Universidade do Minho
Escola de Engenharia

Ivo Alexandre Pereira Baixo

Vision System Hardware: Simulation and Test Automation

november 2023

Universidade do Minho
Escola de Engenharia

Ivo Alexandre Pereira Baixo

Vision System Hardware: Simulation and Test Automation

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
Luís Paulo Peixoto dos Santos (U.Minho)
Diogo Dinis (Smartex)

november 2023

Acknowledgements

A special recognition is reserved for Diogo Dinis, my esteemed company supervisor. Diogo's expert guidance, support, and insightful perspectives have decisively steered this project in the right direction. Learning and growing under his mentorship has been enriching both at the professional and personal levels. Equally deserving of appreciation is Ricardo Brandão, whose collaborative efforts and unique view points have undeniably elevated the quality of this work.

Appreciation is also extended to my university coordinator Luís Santos for his continuous counselling and dedicated assistance throughout this journey, fostering academic excellence that has been a cornerstone of the quality exhibited in this dissertation.

The unwavering love, unyielding belief, and ceaseless encouragement of my parents have served as the very foundation of this work. Their enduring support has been absolutely indispensable in achieving this significant milestone.

I hold a profound appreciation for the unfaltering support of my friends, who stood by me with constant encouragement. Their motivational words and unshakable faith in my capabilities provided the strength to overcome the most challenging moments.

I am very grateful to the Embedded Systems team as well as Smartex, for granting me the opportunity to undertake this research within their dynamic environment and to create something capable of helping make the world a better place.

This work would not have been remotely possible without the invaluable contributions of each individual mentioned above. Your collective efforts have brought this dissertation to fruition.

With genuine and heartfelt appreciation I thank you all,

Ivo

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Assinado por: **Ivo Alexandre Pereira Baixo**
Num. de Identificação: 15796881
Data: 2023.11.24 20:19:25+00'00'

University of Minho, Braga, november 2023

Ivo Alexandre Pereira Baixo

Abstract

This dissertation addresses the challenge of efficiently developing and testing software in hardware-dependent systems, with a specific focus on simulating hardware used in the *Vision Software* project of Smartex. Smartex is a company that applies AI solutions to the textile industry, particularly to the quality inspection cycle in fabric production.

The primary motivation behind this research is to overcome delays in the software development life cycle caused by the software having physical hardware dependencies, whose unavailability can injure the software development life cycle. To achieve this, this dissertation aims to develop software capable of simulating the *Vision Software's* hardware components and provide an application programming interface (API) for seamless interaction with the simulation software.

The research objectives encompass several key aspects. Firstly, the development of simulation software to faithfully simulate the hardware components of the Vision System. Secondly, the creation of an automated test pipeline that leverages the simulation software to enhance Smartex's Quality Assurance processes for the *Vision Software*. Additionally, the reliability and effectiveness of the hardware simulation solution will be thoroughly tested. Lastly, the impact of this work on Smartex's software life cycle, specifically for the Embedded Systems and Quality Assurance teams, will be carefully measured and evaluated.

This dissertation offers a comprehensive approach to hardware simulation through the use of software as well as automated software testing, effectively addressing a critical challenge in modern software development. The achieved solution significantly improved the overall software life cycle for the Embedded Systems and Quality Assurance teams at Smartex, enabling the testing and development of the *Vision Software* without the need for physical hardware dependencies.

Keywords Hardware-dependent systems, Software simulation, Software testing, Software development, API, AI, Industry 4.0

Resumo

Esta dissertação foca-se em abordar o desafio de desenvolver e testar com eficácia *software* em sistemas que dependem de *hardware*, com um foco específico na simulação do *hardware* usado no projeto *Vision Software* da Smartex. A Smartex é uma empresa que aplica soluções de inteligência artificial à indústria têxtil, em particular ao ciclo de inspeção de qualidade na produção de tecidos.

A principal motivação por detrás desta investigação é ultrapassar atrasos no ciclo de vida de desenvolvimento de *software* causados pelo facto de o *software* ter dependências físicas de *hardware*, cuja indisponibilidade pode prejudicar o ciclo de desenvolvimento de *software*. Para tal, esta dissertação tem como objetivo desenvolver *software* capaz de simular os componentes de hardware do *Vision Software* e fornecer uma interface de programação de aplicações (API) para uma interação perfeita com o *software* de simulação.

Os objectivos da investigação abrangem vários aspectos fundamentais. Em primeiro lugar, o desenvolvimento de *software* de simulação para simular fielmente os componentes de hardware do sistema de visão. Em segundo lugar, a criação de um *pipeline* de testes automatizados que aproveita o *software* de simulação para melhorar os processos de garantia de qualidade da Smartex para o *Vision Software*. Além disso, a fiabilidade e a eficácia da solução de simulação de *hardware* serão testadas exaustivamente. Por último, o impacto deste trabalho no ciclo de vida do software da Smartex, especificamente para as equipas de Sistemas Embebidos e *Quality Assurance*, será cuidadosamente medido e avaliado.

Esta tese oferece uma abordagem abrangente à simulação de *hardware* através da utilização de *software*, bem como de testes automatizados de *software*, abordando efetivamente um desafio crítico no desenvolvimento moderno de *software*. A solução alcançada melhorou significativamente o ciclo de vida global do *software* para as equipas de Sistemas Embebidos e *Quality Assurance* da Smartex, permitindo o teste e o desenvolvimento do *Vision Software* sem a necessidade de ter disponíveis as suas dependências físicas de *hardware*.

Palavras-chave Sistemas dependentes de hardware, Simulação por *software*, Teste de *software*, Desenvolvimento de *software*, API, IA

Contents

List of Figures

List of Tables

Acronyms

ABS	Agent-Based Simulation.
AI	Artificial Intelligence.
API	Application Programming Interface.
CI/CD	Continuous Integration/Continuous Delivery.
CKM	Circular Knitting Machine.
DES	Discrete-Event-Simulation.
DTS	Discrete-Time-Simulation.
HAL	Hardware Abstraction Layer.
HIL	Hardware-in-the-loop.
QA	Quality Assurance.
SIL	Software-in-the-loop.

Part I

Introductory material

Chapter 1

Introduction

This chapter introduces the present dissertation's theme and establishes its motivation, beginning by contextualizing it in the current scene of software development.

It is then explained the motivation behind this dissertation, being clarified the objectives it aims to accomplish as well as its possible impacts, along with the methodology under which it will be developed and implemented.

It concludes with a summary of the topics addressed in each chapter of this dissertation, explaining its overall structure.

1.1 Context

In the fast evolving world of software development, it has become clear that being able to efficiently and automatically test software is as important as the software development itself.

However, when software needs to be run on specific hardware, it can become troublesome to test it because it is necessary to have that hardware available, which inevitably delays the whole software development cycle.

A possible solution to tackle this problem consists on simulating the required hardware using software, ending the need for having the physical hardware available. Simulating the hardware, one can even test software for hardware versions that are not even available yet.

With all this in mind, this dissertation addresses:

- The simulation of the hardware used in the Vision System of the company Smartex;
- The development and implementation of tests that measure the simulation impact and benefits;
- The development and implementation of automated tests that take advantage of the simulation created.

Smartex is a company that applies **AI** solutions to textile quality inspection. The textile industry, representing a 3 trillion dollar industry, discards around 10% of its production due to defects, which besides being a significant loss of capital, it is an enormous waste of textile material, chemicals, and water.

Smartex solutions, to this date (August of 2023), have saved up around 807,000 kilograms of textile fabric as well as 90 million liters of water.

This dissertation project was developed within the scope of the Master's degree in Informatics Engineering at the University of Minho.

1.2 Motivation

In Smartex's software development life cycle, in order for a specific version of software to be sent to the Production stage (i.e. to be used by the clients), it first needs to be vigorously tested in order to minimize the chances of something going wrong in a 'real life' scenario.

Since Smartex's product consists of both hardware and software, in order to exhaustively test it, it is necessary to have the physical hardware available to be able to conduct tests. This turns out to be a bottleneck on Smartex's software development life cycle since it makes the testing phase especially long and troublesome, since it's not easy to have the hardware available for testing and, especially, it takes a lot of time to go through all the test scenarios, which are continuously increasing with each software development life cycle.

With the primary objective of improving the testing phase of Smartex's product in mind, this dissertation aims to reach a solution that, by putting an end to the hardware dependencies (by simulating it with software), lowers the required time to test a specific software version. This dissertation also aims to develop and implement automated tests that make use of the solution achieved in order to optimize Smartex's software development life cycle.

1.3 Objectives

The main goal of this project consists of simulating the hardware used by a Smartex's module named *Vision Software* with the use of a software layer and developing automated tests that make full use of this layer. In order to accomplish this, the following steps are required:

- Development of software capable of simulating all of the hardware on which the *Vision Software* module depends. This would make it possible to execute this module in a containerized environment without having the need for any physical hardware besides the computer that runs the software;

- Development of an **API** to interact with the simulation software, allowing for choosing which components should be simulated, what simulation types should be used, and other configurations;
- Testing of both the simulation solution and API developed, in order to access their proper working;
- Development of automated software tests that take advantage of the simulation solution produced and help Smartex's **Quality Assurance** (**QA**) team, by automating and speeding up some of their tasks;
- Measure the impact created by this dissertation work on the software development life cycle of Smartex, more specifically how it influenced the Embedded Systems and Quality Assurance teams.

1.4 Methodology

The work methodology adopted in this dissertation consists of the following steps:

- Bibliographical research on hardware simulation, test automation, and software life cycles;
- Requirements elicitation and analysis for the solution to be developed;
- Technology research in order to achieve the better implementation of the solution possible;
- Implementation of the solution, according to the requirements gathered;
- Testing and performance analysis of the solution;
- Weekly meetings with Smartex co-supervisor;
- Bi-weekly meetings with the supervisor.

1.5 Structure of the Document

In this introductory chapter, it was given a brief introduction about what this dissertation consists of, explained the context in which it is inserted, the motivations that gave birth to this topic, the goals it aims to accomplish as well as the methodology that will be used to achieve them.

The second chapter consists of the State of the Art, where research work is done about what has and is being done around topics that concern this dissertation, namely simulation by software and software testing.

In the third chapter, the dissertation delves into the problem that gave rise to the research, providing a comprehensive technical explanation and highlighting the primary challenges involved. It establishes the starting point for the subsequent work conducted in this study, clarifying the goals of this dissertation.

In the fourth chapter, a comprehensive analysis is conducted to examine the initial state of the Vision Software. This chapter focuses on proposing an approach that addresses and satisfies all the objectives outlined in the third chapter. This approach is backed by an extensive theoretical framework, incorporating the revisions made in the second chapter. By leveraging this theoretical foundation, the chapter aims to provide a detailed and well-informed plan of action to meet the stated objectives and drive the subsequent development and improvement of the Vision Software.

The fifth chapter serves as a comprehensive account of all the developments undertaken throughout this dissertation. With meticulous detail, it thoroughly analyzes the solutions devised, as well as the challenges encountered during the its research and implementation. Each decision made is scrutinized, elucidating the reasoning behind its selection and providing a comparative analysis to alternative paths. The chapter offers a profound understanding of the work conducted, presenting a comprehensive overview of the undertaken actions and achievements within the scope of this dissertation.

In the sixth chapter, the results accomplished in this dissertation are summarized, providing a comprehensive overview of the key findings and contributions and accessing the impact of this dissertation in Smartex's *Vision Software*. Additionally, future possible directions for further development and improvement are presented, outlining potential avenues for future work. This chapter serves as a culmination of this dissertation, encapsulating the significance of the work conducted and providing a roadmap for how to further improve it.

Chapter 2

State of the Art

This chapter serves as a comprehensive repository of knowledge, consolidating the culmination of extensive research conducted on the existing body of work pertaining to the theme addressed in this dissertation. Through a thorough examination of available literature, scholarly articles, relevant studies, and other pertinent sources, this chapter offers a comprehensive synthesis of the insights and findings amassed thus far, providing a solid foundation for the subsequent analysis and discussion within this dissertation.

2.1 Simulation

A simulation is ”an imitation of the dynamics of a real-world process or system over time” [Leonelli, 2021] with the objective of finding flaws and problems inherent in that system in order to rectify them [Bandyopadhyay and Bhattacharya, 2014].

But why should a simulation approach be chosen instead of direct experimentation of the real system? The main advantages that come with simulation are, according to [Leonelli, 2021]:

- It does not disrupt the real system;
- Simulation time can be adjusted so that the simulated system is faster or slower than the real system;
- It can be replicated multiple times.

Before diving into more detail on simulation, it is important to make clear a few key concepts.

A system, as stated in [Law, 2015] and in accordance with [Sasser, 1970] consists of a collection of entities that act and interact together with the aim of accomplishing a logical goal. The state of a system consists of the set of variables necessary to describe that system at a given time, being relative to the goals of a study.

A model, on the other hand, is a simplified version of a system that constructs a conceptual framework that enables an analysis of the system itself [Shishvan and Benndorf, 2017].

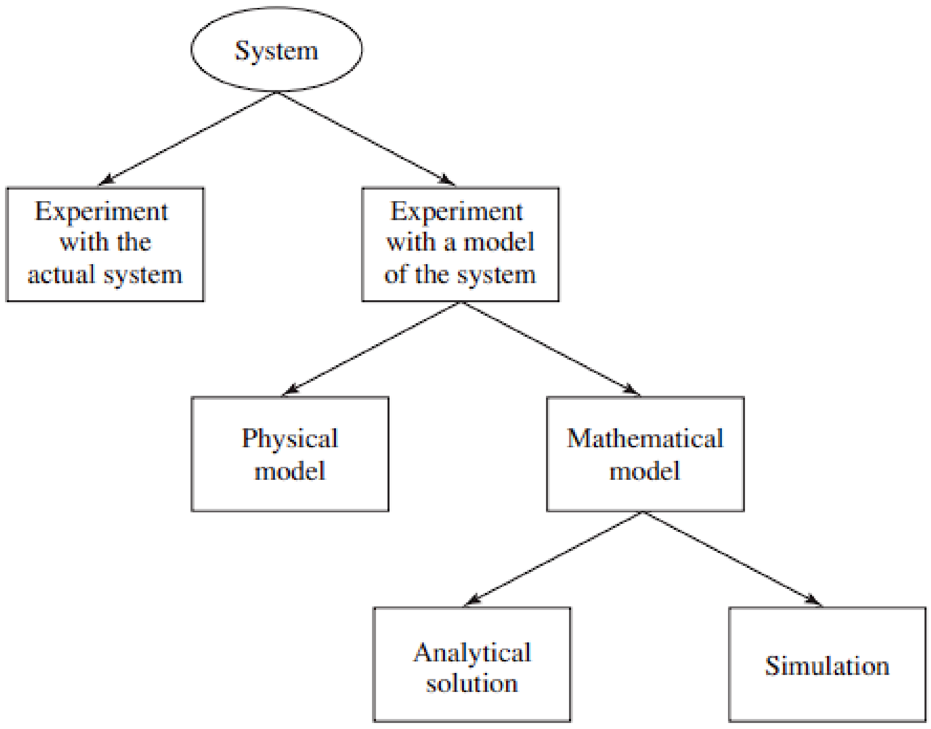
System
Experiment with the actual system
Experiment with a model of the system
Physical model
Mathematical model
Analytical solution
Simulation

Figure 1: Ways to study a system (Source: [Law, 2015])

As shown in Figure 1, in order to study a system there are only two alternatives: experimenting with the system itself (which is not often feasible according to [Law, 2015]) or creating a model of that system. That model can either be a physical one (which is "a miniature of the actual system" [Bandyopadhyay and Bhattacharya, 2014]) or a mathematical model which can in turn be divided into analytical models and simulation models. An analytical model is one that is solved through an analytical expression.

In reality, to study a system the options present in Figure 1 should be considered in a top-down manner. Only when all other options are not possible or viable (due to their sheer complexity), a simulation model should be considered, as explained in [Law, 2015].

2.1.1 Simulation Model

In order to analyze and comprehend the behavior of a system, simulation models are constructed. In accordance to [Borshchev and Filippov, 2004], a "simulation model may be considered as a set of rules (e.g. equations, flowcharts, state machines, cellular automata) that define how the system being modeled will change in the future, given its present state". Therefore, the act of simulation consists of executing a simulation model, passing through different states over time.

The development of a simulation model usually begins with a detailed analysis of the most important system elements, as well as their relationships, and for that it is useful to examine from a wide range of sources (e.g. system logs, questionnaires, interviews) as stated in [Anderson and Fu, 2016].

2.1.2 Types of simulation models

To be able to model a real system with precision, it is important to be able to decide where the real system fits in regard to some different dimensions.

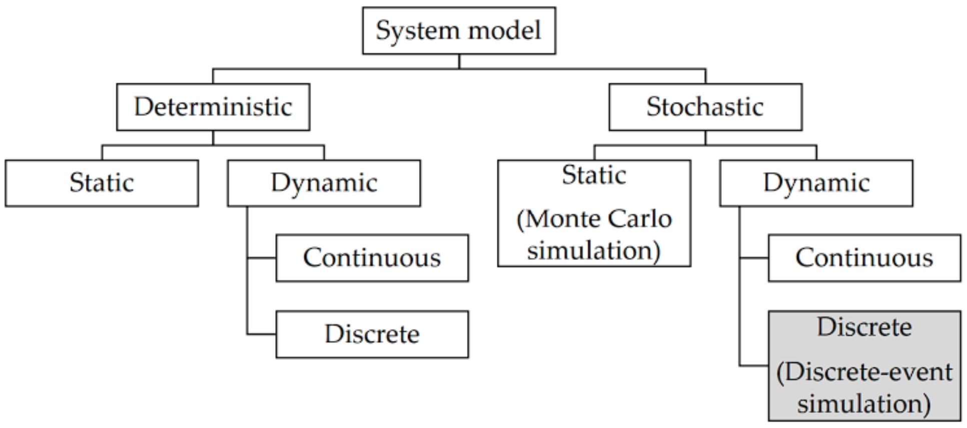

Figure 2: Types of simulation models (Source: [Shishvan and Benndorf, 2017])

The main types of simulation modeling fit on the diagram shown in Figure 2. To create an adequate simulation model it is then necessary to make a few choices according to some characteristics of the real system.

Deterministic versus Stochastic

A deterministic model has an entirely predictable behavior, so for a given input the model will always return an unique output [Shishvan and Benndorf, 2017]. On the other hand, a model is considered stochastic if it returns a probability distribution over the output values [Bandyopadhyay and Bhattacharya, 2014].

Stochastic models are often favored over classical models for various compelling reasons. Firstly, the choice of stochastic models can be motivated when the input values themselves exhibit stochastic characteristics, thereby necessitating a model that can appropriately capture and respond to this inherent randomness. Secondly, the dynamics of the system under consideration may inherently demand a stochastic simulation. This can arise either because the system itself is inherently stochastic, such as in the case of physical quantum processes, or due to the inherent complexity or partial knowledge of the system's dynamics, which may make it more suitable to approximate these dynamics using a stochastic process

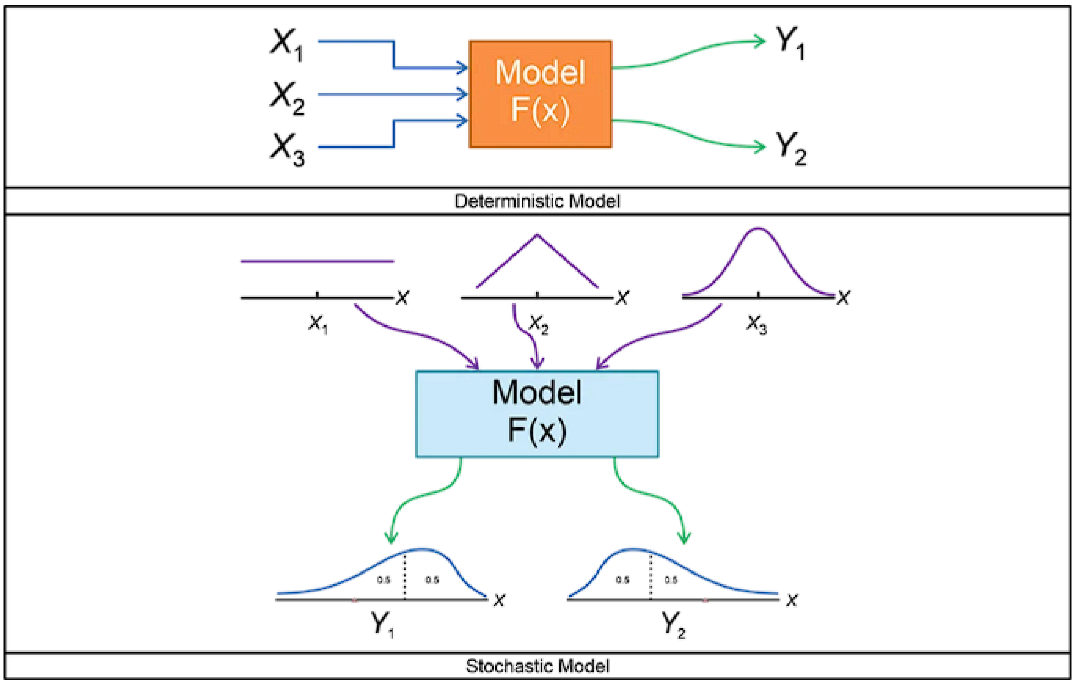

Figure 3: Deterministic and Stochastic Models (Source: [Platon and Constantinescu, 2014])

Static versus Dynamic

In accordance with [Law, 2015], a "static simulation model is a representation of a system at a particular time or one that may be used to represent a system in which time simply plays no role". Static models put together a snapshot of the system in a given instant. Furthermore, a dynamic model "represents a system that changes with the advent of time" as stated by [Bandyopadhyay and Bhattacharya, 2014], since, in dynamic models, variables are denoted as functions of time.

When the system to be modeled is known to change with the passage of time, and time is a factor to be taken into consideration then a dynamic model would be the correct approach for modeling. In situations where the system doesn't change over time or it is only important to model the system at a particular point in time, a static model should be used.

Continuous versus Discrete

As illustrated in Figure 2, dynamic models can be further categorized into continuous or discrete models, depending on how in time the model variables change.

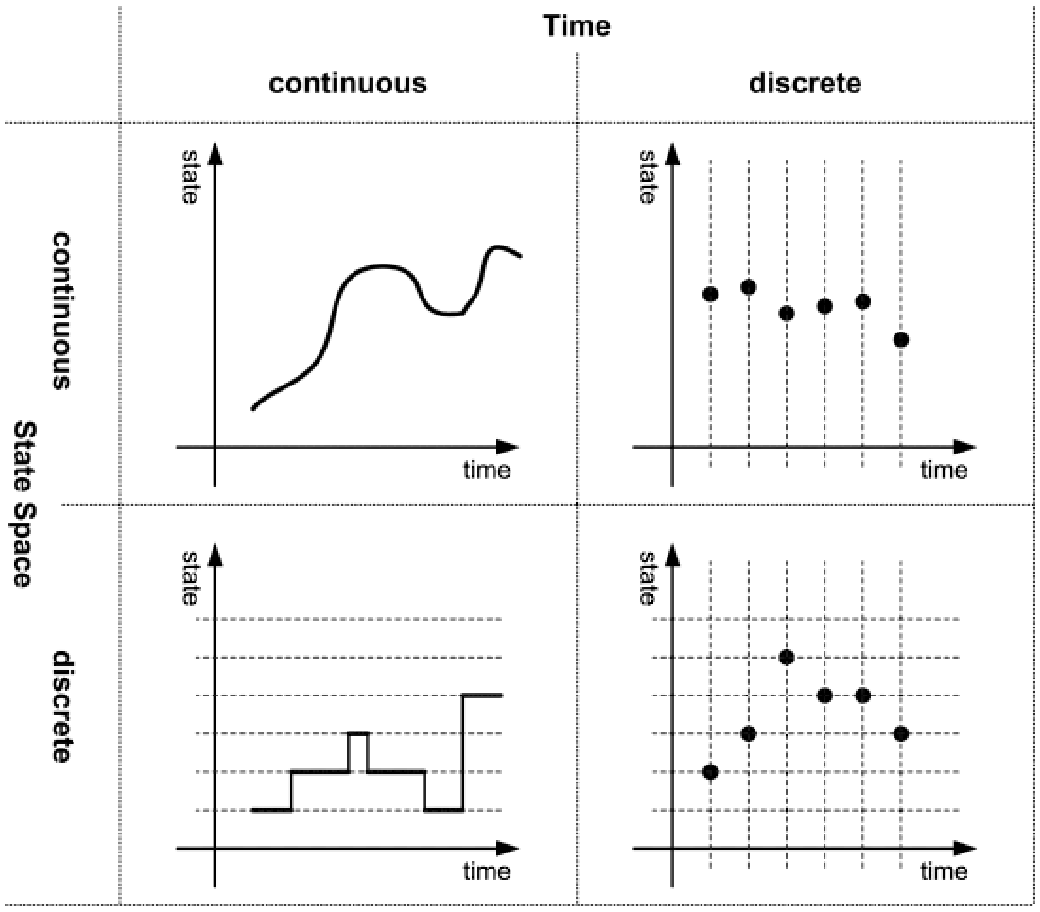

Figure 4: Continuous vs Discrete - time and state (Source: [Holzer et al., 2010])

As claimed by "Continuous simulation concerns the modeling over time of a system by a representation in which the state variables change continuously with respect to time" [Law, 2015], often utilizing differential equations in order to " to quantify the changes in a system continuously over time" [Shishvan and Benndorf, 2017]. In contrast, in discrete models, the variables of interest only change at discrete set of points in time [Leonelli, 2021, Bandyopadhyay and Bhattacharya, 2014].

In regards to how to choose between these two models, it falls down to whether the variables of interest are changing continuously or are only altered in discrete times by discrete steps, as concluded in [Özgün and Barlas, 2009].

It is important to point out that, as portrayed in Figure 4, that variables in simulation models can sometimes be considered discrete or continuous according to their state space instead of time.

Discrete simulations can further be classified into one of the following categories:

- **Discrete-Time-Simulation (DTS)** - In this type of simulation, time is divided into equal length intervals and it progresses by a fixed amount that can be adjusted.

- **Discrete-Event-Simulation (DES)** - In this type of simulation, time only progresses as events occur.

Although at first glance DTS and DES might seem equivalent, they don't converge to the same results every time, as shown in [Buss and Al Rowaei, 2010]. The following Figure 5 shows a visual representation of these two approaches in more detail.

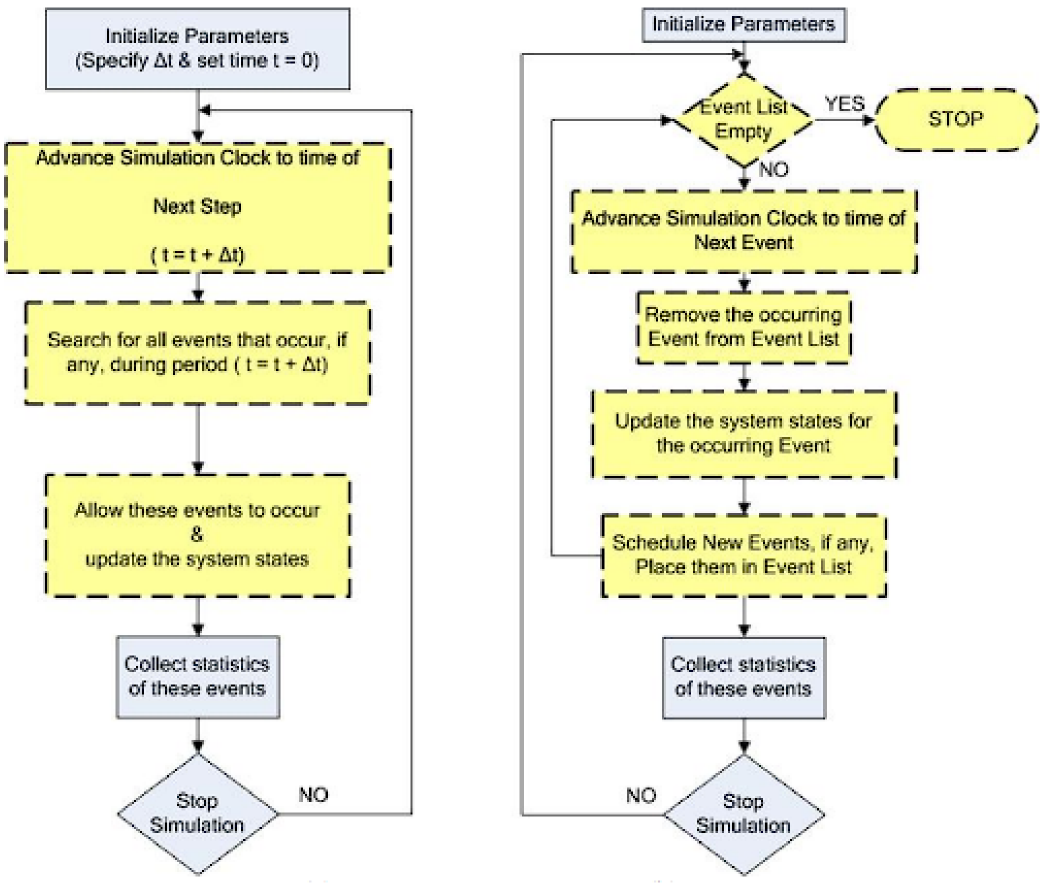
Initialize Parameters (Specify Δt & set time t = 0)
Advance Simulation Clock to time of Next Step (t = t + Δt)
Search for all events that occur, if any, during period (t = t + Δt)
Allow these events to occur & update the system states
Collect statistics of these events
Stop Simulation
NO
Initialize Parameters
Event List Empty
YES
STOP
NO
Advance Simulation Clock to time of Next Event
Remove the occurring Event from Event List
Update the system states for the occurring Event
Schedule New Events, if any, Place them in Event List
Collect statistics of these events
NO
Stop Simulation

Figure 5: Overview of DTS and DES(Source: [Al Rowaei et al., 2011])

It should be noted that in DTS it is necessary to define *a priori* the interval length, and according to [Tang et al., 2020], larger interval lengths lead to faster simulations with lower accuracy.

Agent-Based

Although not depicted in Figure 2, **Agent-Based Simulation (ABS)** is an important type of simulation that could be fitted in the Discrete-event simulation category, and it consists of a bottom-up analysis of the real system, where the focus is understanding and describing the behavior of individual agents and how they interact with each other, as stated in [Law, 2015]. Agent-based models are composed of the agents (entities capable of making independent decisions [Macal and North, 2008]), and a rule set that defines how agents interact with one another and an environment.

This type of simulation model thrives when it is important to study how interactions between agents can influence the global state of the system since "each agent can have differing preferences as when to act" [Robertson, 2005]. as well as when agents can "adapt and change their behaviors" [Macal and North, 2014] in accordance to what is happening around them.

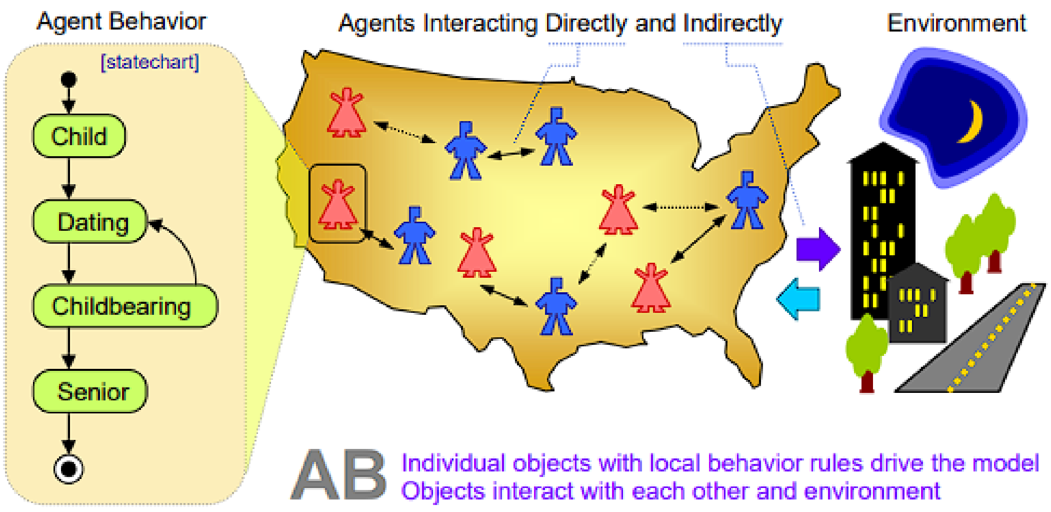
Agent Behavior
[statechart]
Child
Dating
Childbearing
Senior
Agents Interacting Directly and Indirectly
Environment
AB
Individual objects with local behavior rules drive the model
Objects interact with each other and environment

Figure 6: Generic Agent-Based Model Architecture (Source: [Maxim Garifullin])

In Figure 6, it can be seen a representation of an Agent Based simulation, where its shown that each entity can be in one of multiple different states and is capable of making decisions according to its own state and the interactions it has with other entities. The environment, as depicted, can also have an impact on the behaviors and decisions of system's agents.

2.1.3 Steps of a simulation study

In order to build a robust and precise simulation model, it is important to follow the succeeding steps, depicted in Figure 7. It should be noted that, as referred in [Law, 2015], that a simulation study is not a "simple sequential step". As a matter of fact, it actually consists of an iterative process where it is usually necessary to go back and improve on the work done at previous steps (as shown in Figure 7).

- **1 - Problem Formulation:** Define the "purpose for building the model" [Shishvan and Benndorf, 2017].

- **2 - Setting of Objectives:** Identify the system or process that is to be simulated, which involves understanding the system's components and how they interact. The goal of the simulation is prop-

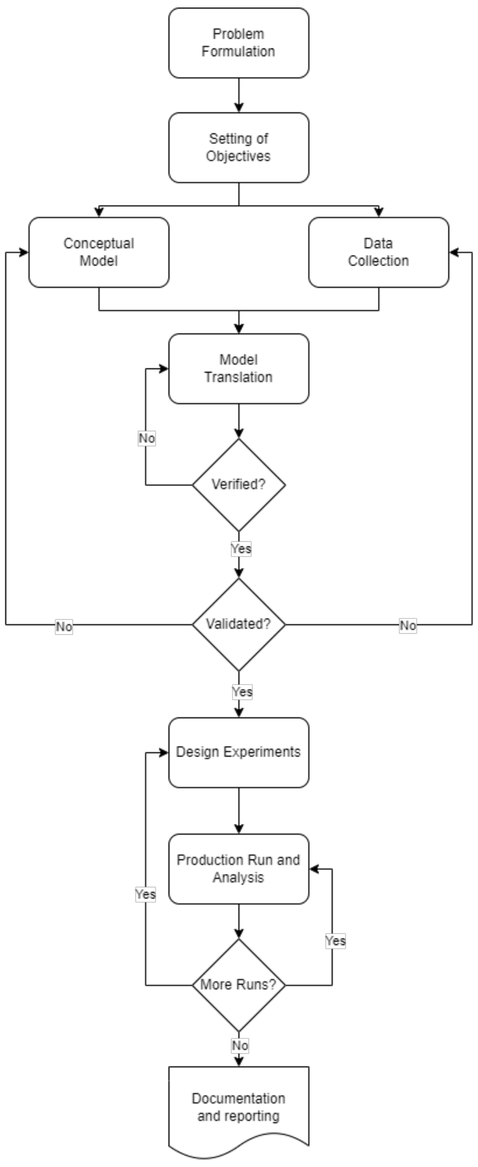

Figure 7: Steps in a simulation study (Based on: [Ban, 1998] and [Law, 2015])

erly defined as well as the performance measures, as stated in [Bandyopadhyay and Bhattacharya, 2014].

- **3 - Data Collection:** Collect information about the system structure, data to specify model parameters and inputs' probability distributions, when possible. It should also be gathered data on the performance of the existing system for validation purposes, as is pointed out in [Law, 2015].

- **4 - Conceptual Model:** Develop a model that represents the behavior of the system as accurately as possible, starting "with a 'simple' model" and embellishing it as needed [Law, 2015].

- **5 - Model Translation:** The model created in step 3 is "coded into a computer recognizable form" [Ban, 1998], either by using a programming language, which offers more greater program control, or by using a simulation software, which results in lower project costs, as shown in [Law, 2015]. In the end, a operational model of the system is obtained.

- **6 - Verified?:** The outputs of the operational model created in step 5 are compared against those that would have been produced by the real system, as stated in [Bandyopadhyay and Bhattacharya, 2014]. In other words, it involves ensuring that the simulation model is being used correctly and that the results it produces are reliable. This can be done by checking the simulation model's code for errors, running it multiple times to ensure consistency, and comparing the results to known results from other sources. If the operational model doesn't match the expectations step 5 should be redone, as depicted in Figure 7.

- **7 - Validated?:** "Validation is the determination that the conceptual model is an accurate representation of the real system" [Ban, 1998]. To validate a simulation model, one must compare the results of the simulation to real-world data and check the model's assumptions and inputs to ensure they are reasonable and realistic. As can be seen in Figure 7, failing the validation check means that the process should fall back to steps 3 and 4.

- **8 - Design Experiments:** "For each scenario that is to be simulated, decisions need to be made concerning the length of the simulation run, the number of runs (also called replications), and the manner of initialization", according to [Ban, 1998].

- **9 - Production Run and Analysis:** The simulation model is ran and its results are taken into analysis, by applying various statistical tools, as stated in [Bandyopadhyay and Bhattacharya, 2014]. The results are used to estimate measures of performance for the simulated scenarios.

- **10 - More Runs?:** Based on the result of the analysis done in step 9, it is decided whether additional runs are needed (in this case step 9 is repeated) or/and if any new scenario needs to be simulated (which would mean going back to step 8), as explained in [Ban, 1998]. If no more runs are necessary, the process can continue.

- **11 - Documentation and Reporting:** Document all assumptions and study's' results in order for the simulation to be usable in the current and future projects, as mentioned in [Law, 2015].

It should be referred that steps 3 and 4 (Data Collection and Conceptual Model) should be done concurrently, since both steps can influence one another and will probably suffer multiple iterations, during the simulation study process.

2.1.4 Hardware simulation vs emulation

The terms simulation and emulation are often used interchangeably, existing a few different definitions of these concepts. However, for the sake of this dissertation coherence, it will be followed the definition made in [McGregor, 2002] which states that simulation and emulation are fundamentally different in the way they are used and how they work, ”Although a simulation model and an emulation model may look to all intents and purposes the same, and may be built largely with the same building blocks, there are significant differences in usage and operation” [McGregor, 2002]. Emulation models attempt to mimic all of the hardware features of a device, allowing for an environment that is closer to the device portrayed when compared to simulation models, being used in a ”much more precisely defined way” [McGregor, 2002]. On the other hand, simulation models only mimic the behavior of a device without worrying about how it works on the inside.

Emulation shines when there is a need to test how software interacts with underlying hardware since it reflects more accurately how the hardware behaves. Simulation is better suited for experimenting and testing software since it has the ”purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by a criterion or set of criteria) for the operation of the system”, according to [Shannon and Johannes, 1976].

To decide which approach to follow its important to understand how these two types of models behave in terms of testing speed and testing repeatability. In regards to testing speed, as a simulation model ”maintains its own clock” [McGregor, 2002], the speed of the tests does not need to be the real-time execution speed, allowing for faster executions. However, emulation is not as flexible when it comes to experimentation and testing, only being able to be executed in real time [McGregor, 2002]. When it comes to repeatability, emulation approaches, due to their precise nature, tend to replicate with more accuracy the system that is being modeled, since they typically involve a complex set of parameters and configurations, which can make replicating emulation approaches more challenging than simulation ones.

2.1.5 Simulation Fidelity

Simulations can be quantified on their fidelity level, which represents the level of realism portrayed in the simulation itself (Figure 8). Thus, a higher level of fidelity assures a higher correctness degree, witch leads to a more accurate system. However, increasing the fidelity degree translates into higher implementation costs for the simulation. Therefore, it is important to choose the fidelity level based on the specific needs of the project in hands (as stated in [Co., 2018], [Carey and Rossler, 2022]).

According to [Co., 2018], there are four fidelity categories: Tiebacks, Low Fidelity, Medium Fidelity and High Fidelity depending on the degree of detail and abstraction provided by the simulation.

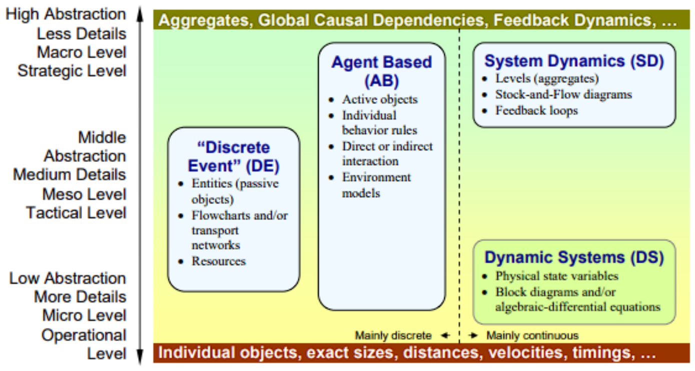

Figure 8: Different Levels of Abstraction in Simulation (Source: [Borshchev and Filippov, 2004])

Simulating hardware can be done in multiple ways, ranging from a low fidelity simulation where all the inner workings of each hardware component (processor, memory system, peripherals, etc) are simulated up to a high fidelity simulation where only the hardware components are seen as a black box, being simulated only the inputs and outputs of those components.

2.1.6 Hardware simulation techniques

Hardware-in-the-Loop

Hardware-in-the-loop (**HIL**) simulation, shown in Figure 9, is a technique used to verify the behavior of a hardware design and to test the interaction between hardware and software components. In other words, hardware-in-the-loop can be described as a "methodology for hybrid system synthesis where selected hardware and software components are immersed in a closed-loop virtual simulation environment" Liu et al. [2012]. According to [Bacic, 2005], the "basic idea of in hardware-in-the-loop simulation is that of including a part of the real hardware in the simulation loop during system development", finding a "middle ground between physical prototyping and virtual simulation, combining the advantages of both approaches" ([Liu et al., 2012]) since the hardware and software components are run in conjunction, being tested together.

One advantage of HIL simulation is that it allows for a more realistic simulation of a hardware system, as it involves the use of real hardware components which can be particularly useful for applications that require a high degree of accuracy.

However, HIL has some downsides, such as requiring of the physical hardware components to simulate the system, making it unfeasible for contexts where the hardware to be tested is not available. Also, adjusting the simulation time to be different than real-time may prove to be impossible if a hardware component can not be accelerated/decelerated, as explained in [Bullock et al., 2004].

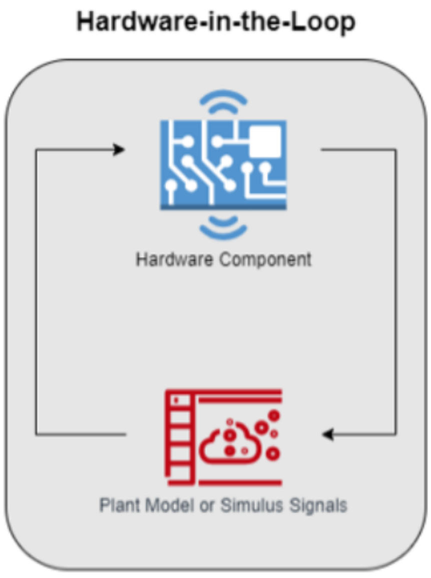

Figure 9: Hardware-in-the-loop structure (Based on: [Chang] and [dsp])

Software-in-the-Loop

Software-in-the-loop (**SIL**) simulation, shown in Figure 10 is a technique that involves simulating the behavior of the hardware components of system using software, helping to "run, test and validate production-ready source code" [Clavijo-Rodriguez et al., 2021].

Although not as "robust" as Hardware-in-the-loop simulation ([Chen et al., 2008]), SIL facilitates the test of the interaction between hardware and software components, and if the simulation software can "simulate complex dynamics in various practical systems" then it can be a "reliable alternative solution to HIL simulation for validation and/or rapid prototyping purposes", as stated in [Chen et al., 2008].

One of the key advantages of SIL simulation is its flexible and low cost nature ([Demers et al., 2007]), that allows for the behavior test and verification of hardware components without having the need for physical hardware. SIL makes it possible to test in an isolated manner the software components of a system which proves to be very useful when developing systems with software dependent on scarcely available hardware components.

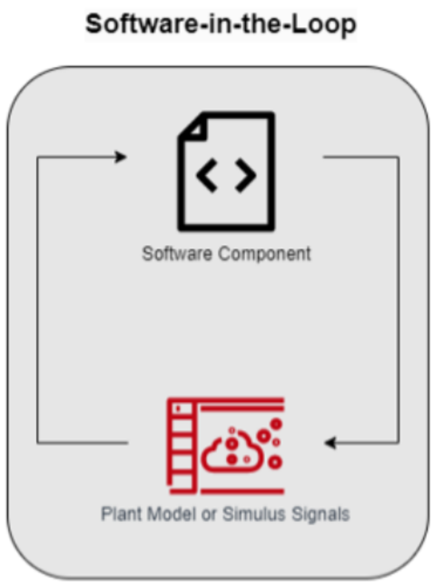

Figure 10: Software-in-the-loop structure (Based on: [Chang] and [dsp])

2.1.7 Hardware Abstraction Layer

Hardware Abstraction Layer (**HAL**), shown in Figure 11, is a software layer that provides a uniform interface for interacting with hardware components by abstracting the hardware access, defined as "all the software that is directly dependent on the underlying hardware" by [Yoo and Jerraya, 2005].

The purpose of a HAL is to isolate the higher-level software from the details of the hardware ([Jörg et al., 2014]), making it easier to develop, maintain and test software that is dependent on hardware components.

Hardware-abstraction-layers often provide a set of functions for reading and writing data to hardware devices, such as a sensor or a display, hiding the details of how this is done from the higher-level software.

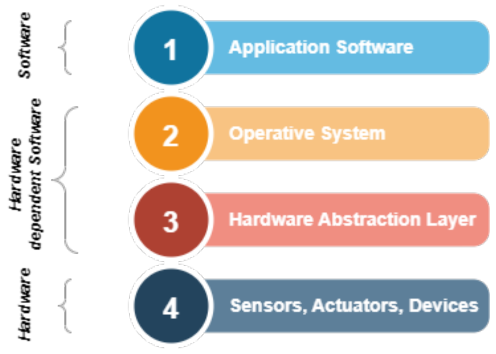
Software
Hardware dependent Software
Hardware
1 Application Software
2 Operative System
3 Hardware Abstraction Layer
4 Sensors, Actuators, Devices

Figure 11: Hardware-abstraction-layer (Based on [Jörg et al., 2014])

In the context of hardware simulation, a HAL can be used to help simulate the behavior of a hardware component by creating a software interface of the simulated hardware component, i.e., a set of functions that define how that component can be used or accessed by other software, providing a standardized way of communicating with it.

A hardware simulation technique could then implement this interface and be used in place of actual hardware when running the application software, making it possible to test and debug the software without having to access the physical hardware.

For example, suppose that an embedded system is being developed, which includes a sensor connected to a microcontroller. To test the software that reads data from the sensor, a hardware simulation technique can be used to simulate the sensor's behavior by running a software model of the sensor. However, the microcontroller would still be running on the actual hardware. To interface the software simulation model of the sensor with the microcontroller, a HAL can be utilized to provide a software interface for the sensor. The HAL would provide functions that the microcontroller's software would use to interact with the simulated sensor as if it was the real hardware. This approach can decouple the hardware-dependent components of the system from the application software.

2.2 Software Testing

Software testing is an essential process in the development of high-quality software applications. ”It is the process of evaluating a software program to ensure that it performs its intended purpose”, verifying safety, reliability, and correct working of a software, quoting [Umar, 2020]. Software tests' goal is to help to identify defects, errors, and issues in the software, ensuring that it functions correctly and meets the specified requirements.

It is important to clarify the meaning of defects and errors in the context of software testing, as they differ from each other. A defect is a deviation from the specified requirements or design of the software, which may cause a system to behave in unintended ways. An error, on the other hand, is a human action or decision that produces an incorrect or unexpected result. Errors can then cause defects to show up on the software, deviating it from its original requirements.

A rigorous testing of the components and systems, and their associated documentation, is important to diminish the risk of failure Board [2018], since it ”verifies that a system is correctly built and validates that it will meet users' and stakeholders' needs” [Graham et al., 2021]. This leads up to a better final product which translates into client satisfaction as well as a cost reduction in the maintenance of the software.

2.2.1 Process of Software Testing

There are various methods for conducting software testing, but some essential test activities are necessary in order to effectively achieve the desired testing objectives (taking into consideration [Board, 2018] and [Jamil et al., 2016]), as can be seen in Figure 12.

- **Test Planning** - In this step a test plan is devised and delivered, taking into consideration the test's context and its objectives. For that there also occurs a requirement elicitation.

- **Test Development** - In this phase, test cases are designed based on the test conditions previously defined and then implemented. ”Each test case specifies a set of test inputs or data, execution conditions, and expected results” [Jamil et al., 2016].

- **Test Execution** - This phase is comprised of the execution of the test cases based on the test plan that was devised. A comparison between expected and obtained results is made for each test and the defects that were found and properly reported. The test results are also registered.

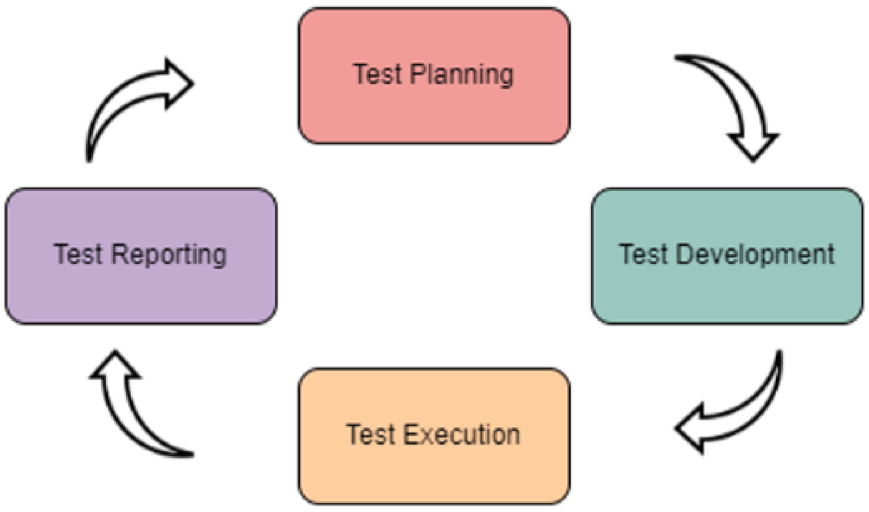

Figure 12: Test Process Phases

- **Test Reporting** - In this phase a report about the execution of the test cases is elaborated, containing a summary of test metrics and results obtained.

As shown in Figure 12, although these activities may appear logically sequential, they often happen iteratively, since most software development disciplines used are of an iterative nature.

2.2.2 Testing Principles

In accordance with [Board, 2018], [Graham et al., 2021] and [Altexsoft, 2016] there are seven fundamental principles that offer general guidelines that are applicable to all testing:

1 - Testing shows the presence of defects, not their absence

Regardless of the number of tests done, it is impossible to guarantee that there are no defects present. "Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, testing is not a proof of correctness" [Board, 2018].

2 - Exhaustive testing is impossible

It is not feasible to test every combination of data inputs and preconditions of software ([Altexsoft, 2016]), except for trivial cases. Instead of testing exhaustively, risk analysis and test techniques should be adopted and an optimal should be found as shown in Figure 13.

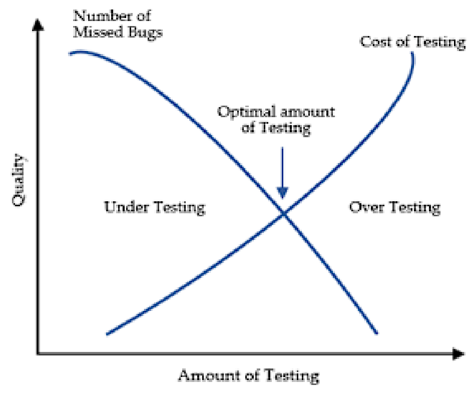

Figure 13: Optimal Amount of Testing (Source: [Jamil et al., 2016])

3 - Early testing saves

Since the cost of an error grows in an exponential manner throughout the development of software, according to [Altexsoft, 2016], testing from an early point will prevent most issues from snowballing into bigger problems.

4 - Defect clustering

Usually, most of the defects are situated in the same modules, being this principle often referred to as an application of the Pareto principle [Altexsoft, 2016] (that affirms that for a lot of outcomes, 80% of the results come from 20% of the causes). Trying to predict the clusters where the defects occur can improve the software development process as a whole.

5 - Pesticide paradox

If the same tests are repeated again and again, eventually they will stop finding any new defects (tests are no longer effective at finding errors just as pesticides are no longer effective at killing in continuous use), citing [Board, 2018]. It is important to review and update tests regularly to avoid this pitfall.

6 - Testing is context dependent

Depending on their purpose and industry, different software should be tested differently, according to [Graham et al., 2021].

7 - Absence-of-errors fallacy

It is a mistaken belief that just finding and fixing a large number of defects will ensure the success of a system, i.e., a system's success is not necessarily determined by the complete absence of errors. As exemplified in [Graham et al., 2021], "testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfill the users' needs and expectations".

2.2.3 Test Approaches

As can be seen in Figure 14, there are two fundamentally different ways to test software that complement each other, as explained in [Umar, 2020].

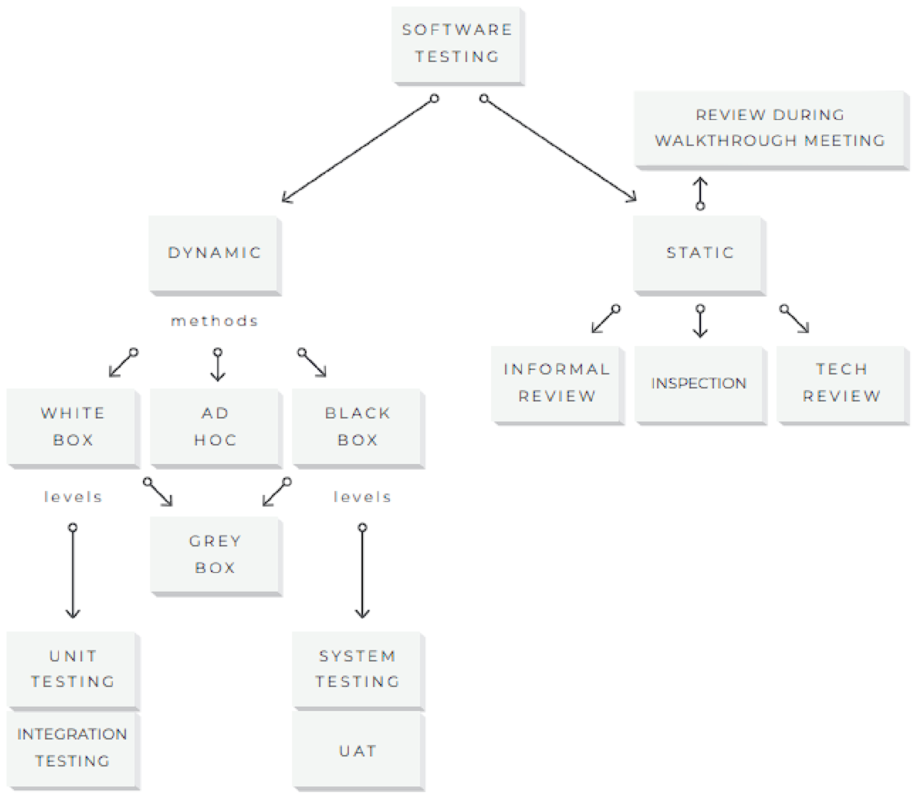

Figure 14: Static and Dynamic Testing (Source: [Altexsoft, 2016])

Dynamic Testing

In a dynamic test, the software is evaluated during its execution to ascertain its quality, being the majority of software tests performed inserted in this category, in accordance to [Umar, 2020].

Static Testing

"Static testing initially examines the source code and software project documents to catch and prevent defects early in the software testing life cycle" ([Altexsoft, 2016]), creating an estimate of the software quality without any actual executions [Umar, 2020].

2.2.4 Test Levels

Software testing can be performed at different phases of the development process, and by performing testing activities at these different levels a better quality of software can be achieved. As explained in Board [2018] "test levels are groups of test activities that are organized and managed together" where each test level has specific objectives and consists of a group of procedures applied to "software at a given level of development".

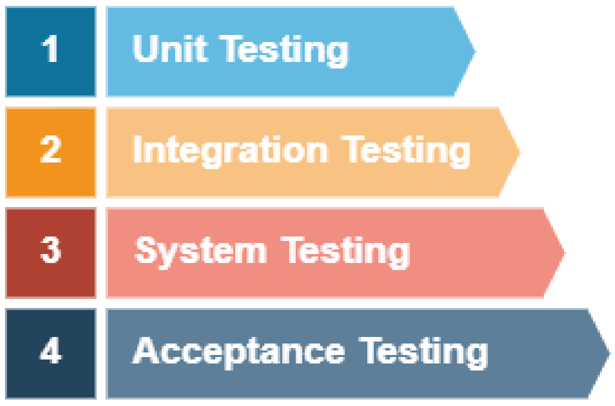

Figure 15: Software Test Levels

Software tests can be fitted in one of the following levels: unit, integration, system, and acceptance tests, as stated by Graham et al. [2021]. These test levels are typically organized in a hierarchy, with each level building on the previous one and increasing in scope and complexity.

Unit Tests

Unit tests, also known as component tests, search for defects by verifying the behavior of software through the test of individual components that are separately testable such as classes, objects, and modules. This type of test is usually based on "the requirements and detailed design specifications applicable to the component under test" [Graham et al., 2021]. According to [Board, 2018], unit testing is normally done in isolation from the rest of the system often requiring the use of mock objects and virtualization of services and may cover both functional and non-functional characteristics of the system in hand.

Unit tests are ”especially useful for guarding the developers against programming mistakes and for localizing the errors when they occur”, as stated in [Peng et al., 2020], reducing risk by preventing defects to propagate to later testing [Graham et al., 2021].

Integration Tests

”Integration testing is the testing applied when all the individual modules are combined to form a working program”, as explained in [Leung and White, 1990], focusing on interactions between components or systems. This type of test is commonly based on ”the software and system design [...], the system architecture [...] and the workflows or use cases by which the stakeholders will employ the system” [Graham et al., 2021]. According to [Board, 2018], integration tests can be categorized into one of the following two types:

- Component integration testing, whose focal points are the interactions and interfaces between integrated components;
- System integration testing, which focuses on the interactions between different systems, packages, and micro-services.

Integration tests can be used to determine whether the modifications implemented have affected the way components and/or systems behave, providing or not confidence to the changes made, as explained in [Graham et al., 2021].

System Tests

System testing takes into consideration the behavior of the system/product as a whole, ”often considering the end-to-end tasks the system can perform and the non-functional behaviors it exhibits while performing those tasks”, as mentioned by [Board, 2018]. In this type of test, it is crucial that the test environment resembles, as much as possible, the final product environment ([Graham et al., 2021]), in order for the test results to be applicable. In other words, system testing enables the verification in a ”non-live” scenario ([Freeman, 2002]).

Acceptance Tests

Acceptance testing has the goal of ”establishing confidence in the quality of the system as a whole”, verifying if functional and non-functional behaviors of the system are as specified, as shown in [Graham et al., 2021]. They represent the interests of the end-user since they provide confidence that the ”application has the required features and that they behave correctly" [Roy W. Miller, 2005]. During acceptance testing, the software is tested in a simulated or real-world environment to ensure that it functions as intended and meets the needs of the end-user. Usually, this type of test is done near the end of the software development process (or at the end of an iteration in case the development is iterative itself) to ensure the product is ready for deployment.

It is necessary to point out that acceptance testing is often confused with system testing, but both test types provide complementary perspectives on the software. While system tests ensure that the software works correctly from a technical perspective, acceptance tests make sure that it meets the needs of the business.

2.2.5 Test Types

"A test type is a group of test activities aimed at testing specific characteristics of a software system" ([Board, 2018]) having in mind a specific test objective, that is, with each test type serving a different purpose, as stated by [Umar, 2020].

Since there are a lot of different test types and there is no consensus across the community on which are the most fundamental test types, this dissertation will take into consideration the results of a survey done in 2018 by the International Software Testing Qualifications Board ([ISTQB, 2018]) as well as what is considered by [Graham et al., 2021].

Functional Testing

Functional tests focus on validating the software behavior against the business functionalities documented in the software requirements and specifications, as explained in [Everett and McLeod, 2007]. This type of test should be performed at all test levels although it should be done thoroughly "within the levels of system and acceptance" [Altexsoft, 2016].

Functional tests can be further categorized into several types, the most relevant being:

- **Use Case Testing** - Use case testing involves testing the system by executing the steps outlined in the use cases to ensure that the system is producing the expected results. A use case is a description of a specific way in which a system is intended to be used. By using as a guideline the use cases of a system, this type of test "checks whether the path used by the user is working" as intended ([Altexsoft, 2016]).

- **Exploratory Testing** - Exploratory testing is a spontaneous approach to testing that involves evaluating a system and its functionality without relying on " predefined and documented test cases and test steps" ([Altexsoft, 2016]), emphasizing the "personal freedom and responsibility of the individual tester to continually optimize the value of her work", as defined in [K08]. In this type of testing, test design, test execution, and test result interpretation are all done concurrently, influencing each other. It is, however, important to document all steps taken so that the test can be replicated if needed.

Non-Functional Testing

This kind of test, as explained by [Board, 2018], "evaluates characteristics of systems and software such as usability, performance efficiency or security" and should be done at all testing levels, since a late discovery of a non-functional error "can be extremely dangerous to the success of a project". Non-Functional tests are concerned with how well the system operates and performs under certain conditions, rather than with the specific functionality of the system.

Non-Functional tests can be further categorized into several types, the most relevant being:

- **Performance Testing** - Aims to investigate and evaluate the system responsiveness and stability under different loads, as stated in [Altexsoft, 2016]. It is concerned with verifying that the system can handle the expected workload and perform at an acceptable level, validating if its execution time, or 'speed' as referred in [Everett and McLeod, 2007], is tolerable;

- **Usability Testing** - It is used to evaluate "user performance and acceptance" of the system, as stated in [Wichansky, 2000], i.e., it accesses how easily a user can learn and use the system and its satisfaction level. As mentioned by [Altexsoft, 2016], this type of test ensures the implementation approach works for the end user.

- **Security Testing** - Security testing is "the process to identify whether the security features of software implementation are consistent with the design", as defined by [Tian-yang et al., 2010]. It aims to identify vulnerabilities in the system that could be exploited by an attacker, ensuring that protection mechanisms are implemented and that they cover the software and its related data [Freeman, 2002].

Change-Related Testing

Change-Related tests evaluate the impact of changes on a system, checking if the changes corrected a defect or added a new functionality correctly, not having created any "unforeseen adverse consequences"

([Board, 2018]). This type of test can be of one of the three following categories:

- **Confirmation Testing** - Also known as Re-Testing, this type of test is done after the correction of a defect to check if the problem source was dealt with properly. To perform this type of test it is important to guarantee that "the steps leading up to the failure are carried out in exactly the same way as described in the defect report", as mentioned in [Graham et al., 2021] to ensure the defect was indeed fixed.
- **Regression Testing** - Since it is possible that "a change made in one part of the code, whether a fix or another type of change, may accidentally affect the behavior of other parts of the code" ([Board, 2018]), that is, that a change might have unintended side-effects, it is necessary to prevent these scenarios. Regression tests make sure that changes don't have collateral damages and that the "system still meets its requirements", as explained in [Graham et al., 2021].
- **Smoke Testing** - Smoke testing is designed to be run quickly in order to establish the software safety or faultiness before further testing, not being exhaustive as regression tests ([Everett and McLeod, 2007]). "The term 'smoke test' comes from the hardware engineering practice of plugging a new piece of equipment into an electrical outlet and looking for smoke. If there is no sign of smoke, the engineer starts using the equipment", as explained in [Everett and McLeod, 2007].

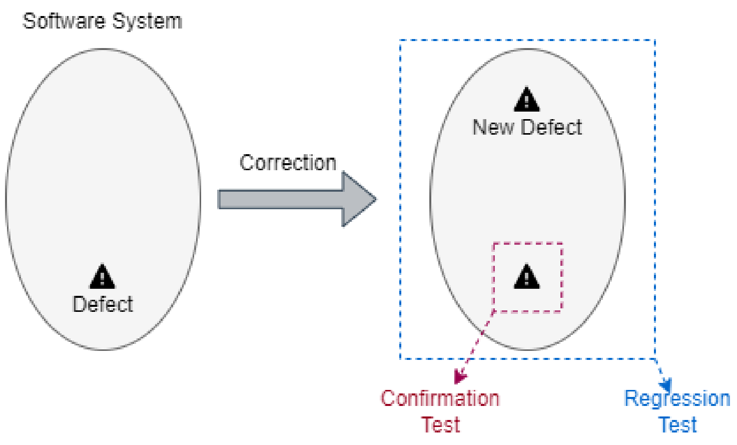

Figure 16: Change-Related Testing Types (Based on [Genius and Gupta, 2018])

As shown in Figure 16, after correcting a defect it's important to check if that change has brought in the desired functionalities (Confirmation Test), but also if no undesired side-effects were introduced (Regression Test). Therefore, both Regression and Confirmation tests should be held.

2.2.6 Testing Techniques/Strategies

A testing technique specifies the strategy that is used to develop the test cases and for analyzing the results [Umar, 2020], helping identify test conditions, test data, and test cases, as explained in [Board, 2018]. Some techniques are more appropriate for certain test levels while others are more applicable to all test levels, and, for that reason, the techniques used will largely depend on the situation ([Graham et al., 2021]).

Black Box Testing

Black Box testing is a technique ”in which the internal structure/ implementation of software being tested is not known to the tester”, citing [Umar, 2020]. In this technique, the system is viewed as a ’black box’ where only the inputs and outputs of the software are considered, existing no knowledge of the system’s internal workings Board [2018]. Its main advantage is to not require access or knowledge of the software code, allowing for quicker test development. It also prevents bias in test development, as the tests are designed without knowledge of the software’s internal structure. On the other hand, this simplification limits the coverage that tests can have. This method can be applied to any testing level but is used mostly for system and integration tests [Umar, 2020].

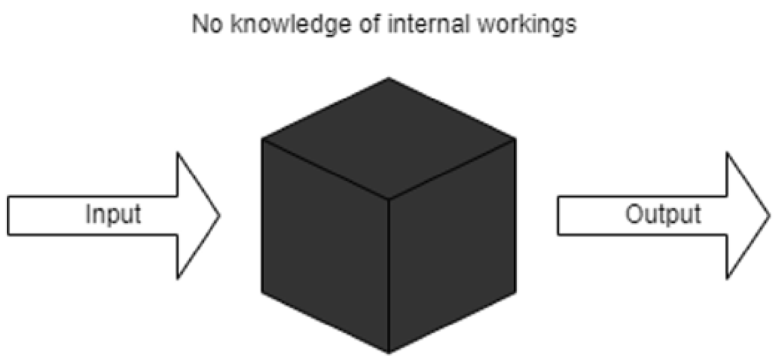

Figure 17: Black Box Testing

White Box Testing

White Box testing is a technique that uses the internal structure of the software to derive test cases [Graham et al., 2021], having as the main objective covering the ”logic” part of the system, as stated in [Everett and McLeod, 2007]. Unlike Black Box testing, it concentrates on the structure and processing done within the object being tested [Board, 2018]. Although it can help to uncover defects that might be missed by other testing techniques such as Black Box Testing, it can also be more time-consuming as it requires a thorough knowledge of the inner workings software being tested.

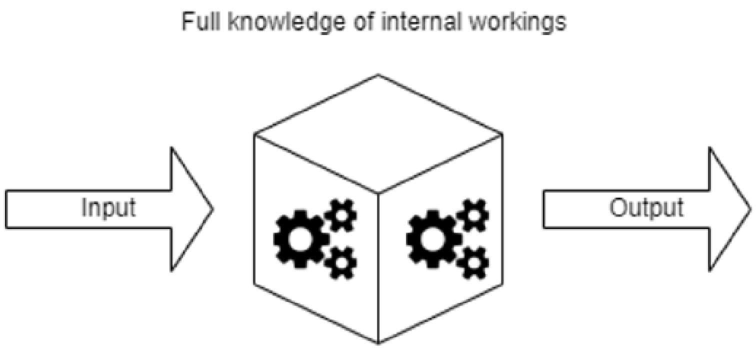

Figure 18: White Box Testing

Grey Box Testing

"Grey Box Testing is the combination of the White Box and Black Box Testing Technique serving the advantages of both" [Jamil et al., 2016], since it involves testing both the structural and functional parts of the system. It requires more understanding of the internal working of the software when compared to Black Box testing but less than White Box Testing, using some knowledge about the software to design Black Box style tests, as explained by [Umar, 2020]. According to [Umar, 2020] and [Altexsoft, 2016], this testing technique is most effective when done at the integration level.

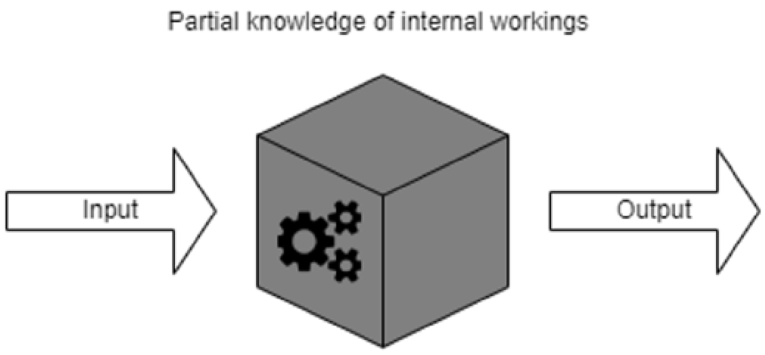

Figure 19: Grey Box Testing

Ad Hoc Testing

Ad Hoc testing is an informal testing style since it is performed without any kind of plan or documentation, as stated in [Altexsoft, 2016]. Testes are conducted randomly, "on the fly" as the tester explores the system and can be helpful in identifying problems that wouldn't be easily spotted by more systematic and formal techniques. However, defects found by this strategy might not be easily reproducible ([Altexsoft, 2016]).

2.2.7 Testing Metrics

Software testing metrics are measurements used in order to evaluate the quality of software, the effectiveness and efficiency of the testing process. They consist of quantifiable indicators that give information about "the quality facets relevant to the process and product" [Jamil et al., 2016] and "can be collected during and at the end of test activities" [Board, 2018]. Test metrics can be used to identify necessary improvements in order to produce a high-quality software product.

Some commonly used test metrics are:

- **Test case pass/fail rate** - Measures the percentage of tests that passed. A high pass rate is generally desirable, as it indicates that the software is functioning properly.
- **Test coverage** - Measures the percentage of software code that is covered by tests. A higher coverage indicates that more of the software has been tested.
- **Test execution cost** - Considers the resource allocation and usage and the time it took to run the test. Smaller costs are preferable since it means that the tests run faster and/or use fewer resources to execute.

The metrics shown above should not be analyzed individually but as a whole. For example, a system with a low number of tests, i.e., a low test coverage, can have a high pass rate in tests and a low execution cost of the tests. If the test coverage would not be taken into account this system could be trusted to function properly, which is not the case.

2.2.8 Manual and Automated Testing

There are two distinct ways to execute software tests: manually and automatically.

- **Manual Testing** - It's the process of manually executing and validating the correctness of software. It requires that a person, following a set of instructions, conducts a certain test and then registers the results obtained.

- **Automated Testing** - It's the process of using software tools to execute and validate the correctness of software. These tools imitate human interactions with the software in an autonomous manner, following a script where all the test instructions are provided.

Automated testing allows for the execution of a large number of test cases in a short period of time, easing "the burden of managing all of the testing needs" as stated in [Altexsoft, 2016]. It is also less prone to human error, as it eliminates the need for manual testing efforts since tests "perform precisely same operation each time they are run" ([Sharma, 2014]). However, automated testing requires a significant investment of time and resources, as it involves the development and maintenance of automated test scripts.

On the other hand, manual testing allows for more flexibility and creativity in the testing process, which can be especially useful in detecting usability problems. However, since it requires the tester to manually perform the tests, it is prone to human error and naturally slower, as explained in [Sharma, 2014].

Both approaches have their advantages and disadvantages being necessary to carefully consider the costs and benefits of each approach before deciding on how to execute the tests.

Chapter 3

The problem and its challenges

The software from Smartex's Vision Project is divided into five main components, which run on four different physical devices: **Sensing Service**, **Machine Service**, **Machine Learning Service**, **Core Service** and **Console Service**.

The software from **Sensing Service** and **Machine Service** are dependent on specific hardware components (sensors and actuators). This dependency makes the development and testing, of these software modules, significantly slower since the physical hardware components need to be available.

With the goal of improving the software development and testing phases of Smartex's Vision project, this dissertation aims to create a simulation platform that bypasses the hardware dependencies, by simulating their exact behaviors in the system and to migrate and implement a set of integration tests done by the Quality Assurance team to the developed platform. Finally, this dissertation also has as a goal the test of the created platform's efficacy in regard to the improvements it brings for the software development and testing phases.

It is important to note that for each type of hardware (sensors and actuators), there are already developed APIs that allow for communication with the different hardware components. For this reason, the simulation will be done at this level, meaning that the software will still communicate with the already built APIs, without having any idea that it will be communicating with the simulation software developed instead of the real hardware component. In other words, the simulation to be developed will not simulate the inner workings of the hardware used, only their interactions with the system.

The main challenges this dissertation faces are:

- Doing a good requirement elicitation, that is able to enclosure all the needs of the Quality Assurance team in order to understand what can be improved with this dissertation work;
- Develop a software solution capable of accurately simulating the hardware components that Smartex's Vision Software needs;

- Establish a testing methodology capable of measuring the impacts obtained by the software solution that will be created in this dissertation.

Part II

Core of the Dissertation

Chapter 4

Vision Software

This chapter provides an in-depth analysis of the initial state of this dissertation and proposes a comprehensive approach to address the problem outlined in chapter 1, framing it in the theoretical context revised in chapter 2.

4.1 Existing Work

4.1.1 Software Architecture of Vision Software

As mentioned before, the *Vision Software* Project, where the scope of this dissertation is set, is composed of five different software services that run individually and communicate between them.

In the diagram portrayed in Figure 20, it is demonstrated how the different software modules interact as well as their hardware dependencies.

The **Core Service**, as hinted by its name, orchestrates all the other services, containing the core logic of the product.

The **Sensing Service** focuses on communicating with cameras in order to take pictures of the fabric as it is being produced, as well as getting data from other sensors installed on the knitting machines. It is then hardware dependent.

The **Machine Service** is also hardware dependent since it interacts with sensors to extract information about the knitting machine but also with actuators in order to stop the knitting machine if needed.

The **Machine Learning Service** is responsible for using artificial intelligence models to infer the quality of the fabric being produced.

The **Console Service** acts as a user interface where it is possible to consult the state of the knitting machine as well as to interact with the services (through the **Core Service**).

It is important to note that the services that interact with hardware do so through the use of **APIs** that

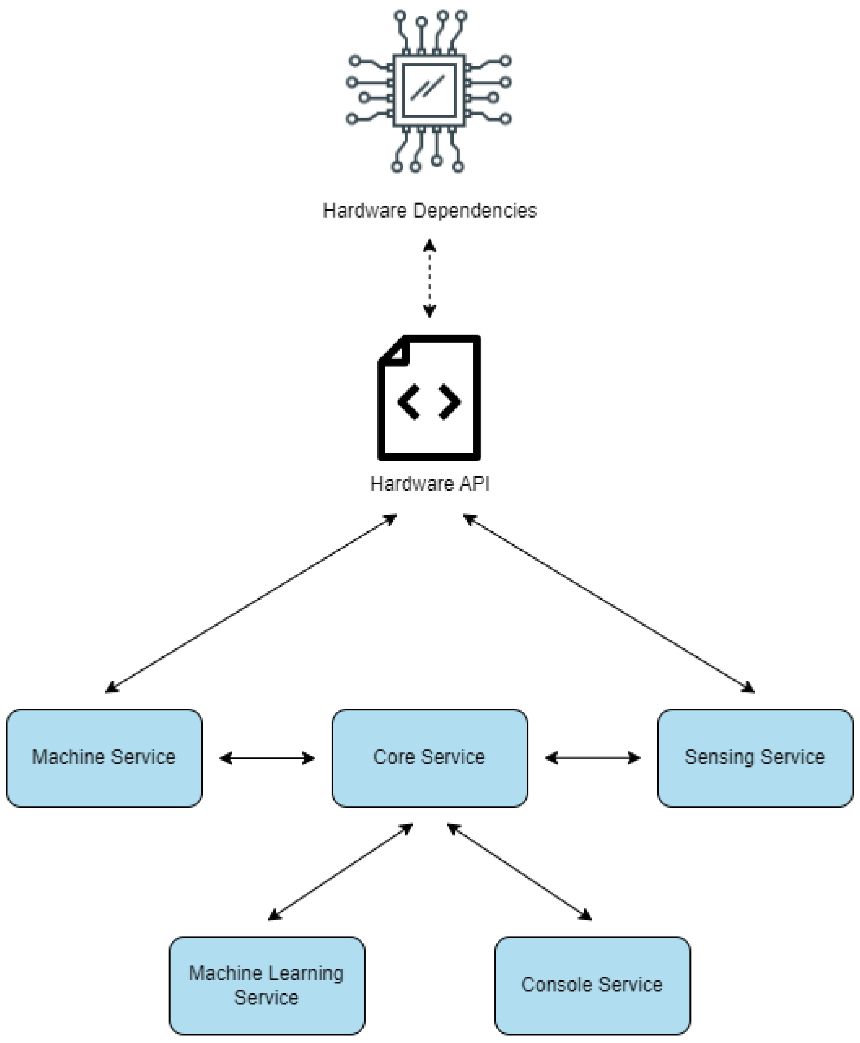

Figure 20: Architecture of the Software

make it possible to exchange data with the hardware dependencies.

4.1.2 Matrix Project

The **Matrix Project** as it was nicknamed, is the approach that was devised with the goal of decoupling the *Vision Software* services from their hardware dependencies, without interfering with their inner workings, i.e., maintaining the structure explained in Figure 20, facilitating the development and testing phases of the *Vision Software*.

At the beginning of this dissertation, the *Matrix Project* had each of the services but the *Console Service* running on individual *docker containers*. The hardware dependencies problem was tackled by having software classes that implemented the same *interfaces* that the hardware components did. These classes, for hardware sensors, would return a value within a given probability distribution. However, for hardware actuators, the classes that implemented their *APIs* would simply not act upon anything, and for that to happen it would be necessary to implement extra logic in the *Sensing* and *Machine* services, since all the classes that implemented the hardware dependencies' *interfaces* were actually instantiated in the services themselves, as can be seen in the *Machine* and *Sensing* services depicted in Figure 21. It should be noted that, when this dissertation began, the Matrix Project was not yet operational. The services had not yet been fully containerized, and testing had not been conducted to ensure proper functionality of the system as a whole.

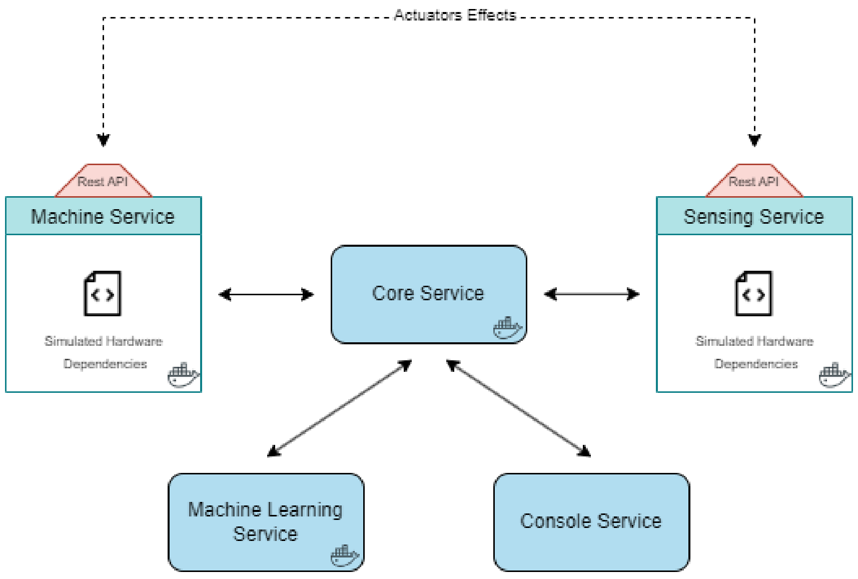

Figure 21: Initial Architecture of the Matrix Project

4.2 Proposed Approach

With the aim of improving the simulation fidelity of the *Matrix Project*, in this dissertation, it was designed and implemented a solution called **Machine Layer**, a software layer that precisely represents a real circular knitting machine, incorporating software classes that simulate all hardware dependencies including the actuators, without the need of changing the code of the *Vision Software* services and risking compromising its correct functioning.

The *Machine Layer*, by encapsulating all the simulation logic within itself, makes it so each *Vision Software* service is totally unaware that it is not interacting with a real **Circular Knitting Machine** (**CKM**).

4.2.1 Proposed Architecture

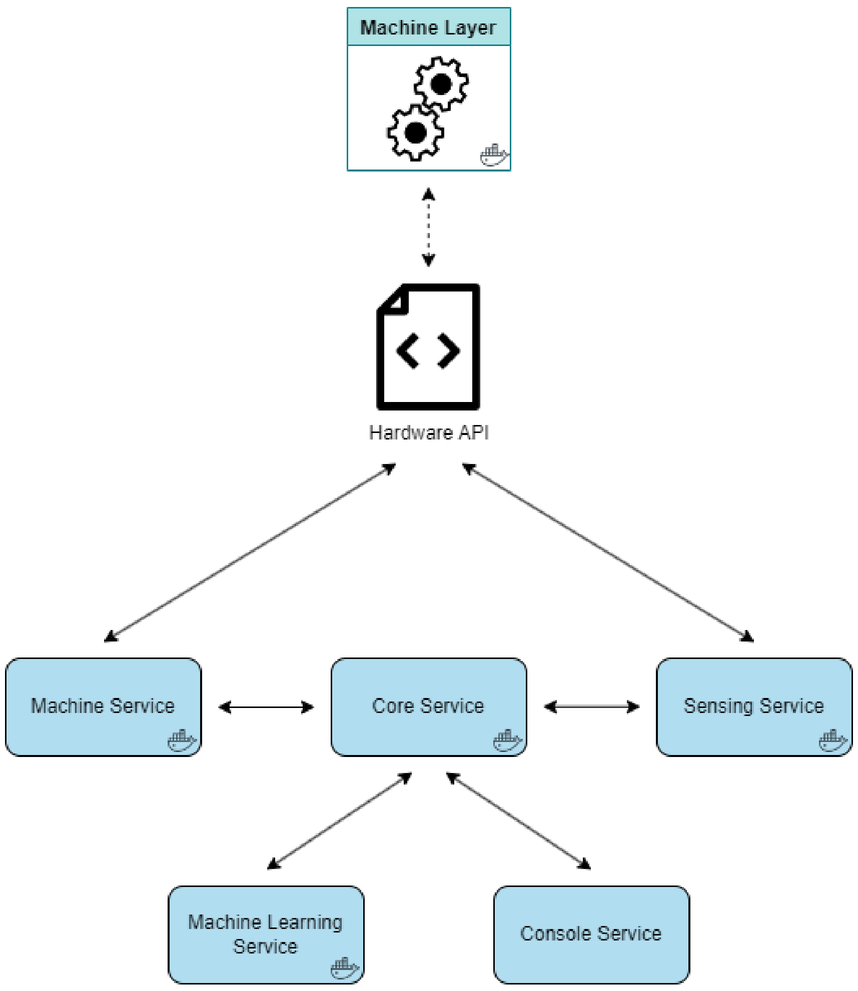

Figure 22: Proposed Simulation Architecture of Vision Software - Matrix Project

As it was already implemented in the *Matrix Project*, each *Vision Software* service runs in an individual docker container, being in a controlled and platform-free environment which allows the *Matrix Project* to be executed in any machine with ease.

With the objective of not changing any of the logic of the services, a new layer that implements all the hardware dependencies *APIs* was created, but instead of communicating with the hardware, it would communicate with a new software layer nicknamed *Machine Layer*, as depicted in Figure 22.

By implementing all the used hardware *APIs*, the *Sensing* and *Machine* services have no idea that they are not talking with real hardware, but instead, with a software simulation of them.

The *Machine Layer*, in order to be able to simulate inputs for every sensor and the actions of every actuator, has an internal state that replicates one of a real knitting machine.

With the goal of mimicking with precision a real machine, the *Machine Layer* not only has a state that contains all the variables present in a *CKM* but also implements the relationships between these state variables.

With this system's architecture, it becomes possible to run the entire *Vision Software* on a single machine, without needing to have any external piece of hardware.

4.2.2 Software Development and Testing Enhancements

With proposed changes, the *Matrix Project* is capable of being used and can have a positive impact on the *Software Development* and *Quality Assurance* teams:

- *Software Development* - it allows the *Vision Software* to be developed without needing its hardware dependencies;
- *Software Testing* - it makes it possible to test only the software by simulating all hardware dependencies or even test a specific hardware component by simulating all the rest.

4.2.3 Theoretical Framing

Simulation Behaviour

In regards to the simulation of the behavior of the hardware dependencies, the simulation model chosen is stochastic in nature, since there are hardware sensors whose outputs have a random component to them, and it is also static in regards to the passage of time because the values produced by the hardware sensors are not dependent of the elapsed time of the simulation. The Machine Layer, however, is based on both Discrete-Time-Simulation and Discrete-Event-Simulation paradigms, blending them together. On the one hand, when a hardware actuator acts upon the Machine Layer, a *DES* approach is taken, being the action itself seen as an event that triggers a state change of the Machine Layer. On the other hand, the passage of time itself also triggers a state change of the Machine Layer, a *DTS* approach, to better replicate the real behavior of a CKM, as explained in more detail further on.

Choosing between Emulation and Simulation

Given the objectives of this dissertation, it was chosen to simulate the hardware dependencies, as there is no need to test how the software interacts with the underlying hardware dependencies. Furthermore, a simulation allows for adjustments in the execution speed of the system, which can prove to be very useful when testing the *Vision Software* project.

Choosing a Hardware Simulation Technique

A Software in the Loop (SIL) approach combined with a Hardware Abstraction Layer (HAL) was utilized during this dissertation in order to facilitate the development and testing of the *Vision Software* project. The SIL technique was used to simulate the behavior of hardware components using software models, while the HAL was used to provide a software interface for the simulated hardware components. By combining these two methodologies, the hardware-dependent details of the system were decoupled from the application software.

It is important to acknowledge that an approach integrating Software in the Loop and Hardware in the Loop is also viable, given that *Matrix* offers the flexibility to simulate certain hardware components while retaining real hardware dependencies. However, the chosen approach for this dissertation is better aligned with its overarching objectives.

Test Levels covered

Regarding test levels, this dissertation's work focused only on **integration** tests. Unit tests for *Vision Software* were already being made and did not take any benefit from the *Matrix Project* and the System and Acceptance test levels required the existence of the hardware dependencies since these types of tests require aim to test the system as a whole. The **integration** tests developed consisted in asserting key product's features and flows, either in a specific service or encompassing several services.

Test Types covered

With respect to the test types, the implemented tests consisted of already developed test cases by the **Quality Assurance** (**QA**) team that were previously done manually. These were mainly **smoke tests**, which quickly checked if the core features of the *Vision Software* were working, and **use case tests**, which aimed to ensure that specific use tests were correctly functioning.

Test Techniques used

The developed tests all made use of *REST APIs* and/or *websockets*, which implies some knowledge of the inner workings of the *Vision Software* services. However, besides this knowledge, there no further information on how the services functioned on the inside was taken into consideration. For these reasons, the test technique that was mainly used was the **grey box testing** technique.

Chapter 5

Contribution

This chapter rigorously analyzes the solutions developed during this dissertation, as well as the challenges encountered along the way. It offers a thorough understanding of the solutions employed by providing a detailed overview of the thought process involved in reaching them.

5.1 Development of the Machine Layer

5.1.1 Defining a Software Representation of a Circular Knitting Machine

Defining all variables present in a CKM

In Smartex's actual *Vision System*, hardware sensors and actuators are divided among two modules: *Apex* and *Finis*, which run the Sensing and Machine services respectively.

Hardware dependencies of the **Machine Service**:

- **HTU:** *Sensor* that gathers information about the temperature and humidity of the machine;
- **Relay:** *Actuator* that allows stopping the machine;
- **Potentiometer:** *Actuator* that works as a variable resistance.

Hardware dependencies of the **Sensing Service**:

- **Backup Battery:** *Actuator/Sensor* capable of providing the *Apex* with energy and measuring its voltage;
- **IMU:** *Sensor* capable of measuring the *RPMs* of the machine;
- **Fans:** *Actuator* that turns on or off the fans in the *Apex*;

- **USB Controller:** *Actuator/Sensor* capable of checking if the USB ports used to communicate with cameras are working and able to turn them on/off;
- **Lamp Controller:** *Actuator* able to change the type of lights used by the cameras;
- **Power Supply:** *Sensor* that provides information temperature, voltage, current, and power about the *Apex* module;
- **Camera:** *Sensor* capable of taking pictures of the fabric being manufactured.

Mapping out interactions between the existing components

Implementing Power Supply - Backup Battery Logic

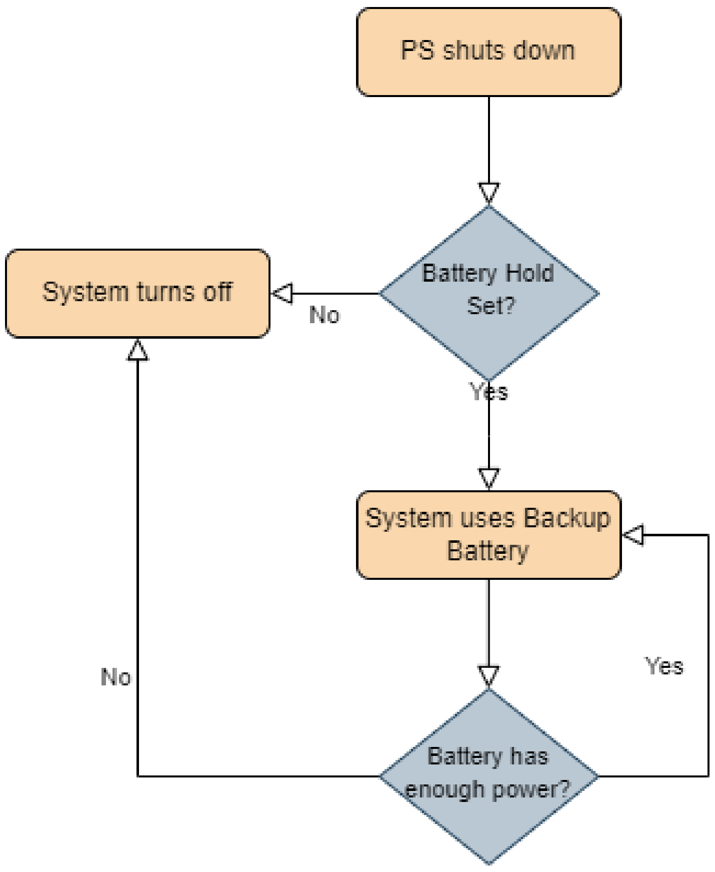

Figure 23: Implementing the power logic in the Machine Layer

The *Vision Software* system is designed for high availability, meaning it is intended to remain operational even in the event of a power supply (PS) failure. The system includes a backup battery that can sustain system operation when the PS is unavailable, relying on the backup battery only when necessary and not using it if the flag called *battery_hold*, which indicates whether the backup battery should be used, is not active.

In the event of a PS failure, if the *battery_hold* flag is active, the backup battery is activated and powers the system until the PS is restored or the backup battery is depleted. If the backup battery is unused or runs out of power, the system shuts down completely.

When the PS is restored, the system automatically switches back to using it and begins charging the backup battery until it is fully charged again. Figure 23 illustrates the operation of the system when the PS fails.

It was essentialt to implement the interaction between these two components to make it possible to test the various scenarios depicted in Figure 23 through the use of the Machine Layer.

Implementing Cameras’ Logic

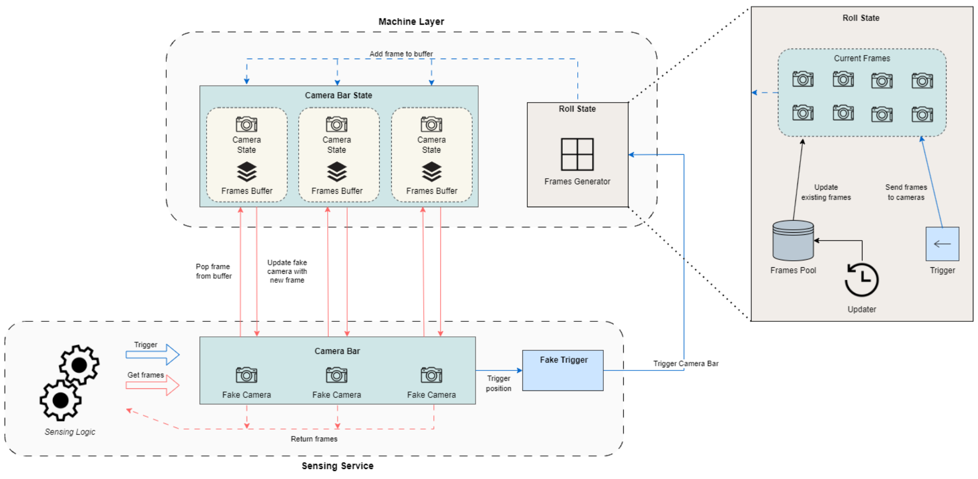

Figure 24: Implementing the logic of cameras in the Machine Layer

The Camera sensors in the *Machine Layer* are controlled by the *Sensing Service* that is responsible for all the logic behind taking photos. This process involves two distinct actions: triggering the camera (represented in blue in Figure 24) and retrieving the frames taken (represented in red).

In the physical world, a trigger is sent to a camera bar to open and close each camera’s shutter, resulting in the capture of a new picture that is stored in a buffer in the camera’s memory. To replicate this behavior in a simulation, the Sensing Service triggers a camera bar (containing only fake cameras) that uses a fake trigger object to send to the *Machine Layer* a trigger request.

Once the trigger request reaches the Machine Layer, it is sent to the Roll State, which retains in memory the current frame for each camera of each camera bar in memory. The trigger effect makes it so that these current frames are then written onto the Camera States of the respective camera bar, being added to a FIFO buffer present in each of the *Camera States*.

When the Sensing Service retrieves frames from a camera bar in the real world, it removes and returns the first frame from each camera’s buffer. In the Machine Layer, as shown in red in Figure 24, each fake camera makes a request to its corresponding Camera State to retrieve and remove, i.e., to *pop* the next frame in the buffer. When all frames have been collected, the Camera Bar returns them to the Sensing Service.

Both the trigger and get frames actions are implemented in the Machine Layer so that the Sensing Service cannot distinguish whether real cameras are being used. This simulation of the two mentioned actions holds up even in edge cases such as when no trigger is done, trying to retrieve a frame will result in an error, or performing a number (**x**) of consecutive triggers bigger than the number of pictures that a camera buffer can handle (**y**) will result in the first **x-y** frames being discarded.

Figure 24 also provides insight into the inner workings of an entity created for the *Machine Layer* named *Roll State*. To accurately represent a real machine's roll, the *Roll State* has a frame for every possible combination of camera bar, light type, and light position. These frames are updated periodically based on the machine's rotations per minute (*RPM*) and the amount of fabric created by each rotation (*rapport*). When a trigger is activated, the frames corresponding to the triggered camera bar are copied onto the appropriate *Camera States*.

Implementing Relay Logic

To facilitate the interruption of a knitting machine in the event of a detected fabric defect, a relay is employed to transmit an electrical signal to the machine, causing it to come to a halt. Despite its apparent simplicity, it was of extreme importance to accurately replicate this behavior in the *Machine Layer* in order to be able to test one of the most important logic flows of the whole *Vision Software*. In the *Machine Layer*, when the relay is activated the RPMs are lowered to 0, which, in turn, results in the frames being taken by the cameras always being the same. This is because the *Frames Pool*, responsible for generating the cameras' pictures, is updated periodically, as mentioned in Figure 24. This updated period, in minutes, is obtained by taking into consideration the amount different factors, as follows:

$$update\_period = \frac{vertical\_fov}{RPM \cdot rapport} \tag{5.1}$$

where:

- **vertical_fov:** represents the vertical field of view of the cameras, measured in *centimeters*;
- **RPM:** represents the rotations per min of a knitting machine, measured in rotations per minute;
- **rapport:** represents the amount of fabric generated by one rotation of the knitting machine, measured in *centimeters* per rotation.

5.1.2 Machine Layer Communication Design

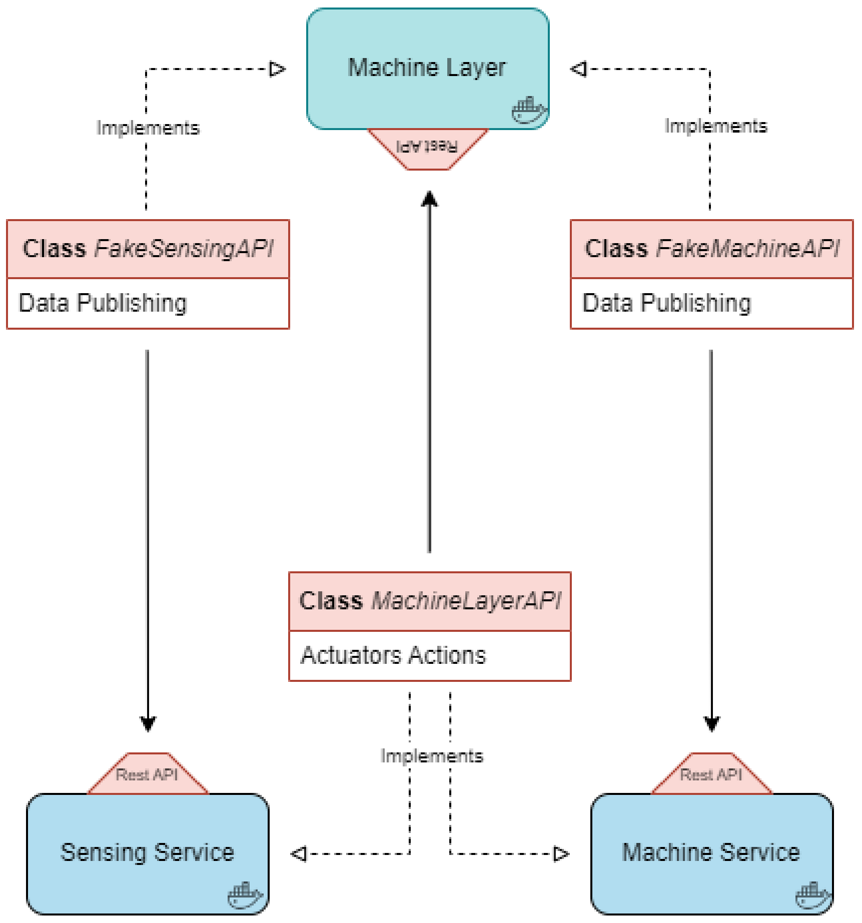

Figure 25: Machine Layer communication with the rest of the Matrix Project

The communication between the Machine Layer and the already existing hardware-dependent services is done through *REST API* over *HTTP* since all the **Vision Software** services already communicated with each other in that way. This communication choice also assured there was backward compatibility with what already has been done in the **Matrix Project**.

As explained before, when a service needs to fetch a value of a hardware sensor it is dependent on, in a normal execution, it would simply get that value. However, in the **Matrix Project** scenario, this would translate in a service fetching the value of that sensor in the Machine Layer. Keeping this traditional flow would prove to be troublesome since Machine and Sensing services fetch their sensor's data at high rates. For example, Sensing Service fetches the value of the RPMs of the machine 100 times per second, which would translate in 100 *HTTP GET* requests per second onto the machine layer, just for this specific data. In order to tackle this, a **PUB/SUB** approach was taken where in lieu of the Machine and Sensing services sending *HTTP GET* requests to the Machine Layer every time, the Machine Layer was in charge of publishing its state (whenever it updated) to both services. This method requires Machine and Sensing services to store in memory a copy of the parts of the Machine Layer state that are of interest to them, as shown in Figure 28.

5.1.3 Inner Working of the Machine Layer

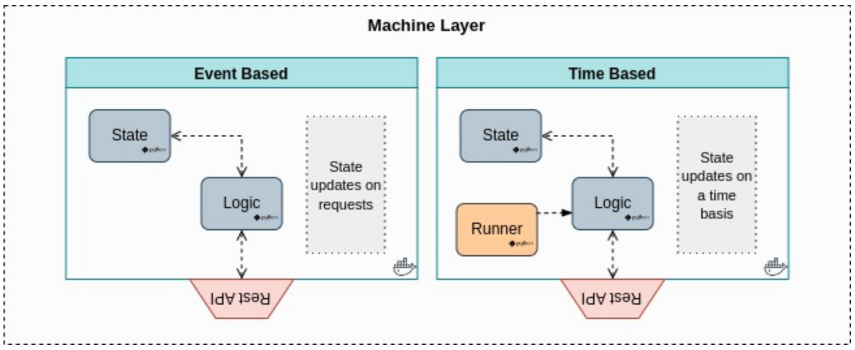

Figure 26: Machine Layer inner workings prototypes

Event Based versus Time Based

When planning how the state of the Machine Layer would be updated there were two different approaches that were considered, shown in Figure 26, updating the state based on events that would occur or based on the passage of time.

In an **Event Based** approach, the Machine Layer state would only update when an *HTTP* request would be received. As explained previously, with a **PUB/SUB** architecture, the Machine Layer would only receive *POST* requests from actuators, so those would be the only events that would trigger an update of the state. In comparison, with a **Time Based** approach, the Machine Layer state is updated on a time basis, so even when no *HTTP* requests are being made to the Machine Layer, its state still updates.

The **Time Based** approach was chosen primarily because it couples better with the chosen *PUB/SUB* design where the state is only published when it changes, allowing for a closer replication of a real *CKM* behavior. Furthermore, choosing an **Event Based** approach would mean that the Machine and Sensing services would only get new sensor data after an actuator had taken action, which is out of touch with how the system works since a *CKM* feeds data into sensors regardless of how the actuators behave.

Detailed approach taken

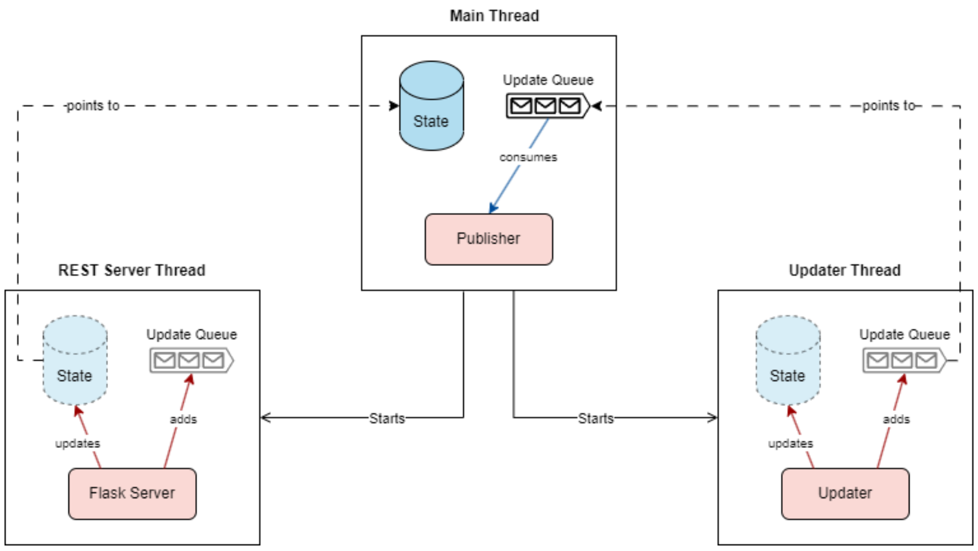

Figure 27: Machine Layer execution

In order to implement an effective system with a *time based* nature and with a *PUB/SUB* communication model, the *Machine Layer* makes use of three different *threads*:

- **REST Server thread** - Contains an HTTP REST server that is continuously listening to incoming messages, which can be subscription requests or *POST* requests made by actuators.
- **Updater thread** - Ceaselessly updates the Machine Layer state at periodic intervals, providing the system with a *time based* characteristic.
- **Main thread** - It creates the data structures that keep the state of the *Machine Layer* and its updates, depicted in Figure 27, spins up the other threads. After that continuously waits for state updates to be added to an update queue, as shown in Figure 27, and consumes all the updates in *FIFO* order, publishing them to all the interested subscribers upon consumption.

As depicted in Figure 27, there are two different data structures that are shared between threads. The **Update Queue** belongs to a synchronized queue class of the *python* standard library, that provides a *FIFO* queue that handles concurrent executions internally. On the other hand, the **State** had to be designed and implemented from scratch, so, it was necessary to develop mechanisms to ensure **thread safety**. To address this, a **Read-Write Lock** class was implemented, allowing multiple readers to simultaneously access the state while ensuring only one writer can update it at a time, thereby preventing **race conditions**. The **Read-Write Lock** implementation ensured that writers couldn't modify the state while readers were accessing it, and readers couldn't read the state while a writer was updating it. To prevent a scenario known as **writer starvation**, where a writer would be waiting for readers to finish reading the state, but new readers would continuously arrive, causing the writer to remain stuck in a waiting state, a mechanism was developed where a writer would announce its intention to write, and readers would check if any writers were waiting before initiating their reading process.

Also, having a thread dedicated to listening to *HTTP Post* requests made by the actuators makes it so the *Machine Layer* can rapidly react to these requests by updating its state, bringing it closer to how a real *CKM* would function.

This 3-thread design makes it possible to switch between an *event based* and a *time based* approach if need be. As it is made clear in Figure 27, both the *REST Server* and *Updater* threads are equivalent, only changing on their update stimulus (events and time respectively). For this reason, running the Machine Layer normally (with the 3 threads) will make it function in a *time based* manner, and disabling the *Updater* thread will make it function in an *time based* manner.

5.1.4 Machine Layer Architecture

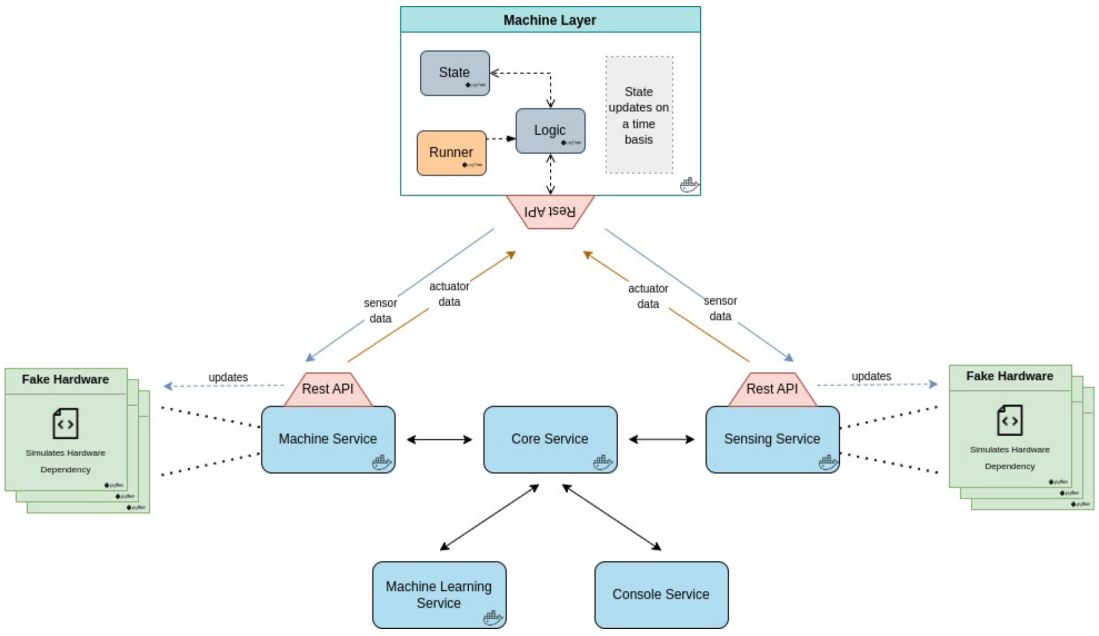

Figure 28: Machine Layer General Architecture

With everything decided and put together, Figure 28 depicts the final architecture of the *Machine Layer*. *Machine* and *Sensing* services, for each hardware dependency they have, instantiate a *Fake Hardware Object* that implements the *Hardware Abstraction Layer* of the given dependency. The *Machine Layer* has for each hardware component an internal state representing it.

When either one of the services needs to consult a hardware sensor value, it just fetches the value that is saved in the respective *Fake Hardware Object*, which is stored in memory. However, when one of the services applies the effect of an actuator, a *HTTP Post* request is sent to the *Machine Layer*. In the *Machine Layer*, upon receiving an actuator effect. the state is updated accordingly and the changes are published in the form of *HTTP Post* requests. *Machine* and *Sensing* services can receive these updates through the new *REST API* that has been added to each one, as can be seen in Figure 28. When an update is received, the respective service updates the *Fake Hardware Objects* included in that update.

5.2 Using the Matrix Project to test software

5.2.1 Requirement elicitation of use cases

To identify the use cases suitable for testing with integration tests using the Matrix Project solution, the first step involved gaining an understanding of the testing procedures conducted by the Quality Assurance team. This process generated a list of use cases that met the required criteria. The subsequent step involved evaluating the use cases to determine which ones would benefit most from automation, based on criteria such as the time required to perform the tests, the level of complexity involved, and frequency of use. This helped identify the most appropriate candidates for automation and determine the implementation order to follow.

5.2.2 Tools Exploration

In order to perform the integration tests, it was necessary to decide on a *framework* that supported *API* testing, since all the tests were going to involve communication using the *REST* servers of the different services. After a state-of-the-art analysis, there were two possible tools capable of performing the desired functions, **Playwright** and **Robot Framework**.

Playwright is an open-source *Node.js* library developed by Microsoft for browser automation and testing. It provides a high-level API for interacting with web pages and supports multiple browsers and programming languages. Playwright also includes built-in support for *API* testing, making it a versatile tool for both *frontend* and *backend* testing. In fact, this framework was already used by the *Quality Assurance* team to perform *frontend* smoke tests.

On the other hand, *Robot Framework* is a generic open-source test automation framework that uses a keyword-driven approach to test automation. It enables testing of both web and non-web applications and supports various operating systems and browsers. The *Robot Framework* also offers a comprehensive set of test libraries for various use cases, such as testing web applications, and *API* testing.

When comparing *Playwright* and Robot Framework for API testing, there are several factors to consider. One of the main advantages of Playwright is that it allows you to create tests using programming languages, which can make the development process faster and more flexible if the test developers are already familiar with one of the supported programming languages. In contrast, *Robot Framework* has its syntax and structure that may require more time to learn and adapt to for experienced programmers but can be easier to get into for non-developer team members.

Another factor to consider is the performance of both frameworks since the time to run software tests scales linearly with the number of tests, being very important to make sure the tests are as fast as possible. *Robot Framework*, with its keyword-driven approach, introduces an overhead in terms of performance to the tests. *Playwright*, on the other hand, is designed with efficiency and speed in mind.

With all these in mind, *Playwright* was the chosen framework to be used since there were members of the *QA* team already familiar with programming languages, being able to streamline the development process and take advantage of existing knowledge and expertise within the team. Additionally, since the team was already using Playwright to test the *frontend* of the company, it made sense to continue using a tool that was already familiar and well-established within the organization.

5.2.3 Test Execution Environment

All tests were constructed to be run in a "fresh" *Vision Software* environment, i.e., the state of the *Vision Software* after starting each of its services. Defining a concrete test environment was crucial to ensure the reliability and repeatability of the test results, since if the same test is not executed in the exact same conditions it can lead to different results.

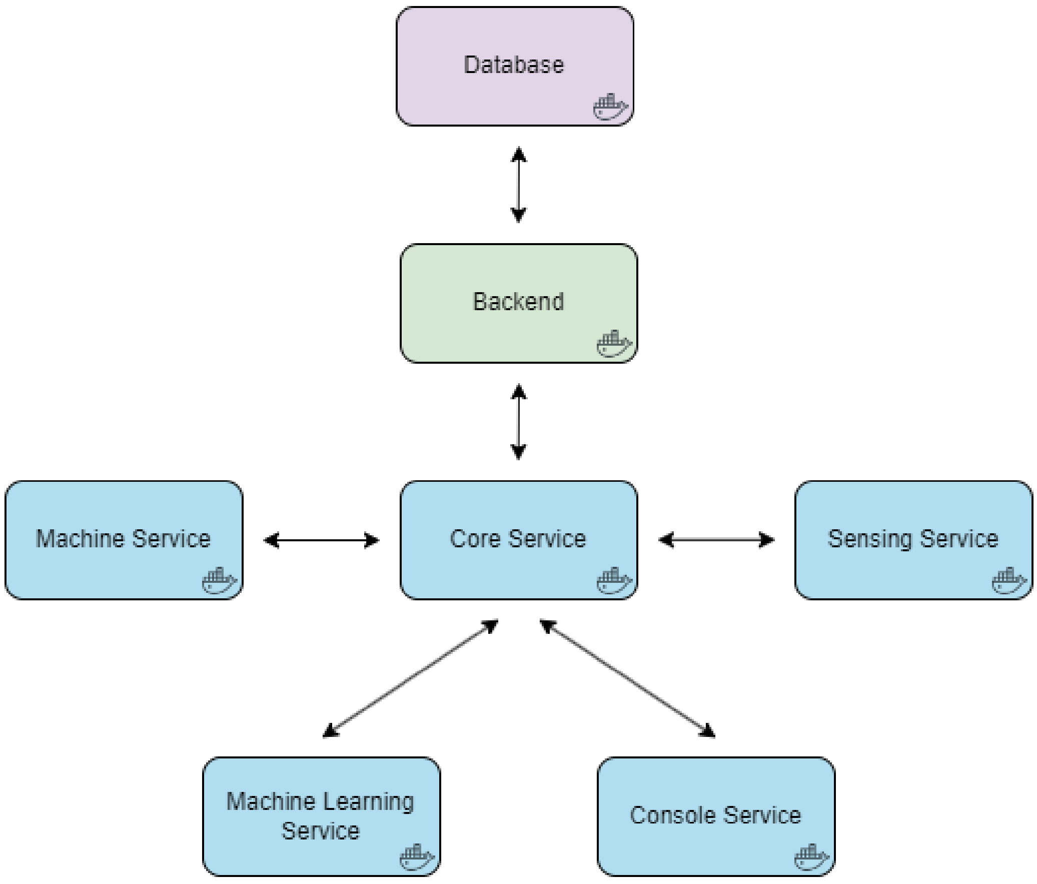

Figure 29: Test Pipeline

To ensure complete control over the test environment and achieve repeatability, a new environment was created for each test case. This involved spinning up all services of the *Vision Software* system, each running in a separate *Docker* container, and configuring the network to enable communication between them. *Docker Compose* was the tool of choice for automating this process, allowing for a consistent and reliable configuration of each test's environment. It made managing the setup and tear down of the containers much easier, ensuring that each test was run in an isolated and consistent environment.

Given that the *Vision Software* interacts with an external *Backend* component that, in turn, communicates with a *PostgreSQL* database, two separate docker containers were created to establish a controlled testing environment. Additionally, prior to each test, the containerized database was populated in a controlled manner.

This approach minimized the likelihood of errors caused by external factors and human intervention and ensured that previous tests did not affect future ones. As a result, repeatability was achieved and the results of each test case could be confidently compared.

5.2.4 Creation of a Test Pipeline

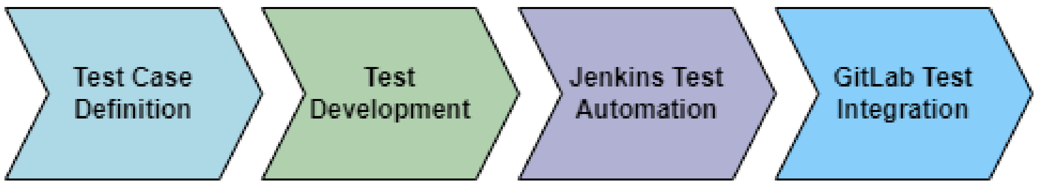

Figure 30: Test Pipeline

In order to optimize the entire test creation and implementation process, the pipeline depicted in Figure 30 was developed to ensure that all the necessary steps from the initial requirement elicitation to the final utilization of the test were systematically defined and rigorously followed. The pipeline streamlined the testing process by standardizing and automating each step, resulting in an efficient and reliable method for detecting and fixing defects in the *Vision Software* system.

The first step in the testing pipeline is the definition of a test case. This involves identifying and documenting all the requirements and expected behavior of the system under test, as referred in 5.2.1, and defining the specific test cases that will be used to validate that behavior. This step is critical to ensure that all aspects of the system are tested thoroughly and that the tests are aligned with the system's intended functionality.

The second step is the development of the test. In this phase, the actual test code is developed, based on the test cases defined in the previous step. The test code should be automated and well-structured, following best practices for test design, in order to ensure that the tests are reliable, maintainable, and scalable.

The third step is the Jenkins pipeline to gain confidence in the test. Jenkins is a popular open-source automation server that provides a wide range of plugins to automate different stages of the testing pipeline. In this step, the test code is integrated into a Jenkins pipeline that automates the execution of the tests on a regular basis. The pipeline provides feedback on the status of the tests and highlights any failures or regressions, enabling developers to quickly identify and fix any issues.

The final step is the incorporation of the test in a GitLab pipeline that only allows merges when tests pass. GitLab is a web-based Git repository manager that provides integrated **Continuous Integration/-Continuous Delivery** (**CI/CD**) pipelines. In this step, the test code is integrated into a GitLab pipeline that automatically runs the tests on every code change. The pipeline checks if the tests pass and only allows code changes to be merged if the tests pass successfully. This step ensures that the system remains stable and reliable, even as new features and changes are introduced.

5.3 Replicating Sensor Outputs

The *Matrix Project* has introduced the possibility of simulating real-life scenarios by recording data from *Vision Software's* hardware sensors and subsequently replaying it in a simulated environment. This section explores the process of recording and storing data from each hardware sensor used by the *Vision Software*, enabling its playback in the *Matrix Project*.

5.3.1 Recording Data

When a hardware sensor utilized by the *Vision Software* is queried, the obtained data is recorded and stored in a dedicated file. The implementation of this recording feature faced two primary challenges: minimizing modifications to the *Vision Software* source code and structuring the file format to store the required data with optimal efficiency, i.e., occupying as little space as possible.

To minimize disruptions to the source code, a decorator function was added to the abstract methods of the interface of each hardware sensor that queried data from the sensor. This approach enabled the activation or deactivation of sensor data recording at *runtime* while minimizing changes to the source code, thereby preserving the software's behavior.

Two potential approaches were considered for the storage of recorded data: storing all the data in a single file (single-file approach) or using separate files for each sensor (multi-file approach). The single-file approach could potentially experience slower data saving due to concurrent writes from multiple threads, necessitating the use of concurrency control mechanisms, which would incur additional overhead that would not exist in a multi-file approach. Conversely, the single-file scenario simplifies the process of replaying the recordings, as only one file needs to be handled. In the multi-file approach, a dedicated thread per file would be required for precise playback, since the events need to be replayed in the exact order and time they were recorded.

Considerations of data size also come into play. A single-file approach would result in a larger file size compared to using one file per sensor, as each data entry needs to include the identification of the sensor responsible. However, this difference becomes insignificant if a compression algorithm is applied to reduce storage requirements.

Scalability is an important factor to consider for future expansion. While both approaches require similar changes for recording data from new sensors, the playback part of the multi-file approach would involve creating a new thread for each new file. This approach lacks long-term scalability, as managing multiple threads may become unwieldy. In contrast, the single-file approach is more scalable as new sensors can be seamlessly integrated without requiring thread creation.

Having considered these factors, the chosen approach is the single-file approach, given its advantages in terms of simplicity, playback efficiency, and scalability.

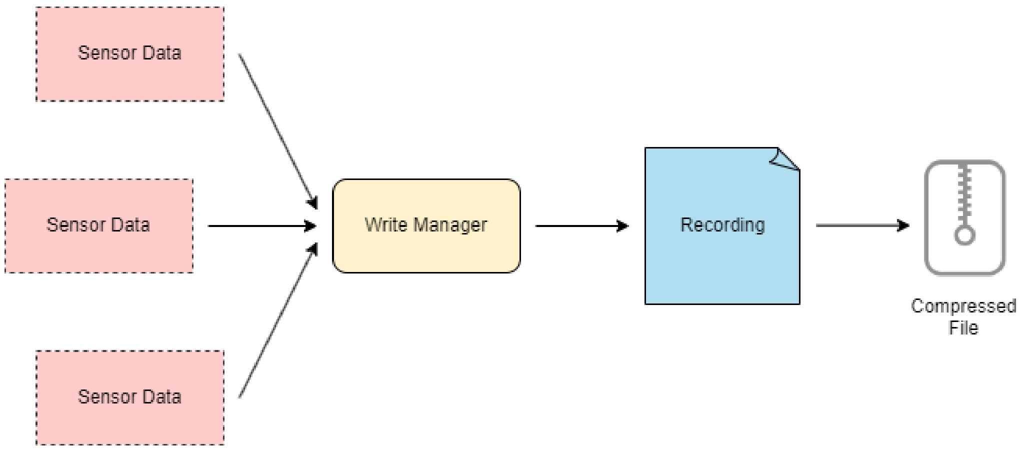

Figure 31: Creation of a Recording File

To ensure that writing to the recording file didn't take too much time, blocking waiting for the file to be free and compromising the normal functioning of the *Vision Software*, a *Write Manager* was created, as shown in Figure 31, that implemented the class ThreadPoolExecutor, where the new recording entries would be stored in memory until they could be safely written into the recording file, while not delaying the normal functioning of the *Vision Software*.

It is important to note that the cameras were the only sensor whose output was not an alphanumerical value, and for this reason, they were stored separately from the other sensors' outputs. However, in the recording file, for each picture taken, was created an entry that pointed to the correct directory where the respective picture was saved, allowing for playback of frames taken by cameras as well.

Recording File Format

The selection of an appropriate file format for the recording file was primarily driven by the objective of minimizing its size while ensuring compatibility for playback, as mentioned earlier.

Given the decision to store data from all sensors in a single file, it became necessary to uniquely identify each data entry by associating it with the sensor name and its corresponding data type. This requirement arose from the fact that multiple sensors' data is being stored in the same file and that a single sensor could provide information about multiple aspects. Therefore, to accurately distinguish and interpret the recorded data, a string identifying the sensor was added as well as a string identifying the data type of that sensor, in the cases where the corresponding sensor can return multiple information.

Fortunately, the impact on file size was negligible, as these constant strings could be easily compressed.

The recording file maintained a chronological order, obviating the need for sorting operations. Events were appended to the file as they occurred, as previously explained. Each event was accompanied by a designated timestamp indicating the precise moment of occurrence. This chronological ordering significantly facilitated the playback process, ensuring that subsequent events corresponded to subsequent lines in the recording file.

The inclusion of timestamps served not only to uphold the chronological sequence of events but also to accurately reflect the elapsed time between them, making it possible to reflect the sensor values changes exactly as they happened in the real execution that originated the recording.

To enhance human readability, each line in the recording file exclusively contained a single data entry, structured as follows:

```
SensorName1;DataType1;SensorValue1;Timestamp1
SensorName2;SensorValue2;Timestamp2
SensorName1;DataType2;SensorValue3;Timestamp3
```

In the above example, a simple recording file is depicted, with three data entries from two different sensors: *SensorName1* and *SensorName2*. *SensorName2* only has one possible data type, therefore, doesn't need to have its name expressed in the file where as *SensorName1* has at least two: *DataType1* and *DataType2*. Furthermore, the timestamps respect the following order:

$$timestamp1 \leq timestamp2 \leq timestamp3 \tag{5.2}$$

Compressing Recording File

The data starts by being recorded in a text file, which is subsequently compressed after the recording process has ended due to its significant size. In order to choose the best compression algorithm, a comprehensive study of compression techniques was made to choose which type of technique to use and then some experimental tests were conducted with real recording files to assess the best compression algorithm to use. It is important to note that there were no attempts to compress the recorded pictures as they were already in a very compact file format (*jpeg*) and their quality could not be decreased not to affect the outputs of the machine learning models used.

- **Lossless Compression Techniques:** Lossless compression techniques, such as Huffman coding, Run-Length Encoding (RLE), and Lempel-Ziv-Welch (LZW) compression, are commonly used to reduce file sizes without losing any information. Huffman coding assigns shorter codes to frequently occurring symbols, while RLE replaces consecutive occurrences of the same value with a count. LZW compression builds a dictionary of frequently occurring patterns. These techniques can effectively reduce the size of your text-based recording file without compromising data integrity.

- **Transform-based Compression Approaches:** Transform-based compression techniques, like Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and Wavelet Transform, analyze the frequency or spatial components of the data for compression. For example, DCT is widely used in image and audio compression algorithms. By converting sensor data into a transformed domain and discarding insignificant components, these techniques achieve higher compression ratios. Consider employing transform-based approaches if the inherent characteristics of your sensor data can be leveraged for efficient compression.

- **Hybrid Compression Approaches:** Hybrid compression approaches combine both lossless and lossy compression techniques, offering a trade-off between compression ratio and data fidelity. Lossy compression methods discard less crucial data, achieving higher compression ratios. By utilizing lossless compression to preserve important metadata or critical sections of the data and applying lossy compression to less significant parts, a balance can be struck. Consider a hybrid approach if your application can tolerate a slight loss of information for improved compression efficiency.

It was decided to go with a lossless compression technique for the text-based recording file because of two crucial aspects: data integrity and file reconstructability. Lossless compression techniques prioritize the preservation of data integrity, ensuring that no information is lost or altered during the compression and decompression process. This attribute is particularly critical for the sensors' data, where precision and accuracy hold the utmost significance. By employing a lossless algorithm, confidence can be placed in the compressed file's ability to faithfully represent the original data. Additionally, lossless compression techniques offer complete reconstructibility, enabling the precise restoration of the original sensor data, data types, values, and timestamps. This ensures that the compressed file can be reliably decompressed, providing access to the exact information as it was before compression.

To decide on which lossless algorithm to use, tests were conducted in order to evaluate different algorithms' compression/decompression speeds as well as their compression rate.

A *Python* script was created in order to determine which algorithm would be the most appropriate to use out of the following four lossless algorithms: *zlib*, *zstd*, *lz4* and *bz2*. In order for the tests performed to be as close as possible to reality, three real recording files were used with sizes *1 Mb*, *10 Mb*, and *100 Mb*. For each of the three files, 100 executions of all the listed algorithms were performed, measuring the time they took to compress and decompress the file and the compression rate achieved (how many times smaller the compressed file is than the original file). It is important to note that all tests were conducted with a previous cache warm-up to ensure all algorithms had an even starting point.

The analysis of the algorithms started by inspecting the compression times of the selected algorithms, as depicted in the graphs in the appendices and in tables Figure 1 and Figure 2. The results revealed a consistent order of speeds regardless of the input dimensions. The slowest algorithm was *bz2*, followed by *zlib*, *zstd*, and *lz4*, which was the fastest. While *zstd* and *lz4* demonstrate consistent results across multiple runs, *zlib* exhibited some variation, and *bz2* appeared to be highly irregular. Similar patterns were observed for the decompression times of the algorithms.

Conversely, when examining the compression rates of the algorithms shown in 3, the opposite trend emerged. The faster an algorithm was, the lower its compression rate. The descending order of compression times is as follows: *bz2*, *zlib*, *zstd*, and *lz4*.

Regarding the time complexity of the algorithms, tables 4 and 5 show how the necessary time to run each algorithm grew as the input size grew. Since the time increase is of the same order as the input size growth, all algorithms have a time complexity of *O(n)*, meaning that the execution time increases linearly with the size of the file being compressed or decompressed.

To aid in the algorithm selection process, a simple metric was devised to measure the time cost of each compression rate point for each algorithm. The metric is defined as follows:

$$metric = \frac{compression\_rate}{time} \tag{5.3}$$

Based on this metric, the *lz4* algorithm would have been chosen since it demonstrates the highest effectiveness across all tested file sizes. However, considering the specific use case of compressing and decompressing recording files, the achieved compression rate holds greater importance than the algorithm's execution time. This is due to the sporadic usage of the algorithm, while the recording files may be stored and utilized for an indefinite period.

Taking all factors into consideration, the *bz2* algorithm was selected for usage due to its remarkable ability to achieve high compression rates, despite having a significantly higher computational cost.

5.3.2 Replaying Data

The playback process, exemplified in figure 32, begins by decompressing the recording file and loading it into memory. Subsequently, a dedicated thread, identified as *Record Player* in 32, initiates the sequential reading of the file, processing each line (each event) individually. Upon reading a line, the thread retrieves the event timestamp associated with that particular line. It then enters a blocked state, patiently waiting until the designated time for that event arrives. Once the specified time is reached, the thread proceeds to update the corresponding simulated sensor value within the *Machine Layer*. This straightforward approach is practical due to the guarantee that the recording file is organized in ascending order based on the event timestamps. Consequently, the blocking mechanism ensures that the accurate timing of events is preserved, without the risk of missing any critical occurrences.

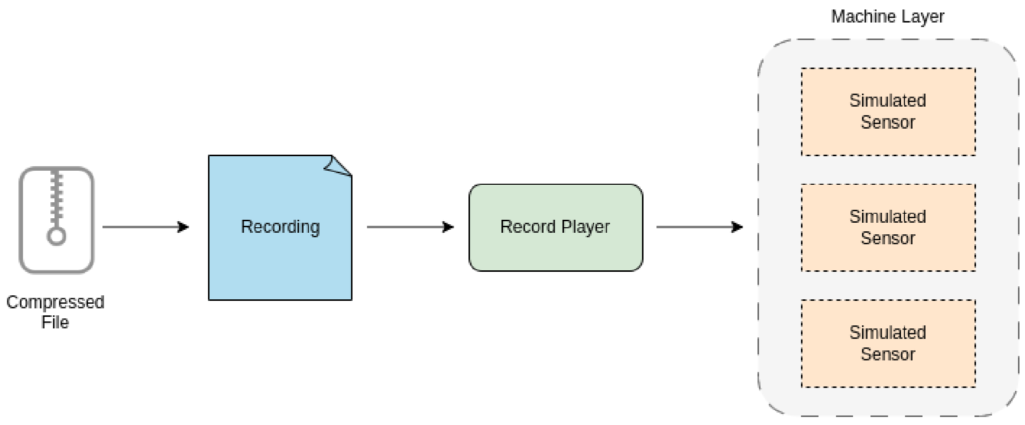

Figure 32: Playing Back a Recording File

5.3.3 Creating Artificial Data Sequences

In addition to recording and replaying real sensor data, another valuable application of the record and play feature described in this chapter is the ability to generate artificial sequences of events. By creating a file formatted as a genuine recording but with artificial entries, it becomes possible to test specific edge cases and simulate sensor behaviors that may not occur naturally.

This capability provides several advantages when it comes to developing and testing the *Vision Software*, enabling developers and testers to intentionally introduce scenarios that might be challenging or that would rarely occur in real-world situations. It would also make it easier to replicate specific errors that only occur in a given set of conditions. By crafting artificial data sequences, it becomes easier to assess the system's robustness, identify potential vulnerabilities, and validate its behavior in critical scenarios.

To facilitate the creation of these artificial data sequences a dedicated page was created on a web server dedicated to interacting with the *Machine Layer* explained in more detail in figure 5.5, where it was possible to create artificial sequences of data and consequently play them back.

5.4 Script to spin up Matrix Project

To facilitate the use of the *Matrix Project*, which involved knowledge of how to work with a distributed version control system (a type of software that helps multiple people collaborate on the same codebase), in this case, *GIT*, and with *Docker* and *Docker Compose*, it was created a *bash* script that abstracted the user from all of this. This way, to utilize the *Matrix Project*, a user did not need to have this technical knowledge. The script starts by letting the user choose which version of *Vision Software* to use, being capable of accepting a specific branch, commit hash or tag. After a version is chosen, it pulls it to the user's machine. The user can choose if it wants to run a specific *Vision Software* service or all the services by interacting with the terminal running the script. It can also consult the services' logs as well as enter their bash terminals.

5.5 Web server to interact with Matrix

As *Matrix's* success grew as an internal tool in *Smartex*, the Product team and the Sales team both expressed interest in leveraging its capabilities to achieve their respective goals.

With the aim of facilitating the process of interacting with *Matrix*, more precisely with the *Machine Layer*, a web server was created, providing a graphical interface for communication. It is important to note that, at the time of writing this dissertation, the web server is still under development and not yet fully finished. The web server communicates with the *Machine Layer* through its *REST* endpoints, being able to get the current state of each simulated hardware component as well as to update it.

The web server encompasses several key features, including:

- Start/Stop Button: This button initiates or halts the simulated machine, providing control over its operation;
- Status Page: Users can access this page to obtain detailed information about the state of each simulated hardware sensor;
- Camera Output Selection: This page allows users to choose the output of the simulated cameras. They can upload a roll of real pictures and replay it at will;
- Artificial Picture Sequence Generation: On this page, users can generate an artificial sequence of pictures, which can then be replayed.

The web server development held a significant value since it extended Matrix to teams that do not possess any programming experience, opening up new use cases for *Matrix* that were never thought of when planning this dissertation.

5.6 Evaluating Matrix Impact

This section evaluates the impact of the *Matrix Project*, the solution developed during this dissertation to aid in the development and testing of the Vision Software. The primary goal of the Matrix project is to enhance the speed of software development and simplify the Quality Assurance (QA) process by developing automated tests. This chapter provides a comprehensive analysis of Matrix's effectiveness, utilizing a combination of quantitative metrics, surveys and interviews, and case studies.

5.6.1 Evaluation Objectives

The evaluation aims to assess the impact of the Matrix tool on software development and quality assurance teams. The specific objectives include:

- Measure the improvement in productivity and efficiency of the software development process;
- Evaluate the effectiveness of the Matrix tool in enhancing the quality and reliability of the Vision Software;
- Understand the perceptions and experiences of team members who utilized the Matrix tool in their daily workflows.

5.6.2 Evaluation Methodology

The evaluation process employed a multi-faceted approach, combining quantitative metrics, surveys/interviews, and case studies. This holistic methodology ensured a comprehensive understanding of the impact generated by the Matrix tool in both the software development and software testing phases.

Quantitative Metrics

This study utilized quantitative metrics to assess different aspects of software development and quality assurance. Prior to the integration of Matrix into the development process, specific metrics were gathered. These metrics encompassed the time taken to develop a new feature, the proportion of automated test coverage, and the time saved as a result of automated testing. To conduct the data collection, historical project data was thoroughly analyzed. Subsequently, a comparative analysis was performed, contrasting the data collected before the implementation of Matrix with the data obtained after its integration into the development workflow.

Surveys and Interviews

Surveys and interviews were conducted to gather feedback from team members who utilized the Matrix tool. The survey questionnaire consisted of targeted questions related to the solution's impact on their work processes, productivity, and the overall quality of the Vision Software. Interviews were conducted with select team members to gain deeper insights into their experiences and capture both qualitative and quantitative feedback.

The sample for surveys and interviews included members from the software development and quality assurance teams, as well as project managers involved in the whole software life-cycle of the *Vision Software*.

Case Studies

Two case studies were conducted to assess the real-world application of the *Matrix Project*, one focused on the software development team and the other on the quality assurance team. Each case study involved utilizing the *Matrix Project* in a real use case and assessing its impact by comparing how the same problem would be solved without *Matrix*. The case study analysis focused on identifying challenges faced and comparing the resources, both time and people, necessary to solve the problem with and without the use of *Matrix*.

5.6.3 Impact on Vision Software Development

With the aim of understanding how the developed tool influenced the development team's efficiency and software delivery an evaluation was made that took into consideration the following key elements:

Analyzing quantitative metrics

One quantitative metric considered in this study was the time required to test new changes in the code of the *Vision Software*. By utilizing *Matrix*, the process involved running the code, waiting for the container images to be built, and then for the services to start, which typically took approximately 45 seconds. On the other hand, the conventional approach necessitated connecting to a remote machine, transferring all the changes to that machine, and executing the designated service. If multiple services were affected by the changes, this entire process had to be repeated for each service. In the most efficient scenario, where changes were limited to a single service and involved a single line alteration, the testing process using the conventional method took approximately 80 seconds to complete. However, in the worst-case scenario, encompassing changes in every service and every single line of code, including the mapping of cameras by the Sensing service, the testing process consumed around 350 seconds. As a result, *Matrix* exhibited a significant testing speed improvement of new code of at least 44% compared to the traditional approach, reaching up to approximately 87% in certain cases.

Creation of a survey

To gather valuable insights from the *Embedded Systems* team, a comprehensive survey was conducted to assess their usage of *Matrix* and its impact on their work. The main objectives were to understand the extent of *Matrix* adoption and its effect on development productivity. The survey results were highly encouraging, as every participating developer reported using *Matrix* on a daily basis, being used for an impressive average of **81%** of their development time. Furthermore, when asked about the impact of *Matrix* on their development speed, the average response indicated an increase of **80%**. Notably, developers expressed a high level of satisfaction with *Matrix*, giving it an average performance rating of **8.8/10**, highlighting its effectiveness and ease of use. Moreover, the feedback was unanimous in recognizing the potential benefits of *Matrix* for other teams as well. Specifically, teams such as *Sales*, *Product*, *Customer Success*, and *Machine Learning* were mentioned, indicating the tool's versatility and applicability across various domains. The survey also yielded valuable suggestions for future improvements. Developers provided extensive feedback on desired new features and recommendations to enhance the current version of *Matrix*.

These inputs will prove instrumental in the ongoing development and refinement of the tool. Overall, the feedback received from the survey is exceedingly positive, underscoring the tremendous success of this dissertation's work and the significant impact of *Matrix* on the productivity and efficiency of the *Embedded Systems* team.

Case study analysis

In order to evaluate the efficacy of the developed solution, referred to as *Matrix* in supporting the software development team, a comprehensive case study was conducted. The study focused on the complete implementation process of a feature called *Anomalies* within the *Vision Software* application, which leveraged the capabilities provided by *Matrix*. The assessment involved conducting interviews with the two software engineers responsible for this particular task.

The primary objective of the *Anomalies* feature was to modify the format of the classifications generated by machine learning models determining whether a fabric was defective or not. The aim was to align these defect measurements more closely with the clients' perceptions of what the defects were. Implementing this feature necessitated a significant refactor of the existing codebase of the *Vision Software*, entailing multiple alterations.

The workflow followed during the implementation of this feature consisted of the following steps:

- Implement of the required code changes within the*Vision Software*;
- Development of appropriate testing methodologies to validate the correct functionality of the implemented modifications;
- Execute the *Vision Software* code using *Matrix*;
- Thorough testing of the software using the devised testing procedures to ensure its overall quality.

Both interviewees agreed that the absence of *Matrix* would have substantially slowed down the implementation of the *Anomalies* feature, primarily due to the inclusion of additional steps in the workflow, as outlined below:

- Implement of the required code changes within the*Vision Software*;
- Development of appropriate testing methodologies to validate the correct functionality of the implemented modifications;
- Wait for one of the development machines to be free in order to execute the *Vision Software*;

- Transfer all the necessary files to the designated machine;
- Execute the *Vision Software* code on the machine;
- Thorough testing of the software using the devised testing procedures to ensure its overall quality.

Furthermore, *Matrix* made it easier to ensure the quality of the software being developed by facilitating the test edge case scenarios by enabling manipulation of the *RPMs* and by removing from the equation all the hardware-related errors. As one of the interviewed software stated, whenever an error related to the cameras would appear during the development, with matrix, he could be sure that it was due to some change made in the software and not because some camera cable was unplugged. For this reason, *Matrix* also made it possible to develop the *Anomalies* feature remotely whenever it was needed.

5.6.4 Impact on Vision Software Testing

With the aim of understanding how the developed tool improved the testing phase of the *Vision Software* for the Quality Assurance team, an evaluation was conducted that took into consideration the following key elements:

Quantitative metrics and data analysis:

One of the crucial quantitative metrics considered in this study was the number of automated tests developed for *Vision Software*. Prior to this dissertation, the absence of automated tests for *Vision Software* was attributed to the challenges posed by hardware dependencies. However, throughout the course of this dissertation, 37 automated tests were successfully developed. These tests comprehensively covered a wide array of use cases of the *Vision Software*, thereby significantly enhancing its test coverage.

Another important metric analyzed was the Quality Assurance team's time saved per test case by automating it. The time it took to manually perform each test case was measured and averaged out to be around 3.46 minutes, which also represents the average time saved by automating a test case since the automated tests took no time from the *QA* team.

The cumulative time-saving effect of all the tests developed amounts to 128 minutes for every time the tests pipeline is run. This means that these tests are now accessible in earlier phases of the software development, for example when merging a feature branch, and not only when a new version of *Vision Software* is being released, which solidifies the quality of the *Vision Software*.

Matrix has demonstrated a profound impact on the testing phase of the software life cycle by eliminating the need for manual integration tests in *Vision Software*. This breakthrough has opened up the possibility of conducting integration tests much more frequently and at earlier stages of the development process. As a result, the overall testing efficiency and effectiveness of *Vision Software* have significantly improved, leading to enhanced software quality and accelerated development timelines.

Case study analysis

To comprehensively evaluate the effectiveness of the *Matrix* platform in supporting the software quality assurance (QA) team, a detailed case study was conducted. The primary objective of this study was to analyze the automation process of a specific integration test that was previously performed manually on the *Vision Software* by the QA team. The focus was on understanding the evolution of this process and its impact. The specific test case under examination involved validating the behavior of the system when it was forced to halt, ensuring that an automatic restart occurred if no client input was received to confirm the stoppage within a predefined period, typically 15 minutes.

In the manual execution of this test case, a QA team member had to remain idle for a 15-minute duration to observe if the system restarted after being stopped. Any lapse in monitoring during this waiting period could potentially lead to a false test failure, because after restarting the machine will soon stop again. Moreover, this test repetition was obligatory for each new release of the *Vision Software*.

The automation initiative for this test case began by deconstructing the underlying steps involved in the manual test:

- Guarantee that the machine is up and running;
- Halting the machine;
- Confirming the machine has come to a stop;
- Waiting for a duration of 15 minutes;
- Validating the machine is up and running again.

Once the necessary test steps were identified, each step was meticulously examined to determine the *Vision Software* services implicated and the relevant endpoints that could facilitate the execution of these actions within the test. In the context of this test case, it was concluded that all *Vision Software* services were integral, and specific endpoints were marked for tasks like machine status verification and halting the machine.

Subsequently, an automated test scenario was meticulously designed and implemented utilizing the *Playwright* framework. Rigorous testing of this automated scenario was conducted iteratively to validate its reliability and effectiveness.

The successful automation of this intricate test case, empowered by the capabilities of the *Matrix* platform, notably liberated the QA team from an onerous and time-consuming responsibility. This led to a substantial time-saving of approximately 15 minutes for each testing cycle of a new version of the *Vision Software*.

Chapter 6

Conclusion

This chapter summarizes the work accomplished in this dissertation indicating future possible directions in which it can further be developed.

6.1 Outcomes

In conclusion, this master's dissertation has been driven by four key objectives, meticulously identified in the abstract, and has delivered comprehensive solutions for each of them:

- **Simulation Software Development**: The primary aim was to craft simulation software capable of faithfully replicating the intricate hardware components of the *Vision Software*. This endeavor led to the creation of the *Matrix* solution, a sophisticated framework that established a scalable and high-performance software layer. *Matrix* effectively severed the ties between the *Vision Software* and its hardware dependencies, enabling a flexible and efficient development environment.
- **Automated Test Pipeline**: A second pivotal goal was the establishment of an automated test pipeline leveraging the *Matrix* simulation software. This pipeline introduced a new paradigm in Smartex's Quality Assurance processes for the *Vision Software*, enhancing its reliability and performance through a comprehensive suite of meticulously documented software tests.
- **Reliability and Effectiveness Testing**: The third objective focused on subjecting the *Matrix* hardware simulation solution to rigorous reliability and effectiveness tests. Through meticulous evaluation, *Matrix's* viability and robustness were conclusively affirmed, validating its pivotal role as an indispensable cornerstone within Smartex's dynamic software ecosystem.
- **Impact Assessment on Software Life Cycle**: The final key point was to assess the impact of this work on Smartex's software life cycle, particularly for the *Embedded Systems* and *Quality Assurance* teams. The successful implementation of *Matrix* not only expedited the development pace

for the Embedded Software team but also established a foundation for elevated quality assurance through software integration tests.

Moreover, the far-reaching success of this dissertation's work has extended beyond the realms of the *Embedded Software* and *Quality Assurance* teams. Its influence has permeated into diverse corners of Smartex, leading to the initiation of an unexpected yet promising project: the development of a web server designed to serve as an interface for *Matrix*. This development will provide a visual gateway to *Matrix*, catering to the requirements of teams such as *Product* and *Sales*. The widespread adoption of *Matrix* within the organization underscores its efficacy and robustness, positioning it as an indispensable and resoundingly successful solution.

6.2 Future work

Although all of the objectives established for this work were achieved and the final application even accounts for some extra functionalities that were not initially planned, there is still room for improvement and expansion, which could enhance the current build.

6.2.1 Web Server

Further developing the web server that acts as an interface for *Matrix* presents a significant opportunity for future work. By providing an intuitive and user-friendly interface, the web server will allow other teams within *Smartex*, such as *Product* and *Sales*, to benefit from the functionalities and capabilities of *Matrix*.

6.2.2 Test Environment Improvements

To create a more consistent and reliable test environment, two key improvements can be implemented:

- Create a container for the database: Introduce a dedicated container within *Matrix* for the database. Before each test, this container can be populated with a controlled set of information, ensuring a predictable and controlled database state for each test. This approach will facilitate accurate and reliable test results, ultimately improving the effectiveness of the testing process;

- Create a container for the backend: Incorporate a separate container within *Matrix* specifically for the backend. By isolating a specific version of the backend, potential compatibility issues can be eliminated, further enhancing the reliability of the tests. This improvement will remove another layer of uncertainty and contribute to more dependable and conclusive test outcomes.

By pursuing these future enhancements, *Matrix* can continue to evolve, addressing potential limitations and expanding its functionalities.

Bibliography

Simulating and testing targetlink code. URL https://www.dspace.com/en/pub/home/medien/videos/productvideos/video-tl-simulation.cfm#175_31979.

Handbook of Simulation. John Wiley Sons, Inc., Aug 1998. doi: 10.1002/9780470172445. URL http://dx.doi.org/10.1002/9780470172445.

Ahmed A. Al Rowaei, Arnold H. Buss, and Stephen Lieberman. The effects of time advance mechanism on simple agent behaviors in combat simulations. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pages 2426–2437, 2011. doi: 10.1109/WSC.2011.6147952.

Altexsoft. Quality assurance - quality control and testing: The basics of software quality management, 2016. URL https://www.altexsoft.com/media/2016/10/Quality-Assurance-Quality-Control-and-Testing-%E2%80%94-the-Basics-of-Software-Quality-Management.pdf.

James Anderson and Rong Fu. *Methods for Modelling and Simulation Studies*. 02 2016.

M. Bacic. On hardware-in-the-loop simulation. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 3194–3198, 2005. doi: 10.1109/CDC.2005.1582653.

Susmita Bandyopadhyay and Ranjan Bhattacharya. *Discrete and continuous simulation*. CRC Press, Boca Raton, FL, June 2014.

International Software Testing Qualifications Board. Certified tester foundation level syllabus, 2018. URL https://castb.org/wp-content/uploads/2020/01/ISTQB-CTFL_Syllabus_2018_V3.1.pdf.

Andrei Borshchev and Alexei Filippov. From system dynamics and discrete event to practical agent based modeling: Reasons, techniques, tools. 2004.

Darcy Bullock, Brian Johnson, Richard B. Wells, Michael Kyte, and Zhen Li. Hardware-in-the-loop simulation. *Transportation Research Part C: Emerging Technologies*, 12(1):73–89, 2004. ISSN 0968-090X. doi: https://doi.org/10.1016/j.trc.2002.10.002. URL https://www.sciencedirect.com/science/article/pii/S0968090X03000792.

Arnold Buss and Ahmed Al Rowaei. A comparison of the accuracy of discrete event and discrete time. In *Proceedings of the 2010 Winter Simulation Conference*, pages 1468–1477, 2010. doi: 10.1109/WSC.2010.5679045.

Jeanne M. Carey and Kelly Rossler. The how when why of high fidelity simulation. 2022. URL https://www.ncbi.nlm.nih.gov/books/NBK559313/#_article-63901_s4_.

Hweedo Chang. Power electronics control design amp; testing in the 21st century. URL https://info.typhoon-hil.com/blog/power-electronics-control-design-and-testing-in-the-21st-century.

Xiang Chen, Meranda Salem, Tuhin Das, and Xiaoqun Chen. Real time software-in-the-loop simulation for control performance validation. *Simulation*, 84(8-9):457–471, August 2008.

Alberto Clavijo-Rodriguez, Victor Alonso-Eugenio, Santiago Zazo, and Ivan Perez-Alvarez. Software-in-loop simulation of an underwater wireless sensor network for monitoring seawater quality: Parameter selection and performance validation. *Sensors*, 21(3), 2021. ISSN 1424-8220. doi: 10.3390/s21030966. URL https://www.mdpi.com/1424-8220/21/3/966.

Emerson Electric Co. Understanding and applying simulation fidelity to the digital twin. https://www.emerson.com/documents/automation/white-paper-understanding-applying-simulation-fidelity-to-digital-twin-mimic-en-507
pdf, 2018.

Stephanie Demers, Praveen Gopalakrishnan, and Latha Kant. A generic solution to software-in-the-loop. In *MILCOM 2007 - IEEE Military Communications Conference*, pages 1–6, 2007. doi: 10.1109/MILCOM.2007.4455268.

Gerald D. Everett and Raymond McLeod. *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Pr, 2007. ISBN 047179371X.

H. Freeman. Software testing. *IEEE Instrumentation Measurement Magazine*, 5(3):48–50, 2002. doi: 10.1109/MIM.2002.1028373.

Software Testing Genius and Yogindernath Gupta. Difference between change related software testing like confirmation testing amp; regression testing, Aug 2018. URL https://www.softwaretestinggenius.com/difference-between-change-related-software-testing-like-confirmation-testing-regres

D. Graham, R. Black, and E. van Veenendaal. *Foundations of Software Testing ISTQB Certification, 4th edition*. Cengage Learning, 2021. ISBN 9780357884157. URL https://books.google.pt/books?id=m0wxEAAAQBAJ.

Richard Holzer, Patrick Wüchner, and Hermann De Meer. Modeling of self-organizing systems: An overview. *Electronic Communications of the EASST*, Volume 27: Selbstorganisierende:kontextsensitive verteilte Systeme 2010, 2010. doi: 10.14279/TUJ.ECEASST.27.385. URL http://journal.ub.tu-berlin.de/eceasst/article/view/385.

ISTQB. Worldwide software testing practices report, 2018. URL https://www.turkishtestingboard.org/files/ISTQB-Worldwide-Software-Testing-Practices-Report-2017-18.pdf.

Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and A. Ahmad. Software testing techniques: A literature review. *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182, 2016.

Stefan Jörg, Jan Tully, and Alin Albu-Schäffer. The hardware abstraction layer — supporting control design by tackling the complexity of humanoid robot hardware. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6427–6433, 2014. doi: 10.1109/ICRA.2014.6907808.

Averill M. Law. *Simulation Modeling & Analysis*. McGraw-Hill, New York, NY, USA, 5 edition, 2015.

Manuele Leonelli. Simulation and modelling to understand change. https://bookdown.org/manuele_leonelli/SimBook/, 2021. Lecture Notes.

H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301, 1990. doi: 10.1109/ICSM.1990.131377.

Qi Liu, Marcio A. Silva, Michael R. Hines, and Dilma Da Silva. Hardware-in-the-loop simulation for automated benchmarking of cloud infrastructures. In *Proceedings of the 2012 Winter Simulation Conference (WSC)*, pages 1–12, 2012. doi: 10.1109/WSC.2012.6465036.

Charles Macal and Michael North. Introductory tutorial: Agent-based modeling and simulation. In *Proceedings of the Winter Simulation Conference 2014*, pages 6–20, 2014. doi: 10.1109/WSC.2014.7019874.

Charles M. Macal and Michael J. North. Agent-based modeling and simulation: Abms examples. In *2008 Winter Simulation Conference*, pages 101–112, 2008. doi: 10.1109/WSC.2008.4736060.

Timofei Popkov Maxim Garifullin, Andrei Borshchev. Using anylogic and agent-based approach to model consumer market. https://www.anylogic.com/upload/iblock/920/920920e8cabbdb2d3d58433618dcad60.pdf. Company Notes.

I. McGregor. The relationship between simulation and emulation. In *Proceedings of the Winter Simulation Conference*, volume 2, pages 1683–1688 vol.2, 2002. doi: 10.1109/WSC.2002.1166451.

Onur Özgün and Yaman Barlas. Discrete vs. continuous simulation: When does it matter? 1. 2009.

Zedong Peng, Xuanyi Lin, and Nan Niu. Unit tests of scientific software: A study on swmm. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, pages 413–427, Cham, 2020. Springer International Publishing. ISBN 978-3-030-50436-6.

Victor Platon and Andreea Constantinescu. Monte carlo method in risk analysis for investment projects. *Procedia Economics and Finance*, 15:393–400, 2014. ISSN 2212-5671. doi: https://doi.org/10.1016/S2212-5671(14)00463-8. URL https://www.sciencedirect.com/science/article/pii/S2212567114004638. Emerging Markets Queries in Finance and Business (EMQ 2013).

Duncan Robertson. Agent-based models to manage the complex. 01 2005.

Christopher T. Collins Roy W. Miller. Acceptance testing, 2005. URL http://www.dsc.ufcg.edu.br/~jacques/cursos/map/recursos/Testing05.pdf.

W. Earl Sasser. Book reviews : J. w. schmidt and r. e. taylor, simulation and analysis of industrial systems, homewood, illinois: Irwin, 1970. *Simulation &amp Games*, 1(4):440–448, December 1970. doi: 10.1177/104687817000100408. URL https://doi.org/10.1177/104687817000100408.

Robert Shannon and James D. Johannes. Systems simulation: The art and science. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(10):723–724, 1976. doi: 10.1109/TSMC.1976.4309432.

RM Sharma. Quantitative analysis of automation and manual testing. *International journal of engineering and innovative technology*, 4(1), 2014.

Masoud Soleymani Shishvan and Jörg Benndorf. Operational decision support for material management in continuous mining systems: From simulation concept to practical full-scale implementations. *Minerals*, 7(7), 2017. ISSN 2075-163X. doi: 10.3390/min7070116. URL https://www.mdpi.com/2075-163X/7/7/116.

Jiangjun Tang, George Leu, and Hussein A. Abbass. *Discrete Time Simulation*, pages 143–155. 2020. doi: 10.1002/9781119527183.ch8.

Gu Tian-yang, Shi Yin-Sheng, and Fang You-yuan. Research on software security testing. *International Journal of Computer and Information Engineering*, 4(9):1446–1450, 2010.

Mubarak Albarka Umar. Comprehensive study of software testing: Categories, levels, techniques, and types. 6 2020. doi: 10.36227/techrxiv.12578714.v2. URL https://www.techrxiv.org/articles/preprint/A_Study_of_Software_Testing_Categories_Levels_Techniques_and_Types/12578714.

Anna M. Wichansky. Usability testing in 2000 and beyond. *Ergonomics*, 43(7):998–1006, 2000. doi: 10.1080/001401300409170. URL https://doi.org/10.1080/001401300409170. PMID: 10929833.

Sungjoo Yoo and Ahmed A. Jerraya. *Introduction to Hardware Abstraction Layers for SoC*, page 179–186. Kluwer Academic Publishers, Boston, 2005. ISBN 9781402075285.

Part III
Appendices

Appendix A
Details of results

A.1 Compression Algorithms Study

A.1.1 Tables

File Size (Mb)	Compression Average Times (sec)			
	zlib	**lz4**	**zstd**	**bz2**
1	0.002011	0.000125	0.000415	0.008886
10	0.018105	0.001276	0.005638	0.098901
100	0.191174	0.014616	0.049172	1.038264

Table 1: Compression Times Comparison

File Size (Mb)	Decompression Average Times (sec)			
	zlib	**lz4**	**zstd**	**bz2**
1	0.000281	0.000038	0.000106	0.002089
10	0.001990	0.000354	0.001287	0.026129
100	0.021302	0.004155	0.011498	0.248564

Table 2: Decompression Times Comparison

File Size (Mb)	Compression Average Rates			
	zlib	**lz4**	**zstd**	**bz2**
1	4.65	2.61	4.46	6.77
10	4.77	2.66	4.39	7.50
1000	4.78	2.66	4.32	7.58

Table 3: Compression Rate Comparison

File Size Increase	**Time Increase Factor**			
	zlib	**lz4**	**zstd**	**bz2**
1 $\rightarrow$ 10	9.00	10.21	13.59	11.13
10 $\rightarrow$ 100	10.56	11.45	8.72	10.50

Table 4: Compression Time Complexity Analysis

File Size Increase	**Time Increase Factor**			
	zlib	**lz4**	**zstd**	**bz2**
1 $\rightarrow$ 10	7.08	9.32	12.14	12.51
10 $\rightarrow$ 100	10.70	11.74	8.93	9.51

Table 5: Decompression Time Complexity Analysis

A.1.2 Graphics

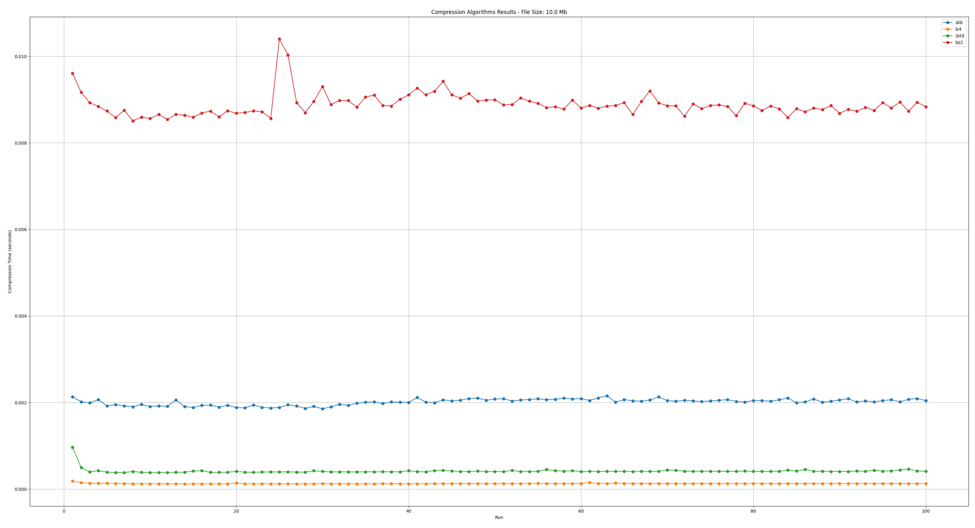

Figure 33: Compression times for a 10 Mb file

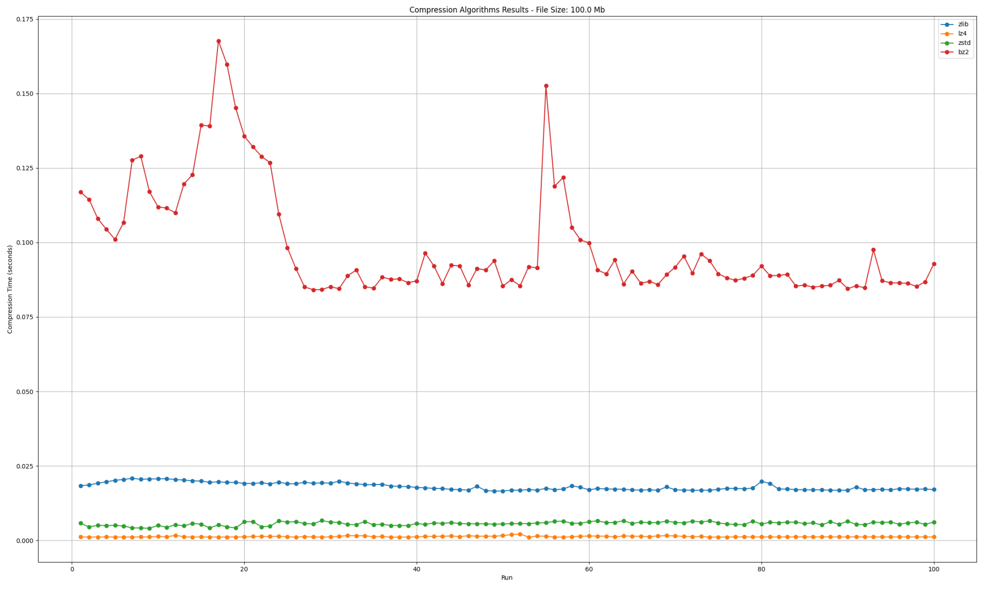

Figure 34: Compression times for a 100 Mb file

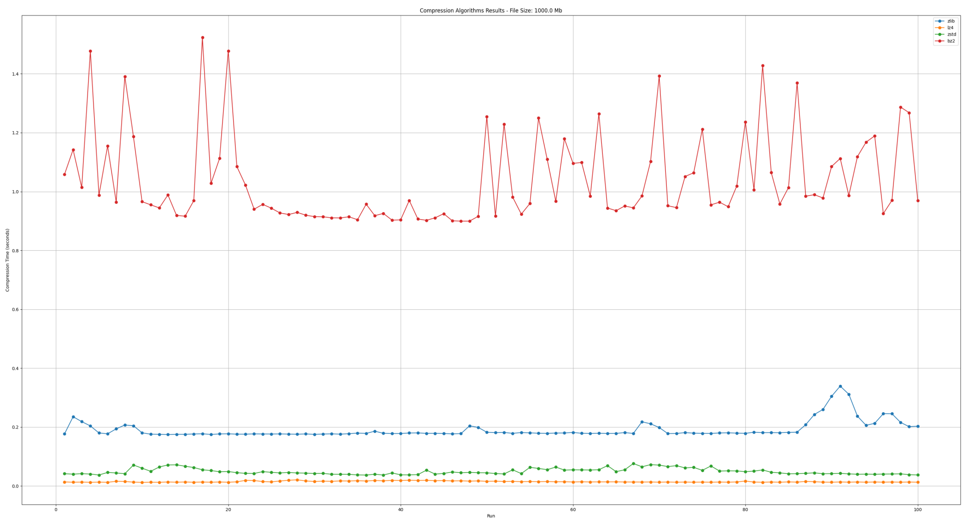

Figure 35: Compression times for a 1000 Mb file

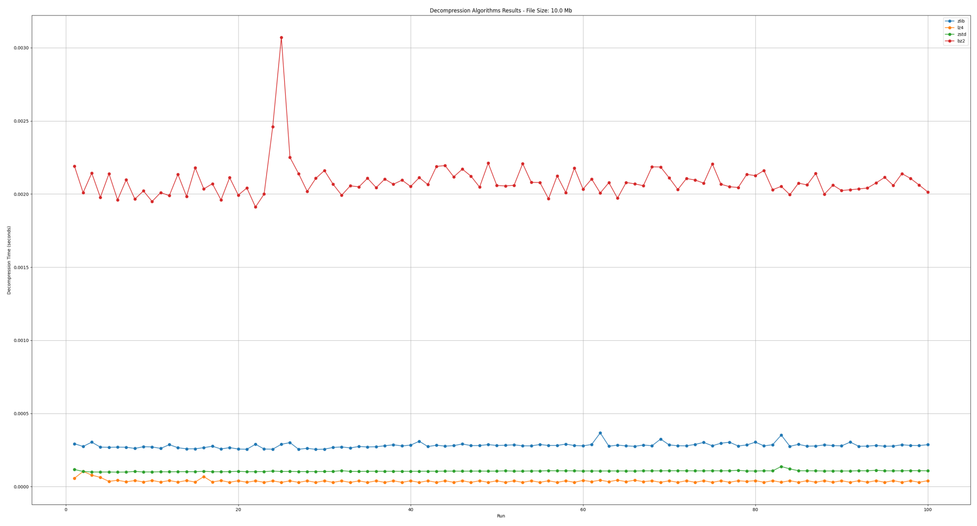

Figure 36: Decompression times for a 10 Mb file

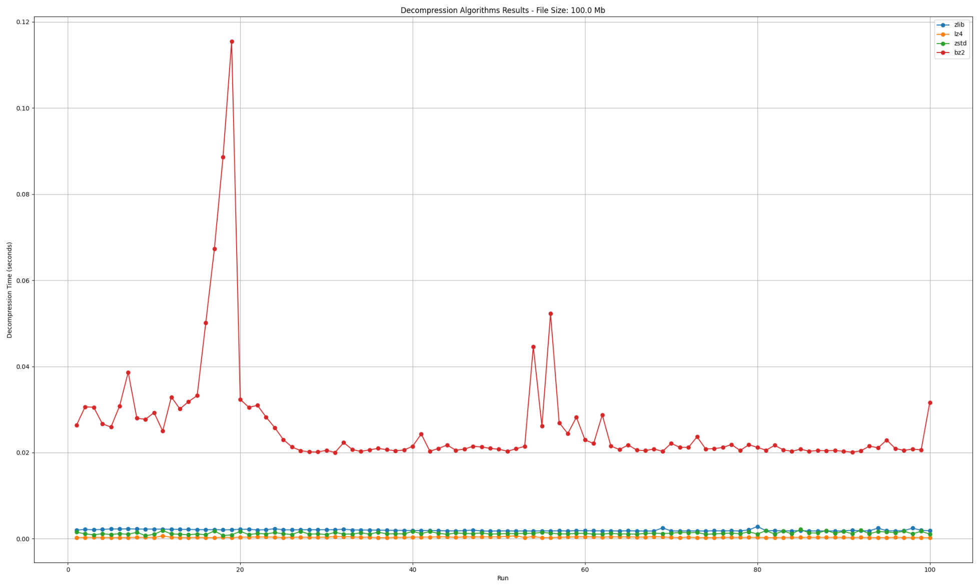

Figure 37: Decompression times for a 100 Mb file

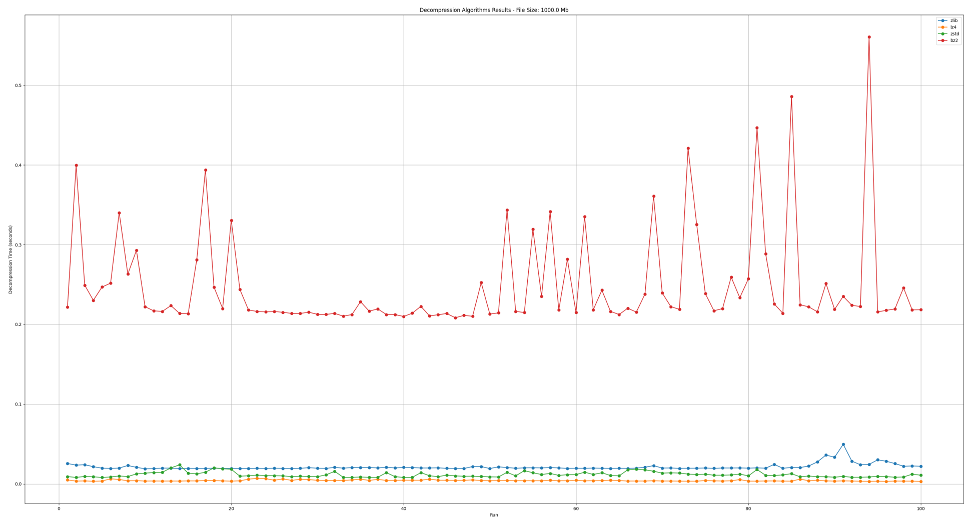

Figure 38: Decompression times for a 1000 Mb file

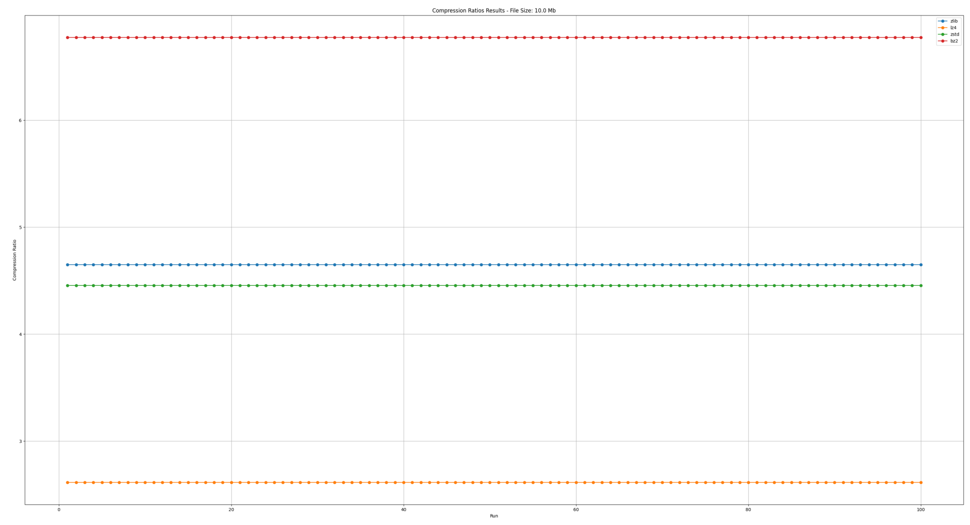

Figure 39: Compression rates for a 10 Mb file

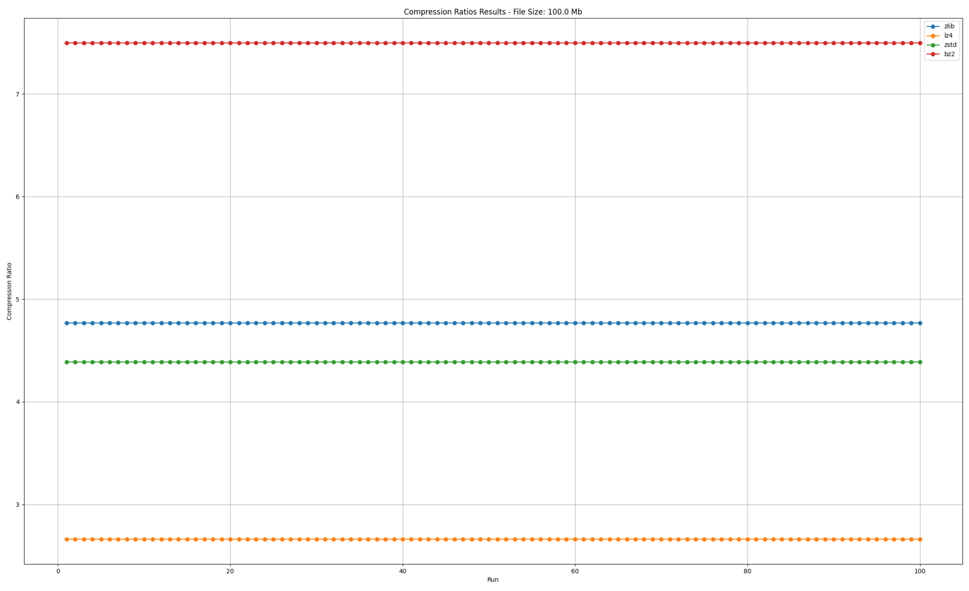

Figure 40: Compression rates for a 100 Mb file

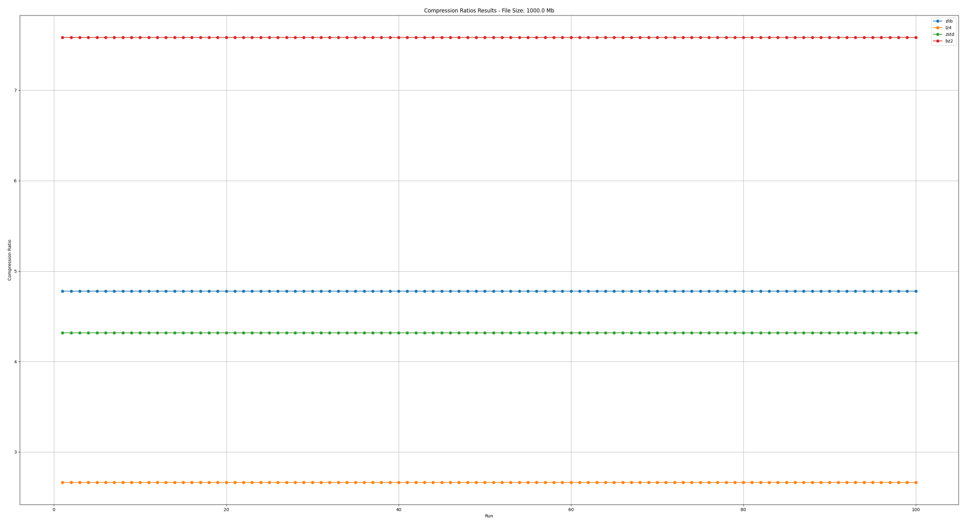

Figure 41: Compression rates for a 1000 Mb file