
**Research Project**

*by*

**Hana BEN SAID**
**Ines HANZAL**

# Test Engineering and Validation Using the VORTEX Drone Simulator

Defended on 26/05/2026 in front of the committee composed of:

**Supervisors:**

F. BOUQUET
F. DADEAU
D. TABARY

**Jury:**

P.-C. HEAM
A. GIORGETTI
D. TABARY
J.-F. COUCHOT

# Contents

# List of Figures

# List of Tables

# Acknowledgments

We would like to express our deepest and most sincere gratitude to our project supervisor, Ms. Dorine Tabary, for her constant availability, patience, and invaluable help throughout every stage of this project. Her continuous guidance and support were essential in keeping us on track and helping us overcome the challenges we faced along the way.
We are also incredibly grateful grateful to Mr. Frédéric Bouquet and Mr. Fabrice Dadeau for introducing this research area and laying the fundamental groundwork for our project topic.
Our warm thanks also go to Antoine Chevrot and Louis Lefèvre for introducing us to the VORTEX tool and walking us through its environment, as well as to Éléa Jacquin for taking the time to explain the entire testing process to us.
In addition, we sincerely thank the members of our examination committee, Mr. Pierre-Cyrille Héam, Mr. Alain Giorgetti, and Mr. Jean-François Couchot, for taking the time to review our work and providing their critical, constructive feedback on our findings.

# Introduction

As part of our first year of the Master's program in Computer Science at the Université Marie et Louis Pasteur, we conducted a research project on the automation of software testing for robotic systems using Large Language Models (LLMs). Software testing plays a crucial role in ensuring the reliability and correctness of robotic applications; however, manually designing unit tests remains time-consuming and challenging, particularly for systems involving sensors and dynamic behaviors.
Recent advances in LLMs have opened new possibilities for automatically generating executable tests from textual specifications and behavioral models. Through this work, we aimed to answer the following research question: What level of automation can LLMs provide for the generation of unit tests in robotic systems? More specifically, we investigated the impact of prompt engineering, modeling abstraction, and LLM selection on the quality of the generated tests.
To evaluate these aspects, several experiments were conducted using the Webots simulator and the CUnit framework on different robotic controllers, including the DJI Mavic 2 Pro, Crazyflie, and e-puck robots.
This manuscript is organized as follows.
**Chapter 1** presents the state of the art related to software testing, Model-Based Testing, and LLM-based test generation approaches. **Chapter 2** introduces the experimental environment and protocol. **Chapter 3** presents the obtained results and a comparative analysis between human-generated and AI-generated tests. Finally, the conclusion summarizes the main findings, limitations, and future research perspectives.

# Chapter 1

# State of the art

This chapter presents the theoretical background related to software testing in section 1.1 and AI-generated testing in section 1.2. First, the fundamental concepts of software testing are introduced, including the structure and importance of test cases. Then, the use of artificial intelligence and large language models in software testing is discussed. Finally, the chapter presents the research field in section 1.3 and several related works relevant to this study.

## 1.1 Software testing

In order to establish the foundations of this work, this section introduces the main concepts of software testing in section 1.1.1, its importance in software engineering in section 1.1.2, and the structure of the testing process in section 1.1.3.

### 1.1.1 Definition of software testing

Software testing is the process of evaluating and verifying that a software system or application meets specified requirements and functions correctly. It involves executing the software with the intent of identifying errors, gaps, or missing requirements in contrast to actual requirements.
According to IEEE, software testing is defined as "the process of executing a system or component to evaluate one or more properties of interest." [IEEE Computer Society, 1990] This IEEE standard provides one of the most widely accepted definitions of software testing and establishes common terminology used in software engineering research and industry.
Testing can be performed manually or automatically and is a critical phase in the software development lifecycle (SDLC).

### 1.1.2 Importance of software testing

Software testing plays a crucial role in ensuring that applications are of high quality and operate reliably. It helps find and fix problems early, which saves time and money in the long run. When software is tested properly, it meets both functional and technical requirements, leading to better performance and reliability. Testing also prevents serious bugs that could be expensive or dangerous. Overall, software testing improves the product's quality, protects the company from major issues, and ensures that users are satisfied with the final product. As software systems become more complex, testing methods are also changing, with a focus on automation and smarter testing techniques to keep up with these advancements.

### 1.1.3 Structure of a test process

A well-structured test case generally includes several key elements. These notably include a unique identifier, a test scenario or objective, prerequisites (initial state), input data, a sequence of steps to execute, and the expected result(s). For example, in a standard test case format, we include: Test Case ID, Test Scenario, Test Steps, Prerequisites, Test Data, and Expected Results. In practice, these components are often presented as clear sections in the test documentation.
The following list outlines this typical structure [ISTQB, 2024]:

- **Test Case ID** (unique identifier)
- **Test Scenario/Objective** (what is being verified)
- **Prerequisites** (configuration or data needed before the test)
- **Execution Steps** (sequence of actions or events to perform)
- **Test Data** (input values used)
- **Expected Results** (expected behavior or output of the software)

This organization facilitates traceability and the repeatability of tests while ensuring that every aspect of the test is accurately documented.

## 1.2 AI-Generated testing

As software systems become increasingly complex, artificial intelligence has emerged as a promising solution for automating software testing activities. This section presents AI-generated testing in subsection 1.2.1 and discusses its advantages and limitations in subsection 1.2.2.

### 1.2.1 Test generation using artificial intelligence

As software systems become more complex, artificial intelligence (AI) is increasingly used in software testing. AI-generated testing refers to the use of machine learning techniques and Large Language Models (LLMs) to automatically generate test cases and test data. Traditionally, test creation required significant manual effort, but recent AI approaches can learn patterns from source code and existing tests, enabling the automatic generation of relevant and high-quality test cases with minimal human input.
Deep learning models, particularly transformer-based architectures, have shown strong capabilities in understanding code and generating tests. Models such as GPT-5 [OpenAI, 2024] and Claude 3.5 [Anthropic, 2024] can produce test inputs comparable to those written by developers. A study by Yoshimoto et al. (2026) found that 16.4% of test additions in real projects came from AI tools. Additionally, AI-generated tests often include more assertions while maintaining similar code coverage to human-written tests, helping generate unit tests, explore edge cases, and improve overall testing efficiency [Yoshimoto et al., 2026].

### 1.2.2 Advantages and Challenges of AI-Based Testing

AI-based testing offers several advantages that make it increasingly attractive in modern software engineering.
One of the primary benefits is automation. AI techniques can significantly reduce the need for manual test creation, saving both time and resources. Additionally, AI models can analyze large volumes of code and generate tests that improve coverage and detect hidden defects.
Another key advantage is adaptability. Unlike traditional rule-based methods, AI systems can learn from data and adapt to different programming languages, frameworks, and testing scenarios. This makes them particularly suitable for dynamic and evolving software systems [Magnitia IT, 2024].
AI-based testing also enhances the following:

- Scalability, by handling large and complex systems
- Efficiency, through faster test generation and execution
- Quality, by identifying edge cases that may be overlooked by human testers

However, challenges remain. A significant limitation is the lack of reliability and explainability; AI-generated tests may not always be correct or meaningful, and understanding how they were produced can be difficult. Implementing AI testing solutions often requires substantial initial investments in infrastructure and team training. Additionally, AI models depend heavily on large volumes of high-quality data, as the quality of results is closely tied to the training data. While AI offers increased efficiency and better coverage, it cannot yet match human creativity, particularly for exploratory or usability testing, which still necessitates human input. Lastly, the use of AI raises security and ethical concerns, such as the risk of data leaks or biases in generated behavior. In summary, while AI provides significant advantages, it must be guided by human expertise to address its limitations.

## 1.3 Research field and related work

To better position our work within the existing literature, this section presents the main research areas related to automated software testing and summarizes several relevant studies.
In order to analyze the selected research articles consistently, an analysis protocol was provided and systematically applied to each selected research paper. This methodology allowed a structured analysis of the related works and provided a global overview of current approaches and research directions in software testing and AI-assisted test generation. The detailed analysis protocol is presented in appendix A.

### 1.3.1 Overview of the Research Field

Software testing research has evolved toward more automated and intelligent approaches for generating and evaluating test cases. Many studies focus on reducing manual effort while improving software reliability, coverage, and fault detection.

#### Articles Focused on Test Generation

Several studies focus on automatic test generation using requirements, formal models, and iterative refinement techniques.

The survey by [Yang et al., 2025] Zhenzhen Yang and colleagues presents an overview of Requirements-Based Test Generation (RBTG). The authors analyzed 267 studies related to automatic test generation from software requirements. Their work classifies different requirement types, generation approaches, tools, and evaluation methods. The study also highlights the growing use of NLP and AI techniques for processing natural language requirements.
Another important approach is Model-Based Testing (MBT). The work of [Castillos et al., 2014] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand proposes generating tests from UML/OCL models and property patterns. Their method improves test coverage and helps detect incorrect system behaviors through automated-based verification.
The RATE approach [Bombarda et al., 2023] proposed by Andrea Bombarda and collaborators combines automatic test generation with iterative model refinement. The method progressively improves the testing model using execution results and code coverage analysis, leading to better fault detection and testing quality.

#### Articles focused on test evaluation

Other studies focus on improving test evaluation, execution analysis, and industrial testing processes. The work of [Mohacsi and Felderer, 2021] Stefan Mohacsi and Michael Felderer introduces AI techniques into an industrial testing tool to help prioritize tests, estimate execution time, and support testing decisions. Their approach shows how AI can improve testing management and efficiency.
Another important contribution is presented by [Kokorin, 2025] Ilya Kokorin and colleagues, who propose a model-based testing approach for distributed systems using TLA+ formal models. Their method generates exhaustive test suites capable of covering many execution scenarios and improving fault detection in complex systems.
Overall, these studies show that software testing is evolving toward more automated, model-based, and AI-assisted techniques to improve software quality and testing efficiency.

### 1.3.2 Large Language Models in software testing

Large Language Models (LLMs) are advanced artificial intelligence models trained on massive amounts of text and source code data. They are designed to understand, generate, and predict natural language and programming languages. According to the survey by Wayne Xin Zhao [Zhao et al., 2023] and collaborators, LLMs are large-scale pre-trained models based on transformer-based architectures that demonstrate remarkable abilities in language understanding and generation. The study also explains that increasing model size and training data significantly improves model performance and enables new emergent capabilities.
LLMs are mainly based on transformer architectures trained on very large datasets containing text and source code. During training, the model learns patterns and contextual relationships by predicting the next word or token in a sequence. One of the key components of LLMs is the attention mechanism, which helps the model identify relationships between different parts of the input. This allows LLMs to process code, requirements, and software documentation efficiently while generating coherent and context-aware outputs[Zhao et al., 2023].
Recent LLMs such as GPT-5 [OpenAI, 2024], Claude 3.5 [Anthropic, 2024], and Gemini 3.5 [DeepMind, 2024] have shown strong capabilities in software engineering tasks, including code generation, debugging, documentation, and software testing. Unlike traditional machine learning methods, LLMs can understand contextual relationships between code and natural language, making them highly suitable for automated test generation and test engineering tasks.

### 1.3.3 Bilan

The reviewed studies show that large language models are increasingly integrated into software testing and test engineering activities. Their capabilities in natural language understanding and source-code generation enable them to support several testing tasks, including automatic unit-test generation, assertion generation, and behavioral validation.

The literature also highlights the potential of AI-assisted approaches for improving test coverage, supporting bug detection, and reducing manual testing effort. In addition, several works combine AI techniques with model-based or requirements-based testing strategies in order to improve test quality and automation.

Overall, the analyzed studies demonstrate that AI-assisted testing is becoming an important research direction in software engineering. However, most works also emphasize that human verification remains necessary to ensure the correctness, reliability, and behavioral validity of generated tests.

# Chapter 2

This chapter presents the experimental methodology in section 2.1 and the different environments used throughout this work for the generation and evaluation of unit tests for robotic controllers developed in Webots. It first describes the hardware configuration, software tools in subsection 2.1.1, simulation platform in subsection 2.1.2, testing workflow in subsection 2.1.3, and large language model environment in subsection 2.1.4 adopted during the experiments. The chapter then introduces the contribution of Large Language Models (LLMs) in section 2.2 for automatic unit-test generation and explains the experimental protocol used to evaluate the impact of prompt design, modeling abstraction, and LLM selection on the quality of the generated tests.

## 2.1 Experimental Environment

This section presents the experimental environment used for the development and evaluation of the generated unit tests. It includes the hardware configuration, software tools in subsection 2.1.1, simulation platform in subsection 2.1.2, testing process in subsection 2.1.3 and the llm used in subsection 2.1.4 throughout the experiments.

### 2.1.1 Hardware and Software Environment

The experiments were conducted using two personal computers presented in Table 2.1. Both systems were based on x64 architecture and provided sufficient resources for robotic simulation, compilation, and unit test execution.

| **Component** | **Computer 1** | **Computer 2** |
|---|---|---|
| Processor | Intel Core i5 8th Generation | Intel Core i5 |
| RAM | 16 GB | 8 GB |
| Operating System | Windows 11 | Windows 11 |
| Architecture | x64 | x64 |

Table 2.1: Hardware used in the experiments

These hardware configurations allowed the execution of Webots simulations [Cyberbotics, 2024], the compilation of C programs, and the automated execution of CUnit test suites under stable conditions. However, the MSYS2 [MSYS2, 2024] environment and CUnit configuration were fully operational on only one computer, which was mainly used for compiling and executing the generated test suites.
On the software side, several tools and technologies were used throughout the experiments. Webots was used as the main robotic simulation platform for validating the behavior of the robotic controllers.

The controllers and generated tests were implemented in the C programming language due to its compatibility with embedded robotic systems.
The experiments were conducted on Windows using the MSYS2 environment, which provides Unix-like development tools and supports GCC compilation on Windows platforms. The GCC compiler was used to compile both the robotic controllers and the generated CUnit test suites. Unit testing and behavioral validation were performed using the CUnit framework, while Visual Studio Code was used as the main development environment for editing, debugging, and managing the project files. In addition, all generated test suites, execution results, and project resources were maintained in a GitLab repository [Ihanzal, 2026] to facilitate version management, collaboration, and experimental reproducibility.

### 2.1.2 Simulation Environment

The experimental study relies heavily on robotic simulation in order to evaluate controller behavior and validate generated unit tests under controlled conditions. The Webots simulator was selected as the main simulation platform because it is open-source, freely accessible, and provides realistic physics simulation capabilities comparable to commercial robotic simulators such as Vortex. In addition, Webots offers a large library of prebuilt robots, sensors, environments, and controllers, making it particularly suitable for robotics experimentation and academic research.
Another important advantage of simulation is safety. Robotic experiments may involve unstable movements, collisions, or incorrect controller behaviors that could damage real hardware. By using simulation, experiments can be performed safely without any physical risks or equipment damage. Simulation also enables reproducible experimental scenarios, allowing the same tests to be executed multiple times under identical conditions, which is essential for fair comparison and evaluation.
Three robotic systems were studied during this work: the **DJI Mavic 2 Pro** drone, the **Crazyflie** drone, and the **e-puck** mobile robot. These systems were selected because they represent different robotic behaviors and different levels of complexity.
For each robotic system, a human analysis phase was first performed. This phase included the definition of the specification document, system requirements, behavioral properties, activity diagrams, and execution paths. These elements were then used as specifications for manual testing and AI-generated test. To ensure transparency and completeness, the comprehensive specification documents, detailed behavioral properties, and operational requirements for all three selected robotic projects are provided in full in the Appendix B.
Although the three robotic systems were analyzed and tested, the experimental evaluation of AI-generated tests focused mainly on the e-puck controller due to its simpler architecture and easier behavioral evaluation.

### 2.1.3 Testing Environment

The testing process was performed using the MSYS2 terminal environment under Windows. After generating the CUnit test files, the tests were compiled and executed manually through command-line instructions. The workflow started by opening the MSYS2 terminal and navigating to the directory containing the test files using the ‘cd‘ command.
Figure 2.1 illustrates the experimental environment used during the testing phase. The left side presents the Webots simulation environment used to execute and validate the robotic controllers, while the right side shows an example of the CUnit execution results generated after running the compiled test suites.

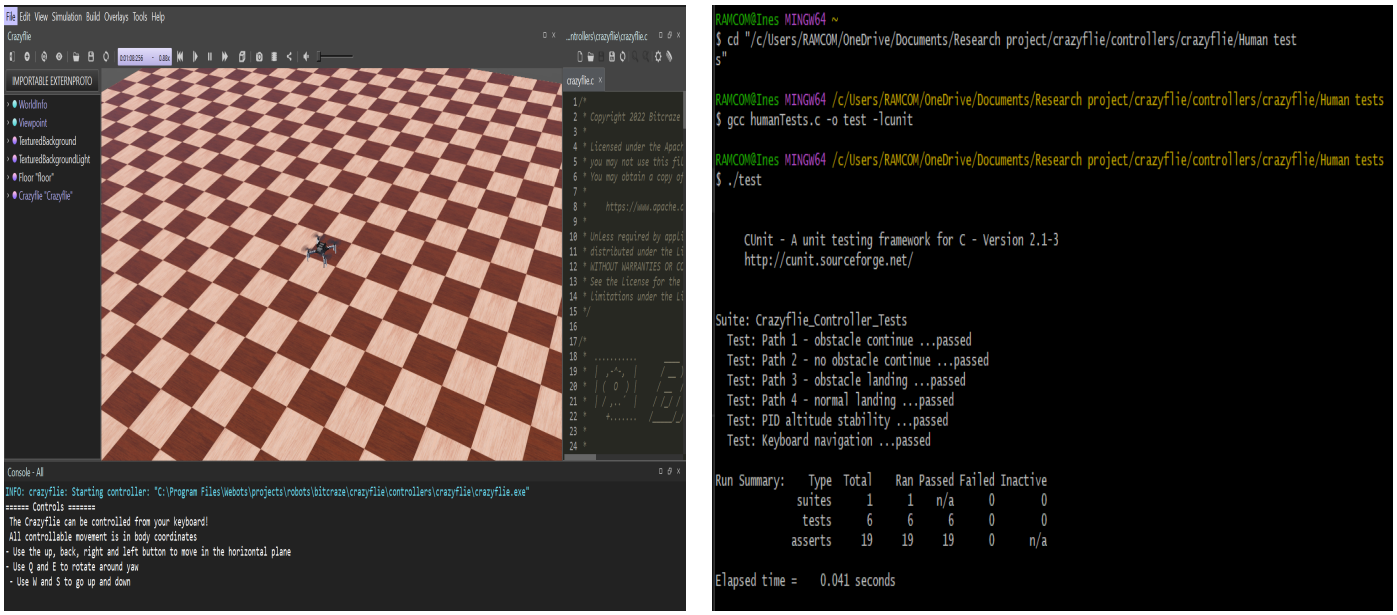


Figure 2.1: Webot interface and testing results

First, the ‘cd‘ command was used to access the folder containing the controller and unit-test source files. Then, the GCC compiler compiled the CUnit test file and generated an executable program named test. Finally, the executable file was launched using the ./test command in order to run all unit tests automatically.
During execution, the CUnit framework generated detailed reports containing information about executed test suites, passed tests, failed tests, assertions, and execution time. Assertions such as CU_ASSERT and CU_ASSERT_DOUBLE_EQUAL were used to verify controller behaviors, sensor processing, stabilization mechanisms, movement logic, and obstacle avoidance behavior. This testing workflow enabled consistent validation of both human-generated and AI-generated unit tests throughout the experiments.

### 2.1.4 LLM Environment

Large Language Models (LLMs) were used in this work for automatic unit-test generation. Two models were selected during the experiments 2.2 wich they are: DeepSeek-V3 and Grok-4. These models were chosen because of their code-generation capabilities, reasoning abilities, and public accessibility.
**DeepSeek-V3** is a Large Language Model designed for advanced natural-language processing and code-generation tasks. The model is capable of understanding programming structures and generating source code in multiple programming languages, including C. It can also assist in software-engineering tasks such as debugging, code completion, and test generation [DeepSeek-AI, 2024].
**Grok-4** is a Large Language Model developed for conversational reasoning and software-related tasks. The model is designed to interpret natural-language instructions and generate structured outputs, including source code and logical assertions. Its reasoning capabilities make it suitable for software-testing and code-generation applications [xAI, 2025].
Both models support natural-language prompting, allowing users to describe software behaviors, requirements, and testing objectives in textual form. Their ability to process both natural language and programming concepts makes them suitable for AI-assisted software-testing tasks and automatic

unit-test generation.

## 2.2 Contributions of Large Language Models (AI-generated Tests)

After presenting the studied robotic systems and their behavioral specifications, the next part focuses on the contribution of Large Language Models (LLMs) for automatic unit-test generation. The following experiments evaluate the ability of AI-based approaches to generate executable and meaningful CUnit tests for Webots controllers under different experimental conditions.

### Experimental Protocol

This study evaluates the impact of prompt design, modeling abstraction, and the choice of Large Language Model (LLM) on the quality of automatically generated unit tests. The experiments focus on generating C unit tests for a robot controller developed using Webots. The system under test is a simple e-puck robot controller selected for its limited complexity and ease of evaluation. All generated unit tests are written in C and executed using the CUnit framework. Three experiments are conducted during this study. The first experiment evaluates the effect of prompt design by comparing a simple prompt with an elaborated prompt. The second experiment evaluates the influence of the selected LLM by comparing two different models using the same prompt. The third experiment investigates the effect of modeling abstraction by comparing controller properties with expected behaviors against execution paths extracted from the activity diagram. For each condition in all experiments, ten test suites are generated.
Two large language models are used during the experiments: DeepSeek-V3 [DeepSeek-AI, 2024] and Grok-4 [xAI, 2025]. DeepSeek-V3 is used for the prompt-design experiment, both DeepSeek-V3 and Grok-4 are used for the LLM comparison experiment, and Grok-4 is used for the modeling-abstraction experiment.
The evaluation method is based on the execution results produced by the CUnit framework. A test suite represents a collection of unit tests executed together. Passed tests correspond to successfully executed test cases, while failed tests represent tests that do not satisfy the expected behavior. Assertions are logical conditions used to validate program behavior during execution. Passed assertions indicate correct validation results, whereas failed assertions indicate incorrect or unexpected behavior. Execution time represents the total time required to execute the generated tests. A global performance score is computed using the following formula:

$$Score = \frac{Passed\ Tests + Passed\ Assertions}{Total\ Tests + Total\ Assertions} \tag{2.1}$$

For each experimental condition, average values are computed for passed tests, failed tests, passed assertions, execution time, and performance score. These averages are then compared using tables and bar charts in order to analyze the impact of prompts, LLMs, and modeling strategies on generated unit-test quality. To ensure fairness and consistency, the same controller, execution environment, evaluation method, and experimental conditions are maintained throughout all experiments.

# Chapter 3

This chapter presents the experimental results obtained from both human-generated tests presented in section 3.1 and AI-generated tests presented in section 3.2 CUnit tests for Webots robotic controllers. It first introduces the results of manually developed test suites, followed by the evaluation of automatically generated tests using large language models (LLMs). The chapter also analyzes the impact of prompt design, LLM selection, and modeling abstraction on the quality of the generated tests and concludes with a comparative analysis between human-generated and AI-generated approaches.

## 3.1 Results of Human-Generated Tests

This section presents the execution results obtained from the manually designed CUnit tests for the studied robotic controllers, namely, the DJI Mavic 2 Pro drone 3.1.1, the Crazyflie drone 3.1.2, and the e-puck mobile robot 3.1.3. The presented tables summarize the number of executed test suites, passed and failed tests, validated assertions, and inactive elements. These metrics provide an overview of the correctness, stability, and reliability of the implemented controller behaviors under different execution scenarios.

### 3.1.1 DJI Mavic Results

The manually generated CUnit tests for the DJI Mavic 2 Pro controller were designed based on the functional requirements, formal properties, and execution paths extracted from the activity diagram. The detailed specifications, properties, and generated tests, as well as the activity diagram and its extracted execution paths, can be found in the corresponding subsection of appendix B.1.

| Type | Total | Ran | Passed | Failed | Inactive |
|---|---|---|---|---|---|
| Suites | 1 | 1 | n/a | 0 | 0 |
| Tests | 7 | 7 | 7 | 0 | 0 |
| Asserts | 9 | 9 | 9 | 0 | n/a |

Table 3.1: CUnit Results for DJI Mavic 2 Pro

The obtained results in table 3.1 show that all generated tests and assertions were executed successfully without any failures. The controller behaviors related to stabilization, movement, altitude adjustment, and safety constraints were validated correctly. No inconsistencies or unexpected behaviors were detected during execution, indicating that the implemented logic satisfies the tested execution paths. The execution time was approximately 0.000 seconds, confirming that the tests remain lightweight and focused on logical verification rather than full simulation execution.

**Conclusion:** The DJI Mavic controller demonstrated stable and reliable behavior for all tested scenarios, with complete validation of all tests and assertions.

### 3.1.2 Crazyflie Results

The manually generated CUnit tests for the Crazyflie controller were developed from the system requirements, behavioral properties, and execution scenarios identified from the activity diagram. The complete specifications, formal properties, generated tests, activity diagram, and extracted execution paths are available in the corresponding appendix subsection B.2.

| Type | Total | Ran | Passed | Failed | Inactive |
|---|---|---|---|---|---|
| Suites | 1 | 1 | n/a | 0 | 0 |
| Tests | 7 | 7 | 7 | 0 | 0 |
| Asserts | 14 | 14 | 14 | 0 | n/a |

Table 3.2: CUnit Results for Crazyflie

The Crazyflie controller achieved successful execution for all defined tests and assertions. The results shown in table 3.2 confirm the correctness of stabilization, movement control, obstacle handling, and landing behaviors implemented in the controller logic. The absence of failed tests demonstrates consistent execution across the evaluated scenarios. The execution time was approximately 0.000 seconds, showing that the tests are efficient and suitable for rapid behavioral verification.
**Conclusion:** The Crazyflie tests validated the expected controller behaviors successfully and confirmed the robustness of the implemented stabilization and navigation logic.

### 3.1.3 e-puck Results

The manually generated CUnit tests for the e-puck controller were developed from the system requirements, behavioral properties, and execution scenarios identified from the activity diagram. The full set of specifications, formal properties, generated tests, activity diagram, and extracted execution paths are documented in the corresponding appendix subsection B.3.

| Type | Total | Ran | Passed | Failed | Inactive |
|---|---|---|---|---|---|
| Suites | 1 | 1 | n/a | 0 | 0 |
| Tests | 5 | 5 | 3 | 2 | 0 |
| Asserts | 7 | 7 | 5 | 2 | n/a |

Table 3.3: CUnit Results for e-puck

The execution results in table 3.3 for the e-puck controller indicate that most behaviors were validated successfully, while some execution paths produced failures. The successful tests confirmed the correctness of normal navigation and movement behaviors in scenarios without obstacles. However, failures were observed in obstacle-related execution paths, indicating inconsistencies in avoidance behavior or speed adjustment logic under certain conditions. Despite these failures, all tests were executed correctly within a short execution time of approximately 0.017 seconds, confirming the efficiency of the testing process.
**Conclusion:** The e-puck controller demonstrated correct behavior in standard navigation scenarios, but the failed obstacle-related tests revealed areas requiring additional validation and refinement.

### 3.1.4 General Conclusion

The manually generated CUnit tests provided an effective validation of the implemented robotic controllers. The DJI Mavic 3.1.1 and Crazyflie 3.1.2 controllers achieved complete success across all tested scenarios, while the e-puck 3.1.3 controller revealed some limitations in obstacle-handling behaviors. Overall, these results demonstrate the usefulness of manually designed unit tests for validating robotic controller logic and identifying potential behavioral inconsistencies before applying automated test generation approaches.

## 3.2 Results of AI-generated Tests

This section presents and analyzes the results of the different experimentations conducted on AI-generated unit tests. The experiments focus on evaluating the impact of prompt engineering in the subsection 3.2.1, LLM selection in subsection 3.2.2, and modeling abstraction on the quality in subsection 3.2.3, reliability, and execution behavior of automatically generated CUnit test suites for Webots robotic controllers.

### 3.2.1 Prompt Design Results

This experiment compares a simple prompt and an elaborated prompt in order to evaluate their impact on the quality of automatically generated C unit tests.

**Prompt 1: Simple Prompt**

Generate C unit tests for the following Webots e-puck robot controller using the CUnit framework.

The following table 3.4 presents the execution results obtained using the simple prompt, including the number of generated tests, passed and failed assertions, and execution time.

| Suite | Total Tests | Passed Tests | Failed Tests | Passed Assertions | Failed Assertions | Time (s) | Fix Required |
|---|---|---|---|---|---|---|---|
| 1 | 9 | 9 | 0 | 68 | 0 | 0.036 | Yes |
| 2 | 10 | 7 | 3 | 74 | 4 | 0.062 | Yes |
| 3 | 12 | 11 | 1 | 118 | 1 | 0.057 | Yes |
| 4 | 12 | 10 | 2 | 168 | 7 | 0.131 | Yes |
| 5 | 12 | 11 | 1 | 118 | 2 | 0.059 | Yes |
| 6 | 12 | 11 | 1 | 90 | 1 | 0.061 | Yes |
| 7 | 13 | 11 | 2 | 158 | 7 | 0.158 | Yes |
| 8 | 16 | 14 | 2 | 91 | 2 | 0.093 | Yes |
| 9 | 17 | 15 | 2 | 109 | 2 | 0.141 | Yes |
| 10 | 15 | 13 | 2 | 87 | 2 | 0.080 | Yes |
| **Average** | **12.8** | **11.2** | **1.6** | **108.1** | **2.8** | **0.088** | — |

Table 3.4: Results obtained using Prompt 1

**Prompt 2: Elaborated Prompt**

Generate comprehensive C unit tests for the following Webots e-puck robot controller using the CUnit framework.
The generated tests should:

- Verify the main behaviors of the controller
- Include meaningful assertions

- Cover normal and edge cases
- Be executable and compilable
- Follow good unit testing practices
- Use clear and organized test functions

Return only valid C test code compatible with CUnit.

The following table 3.5 presents the execution results obtained using the elaborated prompt, including the number of generated tests, passed and failed assertions, and execution time.

| Suite | Total Tests | Passed Tests | Failed Tests | Passed Assertions | Failed Assertions | Time(s) | Fix Required |
|---|---|---|---|---|---|---|---|
| 1 | 15 | 15 | 0 | 80 | 0 | 0.065 | No |
| 2 | 18 | 18 | 0 | 103 | 0 | 0.096 | No |
| 3 | 23 | 23 | 0 | 129 | 0 | 0.096 | No |
| 4 | 22 | 22 | 0 | 133 | 0 | 0.098 | No |
| 5 | 32 | 32 | 0 | 133 | 0 | 0.118 | No |
| 6 | 32 | 32 | 0 | 132 | 0 | 0.121 | No |
| 7 | 31 | 31 | 0 | 129 | 0 | 0.052 | No |
| 8 | 33 | 33 | 0 | 138 | 0 | 0.055 | No |
| 9 | 32 | 32 | 0 | 110 | 0 | 0.040 | No |
| 10 | 28 | 28 | 0 | 134 | 0 | 0.045 | No |
| **Average** | **26.6** | **26.6** | **0** | **122.1** | **0** | **0.079** | - |

Table 3.5: Results obtained using Prompt 2

### Graphical Comparison

Figure 3.1 compares the average passed tests, average passed assertions, performance score, and manual fix rate obtained for both prompts.

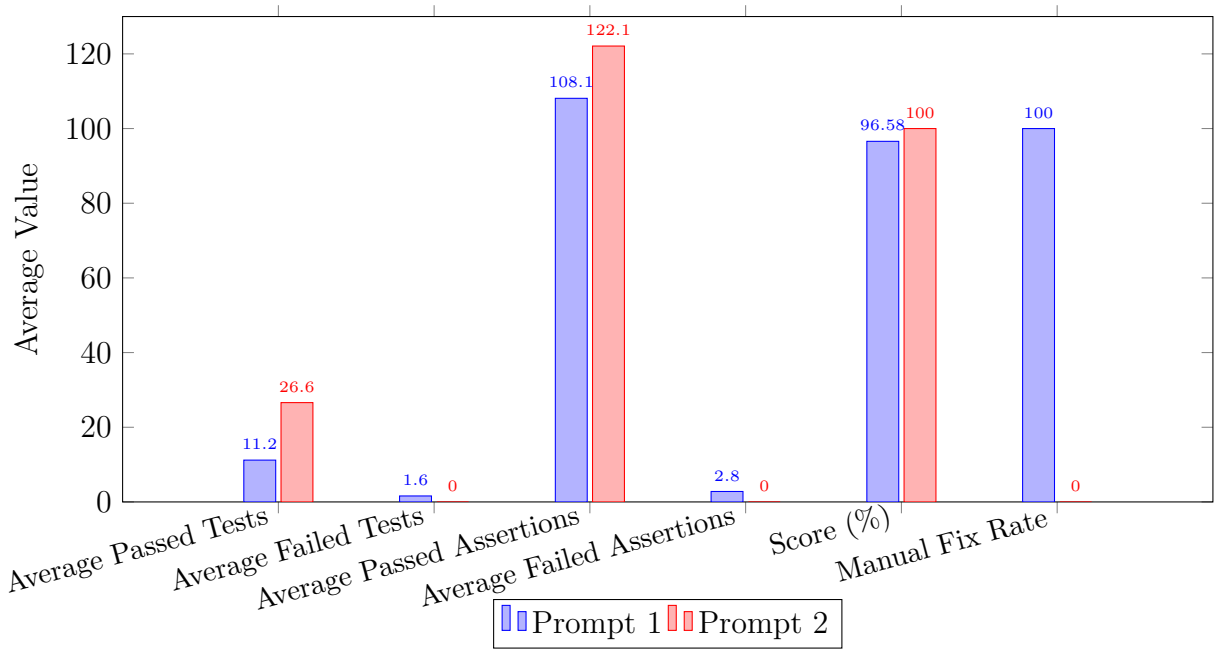


Figure 3.1: Comparison between Prompt 1 and Prompt 2

#### Analysis and Discussion

The results show a clear improvement in unit test quality when using the elaborated prompt. Prompt 1 frequently produced incomplete or inconsistent test suites requiring manual correction before execution. Several generated tests also contained failed assertions related to behavioral logic and speed computation.
In contrast, Prompt 2 consistently generated executable and well-structured test suites without requiring any manual correction. The generated tests provided broader behavioral coverage, included more meaningful assertions, and achieved a 100% score across all executions.
Additionally, the elaborated prompt generated a significantly higher number of tests and assertions while maintaining low execution times.

#### Conclusion

The results of Experiment 1 demonstrate that prompt engineering plays a critical role in the quality of automatically generated unit tests. The elaborated prompt significantly improved the quality and reliability of generated unit tests.

### 3.2.2 Effect of Model Choice

This experiment evaluates the impact of the LLM choice on the quality of automatically generated C unit tests. Using the same prompt and environment, two different models (Grok and DeepSeek) generate test suites for the e-puck controller.

**Prompt for the model choice experiementation**

You are a software testing expert specialized in CUnit and embedded robotic systems.
Generate a complete CUnit test suite in C for the following Webots e-puck robot controller.
Objectives:
- Generate meaningful unit tests for the controller logic.
- Focus on functional correctness, sensor processing, odometry, speed computation, and obstacle avoidance behavior.
- Produce compilable CUnit tests only.
- Do not explain the code.
- Return only the final C code.
Requirements for the generated tests: Use the CUnit framework.
Include at least: sensor validity tests, odometry computation tests, speed computation tests, obstacle avoidance tests, deterministic behavior tests.
Include both normal and edge cases.
Add assertions using: CU_ASSERT, CU_ASSERT_DOUBLE_EQUAL
Create helper functions if necessary, Include a main() function executing the CUnit suite, The generated code must be self-contained and compilable, Avoid pseudocode and comments not necessary for understanding.

#### Results of Grok

Grok generated 10 test suites presented in Table 3.6 with generally good performance. Most tests passed, but failures appeared in obstacle avoidance and edge-case speed computation, showing some inconsistency in complex robotic behaviors.

| Suite | Total Tests | Passed Tests | Failed Tests | Passed Assertions | Failed Assertions | Time (s) |
|---|---|---|---|---|---|---|
| 1 | 10 | 9 | 1 | 57 | 1 | 0.024 |
| 2 | 10 | 9 | 1 | 36 | 1 | 0.027 |
| 3 | 10 | 7 | 3 | 39 | 5 | 0.031 |
| 4 | 10 | 8 | 2 | 48 | 3 | 0.035 |
| 5 | 11 | 8 | 3 | 28 | 5 | 0.038 |
| 6 | 12 | 9 | 3 | 29 | 6 | 0.039 |
| 7 | 10 | 6 | 4 | 33 | 14 | 0.063 |
| 8 | 10 | 8 | 2 | 34 | 3 | 0.038 |
| 9 | 11 | 8 | 3 | 20 | 5 | 0.035 |
| 10 | 9 | 8 | 1 | 15 | 2 | 0.025 |
| **Average** | **10.3** | **8.0** | **2.3** | **33.9** | **4.5** | **0.036** |

Table 3.6: Grok Results in Experimentation 2

### Results of Deepseek

DeepSeek produced the table 3.7 more stable result with a higher number of passed tests and assertions. Failures mainly concerned extreme sensor values and some speed computation cases, but overall performance was better than Grok.

| Suite | Total Tests | Passed Tests | Failed Tests | Passed Assertions | Failed Assertions | Time (s) |
|---|---|---|---|---|---|---|
| 1 | 13 | 11 | 2 | 59 | 4 | 0.050 |
| 2 | 17 | 13 | 4 | 39 | 8 | 0.073 |
| 3 | 13 | 11 | 2 | 36 | 4 | 0.049 |
| 4 | 15 | 10 | 5 | 33 | 7 | 0.050 |
| 5 | 14 | 13 | 1 | 45 | 2 | 0.037 |
| 6 | 19 | 18 | 1 | 62 | 4 | 0.057 |
| 7 | 10 | 5 | 5 | 45 | 12 | 0.061 |
| 8 | 8 | 6 | 2 | 32 | 5 | 0.033 |
| 9 | 12 | 11 | 1 | 51 | 3 | 0.035 |
| 10 | 18 | 16 | 2 | 40 | 3 | 0.062 |
| **Average** | **13.9** | **11.4** | **2.5** | **44.2** | **5.2** | **0.051** |

Table 3.7: DeepSeek Experimental Results

### Graphical Comparison

Figure 3.2 shows a comparison between DeepSeek and Grok in average passed tests, failed tests, passed assertions, and score obtained.

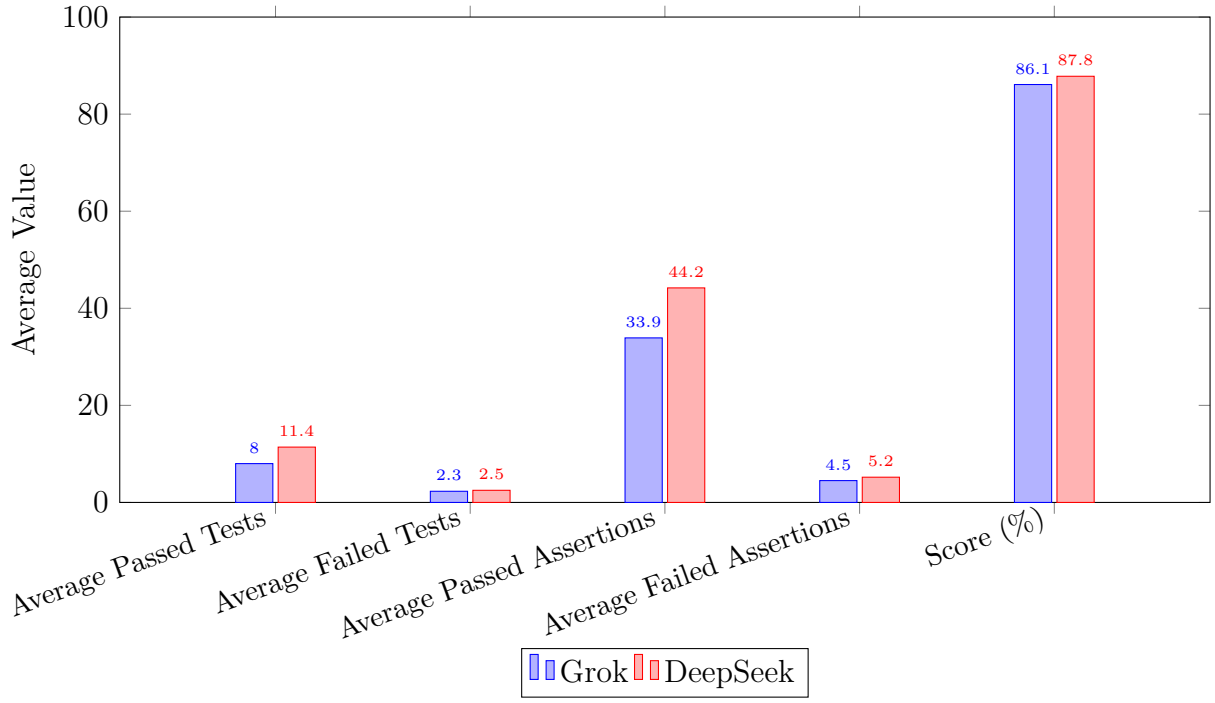


Figure 3.2: Average Results Comparison between Grok and DeepSeek

### Analysis and Discussion

The results obtained during experimentation 3.2.2 demonstrate that both large language models were able to generate functional unit tests for the e-puck robot controller. However, noticeable differences were observed in terms of reliability, correctness, and behavioral validation.
DeepSeek achieved the best overall performance with a score of 87.8%, compared to 86.1% for Grok. In addition, DeepSeek produced a higher average number of passed tests and assertions, indicating better consistency in the generated testing logic.
Most failures in both models were related to the following:

- obstacle avoidance behavior.
- Braitenberg speed computation.
- symmetry validation.
- and edge-case handling for saturated sensor values.

Several generated tests also revealed incorrect assumptions regarding expected wheel speeds and sensor normalization formulas. These failures suggest that although LLMs can automate a significant part of software testing, they still struggle with domain-specific robotic behaviors and precise mathematical expectations.
Overall, the experiment shows that LLM-based automated test generation can considerably reduce development effort, but manual verification and correction remain necessary to ensure test validity and behavioral correctness.

**Conclusion**

This experiment confirms that the choice of large language model has an influence on the quality of generated unit tests. However, this influence remains moderate compared to other factors such as prompt design and specification quality.
More importantly, while LLMs are effective for automating test generation, they are not sufficient on their own to guarantee fully reliable and behaviorally accurate tests without additional verification and refinement.

### 3.2.3 Modeling abstraction results

This experiment compares two different test specification strategies in order to evaluate their impact on the quality and behavior of automatically generated C unit tests.

**Prompt 1: Property-Based Specification**

Generate C unit tests for the following Webots e-puck robot controller using the CUnit framework.
The generated tests must validate the following properties:
**Property 1 – Sensor Reading Validity**
IF the simulation is running
THEN all distance sensor values remain within the valid range [0, 1024].
**Property 2 – Odometry Accuracy**
IF the wheel encoder sensors provide known increments
THEN the computed odometry correctly reflects distance traveled and orientation change.
**Property 3 – Speed Computation Consistency**
IF sensor values are identical
THEN the computed left and right wheel speeds remain consistent between simulation steps.
**Property 4 – Obstacle Avoidance**
IF an obstacle is detected within the threshold distance
THEN the robot adjusts wheel speeds to avoid collision.
**Property 5 – Real-time Execution**
IF the simulation step occurs
THEN all sensor readings, odometry computation, speed calculation, and motor updates are completed within the step time.
**Property 6 – Deterministic Behavior**
IF the same sensor inputs are provided in multiple runs
THEN the robot produces identical wheel speed outputs.
Generate executable and compilable CUnit test code with meaningful assertions.

The following table 3.8 presents the results obtained using the property-based specification strategy for automatically generated C unit tests.

| Suite | Total Tests | Passed Tests | Failed Tests | Passed Assertions | Failed Assertions | Time(s) |
|---|---|---|---|---|---|---|
| 1 | 6 | 6 | 0 | 22 | 0 | 0.012 |
| 2 | 6 | 6 | 0 | 32 | 0 | 0.011 |
| 3 | 6 | 6 | 0 | 25 | 0 | 0.011 |
| 4 | 6 | 6 | 0 | 28 | 0 | 0.011 |
| 5 | 6 | 6 | 0 | 34 | 0 | 0.012 |
| 6 | 6 | 6 | 0 | 34 | 0 | 0.013 |
| 7 | 6 | 6 | 0 | 33 | 0 | 0.015 |
| 8 | 6 | 6 | 0 | 33 | 0 | 0.012 |
| 9 | 6 | 6 | 0 | 35 | 0 | 0.014 |
| 10 | 6 | 6 | 0 | 35 | 0 | 0.015 |
| **Average** | **6** | **6** | **0** | **31.1** | **0** | **0.013** |

Table 3.8: Results obtained using the property-based specification

**Prompt 2: Activity Diagram Path-Based Specification**

Generate C unit tests for the following Webots e-puck robot controller using the CUnit framework. The generated tests must cover the following execution paths extracted from the activity diagram:

**Path 1 – Obstacle detected + simulation continues (loop)**
Start
→ Read distance sensors
→ Read accelerometer & encoders
→ Compute odometry
→ Obstacle detected = YES
→ Adjust wheel speeds (avoid obstacle)
→ Set motor velocities
→ Simulation running = YES
→ Loop back to start

**Path 2 – No obstacle + simulation continues (loop)**
Start
→ Read distance sensors
→ Read accelerometer & encoders
→ Compute odometry
→ Obstacle detected = NO
→ Move forward
→ Set motor velocities
→ Simulation running = YES
→ Loop back to start

**Path 3 – Obstacle detected + simulation stops**
Start
→ Read distance sensors
→ Read accelerometer & encoders
→ Compute odometry
→ Obstacle detected = YES
→ Adjust wheel speeds
→ Set motor velocities
→ Simulation running = NO
→ End

**Path 4 – No obstacle + simulation stops**
Start
→ Read distance sensors

→ Read accelerometer & encoders
→ Compute odometry
→ Obstacle detected = NO
→ Move forward
→ Set motor velocities
→ Simulation running = NO
→ End
Generate executable and compilable CUnit test code with meaningful assertions covering these execution paths.

The following table 3.9 presents the results obtained using the activity diagram path-based specification approach.

| Suite | Total Tests | Passed Tests | Failed Tests | Passed Assertions | Failed Assertions | Time(s) |
|---|---|---|---|---|---|---|
| 1 | 4 | 3 | 1 | 18 | 1 | 0.010 |
| 2 | 4 | 2 | 2 | 15 | 2 | 0.014 |
| 3 | 4 | 2 | 2 | 14 | 2 | 0.015 |
| 4 | 4 | 2 | 2 | 14 | 2 | 0.014 |
| 5 | 4 | 2 | 2 | 14 | 2 | 0.015 |
| 6 | 4 | 2 | 2 | 14 | 2 | 0.015 |
| 7 | 4 | 2 | 2 | 14 | 2 | 0.014 |
| 8 | 4 | 2 | 2 | 14 | 2 | 0.016 |
| 9 | 4 | 2 | 2 | 14 | 2 | 0.016 |
| 10 | 4 | 2 | 2 | 14 | 2 | 0.016 |
| **Average** | **4** | **2.1** | **1.9** | **14.9** | **1.9** | **0.014** |

Table 3.9: Results obtained using the activity diagram path-based specification

### Graphical Comparison

Figure 3.3 compares the average passed tests, failed tests, passed assertions, failed assertions, and score obtained for both specification strategies.

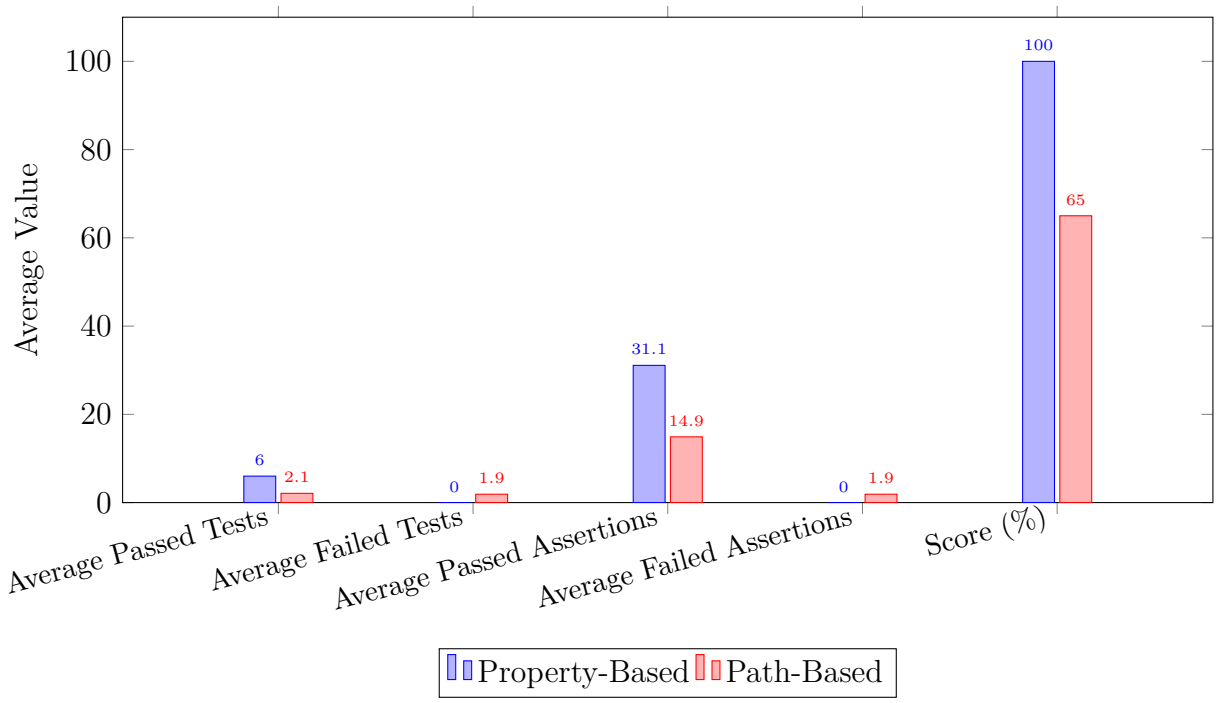


Figure 3.3: Comparison between property-based and path-based specifications

### Analysis and Discussion

The property-based specification produced stable and fully executable test suites with no failures across all generations. The generated tests focused on validating global behavioral properties through modular and reliable assertions. In contrast, the activity diagram path-based specification generated stricter and more scenario-oriented assertions, leading to repeated failures in several execution paths. While the path-based strategy provided deeper behavioral validation and stronger execution-flow coverage, it also introduced less stable and more fragile test behaviors.

### Conclusion

The results show that the specification strategy significantly affects the generated unit tests. Property-based prompts produce stable and reliable tests with high score, while path-based prompts provide deeper behavioral validation but introduce stricter assertions and higher failure rates. Both approaches are complementary and useful for different testing objectives.

## 3.3 Comparative Analysis of Experimental Results

This section presents a comparative analysis of the three conducted experiments in order to evaluate the influence of prompt engineering presented in 3.2.1, LLM selection presented in 3.2.2, and modeling abstraction presented in 3.2.3 on the quality of automatically generated unit tests. The comparison focuses on several evaluation aspects, including test reliability, assertion quality, behavioral coverage, execution stability, and overall score. The results obtained from Experiment 1 showed that prompt engineering had the strongest impact on generated test quality. The elaborated prompt produced more executable tests, a higher number of meaningful assertions, and improved behavioral coverage and eliminated the need for manual corrections compared to the simple prompt.

In Experiment 2, both DeepSeek and Grok successfully generated executable unit tests, although DeepSeek achieved slightly better numerical performance in terms of passed tests and assertions, while Grok demonstrated stronger reasoning capabilities in certain behavioral scenarios but with slightly lower stability. The differences between the two models remained less significant than the impact observed from prompt design. Experiment 3 highlighted the influence of modeling abstraction on test behavior. Property-based specifications generated highly stable and reliable tests with modular assertions and high scores, whereas activity-diagram path-based specifications provided deeper execution-flow validation and more realistic behavioral scenarios but also introduced stricter assertions and more frequent failures. The consistency of the results obtained in Experiments 1 and 2 further reinforces the reliability and validity of the findings observed in Experiment 3. Overall, the experiments demonstrate that LLMs are capable of generating useful unit tests for robotic controllers, although the generated tests still require validation and occasional manual verification. The obtained results indicate that combining detailed prompts, suitable LLMs, and well-structured specifications can significantly improve the effectiveness and reliability of AI-generated software tests.

## 3.4 Human vs AI-generated Tests

This section compares manually written unit tests with AI-generated tests in order to evaluate their strengths, limitations, and effectiveness in robotic software testing.

AI-generated tests provided significant advantages in terms of speed and automation. Large language models were able to generate complete CUnit test suites, assertions, and multiple behavioral scenarios in a short time. This considerably reduced the manual effort required during the testing process and improved the exploration of different execution cases.

In contrast, human-generated tests demonstrated better reliability and behavioral understanding. Manual tests were generally more precise in validating robotic behaviors such as obstacle avoidance, odometry computation, and wheel speed calculations. Human-written tests also contained fewer incorrect assumptions and required less correction after execution.

The experiments also showed that AI-generated tests produced a larger number of assertions and explored more edge cases than manual tests. However, some generated assertions were based on incorrect mathematical expectations or unrealistic assumptions regarding sensor normalization and controller behavior. These limitations were especially visible in complex robotic scenarios involving dynamic obstacle reactions and speed computation logic.

From a maintainability perspective, human-generated tests were generally easier to understand and modify, while AI-generated tests occasionally contained redundant structures and repetitive assertions. Nevertheless, AI-generated tests provided better scalability by allowing rapid generation of multiple testing scenarios with limited development effort.

| Aspect | Human-generated Tests | AI-generated Tests |
|---|---|---|
| Generation Speed | Slow | Very Fast |
| Reliability | High | Moderate |
| Behavioral Understanding | Strong | Limited |
| Assertion Quantity | Moderate | High |
| Edge-case Exploration | Limited | Better |
| Need for Manual Correction | Low | Medium |
| Maintainability | High | Moderate |
| Scalability | Limited | High |

Table 3.10: Global comparison between human-generated and AI-generated unit tests

Overall, the results show that AI-generated tests can significantly support automated software testing

by accelerating test generation and improving behavioral coverage. However, human verification remains necessary to ensure correctness and reliability, particularly for complex robotic behaviors.

# Conclusion

This work studied the use of Large Language Models for automated unit test generation in robotic software systems developed in Webots. Several experiments were conducted to evaluate the impact of prompt engineering, model selection, and modeling abstraction on the quality of generated CUnit tests.
The results showed that detailed prompts significantly improve test quality and reliability, while the choice of the LLM has a moderate impact on the generated tests. The experiments also demonstrated that property-based specifications produce more stable tests, whereas path-based specifications provide deeper behavioral validation but introduce stricter assertions and higher failure rates.
The comparison between human-generated and AI-generated tests highlighted that AI considerably accelerates test generation and improves behavioral exploration, while human-written tests remain more reliable for validating complex robotic behaviors and mathematical correctness.
Overall, the experiments confirm that LLMs can effectively support software testing activities for robotic systems by reducing development effort and generating executable unit tests. However, manual verification remains necessary to ensure correctness and behavioral reliability. The results suggest that combining human expertise with AI-assisted test generation is currently the most effective approach for robotic software testing.

## Future work

Several extensions can be considered for future work. First, the proposed approach could be evaluated on more complex robotic architectures, including multi-agent robotic systems involving communication, coordination, and collaborative behaviors between multiple robots.
Another important extension would be the integration of Model-Based Testing (MBT) techniques already provided by formal models and activity diagrams in order to automatically derive richer and more complete behavioral test scenarios.
Future research could also investigate the use of Large Language Models specialized in software testing and verification tasks. Fine-tuned or domain-specific LLMs dedicated to test generation may improve assertion correctness, behavioral understanding, and robustness for robotic applications.
Additional improvements may include the integration of code coverage analysis, mutation testing, automatic test repair, and continuous integration pipelines for automated robotic software validation on both simulated and real robotic platforms.

# Bibliography


[Anthropic, 2024] Anthropic (2024). Claude 3.5: Sonnet. Accessed: 2026-05-20.

[Bombarda et al., 2023] Bombarda, A., Bonfanti, S., Gargantini, A., Lei, Y., and Duan, F. (2023). RATE: A model-based testing approach that combines model refinement and test execution. *Software Testing, Verification and Reliability*, 33(2):e1835.

[Castillos et al., 2014] Castillos, K. C., Dadeau, F., and Julliand, J. (2014). Coverage criteria for model-based testing using property patterns. *Electronic Proceedings in Theoretical Computer Science*, 141:29–43.

[Cyberbotics, 2024] Cyberbotics (2024). Cyberbotics: Webots robot simulator. Accessed: 2026-05-20.

[DeepMind, 2024] DeepMind (2024). Gemini: Flash. Accessed: 2026-05-20.

[DeepSeek-AI, 2024] DeepSeek-AI (2024). Deepseek-v3 technical report. https://github.com/deepseek-ai/DeepSeek-V3.

[IEEE Computer Society, 1990] IEEE Computer Society (1990). IEEE Standard Glossary of Software Engineering Terminology. Technical Report IEEE Std 610.12-1990, Institute of Electrical and Electronics Engineers, New York, NY, USA.

[Ihanzal, 2026] Ihanzal, H. (2026). Research project repository. https://disc.univ-fcomte.fr/cr700-gitlab/ihanzal/ihanzal_hbensaid_research_project. University GitLab repository, accessed: 2026-05-20.

[ISTQB, 2024] ISTQB (2024). Istqb ctfl syllabus v4.0.1. Accessed: 2026-05-20.

[Kokorin, 2025] Kokorin, I. (2025). Model-based testing of practical distributed systems in actor model. *arXiv preprint arXiv:2512.08698*.

[Magnitia IT, 2024] Magnitia IT (2024). Ai in software testing: Advantages, disadvantages, contributions & the future. https://magnitia.com/blog/ai-in-software-testing-advantages-disadvantages-contributions-the-future/. Accessed: 2026-04-30.

[Mohacsi and Felderer, 2021] Mohacsi, S. and Felderer, M. (2021). Ai-based enhancement of test models in an industrial model-based testing tool. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 561–565. IEEE.

[MSYS2, 2024] MSYS2 (2024). Msys2: A software distribution and building platform for windows. Accessed: 2026-05-20.

[OpenAI, 2024] OpenAI (2024). Introducing gpt-5. Accessed: 2026-05-20.

[xAI, 2025] xAI (2025). Introducing grok. https://x.ai/blog/grok.

[Yang et al., 2025] Yang, Z., Huang, R., Cui, C., Niu, N., and Towey, D. (2025). Requirements-based test generation: A comprehensive survey. *arXiv preprint arXiv:2505.02015*.

[Yoshimoto et al., 2026] Yoshimoto, S., Fujita, S., Horikawa, K., Feitosa, D., Kashiwa, Y., and Iida, H. (2026). Testing with ai agents: An empirical study of test generation frequency, quality, and coverage. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR '26)*. ACM. arXiv preprint arXiv:2603.13724.

[Zhao et al., 2023] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. (2023). A survey of large language models. *arXiv preprint arXiv:2303.18223*.

# Appendix A

# Research Articles Analysis

## A.1 Article 1: RBTG: A Comprehensive Survey [Yang et al., 202

### Article Presentation

- **Authors:** ZHENZHEN YANG, RUBING HUANG, CHENHUI CUI, NAN NIU, DAVE TOWEY.
- **Year of publication:** 06 October 2025

### Context of the Article

the paper represent an overview of 267 studies published up to 2024 talks about software testing and how test cases can be automatically created from software requirements instead of writing them manually

### Main results

They classified RBTG research: requirement types, generation approaches, test case formats, tools, and evaluation methods.
They also noticed a shift from formal models to more NLP-based approaches for natural language requirements.

### Conclusion / future work

They suggest improving tool usability, evaluation quality, and real-world integration.
They also recommend exploring LLM/AI-driven RBTG as a promising future direction.

## A.2 Article 2: Coverage Criteria for MBT using Property Patterns [Castillos et al., 2014]

### Article Presentation

This paper present a model-based testing approach aiming at generating test cases from a UML/OCL model and a given test property.

- **Authors:** Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand
- **Year of publication:** 2014
- **Journal:** Electronic Proceedings in Theoretical Computer Science (EPTCS)

### What is Model-Based Testing (MBT)?

MBT means testing software by using a model of how the system should work. The model generates the test cases and also tells what the correct results should be. The approach described in the paper uses UML/OCL models, specifically a subset called UML4ST .

### How Property Patterns Work

**Defining Properties:** Instead of complex temporal logics, the system uses property patterns to describe what events should or should not happen in the system.
**Translation to Automata:** Each property is turned into an automaton that checks the system's behavior and helps generate tests.

### Coverage Criteria for Testing

The paper introduces new coverage rules for property automata, focusing on key events ("$\alpha$-transitions"). These include:

- **$\alpha$-transition coverage:** Each event is triggered by at least one test.
- **$\alpha$-transition-pair coverage:** Every consecutive pair of events is tested.
- **k-pattern coverage:** Internal loops in patterns are executed k times.
- **k-scope coverage:** Loops in the automaton's scope are repeated k times, covering at least one key event each time.

### Robustness Testing through Automata Mutation

Property automata can detect error states that indicate violations, but these states are normally unreachable in a correct system. To test robustness, mutation operators modify the events leading to these errors, making them triggerable. A new automaton is then created with the error state as the final state, and tests ensure these mutated paths are exercised to check how the system handles potential violations.

## Experimentation

The MBT approach was applied in industrial projects using a framework to define properties, measure coverage, and generate tests. Engineers found the property language and automaton visualizations easy to use and helpful. By generating tests and applying mutations, the method revealed property violations and other faults, proving effective for improving test coverage and fault detection in real industrial settings.

## A.3 Article 3: AI-Based Enhancement of Test Models in an Industrial Model-Based Testing [Mohacsi and Felderer, 2021

### Article Presentation

- **Title of the article:** AI-Based Enhancement of Test Models in an Industrial Model-Based Testing
- **Authors:** Stefan Mohacsi, Michael Felderer
- **Year of publication:** 2021
- **Conference:** IEEE (SANER - International Conference on Software Analysis, Evolution and Reengineering)

### Context of the Article

This article proposes a method to integrate AI-based improvements into an industrial model-based testing (MBT) tool called TEMPPO Designer.
The goal of this work is to improve this tool by adding AI support. This helps testers decide what to test first, better manage time and resources, estimate test execution time, and make the testing process easier and more efficient.

### the problem of this article

Model-Based Testing tools such as TEMPPO already exist, but they mostly depend on human experience and do not really use artificial intelligence. Because of this, creating and prioritizing test models takes time and effort, and Model-Based Testing is still not widely used in real projects.

### Proposed Methodology

The proposed architecture add new components that integrate with system they start from collects testing data from different tools(Data sources), and stores it in a central database (Data Collector) This data is analyzed using artificial intelligence (Data Analyzer) to support decisions such as test prioritization and test execution time estimation. The results are then integrated into TEMPPO Designer to help testers make better decisions based on risk, time, and available resources.

### Limitation and Future Improvements

The main limitation is that The article is conceptual, not experimental. The future work focuses on implementing the approach, evaluating it in real projects.

### Relevance and Contribution

The paper is relevant because it shows how AI can improve an existing testing tool. and the main contribution is making a simple framework that uses data and AI to help modeling and testing decisions.

## A.4 Article 4: Model-based Testing of Practical Distributed Systems in Actor Model [Kokorin, 2025]

### Article Presentation

This paper presents a model-based testing approach for verifying distributed systems implemented in the actor model by generating an exhaustive test suite from a TLA+ model.

- **Authors:** Ilya Kokorin, Evgeny Chernatskiy, Vitaly Aksenov
- **Year of publication:** 2025
- **Source:** arXiv:2512.08698 [cs.DC]

### What are Distributed Systems and Why Are They Hard to Verify?

1. What are Distributed Systems and Why Are They Hard to Verify? A distributed system is a system made of several computers (nodes) that work together and communicate by sending messages over a network. They are hard to verify because:

- Many nodes run at the same time.
- Messages can arrive in different orders.
- some Nodes can crash or restart.
- The network can lose or delay messages.

All these factors create a huge number of possible execution scenarios. It is impossible to manually test all of them.

### Limitations of Existing Approaches

- Unit tests and integration tests only check a few scenarios
- Fuzzing generates random tests, but it does not guarantee full coverage.
- Model checking can verify a formal model, but not the real code.

So even if a system is mathematically proven correct, the actual implementation may still contain bugs

### The Proposed Solution

1. The system uses the actor model, where components react to messages and run in a predictable way for easy debugging
2. A formal model is created in TLA+, and a tool generates a graph of all possible states and actions of the system.
3. A graph algorithm is then used to create a small but complete set of tests that goes through every possible action in the graph at least once.

4. During testing, the real system regularly compares its state with the model, to check that it always behaves as expected.

## Results of the Proposed Solution

The approach was tested on a real industrial system (a replication protocol at VK.com).
The results show that:

- The system can generate thousands of tests per second.
- It can cover millions of states and transitions.
- It successfully verifies that the real implementation matches the formal model.
- Bugs can be reproduced exactly and debugged easily.

This makes the approach fast, scalable, and reliable for testing real distributed systems.

## A.5 Article 5: RATE – A Model-Based Testing Approach Combining Model Refinement and Test Execution [Bombard

### Presentation of the Article

- **Title of the article:** RATE – A Model-Based Testing Approach Combining Model Refinement and Test Execution
- **Authors:** Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Yu Lei. Feng Duan
- **Year of publication:** 16 November 2022

### Context of the Article

This article introduces RATE (Refinement And Test Execution), a model-based testing approach designed to improve software testing by combining model refinement, automatic test generation, and test execution.

### What is Model-Based Testing (MBT)?and his limitations

Model-Based Testing (MBT) is widely used to automatically generate test cases from a formal model of the system. However, traditional MBT approaches face several limitations:

- They require a complete and precise model from the beginning
- Building such a model is time-consuming and complex
- They do not effectively use feedback from test execution results
- They may lead to low code coverage

Because of these limitations, there is a need for a more flexible and adaptive testing approach.

### The proposed solution

The article proposes the RATE approach as an iterative testing method where a simple abstract model of the system is first created using ASM. From this model, test cases are automatically generated and then executed on the real system. The results of these tests, especially the code coverage, are analyzed to identify untested or missing behaviors. Based on this feedback, the model is refined by adding more details or correcting inaccuracies. This process is repeated in a loop, progressively improving both the model and the quality of testing, while combining model-based (black-box) and code-based (white-box) techniques to achieve better fault detection and system validation.
So RATE works as a loop:

- Create an abstract model (using ASM)
- Generate test cases automatically
- Execute tests on the real system
- Analyze results and code coverage

- Refine the model by adding missing behaviors
- Repeat the process

## Results

The approach was tested on different systems (traffic lights, medical systems, and ventilators). Results show:

- Improved code coverage
- Better fault detection
- Effective testing even with simple initial models
- Progressive improvement through iterations

# Appendix B

# Contributions of Human-generated Tests

## B.1 Webots – DJI Mavic 2 Pro Sample World

**SCENARIO DESCRIPTION:**

In this scenario, we use the DJI Mavic 2 Pro drone available in Webots.
The drone starts on the ground. When it receives the takeoff command, it rises into the air and keeps a stable height. While flying, it can move forward, backward, left, or right by slightly tilting. After each movement, it automatically stabilizes and returns to a steady hover.

**specification document:**

**1. Context**

This project studies the behavior of the DJI Mavic 2 Pro drone provided in Webots.
The drone is controlled using the keyboard and the objective is to analyze its behavior and express it in terms of requirements and testable properties.

**2. Functional Objectives**

The drone must be able to:

- Take off by increasing motor thrust
- Maintain a target altitude automatically
- Adjust its altitude using keyboard commands
- Move forward and backward (pitch control)
- Turn left and right (yaw control)
- Strafe left and right (roll control)
- Automatically stabilize roll, pitch, and altitude using sensor feedback

### 3. Inputs

The system receives:

- $\uparrow$ : move forward
- $\downarrow$ : move backward
- $\rightarrow$ : turn right (yaw)
- $\leftarrow$ : turn left (yaw)
- Shift + $\uparrow$ : increase target altitude
- Shift + $\downarrow$ : decrease target altitude
- Shift + $\rightarrow$ : strafe right (roll)
- Shift + $\leftarrow$ : strafe left (roll)

### 4. Outputs

The system controls:

- Motor velocities (4 propellers)
- Roll correction
- Pitch correction
- Yaw correction
- Vertical thrust adjustment
- Target altitude

### 5. Constraints

- The drone must stabilize automatically using PID control.
- Motor speeds must remain within operational limits.
- Altitude control must smoothly converge to the target altitude.
- Movement commands modify disturbances but do not disable stabilization.

**Requirements:**

**REQ-001**
The drone shall stabilize itself automatically using sensor feedback (IMU, GPS, Gyro).
**REQ-002**
The drone shall maintain a target altitude and adjust motor thrust to reach it.
**REQ-003**
When Shift + ↑ is pressed, the target altitude shall increase.
**REQ-004**
When Shift + ↓ is pressed, the target altitude shall decrease.
**REQ-005**
When ↑ is pressed, the drone shall move forward by applying a pitch disturbance.
**REQ-006**
When ↓ is pressed, the drone shall move backward by applying a pitch disturbance.
**REQ-007**
When Shift + → is pressed, the drone shall strafe right by applying a roll disturbance.
**REQ-008**
When Shift + ← is pressed, the drone shall strafe left by applying a roll disturbance.
**REQ-009**
When → is pressed, the drone shall rotate (yaw) right.
**REQ-010**
When ← is pressed, the drone shall rotate (yaw) left.

**Transform into formal properties:**

**Property 1 – Automatic Stabilization**
IF the drone is flying
THEN roll, pitch, and altitude are continuously corrected so that the drone remains stable.
**Property 2 – Altitude Control**
IF the target altitude is changed
THEN the drone adjusts its vertical thrust
SO THAT the actual altitude converges smoothly toward the target value.
**Property 3 – Forward Movement**
IF the forward command (↑) is received
THEN a pitch disturbance is applied
AND the drone moves forward
WHILE stabilization remains active.
**Property 4 – Backward Movement**
IF the backward command (↓) is received
THEN a reverse pitch disturbance is applied
AND the drone moves backward
WHILE stabilization remains active.
**Property 5 – Strafe Right**
IF the strafe-right command (Shift + →) is received
THEN a roll disturbance is applied
AND the drone moves to the right
WHILE stabilization remains active.

**Property 6 – Strafe Left**
IF the strafe-left command (Shift + ←) is received
THEN a roll disturbance is applied
AND the drone moves to the left
WHILE stabilization remains active.
**Property 7 – Yaw Rotation**
IF a rotation command (→ or ←) is received
THEN the drone changes its yaw
BUT altitude and stabilization are not affected.

**Activity Diagram :**

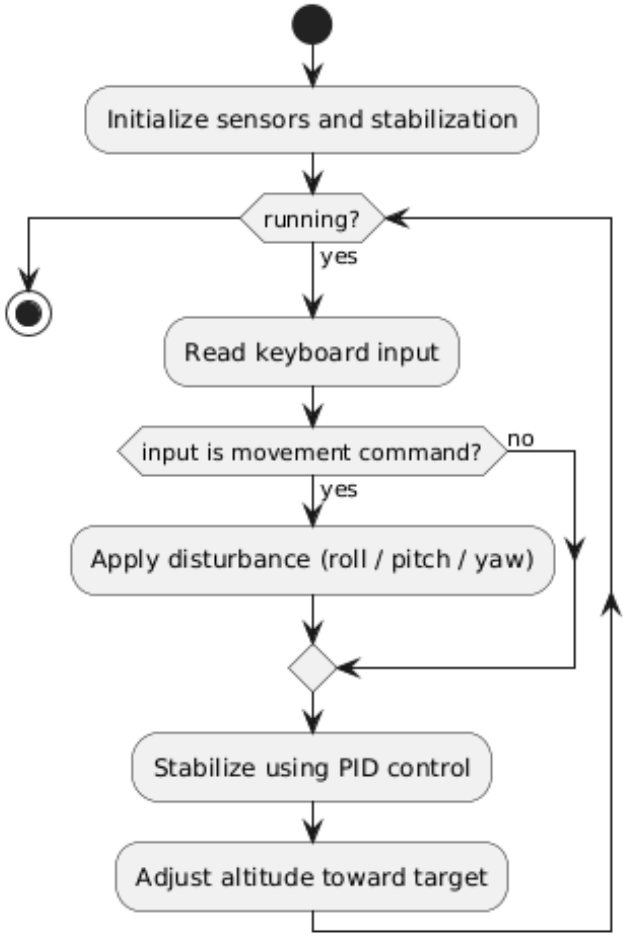


Figure B.1: Activity Diagram of the DJI Mavic 2 Pro Control System

From the activity diagram, we extract independent execution paths:

- **Path 1 – System not running**
  Start → Initialize → running? (NO) → End
- **Path 2 – Running + NO keyboard input**
  Start → Initialize → running? (YES)

→ Read keyboard
→ NOT movement command
→ Stabilize (PID)
→ Adjust altitude
→ Loop

- **Path 3 – Running + Movement command (direction)**
  Start → Initialize → running? (YES)
  → Read keyboard
  → Movement command (UP/DOWN/LEFT/RIGHT)
  → Apply disturbance (pitch/yaw)
  → Stabilize (PID)
  → Adjust altitude
  → Loop

- **Path 4 – Running + Altitude change command**
  Start → Initialize → running? (YES)
  → Read keyboard
  → Movement command (SHIFT + UP/DOWN)
  → Modify target altitude
  → Stabilize (PID)
  → Adjust altitude
  → Loop

## B.2 Webots – Crazyflie

**SCENARIO DESCRIPTION:**

The Crazyflie drone is initially controlled manually via the keyboard (arrow keys for forward/backward/right/left, W/S for altitude, Q/E for rotation) while maintaining a stable altitude with the PID controller. The drone can move horizontally and adjust its height, with automatic stabilization to prevent excessive oscillations.
**Start:** The drone takes off from a specific platform.
**Objectives**

- Navigate through a course with obstacles.
- Reach predefined waypoints.
- Perform specific maneuvers (e.g., turns, climbs, and descents).

**End:** Safely land on the destination platform.

**specification document:**

**Contexte**

This project uses the Webots simulator to control a Crazyflie drone stabilized by a PID controller. The goal is to validate flight robustness and control precision through manual keyboard inputs before implementing autonomous navigation features.

**System description**

The system consists of the following elements:

- Crazyflie Quadcopter Drone
- 4 Motors (Brushless DC)
- Sensors:
  - IMU (Inertial Measurement Unit)
  - Gyroscope
  - GPS
  - Distance Sensors (ToF/Laser)
- PID-based Controller for stabilization and flight correction.

**Principal Functions**

- Manual Takeoff and Hover (via keyboard control)
- Manual horizontal movement (arrow keys)
- Manual altitude control (W/S keys)

- Manual yaw rotation (Q/E keys)
- Automatic stabilization using onboard sensors (IMU, Gyro, GPS)
- Obstacle detection via distance sensors
- Safe landing

**System constraints**

- The system shall use the provided PID controller for stabilization.
- No collision with obstacles or environment boundaries shall be permitted during manual testing.
- Maximum flight altitude shall not exceed 2 meters.
- The drone shall operate in real-time according to the simulation time step.

**Requirements**

- REQ-001: The drone shall stabilize itself automatically using sensor feedback (IMU, GPS, Gyro).
- REQ-002: The drone shall maintain a constant altitude during flight unless a new altitude command is issued.
- REQ-003: The drone shall detect obstacles in its environment using onboard distance sensors.
- REQ-004: The drone shall respond to obstacles to prevent collisions during manual operation.
- REQ-005: The drone shall allow manual horizontal movement via keyboard input.
- REQ-006: The drone shall land safely when commanded or at mission completion.
- REQ-007: The drone shall remain within the predefined mission area boundaries.
- REQ-008: The drone shall operate in real time with continuous sensor monitoring.
- REQ-009: The drone shall limit roll and pitch angles to ensure flight stability.
- REQ-010: The drone shall be capable of transitioning safely between operational states for autonomous missions.

**Properties**

- Property 1 – Automatic Stabilization:
  - IF the drone is flying THEN roll, pitch, and altitude are continuously corrected so that the drone remains stable.
- Property 2 – Obstacle Safety:
  - IF an obstacle is detected within a critical distance THEN the drone shall modify its trajectory to prevent collision.

- Property 3 – Mission Completion:
  - IF the drone reaches the target zone THEN the drone shall initiate landing procedure.
- Property 4 – Safe Landing:
  - IF the landing procedure is active THEN vertical speed shall be reduced progressively until touchdown.
- Property 5 – Operational Boundaries:
  - IF the drone approaches mission boundaries THEN corrective action shall be applied to keep it within limits.
- Property 6 – Continuous Monitoring:
  - WHILE the drone is operating SENSOR data shall be continuously processed to update control decisions.
- Property 7 – Emergency Landing:
  - IF battery level becomes critical OR sensor failure is detected THEN the Crazyflie shall initiate controlled descent SO THAT landing occurs safely.

# Activity Diagram

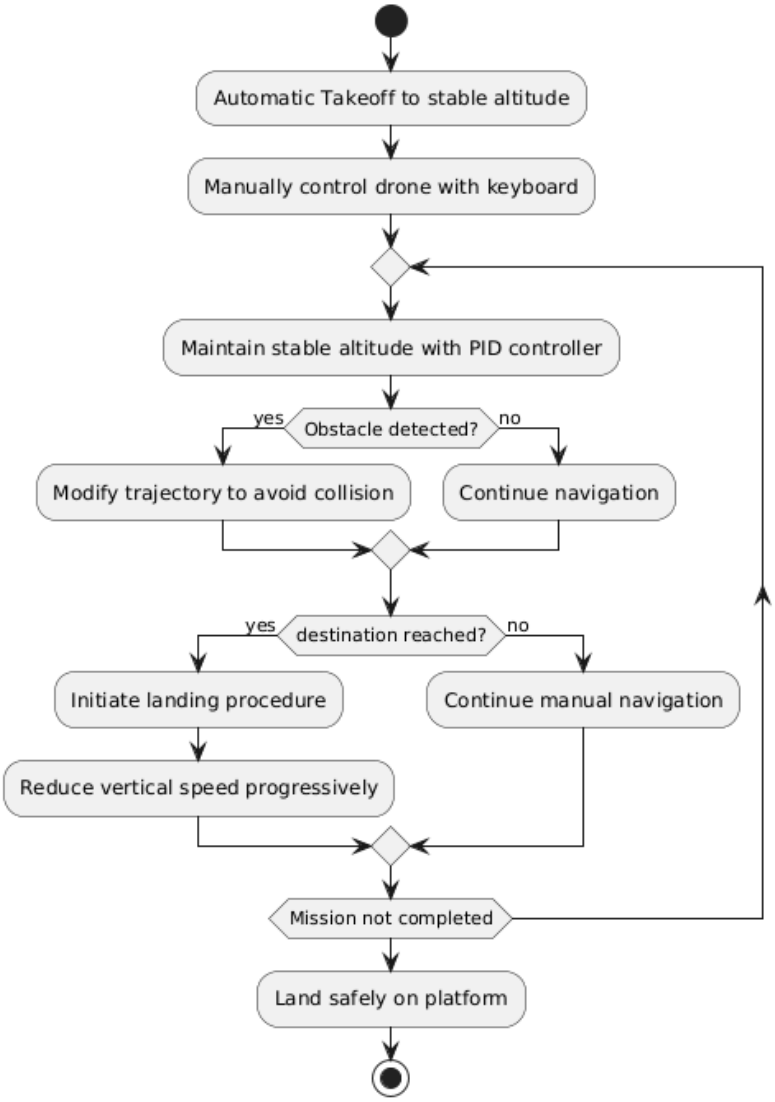


Figure B.2: Activity Diagram of crazyflie system

From the activity diagram, we extract independent execution paths:

- **Path 1 – Obstacle detected + mission continues (loop)**
  Start
  → Automatic takeoff to stable altitude
  → Manually control drone with keyboard
  → Maintain stable altitude with PID controller
  → Obstacle detected = YES
  → Modify trajectory to avoid collision
  → Destination reached = NO
  → Continue manual navigation
  → Mission not completed
  → Loop back to stable altitude control

- **Path 2 – No obstacle + mission continues (loop)**

Start
→ Automatic takeoff to stable altitude
→ Manually control drone with keyboard
→ Maintain stable altitude with PID controller
→ Obstacle detected = NO
→ Continue navigation
→ Destination reached = NO
→ Continue manual navigation
→ Mission not completed
→ Loop back to stable altitude control

- **Path 3 – Obstacle detected + successful landing**
  Start
  → Automatic takeoff to stable altitude
  → Manually control drone with keyboard
  → Maintain stable altitude with PID controller
  → Obstacle detected = YES
  → Modify trajectory to avoid collision
  → Destination reached = YES
  → Initiate landing procedure
  → Reduce vertical speed progressively
  → Land safely on platform
  → End

- **Path 4 – No obstacle + successful landing**
  Start
  → Automatic takeoff to stable altitude
  → Manually control drone with keyboard
  → Maintain stable altitude with PID controller
  → Obstacle detected = NO
  → Continue navigation
  → Destination reached = YES
  → Initiate landing procedure
  → Reduce vertical speed progressively
  → Land safely on platform
  → End

## B.3 Webots – gctronic e-puck.wbt

**specification document:**

The e-puck robot is a small mobile robot. In this scenario, the robot operates in a simple environment with fixed obstacles. The objective is to test basic autonomous navigation: move forward and avoid obstacles detected by the infrared (IR) sensors.

**Functional Objective**

- Read the values of the IR sensors at each simulation step.
- Detect obstacles in the robot's path.
- Turn or move backward to avoid collisions.
- Move forward if no obstacles are detected.

**Constraints**

- Maximum wheel speed limited to 6.28 rad/s.
- No complex behavior or AI: only simple conditional logic.
- IR sensors must operate within their valid measurement range.
- Real-time simulation: all calculations and motor updates must complete within the simulation step time.

**Requirements**

- **REQ-001** The e-puck robot shall read the values of all distance sensors at each simulation step.
- **REQ-002** The e-puck robot shall read the accelerometer values at each simulation step.
- **REQ-003** The e-puck robot shall compute odometry based on wheel encoder sensors to estimate distance traveled and change in orientation.
- **REQ-004** The e-puck robot shall compute the left and right wheel speeds using Braitenberg coefficients and sensor values.
- **REQ-005** The e-puck robot shall set the wheel motor velocities according to the computed speeds at each simulation step.
- **REQ-006** If any distance sensor detects an obstacle within the threshold range THEN the e-puck robot shall adjust its wheel speeds to avoid the obstacle.
- **REQ-007** The e-puck robot shall perform all calculations and motor updates within the simulation step time to ensure real-time behavior.
- **REQ-008** The e-puck robot shall maintain deterministic behavior given the same sensor inputs.

**Properties**

- **Property 1 – Sensor Reading Validity**
  IF the simulation is running
  THEN all distance sensor values remain within the valid range [0, 1024].

- **Property 2 – Odometry Accuracy**
  IF the wheel encoder sensors provide known increments
  THEN the computed odometry correctly reflects distance traveled and orientation change.

- **Property 3 – Speed Computation Consistency**
  IF sensor values are identical
  THEN the computed left and right wheel speeds remain consistent between simulation steps.

- **Property 4 – Obstacle Avoidance**
  IF an obstacle is detected within the threshold distance
  THEN the robot adjusts wheel speeds to avoid collision.

- **Property 5 – Real-time Execution**
  IF the simulation step occurs
  THEN all sensor readings, odometry computation, speed calculation, and motor updates are completed within the step time.

- **Property 6 – Deterministic Behavior**
  IF the same sensor inputs are provided in multiple runs
  THEN the robot produces identical wheel speed outputs.

**Activity diagram**

From the activity diagram, we extract independent execution paths:

- **Path 1 – Obstacle detected + simulation continues (loop)**
  Start
  → Read distance sensors
  → Read accelerometer & encoders
  → Compute odometry
  → Obstacle detected = YES
  → Adjust wheel speeds (avoid obstacle)
  → Set motor velocities
  → Simulation running = YES
  → Loop back to start

- **Path 2 – No obstacle + simulation continues (loop)**
  Start
  → Read distance sensors
  → Read accelerometer & encoders
  → Compute odometry
  → Obstacle detected = NO
  → Move forward

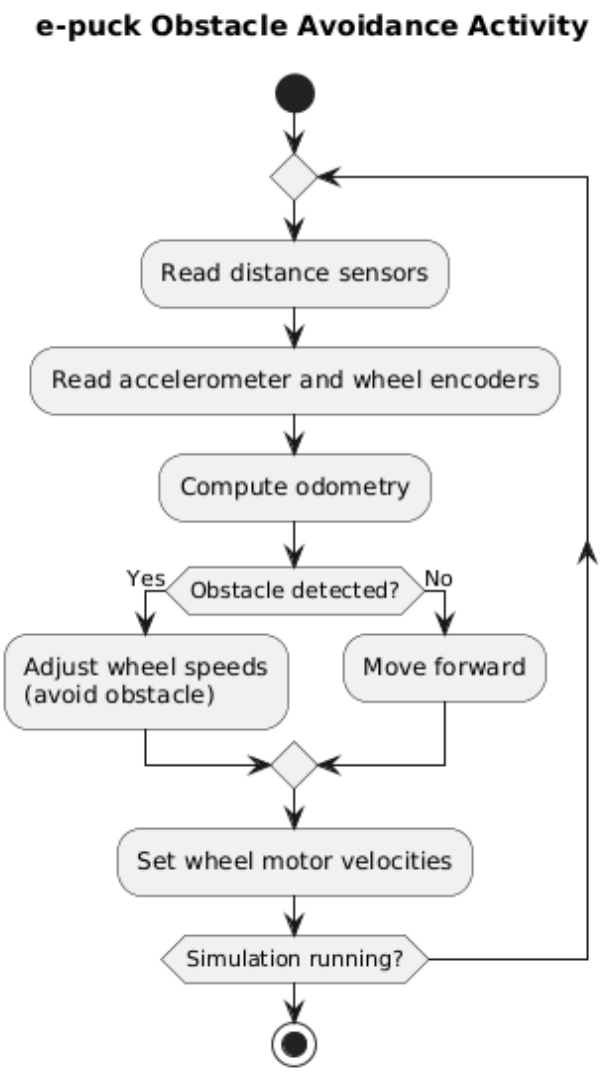


Figure B.3: Activity Diagram of gctronic e-puck.wbt

→ Set motor velocities
→ Simulation running = YES
→ Loop back to start

- **Path 3 – Obstacle detected + simulation stops**
  Start
  → Read distance sensors
  → Read accelerometer & encoders
  → Compute odometry
  → Obstacle detected = YES
  → Adjust wheel speeds
  → Set motor velocities
  → Simulation running = NO
  → End

- **Path 4 – No obstacle + simulation stops**
  Start
  → Read distance sensors
  → Read accelerometer & encoders
  → Compute odometry
  → Obstacle detected = NO
  → Move forward
  → Set motor velocities
  → Simulation running = NO
  → End