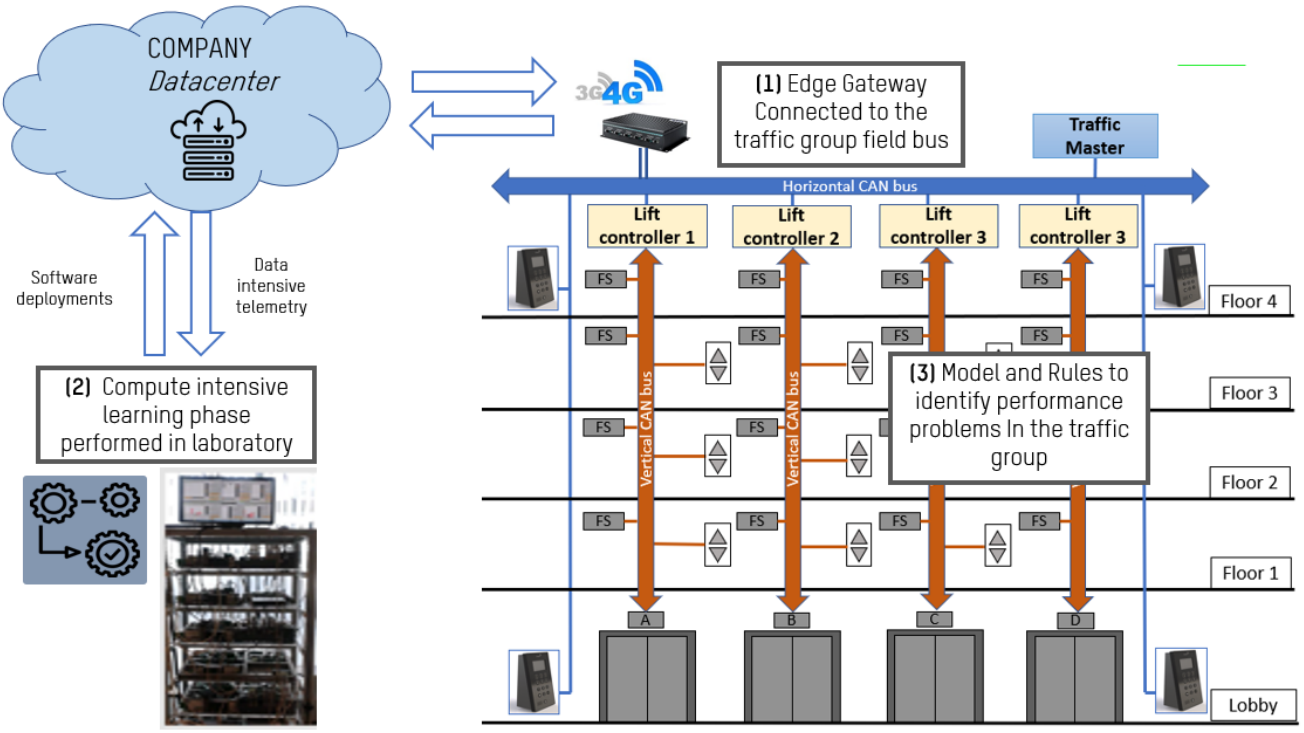


Figure 3: Physical infrastructure to carry out the proposed approach

edge gateway collects the required operational data coming from the real installation and feeds it back to the laboratory for training purposes, closing the loop and enabling a continuous validation schema in operation.

We believe that in office buildings sending the information weekly could be sufficient. In order to select the input data to train the algorithm, the type of building shall be taken into account. Traffic flow in office buildings can be dependent on the season (for example, vacation periods, etc.). Besides, in multi-tenant buildings, traffic flow can be more variable over the time than in single tenant buildings. Therefore, the option of having a different training according to the season or features shall be further analyzed.

Lastly, we envision two alternatives for the identification of performance problems: (1) execute in the laboratory and (2) execute with the dispatcher in the installation. The verdict to identify a performance problem in this version uses the information of the response time of a full day and every 5 minutes. This could be done either in the laboratory or in the execution in the installation. In this case, we think that the best option is to perform this process in the installation as it allows to detect performance problems at operation time and take contingency actions if needed.

5.5 Threats to validity

Internal validity: Our evaluation is subject to some internal threats. One such threat relates to the configurations of the used Machine Learning algorithms. To reduce this threat, we used the default parameters provided by the MATLAB framework to train the algorithms. In addition, the arbiter uses certain thresholds that require configurations. The value of these parameters could significantly change the results of our evaluation. To reduce this threat, these values were discussed with domain experts.

External validity: A potential external validity threat in our evaluation relates to the used benchmark dataset and to only using a single case study. However, this case study is a real industrial case study of high complexity. Furthermore, the dataset is the one used by Orona to test their dispatching algorithms. We are thus using a benchmark with real industrial data and within a real-world setting. Another external validity threat in our approach could relate to how the training was performed. To avoid bias in the results, we did not use the same dataset for training and for testing an algorithm. In those cases were necessary, the k-fold cross validation techniques were used.

6 CONCLUSIONS AND FUTURE WORK

This paper has investigated the use of Machine Learning techniques to detect performance problems after new software release deployments. Then, proposed approach has been tested in an industrial use case, specifically in elevator traffic dispatching algorithms, in collaboration with Orona. The conclusions obtained from the experiments conducted, are the following:

- We have proven that applying regression learners to data collected on elevators installation in order to predict response time is a valid mechanism to detect performance problems.

- The method does not consider time-critical concerns, so may be only applicable to non time-critical CPSs. The use of the method in a time-critical context requires further analysis.
- The regression algorithms which showed the best results for all the experimental setups where TRGP, Regression Tree and Ensemble, while SVM and Stepwiselm have proved to be the least appropriate.
- The best way to train a regression algorithm to predict performance problems in dispatching algorithms is to train them with real field data from the installation, rather than theoretical profiles.

In future, we plan to continue this research by evaluating the effectiveness of this method in its application on different types of installations. In addition, we shall investigate the different training strategies. In this sense, two aspects must be taken into account are (1) the frequency of the training, as people flow is dynamic over the time and (2) the need of training the model with different training sets depending on the season of the year, as traffic may change depending on vacation periods, etc. Lastly, the use of different theoretical traffic profiles will be further analyzed. In addition to the full day profiles used in this work to evaluate the approach, the use of UpPeak, LunchPeak, DownPeak and Interfloor theoretical profiles will be further investigated to analyze whether they can be used to predict the performance.

ACKNOWLEDGMENT

This publication is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319. This work has been partially supported by the Basque Government through the Elkartek program under the DIGITAL project (Grant agreement no. KK/2019-00095). Aitor Arrieta and Goiuria Sagardui are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1326-19), supported by the Department of Education, Universities and Research of the Basque Country.

REFERENCES

[1] Jon Ayerdi, Aitor Garciandia, Aitor Arrieta, Wasif Afzal, Eduard Enoiu, and Aitor Agirre. Towards a Taxonomy for Eliciting Design-Operation Continuum Requirements of Cyber-Physical Systems. In *IEEE 28th International Requirements Engineering Conference*. IEEE, 2020.

[2] Jon Ayerdi, Sergio Segura, Aitor Arrieta, Goiuria Sagardui, and Maite Arratibel. Qos-aware metamorphic testing: An elevation case study. In *International Symposium on Software Reliability Engineering (ISSRE 2020)*. IEEE, 2020.

[3] Sreram Balasubramaniyan, Seshadhri Srinivasan, Furio Buonopane, B. Subathra, Jüri Vain, and Srini Ramaswamy. Design and verification of Cyber-Physical Systems using TrueTime, evolutionary optimization and UPPAAL. *Microprocessors and Microsystems*, 42(2016):37–48, 2016.

[4] G. C. Barney. 2003.

[5] Wing Kwong Chan, Jeffrey CF Ho, and TH Tse. Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, 20(2):89–120, 2010.

[6] Vladimir Cherkassky and Yunqian Ma. Practical selection of SVM parameters and noise estimation for SVM regression. *Neural Networks*, 17(1):113–126, 2004.

[7] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[8] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. Performance mutation testing. *Software Testing Verification and Reliability*, 2020.

[9] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. *Proceedings of the Fourth International Workshop on Software and Performance, WOSP'04*, pages 94–103, 2004.

[10] Vincenzo Ferme and Cesare Pautasso. Towards holistic continuous software performance assessment. *ICPE 2017 - Companion of the 2017 ACM/SPEC International Conference on Performance Engineering*, pages 159–164, 2017.

[11] Ahmet Esat Genç, Hasan Sözer, M Furkan Kıraç, and Barış Aktemur. Advisor: An adjustable framework for test oracle automation of visual output systems. *IEEE Transactions on Reliability*, 2019.

[12] Anton Gulenko, Marcel Wallschlager, Florian Schmidt, Odej Kao, and Feng Liu. Evaluating machine learning algorithms for anomaly detection in clouds. *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*, pages 2716–2721, 2016.

[13] Liang Hu, Xi Long Che, and Si Qing Zheng. Online system for grid resource monitoring and machine learning-based prediction. *IEEE Transactions on Parallel and Distributed Systems*, 23:134–145, 2012.

[14] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47:77–87, 2012.

[15] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[16] Rakesh Kumar Lenka, Pranali Bhanse, and Utkalika Satapathy. Load performance testing on cloud platform. *Proceedings - IEEE 2018 International Conference on Advances in Computing, Communication Control and Networking, ICACCCN 2018*, pages 414–419, 2018.

[17] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Microsoft Azure, Peng Huang, Johns Hopkins University, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Microsoft Research, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[18] Yepang Liu, Chang Xu, and Shing Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. *Proceedings - International Conference on Software Engineering*, (1):1013–1024, 2014.

[19] Elena Markoska and Sanja Lazarova-Molnar. Towards smart buildings performance testing as a service. *2018 3rd International Conference on Fog and Mobile Edge Computing, FMEC 2018*, pages 277–282, 2018.

[20] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015-June:369–378, 2015.

[21] Carla Sauvanaud, Mohamed Kaâniche, Karama Kanoun, Kahina Lazri, and Guthemberg Da Silva Silvestre. Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. *Journal of Systems and Software*, 139:84–106, 2018.

[22] Sergio Segura, Javier Troya, Amador Durán, and Antonio Ruiz-Cortés. Performance metamorphic testing: motivation and challenges. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, pages 7–10. IEEE Press, 2017.

[23] ML Siikonen. On traffic planning methodology. *Lift Report*, (March), 2001.

[24] Zhen Song, Philippe Labalette, Robin Burger, Wolfram Klein, Sudev Nair, Suhas Suresh, Ling Shen, and Arquimedes Canedo. Model-based cyber-physical system integration in the process industry. *IEEE International Conference on Automation Science and Engineering*, 2015-October(September 2016):1012–1017, 2015.

[25] Elaine J. Weyuker. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26:1147–1156, 2000.

[26] Philipp Wieder, Edwin Yaqub, Ramin Yahyapour, and Ali Imran Jehangiri. Distributed predictive performance anomaly detection for virtualised platforms. *International Journal of High Performance Computing and Networking*, 11:279, 2018.

[27] Chen Zhang, Jiaxin Li, Dongsheng Li, and Xicheng Lu. Understanding and Statically Detecting Synchronization Performance Bugs in Distributed Cloud Systems. *IEEE Access*, 7:99123–99135, 2019.

[28] Shenglin Zhang, Ying Liu, Dan Pei, Yu Chen, Xianping Qu, Shimin Tao, Zhi Zang, Xiaowei Jing, and Mei Feng. FUNNEL: Assessing Software Changes in Web-Based Services. *IEEE Transactions on Services Computing*, 11(1):34–48, 2018.

A.3 Microservices for Continuous Deployment, Monitoring and Validation in Cyber-Physical Systems: an Industrial Case Study for Elevators Systems

This paper was presented at the IEEE International Conference on Software Architecture (ICSA) in 2021 and then published in the conference proceedings. The full citation:

Gartziandia, A., Ayerdi, J., Arrieta, A., Ali, S., Yue, T., Agirre, A., Sagardui, G. & Arratibel, M. (2021, March). Microservices for continuous deployment, monitoring and validation in cyber-physical systems: an industrial case study for elevators systems. In 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C) (pp. 46-53). IEEE.

Microservices for Continuous Deployment, Monitoring and Validation in Cyber-Physical Systems: an Industrial Case Study for Elevators Systems

Aitor Gartziandia*, Jon Ayerdi†, Aitor Arrieta†, Shaukat Ali‡, Tao Yue‡, Aitor Agirre*,
Goiuria Sagardui† and Maite Arratibel §
Ikerlan*, Mondragon University †, Simula Research Laboratory ‡, Orona§
*{agarciandia, aagirre}@ikerlan.es, †{jayerdi,aarrieta,gsagardui}@mondragon.edu,
‡{shaukat, tao}@simula.no, §marratibel@orona-group.com

***Abstract*—Cyber-Physical Systems (CPSs) are systems that integrate digital cyber computations with physical processes. The software embedded in CPSs has a long life-cycle, requiring constant evolution to support new requirements, bug fixes, and deal with hardware obsolescence. To date, the development of software for CPSs is fragmented, which makes it extremely expensive. This could be substantially enhanced by tightly connecting the development and operation phases, as is done in other software engineering domains (e.g., web engineering through DevOps). Nevertheless, there are still complex issues that make it difficult to use DevOps techniques in the CPS domain, such as those related to hardware-software co-design. To pave the way towards DevOps in the CPS domain, in this paper we instantiate part of the reference architecture presented in the H2020 Adeptness project, which is based on microservices that allow for the continuous deployment, monitoring and validation of CPSs. To this end, we elaborate a systematic methodology that considers as input both domain expertise and a previously defined taxonomy for DevOps in the CPS domain. We obtain a generic microservice template that can be used in any kind of CPS. In addition, we instantiate this architecture in the context of an industrial case study from the elevation domain.**

***Index Terms*—Microservices, DevOps, Cyber-Physical Systems**

I. Introduction

Cyber-Physical Systems integrate digital cyber computations with physical processes [14]. These systems are inherently complex, and their lifecycle can last up to 30 years in sectors such as railway or elevation [6]. In these systems, an increasing trend is to implement most of the functionalities through software. During the life-cycle of these systems, the software continuously evolves due to hardware obsolescence, requirement changes, vulnerabilities, bug corrections, etc. Consequently, this evolution requires reliable and automatic engineering methods for developing and operating CPSs.

With existing engineering practices for CPS, releasing and deploying new software versions is a time-consuming and error-prone activity. This is mainly due to the impossibility of thoroughly testing the software in a real environment. Furthermore, the deployment process itself is complex, as it is highly important to ensure that the CPS will be in a safe state when the software is updated. Besides, these systems often operate in dynamic and uncertain environment, what makes appropriate self-healing and recovery mechanisms necessary. These problems can be partially solved by implementing design-operation continuum methods for the software development life-cycle, instead of relying on traditional software development methods (e.g., the V model). Nevertheless, to achieve this in the CPS domain, radically new solutions to overcome the limitations of today's CPS development processes need to be adopted.

As an alternative, in the context of the Adeptness H2020 project [1] a reference architecture was proposed to enable Design-Operation Continuum activities in CPSs. The contribution of this paper is instantiating this architecture in an industrial case study from the elevation domain. A system of elevators is a complex CPS where all the aforementioned problems frequently arise. By using this architecture, we foresee significant enhancements in the software development, significantly reducing the software development cost while increasing its quality.

The rest of the paper is structured as follows. Section II presents the industrial case study in which we applied the architecture and the problems that they face. We explain the methodology for developing the architecture in Section III. Section IV presents the architecture based on microservice. Section V presents the prototypical implementation and a qualitative evaluation. We position our paper with the state-of-the-art in Section VI. Lastly, we conclude the paper and discuss the future avenues in Section VII.

II. Case Study and Problem Representation

Orona is a company dedicated to the designing, manufacturing, installing, and maintaining elevators, escalators and moving ramps. Elevator installations are complex CPSs that provide service to the passengers, considering the passengers' active passenger calls and the elevators' status. An overview the different elements of the CPS are depicted in Figure 1.

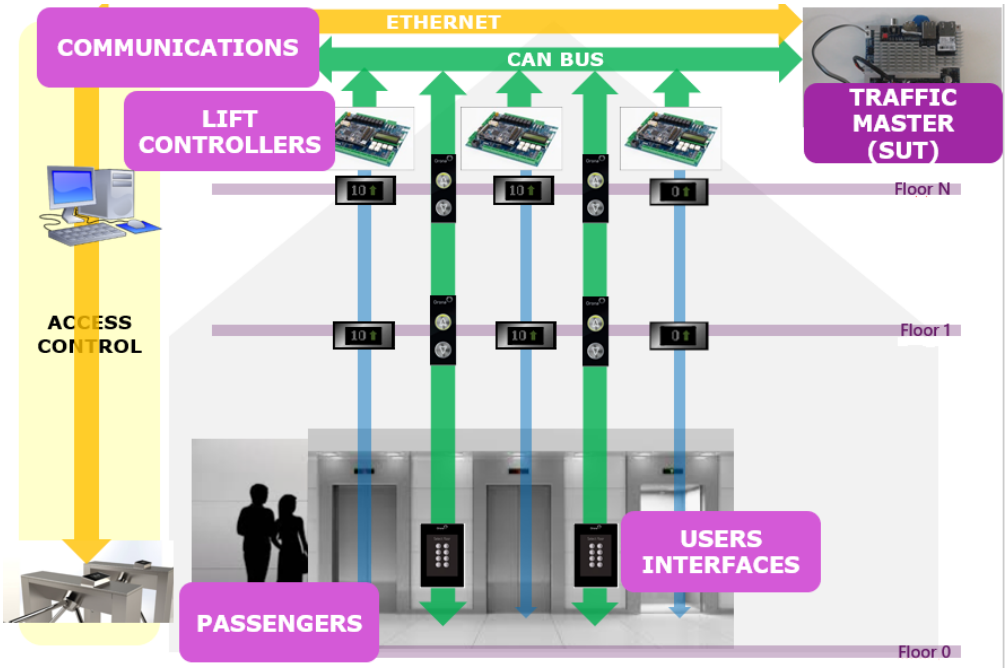


Fig. 1: Overview of the architecture of the Elevators installation

User interfaces allow passengers to introduce calls to the system in different ways. Conventional user interfaces consist solely on Up and Down call buttons, where only the floor on which the passenger is located and the direction of the journey are provided. Inside the elevator, the call panel allows introducing the destination floor of the passenger. On the other hand, destination input devices allow passengers to introduce the destination floor when making a call and then, passengers receive information about which elevator is going to attend them. The elevator call panel inside the elevator is not required in this case. There are also signalling panels indicating the current floor of the elevator and the expected journey. The interaction with the passenger can be extended with access control systems as well.

Each elevator is managed by a controller, which is responsible for the vertical and horizontal (doors control) movements of the elevator. When an elevator receives a landing or a car call (i.e., a call from inside the cabin), the controller decides the order of the stops and the opening and closing of the doors in each floor to attend all the calls by considering different information, including traveling direction, floor position, already assigned calls, etc. This information flows from one device to another through a (vertical) CAN bus.

The traffic master is the component that coordinates the user interfaces with the lift controller. It receives the information from access control devices through Ethernet and checks whether a passenger has the right to issue a particular call. It receives the passenger call from the landing call panels through the (horizontal) CAN bus, and the information of the status, position, etc, from each lift controller. With this information, the traffic master decides which is the best elevator to attend every call, considering different criteria, such as minimising the Average Waiting Time (AWT), the Journey Time (JT) or energy consumption[1]. Finally, the traffic master notifies the lift controller about the assigned call and indicates the assigned lift to the passengers.

This architecture does not provide support for design-operation continuum methods. When the traffic study is performed, there is usually a lack of real and precise data to adequately configure the system. In operation, automatic feedback mechanisms to improve the configuration and detect problems or unknown conditions are lacking. Consequently, when problems or unknown conditions arise in an installation (e.g., degradation in the AWT), the building owner is responsible for communicating the problem to Orona. Validation and deployment of the system are also semi-manual. Regarding validation, information from the operation is not accessible, so it is not possible to reproduce real situations in the laboratory, and the decision of whether a test case has succeeded or not is manual. Regarding deployment, the maintainer is responsible for configuring and updating new versions in the installation, which is also a manual process. In this paper, we present the extension of the architecture to support automatic deployment, continuous monitoring, and validation.

III. Architecture Development Methodology

Before designing the architecture, we defined a methodology that would enable defining the architecture systematically, considering the benefits of microservice architectures over

[1]The AWT is the average time that passengers wait until they enter in the lift. The JT is the average time that passengers wait to reach their destination. Both metrics are used to measure the performance of elevators systems.

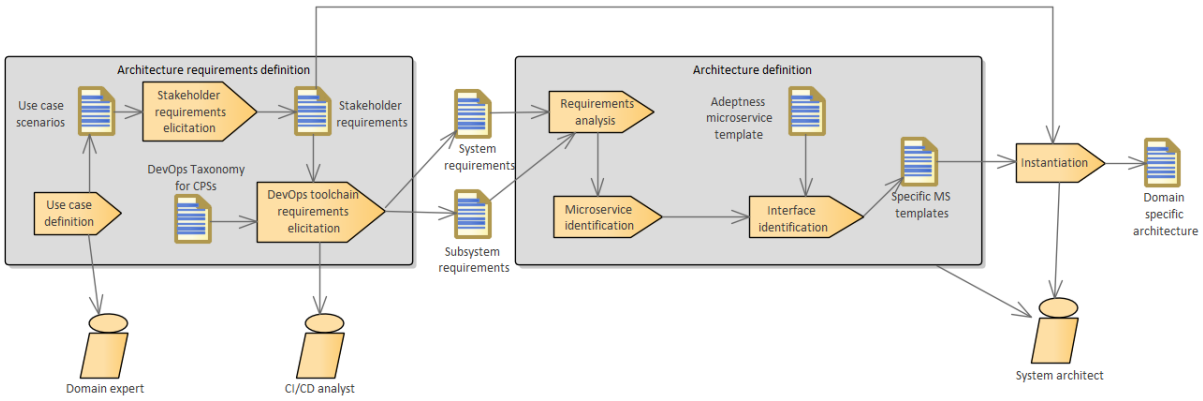


Fig. 2: Methodology for developing the microservices-based architecture and its instantiation to the Elevation domain

monolithic applications [18] and the need for domain awareness when developing an architecture [12]. The architecture development methodology is shown in Figure 2, and it consists of a total of six main steps:

- Use-case definition: First, there was a need to define the use-case scenarios that Orona wanted to handle. To this end, a domain expert defined a set of use-case scenarios, which can be found in [3].
- Stakeholder requirements: With these use-case scenarios, the stakeholder's requirements were elicited. A total of 56 requirements were elicited by an elevation domain expert, which are accessible in [2].
- DevOps toolchain requirements elicitation: By having as input a set of stakeholder requirements and a DevOps taxonomy for CPSs, developed in our previous work [6], two types of requirements were elicited: system requirements, which are those specific requirements from the overall architecture needed to satisfy the stakeholder requirements and subsystem requirements, which are those requirements specific to the subsystems (explained in the following section of the paper) necessary to satisfy system requirements. All these requirements along with the test cases that will be executed to validate them can be found in [2].
- Requirements analysis and microservice identification: With the elicited requirements, a first analysis was performed by a system architect, and a series of microservices were identified.
- Interface identification: For each microservice, the different interfaces were defined and integrated with the Adepteness microservice template, which is available for both C and Python.
- Instantiation: The last step referred to the instantiation and integration of all these templates.

IV. Architecture based on microservices

The HORIZON2020 Adeptness project [1] has proposed a microservice based architecture that will allow DevOps practices to be adopted in the context of CPSs. Microservices permit building a flexible architecture where services can be reused in different life-cycle stages and hardware, seamlessly deploying new services to all the installations and scaling the system. Each microservice within the proposed architecture shall be responsible for a specific well-defined function in the life-cycle of a new software release, and shall provide different lightweight communication mechanisms. Each microservice will provide both synchronous (i.e., HTTP) and asynchronous (i.e., MQTT) communication, offering common interfaces for every microservice within the system and custom interfaces for microservices with specific roles.

A. Common interfaces

All microservices within the architecture provide a set of basic asynchronous and synchronous communication endpoints, regardless to the role of the microservice. These endpoints offer basic information about the execution status, health and performance. The synchronous interfaces allow other services to request microservices' health status, while asynchronous interfaces allow microservices to publish relevant data without knowing the receiver of the messages. The following interfaces are provided by the template developed within the HORIZON2020 Adeptness project [1].

1) Synchronous communication:

- /adms/v1/ping [GET]: Ping service to check that the service is alive. Returns an empty 200 response if the microservice is working correctly.
- /adms/v1/info [GET]: Provides basic information about the microservice. It returns a JSON object containing the microservice ID and microservice role within the architecture.
- /adms/v1/performance [GET]: Provides CPU and memory usage metrics. It returns a JSON object containing the free and allocated memory and the CPU usage.
- /adms/v1/status [GET, PUT]: Permits getting or changing the execution status of the microservice. GET calls to this endpoint will return a JSON object containing the status of the microservice. Changes to the microservice status will be performed by sending a JSON object with the

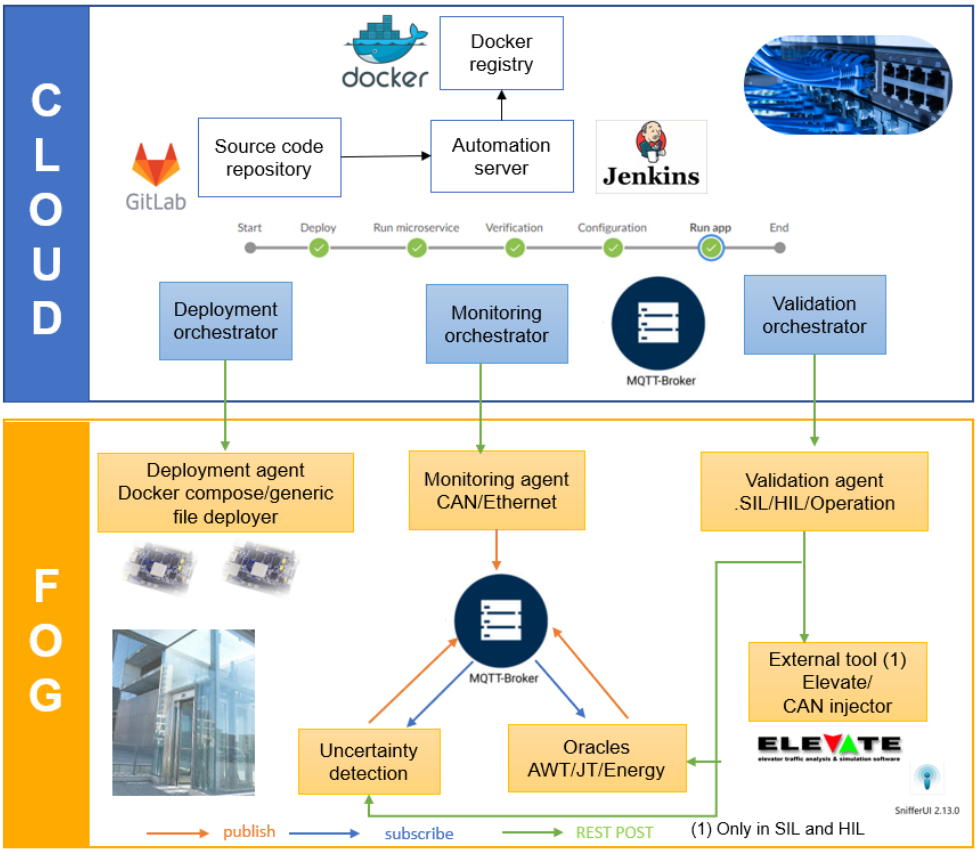


Fig. 3: Overall overview of the microservice-based architecture

desired state. The possible states for the microservice are "Ready" and "Running".

2) Asynchronous communication:

- /adms/v1/discovery [PUB]: On microservice launch, the microservice publishes a hello message in this topic including the identifier, microservice role and its MQTT and REST endpoints, defined as a JSON object.

B. Architecture – Instantiation in the use-case

In Figure 3 we present the instantiation of the Adeptness microservice architecture for ORONA. In particular, we provide microservices for continuous deployment, monitoring and validation, recovery and uncertainty detection.

The main subsystems composing the architecture for ORONA are the following:

1) Automation server: The automation server is in charge of the orchestration of the tasks to be performed by the different subsystems. It interacts with the source code repositories to monitor any changes on the deployment, monitoring or validation plans. When a new plan is updated, the automation server performs the actions to generate the required artifacts, stores the generated Docker images in the Docker registry and pushes the configurations or plans to each subsystem.

2) Deployment subsystem: The deployment subsystem is responsible for downloading, and eventually decompressing and executing, the different microservices and artifacts needed to perform the validation in each of the targets or edge nodes. The deployment subsystem executes a deployment plan and must be aware of the status of the deployment in each node. The plan contains the information regarding the components to be deployed, the repository where they are located in, and the node(s) where they should be deployed. The deployment subsystem is capable of deploying two different types of components, containerized microservices and generic files, and it is composed of two different microservices:

- Deployment orchestrator: The deployment orchestrator receives the deployment plan from the automation server and parses the plan in order to execute it. There is only one instance of this microservice within the architecture and it is usually located in the cloud. The orchestrator sends the deployment instructions to the deployment agents installed in each node by MQTT.
- Deployment agent: The deployment agents must be installed in each edge node to perform the actual deployment of the necessary artifacts. Since there are two different types of components that may be deployed, two types of deployment agents have been designed: a docker-

compose based deployer to deploy docker containers, and a generic deployer to deploy any kind of file, e.g. an executable file, a library, or a zip file. In the latter case, the deployer can perform the actual deployment of the zip file, decompress it, and execute the selected executable file.

3) Monitoring subsystem: The monitoring subsystem supports the configuration of the monitors according to a monitoring plan. This plan specifies the source (physical interfaces, file system, ...) to obtain the data from as well as the value extraction mechanism. This subsystem provides access to telemetry data retrieved from different sources so that other subsystems can subscribe to this data and use it to take decisions. This subsystem consists of two microservices:

- Monitoring orchestrator: This microservice, deployed in the cloud, handles the parsing of the monitoring plan sent from the automation server, and configures all the monitoring agents indicated in the plan accordingly through their HTTP API. The plan specifies the parameters that each monitoring agent needs to specify the actual data source connection parameters (e.g., the CAN baud rate) and the variables to monitor.
- Monitoring agents: The monitoring agents, deployed at the edge nodes, are responsible for reading the operational variables from the different sources and publishing them asynchronously. The monitoring agents can be configured through a common HTTP API, which allows the specification of the variables to be monitored (name and needed parameters to obtain the data) and optionally, the configuration of subscriptions. The concept of subscription is similar to OPC-UA, i.e., a group of variables that are notified asynchronously as events, with the same publishing rate. In this sense, a service (e.g., an oracle) that needs to be notified about the changes of a set of variables can configure a subscription, specifying the publishing rate for those variables.
  There are specific monitoring agents for different data sources. In the case of Orona, two different monitoring agents are used:
  - CAN monitor: The CAN monitor allows the monitoring of the operational variables shared through a CAN field-bus, which may be configured through an HTTP API. For each variable to be monitored, three parameters must be configured: (1) the name of the variable, (2) the identifier of the CAN frame where the variable is published, and (3) the mask to be applied to the frame to actually read the variable. Then, when the monitor starts, it begins to publish the variables asynchronously through MQTT, following the standardized senML[2] payload format.
  - Instrumented code monitor: This is a special monitor type that supports the monitoring of variables that are not exported in any field bus but are needed by the oracles to raise a verdict, for instance, the internal variables of the traffic algorithm that are usually inspected in debugging mode. To do so, a library which publishes code variables through MQTT has been developed. The developer can use it to publish the internal code variables needed by the oracles into the MQTT broker, in the same senML format used by the rest of the monitors.

4) Continuous Validation subsystem: The continuous validation subsystem supports verification and validation activities at Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Hardware-in-the-Loop (HiL), and Operation. At the MiL test level, the software that controls the physical part of the CPS is a model. At the SiL test leve, this model is replaced by executable software. At the HiL test level, the software is integrated with the real-time infrastructure (e.g., real target processor and operating system) and the physical part emulated within a real-time test bench. For the SiL level, Orona uses Elevate, a domain specific simulator for validation. At the HiL level, a hybrid infrastructure where some components are real and others virtual is used. At this test level, the tests are performed in real-time, using all the real infrastructure, including real communication buses and a real-time operating system. The main microservices used for the continuous validation subsystem for Orona's case are the following:

- Validation orchestrator microservice: Located in the cloud, the validation orchestrator manages the execution of a validation plan by communicating with the validation agents. A validation plan can require validations at different test levels.
- Validation agents for SiL, HiL and Operation: these microservices launch validations at the SiL and HiL test environments, as well as in production installations. For the execution of a validation, child test oracles that provide the verdict are activated. Validation agents in SiL and HiL also manage the tools required for simulating test inputs. When an oracle provides a verdict, it notifies the validation orchestrator microservice.
- Oracle microservice: This microservice encompasses a set of test oracles that validate that the CPS behaves as expected. Many of these test oracles are based on domain-specific Quality-of-Service (QoS) measures that are collected from the monitoring microservices. Among these QoS measures, for the elevation domain, the most important ones are the Average Waiting Time (AWT), the Journey Time (JT) and the energy consumption. Each of these test oracles provide a verdict that indicates to which extent the CPS behaves as expected. Different test oracles have been developed, such as those based on metamorphic relations for the SiL and HiL test levels [7], and some based on machine-learning that predict the maximum AWT and JT a system of elevators should have at each moment.
- Uncertainty detection microservice: This microservice supports the automated detection of unforeseen situations in the different life-cycle stages of CPSoS using data

[2]https://tools.ietf.org/html/rfc8428

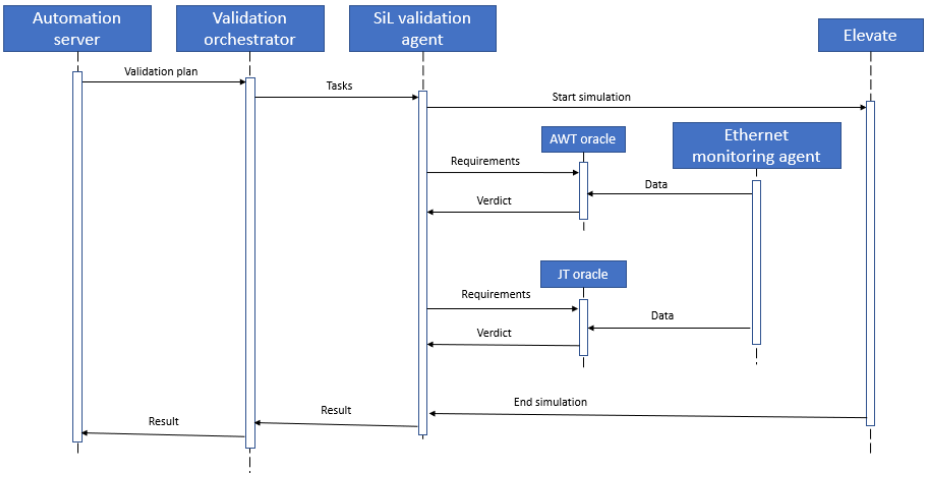


Fig. 4: Overview of the validation process at SiL level of the prototype

from both operation (e.g., live data) and design time (e.g., test logs) with passive and active machine learning techniques. This service supports the validation microservice with uncertainty related test oracles that will be learned from data. Various uncertainties exist in the elevator use case, such as (1) *Passenger data*: Examples include when a passenger arrives at which floor? How much a passenger weighs? (2) *Environment*: Examples include delays in hardware such as motor delay and levelling delay; (3) *Outputs/Quality of Lift Services*: Examples include uncertainties in waiting and transit times.

- External tool microservices for SiL and HiL test levels: This microservice allows launching domain-specific tools required to handle the execution of tests. Two external tool microservices have been instantiated for testing Orona's dispatching algorithm: (1) Elevate (a domain specific simulation tool) that is used for SiL validations and (2) a CAN Bus frames injector for HiL validations.

V. Prototype Implementation and Preliminary Evaluation

To analyse the benefits of the architecture, a preliminary prototype of the architecture for SiL validation in Orona has been developed[3], by setting a pipeline structure in Jenkins. This pipeline starts with the deployment of the components from a deployment plan. If the deployment is successful, the pipeline continues by configuring and starting the monitors, and finally the validation microservices are configured and executed to obtain a verdict. A Docker image for each microservice has been built and pushed to a Docker registry, particularly to the Gitlab container registry.

The deployment pipeline is continuously querying the repositories for changes in the deployment plan, and whenever a change is detected, the deployment pipeline is launched. The deployment pipeline launches a slave agent that logins to the Gitlab container registry and fetches the latest versions of the docker images for the specified components. These images are then started and configured to forward different ports on the slave agent node. The deployment pipeline waits for readiness of every launched microservice by calling to their /adms/v1/ping REST endpoint. The deployment plan consists of a JSON formatted file where each node, identified by its IP address, is assigned one or several components. When the microservices are ready, the monitoring pipeline is called as a build step, or in case of failure, docker containers are stopped and cleaned.

The monitoring pipeline configures the monitoring microservice, by setting the output topic where different values will be published through the HTTP API. When the microservice is ready, the publication of values starts through another call to the HTTP API. If every call succeeds, the validation pipeline is launched as a build step. As in the previous pipeline, on failure, the environment is cleaned and docker containers stopped.

Similar to the monitoring pipeline, the validation pipeline sets up the validation microservice by setting up the input topic where the monitoring microservice is leaving its values, and the conditions that will be evaluated to raise a verdict for the evaluation. After the microservice is configured and started, the validation takes place and the pipeline continuously polls the microservice for a verdict. If the verdict has been marked as passed, the pipeline will succeed, and will fail otherwise. Figure 4 shows an overview of the process.

Table I shows a qualitative overview of the benefits that the proposed architecture can bring during the DevOps activities in Orona. At the deployment level, the automation of the tasks that Jenkins brings reduces effort and saves time for developers by reducing the manual tasks, such as executable generation and deployment for different stages. At the monitoring level, the architecture allows continuously monitoring data from the different sources at all stages, gaining more understanding of the system while reducing the effort. Finally, at the validation

[3]A video of the prototype is available at https://youtu.be/uoq9n9k4kgc

TABLE I: Expected benefits of the architecture in the Deployment, Monitoring and Validation activities for different life-cycle stages in Orona

DEVOPS ACTIVITIES	CURRENT PROCESS	WITH MICROSERVICE ARCHITECTURE	EXPECTED BENEFITS
Deployment			
(SiL) Generate the dll for the domain specific simulator (SiL) Copy files for the simulator (HiL) Compile for the target (HiL) Deploy in the target	Manual compilation & copy	Jenkins pipeline	Not dependence on developer Automatic trigger of the compilation and deploy
(Operation) Deploy in the real installation	Manual deployment by the maintainers	Remotely and automatically deploy a new software release	Effort saving by automatically deploy a new software version. Control over the configuration of the release
Monitoring			
(SiL) Traces of the simulation tools (SiL, HiL, Operation) Traces in the code (HiL, Operation) Monitor the communication buses	Traces are recorded in a txt file Data provided by the simulator in excel and word CAN frames recorded on demand	Data from the code, the simulators and the communication buses will be published by MQTT	Effort saving in analysis of problems in installations Continuous remote monitoring
Validation			
(SiL, HiL) Define test cases: Unitary and QOS (i.e. AWT)	Unitary test cases manually defined QoS test cases from theoretical profiles	Unitary test cases manually defined Automatic profiles from real data of installations.	Test cases defined from real profiles more likely to reproduce real problems
(SiL/HiL/Operation) Execute the validations	Manual configuration of installations Manual trigger in SiL/HiL Manual validation in operation by the maintainer	Set of available configurations for SiL/HiL Automatic Jenkins pipeline for SiL/HiL Continuous validation in operation	Effort saving by automatically validate software release Increase the number of bug detected by continuously validate the software even in operation
(SiL/HiL/Operation) Decide the verdict for the validation	Manual	Reusable oracles	Increase the number of bugs detected Minimise dependency on individuals
(SiL/HiL/Operation) Locate a bug	Visual inspection of logs Manual Debugging	Automatically reproduce a scenario in the laboratory using the information of the monitoring subsystem	Increase the number of bugs detected Effort saving in analysing problems in installations

level, continuously validating the system in an automated manner for all stages increases the number of bugs detected while reducing the time to prepare the validation infrastructure.

VI. Related Work

Microservice-based architectures are spreading in the Internet of Things (IoT) and CPS domains due to the high suitability of this paradigm for these fields, as they share some goals (e.g., lightweight communication, independent deployable software, etc.) [9].

Thramboulidis et al. [21] and Alam et al., applied microservice-based architectures to exploit its benefits in CPSs involved in industrial use cases. Specifically, Thramboulidis et al., [21] proposed a framework which uses model-driven engineering to semi automate the use of microservices on manufacturing systems, remarking the flexibility of such an architecture for plant processes. In [5], the authors combined Docker and microservices using a distributed and modular architecture to execute Industrial IoT (IIoT) applications, showing its validity for deployments on time-sensitive scenarios. These works propose developing microservice-based applications for CPSs, but do not use microservice-based solutions in the development process tasks.

As mentioned, the development of CPSs has typically suffered from long development life-cycles [4]. DevOps practices are now gaining attention in the CPS domain, and many works are focusing on applying different techniques such as Model-Driven Engineering [10] or Digital Twins [22] to ease and

enhance DevOps activities on CPSs. Many tools focus on specific life-cycle stages of CPSs, such as deployment [19] [11] [20], monitoring [17] [23] or validation [8] [15], but do not have whole life-cycle management capabilities, requiring the use of multiple tools to handle all life-cycle phases.

There are also some works which exploit the benefits of microservices to perform DevOps activities. [13] proposed applying the microservice design principles for software deployment and [16] presented a monitoring tool based on microservices, but these works focus on cloud infrastructures management, rather than CPSs.

VII. Conclusion and Future Work

In this work, we have instantiated part of the reference architecture presented in the H2020 Adeptness project for the Orona use case, and a prototype of the architecture for SiL validation has been developed.

Automation of DevOps activities has paramount relevance specially in CPSs, where the life-cycles are so long and the development tasks so fragmented, that easing and speeding these tasks may have a huge impact. For instance, automating software deployment can reduce maintenance effort in deploying the software in each device manually and continuous monitoring and validation may be helpful for understanding the system behaviour at run-time. esides, a microservice-based architecture offers high flexibility, simplifying the architecture's adaptation to different life-cycle stages, and allows scaling the solution to large-scale systems.

In the future, we plan to continue extending the architecture to support Orona's development activities in all life-cycle stages. We also plan on including additional mechanisms to ensure correct software deployment, such as applying Machine Learning techniques to detect performance problems in new software releases.

Acknowledgment

This publication is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319. This work has been partially supported by the Basque Government through the Elkartek program under the DIGITAL project (Grant agreement no.KK/2019-00095). Aitor Arrieta and Goiuria Sagardui are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1326-19), supported by the Department of Education, Universities and Research of the Basque Country.

References

[1] Adeptness project webpage: https://www.adeptness.eu/.

[2] Requirements-and-validation-tests – https://adeptness.eu/wp-content/uploads/2020/09/D1.1-ANNEX-A-Requirements-and-validation-tests.pdf.

[3] Requirements-and-validation-tests – https://adeptness.eu/wp-content/uploads/2020/11/D1.1-REQUIREMENTS$_v$1.1.*pdf*.

[4] Pekka Abrahamsson, Goetz Botterweck, Hadi Ghanbari, Martin Gilje Jaatun, Petri Kettunen, Tommi J. Mikkonen, Anila Mjeda, Jürgen Münch, Anh Nguyen Duc, Barbara Russo, and Xiaofeng Wang. Towards a Secure DevOps Approach for Cyber-Physical Systems. *International Journal of Systems and Software Security and Protection*, 11(2):38–57, 2020.

[5] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, 2018.

[6] Jon Ayerdi, Aitor Garciandia, Aitor Arrieta, Wasif Afzal, Eduard Enoiu, Aitor Agirre, Goiuria Sagardui, Maite Arratibel, and Ola Sellin. Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 280–290. IEEE, 2020.

[7] Jon Ayerdi, Sergio Segura, Aitor Arrieta, Goiuria SagarduiMaite Arratibel, and Maite Arratibel. Qos-aware metamorphic testing: An elevation case study. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 104–114. IEEE, 2020.

[8] Sreram Balasubramaniyan, Seshadhri Srinivasan, Furio Buonopane, B. Subathra, Jüri Vain, and Srini Ramaswamy. Design and verification of Cyber-Physical Systems using TrueTime, evolutionary optimization and UPPAAL. *Microprocessors and Microsystems*, 42(2016):37–48, 2016.

[9] Bjorn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2016-November, 2016.

[10] Benoît Combemale and M. Wimmer. Towards a model-based devops for cyber-physical systems. In *DEVOPS*, 2019.

[11] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre Emmanuel Novac, Stephane Lavirotte, Jean Yves Tigli, and Arnor Solberg. GeneSIS: Continuous orchestration and deployment of smart IoT systems. In *Proceedings - International Computer Software and Applications Conference*, volume 1, pages 870–875. IEEE Computer Society, jul 2019.

[12] Javad Ghofrani and D. Lübke. Challenges of microservices architecture: A survey on the state of the practice. In *ZEUS*, 2018.

[13] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure DevOps. *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, pages 202–211, 2016.

[14] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[15] Elena Markoska and Sanja Lazarova-Molnar. Towards smart buildings performance testing as a service. *2018 3rd International Conference on Fog and Mobile Edge Computing, FMEC 2018*, pages 277–282, 2018.

[16] Marco Miglierina and Damian A. Tamburri. Towards omnia: A monitoring factory for quality-aware devops. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, page 145–150, New York, NY, USA, 2017. Association for Computing Machinery.

[17] K. Monisha and M. Rajasekhara Babu. *A novel framework for healthcare monitoring system through cyber-physical system*. Springer Singapore, 2019.

[18] R. O'Connor, Peter Elger, and Paul M. Clarke. Continuous software engineering—a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29, 2017.

[19] Nenad Petrovic and Milorad Tosic. SMADA-Fog: Semantic model driven approach to deployment and adaptivity in fog computing. *Simulation Modelling Practice and Theory*, 2019.

[20] Luis F Rivera, Norha M Villegas, Gabriel Tamura, Miguel Jiménez, and Hausi A Müller. UML-driven Automated Software Deployment. *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 257–268, 2018.

[21] K. Thramboulidis, D. C. Vachtsevanou, and A. Solanos. Cyber-physical microservices: An iot-based framework for manufacturing systems. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 232–239, 2018.

[22] Miriam Ugarte Querejeta, Leire Etxeberria, and Goiuria Sagardui. Towards a devops approach in cyber physical production systems using digital twins. In *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, pages 205–216. Springer International Publishing, 2020.

[23] Michael Vierhauser, Jane Cleland-Huang, Sean Bayley, Thomas Krismayer, Rick Rabiser, and Pau Grünbacher. Monitoring CPS at runtime - A case study in the UAV domain. *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018*, pages 73–80, 2018.

A.4 Machine Learning-based Test Oracles for Performance Testing of Cyber-Physical Systems: An Industrial Case Study on Elevators Dispatching Algorithms

This paper was published at the Journal of Software: Evolution and Process journal in 2022. The full citation:

Gartziandia, A., Arrieta, A., Ayerdi, J., Illarramendi, M., Agirre, A., Sagardui, G., Arratibel, M. (2022). Machine learning-based test oracles for performance testing of cyber-physical systems: An industrial case study on elevators dispatching algorithms. Journal of Software: Evolution and Process, 34(11), e2465.

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Machine Learning-based Test Oracles for Performance Testing of Cyber-Physical Systems: an Industrial Case Study on Elevators Dispatching Algorithms[†]

Aitor Gartziandia[1] | Aitor Arrieta*[2] | Jon Ayerdi[2] | Miren Illarramendi[2] | Aitor Agirre[1] | Goiuria Sagardui[2] | Maite Arratibel[3]

[1]Information and Communication Technologies, Ikerlan, Mondragon, Spain
[2]Software and Systemcs Engineering, Mondragon University, Mondragon, Spain
[3]Control Systems, Orona, Hernani, Spain

Correspondence
*Aitor Arrieta, Email: aarrieta@mondragon.edu

Present Address
This is sample for present address text this is sample for present address text

Summary

The software of systems of elevators needs constant maintenance to deal with new functionality, bug fixes or legislation changes. To automatically validate the software of these systems, a typical approach in industry is to use regression oracles, which execute test inputs both in the software version under test and in a previous software version. However, these practices require a long test execution time and cannot be re-used at different test phases. To deal with these issues, we propose DARIO, a test oracle that relies on regression machine-learning algorithms to detect both functional and non-functional problems of the system. The machine-learning algorithms of this oracle are trained by using data from previously tested versions to predict reference functional and non-functional performance values of the new versions. An empirical evaluation with an industrial case study demonstrates the feasibility of using our approach. A total of five regression learning algorithms were validated by using mutation testing techniques. For the context of functional bugs, the accuracy when predicting verdicts by DARIO ranged between 95% to 98%, across the different scenarios proposed. For the context of non-functional bugs, were competitive too, having an accuracy when predicting verdicts by DARIO ranging between 83% to 87%.

KEYWORDS:
Test Oracle, Regression Testing, Machine- Learning, Performance Testing

1 | INTRODUCTION

Cyber-Physical Systems (CPSs) combine digital cyber technologies with physical processes [21]. The software of such systems has an increasingly long life-cycle, requiring maintenance over several years. This is the case of Orona, one of the largest elevator companies worldwide, whose software needs to be constantly maintained and updated to deal with (1) new functional and non-functional requirements, (2) the correction of bugs, (3) legislative changes and (4) hardware obsolescence and system degradation [7]. In a system of elevators, one critical subsystem is its traffic master, which is in charge of managing the passenger flow in a building. It is composed of several software modules, such as the dispatching algorithm, which is the component in charge of deciding which elevator must attend each passenger by considering several aspects (e.g., estimation of the waiting time of each passenger or estimated time required to arrive to destination). New generations of these algorithms are also starting to consider additional aspects, such as energy consumption. The algorithms also include certain functionalities based on the building, such as

[†]This publication is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319. Jon Ayerdi, Aitor Arrieta, Miren Illarramendi and Goiuria Sagardui are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1326-19), supported by the Department of Education, Universities and Research of the Basque Country.

controlling access to disable specific floors to unauthorized passengers or special calls for handicapped persons (e.g., by assigning additional space in the elevator or assigning them the closest elevator). Orona, one of the leading elevator companies in Europe, has a large suite of dispatching algorithms that are in constant maintenance and evolution to solve different customers' demands.

When changes are performed in these algorithms, Orona has a well established systematic verification and validation process at different test phases (Software-in-the-Loop (SiL), Hardware-in-the-Loop (HiL) and Operation). At SiL and HiL, two kinds of tests are executed. On the one hand, unitary tests aim to test specific scenarios and verify that the functional requirements of the system are met.[1] Such tests are relatively short, lasting from 30 seconds to 5 minutes. On the other hand, Orona executes what is internally known as full-day passenger tests, which have the objective of verifying the functional performance of a systems of elevators (what in the elevation industry is known as the Quality of Service (QoS) of a system of elevators [11]). In such case, the objective is to detect functional performance failures (i.e., the failures that cause a degradation in the QoS of the system). An example of a typical functional performance metric in this domain is the Average Waiting Time (AWT), which measures the average time that passengers need to wait until an elevator attends their call. To determine whether the quality of an algorithm is satisfactory or not, a regression test oracle is used. Specifically, the same traffic profile (i.e., test input) is executed in a previous version of the dispatching algorithm, and compared with the dispatching algorithm version under test. If the performance metric under study is better or similar, the test is classified as *PASS*. Conversely, if the performance metric under study is not good enough, the test is classified as *FAIL*.

This regression test oracle has two main problems. Firstly, the test needs to be executed both in the algorithm under test as well as in a previous version of the algorithm. This can be time-consuming as the execution is performed using simulation-based testing at system level and the test inputs usually simulate full-day passenger traffic profiles. This problem is exacerbated at the HiL test phase, where the simulation is performed in real-time, and thus, in order to simulate a full-day traffic profile, two full-days are required (i.e., one full-day for the new software version under test and another one for the previous software version). The second problem involves that the test oracle cannot be re-used for testing the new version "on-the-fly" in operation. This is a problem because the faults not detected in the validation phases can manifest in the real installation. The last problem relates to the difficulty of detecting non-functional bugs, which pose significant challenges as compared to functional bugs; these type of bugs typically result in an overuse of computational resources [20], which are scarce in CPSs, and detecting them is starting to become a priority inside Orona, our industrial partner.

The barrier between functional and non-functional bugs in CPSs is blurry, and therefore, we make a distinction between functional and non-functional performance bugs. On the one hand, the purpose of the elevator dispatching algorithm is outputting optimal assignments. Consequently, a sub-optimal assignment (as opposed to being slow at issuing the assignment) is considered an incorrect output for the dispatcher, and is therefore classified as a functional performance failure. This is analogous to a GPS providing a valid but not optimal route between two points. On the other hand, non-functional performance bugs are those that do not cause a functional failure, but result in an overuse of computational resources [20]. Unfortunately, CPSs are not exempt of non-functional performance bugs, which can occasionally lead to more severe consequences, such as functional failures. In our conference-version paper published on Automation of Software Test (AST) conference [3], we explored the use of machine-learning to tackle the test oracle problem. Specifically, we proposed DARIO (Dispatching AlgoRIthm Oracle), a test oracle that relies on regression learning algorithms to automatically validate elevators dispatching algorithms. Instead of using regression oracles, DARIO trains a machine learning algorithm by using data from a previously tested dispatching algorithm version to detect functional performance bugs. In this paper, besides functional bugs, we explore whether DARIO is capable of detecting non-functional performance bugs.

Our approach provides several advantages. Firstly, the training process of the employed regression learning algorithms is much faster than employing a regression test oracle. While the regression learning algorithms used in DARIO take a few seconds to train, the regression test oracles used in practice may take from minutes to hours at the SiL test phase and days at the HiL test phase. Secondly, DARIO can be used as streamlined oracle at SiL, HiL and in operation, permitting the detection of potential inconsistencies within the real system. Thirdly, DARIO returns a quantitative verdict value over the simulation time, which provides information of how close the system was from failing at each simulation step or how severe a fault was. This opens the possibility to include new verification and validation paradigms in the context of dispatching algorithms, including falsification-based test generation [51, 56]. It can also be used for on-line testing, which could save a significant amount of testing time by stopping the simulation if a severe fault is detected, as proposed by Menghi et al. in a recent study [52].

This paper is an extension of our AST 2021 paper[3]. Specifically, we build upon the conference-version paper [3] from the the following perspectives:

- We adapt DARIO, our test oracle that works on top of machine-learning algorithms to detect both, functional and non-functional performance bugs. We show that our approach is effective in both cases by carrying out a new empirical evaluation based on performance mutation testing.

[1] Notice that unitary tests are different to those unit tests, which aim at testing individual software modules

- We significantly extend the background and related work sections. Specifically, in the background section we provide a distinction between two types of performance metrics for CPSs (i.e., non-functional performance and functional performance). In addition, we include a new section to explain background theory related to machine-learning algorithms.
- We extract new lessons learned and open challenges, which can be used by future researchers as starting point for devising novel approaches.

We can summarize our contributions as follows:

- We propose DARIO, a novel approach for building test oracles for performance tesing of dispatching algorithms. DARIO works on top of regression machine-learning algorithms. These algorithms aim at predicting the reference performance metric (either, functional (e.g., AWT) or non functional (e.g., execution time) values of a system of elevators based on the passenger traffic data.
- We perform an empirical evaluation by using an industrial case study provided by Orona, one of the leading elevator companies in Europe. In this evaluation, we used mutation testing to determine how accurate DARIO was when compared to traditionally employed regression test oracles. We compared five regression learning algorithms. For functional performance bugs, the Regression Tree algorithms performed the best overall. For non-functional performance bugs, there was no a clear winner, but all the algorithms performed well.
- We provide a set of lessons learned and open challenges from applying DARIO in practice.

The rest of the paper is structured as follows: Section 2 summarizes the relevant background on performance testing and machine learning. Section 3 provides a description of the case study and the current software testing process at Orona. Section 4 presents our approach to automatically test elevators dispatching algorithms. Section 5 evaluates our approach by means of an empirical evaluation with an industrial case study. Section 6 discusses the results, the lessons learned and the open challenges based on the performed evaluation and the discussions we had with domain experts. We position our work with the current state of the art in Section 7. Lastly, Section 8 concludes our paper.

2 | BACKGROUND

2.1 | Performance metrics for cyber-physical systems

The analysis of the system performance has not been as widespread as the analysis of functional requirements, as deviations in performance may not be as harmful as functional failures in most cases. Furthermore, the performance of a system is often harder to verify due to the difficulty to evaluate this type of properties, combined with a lack of specific requirements in many cases [77]. This makes it necessary to integrate performance analysis activities in all of the system's life-cycle stages, from the system design to operation [17], by employing mechanisms such as static code analysis, performance testing, and run-time monitoring [58, 43].

Ideally, performance requirements should be defined in a requirements document in a concrete, verifiable manner, such that the compliance of the system can be evaluated accurately. Nevertheless, in practice, these requirements are often not defined or ambiguous. Some challenges of performance requirements specification are establishing requirements that can actually be checked in a concrete and precise manner and selecting the appropriate inputs to use when verifying each requirement [77]. The lack of verifiable specifications makes it difficult to assess whether the performance of the system is acceptable or not. This difficulty in verifying the validity of the observed system behavior is known as the *test oracle problem*, and it is recognized as one of the fundamental problems of software testing [12].

We make a distinction between two types of system performance: Non-functional performance and functional performance. While non-functional performance metrics only measure the efficiency of a system's implementation, functional performance metrics are derived (indirectly) from the system output. We argue that both types of properties share many aspects in common in the context of testing and verification, and that performance testing techniques can be applied to either of them in most cases. In this case, this can be considered either functional or non-functional testing based on whether we use functional or non-functional performance metrics. Although performance testing is usually considered non-functional testing, we must consider that the functional performance properties of the system are derived from its output.

2.1.1 | Non-functional performance

Non-functional performance refers to the usage of computational resources of the system, such as response time, CPU usage, memory usage or energy consumption [20]. The non-functional performance of the software in domains such as web pages or mobile apps is very relevant, as performance issues can affect the usability of the system, causing a loss of users and/or money. For CPSs, this type of properties are an even larger concern, since some non-functional performance issues can escalate to major functional failures, and even safety implications; For instance, an

Unmanned Aerial Vehicle (UAV) may run out of batteries during its flight due to an excessive energy consumption, resulting in a loss of functionality, and potentially some additional damage from a crash if there are no additional safety measures.

2.1.2 | Functional performance

Functional performance refers to the properties derived indirectly from the output of the system, rather than the system's efficient usage of the computational resources. This type of properties are typically used for evaluating the adequateness or quality of a system's output due to the lack of a more direct way to evaluate its behavior. As an example, an optimizing compiler's output can be evaluated based on the non-functional performance of the generated program (e.g. execution time), but such properties are actually derived from the functional behavior of the compiler (the code it outputs), and not its non-functional performance (e.g. the time it takes for the compiler to execute). Similarly, CPSs such as multi-elevator systems are typically evaluated based on Quality of Service (QoS) metrics such as the waiting times for the passengers, which are mostly derived from the functional behavior of the system (call assignments to elevators) rather than its non-functional performance.

2.2 | Machine-learning algorithms

Machine-learning is a sub-field of Artificial Intelligence (AI) that provides the software with the ability to "learn" from observed data with the goal of making decisions without being explicitly programmed to do so [44]. These approaches construct models by observing data using statistical analyses, where learning begins with the search for patterns inside the data used for training the algorithm [25].

Machine-learning can be categorized in the categories of supervised, unsupervised and reinforcement learning [2]. In supervised learning, the training data consists of some input vectors with their respective labels, which indicate the output expected for each input vector [2]. The objective of these algorithms is to learn to predict the corresponding output for the given inputs, which could have been unforeseen to the algorithm before, by modifying their internal values [44]. Within this category we can distinguish two subcategories: (1) classification algorithms, where the aim is to map input values to a finite number of categories and (2) regression algorithms, where the output values consist of continuous variables. Unlike supervised learning algorithms, unsupervised learning algorithms do not have labeled data. Their goal is to cluster the data by identifying patterns on the similarities and dissimilarities [25]. Lastly, reinforcement learning approaches aim at selecting actions with certain goals [25]. They use feedback on the effect of the action taken (i.e., rewards), which is later used to improve the estimation and iteratively perform more effective actions [25]. In our study, we use supervised learning algorithms aimed at solving a regression problem.

3 | CASE STUDY

3.1 | The System Under Test

Elevators are CPSs composed of different subsystems that collaborate to transport passengers vertically within a building. Figure 1 shows the architecture of a system of elevators from Orona. Each elevator is managed by a controller, which is responsible for managing both vertical (floor to floor) and horizontal (doors opening and closing) movements of each elevator. The traffic master is the software system that coordinates the controllers to handle the calls made by the passengers. This system is responsible for (1) the execution of the dispatching algorithm (i.e., the allocation of passenger calls to the available cars), (2) the overall system signaling (i.e., registration of the calls, information to the passengers, etc.), and (3) additional functionalities such as access control (i.e. permission for the passengers to access certain floors) or management of special operating modes. Locally, the communication among the different controllers is done through a Controller Area Network (CAN) bus. The CAN bus is an industrial field bus, originally developed by Bosch for the automotive domain, which allows for a real-time communication for distributed embedded systems. In addition, the traffic master can be connected through Ethernet to external computers in order to enable some advanced features (e.g., passenger access control, remote system configuration, etc.).

The System Under Test (SUT) of this paper is the traffic dispatching algorithm. This algorithm is executed periodically to select the optimal elevators to serve the active floor calls. It receives the passenger calls, access control information, and the status (position, direction, etc.) of each elevator as inputs, and then determines which the best elevator to attend every call is. This is done by considering different criteria, such as the reduction of the Average Waiting Time (AWT), the Journey Time (JT) or energy consumption. The AWT refers to the average time that passengers wait until they enter in the elevator, and the JT is the average time that passengers wait inside the elevators until they reach their destination. Both are functional performance metrics used to measure (and test) new versions of the traffic dispatching algorithm. Depending on the algorithm, already allocated floor calls can be reallocated (assigned to another car) to optimize the overall cost function. Due to its effect on the aforementioned fitness criteria, this component is critical to ensure that an elevator installation performs correctly.

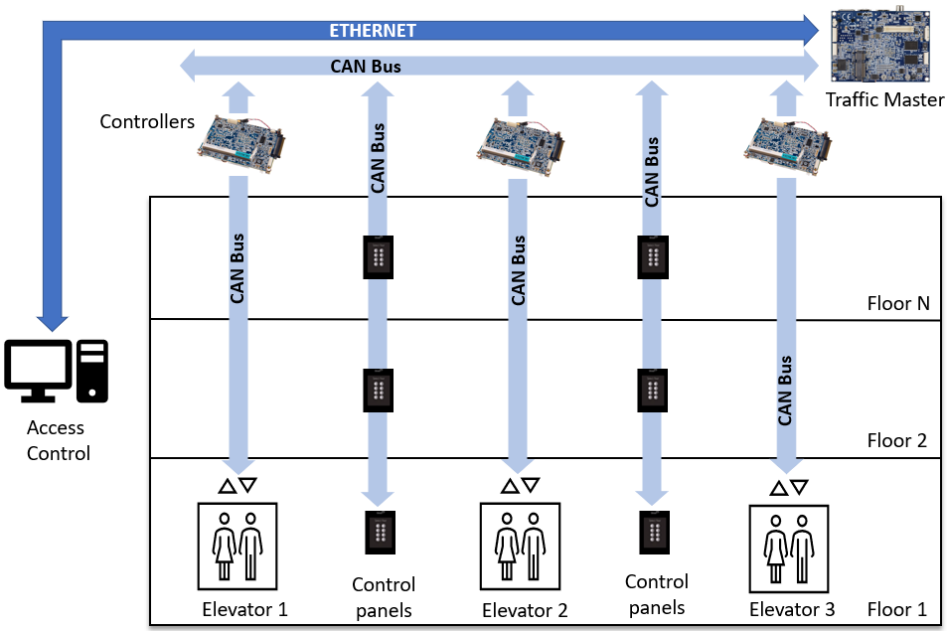


FIGURE 1 Overview of the architecture of an elevator installation from Orona [27]

The traffic algorithm should allocate the complete set of active landing calls within a limited time frame (response time), but it has limited computational resources and is executed within a task that shares these resources with other tasks that provide the functionalities mentioned above (signaling, access control, etc.). This fact, alongside with situations such as a highly demanding traffic profile or a big building with many floors and elevators, can result in the consumption of the computational resources available for the allocation of landing calls. This means that, when developing new dispatching algorithm versions, it is relatively easy to introduce non-functional bugs. Consequently, new approaches to detect these kind of bugs are necessary, and is increasingly becoming an internal priority for Orona.

3.2 | Software Testing Process at Orona

Figure 2 shows an overview of the software development process that Orona has established for dispatching algorithms [7]. The entire development process consists of a total of ten steps divided in different phases:

Phase 0: In the initial phase, the process of developing a new dispatching algorithm starts by taking the necessary pre-development actions to define the characteristics of the new algorithm. The main step during this step is the requirement elicitation.

- **Step 1 – Requirements elicitation:** The first step in the software development and maintenance process refers to elicitation of the requirements for the new dispatching algorithm version. This is carried out by the dispatcher manager and her team. Usually, requirements are defined based on (1) new legislative standards from the elevation domain that require changes and (2) new customer needs. Customer needs might be different depending on the type of building where the elevator needs to be installed. For instance, a hospital might require a special emergency button that will enable transporting critical patients through the building. After this requirement elicitation process, a validation plan is obtained as an output. This plan is later used throughout the testing process, ensuring the compliance of the newly developed system.

Phase 1 – Software-in-the-loop phase: In this phase the new algorithm is developed according to the established requirements and when it is ready for execution it is tested in SiL environment, where all the components of the plant are simulated, permitting a fast execution to find errors in an early stage.

- **Step 2 – Development:** The second step refers to the development of the dispatching algorithm. The dispatcher manager assigns a specific task to one or more developers, which usually consists in implementing some change in one of their existing dispatching algorithms. This development could be either due to (1) a change in the requirements (taken from a previous step) or (2) due to a required change that has been identified (e.g., a bug or a potential improvement). Usually, while the engineer develops the new software version, small (manual)

validations are carried out with a tool named Elevate [59]. This tool allows simulating executions of the software dispatching algorithms at the system level.

- **Step 3 – Testing (SiL):** When the engineer considers that the new software version implements the change correctly, the change needs to be checked for issued (e.g., bugs, performance regressions, etc.). At this stage, the tests are executed at the Software-in-the-Loop test phase. In this case, the SUT is the real dispatching algorithm, but the rest of the system (i.e., cabins, engines, communication buses, etc.), are simulated in a local Personal Computer (PC). Two kinds of tests are performed in this step: (1) short scenario tests and (2) full-day tests. In the short scenario tests, specific functional properties are tested in the most isolated way possible; the expected outcome of a test in this case is obtained by either implementing simple assertions or manually. In the full-day tests, scenarios that mimic a normal full-day (or sub-scenarios of it) in the life-cycle of the system of elevators are executed. The expected outcome of these tests is related to certain functional performance metric values over time windows obtained by re-executing the test with a different algorithm or with an older version (i.e., a regression oracle). The objective at this stage is to detect functional performance bugs in the code.

Phase 2 – **Hardware-in-the-loop phase:** In this phase the simulation environment is substituted by an hybrid environment where some hardware components are included so that the SUT can interact with them. This results in a more complex scenario which should be thoroughly configured and the execution of the SUT is in real-time.

- **Step 4 – HiL Infrastructure Configuration:** After testing the software at the SiL level, the next phase refers to the HiL testing. At the HiL level, the software is integrated with the real-time infrastructure, including a Real-Time Operating System (RTOS) and the real target microprocessor and communications. Unlike at the SiL phase, the physical part of the system of elevators is emulated in real-time with appropriate HiL test benches. The fourth step is the first step at this phase, and encompasses the configuration of the HiL infrastructure. For instance, the necessary controllers need to be configured depending on the type of building that the test needs to be executed in.

- **Step 5 – SUT & Test Artifacts Deployment:** After the HiL infrastructure has been configured, the SUT and the test artifacts need to be deployed. On the one hand, the SUT is deployed and integrated with the remaining hardware infrastructure. On the other hand, the test artifacts (e.g., test oracles, test inputs, etc.) are deployed.

- **Step 6 – Testing (HiL):** The sixth step refers to the execution of the test at the HiL test bench. The tests at this step are performed in real-time, and their goals are, on the one hand, to detect non-functional bugs and, on the other hand, to detect functional bugs that are related with features that were unverifiable at the SiL level due to them not being present in that environment (e.g., VIP calls). As in the third step, both short-scenario tests and full-day tests are being carried out at this stage. When all the tests have been carried out at the HiL phase, the next steps consist in the deployment and testing of the new software version in operation.

Phase 3 – Operation phase: In the last phase, the dispatching algorithm is deployed and executed in the real installation, and its performance is monitored and analyzed.

- **Step 7 – System deployment over real scenario:** Within the seventh step, the new software version is deployed in the real system. To that end, a maintainer travels to the installation and deploys the new software version. After that, a set of manual tests are carried out by the maintainer in order to ensure that everything is functioning properly.

- **Step 8 – System Operation:** After that, the system operates normally. Within this phase, there are no specific actions taken by Orona's personnel.

- **Step 9 – Monitoring and feedback:** The ninth phase relates to monitoring the behavior of the system at run-time. In addition, the maintainer might travel to the installation to check how the system behaves at specific high-traffic hours, noting issues such as queues being formed in the lobbies, etc.

- **Step 10 – Analysis of feedback data:** Feedback is sent back to production, and the dispatcher manager analyzes it to decide whether specific corrective actions need to be taken.

The test oracles play an integral role in three main stages. First, in the stage 3, where different test cases are executed at the SiL test phase. Second, in stage 6, where the tests are carried out at the HiL test phase. Lastly at stage 8, where the system is in operation, but some simple rules can be checked. A test case in the context of Orona for testing a dispatching algorithm version refers to the following fields:

- Building installation: It is the context configuration at which the SUT is being executed. It has different fields, such as the number of floors the building has, the number of elevators, which floor is served by each elevator, etc. For each elevator, there are also different fields, such as the maximum load, the energy it consumes, dynamic information (e.g., speed, acceleration and jerk), etc.

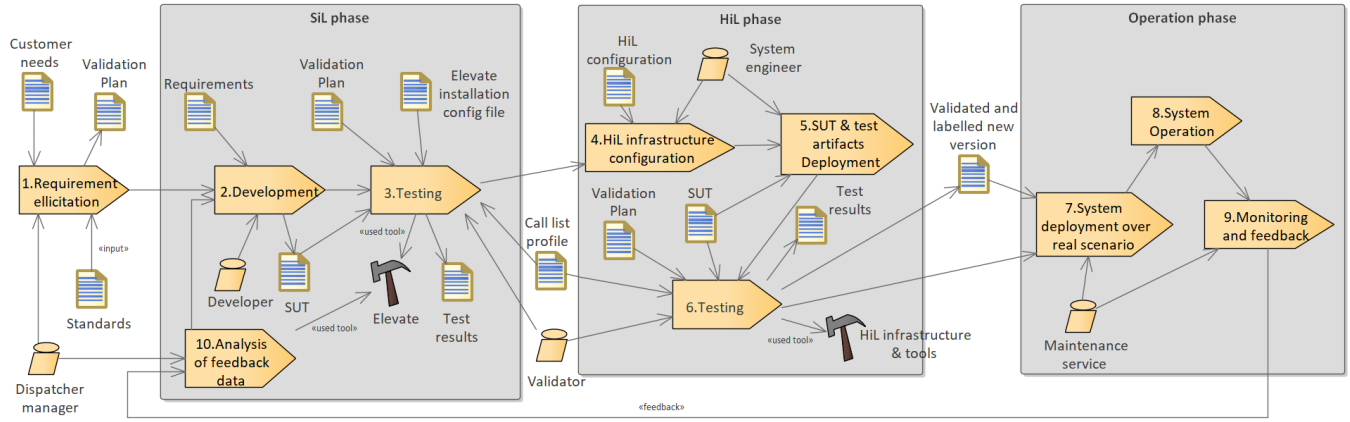


FIGURE 2 Software development process of Orona's dispatching algorithms [7]

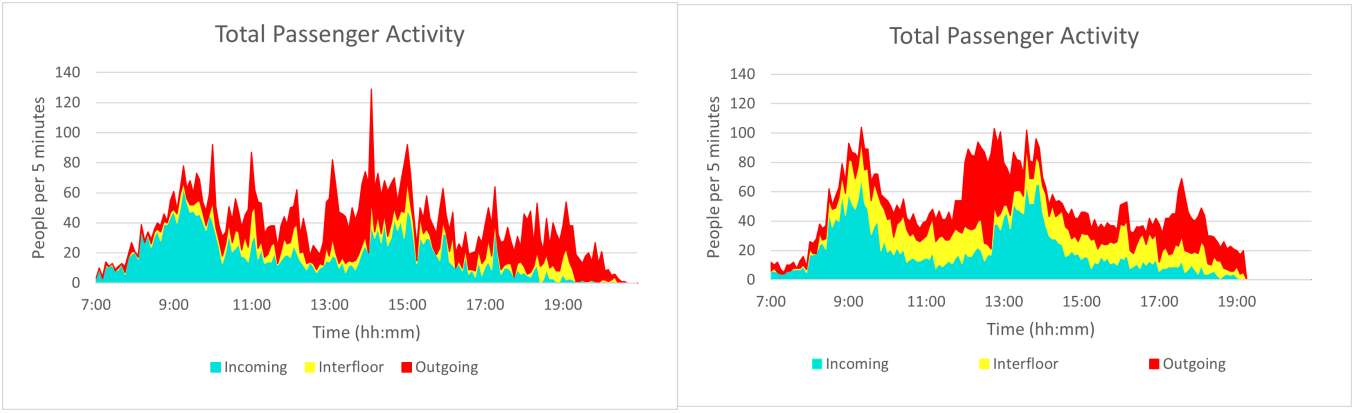
- Test input (call list profile): a test input in this context refers to a *.txt file that includes a list of passengers. For each passenger, this file includes (1) the arrival time (i.e., when the passenger requests an elevator), (2) arrival floor, (3) destination floor, (4) weight of the passenger, (5) capacity factor by mass, (6) the loading time, (7) the unloading time and (8) information related to the behavior of the passenger when not all elevators serve all floors.

- Expected output: Based on the input, what should the behaviour of the system be. This differs depending on the type of testing done. At unitary functional level, the expected output is typically related to a functional behavior of the elevator (e.g., elevator number 1 attends calls from passengers 1 and 2, elevator number 2 attends a call from passenger 3). It is important to reiterate that unitary functional level tests are not unit tests. Unit tests are carried out to test specific function, whereas unitary functional level tests are short tests done with the integration of the entire dispatching algorithm. The goal of these unitary tests is to ensure that functional requirements are met. For long full-day tests, functional behaviour is related to certain QoS metrics, which is addressed in this paper. Furthermore, at the HiL and at Operation, the non-functional metrics are also considered. Specifically, in our study, we considered the execution time of the dispatching algorithm.

  At unitary level tests, the aim is to test specific scenarios and verify that the functional requirements of the system are met. As mentioned, these unitary tests are different to unit tests, which aim at testing individual software modules. The expected output is typically related to a functional behaviour of the elevator (e.g., elevator number 1 attends calls from passengers 1 and 2, elevator number 2 attends a call from passenger 3). For long full-day tests, functional behaviour is related to certain QoS metrics, which is addressed in this paper and non-functional metrics are also considered.

This paper focuses on the oracles applied for, what is known in Orona internally as "long full-day tests". In these kinds of tests, the test inputs encompass long full-day passenger data that simulate the passenger flow in an installation. During the execution of these tests, the expected output refers to a time series of functional and non-functional performance metric values over the time. Typical functional performance metrics include the Average Waiting Time (AWT), Average Time to Destination (ATTD) or the consumed energy. Traditionally, the most used performance metric in Orona has been the AWT, because, according to certain studies, it is the most sensitive measure for a passenger to determine whether a system of elevators performs well or not [11]. Regarding non-functional performance, the most critical metric to check in dispatching algorithms is the response time, as these algorithms have a limited time frame to give a response. These test cases can be differentiated into two groups: based on *theoretical passenger data* and (2) based on *real passenger data*.

On the one hand, *theoretical passenger data* based test cases provide test inputs based on theoretical studies of passenger flows in buildings. For instance, Figure 3 depicts a graph showing the number of up calls, down calls and inter-floor calls in a time window of five minutes for a simulation of 13 hours based on the Siikonen theory for a building of offices. As can be seen, in the first couple of hours, the number of up calls increases as passengers are arriving at the office. After 5 hours, a pattern is seen with an increase in the number of down calls followed by an increase in the number of up-calls, which represents the lunch time break. At the end of the day, there is another down peak, representing the end of the working day. The advantage of theoretical traffic profiles is that they can be easily obtained by using tools like Elevate [59] just by providing data of the building population (e.g., how many people work at each floor). However, a negative aspect of the theoretical profiles is that they might not accurately represent the results obtained during real operation, where unforeseen situations arise frequently.



(a) Full day profile collected from real installation by Orona

(b) Siikonen Full day profile

FIGURE 3 Passengers activity of real installation and theoretical profiles obtained with Elevate

On the other hand, Orona uses data obtained from real installations. This helps with the validation of dispatching algorithms from several perspectives, such as the identification of certain traffic patterns not considered in theoretical traffic profiles or a higher customized validation. For instance, this could happen in a building where the canteen is on the top floor. Then, the traffic profile would significantly change as there would be a second peak in the number of up-calls followed then by an increase in the number of down-calls. An algorithm could also perform better than another depending on the traffic profile, and subsequently, the use of test inputs obtained from the field is a powerful method to validate these algorithms. The advantage of using real traffic profiles is that developers can use actual data from what it is happening in operation. This can help identify specific traffic patterns that can be found in a building. On the downside, obtaining real traffic profiles can be a non-trivial task, it requires the system to be in operation, and it is usually not possible to use this data for new installations.

In Figure 3, the differences between a real and a theoretical traffic profile for a building in Madrid can be seen. One discrepancy that can be observed is that the interfloor calls are not as frequent in the real passenger profiles when compared to the theoretical ones. In addition, at the lunch-peak, a high peak of outgoing passengers can be seen at around 14:00 in the real passenger data, which is a pattern that is not present in the theoretical data.

4 | REGRESSION LEARNING-BASED TEST ORACLE FOR ELEVATORS DISPATCHING ALGORITHMS

Figure 4 shows the overall architecture of the proposed approach. The proposed solution is divided into two main phases: (1) the training phase and (2) the testing phase.

During the training phase, a regression machine-learning algorithm is trained by using data from previous software versions. This data is the one cataloged by Orona as reference to validate other versions of the software. Subsequently, it is considered that the data used for training is from an error free version of a dispatching algorithm. The machine-learning algorithm yields a model, which is used by DARIO in the testing phase. During the testing phase, a test is executed at certain test phase (e.g., SiL or HiL). For instance, if it is at the SiL test phase, Elevate executes a test case by using test input data, information of the building installation (i.e., speed of elevators, structure of the building, etc.) and the SUT itself. When the test has finished, files containing (1) the passenger traffic data and (2) the performance measures over the simulation time (i.e., the test outcomes, which are either, those related to functional performance (e.g., AWT) or non-functional performance (e.g., execution time)) are collected. It is noteworthy that DARIO is applicable at the SiL test phase for detecting functional performance bugs, whereas at the HiL test phase it can be used for both functional and non-functional bugs. This is because, for the case of non-functional bugs, the SUT needs to be executed in the real target processor, which is not available at the SiL test phase.

The performance measures (i.e., test outcomes) may be be affected by a poor implementation of the dispatching algorithm from both a functional or non-functional perspective. From the functional point of view, when testing dispatching algorithms at system level, the main performance measure considered to label a test as PASS or FAIL by test engineers is the Average Waiting Time (AWT). This metric measures the average time each passenger needs to wait until their elevator arrives. The AWT can be a global value that measures the overall AWT for all passengers in the test input or a signal over the simulation time, indicating the AWT of the passengers in the test input for a specific time period. Elevate provides information on both. Regarding non-functional attributes, it is important for the dispatching algorithm to provide a response in a limited time frame,

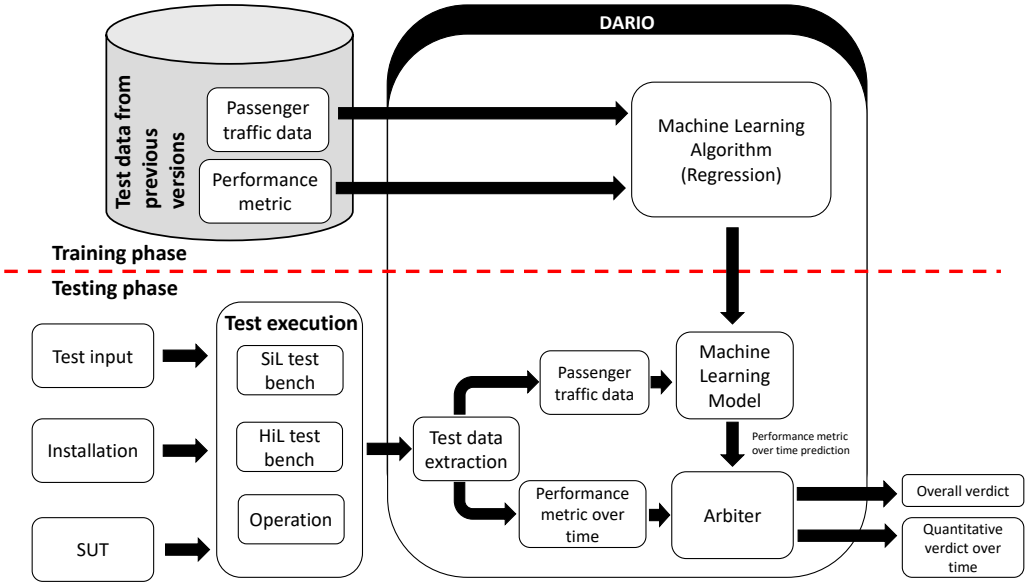


FIGURE 4 Overview of the approach at the SiL test phase

as it is executed every 500 ms, a deadline which it must comply with. Moreover, the algorithm is executed in a resource constrained environment, sharing computational resources with other tasks, so a high execution time may affect other functionalities of the system as well.

4.1 | Training phase

The first phase in our approach aims at feeding a machine-learning algorithm with labeled data to correctly train it. During the training phase, a machine-learning algorithm adapts some internal parameters based on training data so that it performs well on future unseen input data [19]. The traffic dispatching algorithm considers different information, including traveling direction, floor position, already assigned calls, etc. to assign the passengers calls, so these features are used to train the machine-learning algorithm. This data can be extracted from the passenger profiles used to execute the simulation. As stated in section 3.2, these passenger profiles consist of a .txt file with a list of the passengers calling the elevators, including the time of the call, weight of the passenger, origin and destination floor, etc.

In Orona, the Verification and Validation activities are well documented, which gives availability of data from previously tested versions of the dispatching algorithms.

Functional performance training: For training the ML algorithms, the data is categorized into different domain-specific features related to passengers traffic data (e.g., the number of uplanding and downlanding calls, time spent between two different calls in the same floor, number of calls from specific floors, etc). As for the label, the AWT metric is considered as the functional performance measure, as it is the metric that the dispatching algorithm under test used in this paper targets. All of them, for a time window of five minutes. A five minute time window was chosen based on the information provided by Elevate (the SiL infrastructure) as well as the HiL test bench. We developed a script that is able to automatically extract all the required data from a database where Orona saves all the test history. When the data was extracted, the automation script initiates the training phase by using the MATLAB machine-learning toolbox. The regression learning algorithm yields a trained regression model, which can later be used in the testing phase to predict the AWT.

It is noteworthy that typically the passenger traffic data in the historical test database is not the same as the test input in the testing phase because when changes are made in the dispatching algorithm, these changes typically include new functionalities or bug corrections. Subsequently, in the test inputs used during the testing phase the scenario testing the new functionality or a scenario that aims to trigger the fault is usually implemented. In addition, at the HiL test phase, tests also might include scenarios where the test engineer tests the Human Machine Interface (HMI) of the system. In those cases, as the testing is manual, where by the system interacts with the tester, having the exact same test case is impossible.

To train the machine-learning algorithms, we defined different features that could be of interest together with domain experts. Specifically, the data is encoded with the following input features, which are obtained every five minutes from the test outcomes:

- Number of upward calls from low level floors: It refers to the number of calls that were going up from floors 1 to 3.

- Number of upward calls from medium level floors: It refers to the number of calls that were going up from floors 4 to 6.
- Number of upward calls from up level floors: It refers to the number of calls that were going up from floors 7 to 9.
- Number of downward calls from low level floors: It refers to the number of calls that were going down from floors 2 to 4.
- Number of downward calls from medium level floors: It refers to the number of calls that were going down from floors 5 to 7.
- Number of downward calls from high level floors: It refers to the number of calls that were going down from floors 8 to 10.
- Average distance of the travel from the upwards calls: It is the average travel distance, in terms of number of floors traveled, from those calls going up. This feature could be used in meters for those cases where the distance between a floor and another is different, which is not our case.
- Average distance of the travel from the downwards calls: It is the average travel distance, in terms of number of floors traveled, from those calls going down. This feature could be used in meters for those cases where the distance between a floor and another is different, which is not our case.
- Number of upward calls (past 5 minutes): It refers to the number of calls going in up in the previous 5 minutes. We took the decision of including this because we figured out that the previous time step considered had also an impact on the current AWT.
- Number of downward calls (past 5 minutes): It refers to the number of calls going in up in the previous 5 minutes. Similar to the previous feature, we figured out that the previous time step considered had also an impact on the current AWT.

Non-functional performance training: For training the execution time models, different features are extracted from the passengers traffic data. The execution time of the dispatching algorithm is obtained by executing the algorithm in an ARM board and is set as the label. Similar to the functional performance training, together with domain experts, we defined a set of features that could affect the non-functional properties of the algorithm. Specifically, the data is encoded with the following input features, which are obtained every 500 milliseconds from the dispatching algorithm:

- Registered calls: It refers to the number of calls made by passengers but not yet assigned to an elevator.
- Assigned calls: It refers to the number of calls made by passengers and assigned to an elevator but still unattended.
- Calls in travel: It refers to the number of calls made by passengers and being attended by the assigned elevators.
- Car calls: It refers to the number of calls made by passengers from inside the elevators.
- Up calls: It is the number of calls with an ascending trajectory.
- Down calls: It is the number of calls with a descending trajectory.

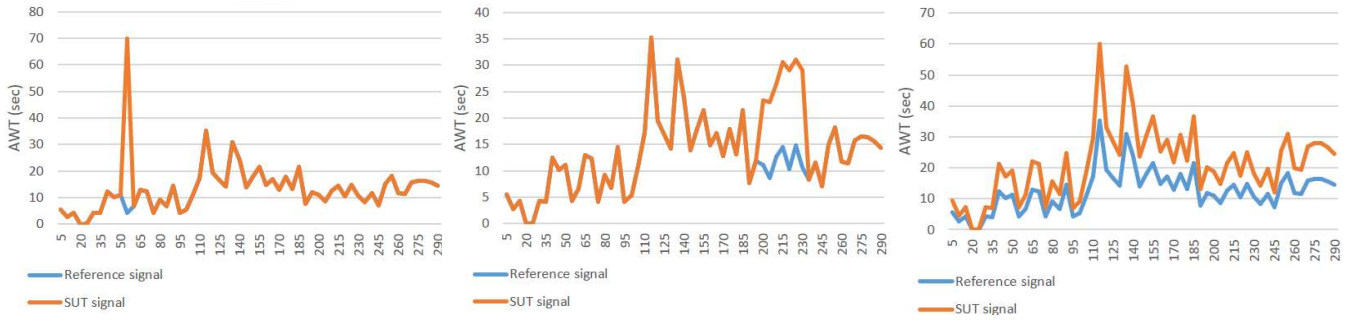
4.2 | Testing phase

When the machine-learning algorithm is trained, it yields a trained regression model, which is used in the testing phase. For the current implementation, this phase has four steps: (1) test execution, where the dispatching algorithm is tested by using simulation-based testing, (2) test data extraction, where the test results and other necessary data is extracted, (3) predictions based on the regression models, which yield the expected AWT and execution time results, and (4) the arbitration process, which compares the AWT and execution time values obtained by the regression oracle with the ones estimated by the regression model. We now explain all these steps in further detail.

Test execution: To execute a test, simulation-based testing is used. As previously mentioned, the test can be executed at two distinct levels: (1) at the Software-in-the-Loop test phase and (2) at the Hardware-in-the-Loop test phase.

The former refers to executing the test by using Elevate. An executable file of the dispatching algorithm is generated, which is considered the System Under Test (SUT). The SUT is called by Elevate at each iteration, which simulates the rest of the parts of the elevators (i.e., speed, accelerations, opening and closing of the doors, etc.). The tool also gets as input data from the installation (e.g., building type, number of elevators, characteristics of each elevator), and the test input, which involves the passenger data.

The latter refers to executing the tests by using the real hardware and other real-time infrastructure. At this test phase, the dispatching algorithm is integrated with other real-time infrastructure, such as the Linux real-time operating system, communication buses, drivers, etc. The real hardware



(a) Failure due to a high peak out of threshold bounds

(b) Failure due to a long time out of threshold bounds

(c) Failure due to a constant degradation

FIGURE 5 The three reasons why a test can be catalogued as FAIL (blue signal refers to the reference valur and orange signal refers to the value obtained by the software version under test)

that will later be used in the real elevator is used, including human-machine interface, target processors and CPUs, communication infrastructure, etc. The mechanical and electrical part of the elevators, though, are simulated within a real-time test bench. It is important to note that the execution of tests in this case is real-time. This test phase also requires substantial manual effort for setting up the test bench, with activities including the deployment of the dispatching algorithm in the target, setting up hardware infrastructure, etc. Both test phases yield several files that include results from the simulation. These files include both, overall performance metrics (e.g., the overall AWT of the simulation, total energy consumed), as well as the performance metrics over the time. This information is provided to DARIO to carry out the validation process and provide the verdicts.

In addition to both test phases, our approach could be used at operation-time. While this remains a future work, where we will need to consider aspects like the uncertainty of the environment (e.g., weight of the passengers, unexpected passenger behaviors), we believe that an advantage of DARIO is that it has the capability to be used at different test phases.

Test data extraction: After the test has been executed, DARIO extracts the necessary data from the testing files yielded by the test execution tools. At both test phases, i.e., SiL and HiL, both files are the same, which allows better re-usability of the implemented test data extraction functionality. In the current version, DARIO needs to extract (1) the passenger traffic data over the time, (2) the AWT over time and (3) the execution time of each execution of the algorithm for the tested SUT version (only at the HiL test phase). In the SiL phase, the traffic passenger data is obtained from the traffic profiles used to run the test, while in HiL and Operation phases, this data can be obtained from the CAN bus, where Controllers and Control Panels publish this information. This data is sent to the predictor, which provides an estimation of the functional or non-functional performance metric for each time step based on the trained regression models.

Prediction: The test data extraction provides the input features of every cycle of the algorithm to the trained regression model. This model, estimates the execution time for each cycle based on the training produced during the training phase. This mimics the execution of the test case in the regression oracle. The AWT and execution time of each cycle are provided to the arbiter, which is the last component in charge of providing a test verdict.

Arbiter: For developing the test arbiter, we discussed the reasons why a test can be catalogued as “FAIL” with test engineers that had the domain knowledge on testing dispatching algorithms. Thus, for the AWT arbitration, three reasons were identified, while for the the execution time arbitration, a fourth criteria has been included. All three cases are illustrated in Figure 5 Experts from Orona have set the tolerable variations in these metrics that could be considered normal in the AWT and execution of the algorithm. The arbitration process inputs the real values new release and compares it to the prediction of the regression model. When an abnormal variation is detected in any of the criteria, a ”FAIL” verdict is given.

The first reason might be that at certain point, the software version under test shows a high peak on the AWT measure (Figure 5a). This is because at a certain point, probably due to a bug, at least one passenger was unattended for a long period of time. The second reason is because the AWT measure for the software version under test exhibits a value higher than the specified threshold for a long period of time (Figure 5b). The last scenario is related to a constant degradation of the AWT value throughout all the steps of the execution (Figure 5c). For detecting execution time errors, the previous criteria are applied. High peaks on the execution time in a single step or a short period of time may be due to a bug affecting functionalities activated in specific situations (e.g., access control).

The developed arbitration algorithm aims at detecting these three scenarios. To this end, in a first step, the algorithm obtains the quantitative verdict for each simulation time step. We obtained this value by computing Equation 1.

A negative value means that the SUT version is performing worse than expected, whereas a positive value means that it is showing a better performance.

$$verdict(t) = \frac{referenceSignal(t) - SUTSignal(t)}{referenceSignal(t)} \quad (1)$$

To detect failures of the first case, a threshold is specified and the arbiter checks whether the verdict exceeds this threshold in any step of the execution, which is the invariant expressed in Equation 2. We refer to this as the single-step arbiter.

$$\forall\, t \in [t_0, t_f] : verdict(t) \geq threshold_{single\_step} \quad (2)$$

where t_0 and t_f are the first and last steps of the execution, and $threshold_{single\_step}$ is the failure threshold for the verdict value defined for the single-step arbiter.

To detect failures of the second case, a different threshold is specified. When this threshold is exceeded, the arbiter checks the following steps in order to determine the duration of the anomaly. If this duration is longer than a specified maximum duration, the test is classified as FAIL. Equation 3 defines this invariant, which we refer to as the multiple-step arbiter.

$$\forall\, t_{start} \in [t_0, t_{f-D}] : \exists\, t \in [t_{start}, t_{start+D}] : verdict(t) \geq threshold_{multiple\_step} \quad (3)$$

where $D + 1$ is the maximum number of steps for multiple-step failures, and $threshold_{multiple\_step}$ is the failure threshold for the verdict value defined for the multiple-step arbiter.

For the last case, the average value of the verdict over time signal is obtained and compared against another threshold. Equation 4 defines this invariant, which we refer to as the average arbiter.

$$\frac{\sum^{t\in[t_0,t_f]} verdict(t)}{T} \geq threshold_{average} \quad (4)$$

where T is the number of steps in the execution.

Generally, the failure threshold for the arbiters is more tolerant for anomalies of shorter duration, since shorter duration samples may be less representative of the system. Therefore, the following will usually hold (note that threshold values are negative, and smaller values imply more tolerance):

$$threshold_{single\_step} < threshold_{multiple\_step} < threshold_{average} \quad (5)$$

4.3 | Implementation

The tool was implemented in MATLAB. There are a few reasons behind this decision. The main reasons are that it provides support for a wide variety of algorithms. Furthermore, for all these algorithms, it provides a powerful C/C++ test generator, which would allow us to generate the code to execute DARIO within the real target processor in operation. The last major reason was that Elevate was integrated with BCVTB for co-simulation of the dispatching algorithm with other components of the system (e.g., the control of the elevators doors), for a higher fidelity level testing purposes [60]. BCVTB allows for the execution of MATLAB code, which permits us the execution of DARIO in this test bench with the goal of performing higher fidelity level simulation-based testing.

Additionally, although the approach is generalisable to any regression machine-learning algorithm, DARIO was implemented on top of the following ones: (1) Support Vector Machines (SVM), (2) Regression Decision Trees, (3) Ensemble, (4) Regression Gaussian Process (RGP) and (5) Stepwise Regression. The reason why these algorithms were chosen was (1) availability within the MATLAB framework and (2) appropriateness for our context in terms of prediction speed, training speed, memory usage and required tuning.

The selected algorithms have a fast prediction and training speed (unlike other algorithms such as neural networks); this is important in order to speed up the verification and validation activities. In addition, these algorithms have a small memory usage, something important when deploying the oracles in operation, where embedded processors with limited resources are used. Lastly, the selected algorithms require minimal tuning, something that is paramount to ease the transfer of the approach to practitioners.

5 | EMPIRICAL EVALUATION

In this section we empirically evaluate our approach. Our evaluation aims to answer the following Research Questions (RQs):

- **RQ1 – Functional performance:** How do machine learning algorithms perform when detecting functional performance faults?
- **RQ2 – Non-functional performance:** How do machine learning algorithms perform when detecting non-functional performance faults?

5.1 | Experimental setup

5.1.1 | Case study

We used the Orona's Conventional Group Control (CGC) algorithm. This algorithm was selected as a case study because it is the most widely used algorithm. Furthermore, there are several versions of this algorithm available in Orona, which allowed us to have access to several sets of relevant test data for performing the experiments. Lastly, its complexity is high as it is continuously evolving. It is important to note that the algorithm is deterministic, and thus, it does not involve random variations, unlike other dispatching algorithms (e.g., genetic algorithms).

In the evaluation, we used a complex building installation that Orona typically uses to validate dispatching algorithms, which is related to a real installation named the communication city, in Madrid. The building has a total of 10 floors and six elevators, each having a capacity of 1250 Kg weight and 16 passengers. Another reason for choosing this building is that Orona has relevant data obtained from the real installation while in operation. To train the machine learning algorithms of our oracles, we used available test data for testing a previous version of the CGC algorithm within the specific building. This data included ten theoretical passenger list test inputs and four real passenger list test input data.

5.1.2 | Evaluation platforms

The evaluation of our approach has been conducted in two different platforms using simulation-based testing. On the one hand, for testing the functional performance-based test oracles, the Elevate simulation environment has been used for the execution of tests with the mutants. This was carried out at the SiL test phase. On the other hand, for testing the non-functional performance-based test oracles, we used an ARM board for the execution of the tests. In practice, those test oracles would be used within the real HiL test bench, but we could not use this test bench for our evaluation because the experiments would take too long. As an alternative, we developed a Processor-in-the-Loop test bench, in which Elevate communicated with the ARM board. However, this communication was too slow, which would have not permitted us to execute all the tests with all the mutants within a reasonable time period. Therefore, the final solution was to first record the inputs to the traffic dispatching algorithm through elevate, and then launch all the tests directly in the ARM board. The traffic dispatching algorithm (i.e., SUT) runs on top of a Linux OS with a real-time patch, which allows measuring the execution time of each task regardless of the interference from other tasks during their execution. Thus, we created a task running the dispatching algorithm and measured its execution time. In order to make these measurements precise, multi-core capabilities of the board had to be disabled, as the multi-core execution was distorting them.

5.1.3 | Evaluation metrics

Mutation testing was used to seed faults through the dispatching algorithm under test. This technique has been found to be a good substitute of real faults [37]. These faults were introduced in a uniform manner throughout the sections of the source code that are relevant in the simulation environment. Two types of mutants were generated: (1) functional mutants and (2) non-functional mutants:

- **Functional mutants:** The dispatching algorithms are programmed in C. Therefore, traditional mutation operators for the C programming language were used, such as relational operator changes, arithmetic. Listing 1 and Listing 2 show an example of a code snippet and a possible functional mutant. We generated a total of 99 mutants. Although this is not a large number, it is important to note that as simulation-based testing was used, executing each mutant took a long time. This number is similar or larger to other studies where simulation-based testing was used to evaluate testing approaches [50, 6, 5]. From these 99 mutants, 18 were removed from the evaluation. The reason was that the inclusion of these mutants led the system to crashing or the simulation not lasting because passengers were not attended. In practice, both types of failures are easily detected by test engineers in Orona, test oracles not being necessary. These 18 invalid mutants were removed from the initial set, using a total of 81 mutants in our evaluation. The 81 mutants were reviewed by a domain expert to check that they were not semantically equivalent to the original program.

```
1 int max(int m, int n)
2 {
3    int max = m;
4    if(n>m)
5         max = n;
6     return max;
7 }
```

Listing 1: Code snippet of the original program

```
1 int max(int m, int n)
2 {
3    int max = m;
4    if(n<m)
5         max = n;
6     return max;
7 }
```

Listing 2: Code snippet of the functional mutant

- **Non-functional mutants:** To evaluate the adaption of DARIO to the context of non-functional performance, we used performance mutation testing [20]. In contrast to traditional mutation testing, performance mutation testing focuses on injecting performance problems, such as

an increase on the execution time or memory usage [20]. However, the functionality of the program remains the same [20], meaning that, in the context of our case study, the functional performance metric used (i.e., AWT) remains the same.[2] We systematically generated 30 performance mutants that affected the execution time based on the performance mutation operators proposed in [20]. In Listing 3 and 4 show an example of a code snippet and how a potential non-functional mutant. The function in this example looks for a given number within an array until the number is found or the end of the array is reached, but the mutant removes the second condition from the *while* loop, so forces the function to go through all the array even the number is already found. Seven of these performance mutants were finally discarded from the test executions for different reasons: two of them did not compile correctly, other two led to crashes at execution time, another one got caught in an infinite loop, and two took a too long time to execute, making its execution unfeasible. In all these cases, the manual detection of the performance bug was trivial, not requiring the use of a sophisticated oracle. Therefore, we ended up with a total of 23 performance mutants.

```
int array_contains(int num, int array[], int size)
{
    ret = 0;
    int i = 0;
    while(i < size && ret == 0)
    {
        if(array[i] == num{
            ret = 1;
        }
    }
    return ret;
}
```

Listing 3: Code snippet of the original function

```
int array_contains(int num, int array[], int size)
{
    ret = 0;
    int i = 0;
    while(i < size)
    {
        if(array[i] == num{
            ret = 1;
        }
    }
    return ret;
}
```

Listing 4: Code snippet of a non-functional mutant

Based on a similar work [30], we selected four measures to evaluate the quality of the test oracles: precision (Equation 6), recall (Equation 7), f1 (Equation 8) and accuracy (Equation 9). In addition, the specificity measure has been included (Equation 10). In our context, classifying faults well is as important as classifying correct behaviour as correct. Therefore, it is necessary to consider metrics involving both true positive rates (precision and recall) and true negative rates (accuracy and specificity).

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

$$F1 = \frac{2 \times (precision \times recall)}{precision + recall} \tag{8}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

$$Specificity = \frac{TN}{TN + FP} \tag{10}$$

$$BalancedAccuracy = \frac{Specificity + Recall}{2} \tag{11}$$

For each mutant and each passenger list, we considered the overall verdict returned by DARIO, catalogued either as “PASS” or “FAIL”. Additionally, we used the same passenger list with a regression test oracle (i.e., an original previous version under test), which is the current practice to determine if a test passes or fails by Orona. Similar to other works tackling the test oracle problem [30, 18], the verdict provided by DARIO was considered a true negative (TN), a true positive (TP), a false negative (FN) or a false positive (FP) as defined below:

- TN: Both the test oracle (i.e., DARIO) and the regression test oracle returned a ”PASS” verdict.
- TP: Both the test oracle (i.e., DARIO) and the regression test oracle returned a “FAIL” verdict.
- FN: The test oracle (i.e., DARIO) returned a “PASS” and the regression test oracle returned a “FAIL”.

[2] Notice that the algorithm used as SUT is deterministic

- FP: The test oracle (i.e., DARIO) returned a "FAIL" and the regression test oracle returned a "PASS".

The tests were executed in Elevate instead of in the HiL due to practicality (i.e., if the tests were executed using the HiL test bench, the experiments would take around 2 years).

5.1.4 | Experimental Scenarios

To answer the proposed RQs, a total of four experimental scenarios were designed, where we aimed to analyse the effect of different training and testing data on the results. Note that the names of the proposed scenarios follow the pattern [*Training data type-Testing data type*]:

- **Scenario *Theoretical-Theoretical***: We first used test cases that involve theoretical test inputs. These test inputs were automatically generated by Elevate, and are based on a study performed by Siikonen [67]. In total, 10 of these test cases were used. We employed the 10-fold cross validation to validate DARIO along with all the selected machine-learning algorithms.
- **Scenario *Theoretical-Real***: In the scenario *Theoretical-Real*, the same type of test cases were used for training, but for testing, test cases obtained from the real installation were used. This scenario would emulate (1) how the theoretical passenger data performed in order to train the algorithms during validation when using data extracted from operation and (2) how the theoretical passenger data perform for training the algorithms when the oracle is used in the real installation.
- **Scenario *Real-Real***: In this scenario, we used test cases obtained with data extracted from the real installation in operation for training. In total, four test cases were available for the building used in the evaluation. We thus employed the 4-fold cross validation to validate DARIO along with all the selected machine-learning algorithms.
- **Scenario *Real-Theoretical***: In this case, we used the same test cases as in scenario *Real-Real* to train the algorithms, but for testing we used the ten theoretical passenger-based test cases.

Table 1 summarizes the main characteristics of the test cases used in these four scenarios, including, the number of up calls, down calls, number of detected mutants by the test case, and the test case duration.

TABLE 1 Main characteristics of the used test cases during the experimental scenarios

Test case	# of Up Calls	# of Down Calls	# of Detected AWT Mutants	# of Detected Performance Mutants	Simulation time (h:min)
real1	2756	1711	18	13	8:30
real2	3086	2366	18	14	9:10
real3	3438	3117	18	10	11:45
real4	3508	3050	21	12	13:35
theoretical1	3994	3377	20	11	12:55
theoretical2	3950	3379	18	13	12:55
theoretical3	3983	3379	26	14	12:55
theoretical4	3989	3402	18	20	12:55
theoretical5	3989	3387	18	12	12:55
theoretical6	3964	3384	19	11	12:55
theoretical7	3977	3386	21	14	12:55
theoretical8	3919	3433	21	12	12:55
theoretical9	3976	3354	18	15	12:55
theoretical10	3945	3407	20	14	12:55

5.2 | Results

We now present the results obtained for the four scenarios designed to answer the RQs.

5.2.1 | RQ1 – Functional performance

Table 2 summarizes the obtained results for the four scenarios designed to answer the RQs. In the scenario *Theoretical-Theoretical* a 10-fold cross validation was performed to measure how DARIO performed when trained ad tested with theoretical traffic profiles. In terms of precision, SVM and regression tree were the techniques showing best results (0.98) followed by stepwiselm, RGP and ensemble. When considering the recall measure, ensemble performed best (0.89), followed by the rest with values around 0.80. In terms of accuracy and F-measure, SVM and regression tree performed well, unlike the remaining three machine learning algorithms, which all dropped below 0.8 in terms of both accuracy and F-measure. Both SVM and regression tree performed best in terms of the Specificity metric, both having a value of 0.99. In contrast, the rest of the algorithms showed a value of less than 0.8. With regards to balanced accuracy, regression tree showed a value of 0.91, followed by SVM, which had a value of 0.88. The remaining algorithms showed a value lower than 0.8.

TABLE 2 Summary of results for the four experimental scenarios for functional mutant detection

Scenario	Metrics	SVM	Regression Tree	Ensemble	RGP	STEPWISELM
Theoretical-Theoretical	**Precision**	0.98	0.98	0.68	0.76	0.83
	Recall	0.77	0.82	0.89	0.78	0.79
	Accuracy	0.94	0.95	0.69	0.71	0.79
	F-1	0.86	0.89	0.70	0.68	0.76
	Specificity	0.99	0.99	0.62	0.70	0.79
	Balanced Accuracy	0.88	0.91	0.75	0.74	0.79
Theoretical-Real	**Precision**	0.95	0.81	0.41	0.61	0.41
	Recall	0.66	0.81	0.86	0.86	0.88
	Accuracy	0.96	0.76	0.40	0.58	0.39
	F-1	0.76	0.73	0.46	0.60	0.45
	Specificity	0.98	0.75	0.27	0.50	0.25
	Balanced Accuracy	0.82	0.78	0.56	0.68	0.56
Real-Real	**Precision**	0.82	0.97	0.30	0.59	0.59
	Recall	0.88	0.94	1	0.88	0.83
	Accuracy	0.93	0.98	0.37	0.58	0.57
	F-1	0.85	0.95	0.45	0.62	0.59
	Specificity	0.94	0.99	0.17	0.50	0.50
	Balanced Accuracy	0.91	0.97	0.59	0.69	0.67
Real-Theoretical	**Precision**	0.25	0.25	0.25	0.30	0.25
	Recall	1.0	1.0	1.0	0.99	1.0
	Accuracy	0.25	0.25	0.25	0.37	0.28
	F-1	0.39	0.39	0.39	0.45	0.40
	Specificity	0.0	0.0	0.0	0.17	0.04
	Balanced Accuracy	0.50	0.50	0.50	0.58	0.52

As for the scenario *Theoretical-Real*, where the ten theoretical passenger data based test cases were used for training the machine learning algorithms but real-world data for testing, in terms of the average precision, in this case SVM performed best, followed by regression tree. Results for the ensemble, RGP and stepwiselm algorithms in terms of precision were overall quite low. These three algorithms, however, slightly outperformed SVM and regression tree for the recall metric. Nevertheless, in terms of accuracy and F-measure, SVM showed the best results followed by regression tree, outperforming the remaining three algorithms with significant difference. The specificity and balanced accuracy measures had a similar distribution, with the superiority of SVM followed by regression tree and far from them the rest of the algorithms.

Scenario *Real-Real* in Table 2 shows the results for the 4-fold cross validation when using the real passenger data based test cases both, for training and for testing. As can be seen, in this case, regression tree performed best in terms of precision, accuracy and F-measure. Furthermore, with a recall of 0.94, the regression tree algorithm was the second best, after ensemble. Nevertheless, the precision, accuracy and F-measure values for ensemble were all below 0.5, meaning that this algorithm had a high number of false positives. In terms of specificity and balanced accuracy, the algorithm which performed best is regression tree, with the values 0.99 and 0.97 respectively, followed by SVM, with an specificity of 0.94 and

a balanced accuracy of 0.91. Far from there, we could found the remaining three algorithms, which showed values below 0.7 for both specificity and balanced accuracy.

Scenario *Real-Theoretical* also used real passenger data based test cases to train the machine learning algorithms. Although all algorithms performed well in terms of recall, meaning that they produced none or a low number of false negatives, their results in terms of precision, accuracy and F-measure were below 0.5. The values were also quite low for both the specificity and balanced accuracy for all the algorithm. All this means that a high number of false positives were produced when following this strategy for training the algorithms but testing the dispatching algorithms with theoretical data.

When training with theoretical traffic profiles, the results for all the four measures in scenario *Theoretical-Real* were lower than those shown in scenario *Theoretical-Theoretical*. Our hypothesis behind this is related to the difference between the types of passenger traffic flow in the test cases that are based on theoretical traffic profiles and the ones obtained from the real installation. It might be possible that the theoretical passenger profiles do not explore areas which the real passenger profiles actually do. This could be the case, for instance, when the canteen or the bar is on a specific floor. The theoretical traffic profiles also make assumptions that might not hold for all offices building. For instance, the theoretical traffic profiles assume that there are lunch peaks from 12:00 to 15:00, where workers from an office building go to have lunch. Nevertheless, there might be companies and buildings where most of the workers have a continuous work day from 7:00 to 15:00.

The hypothesis is similar when training with real traffic profiles. The traffic profiles obtained from the real building installation might not exercise areas or produce situations that are considered in the theoretical traffic profiles. This makes it difficult for the regression algorithms to accurately predict the reference AWT value when trained with real traffic profiles, but tested with theoretical traffic data.

Based on the obtained results, we can answer the first RQ as follows:

SVM and regression tree algorithms were the algorithms performing best when trained with theoretical passenger profiles based test inputs. When training with real traffic profiles, the regression tree algorithm stood out over the rest. Their results were acceptable when training and testing with the same kind of traffic profile. However, the performance of the algorithms decreased when trained with theoretical traffic profiles and tested with real traffic profiles and vice-versa.

5.2.2 | RQ2 – Non-Functional performance

Table 3 summarizes the results for the non-functional performance mutant detection. In the scenario *Theoretical-Theoretical*, where a 10-fold cross validation was performed to measure the performance of DARIO when trained and tested with theoretical profiles, we saw that overall, the five algorithms showed competitive results. In terms of precision, recall, accuracy, and specificity, SVM, Regression Tree, RGP and Stepwiselm showed the most competitive results, with values ranging between 0.81 and 0.87. On the other hand, Ensemble showed the most competitive results when considering the recall metric (0.96). Nevertheless, its precision and specificity were worse than the rest of the algorithms (0.71 and 0.45 respectively), which also penalized the Accuracy and the F-1 measures (i.e., 0.75 and 0.82 respectively), although these results can still be considered acceptable. In terms of Accuracy and F-1, RGP obtained slightly better results than the rest of the algorithms, although the differences are minimal.

In the scenario *Theoretical-Real*, the machine-learning algorithms were trained with theoretical passenger profiles, but tested with real passenger profiles. Similar to what happened in scenario *Theoretical-Theoretical*, the results for SVM, Regression Tree, RGP and Stepwiselm were similar. All of them showed good performance in terms of precision and specificity (between 0.93 and 0.94 for the former and 0.95 for the latter), but their recall measure decreased (0.57 for SVM, Regression Tree and StepwiseLM, and 0.67 for RGP). Their accuracy and F-1 measure remained relatively high (i.e., between 0.75 and 0.79 for the accuracy metric and between 0.71 and 0.79 for the F-1), although the results were worse than those from scenario *Theoretical-Theoretical*. Conversely, the Ensemble algorithm obtained different results. While its precision and specificity were lower, but still competitive (0.9 for both), the rest of the metrics were higher: a recall of 0.8, an accuracy of 0.85, and an F-1 measure of 0.85.

Within the scenario *Real-Real*, where real passenger data profiles are used for both training and testing, and therefore a 4-fold cross validation used, all the algorithms showed a similar performance. In fact, all of them showed the same precision and specificity of 0.95, which is highly competitive. In terms of recall, accuracy and F-1 measure, the Ensemble showed slightly better performance than the rest, although the differences were minimal. All these values were over 0.8 for the recall, and over 0.87 for accuracy and F-1. This means that all the algorithms are highly effective within the scenario *Real-Real*.

The last scenario trained the machine-learning algorithms with real passenger data and used theoretical passenger data for testing. All the five machine-learning algorithms showed similar results. As compared to scenario *Real-Real*, the precision, accuracy, F-1 and specificity metrics dropped, while the recall measure was increased to 0.98. The differences among the algorithms were small, although RGP and Stepwiselm obtained slightly better precision, accuracy and F-1 values when compared with the other algorithms. In general, despite having worse results than in scenario *Real-Real*, the results of the algorithms were still competitive.

TABLE 3 Summary of results for the four experimental scenarios for performance mutant detection

Scenario	Metrics	SVM	Regression Tree	Ensemble	RGP	STEPWISELM
Theoretical-Theoretical	**Precision**	0.87	0.86	0.71	0.87	0.86
	Recall	0.84	0.84	0.96	0.87	0.84
	Accuracy	0.83	0.83	0.75	0.84	0.83
	F-1	0.85	0.85	0.82	0.87	0.85
	Specificity	0.82	0.81	0.45	0.81	0.81
	Balanced Accuracy	0.83	0.83	0.71	0.84	0.83
Theoretical-Real	**Precision**	0.93	0.93	0.90	0.94	0.93
	Recall	0.57	0.57	0.80	0.67	0.57
	Accuracy	0.75	0.75	0.85	0.79	0.75
	F-1	0.71	0.71	0.85	0.79	0.71
	Specificity	0.95	0.95	0.90	0.95	0.95
	Balanced Accuracy	0.76	0.76	0.85	0.81	0.76
Real-Real	**Precision**	0.95	0.95	0.95	0.95	0.95
	Recall	0.80	0.80	0.82	0.80	0.80
	Accuracy	0.87	0.87	0.88	0.87	0.87
	F-1	0.87	0.87	0.88	0.87	0.87
	Specificity	0.95	0.95	0.95	0.95	0.95
	Balanced Accuracy	0.88	0.88	0.89	0.88	0.88
Real-Theoretical	**Precision**	0.75	0.73	0.73	0.75	0.75
	Recall	0.98	0.98	0.98	0.98	0.98
	Accuracy	0.79	0.77	0.77	0.80	0.80
	F-1	0.85	0.84	0.84	0.85	0.85
	Specificity	0.52	0.48	0.48	0.53	0.53
	Balanced Accuracy	0.75	0.73	0.73	0.76	0.76

In the scenarios where theoretical data was used for training, similar to what happened in RQ1, the overall results of scenario such scenario (i.e., *Theoretical-Real*) were worse than those from scenario *Theoretical-Theoretical*. The same happened with scenarios where real data was used for training, where results from scenario *Real-Theoretical* were lower than those from scenario *Real-Real*. This means that the training of the algorithms should be done with the same type of passengers profiles that will later be used for testing. Nevertheless, the overall results were not as low as those from the previous RQ. For instance, in scenario *Theoretical-Real*, the Ensemble algorithm was competitive enough regarding all the five metrics to be usable in practice. In addition, the results for all the algorithms in scenario *Real-Theoretical* were quite competitive. Furthermore, in this case, there was not one specific algorithm that stood out over the rest, which means that any of them could be used in practice. Therefore, we can answer the second RQ as follows:

Overall, all the algorithms performed well when dealing with the problem of non-functional bug detection. Their results were better when training and testing with the same kind of traffic profile. Nevertheless, when training with one traffic profile and testing with the other, the results were still quite competitive.

5.3 | Threats to Validity

We now discuss internal and external validity threats of the performed evaluation:

Internal validity: A potential internal validity threat in our study might be related to the thresholds of the arbiter we designed, which are configurable. To reduce this threat, we discussed the parameters with domain experts to see which thresholds could be appropriate to consider a test as pass or fail. The same concern applies to the parameters of the selected machine-learning algorithms. To reduce this threat we used the default parameters from the MATLAB framework for training the algorithms. Another internal validity threats relate to the usage of a 4-fold cross validation when using the “real-real” scenarios. This was due to the fact that those cases use data obtained from the real buildings (i.e., operational

data). Unfortunately, our industrial partner did not have more operational data. However, we believe that this scenario is realistic given that the it is the number of test cases that the company has. Moreover, it is important to note that each test input has several data points of full-day traffic profiles, meaning that we have sufficient data to train the algorithms.

External validity: An external validity threat in our evaluation is related to using a single benchmark dataset based on test cases for testing dispatching algorithms. To reduce this threat, the dataset was obtained from actual test cases in Orona for testing dispatching algorithms. Furthermore, to avoid bias in the results, we did not use the same dataset for training an algorithm and for testing it, using the appropriate k-fold cross validation techniques in those scenarios where this was necessary (i.e., Exp. 1 and Exp. 3). Another external validity threat relates to the used case study. Although only a single case study was used, it is important to note that it is a real industrial case study, which provides a high degree of complexity to our evaluation. Furthermore, the used dispatching algorithm is the one which is most used in Orona's elevators.

6 | DISCUSSION

We now discuss the results obtained from the evaluation, which was carried out in an industrial context, the lessons learned from it and the open-challenges.

6.1 | Analysis of the Results

The evaluation aimed to assess whether DARIO can be an appropriate technique to be applicable in the context of functional and non-functional performance testing of CPSs. To this end, we carried out an empirical evaluation using both functional and non-functional bugs. Traditional mutation operators were used for the former, whereas we used the study by Delgado-Perez et al., [20] for the creation of non-functional performance mutants. Overall, we believe that the results are positive, indicating that our approach can be applicable in our context. For the case of functional faults, we found that the regression tree machine-learning algorithm performed better than the remaining algorithms. On the contrary, for the case of non-functional faults, there was no clear winner among the five machine-learning algorithms we assessed. In addition, we identified the importance of training the algorithms based on the same type of passenger data that was going to be evaluated by the resulting test oracles; when theoretical passenger data is going to be used, the training should be performed with theoretical passenger data too. On the other hand, if real passenger data is going to be verified, we recommend this kind of data to be used for training the algorithms instead. Based on the performed evaluation, we believe that the results of our approach are good enough to be applicable by Orona engineers when performing long full-day tests.

Similarly to other works in the field of the test oracle problem [30], we employed the precision, recall, accuracy, F-1 metrics, alongside specificity and balanced accuracy, that are commonly used by the machine-learning community. Nevertheless, a recently performed systematic literature review [25] revealed that many studies used the mutation score metric, which aims at assessing the percentage of mutants killed. In our case, the mutation score mostly depends on the type of test inputs used, rather than the test oracle itself. In fact, we used the same criterion as the regression test oracle, currently used in the context of Orona, for determining whether a mutant is killed or not by a test case. It is important to note that test oracles are prone to false positives and negatives [9, 72], and therefore, we have given priority to the right classification done by DARIO, our test oracle, rather than the amount of mutants detected by it.

6.2 | Lessons learned

Based on the results obtained from the evaluation section and the discussion with domain experts about the applicability of the approach, we extracted the following lessons:

Lesson 1 – Training data: The use of the right data for training the machine-learning algorithms play a critical role in the accuracy of DARIO. Exceptions aside, the results obtained in scenarios *Theoretical-Theoretical* and *Real-Real* are much better than the ones of obtained in scenarios *Theoretical-Real* and *Real-Theoretical*, as it is shown in the Section 5.2. Therefore, in order for the oracle to be accurate enough, the training data should be of the same type as used when testing the system. This means that when Orona uses theoretical data for testing their algorithms, they should also use theoretical test data for training DARIO. Conversely, if real data is used for testing their algorithms, they should use real passenger data for training DARIO. However, we saw that the impact of this was larger in the case of functional bugs as compared to the non-functional bugs. A potential reason for this could be that the problem of functional-performance is harder to solve due to a higher variability of the AWT metric to test inputs. Another reason could be the experimental set-up, where we used a larger amount of mutants for the case of functional performance testing than the non-functional performance testing.

Lesson 2 – Machine-learning algorithms: Our empirical evaluation suggested that in the case of functional-performance, there might be large differences among the selected algorithms. In general, the regression tree performed overall best in the first case. Conversely, for the case of non-functional performance bugs, most algorithms performed similarly, with some exceptions in some of the cases; for instance, ensemble was the worst algorithm in the scenario *Theoretical-Theoretical*, whereas the best one in the scenario *Theoretical-Real* and competitive in the scenarios *Real-Real* and *Real-Theoretical* against the remaining algorithms. For the application of the tool in industry, for the first case, the regression tree algorithm is recommended. However, further investigation is required before recommending one of the algorithms for detecting non-functional bugs; this might be based on the data used and a set of preliminary experiments done before pushing one model to production.

Lesson 3 – Importance of real passengers data: In the study, we showed that the oracles performed competitively when using real data obtained from the real installation (even if they were trained only with three real passenger profiles). This happened in both cases, functional and non-functional performance experiements (i.e. RQ1 and RQ2). In other contexts, such as web-engineering, technologies like DevOps permit using data from operation at design-time to enhance software engineering processes (e.g., testing). The good performance of the proposed approach with field test data shows the importance of researching on adapting design-operation continuum techniques (e.g., DevOps) in the context of CPSs and in domains like elevation. The simulation at design-time of situations seen only in operation is of great advantage for engineers. For instance, this permits the detection of unforeseen situations that can only be seen when the system is in operation.

Lesson 4 – Need for dealing with uncertainty in test verdicts: As it could be appreciated in our empirical evaluation, the accuracy of DARIO with certain algorithms is competitive. However, using these oracles in Orona increases the uncertainty in relation to the correctness of the verdicts. This might require re-executing tests in the regression test oracle at certain points in order to confirm verdicts. Although this might increase the cost of testing, the test results can also be used to retrain the algorithms. The increase in test execution costs is further exacerbated at the HiL and at operational levels. Potentially, DARIO could be used as preliminary oracle for testing, and use the regression oracle only at critical stages (e.g., to confirm certain test verdicts). For instance, when a test has failed, but this fail was due to the numerical verdict being close to 0 (i.e., a low severity fault), the same test could be eventually repeated with a regression test oracle. On the other hand, if the test has failed, but the numerical verdict was far from 0 (i.e., high severity fault), the test could be classified as fail without the need for confirming the verdict with a regression test oracle.

Lesson 5 – Consequences of mistaken oracle: An oracle may be mistaken when providing a test verdict. That is, the test oracle is subject to *false positives (FPs)* and *false negatives (FNs)*. On the one hand, a false positive means that a test was cataloged as *"fail"* when it should have been provided a *"pass"*. This results into, probably, the need of the developer debugging where the potential fault is located. Since there is no fault behind, this may result into time spent by an engineer in debugging a fault that does not exist. This issue could be mitigated by confirming the test verdict by using a regression test oracle. In the cases of functional bugs, this issue could happen in 2 to 3% of the cases, given that the precision of the algorithm Regression tree is between 0.97 and 0.98. Re-running tests to confirm the presence of faults in 3% of the cases is not an issue for our industrial partner. However, these values increase in the case of non-functional performance faults, having false positives in around 14% of the cases in the case of the "Theoretical-Theoretical" scenario. This suggests that more research is required to increase the robustness of such algorithms.

On the other hand, a false negative means that a test was cataloged as *"pass"* when it should have it done as a *"fail"*. If this happens at design-time (i.e., at the SiL or at the HiL phases), the bug might be missed and shipped to production. As shown in our experiments, this could happen in 18% of the cases if used a "theoretical-theoretical" test oracle. However, the regression tree algorithm showed a recall of 0.94 in the cases of "real-real" scenarios, which implies that all the many of these faults could be later detected. These faults are in the limit and could be partially solved if we consider the uncertainty that the inputs provoke in the ML models. This will be investigated in the future before fully transferring the method to our industrial partner. The current version could be used as first pass at phases 1 and 2 of Orona's testing process, while traditional oracles can be used for corner cases or to confirm test verdicts. In addition, our approach can be used for operational testing, which already outperforms the current state-of-the-practice in Orona's context, where this type of test was not possible up to now.

6.3 | Open Challenges

In this study we have assessed the performance of DARIO, a test oracle that is built on top of machine-learning algorithms to detect both, functional and non-functional bugs. DARIO is a good alternative to a traditionally employed regression test oracle, as it permits new applications, such as testing of the HMI modules or testing at different test phases (i.e., SiL, HiL and Operation). Deploying DARIO at run-time can be feasible at this point for Orona by using an architecture based on microservices [29]. However, at run-time, new challenges arise, such as the inherent uncertainty produced by the environment at which elevators operate. For instance, when executing a test at the SiL or HiL test phases, the weight of passengers is known beforehand to the oracle, which might impact the performance of the algorithms (e.g., when passengers weight less, more passengers can enter at an elevator). Conversely, this is unknown to the oracle at design-time, and therefore, uncertainty-wise techniques would need to be integrated within DARIO. Another uncertainty related issue at operation time could be the wrong use of passengers of the elevator. For instance, one could be that of calling an elevator, but later not using it. All these uncertain situations need to be classified systematically and integrated in DARIO in order this oracle to be effective at operation-time, where a regression test oracle cannot be used.

Our approach has been exclusively designed for dispatching algorithms. However, performance metrics, both functional and non-functional, can be found in many other CPSs applications, including those from the automotive [35, 74] or aerospace domains [53]. We believe that another line of research could be the application of DARIO in other CPS domains. Furthermore, DARIO has been intended to be used as test oracle for already defined test cases. However, this approach opens the gate to other testing activities. For instance, the use of the numerical verdict to determine whether a failing test is of critical or low severity could be use for activities like falsification-based test case generation. These techniques are expensive to execute [51], and therefore, a technique like DARIO could be of great benefit in such a context. This technique could also be combined with a similar approach of approximation-refinement testing [51] based test generation, in which the tests are only eventually executed in the SUT, and instead, a surrogate model of the new SUT is used.

7 | RELATED WORK

In this section we give an overview of the related work along the following dimensions: (1) use of machine-learning for test oracle generation, (2) simulation-based testing of CPSs and (3) testing of systems of elevators.

7.1 | Use of machine-learning for test oracle generation

The use of machine-learning algorithms for software testing activities has significantly increased in the last few years [23], covering many activities, including test case generation [13, 14], test case selection [70] or test prioritization [70, 10, 41]. Test oracles are one of the core components required to allow full test automation. A recent systematic literature review gathered a total of 22 relevant studies that used machine-learning for the purpose of generating test oracles [25]. Among their findings, they identified that machine-learning was used to (1) generate test verdicts [16, 31, 45, 63], (2) generate metamorphic relations [32, 33, 38, 39, 55, 78] and (3) generate the expected outputs [1, 22, 36, 54, 62, 64, 65, 69, 76, 68, 46, 79]. In this classification, our machine-learning algorithms fall into the category of the generation of expected outputs, although the entire test oracle (i.e., DARIO), provides a final test verdict. Just like all the techniques for generating expected outputs identified by Fontes and Gay [25], our machine-learning algorithms use supervised learning based on prior system executions. In addition, similar to all the studies that used a machine learning component to generate the expected outputs (with the exception of [22]), our approach used regression to determine the expected output. However, there are significant differences between our approach and those studies identified in [25]. Firstly, the type of system that our oracle tackles are CPSs, which have many idiosyncrasies, such as the expected output being a signal over time, or the long test execution time. Secondly, we investigate the performance of our approach in the context of both, functional and non-functional bugs. Thirdly, we compared a total of five machine-learning techniques, including SVM, regression tree, ensemble, RGP and Stepwiselm, unlike most of the studies identified in [25], which use a maximum of two techniques. Lastly, our SUT is a real industrial case-study from the elevator domain (i.e., the Dispatching algorithm for Orona elevators); this last point paves the way towards overcoming one of the limitations identified by Fontes and Gay [25] in the context of using machine learning for test oracle generation, i.e., the use of toy examples.

7.2 | Simulation-based testing of CPSs

Simulation-based testing has been extensively used to test CPSs [51, 52, 48, 49, 50, 13, 14, 26, 53, 66, 24, 34, 75, 6, 47, 4, 5, 6, 73]. Our approach relies on simulation-based testing for detecting performance problems (caused by either functional or non-functional bugs) of elevator systems. In such a context, there are recent studies that tackle the test oracle problem [52, 71]. Menghi et al. proposed a test oracle generation tool for Simulink models [52]. Their approach consists in a Domain Specific Language (DSL) with sufficient expressiveness to specify signal properties-based requirements. Later, a model-to-model transformation is performed in order to generate Simulink subsystems. Similar to our work, their oracles also provide a quantitative measure for the satisfaction degree of a requirement. However, our study is focused on generating a reference signal by a machine-learning algorithm trained with data from previously tested software versions, and later applying and arbitration mechanism. Although we could use their tool to generate oracles by specifying some requirements, this would be infeasible for the context of performance metrics, as inferring the relation between passenger traffic data and functional performance measures (e.g., AWT) as well as non-functional performance measures (e.g., memory consumption) is nontrivial. Furthermore, this relation is highly dependent on the building installation characteristics, and would therefore require a manual change every time the dispatching algorithm is tested in a different context; In contrast, training DARIO with already available data is straightforward and fast. Stocco et al. proposed a technique for testing self-driving cars which use deep neural networks to determine the driving parameters for the actuators of the vehicle [71]. Similar to our approach, their oracle employs simulation-based testing and determines a confidence value for the system at each step of the execution. However, the oracle they propose uses an unsupervised learning

technique based on the camera images (i.e., the input) of the self-driving car, whereas DARIO uses supervised regression learning based on the QoS measures of a system of elevators (i.e., the test output in our context).

7.3 | Testing of systems of elevators

There are some studies that focus on testing software from elevators systems. Nicolas et al., proposed an encoder based on FPGAs for simulation-based testing of elevator controllers in real-time [57]; their goal was to test the position and speed of elevators. Sagardui et al., relied on model-based testing and feature models for testing configurable software systems in charge of controlling the doors of elevators [61]; in this paper, the goal was to test refactored embedded code, and the non-refactored software acted as a golden oracle. In these cases the system was not the dispatching algorithm, but other software components of the elevators. In our previous paper, we used the technique Metamorphic Testing to test the dispatching algorithm of Orona [8], showing promising results. A problem with metamorphic testing was the difficulty to extract effective metamorphic relations. To solve this problem, we proposed a technique that combined mutation testing with genetic programming [9]. The generated metamorphic relations were as competent as those defined with the assistance from domain experts [9]. However, that metamorphic testing approach was mainly designed for short-scenario tests, whereas the technique shown in this paper is designed for long-scenario tests. Our previous work used machine-learning algorithms to detect performance bugs on software updates [28]; nevertheless, the evaluation was not as realistic as in this case, since the execution of the algorithms was performed in a local PC, whereas in this case we used the real microprocessor used by Orona in their elevators. In fact, we noticed that the results were significantly different in this case. Galarraga et al., proposed a test case generation method based on genetic algorithms to the dispatching algorithm under uncertain passenger conditions [27]. While there are many studies in the field of elevators dispatching algorithms where artificial intelligence algorithms adapted to this context are investigated (e.g., ant-colony optimization and neural networks [42], genetic algorithms [15]), to the best of our knowledge, this is the first study that proposes a method for testing them (both from the functional and non-functional perspectives).

8 | CONCLUSION AND FUTURE WORK

In this paper we have proposed DARIO, a test oracle that relies on machine-learning to automatically test elevators dispatching algorithms from the perspective of both, functional and non-functional performance. Compared with the traditionally used regression oracles, which have several disadvantages, DARIO trains machine-learning algorithms with previous test data. This training takes only a few seconds (always less than 3 seconds), whereas executing the regression test oracle takes minutes or hours at SiL (depending on the length of the test case), and hours or days at HiL (not being possible to correctly perform some tests, such as those involving HMI). In our evaluation, where an industrial dispatching algorithm from Orona was used, we tested the proposed approach with two kinds of bugs: functional and non-functional. When using functional bugs, the accuracy of the proposed test oracle when labeling tests as PASS or FAIL ranged between 0.79 and 0.87. When using non-functional bugs, the accuracy ranged between 0.8 and 0.87. The type of training data used impacted these results, although it had a greater impact in the case of the functional bugs. Overall, we believe that these results are competent enough to transfer the tool to practitioners, although further investigation is required to enhance these results.

As future research lines, we would like to explore handling the uncertainty in oracles from different perspectives. Firstly, as explained in the lessons-learned section, using DARIO increases the uncertainty related to the correctness of the verdicts. In deep learning algorithms, Kim et al., used the surprise adequacy [40]. Similar metrics adapted to the algorithms used by DARIO could be employed to measure such uncertainty. Other relevant uncertainties could be those that the CPS itself is exposed to, especially in operation. In the context of elevators at operation, in order DARIO to be effective, the uncertainty from several perspectives needs to be considered (e.g., uncertainty in passengers' behaviors). Another example of uncertainty could be related to the hardware itself, such as the noise of the sensors or delays in the communication systems.

References

[1] Aggarwal, K., Y. Singh, A. Kaur, and O. Sangwan, 2004: A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, **29**, no. 3, 1–6.

[2] Alpaydin, E., 2020: *Introduction to machine learning*. MIT press.

[3] Arrieta, A., J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, and M. Arratibel, 2021: Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms. *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, IEEE, 30–39.

[4] Arrieta, A., G. Sagardui, L. Etxeberria, and J. Zander, 2017: Automatic generation of test system instances for configurable cyber-physical systems. *Software Quality Journal*, **25**, no. 3, 1041–1083.

[5] Arrieta, A., S. Wang, A. Arruabarrena, U. Markiegi, G. Sagardui, and L. Etxeberria, 2018: Multi-objective black-box test case selection for cost-effectively testing simulation models. *Proceedings of the Genetic and Evolutionary Computation Conference*, ACM, New York, NY, USA, GECCO '18, 1411–1418.
URL http://doi.acm.org/10.1145/3205455.3205490

[6] Arrieta, A., S. Wang, U. Markiegi, A. Arruabarrena, L. Etxeberria, and G. Sagardui, 2019: Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information & Software Technology*, **114**, 137–154, doi:10.1016/j.infsof.2019.06.009.
URL https://doi.org/10.1016/j.infsof.2019.06.009

[7] Ayerdi, J., A. Garciandia, A. Arrieta, W. Afzal, E. P. Enoiu, A. Agirre, G. Sagardui, M. Arratibel, and O. Sellin, 2020: Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. *28th IEEE International Conference on Requirements Engineering, RE 2020, Zurich, Switzerland, 2020*.

[8] Ayerdi, J., S. Segura, A. Arrieta, G. S. Arratibel, and M. Arratibel, 2020: Qos-aware metamorphic testing: An elevation case study. *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 104–114.

[9] Ayerdi, J., V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, and M. Arratibel, 2021: Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1264–1274.

[10] Bagherzadeh, M., N. Kahani, and L. Briand, 2021: Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering*.

[11] Barney, G. and L. Al-Sharif, 2015: *Elevator traffic handbook: theory and practice*. Routledge.

[12] Barr, E. T., M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, 2014: The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, **41**, no. 5, 507–525.

[13] Ben Abdessalem, R., S. Nejati, L. C. Briand, and T. Stifter, 2016: Testing advanced driver assistance systems using multi-objective search and neural networks. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 63–74.

[14] — 2018: Testing vision-based control systems using learnable evolutionary algorithms. *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18.

[15] Bolat, B., P. Cortés, E. Yalçin, and M. Alişverişçi, 2010: Optimal car dispatching for elevator groups using genetic algorithms. *Intelligent Automation & Soft Computing*, **16**, no. 1, 89–99.

[16] Braga, R., P. S. Neto, R. Rabêlo, J. Santiago, and M. Souza, 2018: A machine learning approach to generate test oracles. *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, 142–151.

[17] Brunnert, A., A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert, 2015: Performance-oriented DevOps: A Research Agenda.
URL http://arxiv.org/abs/1508.04752

[18] Chan, W. K., J. C. Ho, and T. Tse, 2010: Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, **20**, no. 2, 89–120.

[19] Deisenroth, M. P., A. A. Faisal, and C. S. Ong, 2020: *Mathematics for machine learning*. Cambridge University Press.

[20] Delgado-Pérez, P., A. B. Sánchez, S. Segura, and I. Medina-Bulo, 2020: Performance mutation testing. *Software Testing Verification and Reliability*.

[21] Derler, P., E. A. Lee, and A. Sangiovanni-Vincentelli, 2011: Modeling cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, **100**, no. 1, 13 – 28.

[22] Ding, J. and D. Zhang, 2016: A machine learning approach for developing test oracles for testing scientific software. *SEKE*, 390–395.

[23] Durelli, V. H., R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, and M. P. Guimaraes, 2019: Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, **68**, no. 3, 1189–1212.

[24] Etxeberria, L., F. Larrinaga, U. Markiegi, A. Arrieta, and G. Sagardui, 2017: Enabling co-simulation of smart energy control systems for buildings and districts. *IEEE 22nd Conference on Emerging Technologies and Factory Automation (ETFA2017)*, 1–4.

[25] Fontes, A. and G. Gay, 2021: Using machine learning to generate test oracles: a systematic literature review. *Proceedings of the 1st International Workshop on Test Oracles*, 1–10.

[26] Gaaloul, K., C. Menghi, S. Nejati, L. C. Briand, and D. Wolfe, 2020: Mining assumptions for software components using machine learning. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 159–171.

[27] Galarraga, J., A. Arrieta, S. Ali, G. Sagardui, and M. Arratibel, 2021: Genetic algorithm-based testing of industrialelevators under passenger uncertainty. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE.

[28] Gartziandia, A., A. Arrieta, A. Agirre, G. Sagardui, and M. Arratibel, 2021: Using regression learners to predict performance problems on software updates: a case study on elevators dispatching algorithms. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 135–144.

[29] Gartziandia, A., J. Ayerdi, A. Arrieta, S. Ali, T. Yue, A. Agirre, G. Sagardui, and M. Arratibel, 2021: Microservices for continuous deployment, monitoring and validation in cyber-physical systems: an industrial case study for elevators systems. *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, IEEE, 46–53.

[30] Genç, A. E., H. Sözer, M. F. Kıraç, and B. Aktemur, 2019: Advisor: An adjustable framework for test oracle automation of visual output systems. *IEEE Transactions on Reliability*.

[31] Gholami, F., N. Attar, H. Haghighi, M. V. Asl, M. Valueian, and S. Mohamadyari, 2018: A classifier-based test oracle for embedded software. *2018 Real-Time and Embedded Systems and Technologies (RTEST)*, IEEE, 104–111.

[32] Hardin, B. and U. Kanewala, 2018: Using semi-supervised learning for predicting metamorphic relations. *2018 IEEE/ACM 3rd International Workshop on Metamorphic Testing (MET)*, IEEE, 14–17.

[33] Hiremath, D. J., M. Claus, W. Hasselbring, and W. Rath, 2020: Automated identification of metamorphic test scenarios for an ocean-modeling application. *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, IEEE, 62–63.

[34] Hou, Y., Y. Zhao, A. Wagh, L. Zhang, C. Qiao, K. F. Hulme, C. Wu, A. W. Sadek, and X. Liu, 2015: Simulation-based testing and evaluation tools for transportation cyber–physical systems. *IEEE Transactions on Vehicular Technology*, **65**, no. 3, 1098–1108.

[35] Jahangirova, G., A. Stocco, and P. Tonella, 2021: Quality metrics and oracles for autonomous vehicles testing. *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 194–204.

[36] Jin, H., Y. Wang, N.-W. Chen, Z.-J. Gou, and S. Wang, 2008: Artificial neural network for automatic test oracles generation. *2008 International Conference on Computer Science and Software Engineering*, IEEE, volume 2, 727–730.

[37] Just, R., D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, 2014: Are mutants a valid substitute for real faults in software testing? *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 654–665.

[38] Kanewala, U. and J. M. Bieman, 2013: Using machine learning techniques to detect metamorphic relations for programs without test oracles. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 1–10.

[39] Kanewala, U., J. M. Bieman, and A. Ben-Hur, 2016: Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software testing, verification and reliability*, **26**, no. 3, 245–269.

[40] Kim, J., R. Feldt, and S. Yoo, 2019: Guiding deep learning system testing using surprise adequacy. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 1039–1049.

[41] Lachmann, R., S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, 2016: System-level test case prioritization using machine learning. *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 361–368.

[42] Liu, J. and Y. Liu, 2007: Ant colony algorithm and fuzzy neural network-based intelligent dispatching algorithm of an elevator group control system. *2007 IEEE International Conference on Control and Automation*, IEEE, 2306–2310.

[43] Liu, Y., C. Xu, and S. C. Cheung, 2014: Characterizing and detecting performance bugs for smartphone applications. *Proceedings - International Conference on Software Engineering*, no. 1, 1013–1024.

[44] Mahdavinejad, M. S., M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, 2018: Machine learning for internet of things data analysis: a survey. *Digital Communications and Networks*, **4**, 161–175, doi:10.1016/j.dcan.2017.10.002.
URL https://doi.org/10.1016/j.dcan.2017.10.002

[45] Makondo, W., R. Nallanthighal, I. Mapanga, and P. Kadebu, 2016: Exploratory test oracle using multi-layer perceptron neural network. *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, IEEE, 1166–1171.

[46] Mao, Y., F. Boqin, Z. Li, and L. Yao, 2006: Neural networks based automated test oracle for software testing. *International Conference on Neural Information Processing*, Springer, 498–507.

[47] Markiegi, U., A. Arrieta, G. Sagardui, and L. Etxeberria, 2017: Search-based product line fault detection allocating test cases iteratively. *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, ACM, New York, NY, USA, SPLC '17, 123–132.
URL http://doi.acm.org/10.1145/3106195.3106210

[48] Matinnejad, R., S. Nejati, L. Briand, T. Bruckmann, and C. Poull, 2015: Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, **57**, 705 – 722.

[49] Matinnejad, R., S. Nejati, L. C. Briand, and T. Bruckmann, 2015: Effective test suites for mixed discrete-continuous stateflow controllers. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 84–95.

[50] — 2019: Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Trans. Software Eng.*, **45**, no. 9, 919–944, doi:10.1109/TSE.2018.2811489.
URL https://doi.org/10.1109/TSE.2018.2811489

[51] Menghi, C., S. Nejati, L. Briand, and Y. I. Parache, 2020: Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, IEEE, 372–384.

[52] Menghi, C., S. Nejati, K. Gaaloul, and L. C. Briand, 2019: Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 27–38.
URL https://doi.org/10.1145/3338906.3338920

[53] Menghi, C., E. Viganò, D. Bianculli, and L. C. Briand, 2021: Trace-checking cps properties: Bridging the cyber-physical gap. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 847–859.

[54] Monsefi, A. K., B. Zakeri, S. Samsam, and M. Khashehchi, 2019: Performing software test oracle based on deep neural network with fuzzy inference system. *International Congress on High-Performance Computing and Big Data Analysis*, Springer, 406–417.

[55] Nair, A., K. Meinke, and S. Eldh, 2019: Leveraging mutants for automatic prediction of metamorphic relations using machine learning. *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 1–6.

[56] Nejati, S., K. Gaaloul, C. Menghi, L. C. Briand, S. Foster, and D. Wolfe, 2019: Evaluating model testing and model checking for finding requirements violations in simulink models. *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 1015–1025.
URL https://doi.org/10.1145/3338906.3340444

[57] Nicolas, C. F., I. Ayestaran, I. Martinez, and P. Franco, 2016: Model-based development of an fpga encoder simulator for real-time testing of elevator controllers. *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, IEEE, 53–60.

[58] Olivo, O., I. Dillig, and C. Lin, 2015: Static detection of asymptotic performance bugs in collection traversals. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, **2015-June**, 369–378.

[59] Peters, D. K. and D. L. Parnas, 2002: Requirements-based monitors for real-time systems. *IEEE Trans. Software Eng.*, **28**, no. 2, 146–158, doi:10.1109/32.988496.

[60] Sagardui, G., J. Agirre, U. Markiegi, A. Arrieta, C. F. Nicolás, and J. M. Martín, 2017: Multiplex: A co-simulation architecture for elevators validation. *Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM), 2017 IEEE International Workshop of*, IEEE, 1–6.

[61] Sagardui, G., L. Etxeberria, J. A. Agirre, A. Arrieta, C. F. Nicolas, and J. M. Martin, 2017: A configurable validation environment for refactored embedded software: An application to the vertical transport domain. *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 16–19.

[62] Sangwan, O. P., P. K. Bhatia, and Y. Singh, 2011: Radial basis function neural network based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, **36**, no. 5, 1–5.

[63] Shahamiri, S. R., W. M. N. W. Kadir, and S. bin Ibrahim, 2010: An automated oracle approach to test decision-making structures. *2010 3rd International Conference on Computer Science and Information Technology*, IEEE, volume 5, 30–34.

[64] Shahamiri, S. R., W. M. N. W. Kadir, S. Ibrahim, and S. Z. M. Hashim, 2011: An automated framework for software test oracle. *Information and Software Technology*, **53**, no. 7, 774–788.

[65] Shahamiri, S. R., W. M. Wan-Kadir, S. Ibrahim, and S. Z. M. Hashim, 2012: Artificial neural networks as multi-networks automated test oracle. *Automated Software Engineering*, **19**, no. 3, 303–334.

[66] Shin, S. Y., K. Chaouch, S. Nejati, M. Sabetzadeh, L. C. Briand, and F. Zimmer, 2021: Uncertainty-aware specification and analysis for hardware-in-the-loop testing of cyber-physical systems. *Journal of Systems and Software*, **171**, 110813.

[67] Siikonen, M.-L., 2000: On traffic planning methodology. *Elevator technology*, **10**, 267–274.

[68] Singhal, A. and A. Bansal, 2014: Generation of test oracles using neural network and decision tree model. *2014 5th International Conference-Confluence The Next Generation Information Technology Summit (Confluence)*, IEEE, 313–318.

[69] Singhal, A., A. Bansal, and A. Kumar, 2016: An approach to design test oracle for aspect oriented software systems using soft computing approach. *International Journal of System Assurance Engineering and Management*, **7**, no. 1, 1–5.

[70] Spieker, H., A. Gotlieb, D. Marijan, and M. Mossige, 2017: Reinforcement learning for automatic test case prioritization and selection in continuous integration. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 12–22.

[71] Stocco, A., M. Weiss, M. Calzana, and P. Tonella, 2020: Misbehaviour prediction for autonomous driving systems. *Proceedings of 42nd International Conference on Software Engineering*, ACM, ICSE '20, 12 pages.

[72] Terragni, V., G. Jahangirova, P. Tonella, and M. Pezzè, 2020: Evolutionary improvement of assertion oracles. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1178–1189.

[73] Turlea, A., 2018: Search based model in the loop testing for cyber physical systems. *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, IEEE, 22–28.

[74] Valle, P., 2021: Metamorphic testing of autonomous vehicles: a case study on simulink. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, 105–107.

[75] van der Meer, A. A., P. Palensky, K. Heussen, D. M. Bondy, O. Gehrke, C. Steinbrinki, M. Blanki, S. Lehnhoff, E. Widl, C. Moyo, et al., 2017: Cyber-physical energy systems modeling, test specification, and co-simulation based testing. *2017 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, IEEE, 1–9.

[76] Vanmali, M., M. Last, and A. Kandel, 2002: Using a neural network in the software testing process. *International Journal of Intelligent Systems*, **17**, no. 1, 45–62.

[77] Vokolos, F. I. and E. J. Weyuker, 1998: Performance testing of software systems. *Proceedings of the 1st International Workshop on Software and Performance*, Association for Computing Machinery, New York, NY, USA, WOSP '98, 80–87.
URL https://doi.org/10.1145/287318.287337

[78] Zhang, P., X. Zhou, P. Pelliccione, and H. Leung, 2017: Rbf-mlmr: A multi-label metamorphic relation prediction approach using rbf neural network. *IEEE access*, **5**, 21791–21805.

[79] Zhang, R., Y.-w. Wang, and M.-z. Zhang, 2019: Automatic test oracle based on probabilistic neural networks. *Recent Developments in Intelligent Computing, Communication and Devices*, Springer, 437–445.

How to cite this article: Gartziandia A., Arrieta A., Ayerdi J., Illarramendi M., Sagardui G., Agirre A., and Arratibel M. (2022), Using Machine Learning to Build Test Oracles: an Industrial Case Study on Elevators Dispatching Algorithms, *Journal of Software: Evolution and Process*, *2022*.