Norwegian University of Life Sciences

Master's Thesis 2024 30 ECTS
Norwegian University of Life Sciences

Runtime Verification of Autonomous Robotic Systems in ROS 2

Kjøretidsverifisering av Autonome Robotsystemer i ROS 2

Elias E. Hartmark & Tage Andersen
Applied Robotics

“You got the makings of greatness in you, but you gotta take the helm and chart your own course! Stick to it, no matter the squalls! And when the time comes, you’ll get the chance to really test the cut of your sails and show what you’re made of! And... well, I hope I’m there, catching some of the light coming off you that day.”

— John Silver

Acknowledgments

First and foremost, we would like to thank our supervisors, Associate Prof. David A. Anisi [1] and Mustafa Adam [2], for their invaluable input and continued support.

Secondly, we also wish to thank Dr. Ivan Perez, Ms. Anastasia Mavridou, and Dr. Andreas Katis at NASA Ames Research Center for their expertise, insights, guidance, and enthusiasm during our study.

We would also like to express our deepest gratitude NMBU REALTEK for giving us the opportunity to study this exciting field of engineering over the past 5 years, and for access to top-notch equipment, laboratory, and the grants necessary for the completion of our thesis.

Personally, I would like to thank my family and friends for their continued support and love throughout this five year long journey. I also want to thank my cat for always brightening my day and my fellow master's student, Tage Andersen, for sharing this life experience with me. It is difficult to imagine how different these years would have been without them, and for that, I am forever thankful.
Moreover, I am deeply grateful to my partner, Live, for never ceasing to impress me and for always making me laugh. Thank you!

- Elias Evjen Hartmark

When I close my eyes and think past over the past five years, never does a stressful exam cross my mind, but the moments I shared with my closest, my family and my friends. I would therefore like to thank them all for these wonderful times through thick and thin, and for the lifelong friendships they have given me. Only ever do I wish to be as good as they have been to me. Lastly, thank you, friend and colleague, Elias E. Hartmark for several years of friendship and eventually being the one to pass the finish-line alongside me.

- Tage Andersen

[1] Faculty of Science & Technology, NMBU REALTEK
[2] Same as above

Abstract

Keywords: Runtime Verification, Robotic Autonomous Systems, Online Monitoring, Robot Operating System

The increased use of autonomous robotic systems has necessitated advancements in safety measures due to the potential hazards these systems can pose. This increased focus on automation and robotic systems raises questions on how safety and compliance will be maintained in a steadily advancing automated society. With increasing complexity, it has become evident that rigorous offline testing cannot eliminate all dangers associated with infinite state systems, as safety is limited by the defined test-cases and coverage. To maintain safety and reliability during runtime, multiple tools often need to be set in place. One such tool is called Runtime Verification (RV). Runtime verification, a term that has been growing in popularity, refers to lightweight formal methods for observing, analyzing, and sometimes intercepting the processes of system software or Cyber-Physical Systems (CPS). One of the most common software libraries for CPS robotics is the Robot Operating System (ROS). While ROS has a few runtime verification software tools available, the recent release of ROS 2 has created a gap regarding compatible tools and their usage. Especially with the shift in infrastructure from a centralized- to distributed discovery. In the case of a flawed or failing safety controller, runtime verification plays a pivotal role in detecting what went wrong, when and where it occurred, and how to administer possible mitigating actions. The primary objective of this thesis is to create a pipeline for the automatic generation of RV monitors for ROS 2 from natural language requirements into formally verified temporal logic equations. To achieve this, the pipeline employs NASA's Copilot, OGMA, and FRET tools to convert natural language safety requirements into formally verified temporal logic monitors.

This study also addresses the gap in RV tools for ROS 2, providing a comprehensive overview of existing tools, implementation strategies, and the benefits of RV in CPS. The framework's performance is evaluated through simulation and field testing using the Thorvald-005 agricultural-robot and D435 RealSense camera, focusing on system overhead, event-to-stream-based communication, and violation reporting efficiency.

Key findings demonstrate that the RV framework effectively improves safety and reliability, with the monitor maintaining acceptable system performance even under high message rates. The research also highlights the importance of acknowledging hardware limitations and enhancing predictive capabilities to further increase system robustness. The findings prove that the automatic generation of formally verified RV monitors for ROS 2 is possible with current software and is quick to implement or change, making it feasible to test multiple specification models on the same system without changing the source code.

Future work involves refining the generative RV framework, improving hardware integration, and extending compatibility to other ROS distributions. The results have contributed to the ongoing development of the NASA RV tools, promoting safer deployment of autonomous systems in agriculture and other domains. This thesis itself is a contribution to the methodology and knowledge ascertained to properly understand and use Linear Temporal Logic (LTL) monitors.

Sammendrag

Nøkkelord: Kjøretidsverifisering, Autonome robotsystemer, Online overvåking, Robot Operating System

Den økte bruken av autonome robotsystemer har nødvendiggjort fremskritt i sikkerhetstiltak på grunn av de potensielle farene disse systemene kan utgjøre. Dette økte fokuset på automatisering og robotsystemer stiller spørsmål om hvordan sikkerhet og samsvar vil opprettholdes i et stadig mer automatisert samfunn. Med økende kompleksitet har det blitt tydelig at grundig offline testing ikke kan ta høyde for alle farer forbundet med uendelige tilstandssystemer, da sikkerheten er begrenset av de definerte testtilfellene. For å opprettholde sikkerhet og pålitelighet under drift, er det ofte nødvendig å sette inn flere verktøy. Et slikt verktøy kalles Kjøretidsverifisering (Runtime Verification, RV). Kjøretidsverifisering, et begrep som har blitt stadig mer populært, refererer til lette formelle metoder for å observere, analysere og noen ganger avbryte prosessene til systemprogramvare eller Cyber-Physical Systems (CPS). Et av de mest brukte språkene for CPS-robotikk er Robot Operating System (ROS). Selv om ROS har mange tilgjengelige kjøretidsverifiseringsverktøy, har den nylige utgivelsen av ROS 2 skapt et kunnskapsgap angående kompatible verktøy og deres bruk, spesielt med overgangen fra en sentralisert til en distribuert oppdagelsesinfrastruktur. Ved feil eller svikt i en sikkerhetskontroller spiller kjøretidsverifisering en avgjørende rolle i å oppdage hva som gikk galt, når og hvor det skjedde, og hvordan man kan administrere avbøtende tiltak. Hovedmålet med denne avhandlingen er å lage en pipeline for automatisk generering av RV-overvåkere for ROS 2 fra krav i naturlig språk til formelt verifiserte temporale logiske ligninger. For å oppnå dette, benytter pipelinen NASAs Copilot, OGMA og FRET-verktøy for å konvertere sikkerhetskrav i naturlig språk til formelt verifiserte temporale logiske overvåkere.

Denne studien tar også for seg gapet i RV-verktøy for ROS 2, og gir en omfattende oversikt over eksisterende verktøy, implementeringsstrategier og fordelene med RV i CPS. Rammeverkets ytelse evalueres gjennom simulering og felttesting ved bruk av landbruksroboten Thorvald-005 og D435 RealSense-kameraet, med fokus på systembelastning, hendelse-til-strøm-basert kommunikasjon og effektivitet i bruddrapportering.

Viktige funn viser at RV-rammeverket effektivt forbedrer sikkerhet og pålitelighet, med overvåkeren som opprettholder akseptabel systemytelse selv under høye meldingshastigheter. Forskningen understreker også viktigheten av å erkjenne maskinvarebegrensninger og forbedre prediktive evner for ytterligere å øke systemets robusthet. Funnene beviser at automatisk generering av formelt verifiserte RV-overvåkere for ROS 2 er mulig med dagens programvare og er raskt å implementere eller endre, noe som gjør det mulig å teste flere spesifikasjonsmodeller på samme system uten å endre kildekoden.

Fremtidig arbeid innebærer å forbedre det generative RV-rammeverket, forbedre maskinvareintegrasjon og utvide kompatibilitet til andre ROS-distribusjoner. Resultatene har bidratt til den pågående utviklingen av NASAs RV-verktøy, og fremmer sikrere utplassering av autonome systemer i landbruk og andre domener. Denne avhandlingen er i seg selv et bidrag til metodikken og kunnskapen som er oppnådd for å riktig forstå og bruke Linear Temporal Logic (LTL)-overvåkere.

Contents

Acknowledgments **ii**

Abstract **iii**

Sammendrag **iv**

List of Abbreviations **vii**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Research Question . . . 2

2 Theory & Background **3**
2.1 Runtime Verification . . . 4
2.1.1 Specification Languages, State Machines and Temporal Logic . . . 4
2.1.2 Monitoring Behavior . . . 5
2.2 RV Tools & Architecture . . . 6
2.2.1 FRET . . . 7
2.2.2 OGMA . . . 23
2.2.3 Copilot . . . 25
2.2.4 Flask . . . 26
2.3 Related RV Tools . . . 28
2.3.1 ROSMonitoring Tool . . . 29
2.3.2 TeSSLa-ROS-Bridge . . . 31
2.3.3 ROSRV . . . 32
2.4 ROS & ROS 2 . . . 34
2.5 Robotool . . . 35
2.6 Robot Vision . . . 36
2.6.1 RealSense Depth Camera . . . 36
2.6.2 YOLO . . . 38

3 Requirements **42**
3.1 Goals and Objectives . . . 42
3.2 System Requirements . . . 42
3.3 Safety Requirements . . . 43

4 Methodology **44**
4.1 Implementation of system architecture . . . 45
4.1.1 Installation of software . . . 45
4.1.2 Requirement Logic and Safety Engineering . . . 47
4.1.3 Requirement Declarations . . . 48
4.1.4 Monitor Implementation . . . 51

4.1.5 Flask-ROS Architecture . . . 55
4.1.6 Architecture Performance Optimization . . . 56
4.1.7 WebSocket Implementation . . . 57
4.2 Simulation and Field Testing . . . 59
4.2.1 Field Testing . . . 59
4.2.2 Simulation . . . 59
4.2.3 RealSense Setup with Inferer . . . 63
4.3 Monitor Violation Injection Device . . . 65
4.4 Research Ethics . . . 67

5 Results **68**
5.1 Monitor Behavior . . . 69
5.1.1 System Overhead . . . 69
5.1.2 Asynchronous Publishing . . . 73
5.1.3 Violation Report . . . 75
5.2 Simulation . . . 77
5.2.1 Simulated Robot Data . . . 77
5.2.2 Gazebo Simulation . . . 78
5.3 Field Testing . . . 80
5.3.1 YOLO Field Test . . . 80
5.3.2 Monitor System Field Test . . . 87
5.4 RV Software Tool Comparison . . . 95

6 Discussion **96**
6.1 Results Discussion . . . 96
6.1.1 Monitor Behavior Findings . . . 96
6.1.2 Simulation Findings . . . 100
6.1.3 Field Testing Findings . . . 100
6.1.4 Summary of Key Findings . . . 102
6.2 Comparison with Previous Research . . . 104
6.3 Implications of the Findings . . . 105

7 Limitations **106**
7.1 Software Limitations . . . 106
7.2 Hardware Limitations & Assumptions . . . 107
7.3 Practical Recommendations . . . 107

8 Future Works **108**

9 Conclusion **109**

Bibliography **111**

List of Abbreviations

ROS	Robot Operating System
FRET	Formal Requirements Elicitation Tool
RV	Runtime Verification
OGMA	Operational Goal-Based Mission Analysis
YOLO	You Only Look Once
TeSSLa	Temporal Stream-based Specification Language
ptLTL / pmLTL	past-time metric Linear Temporal Logic
ftLTL / fmLTL	future-time metric Linear Temporal Logic
NL	Natural Language
CPS	Cyber-Physical Systems
FRETISH	FRET restructuring of natural language
NuSMV	New Symbolic Model Verifier
DB	Database
IP	Internet Protocol address
FRP	Functional Reactive Programming
SSH	The Secure Shell Protocol

1 Introduction

With today's steadily evolving robotics market, safety has become paramount with the deployment of robots in human-robot collaborative workspaces. This has become especially true with the seasonal shortages in available manual labor within agricultural areas. The tasks to complete are also often quite repetitive and straining on a human body, furthering the use of autonomous robotic systems [1].

The continuous automation of the workplace leads to the question of how paramount safety will be in the pursuit of efficient production. As John M. Shutske at the Department of Biological Systems Engineering, University of Wisconsin—Madison, shares in his article on "Agricultural Automation & Autonomy": there were fewer injuries and deaths related to horses when we transitioned to tractor-based mechanization, but new issues emerged as farms became more mechanized [2]. The increasing use of Robotic Process Automation (RPA) will likely bring forth new issues that demand for machine learning models and RV (Runtime Verification) systems for maintenance control and the upkeep of worker safety. At the moment, the state-of-the-art is not sufficient anymore when it comes to accuracy and efficiency of these systems in dynamic environments, and a way forward could be to develop better pipelines for RV of systems where the route, task and environment is being analyzed to generate more proficient safety mechanisms.

The idea behind the use of RV is that the utilization of formal verification of a systems realizability leads to more robust safety mechanisms. To translate a system into its mathematical formulas, verify its formal verification and then implement monitors from the results allows the user to know by mathematical proof that all scenarios within the safety margins are acknowledged by the system. One could believe that by writing tests one might be able to have a similar system, but the problem with these tests is that you are limited only to the test coverage [3]. "Ultra-critical systems require high-level assurance, which cannot always be guaranteed in compile time. The use of runtime verification (RV) enables monitoring these systems in runtime, to detect property violations early and limit their potential consequences" [4]. "Since RV does not need to exhaustively check the system behavior, it scales better to real systems, since it does not suffer from state space explosion problems that can be commonly found in model checking" [5]. With the use of formally verified RV, infinite state systems can be efficiently monitored providing property violations when errors occur. If there's a violation, then you trigger either some fault handling mechanism and/or provide the information to the front-end user.

RV and offline Formal Verification (FV) offer complementary approaches to ensuring system reliability and safety, but they differ significantly in their application and capabilities. RV focuses on monitoring the system's behavior during its actual operation, providing immediate detection and response to violations of specified requirements. This real-time oversight is crucial for systems where failures can have severe consequences, such as in agricultural robotics, allowing for prompt mitigation actions like informing users, intercepting erroneous messages, or shutting down the system to prevent harm. In contrast, offline FV involves an exhaustive analysis of the system's design before deployment. This method uses mathematical models to verify that the system meets its specifications under all possible scenarios, identifying potential flaws that can be addressed prior to runtime. While offline FV is thorough and can handle complex state spaces, it can be time-consuming and may not account for all real-world operating conditions. Therefore, combining RV's real-time monitoring with the comprehensive pre-deployment checks of offline FV creates a robust verification framework, enhancing both the reliability and safety of advanced automated systems.

1.1 Problem Statement

Recent advancements in autonomous robotic systems have integrated many tools that can lead to safety-critical environments where hazard mitigation is paramount to adhere to workplace safety guidelines. In this thesis, ultra-violet light is used for plant treatment. This equipment can cause damage to eyes and skin, necessitating increased security for automated use. Without advancements in runtime technology, the field of autonomous robotics will eventually be filled with robotic systems lacking runtime guarantees, which, in the event of safety violations, will not have the tools to implement proper mitigation tactics to ensure safe use. Therefore, there is a growing need for enhanced software control of these Cyber-Physical Systems (CPS) as we advance the automation of various domains.

Since the release of ROS 2 there has been a gap in knowledge related to runtime verification (RV) and its implementation for newer distributions. Tools used for ROS have become incompatible, and there are few resources that clearly demonstrate implementation. This thesis aims to address this gap by providing a clear overview of existing tools, implementation strategies, and exemplifying why RV is a beneficial feature in any CPS application.

The paper ”Monitoring ROS2: from Requirements to Autonomous Robots” ([6]) written by Dr. Ivan Perez at NASA has inspired the problem statement of this paper: To enhance workplace safety and reliability throughout the engineering life-cycle by applying generative runtime verification (RV) to the agricultural domain for ROS 2 applications, turning safety requirements defined in natural language (NL) into formally verified temporal logic monitors.

1.2 Research Question

(1) Is it possible to create a pipeline for the automatic generation of runtime verification monitors for ROS 2 systems using current software tools?

(2) Does the runtime verification monitor help elevate safety during all phases of the engineering life-cycle?

This study aims to investigate whether it is possible to create a pipeline for the automatic generation of runtime verification (RV) monitors for ROS 2 systems using current software tools. While this was previously possible for ROS, the release of ROS 2 has rendered much of the existing software incompatible, raising the question of how to enhance safety in future ROS 2 projects during all phases of the engineering life-cycle. The primary goal of the thesis is to come with a contribution to the robotics environment by developing and testing a RV platform, with the focus on automatic generation of monitors that leads to active hazard mitigation. This builds upon the work done under a summer project directed by PhD student Mustafa Adam and Associate Professor Alireza David Anisi. The system will be tested and analyzed on the RoboFarmer platform developed at The Norwegian University of Life Sciences (NMBU). A secondary goal of the thesis is to compare this contribution to other existing frameworks for runtime verification on similar platforms.

To achieve these goals, we will utilize the methodology proposed in ”Monitoring ROS2: from Requirements to Autonomous Robots” ([6]), employing NASA-developed tools such as Copilot, OGMA, and FRET. This research aims to contribute to the safety and reliability of ROS 2 projects by demonstrating, and comparing, the usability and reliability of these monitors to other frameworks further adding to the research of RV of ROS 2 systems.

2 Theory & Background

In the field of robotics and Cyber-Physical Systems (CPS), the growing complexity and integration of automated systems have necessitated advanced methods for ensuring system reliability and safety. This section of the thesis delves into the theoretical foundations required to fully understand the advancements of Runtime Verification (RV), focusing on the principles and methodologies that are crucial for developing the pipeline proposed by this paper. The theory section conveys key concepts such as runtime verification, specification languages, and the architectural frameworks that facilitate real-time monitoring and compliance checking. Additionally, it introduces the tools used for monitor generation, FRET and OGMA, as well as YOLO, and the model D435 RealSense depth camera contained for the specific use case, which is an autonomous UV-light treatment robot for powdery mildew.

The theory section begins with an in-depth look at RV, (Sec. 2.1), a crucial tool for maintaining system integrity in real-time environments. RV allows for monitoring of a system's behavior against its specifications, thereby enabling the possibility of immediate detection and mitigation of faults or violations. This proactive approach is essential for systems where failure can have significant consequences, such as in agricultural robots.

Next, we explore the core technologies and methodologies that support RV, including specification languages and state machines (Sec. 2.1.1) . These tools provide the formal basis for defining and verifying system specifications, ensuring that they meet their requirements without failure.

Additionally, we discuss various tools and technologies used to implement and support RV, including FRET (Sec. 2.2.1), OGMA (Sec. 2.2.2), Copilot (Sec. 2.2.3), and Flask (Sec. 2.2.4) for monitor generation and user interface. In the related RV tools section (Sec. 2.3), existing software tools for RV are explored, providing a comprehensive understanding of the current state-of-the-art in runtime verification for robotics and CPS.

Later, the integration of runtime verification with the popular framework Robot Operating System (ROS, Sec. 2.4) and its successor, ROS 2, is described in detail. The evolution from ROS to ROS 2 represents a significant shift towards more secure, scalable, and real-time capable systems, which are increasingly necessary in modern robotics and CPS applications. Since the pipeline implementation presented in this paper focuses primarily on ROS 2, understanding some of ROS 2's core concepts is important.

Finally, the tools used for data gathering is addressed in the Robot Vision section (Sec. 2.6). The Intel RealSense depth camera (Sec. 2.6.1), and YOLO tool (Sec. 2.6.2) are used for capturing and analyzing spatial data in real-time environments.

Through this theoretical exploration, this chapter aims to provide a solid foundation for understanding the technologies and methodologies applied in this paper and how to enhance the safety, reliability, and performance of automated systems. The information gathered here has been critical in developing the advanced runtime verification system discussed in the subsequent sections of this thesis.

2.1 Runtime Verification

Runtime verification (RV) is a term that has been growing in popularity over the last few decades, and with good reason. The increased focus on automation and robotics by technology-leading companies raises questions regarding how safety and compliance will be maintained in a steadily advancing automated society. One effective tool for ensuring reliability and safety is implementing RV. RV tools are lightweight formal methods for analyzing the behavior of software or Cyber-Physical Systems (CPS) during runtime [7]. They monitor the specifications of the system during runtime. Thus, in the case of a violation of the specified requirements, a myriad of actions can be performed, such as informing the user, intercepting wrong messages and overwriting them, or implementing mitigation actions like turning the system off in the case of a fault. By implementing RV, the reliability and safety during runtime are increased by decreasing the danger of violations going undetected. RV can eliminate the need for rigorous offline testing and execution traces prior to runtime, instead utilizing formally verified requirements, which aid in solving safety monitoring for systems with infinite state spaces.

In both academia and industry, RV has become more widely deployed across the entire implementation chain, being used from early design phases, through system verification, testing, and during deployment [7], having formal verification at all stages of the engineering life-cycle [8]. The goal is to increase safety and reliability from the moment of conception until deployment. RV continues to prove itself as a beneficial application of formal verification.

When configuring a RV tool, there are three necessary steps [7].

1. Abstraction of (un)desired system behavior in the form of specifications/requirements, ensuring the model of the system is realizable and formally verified.
2. Generation of monitor(s) from model specifications, ensuring an understanding of the monitor logic and its capability to adhere to the specified requirements.
3. Fusing the monitor with the real system by implementing a method of information extraction for monitoring system values. Further development could include hazard mitigation, information interception, and user experience enhancements.

2.1.1 Specification Languages, State Machines and Temporal Logic

This section introduces some of the most used abstractions and languages for system specification. There are numerous ways to specify system behavior, and there is no definitive best method. Understanding the differences between these methods can help in making better choices when choosing or designing a RV tool.

The two most common terms used when discussing specification languages for RV are state machines and temporal logic. These are distinct from each other, and each encompasses numerous variants and combinations that are worth exploring. In the field of temporal logic, the most researched topics are Linear Temporal Logic (LTL), in both past and future time forms, Metric Temporal Logic, Signal Temporal Logic (STL), and Spatial Temporal Logic. Unlike state machines, temporal logic requires monitor

synthesis techniques to produce executable monitors [7]. These monitors are often described similarly to a state machine; thus, state machines have the advantage of being directly executable if the system allows them. For this reason, state machines are often labeled as executable, whereas temporal logic is declarative. By eliminating the need for monitor synthesis, the threshold for using RV is also lowered. For further information on how temporal logic works, see Sec. 2.2.1.

In the context of system traces, one often differentiates between finite and infinite systems. The variants, state machines and temporal logic, have different areas of excellence in this regard. When using state machines, you restrict yourself to a certain number of states and transitions, which creates a finite state space. This works well for testing finite systems with low complexity, but when dealing with more advanced systems that operate over an infinite timeline, temporal logic is often preferred. The expressions of system properties in temporal logic are unbounded by time periods, making it more suitable for infinite state space systems.

2.1.2 Monitoring Behavior

There are four methods of monitoring that are commonly used: offline, online, synchronous, and asynchronous.

Offline monitoring is common in software, where the analysis is carried out after the system execution is complete. Relevant system data is logged during runtime, and the stored data can be modeled or monitored post-execution. Offline monitoring is very useful for scenarios where real-time analysis is not critical for the system and safety is not a concern. This makes offline monitoring less valuable for Cyber-Physical Systems, where safety is paramount. Offline monitoring is also less intrusive to the system and adds very little overhead.

In contrast, online monitoring is executed during system execution, monitoring relevant system events as they occur. This addresses a limitation of offline monitoring, which is that violations can only be reported after the system has terminated. The ability to detect violations at their inception opens possibilities for state mitigation and prevention. While this increases safety and reliability of the system, it also introduces a much larger overhead. Because of the concern with overhead, online monitoring is often done with incremental analysis working under tight constraints. Online monitoring can often be combined with offline monitoring in case there is a wish for further analysis and modeling.

The performance of online monitoring can be achieved either by running a simultaneous or a detached execution. In simultaneous execution, synchronous online monitoring is performed, where the system waits for the monitoring result after each event has been generated. This so-called lock-step approach is an intrusive method that guarantees full control over all event generation. In asynchronous online monitoring, the monitor is detached from the system and operates in parallel. This causes less overhead but often results in a delayed response. Asynchronous monitoring also increases the risk of missing events from the system if the monitor cannot keep up with the publishing speed. Most methods fall somewhere on the spectrum between these two extremes of online monitoring [7].

2.2 RV Tools & Architecture

Proposed by the paper ”Monitoring ROS2: from Requirements to Autonomous Robots” by Dr. Ivan Perez, the FRET/OGMA/Copilot framework provides a streamlined workflow for generating and implementing runtime verification monitors [6]. The workflow begins with user defined system requirements using FRETISH with the FRET tool, see Fig. 1. The requirements are then converted into temporal logic specifications, formally verified, and exported. Then, the formal specifications are exported to OGMA, which translates them into Copilot-compatible code. OGMA generates the necessary C99 monitor code and ROS 2 nodes for data collection and violation reporting, making it possible to utilize the Haskell Copilot language without any prerequisites. The generated C99 code is compiled and integrated into the target system, where the ROS 2 nodes handle real-time data collection and monitor the system's behavior. Ensuring that any violations are promptly detected and reported to the Flask website.

By combining these tools, the framework ensures that system requirements are accurately captured, formally verified, and continuously monitored during runtime. Fig. 1 and Fig. 2 display how this proposed architecture would work. Fig. 2 being in the integration with Robotool, while Fig. 1 shows the general tool-chain to automatically generate monitors for ROS 2, acquired from Dr. Ivan Perez's paper [6].

In this section the RV tools FRET (Sec. 2.2.1), OGMA (Sec. 2.2.2, Copilot (Sec. 2.2.3) and Flask (Sec. 2.2.4) are introduced and explained in depth. Providing the necessary information required for utilizing set tools, and understanding how they function.

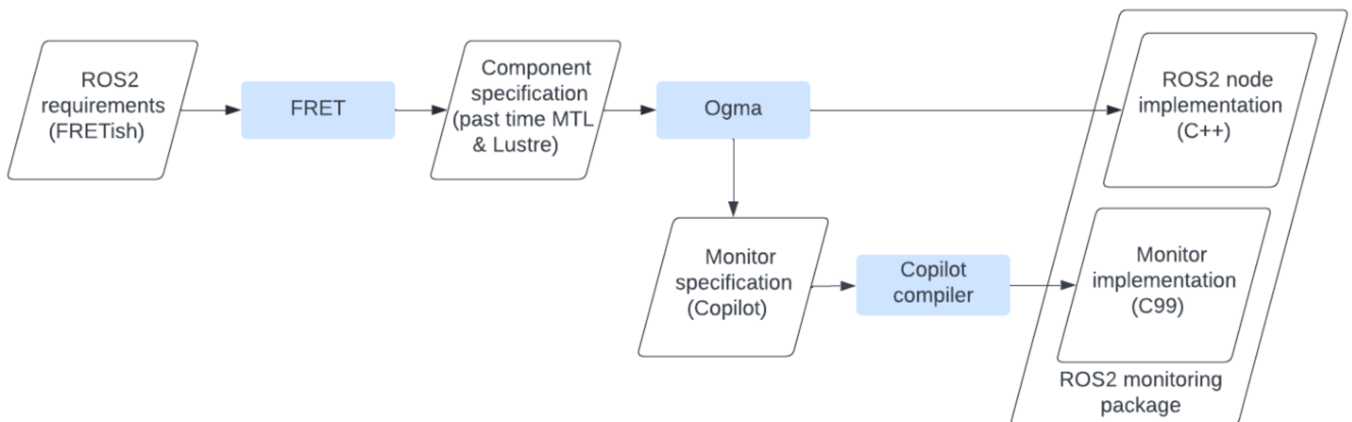


Fig. 1: Toolchain proposed by NASA for automatically generating monitors for ROS 2. *Note.* From ”Monitoring ROS2: from Requirements to Autonomous Robots”, by Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alexander Will, and Patrick J. Martin, 2022, *Electronic Proceedings in Theoretical Computer Science*, vol. 371, p. 210 [6].

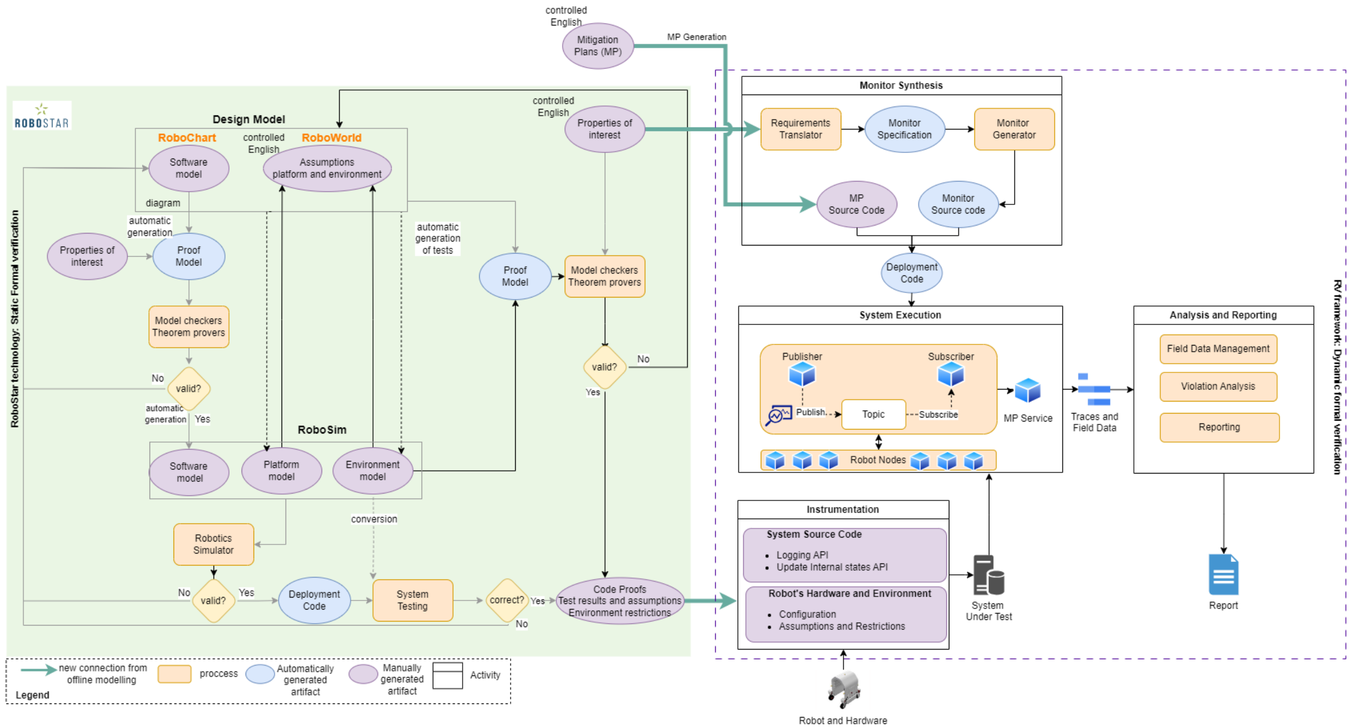


Fig. 2: Monitor synthesis architecture integrated with Robotool. *Note.* From ”Safety assurance of autonomous agricultural robots: from offline model-checking to runtime verification”, by Mustafa Adam, Elias E. Hartmark, Tage Andersen, David A. Anisi, and Ana Cavalcanti, 2024, p. 3 [8].

2.2.1 FRET

As robotics technology advances, user experience becomes crucial in simplifying robot control. FRET is a tool designed to lower the threshold for defining system control requirements. Developed by NASA, FRET is a tool for writing, understanding, formalizing, and analyzing requirements, focusing on allowing users to write in an “intuitive, restricted natural language called FRETISH” [9]. Here natural language refers to any language that develops naturally through use. From this restricted natural language, FRET creates an explanation of the requirement’s meaning, formalizes it, and can create an interactive simulation of the logic. The simplicity of FRETISH also facilitates the easy conversion of a configuration file, containing variables and their requirements, into past-time metric temporal logic formulas. FRET includes an ’import’ function to achieve this, which is particularly appreciated in frameworks for generating runtime monitors.

Initially, FRET was limited to the teams at NASA, but recently the tool has been made available as open source. This has led to valuable feedback from both industry and the research community, which is crucial since it is used in safety-critical contexts. The emphasis on ensuring the correctness of requirement formalization makes it an excellent tool for automatically generating safety monitors for automated systems.

FRETISH requirements are composed of six fields of information: scope, condition, component*, shall*, timing, and response*. The fields marked with * are mandatory for any requirement to be func-

tional. Although not all fields are mandatory, it is generally recommended to use most of them to get a comprehensive requirement. Fig. 3 illustrates how the FRET fields are marked in the app using color, and the resulting transformation from NL to ptLTL.

When creating a FRETISH requirement, follow the fields definition and use the accepted commands. This information can be found in the FRET app or in Tab. 1. The table compiles some of the field information from FRET along with the accepted commands [10].

FRET Specification

NL: ”While in drive mode, when the speed is larger than 10 m/s, make sure the brakes are experiencing a force equal to or above 50 Newtons”

FRET_{ish}: In drive mode if speed > 10, the brakes shall within 1 second satisfy (brake_force ≥ 50)

ptLTL:

```
((H (((! drive) & (Y drive)) -> (Y (((O[1,1] (((speed > 10)
& ((Y (! (speed > 10))) | (drive & (Z (! drive))))) & (!
(brake_force >= 50)))) -> (O[0,0] ((drive & (Z (! drive))) |
(brake_force >= 50)))) S (((O[1,1] (((speed > 10) & ((Y (! (speed
> 10))) | (drive & (Z (! drive))))) & (! (brake_force >= 50))))
-> (O[0,0] ((drive & (Z (! drive))) | (brake_force >= 50)))) &
(drive & (Z (! drive)))))))) & (((! ((! drive) & (Y drive)))
S ((! ((! drive) & (Y drive))) & (drive & (Z (! drive))))) ->
(((O[1,1] (((speed > 10) & ((Y (! (speed > 10))) | (drive & (Z
(! drive))))) & (! (brake_force >= 50)))) -> (O[0,0] ((drive
& (Z (! drive))) | (brake_force >= 50)))) S (((O[1,1] (((speed
> 10) & ((Y (! (speed > 10))) | (drive & (Z (! drive))))) &
(! (brake_force >= 50)))) -> (O[0,0] ((drive & (Z (! drive))) |
(brake_force >= 50)))) & (drive & (Z (! drive)))))))
```

Fig. 3: A simple example of a translation from NL to pmLTL using FRET. The specification becomes increasingly complex when time steps are involved in the requirement.

When creating FRET requirements, ensure that the variables are not under-specified, as OGMA may not be able to generate a functioning monitor script from the output. To check this, open the JSON file exported from FRET and verify that it contains variable information and their types. If not, add this information before exporting from FRET. Variables that have already been created can be found in the Glossary on the FRET create page, in the app [10]. To check if variables in FRET are undefined, press the edit requirement button and use the glossary listed there. FRET features both a diagram and a simulation function. While the simulation requires additional software, the diagram works without any additional programs. The diagram schematic displays the logic of the FRETISH requirement with different colored blocks. The meanings of these blocks can be found under diagram semantics shown in Fig. 4 with an example shown in Fig. 5.

Table 1: FRET Requirement Descriptions [10]

Requirement Description	Mandatory	Definition	Accepted Commands
Scope	No	Specifies where the requirement must hold: in intervals defined with respect to a MODE.	before, only before, in, not in, only in, after or only after MODE
Condition	No	Specifies the condition after which the response shall hold, taking into account scope and timing.	whenever, upon, when, where or if a condition is true
Component	Yes	Specifies the component of the system that the requirement applies to.	For example: front_right_motor or UV_Light
Shall	Yes	Specifies the beginning of the timing response to the requirement	SHALL
Timing	No	Specifies the time points or time intervals, where a response has to occur once scope and condition(s) are satisfied.	immediately, at the next time point, eventually, always, never, within, for, after, until, before.
Response	Yes	Specifies the response that the component must provide to fulfill the requirement.	satisfy BEXP (Boolean Expression)

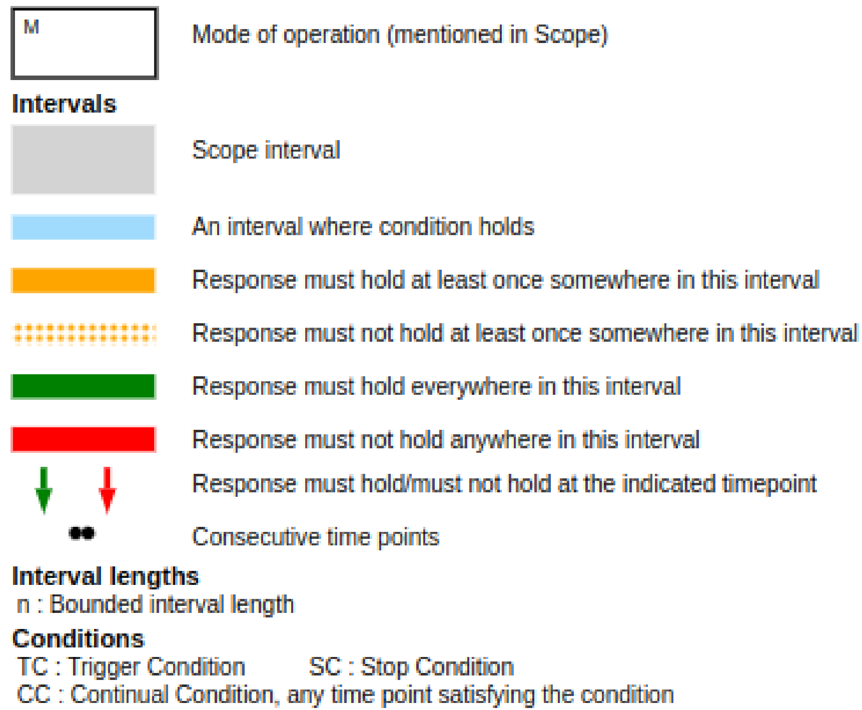


Fig. 4: Diagram semantics, for use with Fig. 5.

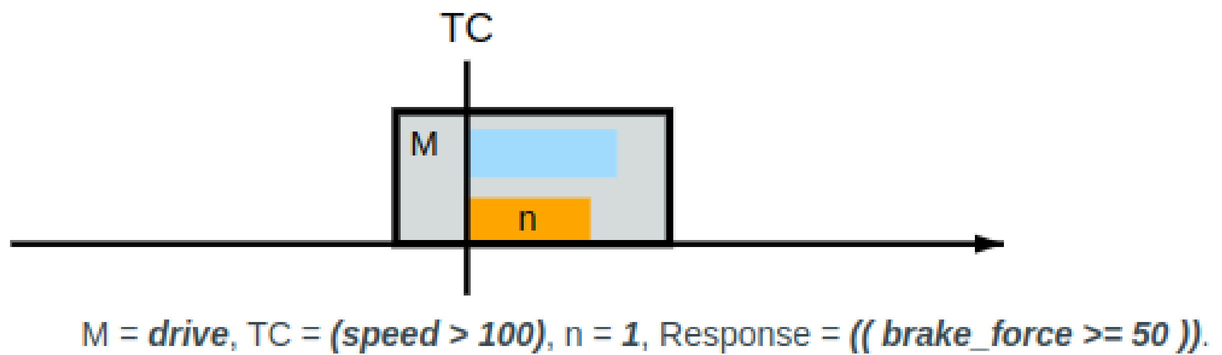


Fig. 5: Semantics Diagram based on the example in Fig. 3.

In Tab. 2, you will find the boolean operators of FRETISH. These are ”allowed in the expression of *while* scope, condition, *until* and *before* timings, and response fields” [10].

Table 2: Boolean Operators [10]

Operator	Definition	Use Case
!	not	*Negation:* The opposite or invertion of a positive value.
&	and [Conjunction]	*Boolean Comparison:* Returns True only when both values are True.
|	or [Disjunction]	*Boolean comparison:* Returns True as long as one of the Booleans are True.
xor	exclusive or	*Boolean comparison:* If two Booleans differ, the result is True. Otherwise it's False.
-> or =>	implication	*Statement:* If p then q
<-> or <=>	equivalence	*Statement:* If p then q, and likewise if q then p.

FRET Architecture

The FRET software is a JavaScript implementation which uses the Electron JS app as a framework for the desktop-suite application [10][9]. Fig. 6 and Fig. 7 display the inner workings of FRET, while Fig. 8 displays a normal workflow when using FRET. It is recommended to follow a similar workflow in order to achieve a realizable model robust enough for deployment.

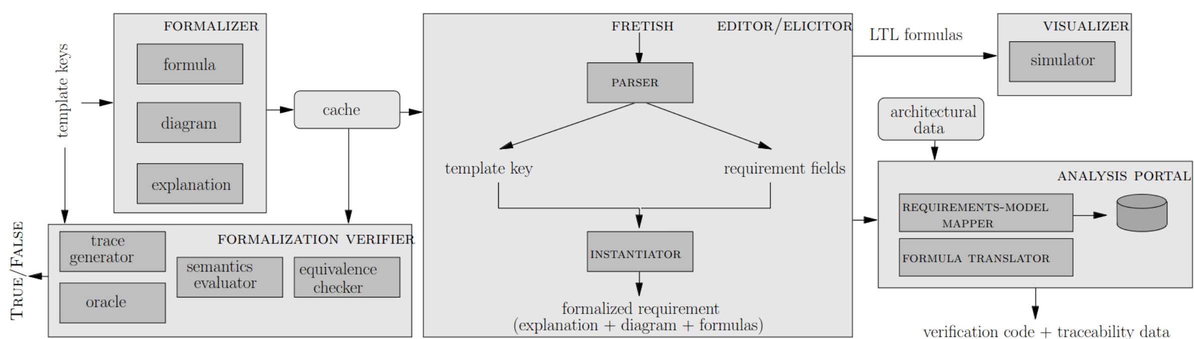


Fig. 6: Architecture of the FRET program [9]. *Note.* From "Formal Requirements Elicitation with FRET", by Dimitra Giannakopoulou, Anastasia Mavridou, Julian Rhein, Thomas Pressburger, Johann Schumann and Nija Shi, 2020, p. 5 [9]. CC BY 4.0.

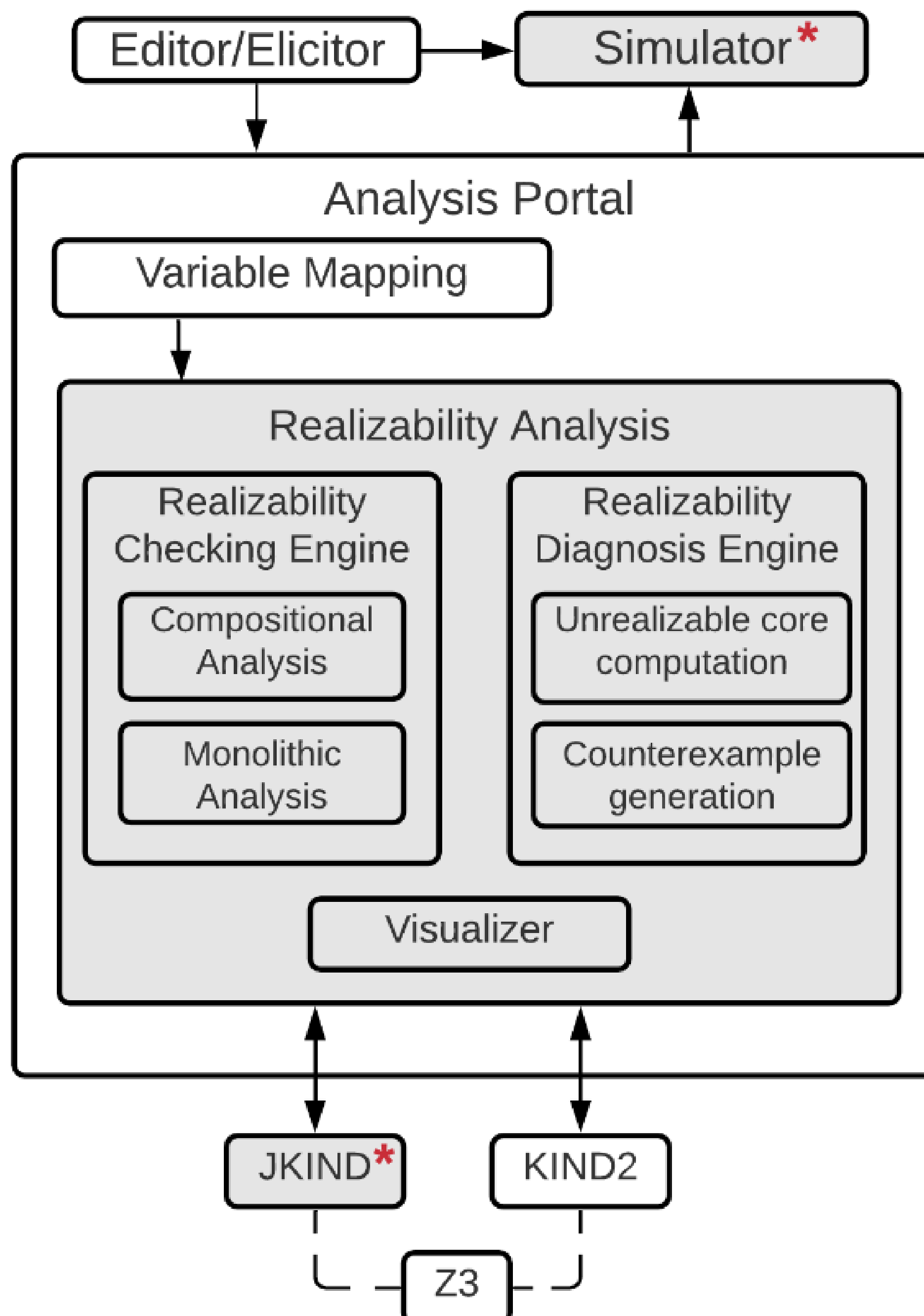


Fig. 7: Architecture of the FRET simulation extension. *Note.* From "Realizability Checking of Requirements in FRET", by Andreas Katis, Anastasia Mavridou, Dimitra Giannakopoulou, Thomas Pressburger, and Johann Schumann, 2022, p. 16 [11].

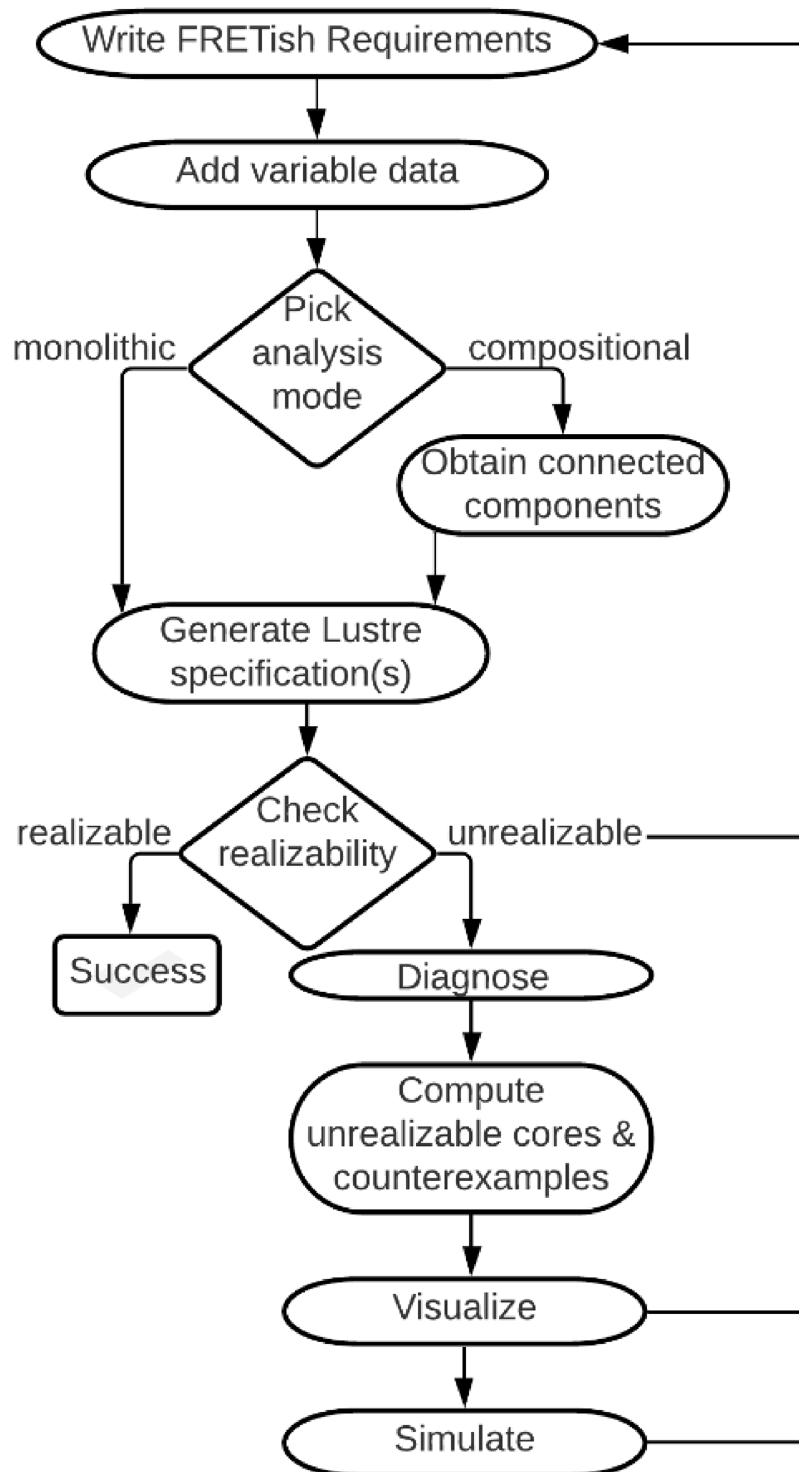


Fig. 8: Workflow for using FRET. *Note.* From "Realizability Checking of Requirements in FRET", by Andreas Katis, Anastasia Mavridou, Dimitra Giannakopoulou, Thomas Pressburger, and Johann Schumann, 2022, p. 16 [11].

Temporal Logic

FRET provides two different types of temporal logic when correctly applied, Past-time Metric Linear Temporal Logic (pmLTL) and Future-time Metric Linear Temporal Logic (fmLTL). The terms past/future and Metric define what formulas can be expressed and how they can be used and evaluated. When implemented with a stream language like Copilot, the time component is essential for determining how the monitor should behave. Temporal logic is particularly suitable because it provides ”... a suitable mechanism to express many of the re-occurring patterns in monitor specifications.” with respects to real-time execution [4].

The important distinction of Past-time Linear Temporal Logic (ptLTL) from Propositional Logic (PL) is that, while in PL ”every variable may take the value true or false, in ptLTL, every variable may take the value true or false at each point in the present or in the past [4]. ptLTL introduces temporal operators that allow logic formulas to consider not only what is true and false at the present, but also past validity. A simple example provided by NASA shows its utility: suppose the boolean p can be either true or false, but the logic needs to check that p has always been true. This requires considering the statement's validity in both the present and the past. The difference with ftLTL (Future-time Linear Temporal Logic) is that it considers future states instead of past ones. The appropriate LTL should be applied based on the monitor requirement.

Metric is an extension on the linear LTL structure, adding the constraint that all system properties must be satisfied within certain time frames.

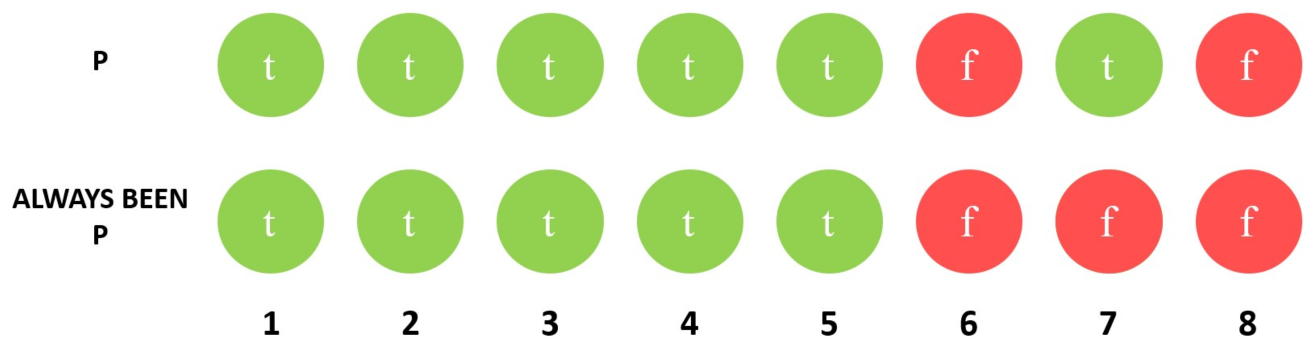


Fig. 9: Example illustrating ptLTL's ability to verify the validity of a requirement from past and present values. (f = false, t = true). *Note.* Adapted from ”Copilot 3”, by Ivan Perez, Frank Dedden and Alwyn Goodloe, NASA, 2020, p. 15 [4].

In Fig. 10 three temporal logic examples have been made for the following definitions:

- **ptLTL:** Deals with conditions over past states without explicit time intervals. It is useful for checking if a condition has always been true in the past.
  Explanation: The value of counter has always been less than or equal to 4 in the past.

- **MTL:** Adds timing constraints to the conditions, allowing for specifying time-bounded properties over past or future states.
  Explanation: Over the past 3 time units, the temperature has always been at least 100, and over the past 40 time units, the airspeed has always been at least 100.

- **BLTL:** Focuses on bounded time intervals for both past and future states, making it useful for specifying time-bound conditions and guarantees.
  Explanation: The airspeed was at some point less than 100 within the past 100 time units, and it has been consistently 100 or more within the past 10 time units.

I. Past-Time Linear Temporal Logic

```
prop = PTLTL. alwaysBeen ( counter <= 4)
```

II. Metric Temporal Logic

```
prop2 = (MTL. alwaysBeen 0 3 (temperature >= 100))
&& (MTL. alwaysBeen 0 40 ( airspeed >= 100)
```

III. Bounded Linear Temporal Logic

```
recover = (BLTL. eventuallyPrev 0 100 ( airspeed < 100))
&& (BLTL. alwaysBeen 0 10 ( airspeed >= 100)
```
[12]

Fig. 10: Temporal Logic examples. *Note.* Adapted from ”Runtime Verification with Ogma”, by Ivan Perez, NASA, 2023, p. 21 [12].

It should be duly stated that the results from exporting a specification model, JSON file, may vary depending on installed dependencies. According to NASA's GitHub page for FRET, the installation instructions specify these dependencies [10]:

- NodeJS (use any version between v16.16.x - v18.18.x)
- Python (Version $\geq$2. Note: if you are having issues using 3.11 or later, use 3.10)
- (Optional) NuSMV
- (Optional) JKind
- (Optional) Kind 2
- (Optional) Z3

Several of the mentioned dependencies are listed as optional. This can be misleading because, while FRET will install and launch correctly if the given instructions are followed, issues may arise. Especially regarding the use of the realizability tools and simulation, most of these functions are only available when all dependencies are downloaded.

Variable Mapping

The first step in variable mapping is to ensure that you have created or imported a FRET project. Next, in the FRET portal, enter the FRET Analysis portal, which can be found in the left-hand side menu. Your project with requirements and variables should be selected from the drop-down menu at the top, revealing a new set of drop-down menus: one for export language and one for components. After selecting a component, all its variables will appear in an interactive table below. Before or after mapping your variables in the Variable Mapper, you get to choose between exporting with CoCoSpec or CoPilot as exporting languages. CoCoSim, is an analysis tool much like Copilot, but is in it's own right a greater automated framework than Copilot and focuses on Simulink/Stateflow. Simulink and Stateflow are graphical languages with diagrammatic environments implemented in Matlab [13]. For this paper, Copilot will be set as the export language, but regardless of the choice, mapping your variables is necessary to allow exporting your requirements and variables.

All information on how to use the FRET Analysis portal can be found on their github under fret/fret-electron/docs/_media/ExportingForAnalysis/analysis.md or in the FRET desktop application, found by clicking the "HELP" button in the Variable Mapper.

The Variable Mapper allows for defining the model variable name, variable type, data type, and description for each variable in a FRET project. Variables can be assigned types such as Input, Output, Internal, and Mode.

The act of creating the variables define the constraints that the requirements impose over the system. Proper typing of variables is crucial, and matters significantly, for should one only define Input variables, the requirements would impose constraints only over system inputs.

Conceptually, some requirements can be thought of as assumptions that we make regarding the system's environment. Currently FRET does not have a way to automatically "flag" such requirements as assumptions for realizability checking, which can cause non-realizable models. Therefore a manual way of adding assumptions has been implemented. By adding the keyword "assumption" somewhere in the requirement ID, for example by renaming "RV-001" to "RV-001-assumption", FRET will consider it as an assumption of the model specification.

This matters because, if an assumption is not properly flagged as such, the requirement will be considered as a constraint that the system must satisfy. In this case the realizability analysis will try to determine whether there is any way to directly control the value of the input variables that appear in the requirement. For assumption statements, this can never happen, as input values are provided from an external source, over which the system has no control.

Realizability Checking

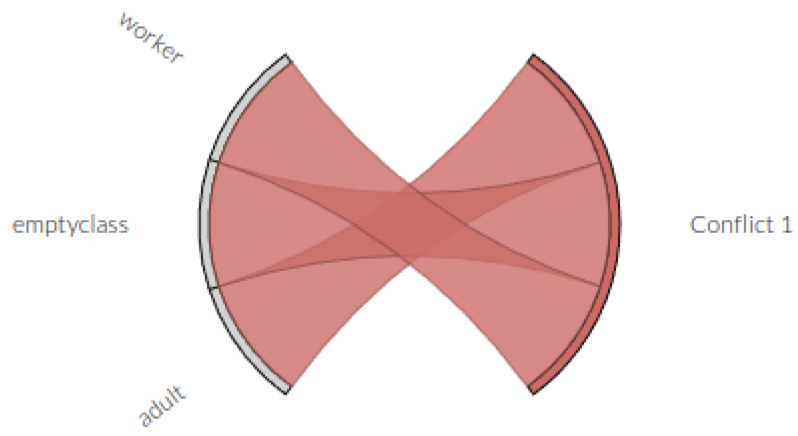
FRET offers a tool for validating the system model based off the written requirements and variable mapping. The realizability interface provides four different configurations for realizability checking: ”JKind,” ”JKind + MBP,” ”Kind 2,” and ”Kind 2 + MBP.” MBP stands for Model Based Projection. The ”MBP” configurations tend to be inferior in performance overall but may solve problems that standard quantifier elimination cannot [10].

On the LTLSIM simulator, four major elements are listed:

- Control buttons and menu
- requirements field
- variable traces
- output traces

Reiterating on Sec. 2.2.1, an important note to be aware off when creating the FRETISH requirements is that by adding the keyword “assumption” somewhere in the requirement ID, you are disclosing that the requirement is not a constraint [10]. Failing to correctly define an assumption will result in an ”UNREALIZABLE” outcome, meaning that a system cannot be implemented to meet the specified requirements. This indicates that no implementation can conform to the output variable constraints given any input. Non-assumption requirements will be considered constraints that the system must satisfy, and the realizability analysis will attempt to determine whether the system can control the input variables. As input values are provided externally and cannot be controlled by the system, the set of requirements will be declared “UNREALIZABLE.” Assumptions are often necessary to achieve ”REALIZABLE” results in FRET. In Fig. 11 are examples of how FRET visualizes the system when the specification is setup incorrectly, resulting in a unrealizable result.

There are two ways of conducting realizability checking in FRET: monolithic and compositional. Monolithic is the default option, providing a formal analysis of a single component or parent group. Compositional, on the other hand, allows the analysis tool to decompose components into smaller, connected pieces where requirements are expressed across all connected components. This provides a powerful tool for compositional formal verification, offering a comprehensive view of all dependencies [14].



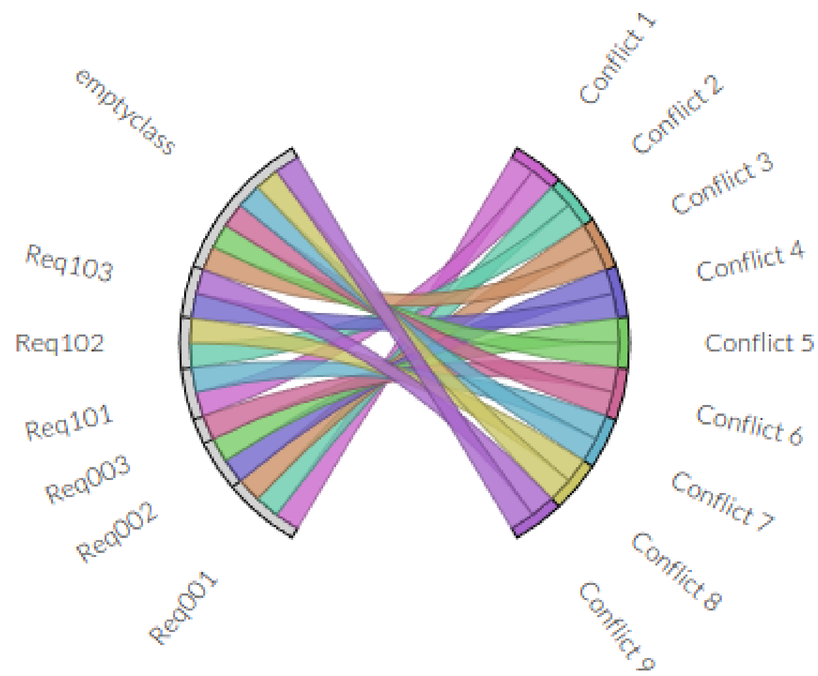


Fig. 11: Visualization example of unrealizable results.

Lustre

Lustre is a synchronous dataflow programming language designed for modeling and verifying reactive systems. It provides a concise and expressive syntax to define how data flows between different system components over time. The language is particularly well-suited for applications in control systems, real-time monitoring, and embedded software, where precise and predictable system behavior is crucial [14][15].

Lustre plays a significant role in the workflow of formal requirement verification tools like FRET. The integration of Lustre into FRET's framework allows for seamless transition from high-level requirement specifications to detailed formal models that can be used for verification and synthesis [11].

Kind-2, Model Realizability

FRET supports the following programs for realizability checking of FRETISH requirements in Linux: Jkind, JKind + MBP (Model Based Projection), Kind 2, and Kind 2 + MBP. The role of the realizability software is to perform a formal analysis of the system requirements, determining whether the safety properties are provably realizable or not [15, 14]. Choosing between the realizability software may impact performance, but the differences are generally not significant.

"Kind 2 is an open-source, multi-engine, SMT-based model checker for safety properties of finite- and infinite-state synchronous reactive systems" [15]. Kind-2 translates LTL safety properties expressed in Lustre into an encoded format based on three state transition systems: $s, I(s), T(s, s')$. Here, s represents the vector state variables at any given time, $I(s)$ the initial state of the system, and $T(s, s')$ the properties of the transition between two states (where s is renamed as s') [15]. Kind-2 also supports invariant properties, which are variables or conditions that remain unchanged even after the system state changes. These invariant properties, denoted as P, hold in any reachable instance of the system.

When compared to previous realizability models, it was concluded that "Kind 2 is very competitive with its peers, outperforming its predecessor PKind and providing an answer (either valid or a counterexample) in more cases than any other tool" [15]. In Fig. 12 the performance of different model checkers have been compared, showing the number of benchmarks solved in relation to time.

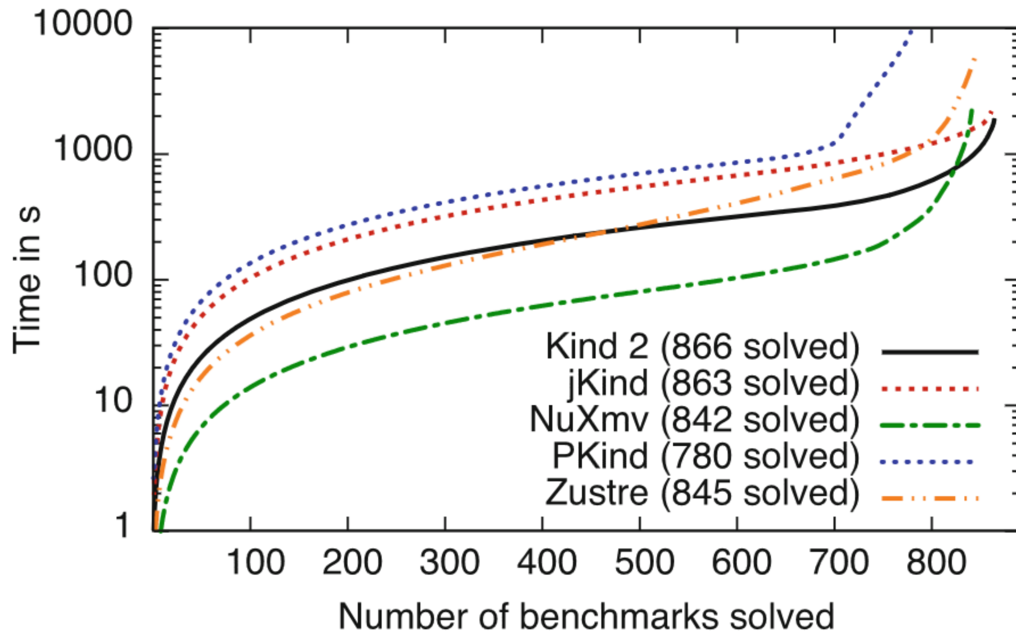


Fig. 12: Comparison between realizability models (infinite-statemodel checkers). *Note.* From "The kind 2 model checker", by Adrien Champion, Alain Mebsout, Christoph Sticksel and Cesare Tinelli, 2016, *International Conference on Computer Aided Verification*, p.515 [15].

Jkind, Model Realizability

JKind, developed by Rockwell Collins and the University of Minnesota, is an "open-source industrial model checker"[16]. Model checkers are used to prove or falsify safety properties of infinite-state system models. Unlike finite-state models, which can often be too restrictive, infinite-state models handle a boundless number of possible states, which can lead to significant computational consequences [17].

To manage this complexity, JKind uses multiple parallel engines in the form of SMT-solvers (Satisfiability modulo theories) for its verification process. Where the SMT results are used by JKind for traceability of properties and model elements. The emphasis on the usability of results distinguishes JKind, making it essential for simulating unrealizable results in FRET. This is made possible since "For a falsified property, JKind provides options for simplifying the counterexample in order to highlight the root cause of the failure" [16]. Information about the root cause of failures can be as valuable as the results of the realizability testing, leading to a better workflow for model requirements. Written in Java, JKind uses SMTInterpol but can be configured with Z3, YICES 1, YICES 2, CVC4, and MathSAT [16]. For FRET, Z3 is often the primary option.

JKind provides a smoothing algorithm when combined with Z3 or YICES. This is a post-processing step that minimizes the number of input changes in its counterexample, making system faults more readable for the user. The smoothing process adds 40% of runtime, but typically removes more unnecessary changes per test case [16]. In FRET this might be useful when applied to complex model verification issues, where Kind-2 will outpace the JKind solution.

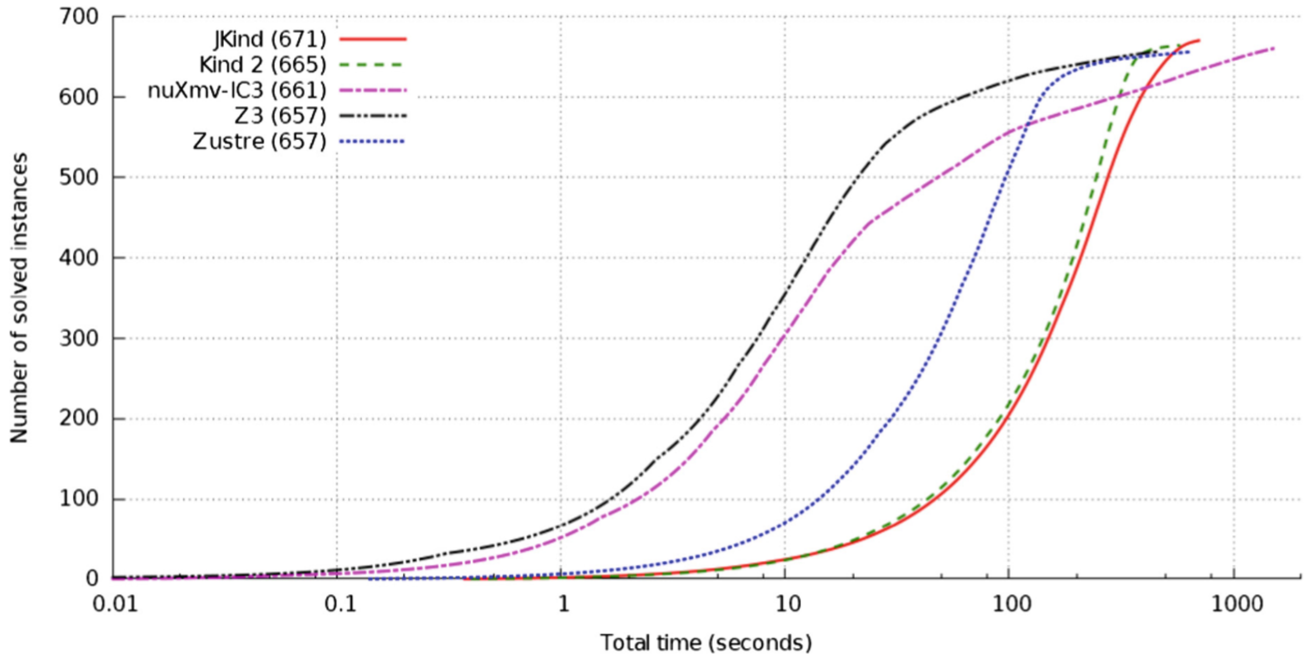


Fig. 13: Comparison between infinite-state model checkers. Note that the axes are flipped compared to Fig. 12. *Note.* From ”The jkind model checker”, by Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani, 2018, *Computer Aided Verification: 30th International Conference*, p. 23 [16].

Model Realizability Summary

When comparing the performance of JKind and Kind-2 in Fig. 12 and Fig. 13, there is no definitive choice of the better model checker. Each has its advantages, but in most situations, JKind and Kind-2 will perform well. Of the two, Kind-2 is the faster model checker but offers fewer functions in the FRET software. JKind, on the other hand, is built in Java and is slower for smaller models but excels in cases where the models are unrealizable, as it displays the root cause of the failure and applies a smoothing algorithm.

FRET Simulation

FRET offers a simulation tool to test the logic of the LTL requirements. Provided Kind-2 is installed, this tool will be available when the system is unrealizable. With Jkind, the simulation tool is also available for realizable systems. This tool allows users to set up scenarios where the logic of system transitions is evaluated, visualizing the results clearly and providing a good perspective on how FRETISH logic functions. Fig. 14 shows an example of this scenario tool, where pressing the small circles changes the state of the variable. To use JKind or Kind-2, NuSMV must be installed. NuSMV (New Symbolic Model Verifier) is a ”symbolic model checker”, which FRET utilizes for its verification software [18] [19]. The results are visualized using JKind or Kind-2.

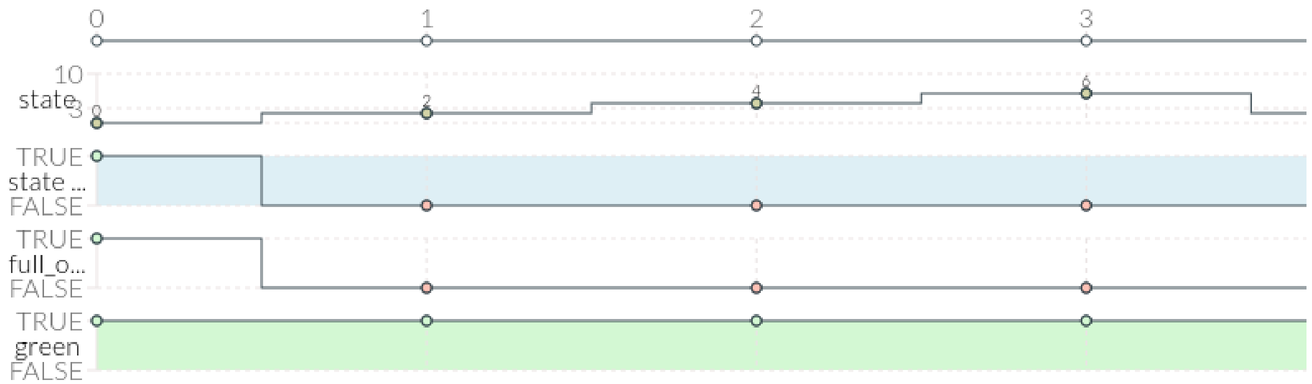


Fig. 14: Example of how FRET visualizes LTL logic after realizability testing using JKind.

Exporting and Importing

Effective management of requirements and system models involves seamlessly transferring data between different tools and platforms. FRET supports a streamlined process for importing requirements from other tools and exporting them into various formats for further analysis and implementation. This functionality ensures that the requirements are consistent, up-to-date, and easily integrated into different phases of the system development lifecycle.

FRET provides robust capabilities for importing requirement specifications from other tools, allowing for a smooth integration of existing data into the FRET environment. One such tool commonly used for developing robotic applications is Robotool. From Robotool a JSON or CSV file of a system model could be exported, and used for requirement generation.

After specifying and formalizing requirements in FRET, you might need to export them into other tools for further development and verification. OGMA, a tool used for generating runtime monitors and analyzing system behaviors, is one such example where FRET requirements can be exported, using the Copilot language.

2.2.2 OGMA

OGMA was created by NASA to bridge the gap between FRET and the RV system Copilot [20]. Leading to an automated process from structured natural language input called FRETISH to the generation of C99 monitor code that can be implemented in Copilot. OGMA translates requirements to a precise mathematical formalism that could be used in safety-critical systems. In this case, safety-critical systems are processes where failure can result in injury or death of a human [20]. To prevent safety-critical situations, a crucial aspect of the monitors generated from OGMA is that they perform in hard real-time, thereby ensuring that any reaction to a property violation does not miss the activation deadline. Hard real-time code lends itself well to RV of advanced automatic systems where failures could lead to safety-critical events.

For ROS applications, the following works as input for OGMA:

Table 3: Command Line Arguments Description. *Note.* Adapted from OGMA GitHub by NASA [21].

Argument	Description
`--app-target-dir DIR`	Location where the ROS application files must be stored.
`--variable-file FILENAME`	File containing a list of variables that must be made available to the monitor.
`--variable-db FILENAME`	File containing a database of known variables, and the topic they are included with.
`--handlers FILENAME`	File containing a list of handlers used in the specification.

Table 4: List of Produced Files. *Note.* Adapted from OGMA GitHub by NASA [21].

File Path
`ros_demo/CMakeLists.txt`
`ros_demo/src/copilot_monitor.cpp`
`ros_demo/src/copilot_logger.cpp`
`ros_demo/src/.keep`
`ros_demo/package.xml`

Not all arguments from table 3 are mandatory for OGMA to work. On the GitHub page, it is stated that ”You should always provide either a FRET component specification, or both a variable file and a handlers file.” [21]. In this case, the FRET component specification is the most straight forward to use. In the case where both the FRET file and the variables/handler files are provided, OGMA will prioritize the variables/handlers.

Although the DB (database) file is not mentioned as mandatory, it is needed for any subscriber and publisher nodes to be generated in the ROS directory. This is the case since the DB file is where the ROS topics are declared and from which the monitors will subscribe to. The DB file requires four inputs in order, namely: variable name, variable datatype, ROS topic, and ROS topic datatype. An example of what these inputs might look like is in Tab. 5. OGMA will not produce any warnings if a DB file is not provided, and the same goes for variables that are not located in the DB file. For this reason, it is important to make sure all required fields are declared before advancing.

Table 5: ROS DB-file example for OGMA.

Variable	Variable Datatype	ROS Topic	Topic Datatype
distance_to_target	int64_t	/scan	Int64_t
classifier	int64_t	/sRobotClassifier	Int64_t
alert	bool	/sRobotAlert	bool
halt	bool	/sRobotHalt	bool
slowdown	bool	/sRobotSlowdown	bool
state	int64_t	/sRobotState	Int64_t
turnoffUVC	bool	/sRobotTurnoffUVC	bool

A current limitation of OGMA is that the ROS command does not produce the copilot monitors which are required for the ROS structure to work. To address this, the Copilot-compiled 'monitor.c' and 'monitor.h' files need to be placed in the 'src' of the ROS directory [21].

Haskell

There are many software packages available to use when looking into applying RV to ROS systems. One deciding factor for using OGMA, which uses Copilot, is because they are built in Haskell. In this section, the advantages of Haskell as a language for RV monitor generation are presented.

Dr. Ivan Perez, Senior Research Scientist at NASA and one of the lead programmers behind OGMA, stated when discussing Haskell that "Low-level languages may be more error-prone, and some classes of errors are easier to make. A high-level, safe language can facilitate readability and maintenance, and limit the likelihood of introducing (some) bugs." [12]. To explain this choice of programming language, the following features of Haskell are presented: immutability, statically typed (strong type system), higher-order functions, lazy evaluation, and referential transparency [22].

Immutability: Haskell enforces immutability, a method where data structures cannot be modified after they are created. This feature simplifies code clarity, reasoning, and bug fixing since the history of changes can be represented as new values rather than mutations of existing data, providing a history of changes as they happen within the software.

Statically typed (strong type system): Haskell is known for its statically typed and strong type system, which means that the types of all expressions in a program are known at compile time, and the language enforces strict rules on how types can interact. This characteristic of Haskell plays a crucial role in ensuring program correctness and reliability, especially in the context of Functional Reactive Programming (FRP). This system acts as a sort of guarantee since "All the types composed together by function application have to match up. If they don't, the program will be rejected by the compiler" [22].

Higher-Order Functions: A fundamental part of Haskell is its support for higher-order functions. Higher-order functions take other functions as arguments or return a function as a result, which can be beneficial for constructing and combining reactive behaviors in an FRP system [23]. This feature is not special to Haskell, but the type system further enhances the power and safety of these functions.

Lazy Evaluation: Haskell enforces a lazy evaluation model. Lazy evaluation means that computations are deferred until their results are needed, meaning that arguments are evaluated only when the values are actually used. This fits well with FRP models of data streams, as it allows for efficient, on-demand computation of reactive expressions. This also benefits the higher-order functions by making them compose together well, "Haskell code makes it easy to fuse chains of functions together, allowing for performance benefits" [22].

Referential Transparency: A feature mentioned by Dr. Ivan Perez when commenting on the advantages of Haskell is referential transparency. This means that an expression, when evaluated, will always produce the same result without causing any abnormal effects. This property is crucial for testing because it ensures that given the same inputs, a function will always return the same outputs as if it was always running on the same seed, making tests reliable and predictable. The primary purpose of referential transparency is to make code more predictable, understandable, and easier to reason about by ensuring that functions are behaving as expected. Dr. Ivan Perez stated the following in a podcast regarding how NASA uses Haskell for RV: "And we can actually use some of Haskell's assets to our advantage or some of the best-selling points of Haskell to our advantage. For example, when it comes to testing, we can claim that because we run certain tests, we have the guarantee from the compiler that if we run the same test cases again, we will get the same result. This is something that you would not get in another language." [3].

2.2.3 Copilot

OGMA is built around Copilot which is a runtime monitoring language. Copilot is a stream programming language, and that's where it connects to FRP (Functional Reactive Programming) [3]. The relation between OGMA and Copilot is that OGMA translates the incoming property requirement and feeds it into Copilot which generates C code monitors. OGMA would for instance give a Haskell file that "when you compile it, it generates hard real-time C99" [3] and "The C99 backend ensures us that the output is constant in memory and time, making it suitable for systems with hard realtime requirements." [24]. OGMA simplifies this process by being able to translate other languages to Copilot, which works since the difference between Copilot and other libraries Copilot uses like Lustre and temporal logic are very small. Note that Copilot made by NASA has nothing in common with the GitHub Copilot extension tool [25].

The Copilot software developed by NASA is a stream-based runtime monitoring language [20]. Made with the intent of creating RV (runtime verification) frameworks for real-time embedded systems. RV has been increasingly sought after because of advancements in the robotics fields that require high-level assurance, and both the Copilot and OGMA software is under active development under NASA [4]. The goal of Copilot is to "automatically generate C monitor programs from a high-level DSL embedded in Haskell." [26].

Copilot is implemented as a Haskell embedded domain-specific language (EDSL), designed and structured within the Haskell programming environment, a programming language with a strong performance in static typing and mathematical precision [4]. Through the Haskell language, Copilot monitors operate on streams. Streams are infinite successions of values that are being monitored by the system to ensure

everything is as it should be. Streams are beneficial to real-time monitoring, keeping high accuracy when it comes to timely response and tracking of risk assessment. ”Copilot supports the standard logic operators from propositional logic, as well as temporal combinators based on temporal logics” [4].

Since Copilot is manufacturing the monitors a degree of assurance that the result is correct is important. For this, Copilot Verifier was created by Ryan Scott at Galois [27]. Copilot Verifier is ”a tool that establishes a format proof between the C code that comes out of Copilot and the denotational semantics of the language.” [3]. Verification of the C code provides evidence of the safety of the product Copilot has generated.

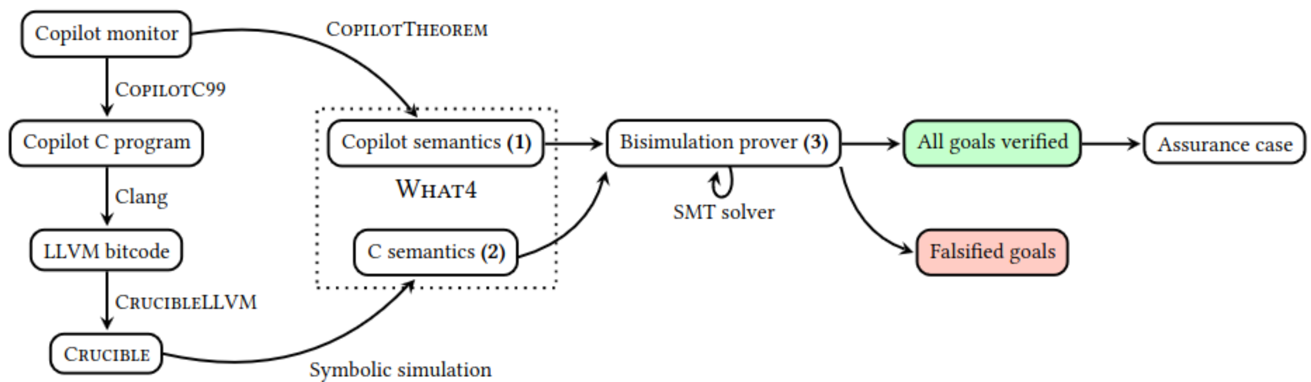


Fig. 15: Copilot Verifier architecture. *Note.* From ”Trustworthy Runtime Verification via Bisimulation (Experience Report)”, by Ryan G Scott, Mike Dodds, Ivan Perez, Alwyn E Goodloe, and Robert Dockins, 2023, *Proceedings of the ACM on Programming Languages*, vol. 7, no. ICFP, p. 6 [26].

Copilot supports temporal logic (LTL, PTLTL, and MTL) and provides a library for higher-level constructs for ”defining clocks, a Boyer-Moore majority voting implementation and various temporal logics” [24].

2.2.4 Flask

To create a web interface that continuously operates on a server, Flask can be implemented. Flask is a lightweight Web Server Gateway Interface (WSGI) microframework that allows any computer on the network to access the web server given the IP address of the computer running the application. Originating from the Python community, Flask was developed by Armin Ronacher as a part of the Pocoo project. Its simplicity and extensibility have made it a popular choice. A significant reason for its popularity is the way it was designed, focusing on versatility and modularity in its usage. Based on GitHub ratings among Python web development frameworks, Flask ranks second, only slightly behind Django, another web development tool that includes many built-in cybersecurity features. Flask is also used by major franchises, having both Netflix and Uber as users of the microframework. Flask provides the essentials to build a web application, offering developers more control and flexibility.

Flask is activated by invoking the app.run() method, which runs the app on a local server that can be accessed by any computer within the same network [28]. While running, you can access the HyperText

Markup Language (HTML) page through a Uniform Resource Locator (URL) link based on the server's IP address. From this address, multiple pages can be created with distinct functionalities and designs.

The aesthetic and design aspects of the web pages within the Flask application can be shaped using JavaScript and CSS. With the addition of D3, advanced interactive elements can be created that behave in relation to each other. These elements enhance user experience and the clarity of the presented information. One such use case is the automatic generation of state machine charts based on gravity between nodes of operation. Together, these technologies can create a full-fledged web application that is both functional and visually appealing.

Rosbridge, SocketIO & rospy for ROS/Flask Integration

To enable Flask to communicate efficiently with ROS for real-time applications, some middleware is needed to bridge the communication gap. Flask operates as a traditional HTTP request-response model, which closes the connection after each request is fulfilled. This complicates the process of receiving live updates from a ROS service since a web page update is needed for each time step. A WebSocket, on the other hand, allows the server and the client to send messages back and forth at any time without the need for a request to be opened and closed. Once a connection is established, the information streams both ways uninterrupted. This makes WebSockets particularly well-suited for real-time applications such as RV of ROS systems, where low latency and continuous data exchange are required to maintain safety.

WebSocket communication over a single long-lived connection allows for two-way communication between the client and server. A WebSocket's efficient and flexible method for real-time web application enables the server to push updates to the client as soon as they happen. Rosbridge provides one such system: "a JSON API to ROS (Robot Operating System) functionality for non-ROS programs" [29]. Essentially, it is a WebSocket server that provides communication between a ROS system and non-ROS clients. The Rosbridge service enables clients to subscribe to ROS messages, publish ROS messages, and invoke ROS services. This makes it possible to interact with ROS over the web and integrate ROS functionalities into web applications such as Flask.

Another WebSocket tool that befits the needs is Flask-SocketIO, which is an extension that gives Flask WebSocket capabilities. With Flask-SocketIO, real-time bidirectional communication is made possible between the Flask web client and the server. This gives Flask web applications the ability to instantly exchange data or updates without needing to refresh the page [30].

Since Flask is built on a Python framework, you can use ROS Python libraries or rospy within the Flask application to publish, subscribe, or call ROS services directly. This simplifies the communication to and from the website but requires that your Flask application runs on a ROS-integrated system. This method is simpler and more direct than using Rosbridge but limits your Flask application to environments where ROS is installed. The development behind Rospy "favors implementation speed (i.e. developer time) over runtime performance" making it non-ideal for critical-path code [31].

Comparative Aspect: Flask-SocketIO is focused on web development, whereas Rosbridge and Rospy are tailored to robotics applications. Rosbridge provides an interface for non-ROS applications (including those built with Flask-SocketIO) to communicate with ROS systems, acting as a mediator. Rospy is for direct interaction with ROS, enabling the programming of ROS nodes. A possible architecture might involve a ROS-based robotic system where Rospy is used for creating nodes that handle robot control and sensor data processing. Rosbridge would then provide a WebSocket interface to this ROS system, and a Flask application augmented with Flask-SocketIO could offer a real-time web interface to interact with the robot. This setup leverages the strengths of each tool for a robotics application with a web interface.

When using multiple channels for information publishing and gathering, a problem called blocking arises, where systems like Rosbridge and Flask cannot run simultaneously when they operate on the same thread. To solve this issue, multi-threading can be applied.

D3

D3.js, also known as Data-Driven Documents, is an open-source Javascript library for manipulating the Document Object Model based on data. It is designed for data visualization in web browsers with integration into web standards such as CSS, SVG, HTML, and JavaScript. This enhances the integration with web technologies, and its flexibility allows for a broad range of applications in data visualization. [32]

Part of the work done prior to this thesis was during a summer project in 2023 financed by the research project "Robofarmer", which is run by SINTEF. The work was executed by a team of three students, two of whom are the authors of this paper. The focus was on utilizing Human Interface Device (HID) and Robot state machines, building upon the Yolo machine learning package for human detection, and a Flask backend web framework for handling logic, data processing from a RealSense 3D camera, and interactions between frontend user interfaces and the robotic system model. The frontend consisted of a HTML site for RV. With factors such as flexibility and integration with web technologies, D3.js was considered the best approach for visualizing the state machines.

2.3 Related RV Tools

Runtime verification (RV) is essential for ensuring the reliability and safety of robotic systems, particularly those utilizing the Robot Operating System (ROS). Various tools have been developed to facilitate this verification, each offering unique features and methodologies. This section examines prominent RV tools, comparing their capabilities and limitations to provide a comprehensive overview of the current landscape. The tools discussed in this text are ROSMonitoring [5], TeSSLa-ROS-Bridge [33], and ROSRV [34].

Each of these tools makes a unique contribution to the field of runtime verification for ROS, offering different levels of flexibility and real-time monitoring capabilities. By examining these tools, valuable insights can be gained into the strengths and limitations of current RV methodologies, which can guide future developments and improvements in this critical area. The following sections will explore the specifics of each tool, highlighting their distinctive features and evaluating their performance in certain contexts of particular, real-world importance. As a novel contribution, a comparative table is presented of these tools alongside the OGMA-compiled Copilot monitor system in Sec. 5.4.

2.3.1 ROSMonitoring Tool

For ROS implementations, ROSMonitoring [5] is a powerful tool that facilitates the creation of runtime verification monitors for both offline and online use. Currently, ROSMonitoring has not been developed with ROS 2 in mind, but its features and methods remain relevant when compared to those lacking in the ROS 2 monitor generation through OGMA. In this section, some of the features of ROSMonitoring are discussed and evaluated.

Similar to OGMA and FRET, ROSMonitoring checks requirements against formally verified logic specifications, often using temporal logic sequences like LTL. Through the use of formal verification, the system can be checked both at an offline stage and at runtime. The advantage of an offline check is that, during the design phase of a ROS application, ROSMonitoring ”exhaustively checks the behavior of a system.” by performing a variant of a realizability check, as the one in FRET [5]. In ROSMonitoring, this check is conducted by creating a formal model of the system in question and then performing a ”state space search” to test the satisfaction of requirements in all configurations[5].

While ROSMonitoring is not unique as a runtime verification platform, it stands out when compared to existing software due to its integration as a ROS solution and its high portability across different ROS distributions. Although it currently does not support ROS 2 distributions, efforts are underway to port ROSMonitoring for use with the ROS 2 system. The structure of the formally verified requirements in ROSMonitoring is adaptable to various use cases, as it is ”formalism agnostic.” Formalism agnostic means that the software does not depend on a specific verification formalism in order to understand and check requirement properties. By leaving the configuration up to the user's choice of formalism and ROS distribution, ROSMonitoring offers versatility in its applications. As stated in their documentation, ”ROSMonitoring can be applied to any kind of ROS-based robotic application, with no limitation on how each communication endpoint is implemented” [5].

A common approach to implementing runtime verification is to place the monitor outside of the ROS node operation area, where it can serve based on incoming topics and messages. This is how OGMA sets up the Copilot node, however, in ROSMonitoring the monitors are ”placed between ROS nodes to intercept messages on relevant topics and check the events generated by these messages against formally specified properties.” [5]. The advantage of intercepting messages before they have arrived is that the monitor can act as middleware. Acting upon interception with analysis and possibly modification of the outgoing message. This enhances system responsiveness and allows the monitor to prevent violation states as they occur. For applications that do not require the invasive feature of message interception, ROSMonitoring can be configured to function as a standard runtime verification application, turning off the feature.

Similar to OGMA, all ROS monitors in ROSMonitoring are automatically generated into ROS nodes that listen to and log incoming messages. The monitors are generated from a configuration YAML file, which is interpreted by the instrumentation software. The resulting nodes and monitor communicate with the oracle either through an offline or online approach. The oracle is an external component responsible for determining the correctness of incoming states. The oracle can adapt to different formal verification methods without needing to alter its structure, which contributes to the formalism-agnostic nature of ROS-Monitoring. In Fig. 16 the general architecture of the ROSMonitoring tool can be seen. Fig. 17 shows the performance decrease of the monitors at varying publishing speeds, which will be referenced in the discussion when comparing the RV tools.

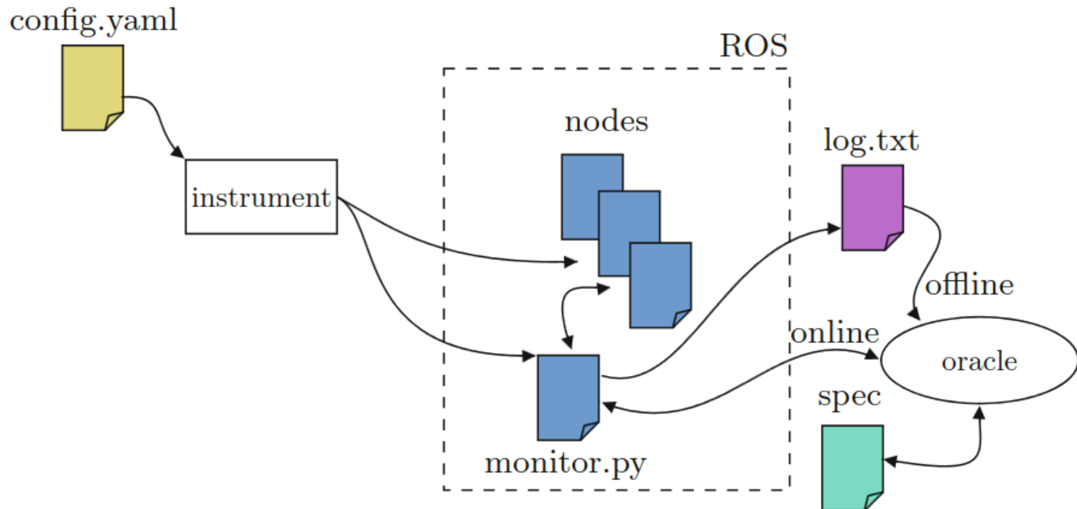


Fig. 16: Structure of the ROSMonitoring software. *Note.* From ”ROSMonitoring: A Runtime Verification Framework for ROS”, by Angelo Ferrando, Rafael C Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascard, 2019, *Towards Autonomous Robotic Systems: 21st Annual Conference*, p. 389 [5].

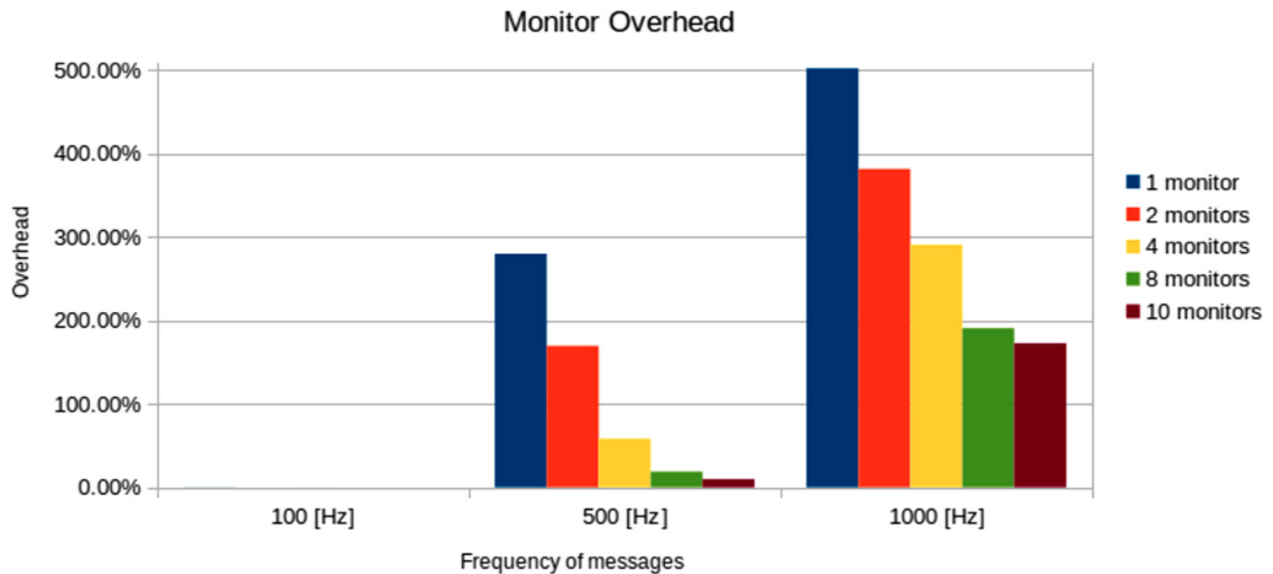


Fig. 17: ROSMonitoring overhead percentages based upon the number of messages being posted per second on ten different topics. Becoming 1000, 5000, and 10000 messages being publishes per second. Since only the presence of the monitors was being checked they kept the property to be verified as fixed. *Note.* From ”ROSMonitoring: A Runtime Verification Framework for ROS”, by Angelo Ferrando, Rafael C Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascard, 2019, *Towards Autonomous Robotic Systems: 21st Annual Conference*, p. 396 [5]

2.3.2 TeSSLa-ROS-Bridge

TeSSLa-ROS-Bridge, created by Marian Johannes Begemann and her team at the University of Lübeck, Germany, is a runtime verification tool that integrates ROS with the Temporal Stream-based Specification Language (TeSSLa) [35, 33]. This integration allows a TeSSLa monitor to operate as part of the ROS-based system, facilitating formally verified safety requirements that monitor the robot during runtime [33]. TeSSLa is particularly effective for ROS because it employs an asynchronous stream-based runtime verification (SRV) language, allowing input and output events to be processed independently without a fixed time grid, relying instead on a timestamp system. This makes TeSSLa suitable for Cyber-Physical Systems (CPS), which are physical systems controlled by software. The challenge with combining CPS with synchronous SRV is that data refreshment is not instantaneous, and small delays between updates can cause violations. TeSSLa removes this problem with the use of timestamps as it has been ”tailored for SRV of cyber-physical systems” [13].

TeSSLa-ROS-Bridge communicates with the ROS platform by using the publish-subscribe system native to the Robot Operating System. This is facilitated by the specification language TeSSLa. The concept behind TeSSLa is that a requirement or specification describes a transformation in the form of a stream input/output. From the requirement, a TeSSLa boolean stream can be defined, reporting whether the property is upheld for the current configuration. Advanced output streams are also possible with TeSSLa, allowing for the checking of value bounds or statistical properties. Additionally, all requirements defined in TeSSLa have a timestamp attached, enabling timing-based specifications. When an event is processed by the TeSSLa monitor, the output is logged, notifying the user of the result. If desired, the TeSSLa monitor can also feed the stream back into itself, allowing for interception strategies or mitigation actions for perceived violations. Fig. 18 generalizes the TeSSLa-ROS-Bridge architecture [33], demonstrating how ROS nodes are monitored and interacted with. The setup for the TeSSLa-ROS-Bridge is similar to the system structure of the OGMA-generated Copilot monitors, as shown in Fig. 25.

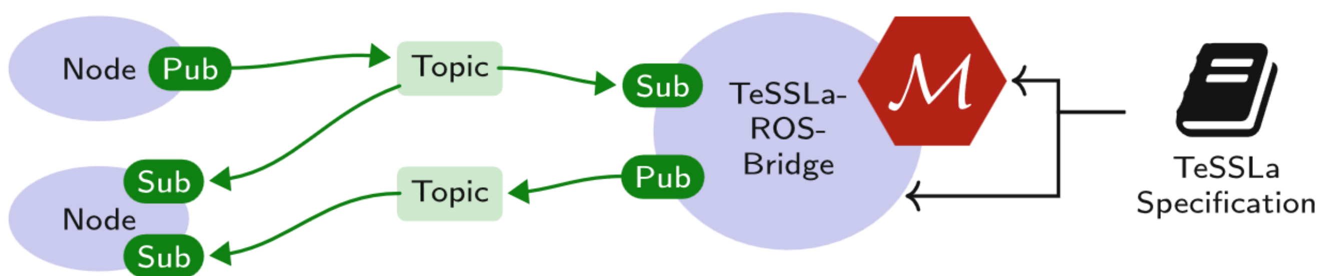


Fig. 18: Generalized diagram of how the TeSSLa-ROS-Bridge connects with ROS nodes. The topic information received by the monitor is treated as events on the stream, and output streams are made accessible to other nodes. *Note.* From ”TeSSLa-ROS-Bridge–Runtime Verification of Robotic Systems”, by Marian Johannes Begemann, Hannes Kallwies, Martin Leucker, and Malte Schmitz, 2023, p. 392 [33]. Springer Nature Switzerland AG.

TeSSLa was originally not designed to support ROS, and a bridge was necessary to facilitate communication between the two. To achieve this, the TeSSLa monitor was encapsulated within a ROS node, subscribing to existing topics and publishing outputs. The ROS node used for the bridge application runs ROS 2 and functions like any other ROS 2 node. This simplifies the process of integrating the monitor into existing ROS systems, requiring no alterations to the ROS structure in order to utilize the TeSSLa monitor. The bridge also ensures that the monitor remains separate from the system software, providing a non intrusive monitoring experience. If desired one could implement mitigation actions or intercept commands based off the monitor outputs. Natively the TeSSLa monitor does not manipulate any ROS messages. The bridge is programmed in Python using the rclpy library, a ROS client tool for Python. The TeSSLa monitor within the ROS node is compiled in Rust, with PyO3 linking it to the python bridge.

The TeSSLa-ROS-Bridge was tested in both a practical example, bridging a ROS robot with ROS 2, and in a simulated environment. Using a ROS node that generated "dummy events" on an "Intel Core i7 CPU and 8GB RAM" virtual machine, an average latency of 15 milliseconds from publishing to response per event was recorded [33]. The study noted that latency was not "noticeably affected by the size of the specification," suggesting that most of the latency is attributed to the bridge implementation. The paper also mentions that a high frequency of event changes can present challenges to the system, though no performance results are provided for this scenario.

2.3.3 ROSRV

ROSRV is "a runtime verification framework for robotic applications on top of the Robot Operating System (ROS)", engineered with a focus on enhancing the safety and security of robots within the ROS ecosystem. ROSRV shares the invasive approach of the ROSMonitoring tool by inserting monitoring nodes on topic paths, allowing it to intercept and optionally manipulate point-to-point communication between nodes. Consequentially, the monitor nodes of ROSRV can act as "men-in-the-middle" [34]. However, ROSRV lacks development for ROS 2.

While both ROSRV and ROSMonitoring utilize intercepting monitor nodes to ensure application safety, ROSRV introduces an additional, centralized, node called the RVMaster, designed to address the security issues in ROS-based robotics. This new master node acts as a gateway with stricter management than the standard ROSMaster, enforcing security policies to make the system safer overall. This means that all monitoring nodes are managed within a single multi-threaded process, which could pose problems with a large number of nodes [34]; serving no purpose with the intention of runtime verification, instead possibly acting as a bottleneck in the system. There are still limitations regarding security in the current implementation due to its "reliance on IP addresses in particular and network routing, in general, to guarantee security" [34]. Despite these limitations, ROSRV offers significantly better security measures compared to similar monitoring tools.

Mirroring the testing procedures of ROSMonitoring, ROSRV's performance was assessed by the overall overhead introduced into the system with active monitors. The key difference lies in ROSRV opting to compare the rate of message delivery from a publisher to a subscriber, while ROSMonitoring compared the message output of the system to several topics. The experiments involved implementing two nodes; a publisher and a subscriber, with message transmission and reception measured under three conditions: 1) using ROSMaster, 2) using ROSRV without a monitor, and 3) using ROSRV with a monitor. The first experiment ran for 10 seconds, while the second experiment extended the run time to 10 minutes.

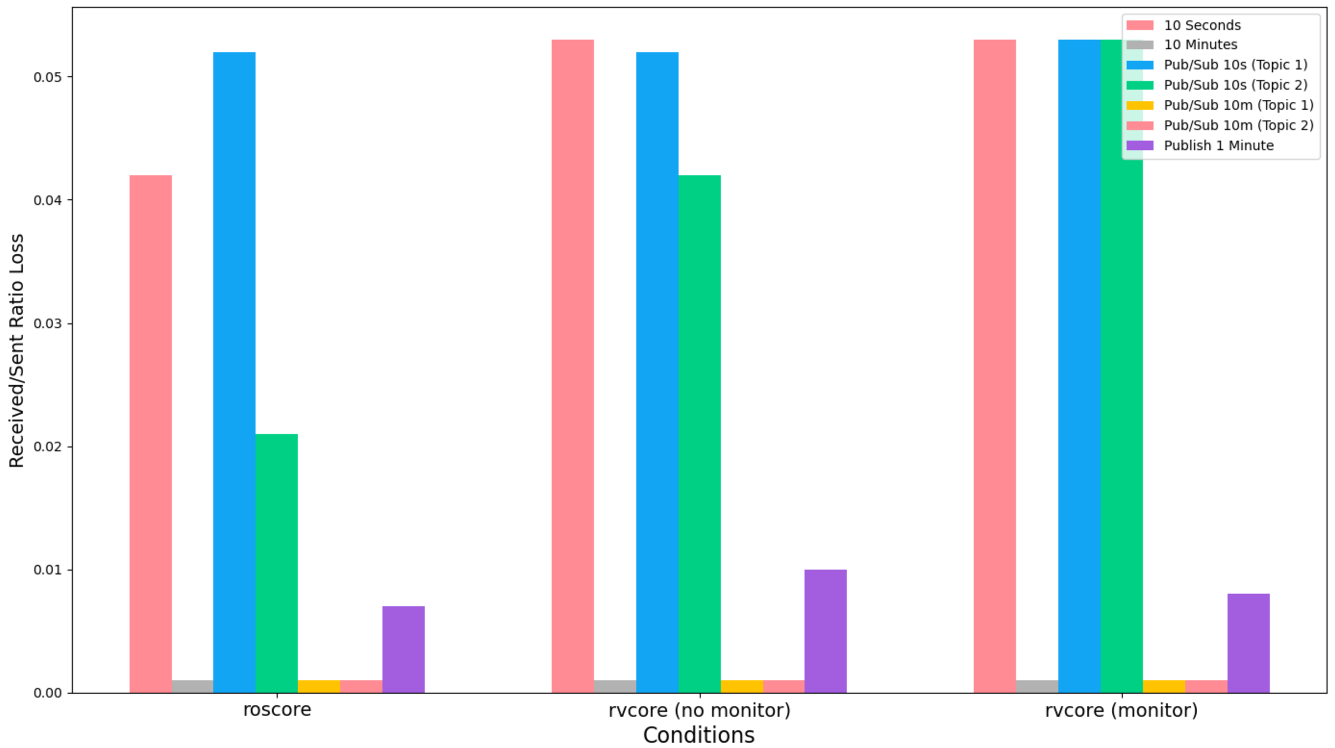


Fig. 19: Performance barplots on the ROSRV framework, where overhead metrics are given in decimal. *Note.* Adapted from "ROSRV: runtime verification for the Robot Operating System", by Cansu Erdogan, 2015, pp.32-35 [36].
*roscore = ROSMaster, rvcore = ROSRV

The performance metrics in Fig. 19 indicate the minimal impact of the overhead introduced by ROSRV on the system. Although the 10-second experiments show a marginal decrease in performance, they are outperformed by the 10-minute experiments demonstrating a negligible overhead over longer periods of time. The 1-minute experiment indicates minimal differences between conditions, suggesting a rapid improvement in performance over runtime, further supporting the findings. Despite these positive results, further testing is required to quantify the scalability of the framework as ROSRV employs a centralized architecture [34].

It should be noted that due to the advancements in security mechanisms in ROS 2, as mentioned in Sec. 2.4, the RVMaster designed to enforce security policies is an outdated approach and may serve only as a potential bottleneck.

2.4 ROS & ROS 2

ROS (Robot Operating System) is an open-source framework designed by robotics research lab and technology incubator Willow Garage and is, in many ways, the foundation of this paper. ROS was designed to help researchers and developers create complex and robust robotic applications. With development from Open-Source Robotics Foundation (OSRF), ROS has become the de facto standard in the field of robotics research and academia for over 15 years due to its strong, yet simple, core concepts. This has led to ROS having an extensive set of tools and libraries, as well as an active community and ecosystem.

At a high-level, ROS is a peer-to-peer network of ROS processes [37]. Central to this network is the ROS Master, which acts as a name service and lookup for the system and is essential for establishing communication within the network. The network architecture consists of nodes that perform various computational processes. The nodes communicate via messages through channels called topics. Each node in a system has the capability to publish and subscribe to any topic they want, thus becoming modular and reusable. The primary mechanism of ROS applications is callbacks. Whenever a node gets a message through a topic, a callback assigned to the node's subscription is activated, much like a function in general-purpose programming languages. Fig. 25 showcases a generalized high-level view of a ROS system by Clearpath Robotics [38].

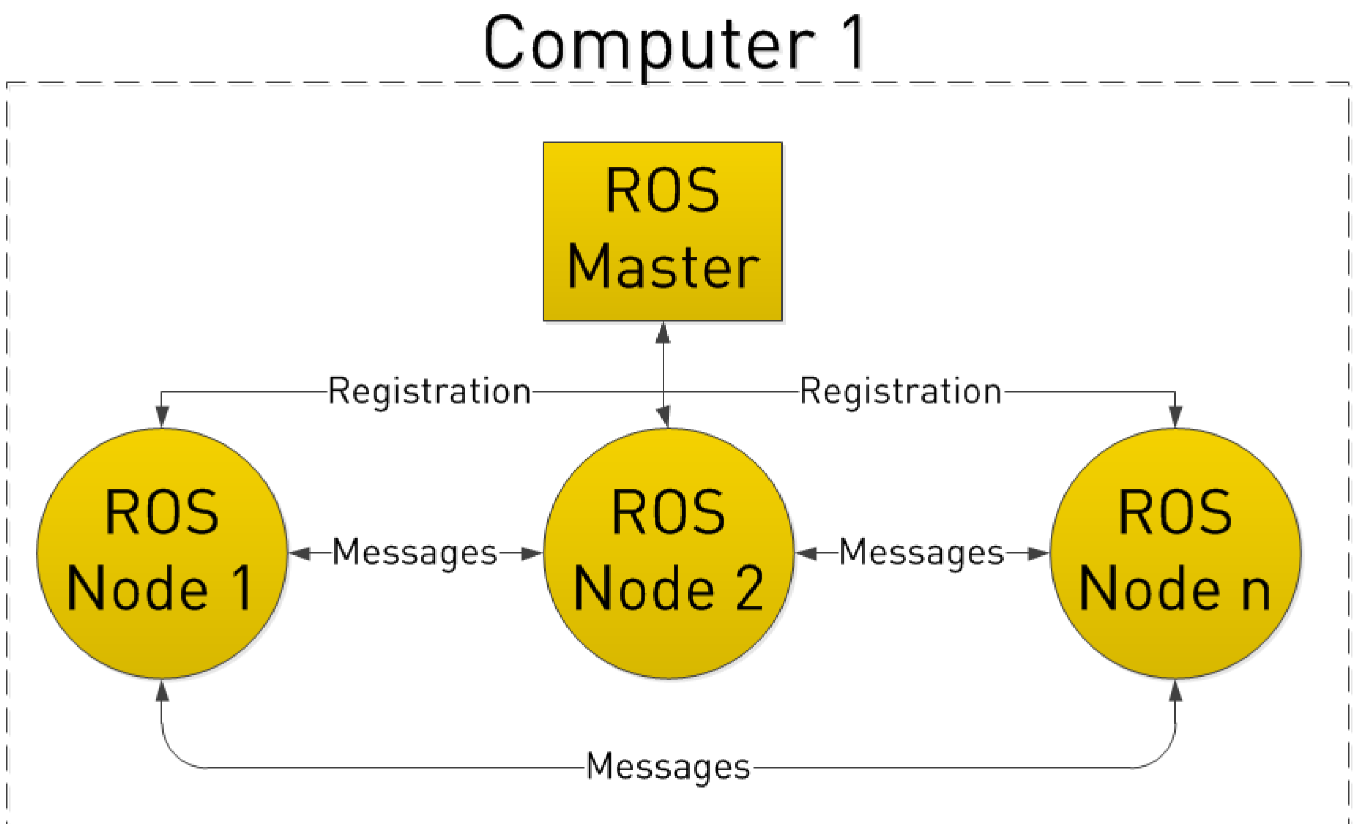


Fig. 20: High-level structure of a ROS system. *Note.* From Clearpath Robotics [38].

However, as with most initial software iterations, ROS has some significant problems. First, ROS does not provide real-time capabilities, making it unsuitable for deployment in critical operations where precise timing is essential. Another limiting factor is minimal network security. ROS relies on IP-based access control policies to manage security and depends heavily on the security of the network it is implemented on or on external network-monitoring tools such as ROS-FM [39]. With an expanding library of ROS platforms in various mission-critical scenarios, these limitations have driven researchers to develop ROS 2 to address this, as well as several other issues.

ROS 2 is built on top of Real Time Publish Subscribe (RTPS) as its middleware. RTPS is a Data Distribution Service (DDS) which provides discovery, serialization and transportation [40]. This offers authentication, encryption, access control, and audit logging as part of the middleware. Especially relevant for runtime verification is the feature of distributed discovery. In distributed discovery, there is no centralized service to establish communication among individual members of the system, in this case, the nodes. Instead, the members discover each other. This shift in infrastructure significantly improves scalability by reducing the risk of bottlenecks by eliminating the centralized service and distributing the load among the nodes. In centralized discovery, a system-wide impact can be felt if there is a failure, potentially causing complete shutdown of operation. In contrast, with distributed discovery, the other nodes can continue to operate after one node fails, enhancing resilience. This introduces runtime verification as a pivotal role, not just in detecting what, when, and where something went wrong, but also in taking mitigating actions based on the errors.

One notable feature of ROS and ROS 2 is their event-driven nature, which processes messages sequentially, even when multiple messages need to be handled quickly. This sequential processing ensures orderly communication but can introduce minimal delays when publishing multiple messages simultaneously. Although it is not possible to publish all messages at the exact same instant, the delay can be minimized to achieve an effect that is nearly simultaneous.

2.5 Robotool

RoboTool, developed by RoboStar at the Department of Computer Science, University of York, is mentioned in this paper [41]. The idea is that system models generated in RoboTool can be exported to FRET and automatically made into safety monitors. This is made possible since FRET's emphasis on temporal logic enables seamless integration with the offline verification tool, RoboTool. Since Robotool is not directly used in the paper not too much will be said about it, instead some general information regarding Robotool is presented.

RoboTool presents itself as a specialized robotics platform specifically designed to support development, analysis, and verification of robotic systems [42]. The platform is based on the RoboChart modeling framework and language, and provides a dedicated environment for the construction of detailed models; parsing, type checking, validation, graphical editing, and model generation. The aim is to simplify the process of modeling the complex robotic systems that are prevalent in RoboChart. For context, RoboChart is a Domain-Specific Language (DSL) based on the Unified Modeling Language that allows for modeling of robotic applications and supporting verification via model checking [43].

RoboTool facilitates the creation and editing of RoboChart diagrams and automates the generation of mathematical Communicating Sequential Processes (CSP) and PRISM models [44]. Additionally, it offers a simplified notation employing controlled English to define assertion-capturing properties of interest, supporting its formalization of properties.

2.6 Robot Vision

Robot vision is a crucial component in the field of robotics, enabling machines to interpret and understand their environment through visual data. This capability is essential for autonomous systems to navigate, identify objects, and interact effectively with their surroundings. The integration of advanced vision systems in robotics, such as depth cameras and sophisticated object detection algorithms, significantly enhances the operational capabilities and safety of these systems. Here the RealSense D435 depth camera and the You Only Look Once (YOLO) imaging tool are described. These tools are used in the thesis for data collection for usage with the safety monitor.

2.6.1 RealSense Depth Camera

In Cyber-Physical Systems (CPS), the software is combined with physical parts or sensors, and one such sensor used in this paper is the Intel RealSense D435 depth camera. The RealSense D435, first announced in 2018, is a stereo solution camera with high-quality depth capabilities. The D435 model has a wide field of view and an announced range of ten meters, making it a good pick for robotic applications.

In Fig. 21 a photo of the RealSense has been labeled into sections for understanding the features the camera has to offer. Labeled ”a” and ”c” are the right and left imager sensors, which when combined offer a stereo infrared (IR) image with depth information. The IR light the imager modules receive is produced by the IR projector, ”b”, in a structured pattern across the field of view. With infrared light, otherwise featureless objects like walls become easy to read for the depth module. Infrared light is invisible to the naked eye, but under low light conditions, it is possible to see the light pattern on a reflective surface, as shown in Fig 22. The last module ”d” is the RGB sensor. The RGB module records a high-resolution color image used in conjunction with the depth image in the camera output. The data from the RealSense camera can be used for object/distance measurement, 3D scene reconstruction, or as any of its parts being either: an IR depth sensor or RGB camera.

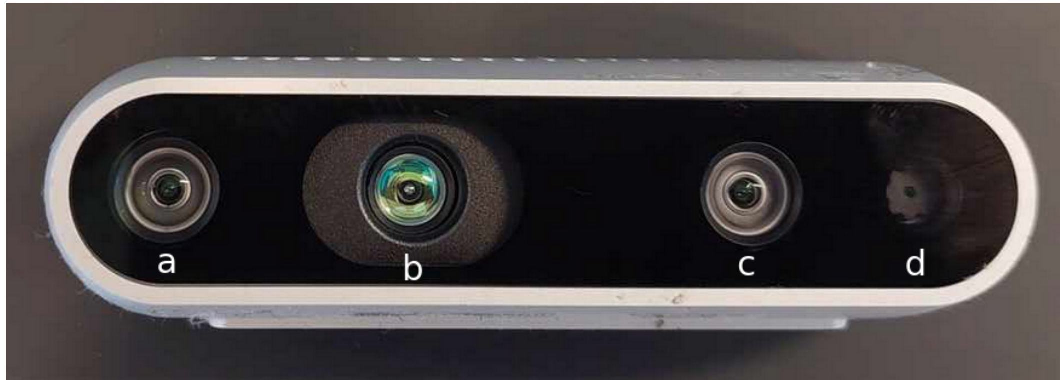


Fig. 21: Intel RealSense camera, model D435. The camera in the image has the following labels: a - right imager, b - infrared (IR) projector, c - left imager, and d - RGB module [45].

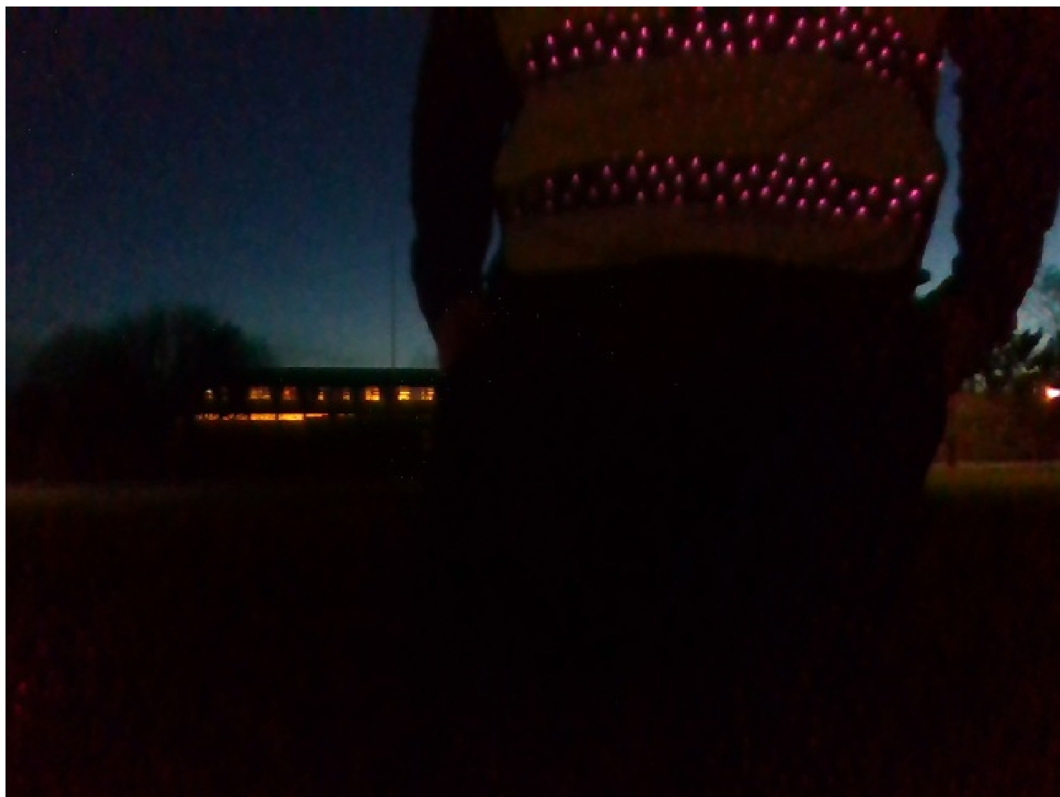


Fig. 22: Image from the RealSense in the dark, showing the infrared (IR) pattern on the reflective parts of a safety vest.

2.6.2 YOLO

While the novelty may lie in the implementation of the depth camera, YOLO is introduced as the object detection software that leverages this technology. To be precise, YOLO is a single-stage object detection framework and has been the de facto industry-level standard for many years [46]. ”Single-stage” defines the trait of being able to predict bounding boxes and class probabilities directly from a full image in a single evaluation.

High-level Abstraction of YOLO’s Workflow

To initialize the process of object detection, an input image is divided into a grid of size S^2. Each grid cell then produces two things. Firstly, a set of bounding boxes centering a point inside the cell with associated confidence scores for whether an object is contained in each bounding box. Secondly, a class probability map pertaining to the object class most likely to be linked with that cell, given an object presumably exists in the cell. YOLO then combines these elements to yield the output image with detected objects bounded.

Architecture

The architecture of YOLO follows that of a standard CNN (Convolutional Neural Network). Consisting of a stack of convolutional layers, the input can be distilled. This is followed by two fully connected layers where the abstract information is parsed and transformed into output vectors for each grid cell.

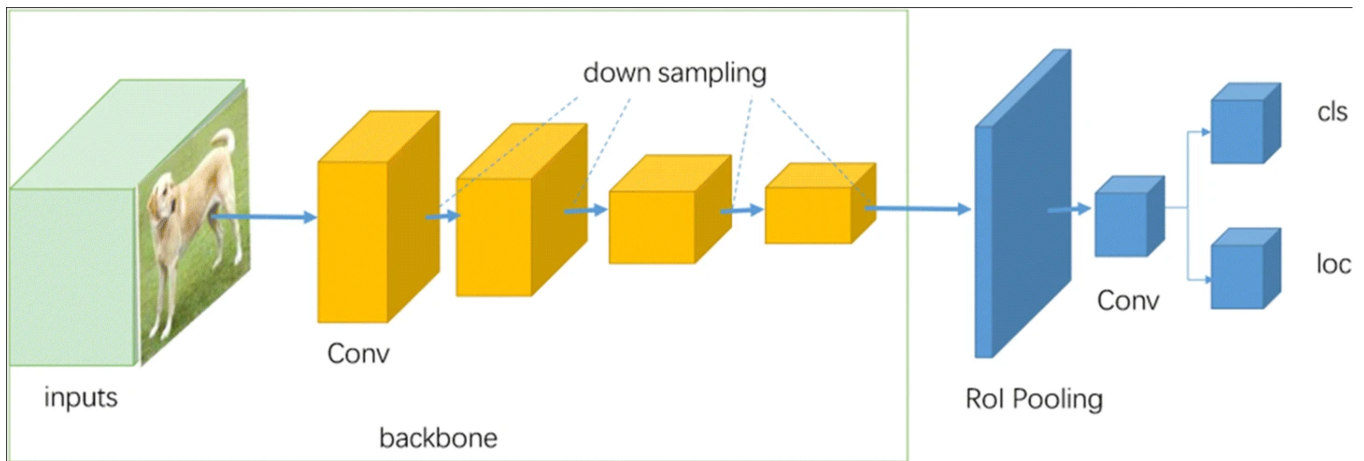


Fig. 23: YOLO architecture. The input image with three channels, each for the red, green, and blue values respectively, is distilled into an abstraction and transformed into output vectors used for predictions. *Note.* From ”A survey of deep learning-based object detection”, by Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng and Rong Qu, 2019, *IEEE access*, vol. 7, p. 2 [47]. CC BY 4.0.

YOLOv6

More specifically, version YOLOv6 was used for the research in this paper. This is due in part to the renovated design of the framework along with its impressive performance in real-time object detection.

”The renovated design of YOLOv6 consists of the following components, network design, label assignment, loss function, data augmentation, industry-handy improvements, and quantization and deployment”[46].

Table 6: Comparison of Detection Frameworks. *Note.* Adapted from ”A review: Comparison of performance metrics of pre-trained models for object detection using the TensorFlow framework, by SA Sanchez, HJ Romero, and AD Morales, 2020, *IOP Conference Series: Materials Science and Engineering*, p. 9 [48], and the YOLOv6 GitHub, by Meituan [49].

Detection Frameworks	Train	mAP	FPS
Fast R-CNN	PASCAL-VOC 2007+2012	70.0	0.5
Faster R-CNN VGG-16	PASCAL-VOC 2007+2012	73.2	7
Faster R-CNN ResNet	PASCAL-VOC 2007+2012	76.4	5
YOLO	PASCAL-VOC 2007+2012	63.4	45
SSD300	PASCAL-VOC 2007+2012	74.3	46
SSD500	PASCAL-VOC 2007+2012	76.8	19
YOLOv6-N	COCO val2017	37.5	779
YOLOv6-S	COCO val2017	45.0	339
YOLOv6-M	COCO val2017	50.0	175
YOLOv6-L	COCO val2017	52.8	98

Tab. 6 contains data collected from the paper ”A review: Comparison of performance metrics of pre-trained models for object detection using the TensorFlow framework”[48], as well as the YOLOv6 repository. All code for YOLOv6 is made publicly available by the researchers at Meituan Vision AI Department on GitHub alongside comprehensive documentation and resources:
`https://github.com/meituan/YOLOv6`

Tab. 6 portrays benchmarks of several frameworks with a common goal of object detection. The data was obtained from testing various frameworks on the PASCAL-VOC 2007 and COCO val2017 datasets. Datasets for object detection normally consist of two elements; images and annotations. Images come with a range of object categories, meaning different types of objects to be classified, such as ”dog”, ”plane” and ”bicycle”. Attached to these images are annotations. Annotations include bounding boxes which are the frames surrounding the objects, and object labels. mAP stands for mean average precision and is the main metric for evaluation. It is given at varying IoUs (Intersection over Union), which is the overlap of the ground truth bounding box and the predicted bounding box. The frameworks tested on the PASCAL dataset evaluate the mAP at a 0.5 IoU threshold. The YOLOv6 frameworks evaluates mAP across a 0.5:0.95 threshold range with an increment of 0.05.

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

By averaging the precision and recall across all classes, the metric conveys how well a given framework performs with different types of objects.

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

Here precision can be seen as the relation between the correctly detected objects and the total number of objects detected in an image. Recall can be seen as the relation between identified objects and the total number of objects in an image.

The PASCAL-VOC 2007 dataset is a standard dataset consisting of 9963 images with annotations and is related to the PASCAL Visual Object Detection challenge. These images come with less complexity and occlusion, giving clearer pictures with more defined classes. The COCO val2017 dataset, on the other hand, includes an extensive 118,000 images with annotations with much higher complexity due to a greater variety of object classes, distances, and occlusion.

From the frameworks tested on the PASCAL dataset, YOLO and SSD300 performed especially well with 63.4 mAP at 45 frames per second (FPS) and 74.3 at 46 FPS respectively. The YOLOv6 frameworks tested on the COCO dataset boasted a range of 37.5-52.8 mAP at higher rates of 98 to a staggering 779 FPS. This is especially impressive considering the heightened complexity of the COCO dataset.

Confusion matrices are effective for evaluating classification models, from which accuracy, precision, recall, specificity, F1 score, fall-out, and miss rate can be calculated. A positive value (P) is the worker class and a negative value (N) is an adult. Fig. 24 is used to define the equations used for the YOLOv6 evaluation.

	P (worker)	N (adult)
P prediction	TP	FP
N prediction	FN	TN

Fig. 24: Use the following matrix for the abbreviations used in the formulas.

① **Accuracy**

Accuracy measures the overall correctness of the model and is calculated as the ratio of correct predictions (both positive and negative) to the total number of cases examined.

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

② **Precision (Positive Predictive Value)**

Precision is the ratio of correct positive predictions to the total predicted positives. It is a measure of the quality of the positive class predictions. In this case the worker class.

$$\text{Precision} = \frac{TP}{TP + FP}$$

③ **Recall (Sensitivity or True Positive Rate)**

Recall is the ratio of correct positive predictions to the actual positives. This metric tells us how well the model can identify positive results.

$$\text{Recall} = \frac{TP}{TP + FN}$$

④ **Specificity (True Negative Rate)**

Specificity is the ratio of correct negative predictions to the actual negatives. It shows how well the model can identify negative results. In this case the adult class.

$$\text{Specificity} = \frac{TN}{TN + FP}$$

⑤ **F1 Score**

The F1 Score is the harmonic mean of precision and recall, providing a balance between them. It is particularly useful when the class distribution is uneven.

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

3 Requirements

In this section, the requirements of the paper are outlined, providing information regarding the research objectives, the specific criteria for designing our desired pipeline for RV, and what safety requirements we are adhering to when executing simulations and field tests.

3.1 Goals and Objectives

The primary goal of this project is to automatically develop a robust RV platform for ROS 2 applications from Natural Language (NL) requirements, equipped with a user interface. This platform aims to facilitate user interaction with the ROS system through simple commands and will be applicable to both CPS and simulation environments. The monitor should demonstrate reliable, high-performance results during execution. Additionally, the RV monitors should be capable of being automatically generated from near-natural language requirements, which can be converted into formally verified temporal logic for precise verification.

3.2 System Requirements

The system is designed to meet the following requirements, in no particularly order:

1. **ROS 2 Compatibility:**
   The system must be fully compatible with the Robot Operating System 2 (ROS 2), supporting seamless integration and communication with ROS Master nodes. This can be achieved with middleware.

2. **Low Latency and High Performance:**
   The system must exhibit low latency and maintain high performance, even under high load conditions, to ensure timely and efficient operation.

3. **Bidirectional Communication:**
   Support for bidirectional communication is required to enable effective data exchange between nodes and the front-end. Also benefiting the implementation of front-end user control.

4. **Linux Compatibility:**
   The system must be compatible with Linux operating systems, ensuring broad usability and integration with various Linux-based environments.

5. **Hard Real-Time Capabilities:**
   The system must support hard real-time operations, ensuring predictable memory usage and execution time. The system must not use dynamic memory allocation (malloc), for loops, or recursion to maintain constant memory and time bounds. This is in order to maintain predictable monitor execution through deterministic behavior and a simplified process.

6. **Formal Verification:**
   The implementation must be formally verified or able to perform verification of requirements, to guarantee correctness and reliability, particularly in safety-critical applications.

7. **Violation Detection and Reporting:**
   The system must include mechanisms to detect and warn the user of any violations in real-time, providing alerts through the front-end interface.

3.3 Safety Requirements

Safety requirements are founded on risk assessments and mitigation planning. The risk assessments for this thesis were made by researchers at the Lincoln Centre for Autonomous Systems, University of Lincoln, UK, and presented in an article about probability of human injury during UV-C treatment of crops by robots [50]. Mitigation plans were further developed and presented during the 2023 IEEE 19th International Conference on Automation Science and Engineering (CASE) [44]. They have since then been updated with the input of a Saga Robotics safety engineer and are listed below.

Table 7: Mitigation plan based on the analysis. *Note.* From "Probabilistic modelling and safety assurance of an agriculture robot providing light-treatment", by Mustafa Adam, Kangfeng Ye, David A. Anisi, Ana Cavalcanti, Jim Woodcock, and Robert Morris, 2023, *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, pp. 1-7 [44].

ID	**Human detected**	**Zone**	**Mitigation Actions** (μ_{act})
R_1	Trained	Green	Activate sound
R_2	Untrained	Green	Activate sound & slow down
R_3	Trained	Yellow	Activate sound & slow down
R_4	Untrained	Yellow	Turn off UVC & Stop robot
R_5	Trained/Untrained	Red	Turn off UVC & Stop robot

Table 8: Safety requirements, corresponding mitigation plan for the presence of trained/untrained person. Human is used in the case where both classes apply.

ID	Possible situation	Mitigation plan
R0	Robot is performing a transition between rows and no human is detected	Nothing
R1	Robot is performing a transition between rows and trained worker detected in the green zone (more than 7m distance)	Activate sound
R2	Robot is performing a transition between rows and untrained human detected in the green zone (more than 7m distance)	Activate sound & slow down
R3	Robot is performing a transition between rows and trained worker detected in the yellow zone (between 3-7m)	Activate sound & slow down
R4	Robot is performing a transition between rows and untrained human detected in the yellow zone (between 3-7m)	Turn off UVC & Stop robot
R5	Robot is performing a transition between rows and trained worker detected in the red zone (less than 3m)	Turn off UVC & Stop robot

4 Methodology

The methodology chapter outlines the systematic approach and processes undertaken to achieve the objectives of this research. It provides a detailed account of all tools, frameworks, and interesting techniques used in the development and validation of the RV pipeline for ROS 2. The goal of this chapter is to give the reader an understanding of the research process of this paper and to serve as a resource in aiding future research on ROS based RV monitoring.

The methodology chapter is divided into several sections, each detailing critical aspects of the research process. The chapter begins with an introduction to the system architecture, including its development and implementation (Sec. 4.1), and examines the installation process and integration of FRET and OGMA into the workflow. Subsequently, the methodologies behind the system's testing in both simulated and real-world environments is explained (Sec. 4.2). Lastly, there are two shorter sections addressing the inclusion of a violation injection device for testing purposes (Sec. 4.3) and the assessment of ethical considerations upheld during data collection (Sec. 4.4).

The inspiration for advancing RV in ROS 2 applications came from the paper "Automated Translation of Natural Language Requirements to Runtime Monitors" by Dr. Ivan Perez and his team at NASA. The paper outlines the steps for creating C monitors for ROS 2 from FRET input in the following seven steps [21]:

1. Fret automatically translates requirements into pure Past-time Metric Linear Temporal Logic (ptLTL) formulas.
2. Information about the variables referenced in the requirements must be provided by the user.
3. The formulas and provided variable data are then combined to generate the Component Specification.
4. Based on this specification, OGMA creates a complete Copilot monitor specification.
5. Copilot then generates the C Monitor.
6. This monitor, along with other C code, is given to a C compiler.
7. The final object code is generated.

4.1 Implementation of system architecture

This section details the transformation of a theoretical design into a functional system, outlining the process and tools used for setting up the configuration. The development and deployment were carried out on Ubuntu 22.04.4 (Jammy Jellyfish) using both ROS 2 iron and humble distributions.

Fig. 25 provides a generalized diagram of the monitor architecture used in this project. This architecture integrates NASA's FRET and OGMA tools to facilitate the automated translation of natural language requirements into executable runtime monitors, which run in parallel with the ROS 2 nodes. Refer to Fig. 25 when following the methodology, for a better understanding of how the components are connected.

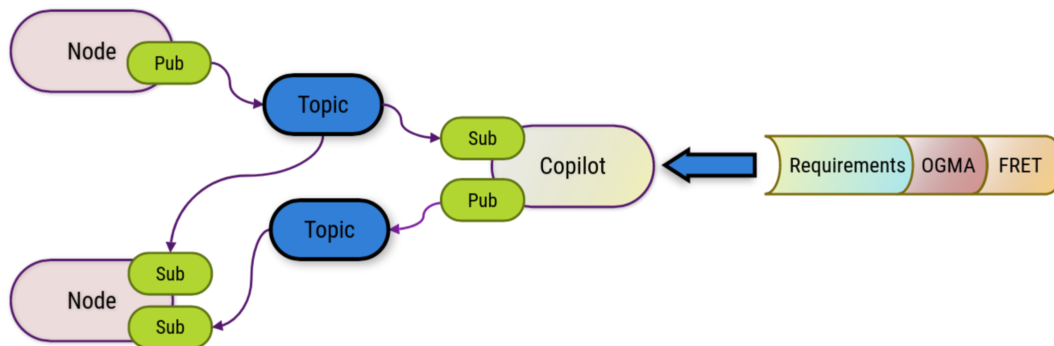


Fig. 25: Generalized diagram of monitor architecture.

4.1.1 Installation of software

The successful implementation of the system architecture relies heavily on the correct installation and configuration of various software tools. Here is a detailed guide on the steps and considerations involved in setting up the software environment, focusing on the essential tools required for the project. All installation sequences are displayed as bash scripts, with each line representing the next installation step.

FRET

NASA's Formal Requirements Elicitation Tool (FRET) is used for specifying the robot platform's requirements. To get started with FRET, refer to the tool's GitHub page containing installation instructions: `https://github.com/NASA-SW-VnV/fret`. Here is a detailed guide on how the tool was set up for this paper.

The FRET software depends on several other tools:

- **NodeJS** `https://nodejs.org/en/download/`
- **NuSMV**: `https://nusmv.fbk.eu/`
- **JKind** `https://github.com/andreaskatis/jkind-1/releases/tag/v2.2`
- **Kind 2** `https://github.com/kind2-mc/kind2/blob/develop/README.rst`
- **Z3** `https://github.com/Z3Prover/z3/releases`

NuSMV, JKind, Kind 2, and Z3 are optional dependencies that provide additional functionality to aspects of the FRET portal. **The specifics versions used in this paper are:**

```
NodeJS v16.20.2 with npm 8.19.4
NuSMV 2.6.0, JKind 4.5.1-202311081135 with smtinterpol and z3
kind2 v2.1.1-23-gd217e3c
Z3 version 4.8.10 - 64 bit.
```

For first-time installation and running of FRET, a streamlined process is provided in the bash-script of Lst. 1:

Listing 1: FRET install script.

```bash
#!/bin/bash
git clone https://github.com/NASA-SW-VnV/fret.git
cd fret-electron
npm run fret-install
npm start
```

OGMA

The OGMA tool by NASA is used for generating the ROS monitoring application. To get started with OGMA, refer to the software's GitHub page:
`https://github.com/nasa/OGMA`

The installation of OGMA involves installing GHC and Cabal. This can be done simply through command-line code given on GitHub. It is important to note the version when working with Cabal. "OGMA has been tested with GHC versions up to 9.2 and cabal-install versions up to 3.6" [21]. However, the recommended versions are GHC 8.6 and Cabal 2.4 and 3.2. For this paper, GHC 8.6.5 and Cabal 2.4.1.0 were used, giving the desired result when compiling the monitor. These versions can be installed through the GHCup tool, which manages different versions of the Glasgow Haskell Compiler and Cabal. The streamlined process is provided in the bash script of Lst. 2.

Listing 2: GHC and Cabal install script.

```bash
#!/bin/bash
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
ghcup install ghc 8.6.5
ghcup install cabal 2.4.1.0
ghcup set ghc 8.6.5
ghcup set cabal 2.4.1.0
```

Use `ghcup list` to check the installations. The installed and set versions should be marked with green double check marks. After installing the correct versions of GHC and Cabal, follow the OGMA GitHub guide for installing the OGMA software. The installation script for OGMA is shown in the bash script, Lst. 3.

Listing 3: OGMA install script.

```
1 #!/bin/bash
2 git clone https://github.com/nasa/OGMA.git
3 cd OGMA
4 export PATH="$HOME/.cabal/bin/:$PATH"
5 cabal v1-update
6 cabal v1-install alex happy
7 cabal v1-install BNFC copilot
8 cabal v1-install OGMA-*/
```

Note that appending the `--force-reinstall` argument might be necessary to install the 'OGMA' executable. Another option for installation would be the Dockerfile mentioned under the OGMA Github.

4.1.2 Requirement Logic and Safety Engineering

Transitioning from installation, we now examine how safety requirements can be defined for a specific implementation. The logic behind the the monitor's safety requirements involves risk assessments and mitigation planning based on a CPS. The CPS in question consists of a robot platform mounted with 360-degree vision of the surrounding area, and object detection software. The area surrounding the robot is visualized from above in Fig. 26 as a circle with three radii. These radii separate the area into three distinct zones spanning the spaces of 0-3, 3-7, and 7-10 meters originating from the edges of the robot. These zones are "mitigation zones" with increasing threat-levels as the distance to the robot decreases. Due to the safety risks involved for humans, stricter safety requirements are defined based on proximity to the robot. The critical question to ask and evaluate is simply, what objects are in any of the given mitigation zones? This forms the basis for the implementation's mitigation planning. The mitigating actions are defined with increasing strictness in Tab. 7 under Sec. 3.3.

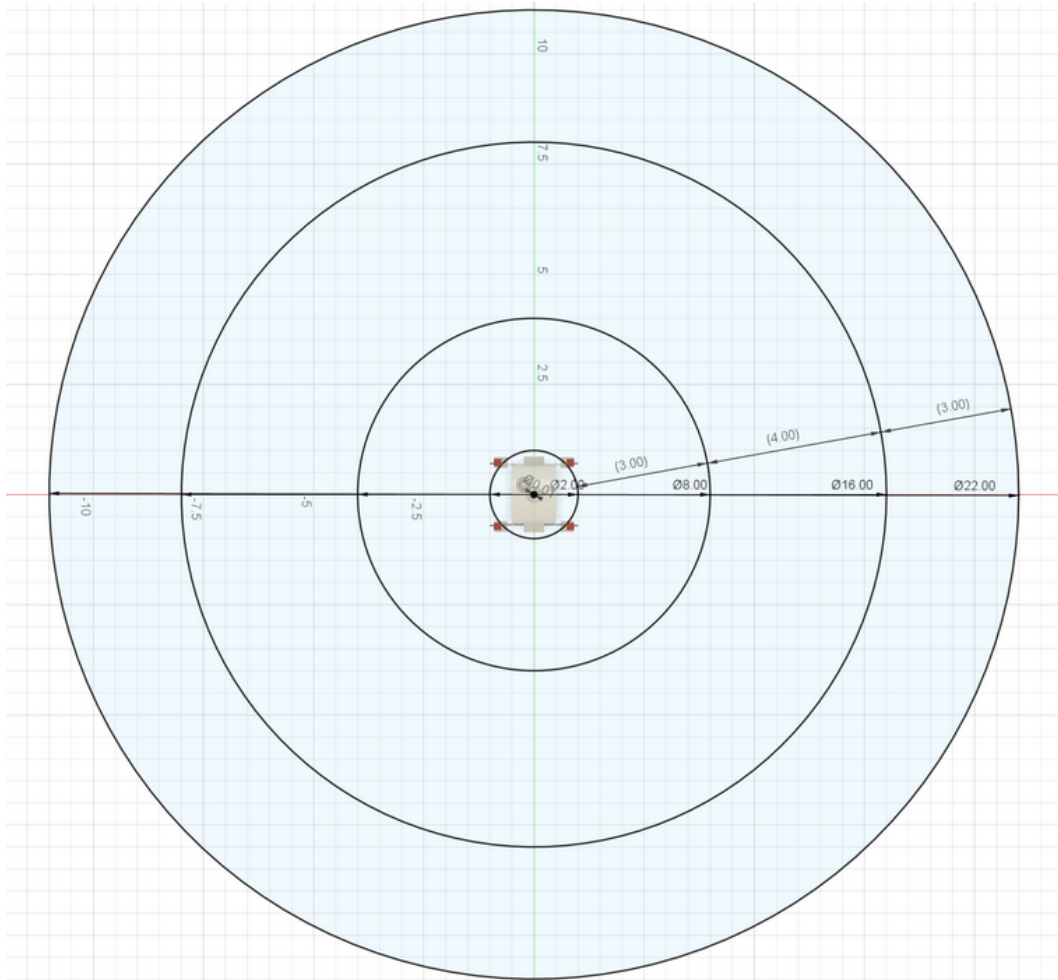


Fig. 26: CAD of mitigation zones in scale with robot to demonstrate dimensions. In reality the green zone (outer zone) goes indefinitely from the boundary of seven meters. This is only limited by the quality of the sensor. In this case the RealSense D435 has a graded performance up to ten meters.

4.1.3 Requirement Declarations

Requirement declaration is crucial for translating system specifications into formal, verifiable statements that can be monitored. To declare a robust model that performs reliably, NASA's Formal Requirements Elicitation Tool (FRET) was used. FRET allows for the specification of the requirements to be written in a structured, restricted natural language called FRETISH , as seen in Tab. 10. The table contains all system specification requirements used in this thesis. During requirements markdown process in FRET, multiple iterations are often necessary to model a system properly. Fortunately, FRET offers tools for checking the finished model. It is recommended to follow the workflow shown in Fig. 8 in Sec. 2.2.1. Once a model is deemed realizable by FRET, the project overview will look similar to Fig. 27. In this overview, each requirement is represented by a circle: the color green indicates a completed specification, white for undefined requirements, and red for unverifiable ones.

Building upon the variable mapping information in Sec. 2.2.1, Tab. 9 lists all variables with their type information and a short description. These correspond to the requirements in Tab. 10, and without the correct variable mapping this specification model would not work. Understanding the correct mapping of inputs and outputs is crucial for creating a functional model. In the case of unrealizable results during testing, one should foremost reconsider the variable types. Secondly, follow the workflow in Fig. 8. By following our example, it should be easier to understand how to set up the variable mapping correctly.

There are only two inputs in Tab. 9: classifier and distance to target. These are the two readings acquired from the robot's sensors. You could therefore look at FRET inputs as sensory inputs or uncontrollable data. Outputs represent different states of individual actions the robot can take, such as halting, slowing down, or turning off ultraviolet lights, all being controllable data. OpState represents the current active mitigation state of the robot and can have any value from 0 to 3, based on the four mitigation plans mentioned in Sec. 4.1.2 and 3.3. `dgt_3` and `dgt_7` (dgt, distance greater than) are what we call auxiliary variables and define our distance thresholds for mitigation zones. FRET needs to know at what points the mitigation states should hold, but it does not accept mathematical computations within requirement declarations. Mathematical operations should instead be assigned under "Variable Assignment in Copilot" as shown in Fig. 29.

Table 9: Variable Mapping.

Variable Name	**Variable Type**	**Data Type**	**Description**
Alert	Output	boolean	Mitigation to sound an alert under determined condition
Classifier	Input	integer	Identification variable for human detected by system
dgt_3	Internal	boolean	Auxiliary variable used to mark critical threshold in distance related to requirements
dgt_7	Internal	boolean	Auxiliary variable used to mark critical threshold in distance related to requirements
distance_to_target	Input	integer	Distance to the identified human given by the depth camera
Halt	Output	boolean	Mitigation to stop the robot under determined condition
OpState	Output	integer	Current active mitigation state of the robot
Slowdown	Output	boolean	Mitigation to slow down the robot under determined condition
TurnoffUVC	Output	boolean	Mitigation to turn off the ultra violet lights under determined condition

Table 10: FRETISH Requirements.

Requirement ID	FRETISH
classifier info: none = 0, trained person = 1, untrained person = 2 The different states are defined by the operationalstate requirements below	
classifier_assumption	sRobot shall always satisfy classifier=0 xor classifier=1 xor classifier=2
dtt_assumption	While !(classifier=0) the sRobot shall always satisfy distance_to_target>=0
operationalstate_0	While OpState = 0 sRobot shall always satisfy (!slowdown & !halt & !alert & !turnoffUVC)
operationalstate_1	While OpState = 1 sRobot shall always satisfy (!slowdown & !halt & alert & !turnoffUVC)
operationalstate_2	While OpState = 2 sRobot shall always satisfy (slowdown & !halt & alert & !turnoffUVC)
operationalstate_3	While OpState = 3 sRobot shall always satisfy (!slowdown & halt & !alert & turnoffUVC)
state_req000	While classifier = 0 sRobot shall always satisfy OpState=0
state_req101	While classifier = 1 sRobot shall always satisfy (dgt_7 ⇒ OpState=1)
state_req102	While classifier = 1 sRobot shall always satisfy (! dgt_7 & dgt_3 ⇒ OpState=2)
state_req103	While classifier = 1 sRobot shall always satisfy (! dgt_3 ⇒ OpState=3)
state_req201	While classifier = 2 sRobot shall always satisfy (dgt_7 ⇒ OpState=2)
state_req202	While classifier = 2 sRobot shall always satisfy (! dgt_7 & dgt_3 ⇒ OpState=3)
state_req203	While classifier = 2 sRobot shall always satisfy (! dgt_3 ⇒ OpState=3)

Fig. 27: FRET visualisation of requirements under the aRobot project, having formally verified the system. Each green circle contains a requirement of the system.

4.1.4 Monitor Implementation

To better illustrate the implementation of the monitor, it is beneficial to provide an example. Consider a safety requirement based on the presence of an untrained person, R4, defined in Tab. 8. The FRETISH translation of this natural language requirement can be input into the FRET Requirements portal as shown in Fig. 28. In the portal the different aspects of the requirement are given a field description, and the resulting semantics can be analyzed with the diagram assistant. Parent requirements IDs can also be used in the cases where requirements are dependent on each other. Otherwise, the process is straightforward, and much can be learned by studying the templates given by FRET. Previously inputted variable names can be found in the glossary, ensuring that common variables shared between requirements are not duplicated. The auxiliary variables of the given FRET requirement will need to be defined in the FRET Variable Mapping portal. For the requirement in Fig. 28, this would mean that `dgt_7` and `dgt_3` need to be defined. This is done in the variable mapping section, where thresholds like `dgt_7` (distance greater than 7 meters) are declared, as illustrated in Fig. 29.

Fig. 28: FRET Requirement example.

Update Variable

FRET Project
sRobot

FRET Component
sRobot

Model Component

FRET Variable
dgt_7

Variable Type*
Internal

Data Type*
boolean

Variable Assignment in CoPilot*
distance_to_target > constant 7

☐ Lustre ☑ CoPilot

Fig. 29: Mapping of an internal variable for a distance greater than seven meters.

As mentioned in the FRET realizability manual, auxiliary variables must be assigned using the Verimag Lustrev6 syntax and can only contain defined variables. In this case, `distance_to_target` and `classifier` are variables mapped in the Variable Mapper as integer inputs. This follows the logic that both the distance value and the class are values provided by sensors, and not outputs for the monitor to control.

After exporting the finished specification file with Copilot syntax, which includes all requirements for the project, a CSV file containing a variable database must be created. The CSV file should contain the FRET variables, variable types, and associated ROS topics. The CSV structure is illustrated in Fig. 11, and further information regarding database setup can be found on the OGMA github. Once the necessary files are ready, including the JSON specification file from FRET and the database CSV file, then the complete workflow for the OGMA software can be followed. The guide is provided in Fig. 30. In this workflow example, the JSON file is named sRobotSpec, and the CSV file is named sRobot. Remaining file names are for files created during compiling and monitor generation, and can be changed.

Table 11: Variable Database Example.

Variable Name	Variable Datatype	ROS Topic	Topic Datatype
”distance_to_target”,	”int64_t”,	”/scan”,	”Int64_t”
”classifier”,	”int64_t”,	”/sRobotClassifier”,	”Int64_t”
”alert”,	”bool”,	”/sRobotAlert”,	”bool”
”halt”,	”bool”,	”/sRobotHalt”,	”bool”
”slowdown”,	”bool”,	”/sRobotSlowdown”,	”bool”
”state”,	”int64_t”,	”/sRobotState”,	”Int64_t”
”turnoffUVC”,	”bool”,	”/sRobotTurnoffUVC”,	”bool”

```
OGMA workflow
#Generate the monitor from FRET specification:

OGMA fret-component-spec --target-file-name monitor
--fret-file-name sRobotSpec.json > monitor.hs

#Compile the monitor:

cabal v1-exec -- runhaskell monitor.hs

#Generate the ROS monitoring application using FRET requirements
and the variables database:

OGMA ros --fret-file-name sRobotSpec.json --variable-db sRobot.csv
--app-target-dir ros_demo

(The application will be generated in a package by any name
given to the input argument --app-target-dir)

Change "-" to "_" in the monitor.hs, if necessary
(errors could arise otherwise).

Move the .c .h and types.h files into ros_demo/src/ folder.

#In ros_demo/src/copilot_monitor.cpp
remove "prop" from handlers, if they cause issues.

#In ros_demo/src/copilot_logger.cpp
remove "prop" from handlers, if they cause issues.
And in the case of 'warning: unused parameter ‘msg’':
remove msg from SharedPtr in callback functions

#If you plan on using copilot_logger:
In ros_demo/src/CMakeLists.txt
uncomment add_executable section for copilot_logger,
add copilot_logger as an installation TARGET

#Everything should be set up correctly now, test by
going to the workplace directory and colcon build:
cd ~/ROS˜2_ws && colcon build

#Running the monitor:
ros2 run copilot copilot
#If uncommented from CMake:
ros2 run copilot copilot_logger
```

Fig. 30: Workflow of pipeline solution through OGMA.

The rest of this section will explore the monitoring application we have just created through an example C++ code. In Lst. 4, the generated monitor code runs the CopilotRV node. This node checks the system's states and reports violations when requirements are broken. This functionality is achieved through the use of callback functions that execute the step function each time a topic is updated. If the step function detects a violation, then the handler function for that requirement will forward the violations message, publishing it on a handler ROS topic. This is later addressed by the safety controller.

Listing 4: Example of OGMA-Copilot generated monitor.

```
1  // Define the monitoring node:
2  class CopilotRV : public rclcpp::Node {
3    public:
4      CopilotRV() : Node("copilotrv") {
5      // Define subscribers and publishers
6        classifier_subscription_ = this->create_subscription<std_msgs::msg::Int64>
7        ("/sRobotClassifier", 10,
8          std::bind(&CopilotRV::classifier_callback, this, _1));
9          ...
10     }
11     // Define monitor violation publishers:
12     void handleroperational_state0() {
13         ...
14     }
15     // Define
16     void classifier_callback(const *ROSDataType* msg)
17         ...
18         step()
19     }
```

OGMA uses past-time Linear Temporal Logic (ptLTL), where time is considered discrete, meaning time is modeled as a sequence of separate moments. The increment between successive events are often referred to as steps. For OGMA, the triggering of the step function is the defining moment when the monitor evaluates whether the system complies with the requirements. As shown in Lst. 4, the step function is by default called at every variable callback. This is because the generated monitor will have the step function set up for stream-based communication, whereas ROS operates on an event basis. This discrepancy requires manual adjustments to ensure that the generated monitor functions properly. There are several ways to address this issue, but one approach is to use timed calls of the step function running at an interval or to externally group publishing where the step function runs in-between groups, emulating a synchronized system. Proper integration will ensure that the system properties are refreshed before the states are verified by the monitor. This thesis went with the latter approach.

4.1.5 Flask-ROS Architecture

The Flask-ROS architecture consists of two main components:

Flask Web Server: Acts as the front-end interface, serving HTML pages and handling HTTP requests from web clients.

ROS 2 Node: Manages communication with ROS topics, services, and actions, processing incoming data from various sensors, actuators, and software.

The Flask application and the ROS 2 node are initialized separately to ensure non-blocking operation, as ROS runs in a parallel thread. Blocking occurs when the execution of one task prevents the progress of another. During early testing, this was a problem since when launching the ROS node the Flask website would not initialize. The Flask app was blocked by the ROS node spinning, meaning that it only began initialization after the ROS node had terminated. In Lst. 5 an example of the ROS multi-thread initialization is shown, which allows Flask to initialize normally by running the app.run command.

Listing 5: Multi Thread Execution of ROS Node.

```
1  def init_ros_node(app):
2      rclpy.init(args=None)
3      global node
4      node = TeleopNode(app)
5      executor = rclpy.executors.MultiThreadedExecutor()
6      executor.add_node(node)
7
8      try:
9          executor.spin()
10     except Exception as e:
11         node.get_logger().error('Exception in executor spin: %r' % (e,))
12     finally:
13         executor.shutdown()
14         node.destroy_node()
15
16 # Start ROS node in a separate thread
17 threading.Thread(target=init_ros_node, args=(app,),
18 daemon=True).start()
```

After initialization is complete, the software is ready to receive and post information. The Flask app cannot by itself connect to ROS topics and read the incoming messages. To achieve this, subscribers, callback functions, and data processing are implemented using the rospy library. The ROS node sets up subscriptions to all necessary topics and delivers the data to their respective callback functions. Each subscription has an associated callback functions that processes the incoming data and updates the Flask app's configuration dictionary. This process is used for simulated data, real values captured with sensors, and incoming images from the Yolov6 package. This process is also used for surveillance of the handler file topics, which report violations when they occur. The Flask app serves static files and templates, where various routes are handled. On these routes client requests are performed often delivering data to the front-end website through the use of HTML scripts. In Lst. 6 one such route is displayed delivering classification data from the Yolov6 package to the front end Flask application. The Flask routes handle

HyperText Transfer Protocol (HTTP) GET requests, which retrieve the latest data stored in the app.config dictionary and returnes it as a JSON response. This system allows web clients to fetch real-time data from the ROS system and sensors over the connected Wi-Fi network.

The Flask website is also capable of bi-directional communication, as shown in Sec. 4.1.7. This allows the user to send ROS messages back from the user interface, which is useful for testing or in the case of implementing or calling hazard mitigation actions.

Image Handling is served in a similar way. The ROS node processes incoming images from the /yolo_im topic, saves them, and updates the Flask app's configuration. The Flask app then serves these images via an HTTP route. With sufficient system hardware performance, these images can later be compiled into a video of the event.

Listing 6: ROS Flask.

```
1  class TeleopNode(Node):
2      def __init__(self, app):
3          super().__init__('teleop_flask')
4
5          qos_profile = QoSProfile(depth=10,
6          durability=DurabilityPolicy.VOLATILE)
7          self.bridge = CvBridge()
8
9          # ROS topic subscriber
10         self.classifier_subber = self.create_subscription(Int64,
11         '/sRobotClassifier', self.classifier_callback, qos_profile)
12
13     # Callback function
14     def classifier_callback(self, msg):
15         self.app.config['classifier'] = msg.data
16
17 # In the Flask app:
18 @app.route('/classifier')
19 def classifier():
20     classifier_value = app.config.get('classifier', 0)
21     return jsonify({'classifier': classifier_value})
```

4.1.6 Architecture Performance Optimization

The implementation of the Flask-ROS architecture, complete with the RealSense depth camera and classification software, is somewhat resource intensive for lower-end hardware. To better utilize hardware and reduce the workload on the computer's central processing unit, certain optimizations must be made. One optimization is to delegate tasks to the computer's graphics processing unit (GPU). The favorable performance of deep neural networks has led to the increased popularity of deep learning in recent years. However, this powerful tool comes at a high resource cost and can quickly become a bottleneck [51]. Fortunately, Pytorch supports hardware accelerators such as GPUs [52]. This support is primarily for NVIDIA hardware through CUDA, NVIDIA's parallel computing platform. There is also support for AMD GPUs

through ROCm and experimental support for Google's Tensor Processing Units, designed to accelerate machine learning tasks [51].

4.1.7 WebSocket Implementation

After testing multiple configurations of different WebSocket options mentioned in Sec. 2.2.4, Rosbridge proved to be the easiest option to configure and use in our case.

To set up the WebSocket connection for sending ROS commands from our Flask app to the ROS nodes controlling the robot simulation or system as a whole, the rosbridge_server package was utilized. This package provides a JSON API to ROS functionality for non-ROS programs, allowing them to communicate with ROS nodes using web protocols.

In our implementation, we launched the rosbridge_websocket using the following launch command in a separate terminal:

```
$ ros2 launch rosbridge_server rosbridge_websocket_launch.xml
```

This command initiates a WebSocket server that listens for incoming connections and allows connected users to send commands to ROS nodes through the WebSocket interface. This setup is crucial for the web-page platform, which aims to monitor the runtime verification. The WebSocket reduces the latency between the Flask page and ROS nodes, allowing data to be almost instantaneously displayed on the web page when it is received on the ROS topics. With Rosbridge, it becomes possible to create a seamless communication channel between the Flask application and the ROS environment. This enabled the Flask app to send commands to the robot simulation, facilitating real-time interaction and control. A feature which allowed testing of mitigation actions such as calling a shutdown of the robot during field tests, see Sec. 5.3. The WebSocket connection established by Rosbridge supports various ROS operations, including publishing to topics, subscribing to topics, and calling services, all through a straightforward JSON-based protocol.

Implementation Steps:

1. **Setup Rosbridge Server**: Ensure rosbridge_server is installed and properly configured in your ROS 2 environment.
2. **Launch Rosbridge WebSocket**: Use the provided command to launch the WebSocket server, as described in Sec. 4.1.7.
3. **Configure Flask App**: In the Flask application, use a WebSocket client library to connect to the Rosbridge WebSocket server. In our implementation the Rosbridge is connected through the HTML template file. In the example code Lst. 7 the structure behind how the injection buttons send ROS messages from the website is shown. In this example, only the dtt_assumption is implemented, which sends a negative distance value, causing the distance assumption requirement to be violated.

This integration simplifies the development and testing process, making it easier to ensure that the robot simulation behaves as expected under various conditions. Leveraging the flexibility of Rosbridge, our platform can effectively monitor and control the ROS-based system, ensuring robust runtime verification.

Listing 7: ROSBridge Setup for use with Injection Handler.

```
<script>
    var ros = new ROSLIB.Ros({
      // default address for ROSBridge;
      url : 'ws:// localhost :9090'
    });

    ros.on('connection', function () {
      console.log('Connected to websocket server.');
    });

    ros.on('error', function(error) {
      console.log('Error connecting to websocket server: ',
      error);
    });

    ros.on('close', function () {
      console.log('Connection to websocket server closed.');
    });

  function injectHandler (handlerName) {

    var injectiontalker = new ROSLIB.Topic({
    ros: ros,
    name: '/ injection_detection ',
    messageType: 'std_msgs/ String '
    });

    var injectionmsg;
    // data order:
    // class, distance, state, slowdown, halt, alert, uvc

    if (handlerName == "dtt_assumption") {
      var injectionmsg = new ROSLIB.Message({
      data: "1,-1,3, false, true, false, true"});
      }

     injectiontalker.publish(injectionmsg);
  }
</script>
```

4.2 Simulation and Field Testing

Simulation and field testing are critical phases in the development and validation of robotic systems and automated applications. This section provides an overview of the methods and tools employed to rigorously test and validate system functionality under controlled and real-world conditions. Through a combination of simulated environments and practical field tests, the aim is to ensure that the system meets its design requirements and performs reliably. This section is divided into three parts: Field Testing (Sec. 4.2.1), Simulation (Sec. 4.2.2), and RealSense Setup with Inferer (Sec. 4.2.3). Each giving a detailed description of the setup and execution.

4.2.1 Field Testing

For the accumulation of classification data, a simple sequence of walking towards and away from the camera was repeated several times over the span of ten minutes. The sequence was documented at a capture rate of one frame every three seconds. Two individuals were present; one wearing a vest to classify as worker, and another in civilian apparel classifying as adult. This process was conducted at both noon and in the evening.

A similar method was used when the Thorvald robot was active. Running the monitor through a laptop placed on the top of the robot, with the RealSense camera pointed forwards to collect data. All field tests with the Thorvald robot were performed in a polytunnel, driving in parallel to strawberry hedges. The worker-class subject walked back and forth unpredictably, while the robot was driven manually. All while performing data collection of the monitor behavior. Testing was performed under dry conditions on sunny and cloudy days, with additional tests to assess how natural light affected the classification data.

4.2.2 Simulation

In this study, the Gazebo 11 software was employed as the simulation environment. Gazebo is a standalone application independent of ROS and ROS 2, offering rapid testing of realistic simulations in both simple and complex environments as well as being a robust platform familiar to many in the field of robotics.

The open-source TurtleBot3 package was chosen for its widespread use in educational robotics. Designed to work seamlessly with ROS and ROS 2, the TurtleBot3 facilitates quick setup of simulations for demonstration purposes, and is fully compatible with Gazebo. The integration of Gazebo with ROS 2 was done through a set of packages called gazebo_ros_pkgs. For this project, custom CAD models for the robot, as shown in Fig. 31, and mitigation zones were designed in Fusion 360 to more accurately reflect the system addressed in this thesis. The simulation environment was configured to replicate an agricultural setting, as provided by the FieldRobotEvent, Virtual maize field GitHub (2023), `https://github.com/FieldRobotEvent/virtual_maize_field`.

Fig. 31: Fusion 360 model of the Thorvald robot, used for demonstration in the gazebo environment.

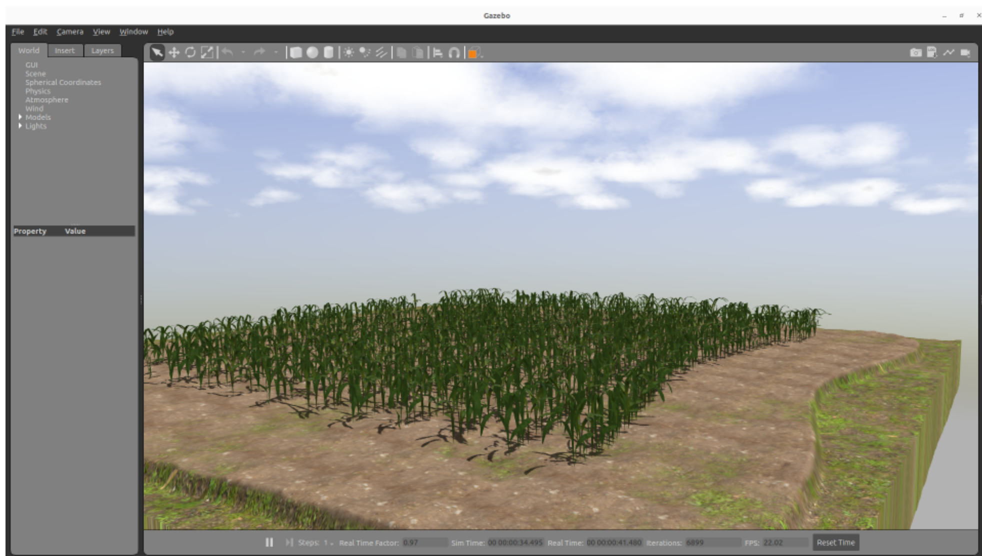


Fig. 32: Gazebo environment. *Note.* From the Virtual maize field GitHub, by FieldRobotEvent [53].

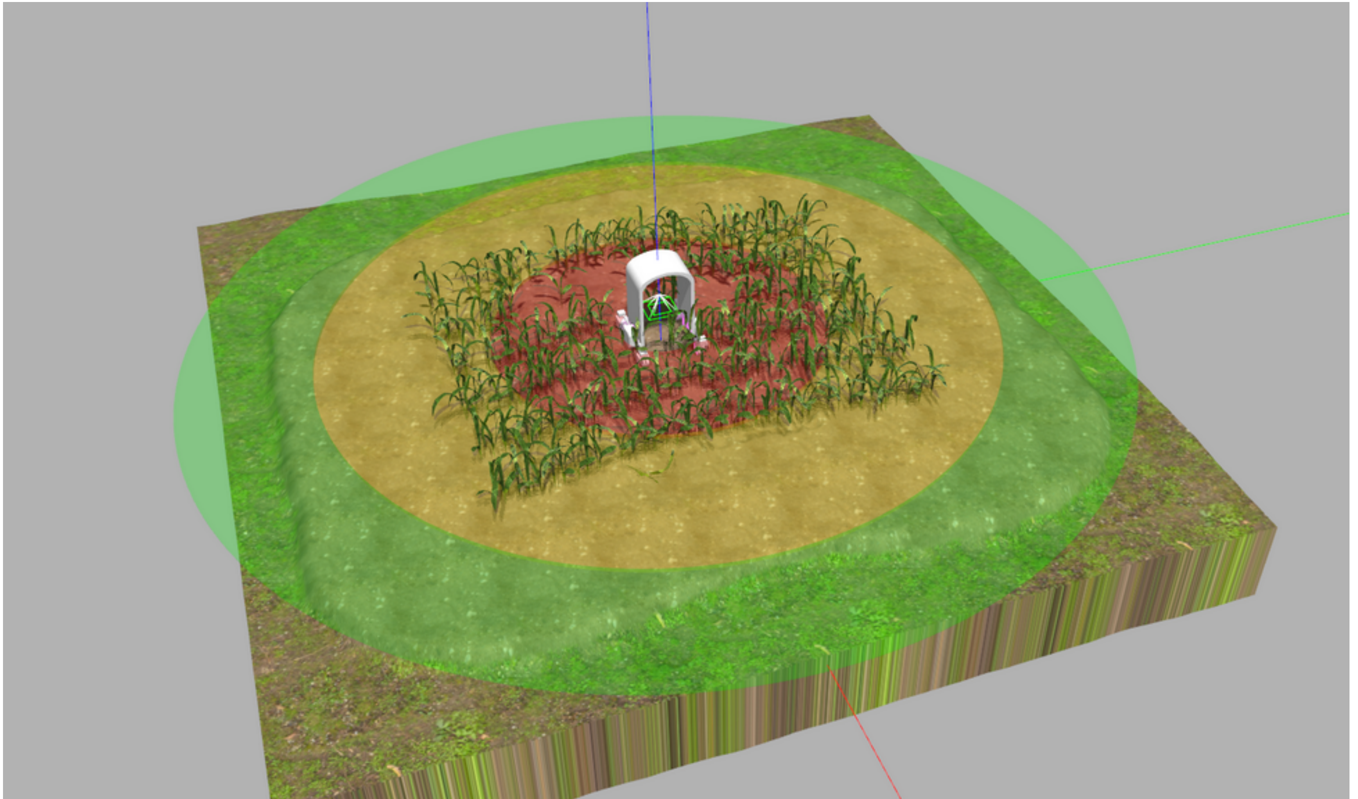


Fig. 33: Gazebo simulation of mitigation zones. See Sec. 4.1.2 for context.

A virtual joystick and teleoperation ROS 2 node facilitating real-time control and simulation interaction were provided by Lars Grimstad and translated to ROS 2 by co-student Henrik Nordlie. This effectively gave control of the robot's directional movements during simulation, only requiring topic inputs on whether the robot was to slow down or halt operation by altering the output from the velocity callback of the teleoperation node.

A spotlight was constructed for the Gazebo environment to simulate an ultraviolet light module, and a black cylinder to represent a human. Consequently, to manipulate the models' positions based on the robot's pose data and allow ROS 2 integration, custom Gazebo plugins were made. The first Gazebo plugin consists of a "ToggleLightPlugin" class that inherits from the gazebo::ModelPlugin class that manages Gazebo models during simulation and a ROS 2 node that allows for communication with the ROS 2 platform. The plugin effectively toggles the spotlight based on Boolean messages by listening to the /sRobotTurnoffUVC ROS 2 topic. The second Gazebo plugin handles positional variables for the spotlight and mitigation zone models to follow the robot platform through the environment. This plugin also inherits from the gazebo::ModelPlugin and interfaces with ROS 2 to listen to topics. The spotlight mimics the robot's complete pose, but the mitigation zones and cylinder only trace the robot's X and Y coordinates. The cylinder additionally moves linearly along the X-axis of the robot based on the float values of the /scan topic to visualize the depth of a person within the robot's field of view. This approach more clearly visualizes the robot's actions based on the identified object's position relative to it. The spotlight model was defined with the toggle plugin configuration within the .world file of the simulation. The other models are defined within the same .world file, but the follower plugin is included in the robot's SDF model.

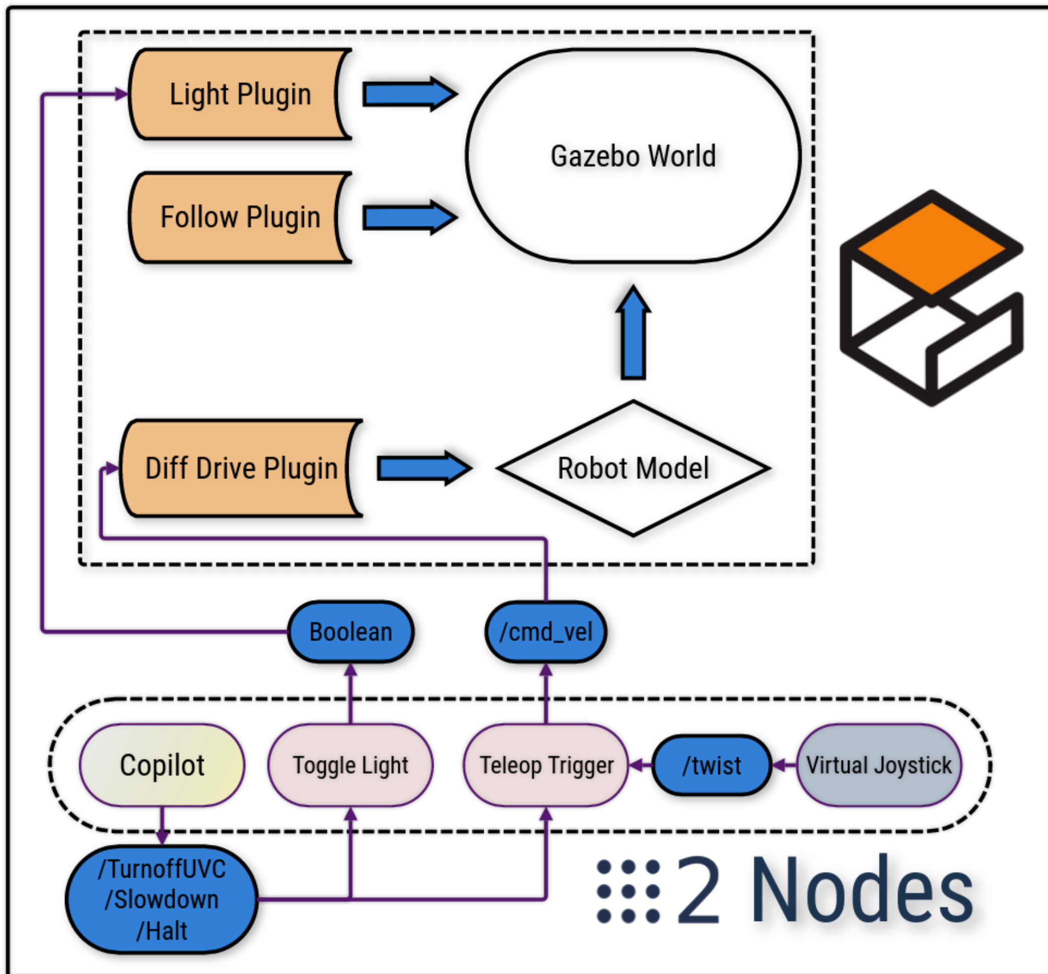


Fig. 34: System diagram of simulation setup consisting of Gazebo and ROS 2 nodes.

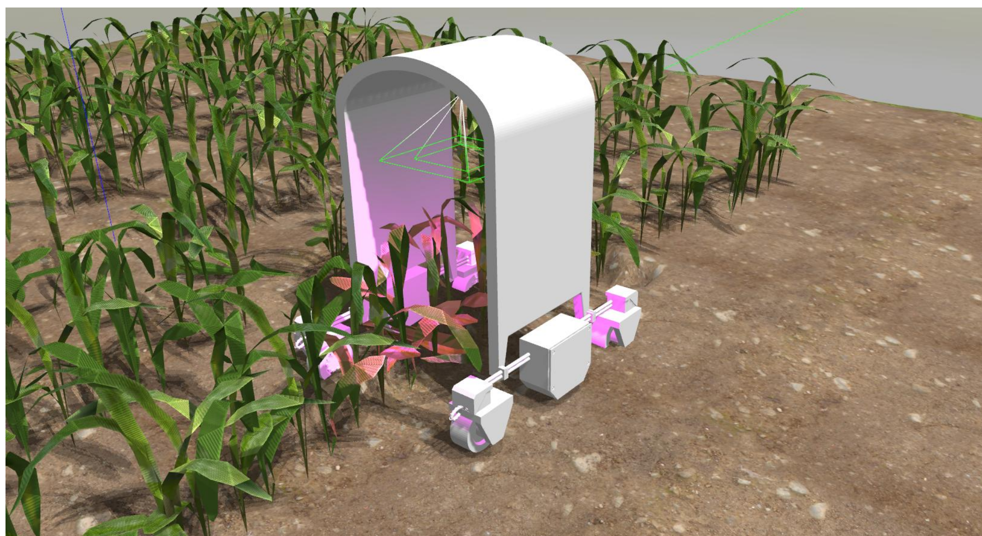


Fig. 35: Simulation of active Thorvald robot applying UV lights in the field.

4.2.3 RealSense Setup with Inferer

This section explains the initialization and processing of image data from the RealSense D435 camera. Most steps are mentioned, but some related to processing are skipped for the sake of brevity. For a more detailed explanation of the initialization, refer to the RealSense-ROS installation page at Github.

① To begin using the RealSense D435 camera for linux operating systems, several packages needs to be downloaded. Firstly, the RealSense library and camera distros, shown here:

```
$ sudo apt install ros-<ROS_DISTRO>-libRealSense2*
$ sudo apt install ros-<ROS_DISTRO>-RealSense2-*
```

② When using the YOLO package, you will need additional packages like CV2 and pytorch. If this is the case, then follow the installation guide for the chosen YOLO model. Here the YOLOv6 is deployed.

③ Before running any YOLO prediction, the camera node must be launched. With the D435 model, the RealSense2 camera package is used. Connect the camera via USB and use the following launch command:

```
$ ros2 launch RealSense2_camera rs_launch.py enable_rgbd:=true
enable_sync:=true align_depth.enable:=true enable_color:=true
enable_depth:=true
```

Among other things, this ensures that the resolution and FOV of the RBG and depth image are the same. This is required by the current YOLO solution.

④ To run the YOLO prediction on the image:

```
$ ros2 run yolov6 inferer
```

If this command results in an import error related to allocated memory, try the following command:

```
$ export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6:/
usr/lib/x86_64-linux-gnu/libgcc_s.so.1:$LD_PRELOAD
```

⑤ The RealSense should now be functioning with the Inferer YOLO model running. Publishing classified images to corresponding topics with related depth values. The path to the PT and YAML file containing the corresponding labels is located at the bottom of ”yolov6/yolov6/core/inferer.py”. Any software edits will most likely be made in the Inferer code.

The Inferer code handles much of the post-processing on the YOLO-processed RealSense image. This includes running the YOLOv6 model on the RGB image, publishing data to ROS topics, defining the depth value to the class subject, visualizing the class and borders of the class, and showing the distance zones based on color. The latter is shown in Fig. 8, illustrating how the RGB image is labeled and bordered based on classified subjects, with the border color indicating the zone in which the distance is measured.

Listing 8: Definitions of Prediction Box Colors and Distances in Inferer.

```
if draw_predictions:
    class_num = int(cls)
    label = None if hide_labels
    else (self.class_names[class_num]
    if hide_conf
    else f'{self.class_names[class_num]} {conf:.2f}')

    #Coloring the Boxes Based on the Zone
    depth_int = int(depth.data)
    zone_msg = String()

    if depth_int < 3000:
        zone_msg.data = "red"
        zone_color = (0, 0, 255)

    elif depth_int < 7000:
        zone_msg.data = "yellow"
        zone_color = (0, 255, 255)

    else:
        zone_msg.data = "green"
        zone_color = (0, 255, 0)

    self.plot_box_and_label(img_ori, max(round(sum(img_ori.shape)
    / 2 * 0.003), 2), xyxy, label, color=zone_color)

    self.zone_pub.publish(zone_msg)
```

The RealSense camera delivers the depth information in an image of similar size to the RGB image. This depth information is not inherently connected to any sections or classifications made, so selecting the correct depth information to use is done via code. The depth zones are created from the zones identified during classification. These zones contains a lot of information, and simply taking the average will not yield accurate results due to background and foreground elements that are separate from the subject. To address this issue, a `middle` section of the depth zone is separated, and the mean value is calculated based on this new section. The assumption is that the classification zones will always center around the subject, making it likely that the center of the zone contains mostly accurate depth information. Fig. 9 shows a section of the Inferer code used for depth calculation. The lambda function, labeled `middle`, is designed to extract the central part of a N-dimensional array by slicing each dimension into its `middle` part.

If the result of the `middle` function returns an array, then the mean of the data is calculated and published. This is the distance value used by the monitor. If the `middle` function returns no data array, then the depth value is set to zero. The decision was made to set the depth value to zero when no class is returned, providing consistent results.

Listing 9: Prediction Box Processing for Distance Calculation in Inferer.

```
# Center Lambda Function
middle = lambda x: x[tuple(slice(int(np.floor(d/4)),
 int(np.ceil(3*d/4))) for d in x.shape)]

# Use the Lambda Function on Depth Image
 middle_section = middle( depth_detection_section )
 if middle_section.size > 0:
     # Calculate the Mean Depth in the Middle Section
     depth_value = np.mean(middle_section)
 else:
     # Default Value
     depth_value = 0
```

4.3 Monitor Violation Injection Device

The main tool used for testing the system's specifications was the violation injection device. This tool consists of a set of buttons, each targeting a different requirement based on the handler topics. When pressed, the button will deliver a message to a ROS node over a Rosbridge WebSocket.

The message contains the target states required to force a specific violation to occur. When activated, the injection button pauses the data stream from the camera, ensuring that the violations states are not overwritten before the system has been verified by the monitor. The injection process also logs a message to the Violation Log Messages section, declaring which violation has been injected. If everything is set up correctly, a second log message is published afterward, declaring that a violation has been reported. The tool was instrumental in observing that all requirements worked as intended and in ensuring that the Copilot monitor performed as expected during testing. Fig. 36 shows a screenshot of the entire web page, where the injection buttons are located on the right side under "Handler Injectors". Below the buttons is where you find the log page. Fig. 37 provides a close-up of the buttons along with the log post used to inject violations. The violation messages come with a timestamp indicating when the violation was reported. Lastly, Fig. 38 focuses on the sRobot state information from the web page in Fig. 36, displaying the latest topic info and the last reported violation. This provides a comprehensive overview and a valuable tool for understanding why a violation occurred.

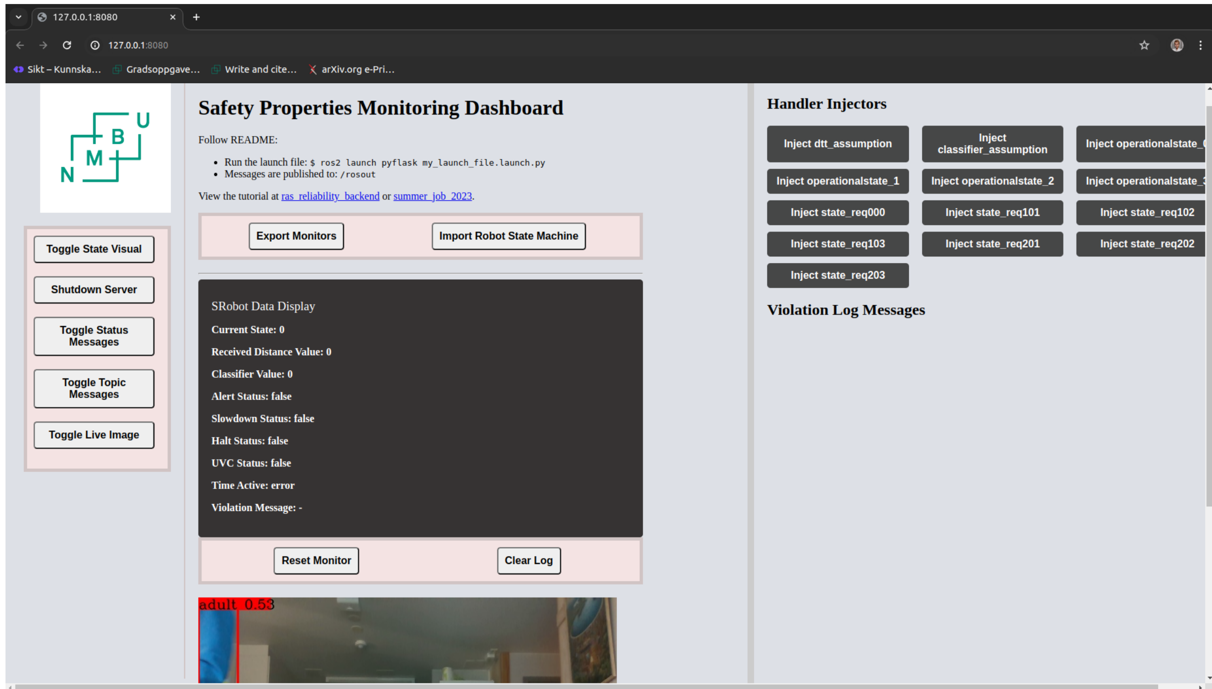


Fig. 36: Snapshot of webpage. Injection buttons on the right side, information regarding robot state on the left side.

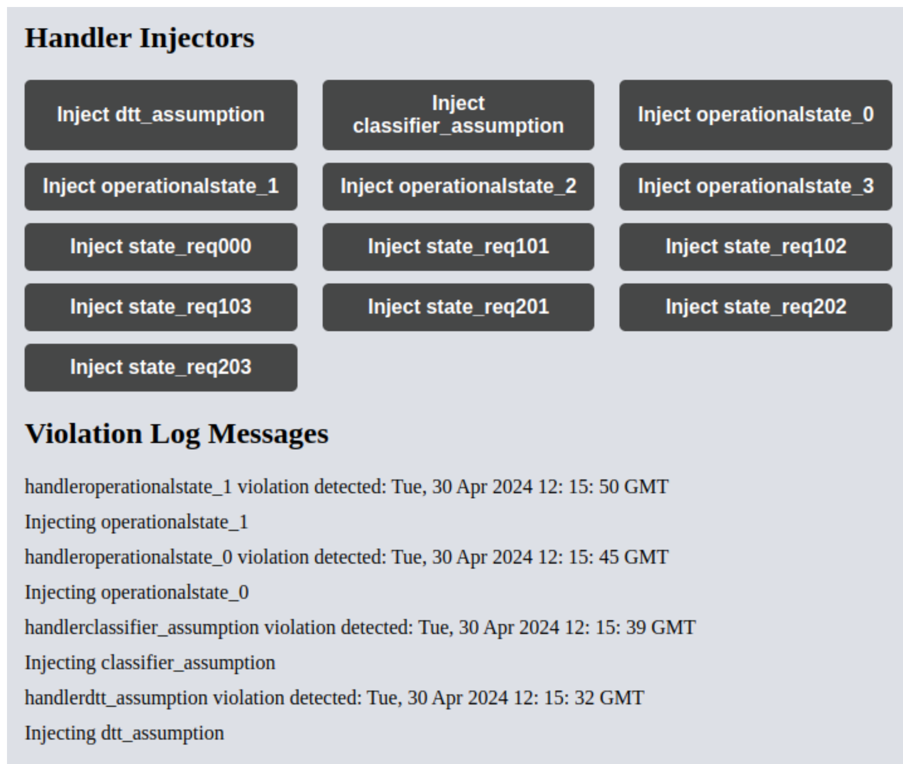


Fig. 37: Violation injection buttons. Each button triggers a violation scenario to be published that only breaks that single requirement which the buttons are named after.

SRobot Data Display

Current State: 3

Received Distance Value: -1

Classifier Value: 1

Alert Status: true

Slowdown Status: false

Halt Status: true

UVC Status: true

Time Active: error

Violation Message: handlerDtt_assumption violation detected: Tue, 09 Apr 2024 14: 37: 58 GMT

Fig. 38: Display block on the webpage, from Fig. 36, containing all the variables of the test system and the latest violation message.

4.4 Research Ethics

When collecting data in the field and during testing, human subjects were used. To maintain the privacy of unwilling volunteers, all images containing identifiable people were deleted after classification data was collected, retaining only anonymous information related to class and distance. Images with identifiable people that have not been deleted were retained with the agreement of the subject in the frame, maintaining the ethical responsibility of the thesis.

5 Results

Runtime Verification (RV) relies on the creation of monitors that observe the system's execution traces against predefined properties or specifications. These monitors can provide timely warnings or trigger mitigation actions when deviations from the expected behavior are detected. The effectiveness of an RV system depends heavily on its ability to perform with minimal overhead, ensuring that the monitoring process does not unduly burden the system being observed.

This section explores the empirical data behind the implementation of the Copilot RV framework, generated by OGMA from FRET requirements, when applied to a robotic system. Specifically, it focuses on the integration effects of RV within the Robot Operating System (ROS) environment in both simulation and field testing. The primary objectives include assessing the overhead introduced by the RV monitors, evaluating their performance under different operational conditions, and demonstrating their effectiveness in detecting and responding to safety violations.

The structure of the chapter is as follows:

Monitor Behavior, Sec. 5.1 - This subsection details the performance metrics of the monitoring software, including its overhead, the asynchronous nature of data publishing, and the latency between violation injection and detection.

Simulation, Sec. 5.2 - This subsection describes the methods used for simulating robot data and presents the results from simulated scenarios, demonstrating the monitor's functionality.

Field Testing, Sec. 5.3 - This subsection provides insights into the practical application of the RV system in real-world settings, highlighting the robustness and reliability of the monitoring framework. Additionally, it gives an example of a mitigation action tactic.

By systematically analyzing the performance and impact of the RV system, these results aim to contribute to the discussion of safer and more reliable robotic applications, ultimately facilitating their integration into human-centric environments.

5.1 Monitor Behavior

In this section, data regarding the performance of the monitor software is tested and displayed. The data here is important in determining the advantages and disadvantages of the system pipeline. During testing, the areas of performance which have been tested are: overhead of the system (Sec. 5.1.1), asynchronous nature of publishing (Sec. 5.1.2), and the time between injection of violation until report together with the delay caused by removing the LTL nature of the monitor (Sec. 5.1.3).

5.1.1 System Overhead

In investigating the performance of an RV tool, overhead is an important factor. The level of overhead is what limits the amount of information the monitor can evaluate and commit. To identify the system's bottlenecks, a robot scenario was established where messages were published at varying frequencies to the monitor, each time letting the monitor evaluate the nominal state. For this test, a nominal scenario was used, as violations are infrequent, thus testing the total number of messages the monitor can handle. This is similar to the ROSMonitoring overhead test, where a fixed nominal message was returned for each state [5]. The Hertz values used in the test also mirror some of the values used in ROSMonitoring for clarity when comparing the two RV platforms. The Hertz values tested were 5,000, 10,000, 12,500, 15,000, 17,500, and 20,000, corresponding to the number of messages published per second during runtime. The number of messages was distributed over seven different topics, as shown in the ROS DB-file in Tab. 5. To clarify, the number of messages published per second on each topic is equal to the average hertz rate, the topics summed up equal the Hertz value. The graph values in Fig. 39 indicates the average hertz across all seven topics, while Fig. 40 relays the performance decrease when the monitor is introduced compared to without. Fig. 41, 42, and 43 display the values gathered for three different publishing rates over the span of three minutes, showing how the average times diverge at higher frequencies.

In Fig. 39, data from two tests are plotted next to each other. The blue bars (without monitor) represent the average frequency of messages published without the monitor running. This data was gathered by averaging the time between each data publishing across topics. The orange bars represent the average frequency with the monitor running. For lower Hertz values, the monitor had little impact; at 10,000 Hertz, the natural variance in the data even showed better performance with the monitor. As expected, at higher Hertz values, significant overhead was introduced, and the monitor bar flattened out at around 1,800 messages per second per topic, totaling around 12,600 messages. Testing the upper bound of standalone publishing (without monitor) flattened out at around 22,300 Hertz, meaning 3,186 messages per topic.

To understand the performance decrease during testing, the overhead percentage formula (1) was applied. The results, seen in Fig. 40, show a peak performance decrease of 53.9 % at 20,000 Hertz.

$$\text{Overhead} = \left(\frac{P_{\text{standalone}} - P_{\text{monitor}}}{P_{\text{monitor}}} \right) \times 100\% \tag{1}$$

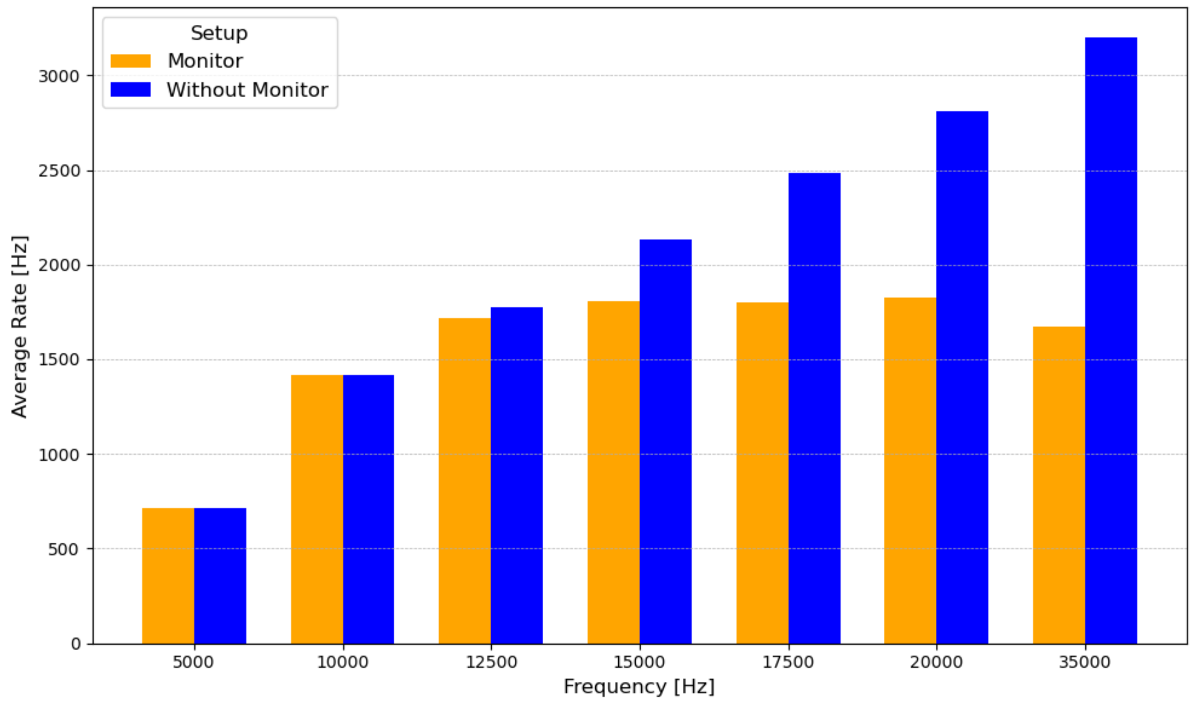


Fig. 39: Publish Frequency in Hertz.

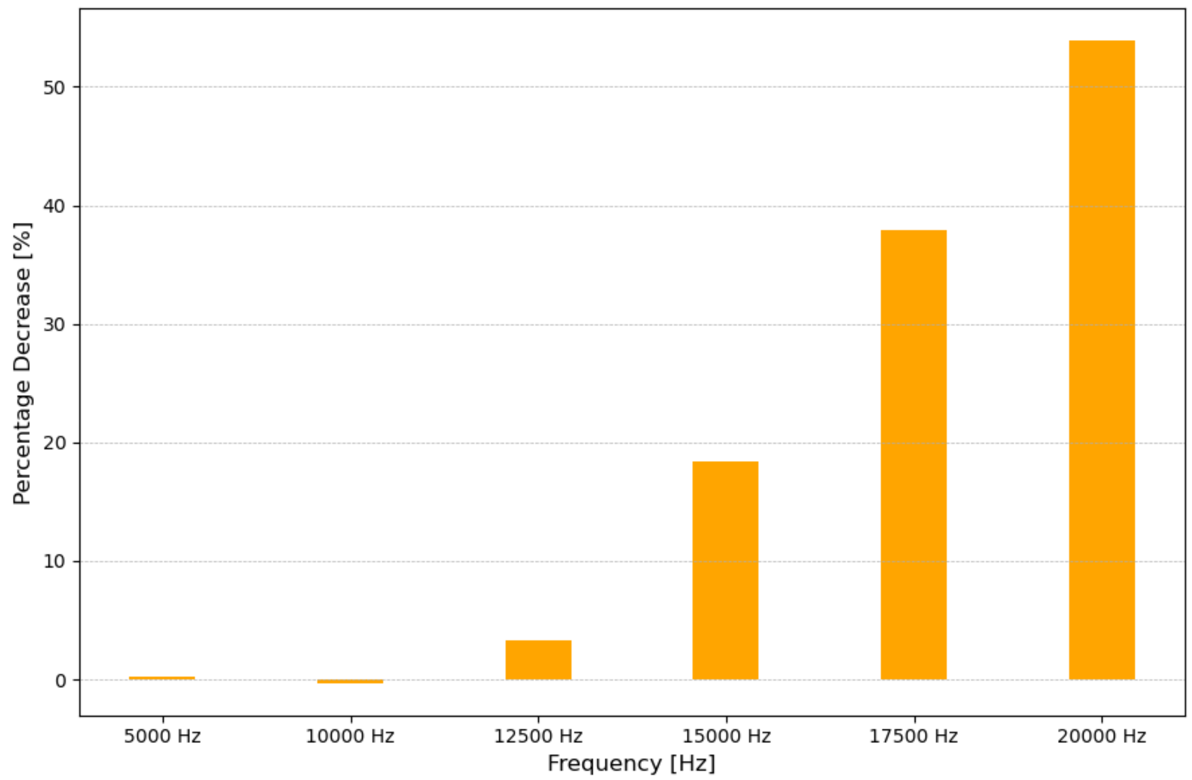


Fig. 40: Percent Overhead.

The average Hertz values were gathered by logging timestamps before each publish on the `turnoffUVC` topic. All topics were published in tandem, thereby logging the `turnoffUVC` average was the average across all topics. In Fig. 41, 42, and 43, it is observed that with larger Hertz values, the divide steadily increases between the baseline values and those with the monitor implemented. Initial values with the monitor sometimes show a rapid increase in the average Hertz due to initialization completing, stabilizing the values after a few seconds. In the figure where the total Hertz was meant to be 15,000, the monitor values plateau at around 1,800 Hertz. With the increased difference between the data with and without the monitor, the noise in the data becomes less visible, making the values easier to read.

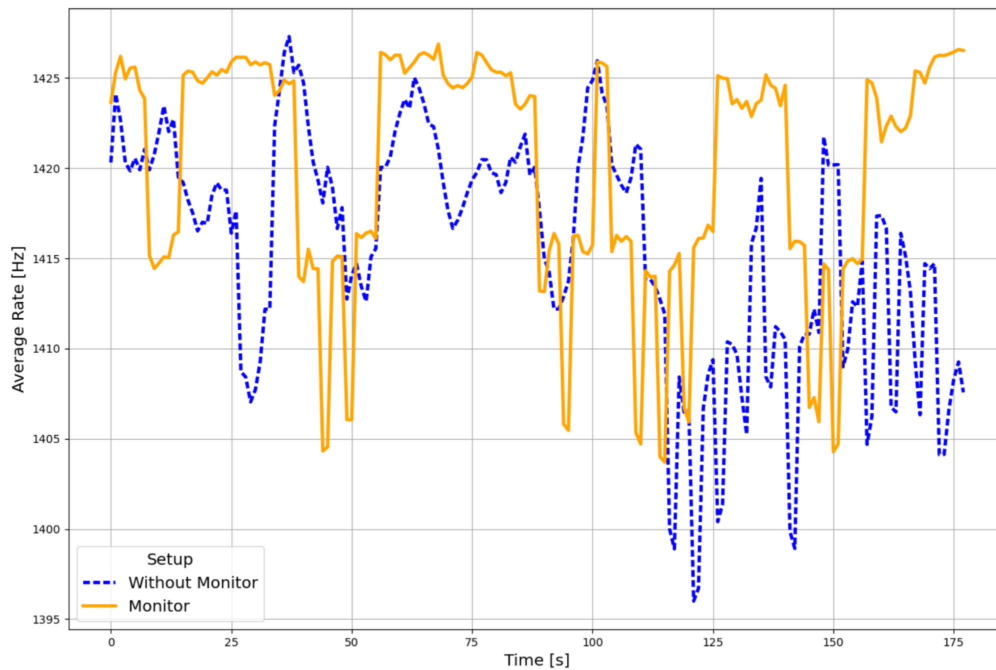


Fig. 41: Testing with a publishing speed of 10kHz over seven topics, both with and without the monitor running. Data collection was performed over the span of three minutes.

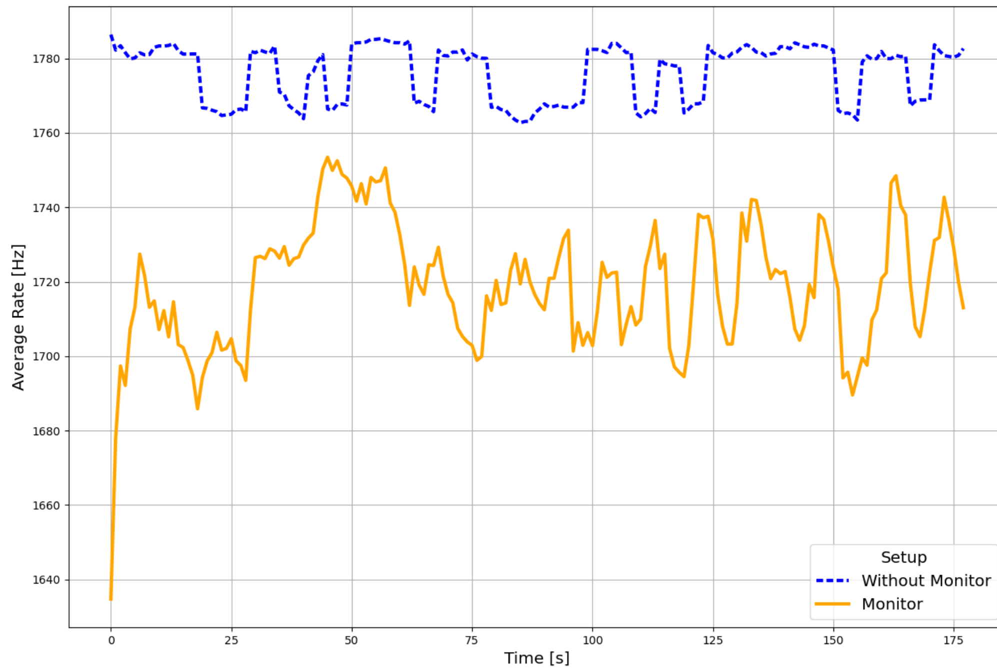


Fig. 42: Testing with a publishing speed of 12.5kHz over seven topics, both with and without the monitor running. Data collection was performed over the span of three minutes. From above 10kHz, the data starts to diverge, revealing the effect of the monitor on the performance of the software.

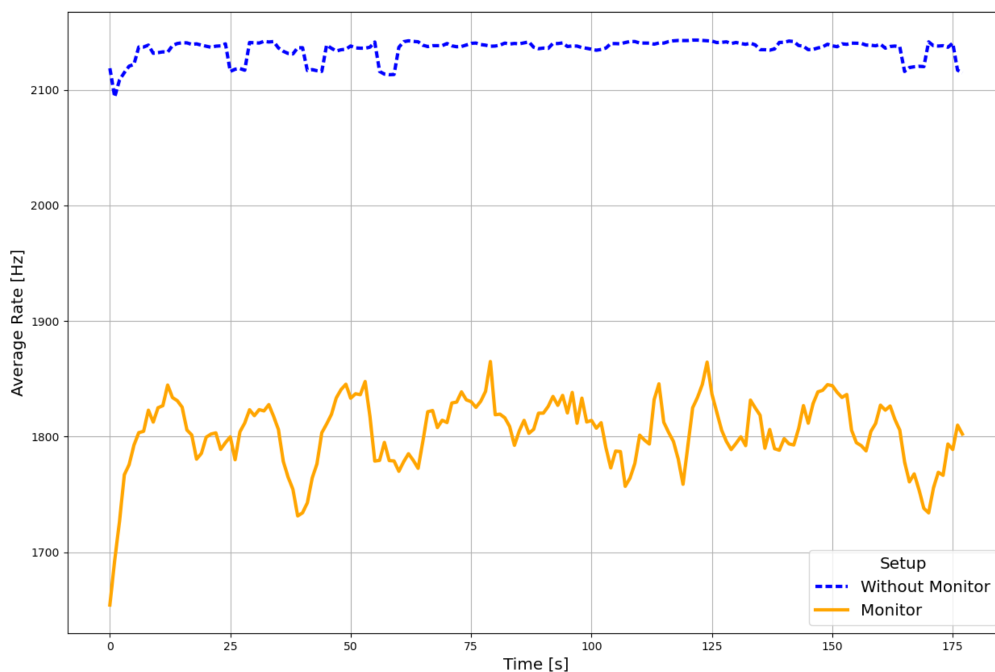


Fig. 43: Testing with a publishing speed of 15kHz over seven topics, both with and without the monitor running. Data collection was performed over the span of three minutes. Beyond 12.5kHz, the data with the monitor running starts to stabilize, indicating the upper bound publishing speed possible with the monitor in use.

5.1.2 Asynchronous Publishing

The original output from using OGMA was a ROS 2 monitor script intended for stream-based communication. Since ROS is event-based, multiple violations were observed in early testing, with the conclusion that the current setup did not work with the requirements as written. One way of fixing this is by changing the requirements to allow time gaps before states should be nominal, allowing the code to refresh all topics in time, using accepted timing commands like ”within” from Tab. 1. However, to keep the instantaneous nature of the requirements, a change was made to how the system checked the requirements. By moving the call of the step function, the code could use the requirements as written with very minimal performance impact. Fig. 44, 45 and 46 were collected on a Ryzen 3 5000 series, to determine why violations appeared. They show the asynchronous nature of publishing, which caused the violations since all topics were not updated before the step function call. This data was crucial in defining the method of calling the step function, either by calling it on a timed interval or always calling it after the last topic was refreshed, with the latter being used.

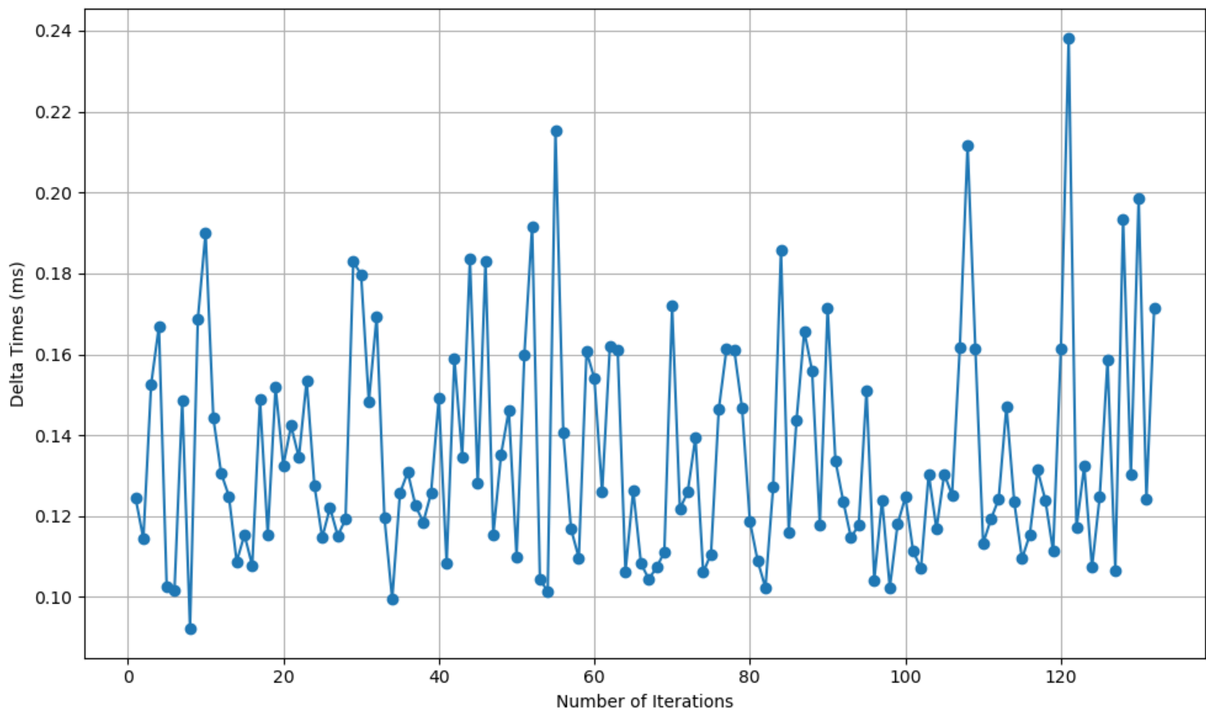


Fig. 44: Data results from testing the delta time from first topic publish to last. Testing the asynchronous nature of the publishing system, which can lead to monitor violations given how OGMA generates the monitor files.

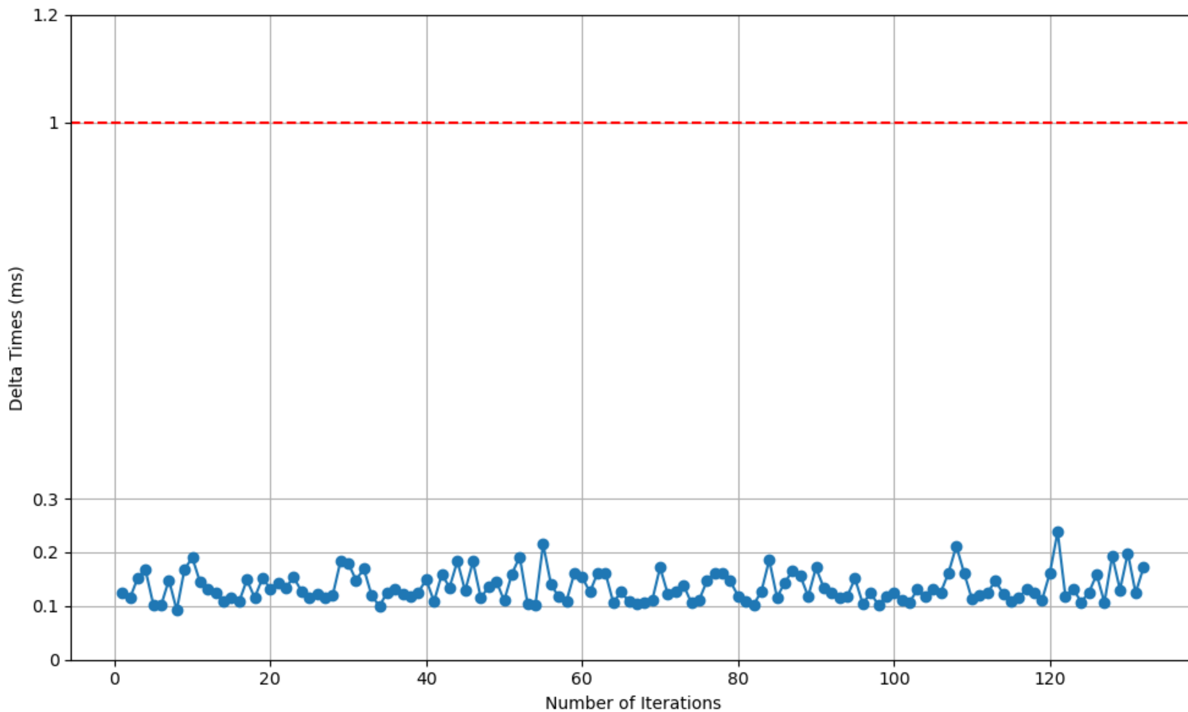


Fig. 45: Results from Fig. 44 compared to the threshold of one millisecond. One millisecond was the threshold for calling the step function in early testing phases. The issue related to asynchronous publishing can be solved either by editing the monitor call function or changing the requirements.

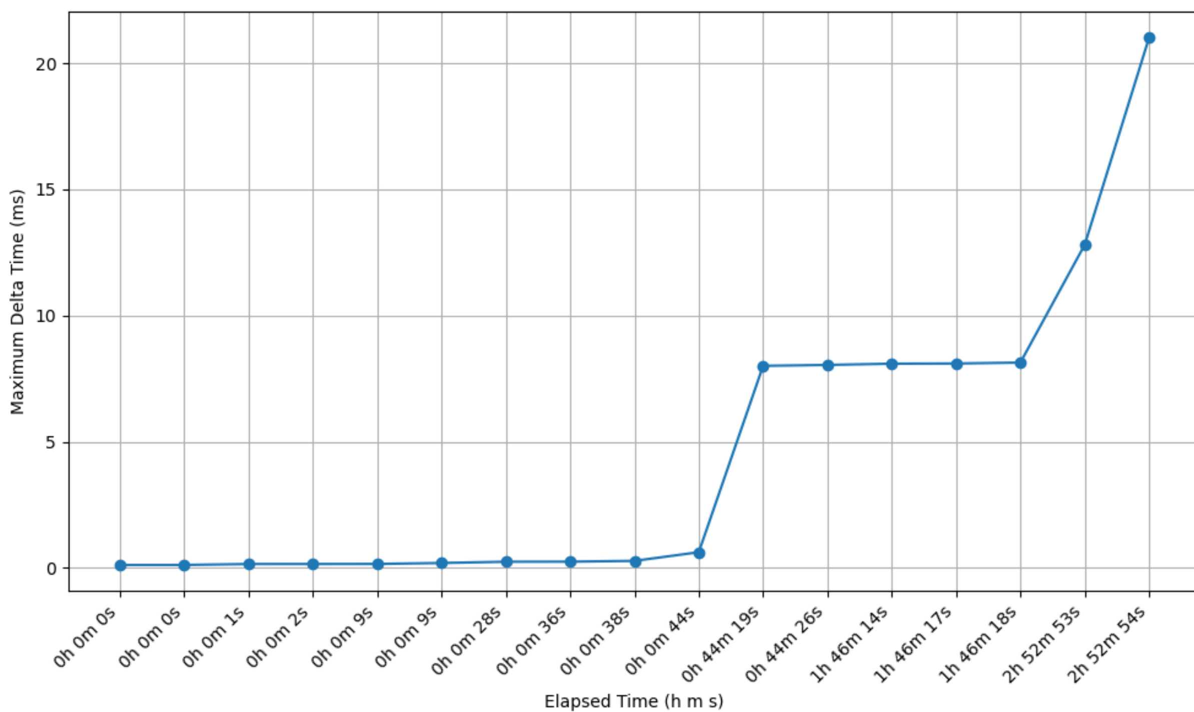


Fig. 46: Results from running the publisher node for around three hours, looking for the max delay between first and last publish in a strained environment. The highest value at 2h 52m 54s was achieved when the system crashed, resulting in a gap between first and last publish of around 20 milliseconds. Overall, the delta times were very consistent and low.

5.1.3 Violation Report

So far, the monitor shows it can handle a great volume of messages and that it is possible to configure the step function for event-based information. The following figures analyze the time it takes from reporting a violation state until the violation is flagged and logged. Without this knowledge, it is difficult to trust the monitor implementation, as safety-sensitive systems require quick responses when faults are detected. Firstly, in Fig. 47, an early version of the monitor code was running, logging the time from sending the violation state until a violation was logged on the handler topic. The handler topics are what the `copilot_monitor.cpp` and `copilot_logger.cpp` code communicate over; by intercepting these messages, the Flask website can display violations when they are flagged. The most important take away from the initial data regarding the violation report time was that it was fast enough to be viable for robotic system testing. On a later date, a larger test was performed, getting the average response time across over one hundred iterations. The results can be seen in Fig. 48. At the point of running the larger data collection, the monitor code and system had become more computational heavy, and the violation injection was performed from the Flask website. This might be the reason behind why the average values seem to be higher than in the initial tests, in Fig. 47. Still, the times were quite low, averaging at 33 ms. During testing, the LTL nature of the monitor often clogged log files because when a violation is reported, it will continue to be reported since the monitor also acknowledges violations of the past. To prevent the LTL method, a simple solution was applied were each time a violation was reported, the system would restart the safety monitor. In this way, the monitor memory was reset while all log files remained as normal. This is not a recommendable change as it adds a big delay between violations where nothing is reported. The delay caused by closing the CopilotRV node and relaunching it can be seen in Fig. 49. For the relaunch, an average of 1.5 seconds was spent without violations being reported, it was kept for testing purposes.

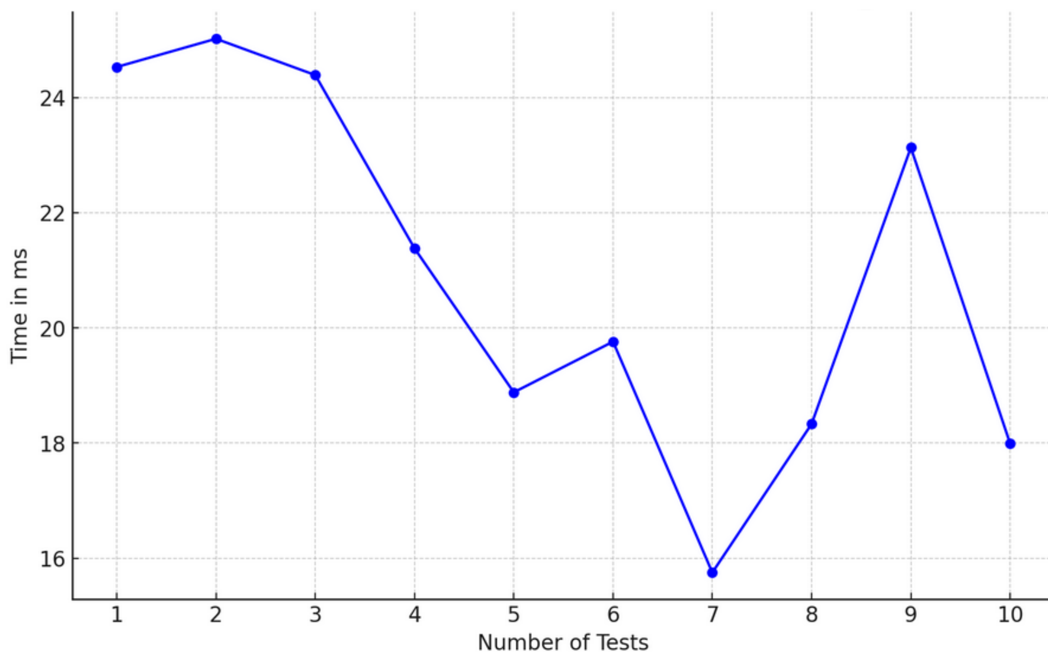


Fig. 47: Results from running the publisher node and checking how much time goes before a violations is reported. The step function in this case was run every 10ms. Therefore, any time collected is subject to a time discrepancy of 0-10ms delay where the violation could have been reported already if the step function was run instantly.

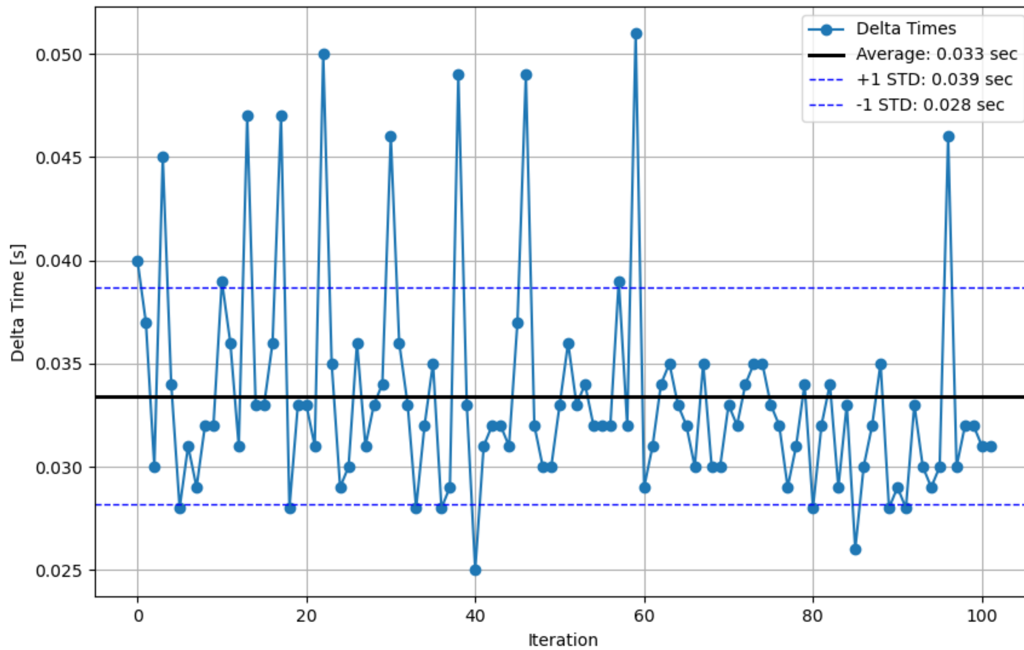


Fig. 48: Time between injection of violation until state was flagged by the copilot monitor. Getting the average value across over one hundred iterations. STD stands for standard deviation.

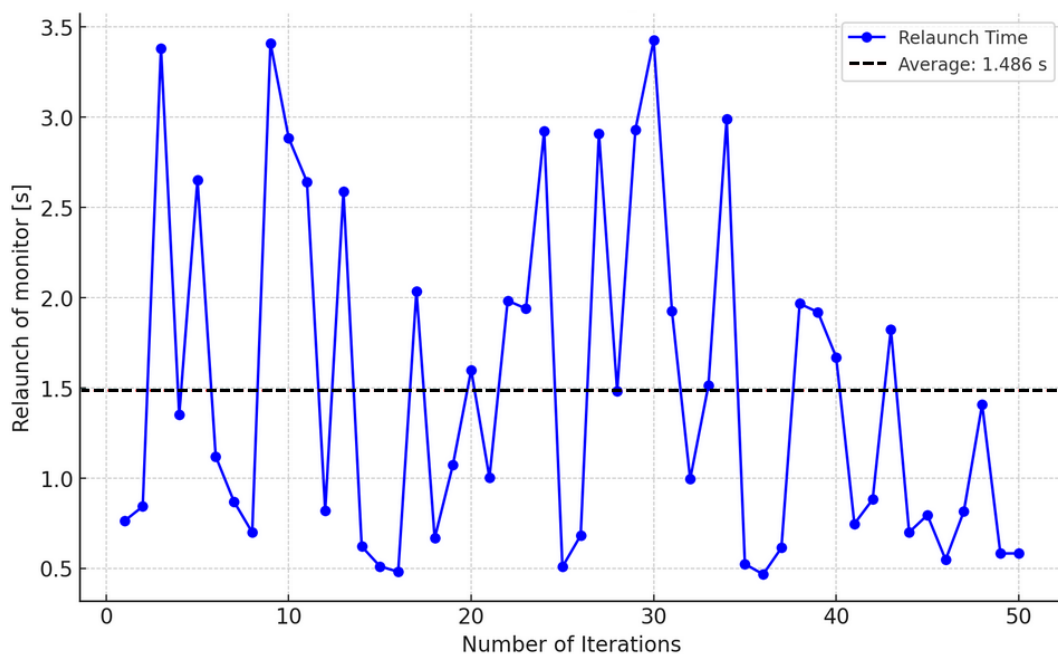


Fig. 49: Time between relaunches of the CopilotRV node in which the monitor system is inactive when receiving state changes. States will still be updated, but in the case of violations, none will be reported. This is to avoid the temporal logic system which will continue to report violations when one has happened in the past. These results were gathered by listening to when the CopilotRV node became active again.

5.2 Simulation

Two methods of data simulation are presented with the aim to showcase the functionality of the monitor system. During the thesis these methods have been crucial during development of the pipeline and without them it would be far more time consuming to develop the requirements and monitoring tool. Sec. 5.2.1 mentions underlying data simulation tools that have been publishing behavior data related to the inputs from external sensors or tools. These have been used to give state information following the logic of the requirements in Tab. 10. Data simulation has also been used to make speed plots which give a good view of the state events during testing. In Sec. 5.2.2, images and data from Gazebo simulation are presented. Here a CAD model of the Thorvald has been made and applied in a simulated environment, which in difference with the real Thorvald is now running on ROS 2. Making it possible to bridge the gap between the communication happening on the robot and the mitigation actions or events the monitor wants to apply.

5.2.1 Simulated Robot Data

Many ROS 2 publisher- and subscriber nodes gathered information and data regarding the performance and robustness of the system during development. The subscriber node of the `srobot_realistic_output` package is still in use. All logic of the system requirements is within this node, and when connected with a RealSense camera or injection device, it will publish state information based on class and distance data. The node is a tool for violation injection, state publishing, and robot simulation. It is additionally a safety controller. Before acquiring the RealSense camera, the `srobot_realistic_output` package also had a publisher node that emulated class and distance data for early testing of monitor capabilities, as well as publishing speed values based on the halt and slowdown topics. In Fig. 50, an example shows how the emulated speed data works, visualizing the different speed modes of the robot, giving a perspective on the robot's behavior under an experiment based on log files saved during each run. In the image, the system starts in halt mode because an adult is in the frame close to the camera; then the adult leaves and a worker is in the frame at a medium distance, triggering the slow-down response. Lastly, the worker leaves the frame, and the speed returns to full capacity. The dip in speed at 230 seconds is because the worker walked towards the camera to get out of frame, triggering the halt response. There is a ramping period between modes of operation where the simulation quickly gains or loses speed.

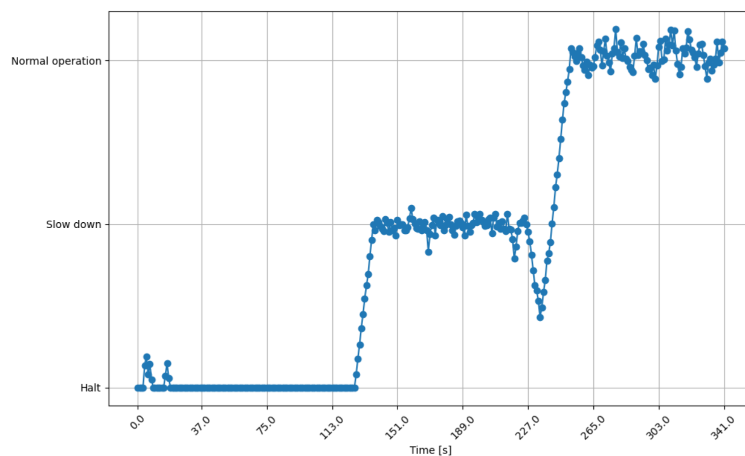


Fig. 50: Simulated speed data under the effect of slow-down and halt messages.

5.2.2 Gazebo Simulation

A simulation, constructed as described in Sec. 5.2, demonstrates the monitor's effectiveness in mitigating safety risks in human-robot environments. The simulation plays a vital role in validating the integration and functionality of the monitoring system. By altering the safety controller so the robot fails to fulfill safety requirements, one can trigger handlers to post violations. The violations can, in turn, activate mitigating actions, serving as a second layer of safety assurance. The monitor can identify incorrect or missing logic in the safety controller by listening to all the topics. It can also take action in critical situations when the safety controller malfunctions.

To test the first statement, one can contradict any mitigation plan in Tab. 7. One could for example remove the logic to halt the robot when a worker is in the red zone, contradicting R_5. The goal would be to trigger a handler violation for

`operationalstate_3` that should trigger a callback in the safety controller, stopping the robot. For the second statement, simply strip the controller of its logic to trigger `operationalstate_3` upon the conformed worker and distance arguments in Tab. 1. The goal would be to trigger a handler violation for `state_req103` in Tab. 7. The monitor's violation handler for `state_req103` was modified to post not only to the violation topic but to the halt topic in hopes of stopping teleoperation.

Fig. 51 showcases a scenario where the safety controller fails to publish a halt message due to flawed logic. The monitor does not read a halt message in the system and warns that `operationalstate_3` was not upheld. The safety controller listens to the warning and appropriately stops the robot, proving that the monitor can identify flaws in the safety controller.

Fig. 51: Compiled screenshots of front-end showcasing the first violation scenario triggering a handler, correcting the safety controller to publish to the halt topic.

In Fig. 52, the safety controller fails to uphold `operationalstate_3` when a worker is less than 3 meters from the robot. The monitor realizes that the state has not changed and warns that `state_req103` is not fulfilled and personally publishes a halt message to stop the robot, proving that the monitor can take mitigating actions of its own in critical situations with little needed modification. This proves useful in a scenario where you want the monitor to be stricter with mitigation than the safety controller due to any concern of an inherent flaw in the system.

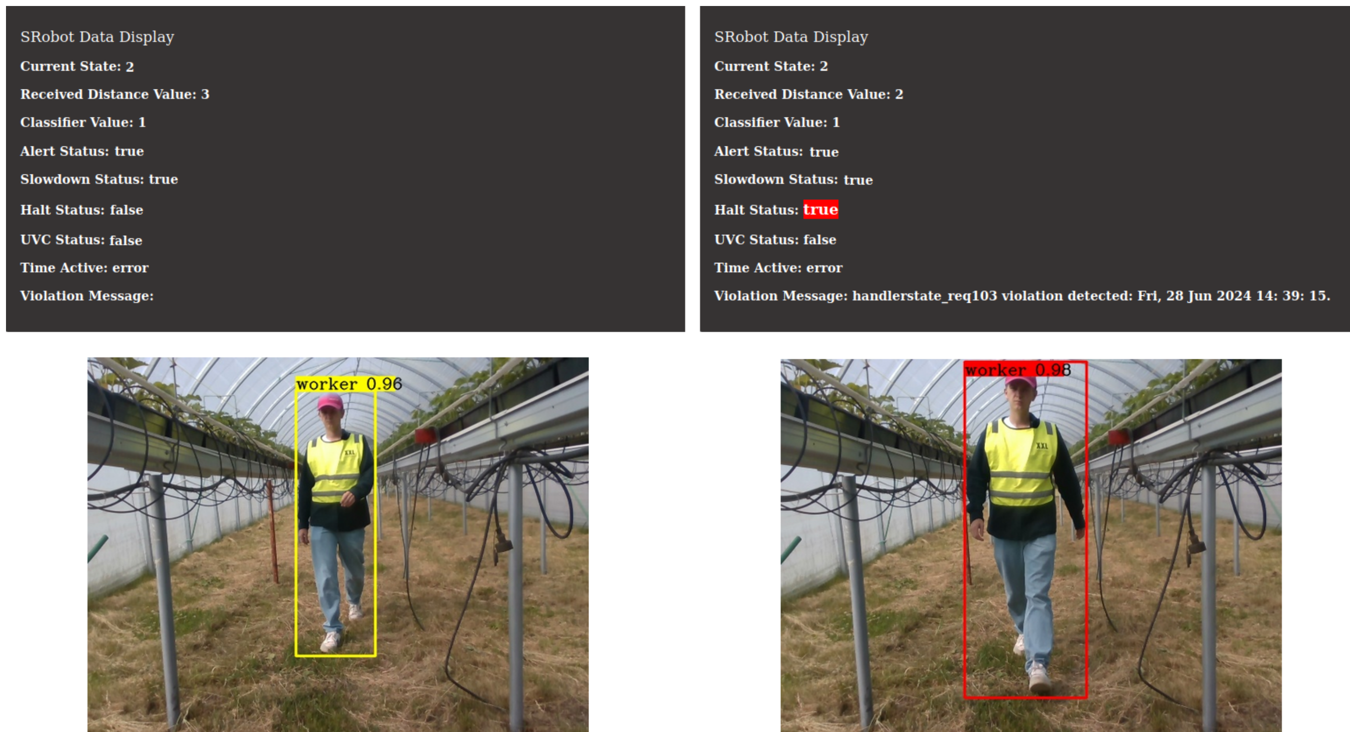


Fig. 52: Compiled screenshots of front-end showcasing the second violation scenario triggering a handler, causing the monitor to publish to the halt topic.

The following collection of videos on GitHub provides dynamic representations of the robot's capabilities and performance: `https://github.com/Andersen0/Thesis_Workspace`

5.3 Field Testing

The results of the field testing are used to evaluate the performance under different operational conditions and the effectiveness in detecting and responding to safety violations. This section contains information and figures from all testing conducted outdoors at campus NMBU Ås and the surrounding areas, including the poly tunnel at the farm field located in Ås. The data collected is divided into two sections. The first section aims to build confidence in the manually configured YOLOv6 package, which was used for subjects classification, with data compiled into confusion matrices. The second section build upon previous data, showcasing the robustness of the monitor system by using it in a field environment with a robot driving alongside it. This includes examples of use cases, mitigation actions, violation behavior, and expected results when using the Copilot monitor system. The data is presented in the form of event images, log outputs, and monitor behavior plots based on speed values from simulation and odometry.

5.3.1 YOLO Field Test

To clarify, the underlying intention of these tests is not to benchmark the YOLOv6 classifier, but rather to prove that the model reaches the accuracy threshold required to test the monitor. An accuracy threshold of 70 percent was set to ensure the model performs reliably in practical applications, defined in the "Probabilistic Modelling and Safety Assurance of an Agriculture Robot Providing Light-Treatment" paper [44]. The YOLOv6 model is trained on manually labeled data, necessitating the need to test the performance of the model to ensure high accuracy during monitor use. To test the model for intended use, field tests were conducted both in sunny areas and in the polytunnel. Larger tests were conducted over ten minutes, while shorter tests examined changes caused by factors like clothing. Data was collected for both the adult and worker classes, compiling the results into confusion matrices. In all tests, the subject moved in a straight line back and forth between two points, ensuring testing conditions remained similar.
Classification results from outdoor tests of the configured YOLOv6 package trained on manually labeled data are displayed in confusion matrices in Fig. 54, 55, 56, and 59.

Fig. 53: Classification of a worker and an adult, differentiated by a safety vest, at noon.

	Worker	Adult
Predicted Worker	36	4
Predicted Adult	0	32

Fig. 54: Initial outdoor tests provided good results. Showing that the model performs especially well in identifying workers, while non-workers (adults) are more challenging. One reason of which might be more variation in the adult images, which could benefit from having more training data.

$$TP = 36, TN = 32, FP = 4, FN = 0$$

$$Accuracy = \frac{36 + 32}{36 + 32 + 4 + 0} = 0.94$$

$$Precision = \frac{36}{36 + 4} = 0.90$$

$$Recall = \frac{36}{36 + 0} = 1.00$$

$$Specificity = \frac{32}{32 + 4} = 0.89$$

$$F1 - Score = 2 * \frac{0.9 * 1}{0.9 + 1} = 0.95$$

	Worker	Adult
Predicted Worker	32	17
Predicted Adult	0	15

Fig. 55: When changing into a white sweater as seen in Fig. 53, the classifier more often than not classified the person as a worker. Further development or upgrade of the YOLO model and training data provided is advised.

$$TP = 32, TN = 15, FP = 17, FN = 0$$

$$Accuracy = \frac{32 + 15}{32 + 15 + 17 + 0} = 0.73$$

$$Precision = \frac{32}{32 + 17} = 0.65$$

$$Recall = \frac{32}{32 + 0} = 1.00$$

$$Specificity = \frac{15}{15 + 17} = 0.47$$

$$F1 - Score = 2 * \frac{0.65 * 1}{0.65 + 1} = 0.79$$

	Worker	Adult
Predicted Worker	200	33
Predicted Adult	0	167

Fig. 56: Number of true and false classifications for each class over the span of ten minutes, test performed outside, taking a sample image every three seconds. Calculation and counting of the results was done manually, and images where the data was obscured were discarded.

$$TP = 200, TN = 167, FP = 33, FN = 0$$

$$Accuracy = \frac{200 + 167}{200 + 167 + 33 + 0} = 0.92$$

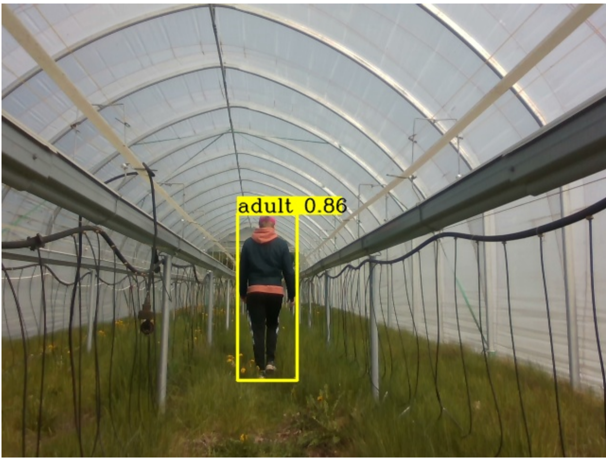
$$Precision = \frac{200}{200 + 33} = 0.86$$

$$Recall = \frac{200}{200 + 0} = 1.00$$

$$Specificity = \frac{167}{167 + 33} = 0.84$$

$$F1 - Score = 2 * \frac{0.86 * 1}{0.86 + 1} = 0.92$$

The first experiments were performed in sunlight at noon, but much of agriculture is done inside poly-tunnels. A similar test, taking classification data of a subject moving back and forth inside the tunnel, was performed for both the worker and adult classes. This time, a simulated speed plot was logged alongside the classification data to observe the behavior of the monitor when seeing a person moving back and forth inside the tunnel. This test was important to observe that the tunnel and objects within did not have an adverse effect on the 3D camera or the YOLOv6 package output. It was observed that elements of the tunnel would occasionally trigger the YOLOv6 classification, but overall, the monitor's behavior remained consistent. The data is visualized in two sets of figures, Fig. 57 and Fig. 58, containing a static image from the data gathering and the corresponding speed plot. The data from the two tests have been compiled into the confusion matrix of Fig. 59.



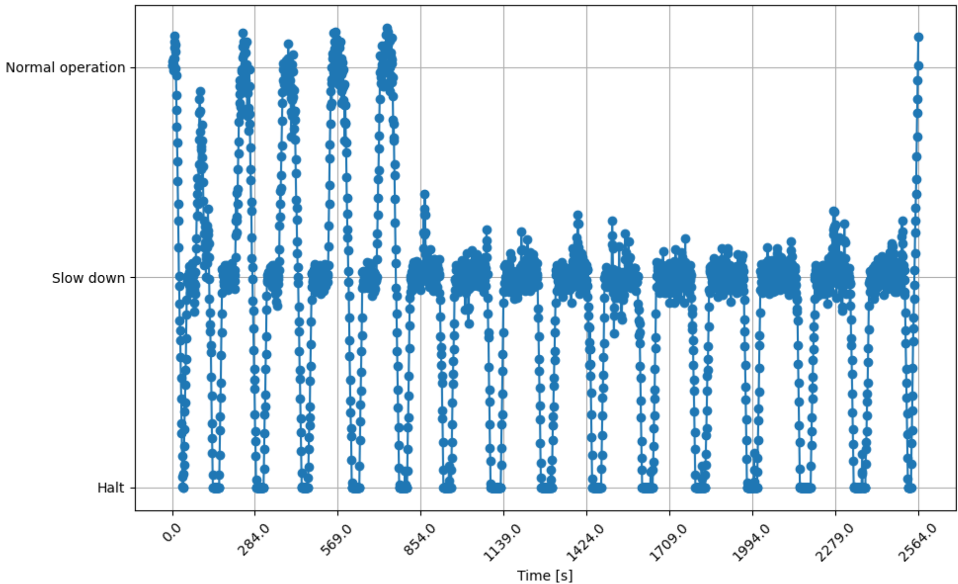
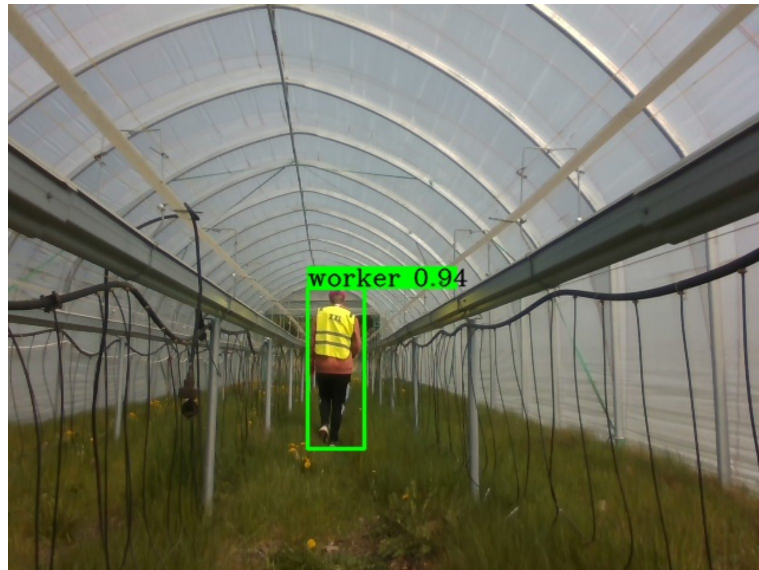


Fig. 57: YOLOv6 class identification experiment performed in the poly-tunnel over the span of ten minutes. The class under testing is the adult class, which effects the monitor behavior by more often going into slow down mode compared with the worker class in Fig. 58. The simulated speed values follow the logic of the requirements in Tab. 1.



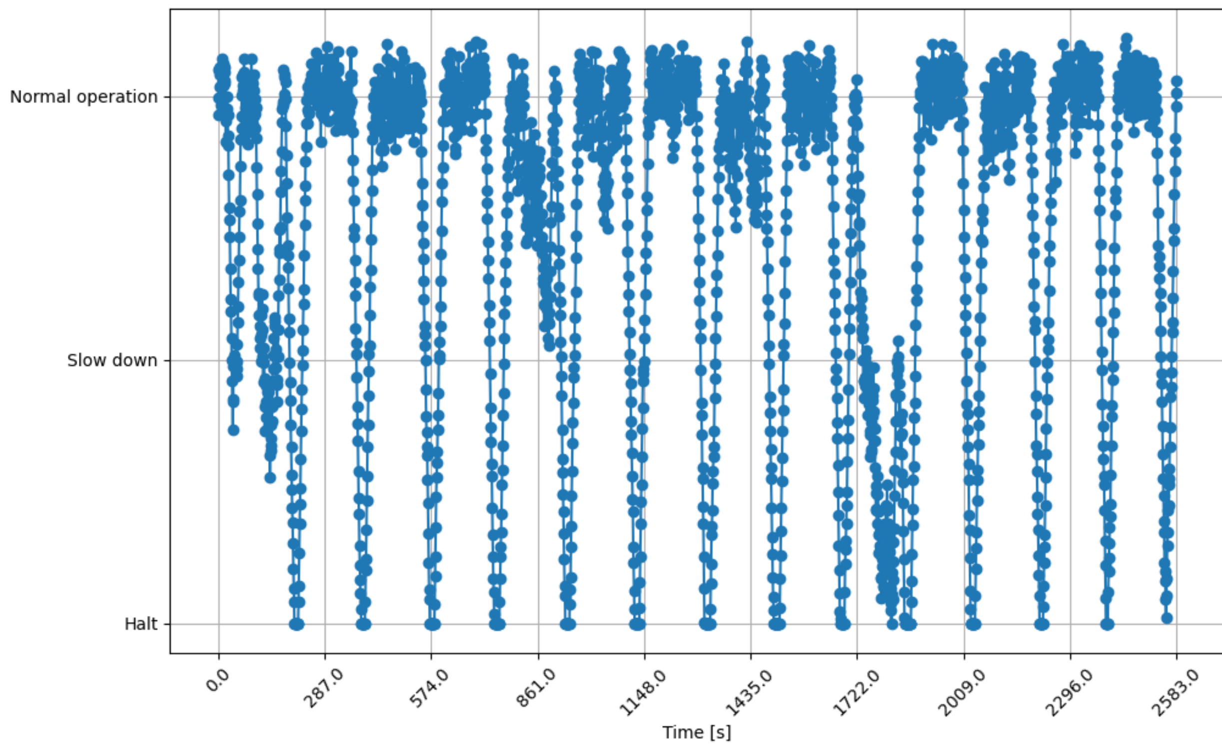


Fig. 58: YOLOv6 class identification experiment performed in the poly tunnel over the span of ten minutes. The class under testing is the worker class, which demands very small distances between the camera and subject before going into halt states. The simulated speed values follow the logic of the requirements in Tab. 1.

	Worker	Adult
True	206	23
False	0	183

Fig. 59: Number of true and false classifications for each class over the span of ten minutes, test performed inside the poly-tunnel, taking a sample image every three seconds. Calculation and counting of the results was done manually, and images where the data was obscured were discarded.

$$TP = 206, TN = 183, FP = 23, FN = 0$$

$$Accuracy = \frac{206 + 183}{206 + 183 + 23 + 0} = 0.94$$

$$Precision = \frac{206}{206 + 23} = 0.90$$

$$Recall = \frac{206}{206 + 0} = 1.00$$

$$Specificity = \frac{183}{183 + 23} = 0.89$$

$$F1 - Score = 2 * \frac{0.90 * 1}{0.90 + 1} = 0.95$$

5.3.2 Monitor System Field Test

This section shows images and data from field testing with the Thorvald-005 robot. These tests were performed in the poly-tunnel at NMBU, Ås. the runtime verification platform ran throughout the use of the Thorvald, collecting Realsense images, violation logs, speed logs, and internal odometry from Thorvald. The purpose of these tests was to visualize the intended use of the RV platform for ROS applications and to demonstrate how mitigation actions can be used alongside the monitors.

Two variations of the field test scenario were used. One for nominal operation where the robot was performing as intended with no changes to the monitor system, checking the robustness of the monitor while state changes and hazards situations were occurring. The second scenario was done in the exact same way, but with a fault injected where there was a one percent chance that when the UV light should turn off, it would not. The intention of this was to test that the monitor would respond and to have an example of a mitigation action being performed. Having a active UV light in the presence of humans can be damaging, so in accordance with this danger, the UV light violation would prompt the monitor to run a bash script. The bash script took control of the Thorvald through a local WiFi connection and shut down all operations, essentially turning the robot off.

The RV monitor was originally meant to run on the Thorvald or be more closely connected to the ROS system of the robot. This became too much of a challenge since the Thorvald robot runs an old version of ROS called Kinetic. In order to communicate efficiently with ROS kinetic, a ROS bridge would have to be used, but most ROS bridge services had no support for ROS Kinetic. A possible solution would be to download an old version of Linux with ROS Kinetic on the computer and use that for bridging the information, but this became too lengthy a process for acquiring examples of how to intertwine the ROS systems for this thesis.

In Fig. 60, there are images captured with a camera from the experimentation area and from the Realsense camera mounted on the Thorvald are shown. The Realsense images have been processed by the YOLOv6 package and our Inferer code to produce classifications with a confidence value. The color of the bounding box around the target is based off the Realsense distance values. These images showcase the environment in which the tests were performed and how the class and distance values were collected, which define what state the robot is in and what requirements are active.

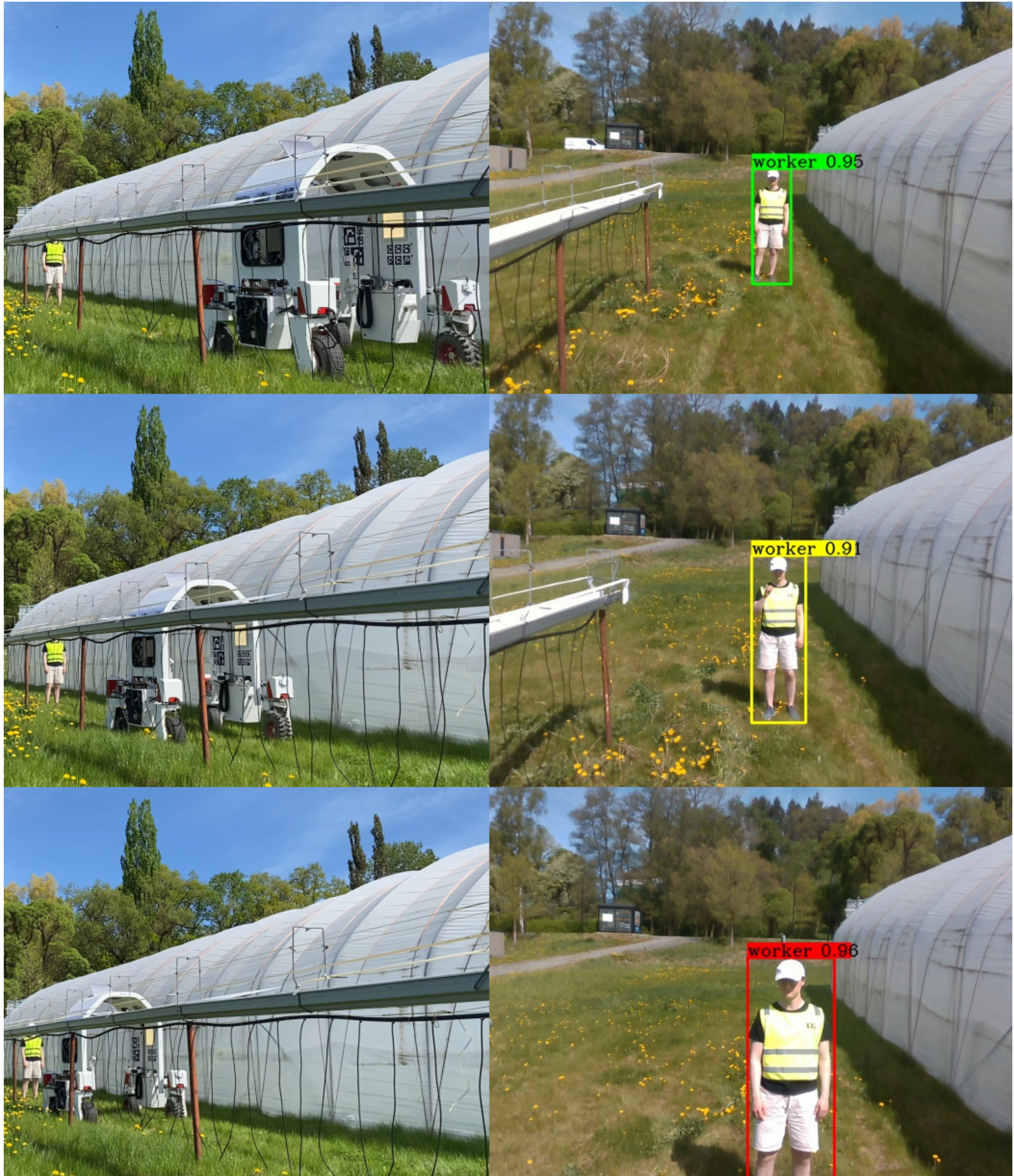


Fig. 60: Experimentation with the Runtime Verification monitors next to the poly-tunnels at NMBU. The Inferer function uses the data from the Realsense and YOLOv6 package to collect class and distance data, which is further used to determine the robot's current state. Further data collected from this can be observed in Fig. 61 and Fig. 62.

During the testing, the robot and monitor were connected over a local WiFi hot-spot generated by a mobile phone. This setup facilitated a stable internet connection and hosted the Flask website on a reachable IP. By hosting the website locally, all current state information became accessible to any unit connected to the WiFi. This was used to control the behavior of the robot and to observe faults or violations that occurred. Fig. 61 shows two screenshots of the website taken from a mobile phone connected to the IP. On the website, the information related to the state information and the latest violation is displayed. When the robot is in slowdown or halt mode, the tags for those states blink in yellow for slowdown and red for halt. The last image containing a class of type adult or worker is displayed at the bottom of the page, this could also be altered to show the latest image without the consideration of class.

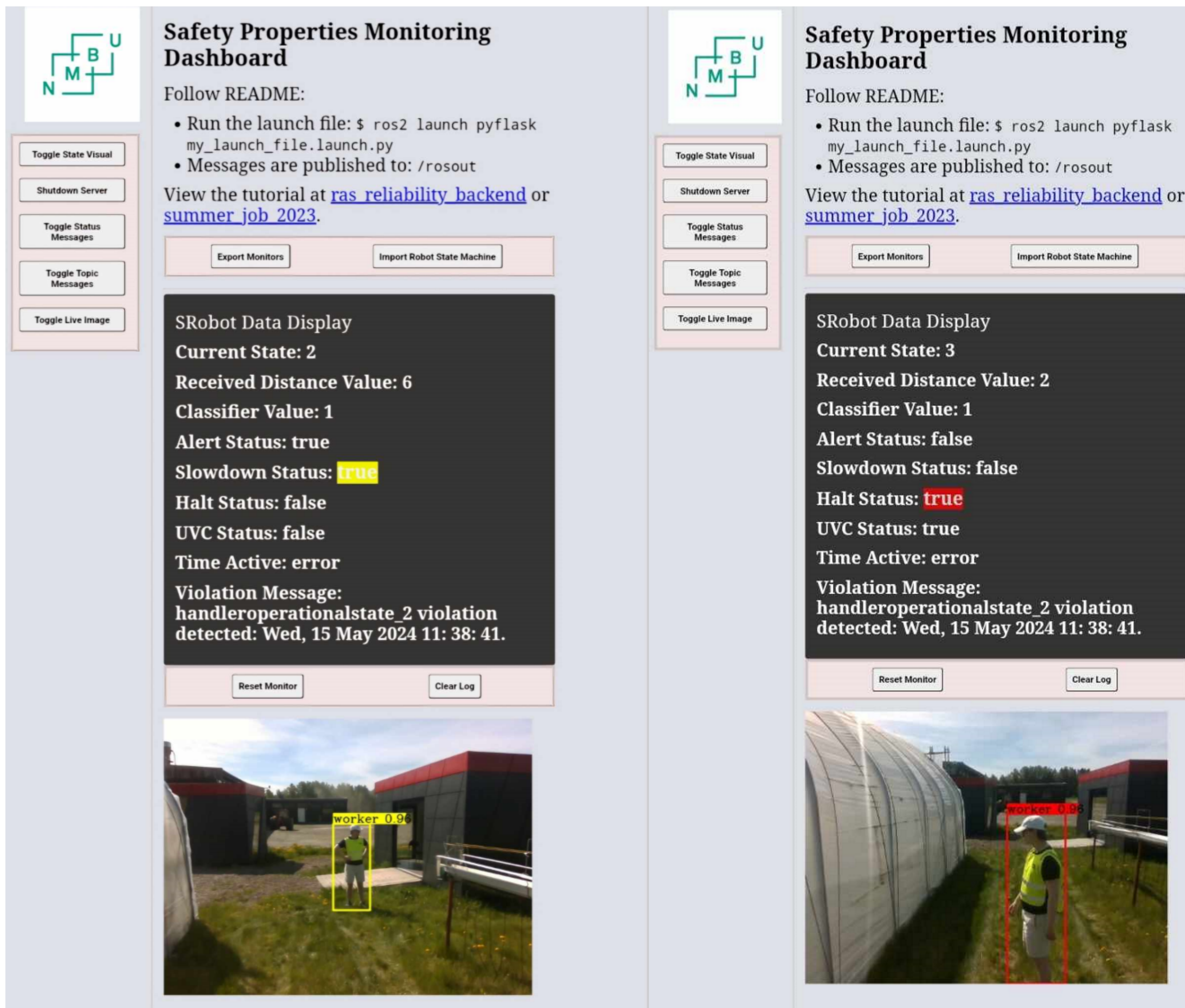


Fig. 61: Two images taken not far apart showing current state information and visualization of the website from a mobile phone connected to the web page IP. Demonstrating the Flask tool used to monitor the state information and the behavior of the monitor/robot.

The simulated speed data was also logged for all field tests, providing a plot showing the intended behavior of the robot given the safety requirements. These are categorized into three speed modes; normal operation, slowdown, and halt. The plot in Fig. 62 shows the expected behavior of the entirety of the field test, as seen in Fig. 61, and 60. The nature of safety requirements can become chaotic when people work in close proximity, as seen in the field test, underscoring the importance of proper safety systems.

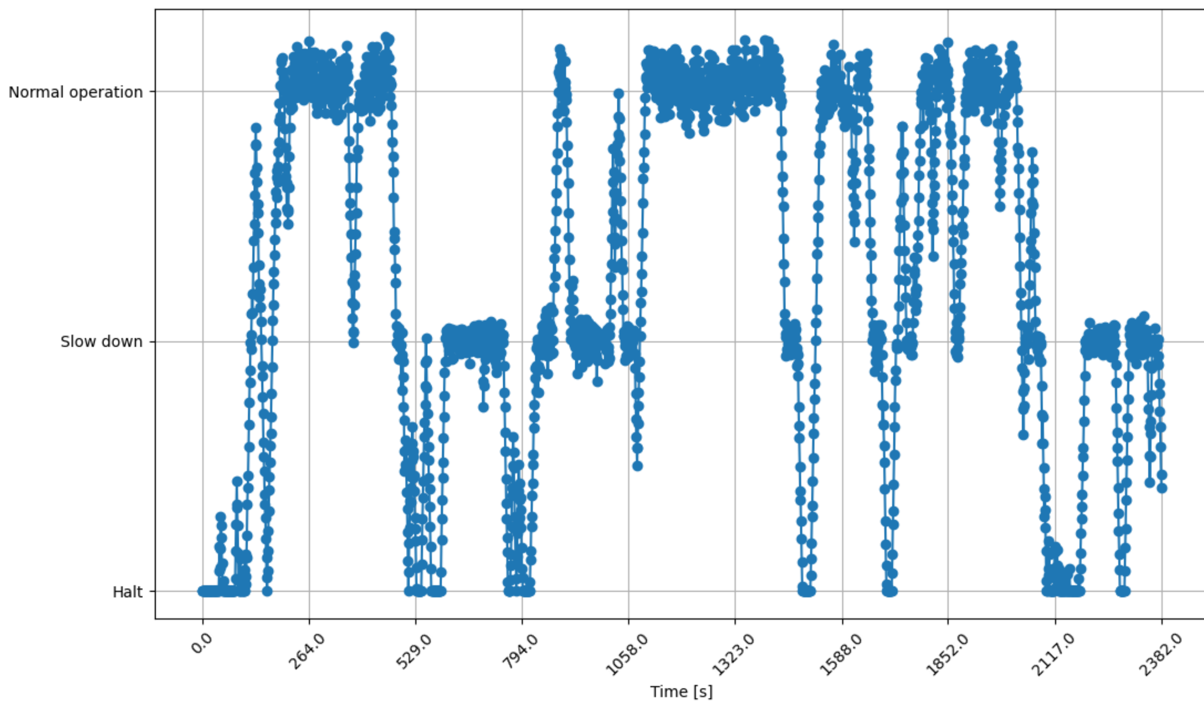
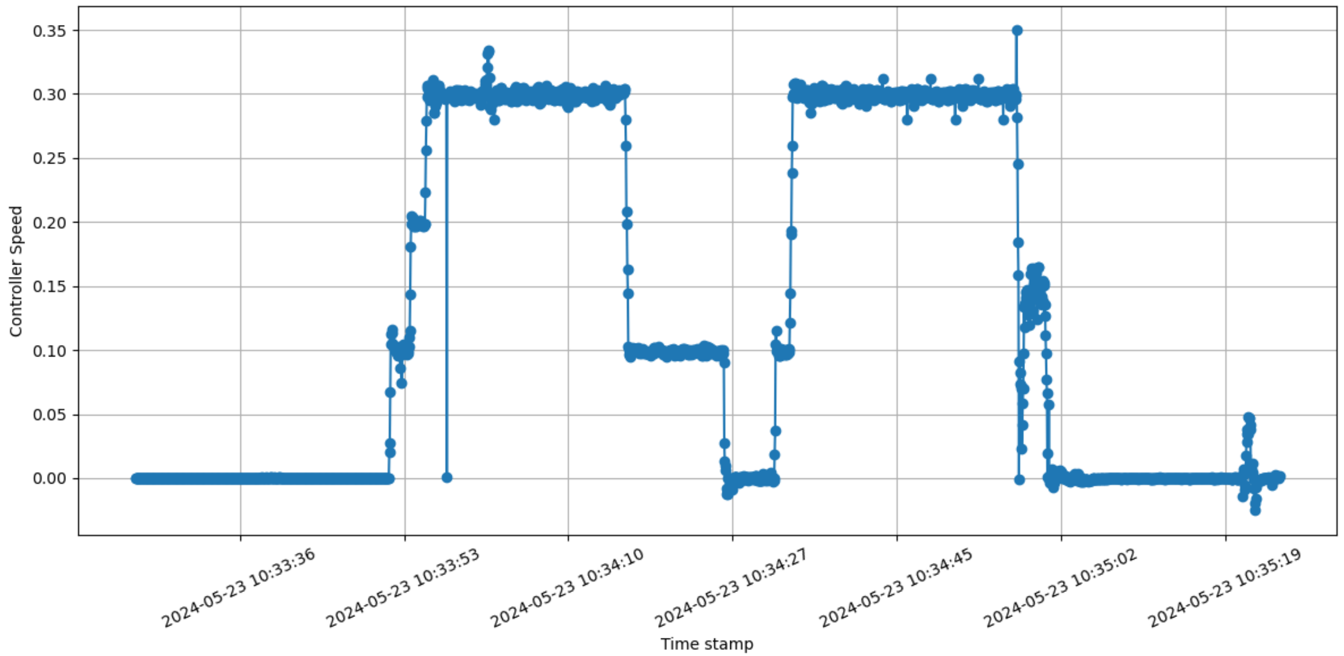


Fig. 62: Simulated speed values from field test. The test was performed with a person of class worker moving around the mission area. Since the subject was correctly labeled as a worker, the robot rarely needed to halt operation but often slowed down.

A smaller test was performed to compare the output of the monitor with simulated speed data, checking that the monitor followed expected behavior. Fig. 63 contains two plots: the first is gathered from linear speed odometry data on the Thorvald robot, and the second is the simulated speed. The Thorvald was driven manually using the web page monitor as a guide, and later, the correlation between the monitor data and the simulation was checked. The odometry data was only from linear speeds, so sections involving turning gave zero speed linearly. This is why the beginning and end of the plot differ between the two figures. Additionally, the ramping time on the simulated speed is longer than that of the Thorvald robot.



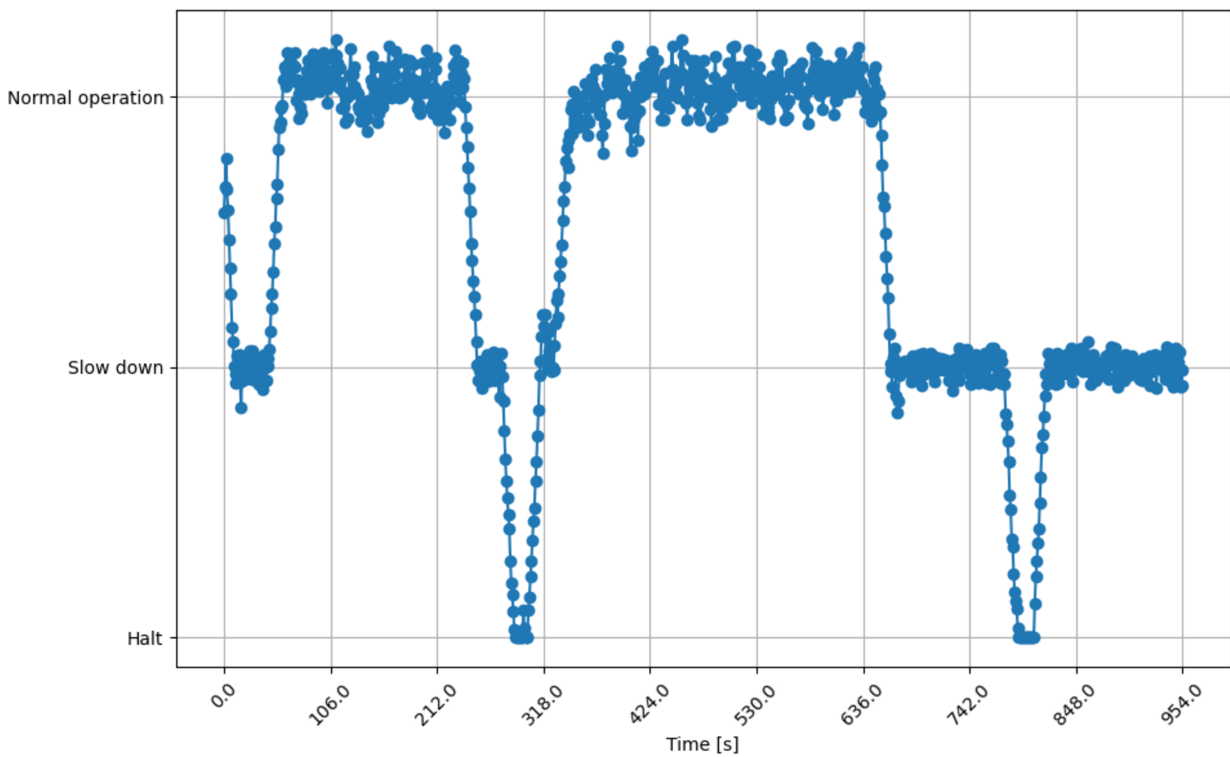


Fig. 63: The first plot shows the linear odometry values from the Thorvald during a test of driving parallel to a strawberry field towards a worker. The robot slowly moves towards the stationary person until a halt is achieved, then the person moves out of the way and the robot returns to normal operation. The second plot mirrors this with simulated speed data of the encounter. Only linear speed was gathered from the odometry, meaning that when turning, the speed was measured as zero. Other than that, the figures correlate well. The Thorvald was driven manually based on the monitor safety data.

In a low complexity environment, it is not expected that the monitor system will encounter violations from normal use. Therefore, to test the violation reporting and mitigation action part of the system, an injection device/system was used. For general testing of all the requirements, injection buttons were created with a violation event designed to trigger a specific requirement. This method worked for all requirements in Tab. 10, and the buttons can be seen in Fig. 37. A second way of triggering the violations was implemented with a field test in mind. In this scenario, a one percent chance was programmed so that when the UV light would normally turn off, it would instead turn on. This caused a violation in the `handleroperationalstate_3` requirement. Additionally, to prove that mitigation actions could be implemented from the monitor output, a bash script was created to SSH onto the ROS platform and shut down the robot in case of a UV light violation. A bash script was used since no ROS bridge was implemented for the Kinetic software on the robot platform due to incompatibility issues and limited resources. Fig. 64 shows the violations that occurred during the field test with the one percent chance that the UV light would not turn off. The monitor receives new data frequently, so even with a one percent chance, it still triggers quite often. The monitor and the robot are two separate systems, which is why shutting down the robot does not close down the monitor system or stop it from gathering data from its surroundings.

Violation Log Messages

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 34: 13.

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 33: 10.

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 33: 08.

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 33: 04.

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 32: 40.

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 32: 39.

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 31: 47.

handleroperationalstate_3 violation detected: Wed, 15 May 2024 13: 31: 46.

Fig. 64: Violations that occurred because of the UV light not turning off when it should. This was achieved by adding a one percentage chance that the violation would happen when state three was reached. All violations are logged with a timestamp, both on the website and in a separate log file. The newest violation is always displayed on top.

The image in Fig. 65 shows a typical case where the UV violation was triggered. The adult class is more susceptible to this violation since state three is active more often than with the worker class. While driving, the violations and system information were tracked with a mobile phone, also seen in the image. The image was taken at the moment the induced UV light violation occurred. This happened multiple times during the course of this experiment, all of which can be seen in the log in Fig. 62. The violation happens frequently since the UV state is updated at the same speed as the incoming data from the RealSense. This frequency depends on the performance of the computer it is attached to, which is often limited by the YOLOv6 and Inferer code since they require significant computing resources for each image.

Fig. 65: Image from timestamp 13:33:08, which is the same timestamp a violation occurred because of the UV light not turning off.

When testing with the UV violation, both the mitigation action and the violation reporting worked as intended. The monitor was capable of running a shutdown command on the Thorvald robot when the UV light did not turn off, and the monitor reported and logged everything as it happened. The simulated speed plot in Fig. 66 shows how the test was performed from beginning to end. First, the robot was allowed to drive normally to ensure no violations or bugs were reported. Then, a person of the adult class entered frame, triggering state three. This resulted in the violations seen in Fig. 64. After being in halt mode, the adult walked slowly away from the robot, and since the monitor continued collecting data, the plot continued to show transitions to slow down and normal operation mode, even though the robot was shut down due to the violation.

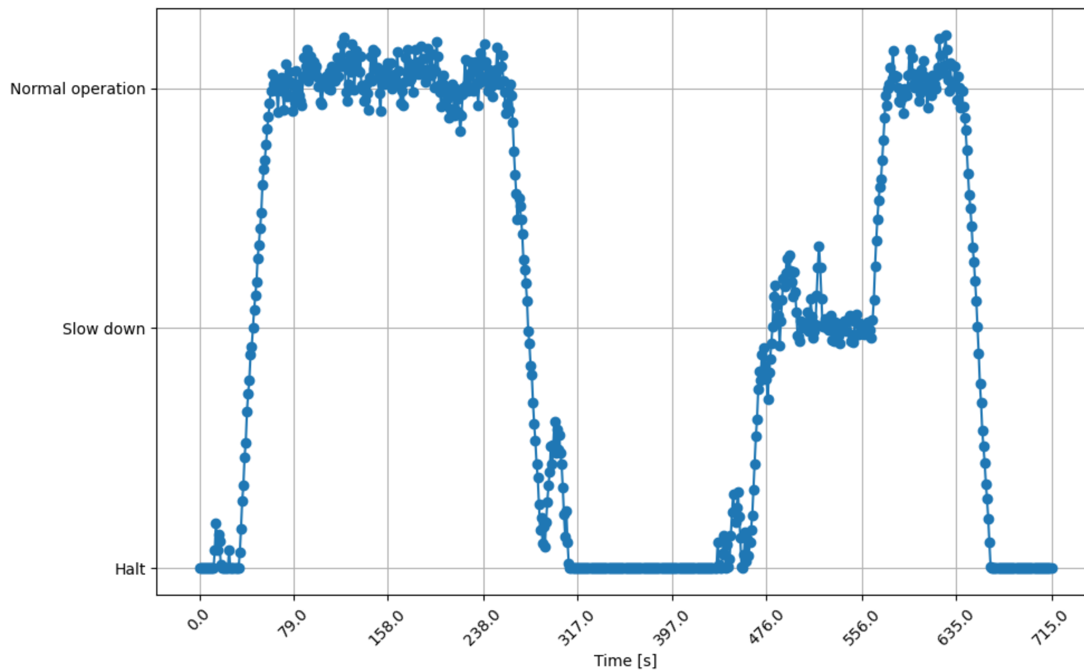


Fig. 66: Simulated speed data from field test with UV violation being reported.

5.4 RV Software Tool Comparison

The intention behind summarizing the software used in the RV tools ROSMonitoring (Sec. 2.3.1), TeSSLa-ROS-Bridge (Sec. 2.3.2), and ROSRV (Sec. 2.3.3) is to provide an overview of existing software alternatives to Copilot and the features offered by different methods. This comparison aims to encourage in making an informed decision regarding which RV software to consider for a given problem. Gaining insights into the capabilities of other models can also benefit the ongoing development of the Copilot-OGMA solution for ROS. During the preparation of this paper, issues related to the Copilot-OGMA solution for ROS have been flagged and addressed. Although the ROS support is currently preliminary, it is hoped that this paper will inspire continued use and support for CopilotRV with ROS. In Tab. 12 all methods mentioned in Sec. 2.3 are compiled along with Copilot-OGMA. Highlighting some of the more noteworthy features they offer and at least one drawback. In Tab. 12, Intcp indicates whether the RV method can intercept and modify messages to prevent violations.

Table 12: Comparison of Runtime Verification Methods

Software tools	Monitor Logic	Features	ROS 2 support	Intcp*	Hard Real-Time	Pros and Cons
Copilot-OGMA	LTL, ptLTL	Haskell EDSL, generates C code	Preliminary ✓	✗	✓	+ Suitable for real-time + Formal verification - Requires Haskell knowledge when used without OGMA
ROS-Monitoring	Flexible (formalism agnostic)	Integrated ROS solution	TBR⊛ ✓	✓	✗	+ High portability + Real-time message interception - Invasive, can add overhead
TeSSLa-ROS-Bridge	temporal stream-based language	TeSSLa language within ROS node, Python bridge	✓	✗	✗	+ Asynchronous verification + Complex timing constraints - Lacking data regarding performance under heavy load
ROSRV	Custom (based on safety/security policies)	Integrated ROS solution with custom master node & monitor nodes	✗	✓	✗	+ Seamless ROS monitoring + Customizable - Security relies on network/IP - Outdated, no longer supported

*Intcp = Ability to intercept and modify messages. ⊛TBR = To Be Released.

6 Discussion

In this section, we provide a detailed analysis of our findings from the implementation of automated generation of monitors from Natural Language (NL) to formally verified specification models in a ROS environment. This analysis includes a discussion of results (Sec. 6.1), a comparison with previous research and RV methods (Sec. 6.2), and an exploration of the broader implications of our findings (Sec. 6.3). Our goal is to assess the strengths and limitations of the RV framework, highlight its practical implications, and discuss its potential to enhance the safety and efficiency of autonomous robotic systems. This section addresses the research questions and objectives outlined in earlier chapters, providing a critical evaluation of the framework's effectiveness and identifying areas for future improvement which are later reiterated on in the future works section (Sec. 8).

6.1 Results Discussion

The results discussion section delves into the analysis and interpretation of the findings obtained from the implementation and testing of the OGMA compiled Copilot Runtime Verification (RV) framework within a ROS environment. This section aims to provide a comprehensive evaluation of the monitor behavior, simulation outcomes, and field testing results. By examining these findings, we aim to understand the strengths, limitations, and practical implications of the RV framework, as well as its potential for enhancing the safety and efficiency of autonomous robotic systems. The following sections will detail the specific findings related to monitor behavior (Sec. 6.1.1), simulation scenarios (Sec. 6.1.2), and field testing (Sec. 6.1.3). Through this discussion, we will address the research questions and objectives outlined in earlier chapters, providing a critical assessment of the framework's effectiveness and areas for future improvement.

6.1.1 Monitor Behavior Findings

The most significant results for discussion pertain to the monitor behavior during testing. These findings are crucial in determining the functionality of the OGMA-Copilot solution and the usability of the pipeline, which transitions from natural language expressions of a model specification in FRET, and finally to Copilot using OGMA. The empirical data shown in Sec. 5.1 is summarized in Sec. 6.1.1. Each topic being discussed in the following paragraphs to evaluate the findings.

The implementation of the Copilot Runtime Verification framework in the ROS environment showed a significant improvement in detecting and responding to safety violations. The monitor was tested for system overhead (Sec. 5.1.1), limitations of stream to event-based language caused by asynchronous publishing (Sec. 5.1.2), and the responsiveness of the system when detecting violations (Sec. 5.1.3).

System Overhead and Performance: The overhead introduced by the RV monitors was consistently within acceptable limits, even under high message publishing rates. The study quantified the system overhead introduced by the monitors at different publishing speeds and found that up to a certain threshold (12.5kHz or 1.785HZ per topic), the system maintained stable performance. Beyond 12.5kHz, the overhead started to impact the system's responsiveness, increasing the bottleneck delay caused by the monitor not being able to respond to all messages in time. This resulted in a plateau of the average rate of messages being delivered per second to the monitor as shown in Fig. 39. The publishing limit on a single ROS topic was also met during testing, but higher Hz values were reached by distributing the message load onto all seven topics used in the system model, as seen in Tab. 5. The current implementation of the monitor code being able to process 12.5k state messages per second was seen as a significant achievement, although the monitor should not be exposed to such loads. Due to the changes made to the step function, messages were processed in groups, reducing unnecessary calculation. Therefore, while the monitor tracked all 12.5k state messages, it did not need to run the step function for each single message, instead checking the requirements when all states where updated. Around 1,800 iterations of the step function per second seemed to be the limit, while the ROS publishing plateau on a single topic was 3,186 messages per second, indicating that ROS is not the current limiting factor. The specification model used in this example was also a simpler scenario to model. Higher complexity models are more likely to increase the time spent on requirement computation. Still, the pipeline system performs well and will be able to be used in most modern software or cyber-physical systems (CPS) without issue. A possible solution for higher complexity models is to split the requirements into groups that can be tested separately, lessening the load on the system as a whole. This approach has been proposed by other RV methods as a way to increase performance for larger systems. FRET allows for designating requirements to groups following a parent name, which could be used for creating a workflow towards monitor generation for multiple system areas.

Event- vs. Stream-Based Logic: The study revealed the asynchronous nature of ROS data publishing and its impact on the Copilot monitor, which assumes synchronous stream-based Linear Temporal Logic (LTL). The difference in logic caused multiple violations to arise, necessitating a solution to fix the issue. As shown in Fig. 44, even though the topics are set to publish at the same time, there is a small delay between each topic. This delay, while minor, causes violations because the monitor expects all topics to be updated simultaneously. In Fig. 67, the delayed topics are visualized, with a step function call following each. This setup can potentially cause violations during state transitions since requirements are checked before all topics are updated.

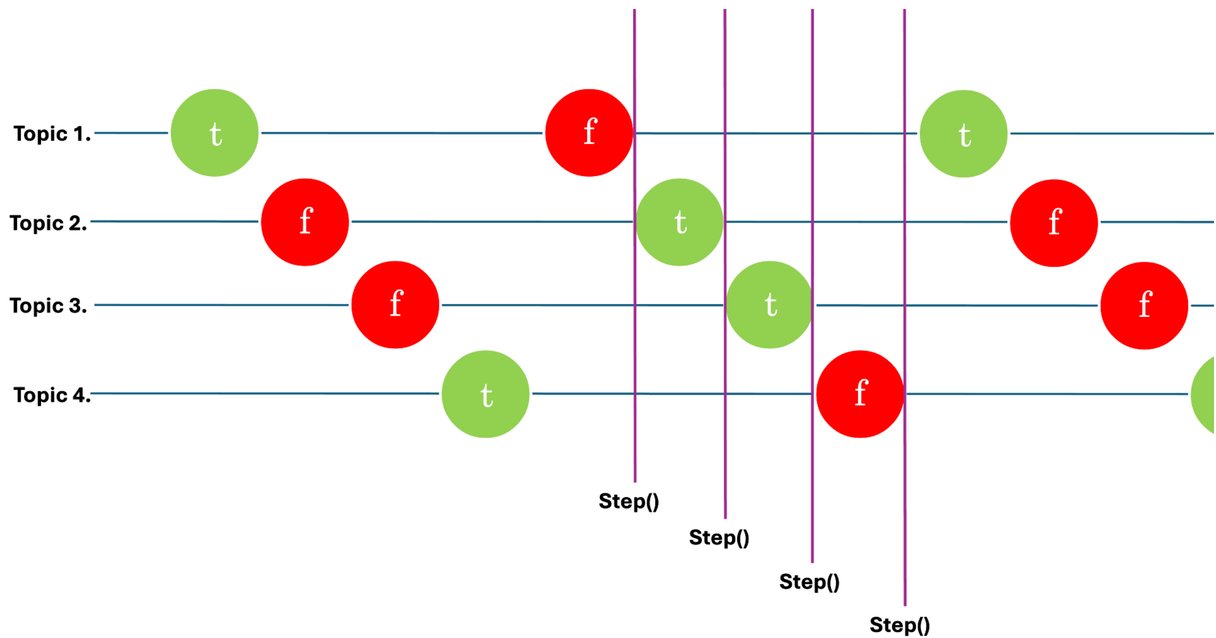


Fig. 67: The timing of the step function, after compiling the Copilot monitor without configuration, when used on asynchronous data publishing.

Two methods were tested to solve this issue. First, the step function was called with a timer function that, if properly implemented, could ensure the function always resolves between state changes. The limiting factor was that the delay between each function call is a period of time when violations could occur without instant flagging, delaying the response. The second solution, which was adopted, involved running the step function after the last topic was updated, ensuring all previous topics are refreshed. An example of this is shown in Fig. 68, where the step function is called after the last topic (topic 4) is updated on the transition state. The limitation of this solution is that the order of topic publishing must always be followed, and the time from the first topic to the last must be kept as short as possible. From Fig. 44 and Fig. 45, one can see that the delay caused by waiting for all seven topics is small, but in Fig. 46, the delay can increase as the performance strain increases. A major increase in delay was only observed in the event of a system crash, as seen in Fig. 46, otherwise, the results remained consistent. A problem encountered when switching to faster hardware was that state changes could occur before the step function was able to run, as publishing happened before the monitor received the previous topic. This caused violations during state changes, but a simple solution was to incorporate a minor delay between each publish of the seven topics, allowing the step function to run without simultaneous refreshes. When the system was functioning with the hardware, few faults were observed without intentional injection.

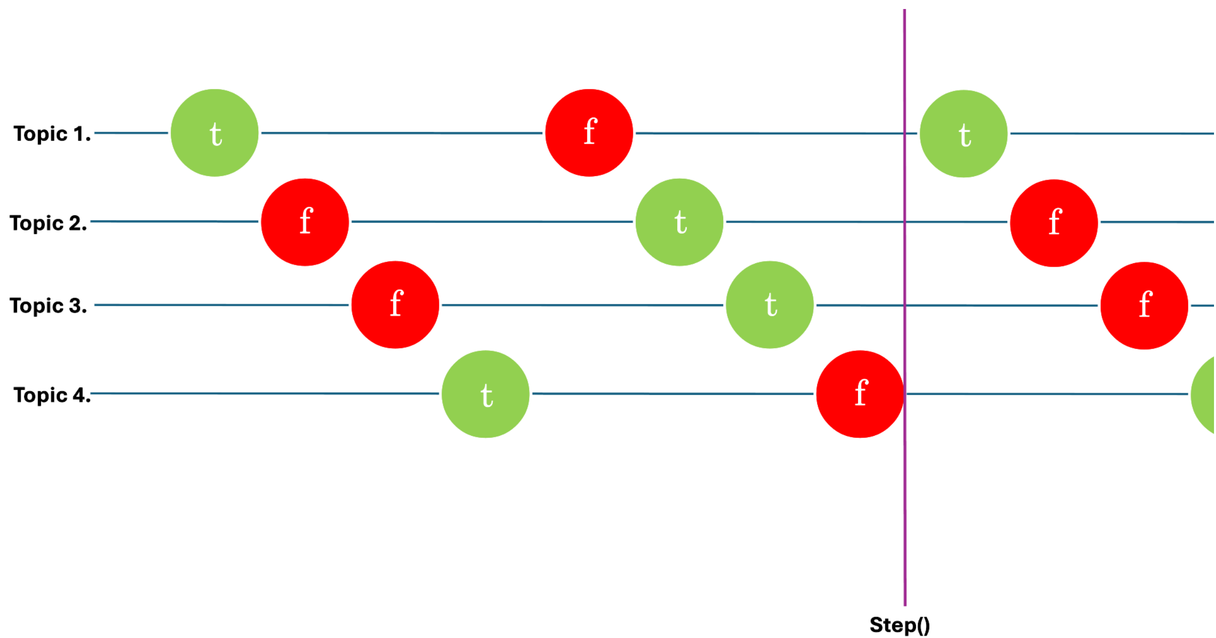


Fig. 68: The timing of the step function post-Copilot monitor configuration, when used on asynchronous data publishing.

Violation Reporting and Behavior: The framework's ability to report violations accurately and promptly was tested extensively. The average time to detect and report a violation was within the expected range, demonstrating the framework's efficiency. In early implementation phases, the time from a violation state being published to the violation message being received averaged below 20ms. After additional backend features were added, the average time increased to 33ms, as shown in Fig. 48. For most safety-critical systems, a time of 33ms is efficient for detecting a violation and initiating necessary mitigation actions. By changing the nature of the monitor to add safety controller features, one could intercept and change states within this time frame to prevent violations, further enhancing safety. An example of this is discussed in the simulation findings (Sec. 6.1.2). For testing purposes, some changes were made to limit the LTL nature of the monitor. This was because ptLTL logic makes past violations control the logging of the present, causing violation handler functions to continue posting. This overflow of violation logs complicates debugging and controlled use of the monitor. To solve this, the CopilotRV node controlling the monitor code was relaunched each time a violation was flagged using a bash script. This caused a major delay in monitoring where no violations could be logged but also limited the ptLTL logic to the point where only one or two violations were flagged each time a non-nominal state was received. The delay averaged at 1.5 seconds as seen in Fig. 49. This is not recommended for any use other than testing. Additional solutions could involve editing the Copilot monitor code or maintaining the ptLTL logic since violations are rare in normal operation. Dividing requirements into groups based on hazard severity could also help define different monitor responses. High-severity groups could keep ptLTL logic, while less critical issues reset themselves if the problem does not persist.

6.1.2 Simulation Findings

Simulation proved to be an invaluable tool throughout the development and testing of the system. Without simulation, developing the requirements and monitoring tools would have been far more time-consuming. The simulation tests, which included various scenarios such as speed variations and object detection, confirmed the reliability of the RV framework in maintaining system integrity and performance under different conditions. Simulated robot data, such as speed and positional information, was accurately monitored and reported by the RV system, ensuring that the robot's actions were consistently aligned with the safety requirements. The Cyber-Physical System (CPS) Thorvald-005, addressed in this study, originally runs on an End-Of-Life (EOL) ROS Kinetic distribution. The Gazebo simulations provided a robust platform compatible with the ROS 2 system to test the RV framework under controlled conditions. The results showed that the RV monitor could handle complex scenarios involving multiple moving objects and dynamic environmental changes. By simulating conflicting mitigation plans and deliberately breaking the safety controller, Sec. 5.2.2 successfully demonstrated that the monitor can identify flaws in the safety controller and act as a second-layer safety controller in critical situations.

The simulation also revealed a significant issue with the depth output of the RealSense camera during testing. Due to the method of gathering depth data from the center point of the closest bounding box, any obstruction that would pass along the middle of the bounding box would shorten the reading, making the classified person appear closer to the robot in simulation than one could perceive in the actual image. With many objects in the frame, this could potentially lead to extremely erratic measurements. To address this, a Kalman filter was applied to smooth the depth data. Kalman filtering, a common algorithm used to minimize mean squared error, tracks the estimated state of the system and the variance over time [54]. In deployment, the use of an advanced camera grid could also improve the data consistency. Another issue was the camera's decreasing accuracy at greater distances, manifesting as increasing noise beyond 3 meters.

6.1.3 Field Testing Findings

YOLOv6 Performance: The integration of the YOLO object detection system with the RV framework proved effective in identifying and classifying objects in the environment. This was crucial for ensuring the safety of the robotic operations, particularly in identifying human workers, adults, and the distance to the subject. The accuracy of the classification model was crucial in collecting stable and correct data for testing of the monitor system.

The YOLOv6 model was trained with manually labeled data of workers and adults in different environments, raising some concerns about the model's performance. To evaluate the system, a 70% accuracy threshold (defined in CASE2023 paper [44]) was set as the lower limit, and multiple tests were conducted in both indoor and outdoor areas. These tests were carefully planned and executed in a controlled manner, limiting variables that could affect the results. The evaluation concluded that the model achieved accuracies above the lower threshold in all daylight tests, with the highest performance observed in the data collected from the poly-tunnel Fig. 59.

However, the model was not without faults. It performed best on the worker class, never misclassifying the subject as an adult, but the adult class was more challenging. This is likely due to a larger variation in the training data and a limited data pool. This sometimes caused the adult class to be handled as a worker, especially when the subject wore lighter clothes resembling a safety vest. Overall the reliance on the safety vest for worker classification became a problem when objects in frame obscured the vest.

To improve the robustness of the adult and worker classifications, a memory function could be implemented to track subjects in the frame, ensuring class changes only occur when the probability of a certain class is sufficiently high. The confidence threshold for the YOLO model changed throughout the project, ultimately settling at 40% to prevent subjects in the frame from becoming unclassified when the confidence value was too low. The low confidence value occasionally caused misclassification of objects, though this was rare and typically occurred with bright yellow or tall objects. This issue could be mitigated with additional training data for the YOLO model. The speed graph in Fig. 57 shows the stability of the classification, clearly following the pattern of back-and-forth walking without misclassifying objects in the meantime.

Monitor System Performance: The RV system's performance in field tests was consistent with the simulation results. The system was able to maintain its monitoring capabilities and ensure compliance with safety protocols even in outdoor environments with varying conditions. The monitor was also used to activate a mitigation action for an induced violation caused by the UV light not turning off. In this test, the monitor took control over the Thorvald-005 robot through SSH and forced a shutdown of the hardware, ensuring the safety of the subject. The Flask app during testing displayed its capabilities of reliably delivering real-time data concerning the robot's state and violations that occurred, making it possible to receive data on a WiFi-connected phone or system.

Hardware: In real-world field tests, the RV framework successfully identified and mitigated safety hazards, although sometimes the results were limited by the accessible hardware. For instance, the Thorvald-005 was running an old version of ROS Kinetic, which made using Rosbridge for ROS communication between the monitor and robot difficult to achieve and too time-consuming for the thesis. The limiting hardware for the performance of the monitor was the RealSense camera and laptop used in the experiments. The RealSense camera worked optimally at distances of 0-3 meters, which was not ideal for robotic field work. While the classification performed by the YOLO model did not suffer from the camera hardware, the distance values were inaccurate for larger distances. The frame rate from the camera was impacted by the connected system, which in this instance was an AMD Ryzen 3 laptop that struggled with performance under high load. Overall, the received data was stable enough and reliable for testing the monitor capabilities.

6.1.4 Summary of Key Findings

This section provides a comprehensive overview of the most important results from our research, emphasizing how these findings address the research question, meet system requirements, and contribute to the existing body of knowledge.

The problem statement of this project was "To enhance workplace safety and reliability throughout the engineering life-cycle by applying generative runtime verification (RV) to the agricultural domain for ROS 2 applications, turning safety requirements defined in natural language (NL) into formally verified temporal logic monitors." Sec. 1.1. This aim was driven by the intention to enhance workplace safety, serve as an example of the implementation of automatically generated monitors, and contribute to the ROS 2 RV space. The research questions were: "Is it possible to create a pipeline for the automatic generation of runtime verification monitors for ROS 2 systems using current software tools?" and "Does the runtime verification monitor help elevate safety during all phases of the engineering life-cycle?", Sec. 1.2.

In summary, all goals were achieved, proving that the automatic generation of formally verified RV monitors for ROS 2 is possible with current software and quick to implement for existing systems during active development and for systems already in deployment. This approach further bolsters the safety and reliability of the system, with the potential to implement mitigation tactics and safety controller behavior. By using the NASA created tools FRET, OGMA, and Copilot we were able to fulfill all system requirements of the research, as seen in Sec. 3.2.

Problems encountered during development included the lack of time to properly connect the ROS Kinetic system with the ROS 2 RV platform with the use of a bridging tool, which made implementing mitigation tactics for the unsupported operating system of the Thorvald-005 robot difficult but achievable through SSH connection. Another issue was that the automatically generated monitors from Copilot expected stream-based communication, while ROS operates on an event basis, leading to asynchronous data publishing and violations. This was solved by a small change in the Copilot monitor code.

In retrospect, further research should have been conducted on the buildup of requirements and the limits of FRET as a generation tool for ptLTL logic. The specification model used in the project (Tab. 10) demanded immediate change of system states to validate nominally. Due to asynchronous nature of publishing, this was not possible, necessitating a change to the step function. By altering the requirements, it might be possible to use a time tick validation to approve requirements within a time frame. This would enable the safety monitor to handle event-based topics with a stream-based approach. However, OGMA uses the ptLTL version of the formula, where time is considered discrete, making a full implementation of continuous-time metric temporal logic non-trivial. A possible change is to use the step function as a metric of time, creating a time unit of a specified resolution, and re-writing the properties to be "WITHIN X steps". Although there is no guarantee that the monitor will not miss an event, this uncertainty is common for all computer property evaluations at runtime. Another possibility could be to manually edit the Copilot code to include an input signal called time that contains the current real time. There are other languages in which this might be easily expressed, but they lack guarantees about memory consumption during execution.

The RV system displayed low latency for both violation reporting at 33 ms and front-end warnings, while maintaining high performance on tested hardware. The monitor was capable of bidirectional communication, which facilitated the implementation of safety controller features tested in simulation environments. All of this was achieved on Ubuntu systems, utilizing formally verified hard real-time code generated by Copilot.

To summarize, the implementation of the Copilot RV framework in a ROS 2 environment demonstrated significant improvements in detecting and responding to safety violations in both simulated and real-world tests. The overhead introduced by the RV monitors was within acceptable limits, and the change made to accommodate the asynchronous nature of data publishing ensured minimal delays in violation detection.

Significance of Findings: Runtime Verification is a term growing in popularity, and with good reason. The increased focus on automation and robotics by technology-leading companies raises questions regarding how safety and compliance will be maintained in a steadily advancing automated society. In the agricultural domain, the increased use of automation with heavy machinery begs the question: how safe are our systems? One effective tool for ensuring reliability and safety is implementing RV. This choice is made even simpler with RV being increasingly lightweight formal methods for analyzing the behavior of software or cyber-physical systems during runtime [7]. With automated generation, the bar of entry is lowered for using set tools. Thus, in the case of a violation of specified requirements, a myriad of choices are made available, such as delivering information to the user, intercepting wrong messages and overwriting them, or implementing mitigation actions like turning the system off in the case of a fault, as in field test (Sec. 5.3). By implementing RV, the reliability and safety during runtime is increased by lessening the risk of violations going undetected. With proper implementation we have seen that RV can eliminate the need for rigorous offline testing and execution traces prior to runtime, instead utilizing formally verified requirements, which aid in solving safety monitoring for systems with infinite state spaces.

The results shown here should inspire RV to become more widely deployed across the entire implementation chain, being used from early design phases, through system verification, testing, and during deployment, all stages of the engineering life-cycle [8]. The goal is to increase safety and reliability from the moment of conception until deployment, with simplified tools that anyone can use through the monitor generation based on NL. RV continues to prove itself as a beneficial application of formal verification, featuring the following enhancements from our findings:

Enhanced Safety
The findings highlight the RV framework's potential to significantly enhance the safety of robotic systems operating in dynamic and potentially hazardous environments.

Improved Efficiency
The ability to detect and mitigate violations in real-time contributes to more efficient and reliable robotic operations, reducing the risk of accidents and system failures.

Scalability and Flexibility
The RV framework's scalability and adaptability to different ROS environments, specification models, and scenarios demonstrate its potential for widespread application in various fields, including agriculture.

6.2 Comparison with Previous Research

Our findings align with previous studies such as the ROSMonitoring tool, ROSRV, and TeSSLa-ROS-Bridge, which also showed that runtime verification can effectively enhance the safety of robotic systems. There are, however, pros and cons to weigh when faced with the choice between several frameworks. When weighing the impact of the Copilot monitor, it would be best to compare it with the ROSMonitoring tool, as it has more extensive research on the effects of introducing monitors to the system. Our approach that uses asynchronous verification with the synchronous Copilot language, was able to achieve competitive performance with the ROSMonitoring software. By presenting the monitor under gradually increasing system loads in a similar fashion as performed on the ROSMonitoring tool [5], distinguishing the weights of the frameworks becomes a simple task of comparing the empirical data of Fig. 17 and Fig. 40. The ROSMonitoring monitor shows an introduction of overhead at message rates just shy of 5000 Hz with 500% overhead at 10000 Hz. The OGMA-Copilot monitor, on the other hand, introduces overhead at message rates just shy of 12500 Hz with just over 50% overhead at 20000 Hz. In both cases only the impact of the presence of the monitor was tested, as such the property was fixed to be verified. This demonstrates how lightweight the OGMA-Copilot monitor is over other RV monitors, making it better suited for running on lower to middle-end hardware. This is likely to be the case when comparing OGMA-Copilot to ROSRV as well, with ROSRV having a high risk of bottle-necking due to it's centralized architecture mentioned in Sec. 2.3.3.

The OGMA-generated monitor and the TeSSLa-ROS-Bridge share the most common properties, both being designed with ROS 2 in mind, with similar structures when comparing generalized diagrams in Fig. 18 and 25. Seemingly, one of the most important aspects to consider is the requirement specification language for each respective framework: TeSSLa and FRETISH . TeSSLa is marketed as a convenient language for specification and verification of cyber-physical systems with its natural and short syntax, offering more than just RV, with datastream analysis and semantic documentation. The TeSSLa-ROS-Bridge monitor additionally offers advanced output streams allowing for the checking of value bounds or statistical properties, and timing-based specifications. This should be possible for the OGMA-Copilot monitor, but would conflict with the predictable memory and time bounds as OGMA-Copilot prides itself on being hard real-time C99.

OGMA and FRET being compatible brings several merits for the development process of cyber-physical systems. Given that FRETISH is so close to natural language, the small learning curve allows researchers and system developers to quickly set up and deploy a monitor to test a system solution against its intended behavior. By pin-pointing errors and being able to act as a safety controller through minimal alterations, the monitor makes for a safer developmental process with faster localization of bugs or defects. The solution being non-intrusive guarantees that the monitor will not affect the system. The continued development by NASA on the OGMA-Copilot monitor could only serve to simplify the RV process and deliver formally verified specification systems to early prototyping and deployment of robotics. Both have the benefit of being open-source and readily available for ROS 2. ROSMonitoring is, however, currently developing a ROS 2 system, but it is yet to be released. At the inception of the study, few options for RV of ROS 2 existed.

6.3 Implications of the Findings

The reduced overhead and improved response times observed in our study suggest that the Copilot framework is well-suited for deployment in safety-critical applications. This can potentially lead to more robust and reliable robotic systems capable of operating in dynamic environments such as agricultural robotics. The method has been proven to be competitive with existing tools, with strong benefits in ease of use and quick deployment times. Continued development of automatic generation of RV monitors, like the method proposed here, will in the future help in lowering the threshold of safety upkeep and reliability enhancement. This includes eliminating the need for meticulous offline testing for infinite state machines, where formal RV requirements provide improved model coverage and quick response times. Inclusion of advanced tracking and prediction algorithms will possibly further improve the safety and reliability of agricultural robots by predicting safety-critical situations and reliability concerns before they become real problems.

Throughout the thesis, our research has contributed to the active NASA development of the RV tool OGMA and Copilot by addressing code issues and engaging in conversation with the lead developer, Dr. Ivan Perez. Through our work OGMA and Copilot issues on GitHub have been addressed and solved by the diligent workers at NASA. The following is a list of issues we have addressed during development, which have been handled by NASA:

Fixed for version 1.3 of OGMA:
(ogma-core) Equivalence operator translation issue: `https://github.com/nasa/OGMA/issues/126`
(ogma-core) Spec2Copilot translator not sanitizing handler names: `https://github.com/nasa/OGMA/issues/127`
(copilot-core) Linker error during compilation of ROS application: `https://github.com/nasa/OGMA/issues/130`

Fixed for version 1.4 of OGMA:
(ogma-cli) Add docker-file with complete workflow for ROS 2 backend including compiling the ROS app: `https://github.com/nasa/OGMA/issues/136`
(ogma-core) ROS backend incorrectly mapping float and double types: `https://github.com/nasa/OGMA/issues/138`
(Github)
(ogma-cli) Missing explanation of DB structure for ROS: `https://github.com/nasa/OGMA/issues/143`

7 Limitations

This section outlines the various limitations encountered in this study, both from a software (Sec. 7.1) and hardware (Sec. 7.2) perspective, as well as practical recommendations (Sec. 7.3) for overcoming challenges in future implementations.

7.1 Software Limitations

The software tools used in this thesis consisted of FRET, OGMA, Copilot, Flask, ROS 2, and mainly Python for back-end programming on a Linux distribution. Summarized, the limitations experienced with the software consisted of timing, communication, and dependency issues, often related to the Copilot to ROS 2 language barrier, which is preliminary at this point.

A limitation of most runtime verification software for ROS is that topics are the only monitor messages checked. In complex ROS applications, however, it is quite common to have external commands, but usually, these commands are received on a topic to which an action client subscribes. This way, the runtime verification platform can also react to external commands.

Given the monitor-to-system language barrier, with Copilot expecting synchronous information streams, and ROS 2 adapting an asynchronous, event-based, pub-sub architecture, integrating the two systems requires a transition-point to synchronize data flows. Keep in mind that all code that OGMA and Copilot generate for the monitors is hard real-time, meaning it has predictable memory and time bounds, no mallocs, no for-loops, and no recursion. To circumvent timing issues, you can allocate a discreet time interval to the past-time algorithm while writing requirements. You can check the system state as many times as you want within this interval and evaluate if there have been any violations of requirements each time you transition to the next interval. Let's say you have a time interval of 1 second. You can have a set resolution of one tenth or hundredth of a second to log the state 10 or 100 times per second and evaluate whether one of these 10 or 100 states triggers a violation. This, however, means that you have to allocate memory of unbounded size, losing the predictable memory and time bounds. There's also no guarantee that you won't miss an event, even with the smallest possible window, as computers still poll at fixed intervals.

One could avoid a step interval by encoding some of these properties directly in Copilot by providing an input signal called "time" that contains the current (real) time. This requires writing custom Copilot expressions and extensions. Researchers are encouraged to see the documentation for the Haskell Copilot library [55], as well as the official technical report by NASA [56].

One limitation of our study is the dependency on specific ROS distributions, which may limit the generalizability of our findings to other robotic platforms. Additionally, the simulation environment cannot fully capture the complexity of real-world scenarios, which may affect the performance of the RV framework.

The Thorvald-005 robot originally runs on an end-of-life (EOL) ROS Kinetic distribution. This was a problem which caused limitations to the degree of bidirectional communication we were able to perform

without extensive work on setting up a Rosbridge on a Kinetic operating system. While possible, it was deemed too large a task for the value of the outcome. For use of the monitor system we assume one would either have the time to properly set up a bridge connection or that the CPS hardware will use compatible distributions that are up to date, making the bidirectional communication possible.

7.2 Hardware Limitations & Assumptions

The hardware used in the project consisted of laptop computers, a desktop computer, one RealSense camera model D435 and the Thorvald-005 robot. Summarized, the limitations experienced with the use of these consisted of performance and incompatibility issues that limited the scope of what we were able to achieve but not the results themselves.

For testing purposes, the 10 meter range of the RealSense was a problem that created a lot of noise on the distance results. The limited number of depth cameras led to a limited field of view (FOV) with no mitigation for objects blocking the frame. In an ideal environment there would be multiple cameras throughout the poly-tunnel and on the robot, giving a 360° view of the surroundings mitigating objects from blocking the view. These were assumptions we made for deployment of such a system, meaning that all issues related to limited FOV and detection of a subject was not important for the end results. The distance inaccuracy was also accommodated with the use of data filtering.

The laptops used during the project often had issues when all programs were running, limiting the number of frames we were able to process from the RealSense. This was mostly related to back-end systems for the website and logging that caused a high computational cost to the system during runtime. This was not ideal, and would assumably not be active during normal operation, only for testing. Otherwise, the systems upheld the monitor system and was able to react to violations and call the Thorvald-005 robot during field-testing. The monitor itself is very lightweight, which is a benefit for implementing it on a CPS. During simulation there were no issues related to performance since the desktop computer was used, which had a major performance increase when compared to the laptops.

7.3 Practical Recommendations

Practitioners deploying runtime verification in robotic systems should consider the asynchronous nature of data publishing to minimize system overhead. Moreover, adapting the Copilot framework to support predictive verification could further enhance its effectiveness in preemptively addressing potential safety violations. We also recommend careful study of how the requirements are defined as they make or brake the monitor system if not properly maintained, for this FRET is useful with its realizability checking features. Especially timing constraints can be difficult to define, as proper knowledge of the desired LTL logic is instrumental in understanding how to correctly create the model specification. The use of metric temporal logic (MTL) operators in continuous or discrete time lands into that category of unmonitorable in runtime very quickly, due to the complexity and potentially high resource demands of continuously verifying time-bounded properties. One way of allowing MTL operators involves modifying the OGMA-generated apps to run Copilot's step function at discrete intervals, using a resolution of ”x” number of steps as a timing constraint. This method serves as a workaround, while waiting for a permanent fix.

8 Future Works

The exploration and development of generative Runtime Verification (RV) for ROS 2 systems has yielded significant insights and advancements. However, several areas remain ripe for further research and development to enhance the capabilities, efficiency, and applicability of RV in robotic systems. This section outlines key directions for future work, aimed at addressing current limitations and expanding the framework's utility. By integrating predictive capabilities, automating user interface generation, improving signal processing, and enhancing hardware and software compatibility, the RV framework can be further refined to meet the evolving demands of modern robotics. Collaboration with industry and research organizations will be crucial in driving these innovations forward. The topics related to future work is addressed in separate paragraphs for added clarity and understanding.

Predictive Runtime Verification: Integrating predictive RV capabilities to anticipate and address potential violations before they occur would significantly enhance system safety and reliability. This could involve developing algorithms to predict future states of the robot and preemptively mitigate safety issues.

Automated User Interface Generation: Creating automated tools for generating user interfaces that facilitate the monitoring of online RV systems. This would improve usability and make it easier for operators to interact with and understand the system's status in real-time. Reintegrating the D3 library of displaying state machine systems, with simple interface for choosing hazard mitigation tactics for different requirements.

Enhanced Stream to Event-Based Signaling: Improving solutions for converting stream-based signals to event-based ones. Given the challenges encountered with asynchronous data publishing in ROS, refining this aspect would lead to more accurate and efficient monitoring. Solved either by improved utilization of the step function, changing the copilot monitor code, or reworking the system requirements to have time from step count resolution.

Advanced Tracking and Prediction Algorithms: Including advanced tracking and prediction algorithms to improve the system's ability to handle dynamic environments. This enhancement could help in predicting and preventing safety-critical situations, further increasing the reliability of agricultural robots and other CPS applications. This would solve issues related to the Inferer switching class of subject when objects are blocking sight, and would change the dynamic of how mitigation actions are used, calling necessary actions before safety-critical events arise instead of after.

Hardware Improvements: Addressing hardware limitations, such as the range and accuracy of RealSense cameras and the performance of YOLOv6 under different conditions. Upgrading to more powerful computing platforms and improving sensor accuracy would enable the RV framework to perform better in diverse and challenging environments. Improved hardware would also benefit in finding a improved stream to event-based signaling method, by lessening limitations connected to performance.

Integration with Bridge Solution for Other ROS Distributions: Extending the compatibility of the RV framework to support a wider range of ROS distributions beyond ROS 2, ensuring broader applicability and ease of integration with various robotic platforms. This would have helped during field testing with the ROS Kinetic Thorvald-005 robot.

Refinement of Requirement Specification Models: Further research into refining the requirement specification models in FRET to handle asynchronous publishing more effectively. This could involve developing new approaches to validate requirements within specific time frames, thereby improving the monitor's ability to handle event-based topics using a stream-based approach.

Continued Development of RV Tools for ROS 2: Continued collaboration with organizations like NASA and the team behind ROSMonitoring to address issues and improve tools. Engaging with the broader research community and industry partners will help in refining the RV framework and ensuring it meets the evolving needs of modern robotic systems in agricultural domains, as well as others.

9 Conclusion

To reiterate, this thesis set out to develop and evaluate a robust Runtime Verification (RV) platform for ROS 2 applications, automatically generated from natural language requirements and equipped with a user interface. The first research question was stated:

(1) Is it possible to create a pipeline for the automatic generation of runtime verification monitors for ROS 2 systems using current software tools?

The primary goal was therefore to provide a reliable implementation example of an automatically generated monitor, and contribute to the ROS 2 RV space. Through the use of NASA-developed tools such as FRET, OGMA, and Copilot, we successfully achieved this objective. The second research was stated:

(2) Does the runtime verification monitor help elevate safety during all phases of the engineering life-cycle?

The secondary goal was therefore to demonstrate the functionality and effectiveness of the monitor and compare it to previous runtime verification software tools. It was demonstrated that the monitor is capable of elevating safety through debugging during all phases of the engineering life-cycle. Key findings also illustrate that the RV framework developed in this research significantly improves the safety and reliability of robotic systems by detecting and responding to safety violations effectively. The framework's low overhead and rapid response times makes it well-suited for deployment in dynamic environments, such as the agricultural domain. This was demonstrated in the specified use case of an autonomous UV-light treatment robot of powdery mildew of strawberries.

However, this research also encountered several limitations. The dependency on specific ROS distributions and the simulation environment's inability to fully capture real-world complexities are notable constraints. Additionally, hardware limitations, such as the performance of RealSense camera and computational constraints of the laptops used, posed challenges during testing.

Future research should focus on integrating predictive RV capabilities, automating user interface generation, and improving stream to event-based signaling. Enhancing hardware configurations and extending compatibility with other ROS distributions will further improve the framework's effectiveness. Continued collaboration with industry organizations like NASA and the broader research community will be crucial in refining the RV framework and addressing its current limitations.

In conclusion, this research has made significant strides in the field of runtime verification for ROS 2 systems, contributing valuable insights and methods to enhance robotic safety and reliability. The successful implementation and evaluation of the Copilot RV framework for ROS 2 mark an important step forward, with promising implications for future advancements in this critical area. We hope our work will contribute in the further enhancement of workplace safety and reliability of robotic operations, with our thesis being used as an example for teaching how to employ RV systems for simulation and field deployment. Our work has prefaced how integration of RV monitors are easier to implement than ever through the use of generative monitor creation, and that further research on the topic will be crucial in creating a lower bar for entry, making sure that all safety critical CPS equipment is not only adhering to requirements during offline testing, but also during runtime.

"Now is no time to think of what you do not have. Think of what you can do with what there is."

— Ernest Hemingway, The Old Man and the Sea

References

[1] Marcel Bergerman, John Billingsley, John Reid, and Eldert van Henten, *Springer handbook of robotics*, vol. 200, Springer, 2008.

[2] John M Shutske, "Agricultural automation & autonomy: safety and risk assessment must be at the forefront," 2023.

[3] Haskell Foundation, "43 – ivan perez," Haskell Foundation Podcast, March 2024, Accessed: 2024-03-08.

[4] Ivan Perez, Frank Dedden, and Alwyn Goodloe, "Copilot 3," Tech. Rep., NASA, 2020.

[5] Angelo Ferrando, Rafael C Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascardi, "Rosmonitoring: a runtime verification framework for ros," in *Towards Autonomous Robotic Systems: 21st Annual Conference, TAROS 2020, Nottingham, UK, September 16, 2020, Proceedings 21*. Springer, 2020, pp. 387–399.

[6] Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alexander Will, and Patrick J. Martin, "Monitoring ros2: from requirements to autonomous robots," *Electronic Proceedings in Theoretical Computer Science*, vol. 371, pp. 208–216, Sept. 2022.

[7] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger, "Introduction to runtime verification," *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 1–33, 2018.

[8] Mustafa Adam, Elias E. Hartmark, Tage Andersen, David A. Anisi, and Ana Cavalcanti, "Safety assurance of autonomous agricultural robots: from offline model-checking to runtime verification," in *IEEE 20th International Conference on Automation Science and Engineering (CASE)*, 2024.

[9] Dimitra Giannakopoulou, Anastasia Mavridou, Julian Rhein, Thomas Pressburger, Johann Schumann, and Nija Shi, "Formal requirements elicitation with fret," in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020)*, 2020, Document ID: ARC-E-DAA-TN77785.

[10] NASA, "Fret: Formal requirements elicitation tool," `https://github.com/NASA-SW-VnV/fret`, 2024, Accessed: 2024-01-26.

[11] Andreas Katis, Anastasia Mavridou, Dimitra Giannakopoulou, Thomas Pressburger, and Johann Schumann, "Realizability checking of requirements in fret," NASA Technical Memorandum NASA/TM-2022-22007510, NASA, 2022.

[12] Ivan Perez, "Runtime verification with ogma," in *Invited Talk to University of California*, 2023.

[13] Hamza Bourbouh, Guillaume Brat, and Pierre-Loic Garoche, "Cocosim: an automated analysis framework for simulink/stateflow," in *Model Based Space Systems and Software Engineering-European Space Agency Workshop (MBSE 2020)*, 2020.

[14] Kind2 Development Team, "Kind 2 model checker," `https://github.com/kind2-mc/kind2`, 2024, GitHub repository.

[15] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli, "The kind 2 model checker," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 510–517.

[16] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani, "The jkind model checker," in *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*. Springer, 2018, pp. 20–27.

[17] Stéphane Demri and Denis Poitrenaud, "Verification of infinite-state systems," 2011.

[18] NuSMV Team, "Nusmv: a new symbolic model checker," `https://nusmv.fbk.eu/`, Accessed: [19.02.2024].

[19] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri, "Nusmv: a new symbolic model checker," *International journal on software tools for technology transfer*, vol. 2, pp. 410–425, 2000.

[20] Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alwyn Goodloe, and Dimitra Giannakopoulou, "Automated translation of natural language requirements to runtime monitors," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 387–395.

[21] NASA, "Ogma: Operational goal-based mission analysis," `https://github.com/nasa/ogma`, 2024, Accessed: 2024-01-26.

[22] Haskell Language, "Haskell language official website," 2024, Accessed: 2024-03-11.

[23] Haskell Wiki, "Haskell," `https://wiki.haskell.org/`, 2024, Accessed: 2024-03-12.

[24] "Copilot language," `https://copilot-language.github.io/index.html`, Accessed: 2024-03-08.

[25] Copilot Development Team, *Copilot Tutorial*, 2024, Accessed: 2024-03-08.

[26] Ryan G Scott, Mike Dodds, Ivan Perez, Alwyn E Goodloe, and Robert Dockins, "Trustworthy runtime verification via bisimulation (experience report)," *Proceedings of the ACM on Programming Languages*, vol. 7, no. ICFP, pp. 305–321, 2023.

[27] The Copilot-Language Team, "Copilot verifier," `https://github.com/Copilot-Language/copilot-verifier`, 2024, Accessed: 2024-03-13.

[28] Miguel Grinberg, *Flask web development: developing web applications with python*, " O'Reilly Media, Inc.", 2018.

[29] ROS Wiki, "rosbridge_suite," `http://wiki.ros.org/rosbridge_suite`, 2024, Accessed: 2024-03-09.

[30] Miguel Grinberg, "Flask-socketio documentation," `https://flask-socketio.readthedocs.io/en/latest/`, 2024, Accessed: 2024-03-10.

[31] ROS Wiki, "rospy," http://wiki.ros.org/rospy, 2024, Accessed: 2024-03-09.

[32] Roland van Dierendonck, Sam van Tienhoven, and Thiago Elid, "D3. js: Data-driven documents," *JS: Data-Driven Documents*, 2015.

[33] Marian Johannes Begemann, Hannes Kallwies, Martin Leucker, and Malte Schmitz, "Tessla-ros-bridge–runtime verification of robotic systems," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2023, pp. 388–398.

[34] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu, "Rosrv: Runtime verification for robots," in *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. Springer, 2014, pp. 247–254.

[35] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma, "Tessla: temporal stream-based specification language," in *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*. Springer, 2018, pp. 144–162.

[36] Cansu Erdogan, *ROSRV: Runtime verification for the robot operating system*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2015.

[37] ROS Wiki Contributors, "Ros concepts," 2022, Accessed: 2024-06-09.

[38] Clearpath Robotics, "Computer 1," https://www.clearpathrobotics.com/assets/guides/noetic/ros/Intro%20to%20the%20Robot%20Operating%20System.html, 2015, Accessed: 2024-06-11.

[39] Sean Rivera, Antonio Ken Iannillo, Sofiane Lagraa, Clément Joly, and Radu State, "Ros-fm: fast monitoring for the robotic operating system (ros)," in *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2020, pp. 187–196.

[40] ROS 2 Documentation, "About different middleware vendors," https://docs.ros.org/en/foxy/Concepts/About-Different-Middleware-Vendors.html, 2023, Accessed: 2023-06-11.

[41] RoboStar, "Robostar: Software engineering for robotics," https://robostar.cs.york.ac.uk, Accessed: 2024-07-10.

[42] Alvaro Miyazawa, Pedro Ribeiro, Kangfeng Ye, Ana Cavalcanti, Wei Li, Jim Woodcock, and Jon Timmis, "Robotool developer's manual," *Sofware Engineering for Robotics*, 2020.

[43] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock, "Robochart: modelling and verification of the functional behaviour of robotic applications," *Software & Systems Modeling*, vol. 18, pp. 3097–3149, 2019.

[44] Mustafa Adam, Kangfeng Ye, David A. Anisi, Ana Cavalcanti, Jim Woodcock, and Robert Morris, "Probabilistic modelling and safety assurance of an agriculture robot providing light-treatment," in *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, 2023, pp. 1–7.

[45] Intel Corporation, "Intel realsense depth camera d435," 2024, Accessed: 2024-06-07.

[46] Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, et al., "Yolov6: A single-stage object detection framework for industrial applications," *arXiv preprint arXiv:2209.02976*, 2022.

[47] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu, "A survey of deep learning-based object detection," *IEEE access*, vol. 7, pp. 128837–128868, 2019.

[48] SA Sanchez, HJ Romero, and AD Morales, "A review: Comparison of performance metrics of pretrained models for object detection using the tensorflow framework," in *IOP Conference Series: Materials Science and Engineering*. IOP Publishing, 2020, vol. 844, p. 012024.

[49] Meituan, "Yolov6: a single-stage object detection framework dedicated to industrial applications," `https://github.com/meituan/YOLOv6`, 2022, Accessed: 2023-06-09.

[50] Leonardo Guevara, Muhammad Khalid, Marc Hanheide, Simon Parsons, et al., "Assessing the probability of human injury during uv-c treatment of crops by robots," 2021.

[51] Yong Shing Voon, Yunze Wu, Xinzhi Lin, and Kamran Siddique, "Performance analysis of cpu, gpu and tpu for deep learning applications," *Professor Ka Lok Man, Xi'an Jiaotong-Liverpool University, China Professor Young B. Park, Dankook University, Korea Chairs of CICET 2021*, vol. 16, pp. 12, 2021.

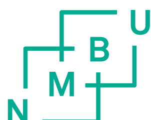
[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[53] Field Robot Event, "Virtual maize field: A simulation environment for the field robot event," `https://github.com/FieldRobotEvent/virtual_maize_field`, 2024, Accessed: 2024-05-12.

[54] Neil Thacker and A Lacey, "Tutorial: The kalman filter," *Imaging Science and Biomedical Engineering Division, Medical School, University of Manchester*, vol. 61, 1998.

[55] Copilot Development Team, "Copilot Library PTLTL - Documentation," `https://hackage.haskell.org/package/copilot-libraries-3.19.1/docs/src/Copilot.Library.PTLTL.html`, 2023, Accessed: 2024-07-06.

[56] NASA, "Title of the Report," Tech. Rep. 20200003164, NASA, 2020, Accessed: 2024-07-06.



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway