Trustworthy ROS Software Architecture for Autonomous Drones Missions: From RoboChart Modelling to ROS Implementation

*

Nabil Shaukat[1,a], Shival Dubey[1,b], Bilal Kaddouh[1,c], Andy Blight[1,d], Lenka Mudrich[1,e],
Pedro Ribeiro[3,f], Hugo Araujo[4,g],
Rob Richardson[1,h], Louise Dennis[2,i], Michael Fisher[2,j], Ana Cavalcanti[3,k], and Mohammad Mousavi[4,l]

[1] School of Mechanical Engineering, University of Leeds,UK
[2] Department of Computer Science, University of Manchester,UK
[3] Department of Computer Science, University of York
[4] Department of Informatics, Kings College London, UK
Emails: {nabil[a], shival[b], bilal[c], andy[d],, rob[e], lenka [f], pedro[g], hugo[h], louise[i], michael[j], ana[k], mohammad[i]}@ac.uk

***Abstract*—Drones are becoming essential tools in emergency situations such as search and rescue, surveillance, and firefighting. These applications make the development of trust worthy software a priority to increase efficiency, cut costs, and reduce risks, in future replacing human personnel in challenging areas. RoboChart, a platform-agnostic tool, brings numerous benefits to the robotics community by providing a language and high-level tool of describe model and via automatic verification and exhaustive testing of model. However, when it comes to real robotics it is essential to use testing tools suitable for specific robotic software. In the context of Robotic Operating System (ROS), it imposes its own design constraints, best practices, and communication requirements. This paper aims to apply RoboChart modeling approach to autonomous firefighting drone, putting forward a low-level ROS software architecture that aligns with RoboChart models, ensuring the trustworthiness of the system during operation.**

***Index Terms*—robochart, ros, drone, firefighting, software, architecture,trustworthy**

I. Introduction

As technological advancements progress, drones are becoming essential in addressing a variety of emergency operations. Drones demonstrate adaptability in disaster response, healthcare, and safety emergencies, reshaping approaches with advanced features and redefining the approach to addressing challenges. In disaster response, they survey affected areas, identify hazards, and provide real-time information to emergency responders, contributing not only to surveillance but also improving the efficiency of search and rescue operations [1], [2], [3]. Furthermore, when drones are used for firefighting, their capabilities not only enhance the assessment of fires but can allow them to actively participate in extinguishing them. When suitably equipped, these drones contribute to firefighting efforts, providing an additional dimension to their role in emergency missions [4]. By enhancing situational awareness for firefighting teams, drones contribute to the effectiveness of firefighting strategies and operations. For these drones to gain societal acceptance and deliver benefits, they must be trustworthy. This means they must operate consistently and predictably within their intended functions safely, minimizing unexpected behavior or errors with following ethical principles. This requires better design, rigorous testing and verification during their development and deployment.

Software models and verification tools for robots are gaining popularity and evolving to ensure accurate representations for reliable, and safe behavior [12]. These tools aid in compliance with specifications and building trust among stakeholders. They contribute to iterative development, legal and ethical compliance, and adaptability to evolving requirements, fostering the responsible deployment of drone technology. Various approaches have been utilized to model behavior and verify systems, some of the notable research in this direction are [7], [8], [9], [10]. RoboStar, Robotool and RoboChart are distinguishes by providing standard semantics with a CSP-based formal verification tool which help in building trustworthy autonomous systems through rigorous mathematical analysis [11], [12]. Moreover, it validates critical properties, enhances transparency with a state machine-based representation, and enables early error detection.

ROS (Robot Operating System) is widely used by the robotics community due to its open-source nature which promotes global collaboration and information sharing within the robotics community. By abstracting underlying hardware details, ROS allows developers to focus on software, facilitating the creation of robot applications compatible with various hardware platforms. The framework offers a rich set of tools and libraries for diverse robotics tasks, including simulation, visualization, control, perception, and planning [14].

The ROS software architecture for autonomous drone provides a comprehensive and coherent representation of the entire system, including its components, interactions, and overall structure. It typically illustrates the nodes, topics, messages, services, and actions that constitute the ROS-based system. An undefined architecture may result in inefficient resource utilization and communication patterns within the ROS system. This can lead to suboptimal performance, including increased latency, and higher resource consumption. Moreover,the absence of a well-defined ROS software top-level architecture can pose significant challenges in applying formal verification techniques effectively, potentially impacting the completeness, accuracy, and scalability of verification efforts.

This paper propose top-level ROS software architecture for firefighting drone, a step towards constructing truly trustworthy autonomous drone by supporting RoboChart formal verification process.

II. Challenges of Formal Verification without a Well-Defined ROS Top-Level Architecture

In the context of RoboChart formal verification, a well-defined top-level architecture in ROS software development is essential for following reasons:

- A clear architecture helps define the scope of formal verification efforts. By delineating the system's high-level structure, including ROS nodes, topics, messages, and services, it enables developers to identify which components need to be verified formally.
- With a defined architecture, developers can specify formal properties and requirements more precisely. They can express desired system behaviors, safety properties, and functional constraints in a formal language, facilitating formal verification techniques' application.
- Formal verification often relies on assumptions about the system's environment and behavior. A clear architecture helps clarify these assumptions by defining interfaces, communication patterns, and dependencies between ROS components, ensuring that verification assumptions are well-founded.
- A well defined architecture aids in ensuring comprehensive verification coverage. Developers can systematically verify each aspect of the system's behavior against formal specifications, including error-handling mechanisms, fault tolerance strategies, and safety-critical functions.

III. System Overview

The system described herein is a case study within the UKRI Trustworthy Autonomous Systems project, focusing on a firefighting drone designed for search, track, and extinguish operations in known building layouts [15]. The firefighting drone autonomously locates and extinguishes fires, directing its suppressant towards identified blazes. The execution of firefighting drone operations relies on perception system. The perception system accurately detects fire sources, enabling the drone to navigate safely near the fire and precisely align itself for effective extinguishing. The success of the firefighting drone hinges on its trustworthiness. This means reliable decision-making, safeguarding both itself, operator and the environment as the drone navigates close to fire and extinguishes it, even in complex situations.

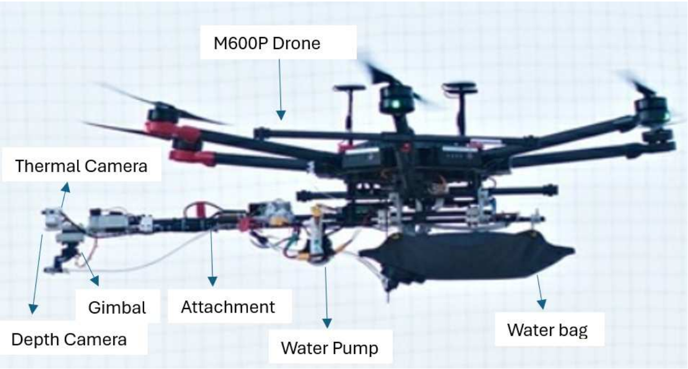


Fig. 1. Real Robotics Aerial Firefighting System

The Real Robotics Aerial Firefighting System shown in Figure 1 utilizes the DJI M600 Pro drone, integrating an Intel NUC for on-board vision processing and communication. Two key sensors, the Intel RealSense D435i camera for depth information and the TR-EVO-T90-USB thermal camera for fire detection, contribute to the system's vision data. The drone incorporates a custom-designed water pump subsystem for firefighting purposes. Planning and control operations are managed through a ROS-based Aerial and Ground Control Station. The entire system is designed to comply with size and weight constraints for safety.

For the mounting of the RealSense D435i and TR-EVO-T90-USB cameras, as well as a gimbal and nozzle, a custom-designed 3D-printed adapter attachment is utilized. This attachment is fixed to the bottom of the drone, designed to ensure the camera and nozzle assembly remains unaffected by the UAV rotor downdraft. To enhance stability when aiming a jet of water, three carbon fiber tubes are incorporated into the design.

IV. ROS Software Architecture

The primary objective is to establish a ROS top-level architecture with communication interfaces between the nodes that are capable of implementation with both ROS1 and ROS2. This architecture is designed to integrate with the RoboChart framework while adhering to the best practices within the ROS ecosystem. ROS architecture facilitates the understanding and verification of the interactions and dependencies between these components, ensuring correct system-level behavior.

Without a defined architecture, developers may lack a clear understanding of how different components of the ROS system should be structured and interact with each other. This can lead to confusion and inefficiencies in the development process.

A. Mapping RoboChart Model building blocks to ROS nodes

RoboChart divides a large firefighting drone system into a number of smaller models. This breakdown enables the isolated evaluation of each functional component, facilitating unit testing and systematic verification of requirements. By addressing potential issues at the individual block level before integration, this approach significantly enhances the overall reliability of the system.

The following sections give an overview of the individual blocks.

1) Flight Controller Interface Block: The Flight Controller Interface (FCS) is the primary interface to the drone's software development kit (SDK) flight controller, designed to make the other blocks independent of the drone flight controller SDK. This ensures that the code can be easily implemented on other drones by modifying the FCS, while keeping the other blocks unchanged. The module handles tasks such as take-off, landing, global positioning through waypoints, and local positioning through visual navigation.

2) Fire Detection Block: The Fire Detection Block utilizes inputs from both the depth and thermal cameras to identify the location of a fire, determine its distance from the wall, and align the drone toward the wall. By fusing data from the thermal and depth cameras, the drone gains a comprehensive understanding of the environment enabling better situational awareness for firefighting operations.

3) Spray Aim Block: The Spray Aim Block receives real-time fire tracking data, directs the spraying nozzle accurately towards the fire location, and provides status feedback. It takes input from the fire detection block, processes fire status data, calculates yaw and pitch for motor control, sends commands, receives feedback on motor angles, and coordinates with the flight planner to turn the pump on when signalled and water level is sufficient, and turn pump off when signalled by planner.

4) Battery Monitor Block: This block monitors the drone's battery status and transmits information categorized as OK, Mission Critical, and Safety Critical to the planner, allowing for appropriate actions to be taken by Planner.

5) Water Monitor Block: Water monitoring involves tracking the amount of water used based on pump flow and providing feedback to other blocks on the amount of water remaining. This process includes retrieving pump status data from the Planner and sending feedback to both Flight Planner and the spray aim block, ensuring awareness of the water remaining status.

6) Visual Navigation Block: The visual navigation block ensures that the drone is positioned facing the center of the fire on the wall at the correct distance. It achieves this by sending relative positioning commands to the flight controller interface. The status of visual positioning is also transmitted to the flight planner.

7) Flight Planner Block: The flight planner maintains control through higher level mission control, intelligent path generation, and real-time decision-making. It incorporates data from the fire detection block, dynamically adapting the drone's trajectory and actions while ensuring the mission's safety and facilitating water spraying on the fire. This active involvement includes coordinating responses to detected fires, adjusting flight paths, and intervening when necessary.

RoboChart blocks operate at a high level of abstraction, focusing on conceptual modeling the behavior by using state machines. Directly mapping the models to ROS may lead to integration complexities. ROS deals with lower-level robot control using commands, sensor data processing using callbacks. Bridging this abstraction gap requires careful mapping of behaviours and ROS functionalities. The following principles are employed to map RoboChart model to ROS node functionality.

1. Division of the ROS Nodes dedicated to specific hardware components, provide a clean interaction with the hardware. This separation allows for modular development and easy replacement or upgrade of hardware components. This division improves maintainability, extensibility making it easier to adapt to evolving hardware configurations and requirements without compromising the state machine's overall behavior.

2. In case of complex behavior represented by a block, it is better to break them into smaller behaviors, represented by individual nodes. This can be accomplished using behaviour trees which fosters modularity and reusability. Moreover, nodes can be combined and arranged in various ways using different control structures to create complex, flexible trees. This allows complicated concurrent asynchronous behaviors of the robot while maintaining clarity and structure.

The following table showing mapping RoboChart major blocks onto the ROS nodes with justification

TABLE I
ROS NODES MAPPING

SR	RoboChart Blocks	ROS nodes	Justification
1	Fight Planner	10	Behaviours
2	Fire Detection	3	Hardware division
3	Spray Aim	2	Hardware division

The Spray Aim block was divided into two ROS nodes, with a Spray Aim Main Node handling state machine logic, gimbal and pump clients, and a Hardware Interface Node managing pump and gimbal servers communicating with a microcontroller through a serial interface shown in Figure 2. This division ensures that any issues in the Hardware Interface Node are less likely to propagate to the Spray Aim Main Node. Moreover, if there is a need to upgrade or replace specific hardware components, the Hardware Interface Node can be adapted without altering the overall architecture.

B. ROS Communication Interface

ROS facilitates communication between nodes in a robotic system using three methods: topics for publisher-subscriber communication, services for request-response, and actions for goal-oriented communication. RoboChart, on the other hand, models system behavior at a higher level, abstracting state machines, events, and interactions among robotic components without explicitly defining ROS communication mechanisms. Clearly defined communication interfaces in ROS, in conjunction with RoboChart, verify that communication aligns with

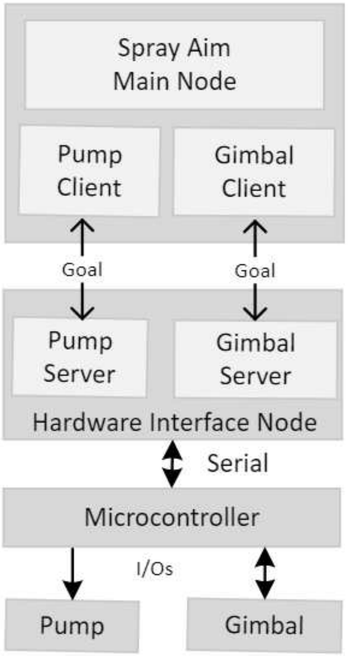


Fig. 2. Division of Spray Aim into Main Node and Hardware Interface Node

expected behaviors modeled, ensuring integration testing and reducing compatibility issues.

For instance, publisher-subscriber and action server are both forms of asynchronous communication. However, designing communication for RoboChart blocks only by using the ROS publisher-subscriber model has some limitations. The asynchronous nature of publisher-subscriber, while beneficial in many scenarios, may not suit strict request-response patterns. Furthermore, the publisher-subscriber model lacks a built-in assurance of message delivery, risking message loss if subscribers are inactive. In large-scale systems, the substantial volume of messages can lead to increased network traffic and processing overhead on subscribing nodes. However, publisher-subscriber is very well suited when information needs to be broadcasted to multiple nodes. Selecting an inappropriate ROS communication method for your firefighting drone can result in adverse outcomes, affecting both mission effectiveness and safety.

The design decisions for ROS communication in the software architecture of the firefighting drone were reached through rigorous analysis of the pros and cons to tailor them specifically for the requirements of a firefighting drone shown in Figure 2. Here are some guidelines developed and used for the fire fighting drone communication design in ROS architecture.

1) Drone Critical Control Commands: The ROS Action Server is chosen for this type of communication as they offer robustness to the design. They can provide detailed real time feedback during goal execution to allow timely intervention and the ability to adapt to dynamic conditions during mission. Moreover, goal preemption enables firefighting drones to respond swiftly to changing priorities or if there is error in the system. It also provides timeout handling mechanisms to prevent indefinite delays, ensuring that critical operations are executed within a reasonable time frame. This is particularly important in firefighting scenarios where timely deployment is necessary for extinguishing fire. Lastly,the result reporting feature ensures conclusive feedback about the completion of task which helps in deciding further actions during the mission.

Drone safety critical commands such as Take off and Landing $fcs/special\_movement$, Searching fire going through waypoints $fcs/FlytoWP$, relative positioning for track the fire and align to the drone in front of fire $fcs/relative\_position$, and water spray command $fp/pump\_start$ in ROS software architecture are shown in red in Figure 3. Using ROS actions ensures that these actions are executed in a controlled and safe manner. Moreover, it allow robust error handling and recovery mechanisms.

2) Enable and Disable Commands: Short duration functionalities of enabling or disabling a mission from a ground station $fp/enable\_mission$ and visual navigation $fp/enable\_vn$ and water monitor $fp/enable\_water\_monitor$ were mapped to ROS services for the firefighting drone and these services are show in brown in ROS architecture 3. Services in ROS operate on a synchronous request-response model which has advantageous for enabling or disabling a mission or initiating water monitor visual navigation, where a clear and immediate response is required. ROS services are inherently blocking, meaning that the client node waits for a response before proceeding. For critical operations like enable or disable the mission, this ensures that the drone acknowledges the request so that the caller knows that the message has been received and has been or will be acted on. If there is an issue with request, the service can return an error response, providing specific details about the problem. This aids in diagnosing issues and taking corrective action.

3) Continuous Broadcast of Data streams: The ROS publisher-subscriber method of sending messages is intended for the transmission of sensor data. Messages can be published on specific topics, allowing multiple nodes to subscribe and receive real-time information for situational awareness and decision-making. This allows the firefighting drone to continuously publish its status, including battery levels, operational conditions water level status, and completion of tasks visual navigation status. This information can be subscribed to by monitoring nodes ensuring continuous monitoring of drone health and performance during firefighting missions

A properly designed ROS architecture for a firefighting drone communication interface can support in verification using RoboChart and help in the achievement of a trustworthy drone. Moreover, a well-structured ROS communication architecture allows testing to enable early issue detection, ensuring discrepancies between the model and desired behavior are identified promptly. Unforeseen issues in an improperly designed architecture may go undetected and leave the drone

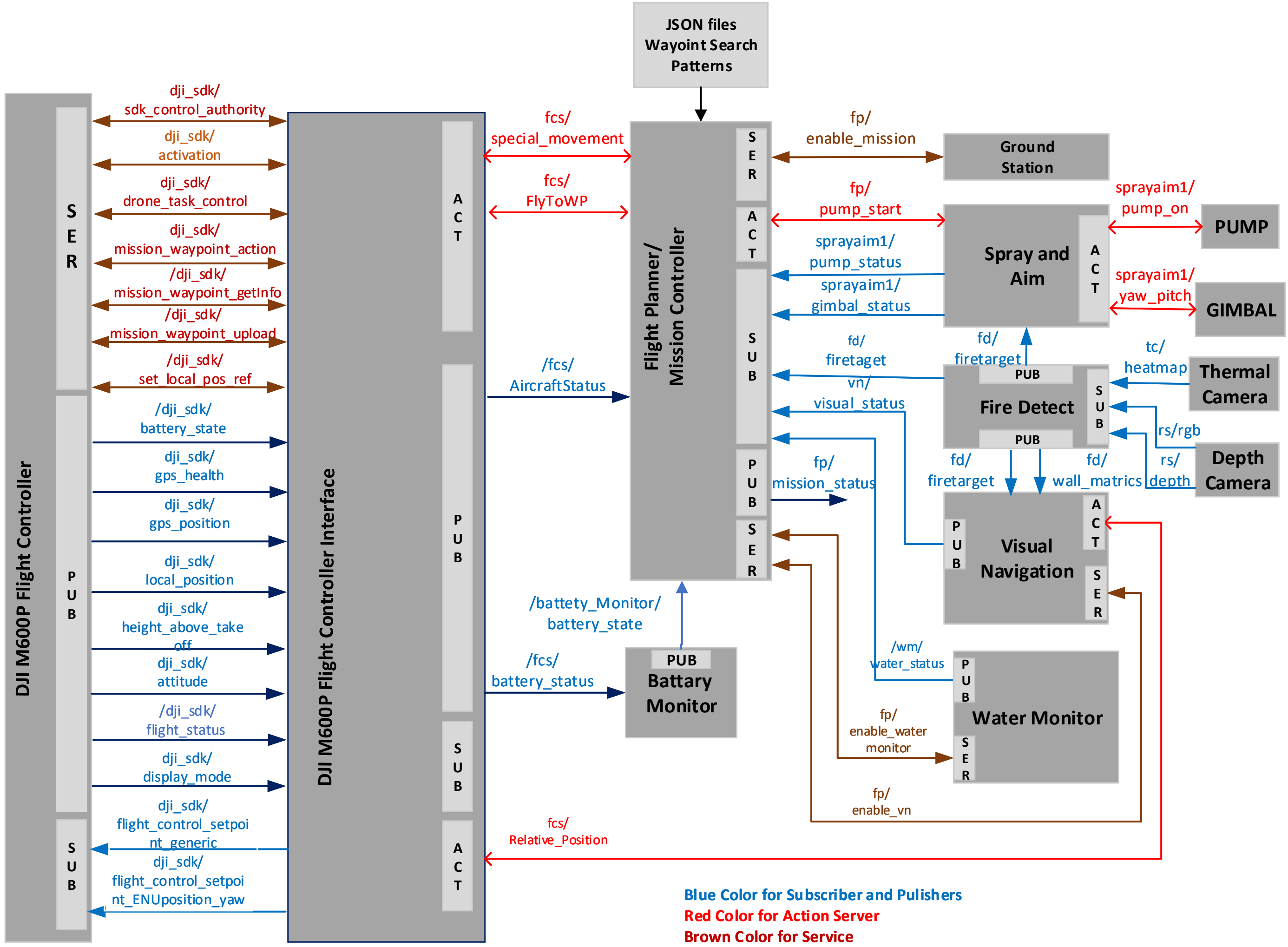


Fig. 3. RoboChart to ROS Software Architecture showing Communication Interface

vulnerable to failures. Inaccurate selection of communication can produce false positives or negatives which highlight issues in non-existent scenarios. A well-structured ROS communication architecture also provides effective error handling mechanisms, contributing to the overall trustworthiness of the drone.

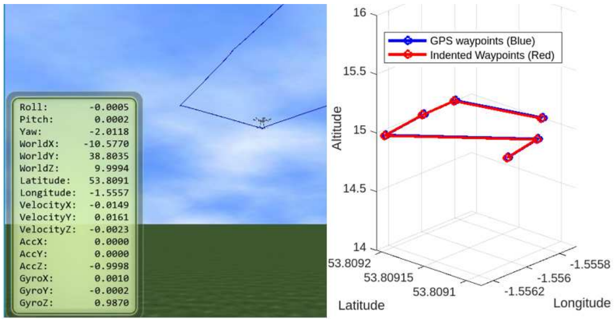
V. ROS Callbacks and State Machine Behavior Models

Synchronization mechanisms are important to address the problems of multiple callbacks in state machines in ROS, particularly when ensuring trustworthy behavior in complex systems like fire fighting drones [17]. In ROS callbacks often access or modify shared resources like global variables, sensor data, or state information. Without proper synchronization, multiple callbacks accessing the same resource concurrently can lead to inconsistencies and unpredictable behavior. Mutexes solve this problem by ensuring mutual exclusion only one callback can access the protected resource. Furthermore, designing state transitions in the state machine with guards that check for synchronization conditions can also be useful. Alternatively for more complex scenarios with numerous callbacks, considering using time stamps in messages [15], synchronization policy [16], services or actions for complex operations can help to solve this issue.

VI. Hardware in Loop Flight Test

Hardware in Loop (HIL) testing was used to provide a controlled environment to identify and address potential issues or errors in the ROS architecture before engaging in real-world flight operations. It reduces the likelihood of costly failures during real-world tests. Data is logged during the HIL mission. This includes logging the drone's actual position, orientation, and the executed commands. The recorded data is compared with the expected trajectory to ensure alignment with the predefined waypoints.

Figure 4 illustrates the Hardware-in-the-Loop (HIL) testing of a ROS software, with a drone actively searching for fire using waypoints depicted on the right side. On the left side of the figure, recorded plots are displayed, and upon comparison with the anticipated trajectory,it shows that the drone is precisely



```
Roll:        -0.0005
Pitch:        0.0002
Yaw:         -2.0118
WorldX:     -10.5770
WorldY:      38.8035
WorldZ:       9.9994
Latitude:    53.8091
Longitude:   -1.5557
VelocityX:   -0.0149
VelocityY:    0.0161
VelocityZ:   -0.0023
AccX:         0.0000
AccY:         0.0000
AccZ:        -0.9998
GyroX:        0.0010
GyroY:       -0.0002
GyroZ:        0.9870
```

Fig. 4. Hardware of Testing of ROS Software Architecture

moving along the designated path, adhering to the predefined waypoints. This is crucial for tasks like searching for fire, as it ensures that the drone covers the intended area systematically. Furthermore, accurate waypoints following implies that the ROS software, in conjunction with the hardware, is capable of interpreting and executing the navigation commands accurately, such as detecting and responding to a fire in the specified locations.

VII. Conclusion

In conclusion, converting a RoboChart model into a ROS top-level software architecture presents substantial advantages in the development and verification of robotic systems. This integration addresses platform-dependency challenges, offering a clear and high-level specification aligned with ROS for improved clarity in specifications and testing. Furthermore, it ensures consistency between formal specifications and actual implementation, minimizing the risk of misinterpretation and enhancing the accuracy of the robotic system representation. The accurate translation also has the potential to elevate the verification process, fostering trustworthiness by identifying and rectifying issues early in the development cycle. In summary, this integrated approach not only overcomes challenges associated with platform dependency but also establishes a robust foundation for developing trustworthy and resilient robotic systems throughout the entire development lifecycle.

Acknowledgment

This work was supported by the Engineering and Physical Sciences Research Council on the UKRI Trustworthy Autonomous Systems Hub programme.

References

[1] S. M. S. Mohd Daud et al., “Applications of drone in disaster management: A scoping review,” Science and Justice, vol. 62, no. 1, pp. 30–42, Jan. 2022, doi: https://doi.org/10.1016/j.scijus.2021.11.002.

[2] [1] P. Petrides, P. Kolios, C. Kyrkou, T. Theocharides, and C. Panayiotou, “Disaster Prevention and Emergency Response Using Unmanned Aerial Systems,” Progress in IS, pp. 379–403, 2017, doi: https://doi.org/10.1007/978-3-319-54558-5-18.

[3] R. M. Carrillo-Larco, M. Moscoso-Porras, A. Taype-Rondan, A. Ruiz-Alejos, and A. Bernabe-Ortiz, “The use of unmanned aerial vehicles for health purposes: a systematic review of experimental studies,” Global Health, Epidemiology and Genomics, vol. 3, p. e13, 2018. doi:10.1017/gheg.2018.11

[4] [3] Mahdi Jemmali, L. Kayed, Wadii Boulila, Hajer Amdouni, and M. T. Alharbi, “Optimizing Forest Fire Prevention: Intelligent Scheduling Algorithms for Drone-Based Surveillance System,” Procedia Computer Science, vol. 225, pp. 1562–1571, Jan. 2023, doi: https://doi.org/10.1016/j.procs.2023.10.145.

[5] R. Nouacer, H. Espinoza Ortiz, Y. Ouhammou and R. Castiñeira González, ”Framework of Key Enabling Technologies for Safe and Autonomous Drones’ Applications,” 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, 2019, pp. 420-427, doi: 10.1109/DSD.2019.00067.

[6] W. Li, A. Miyazawa, P. Ribeiro, A. Cavalcanti, J. Woodcock, and J. Timmis, “From Formalised State Machines to Implementations of Robotic Controllers,” Springer proceedings in advanced robotics, pp. 517–529, Jan. 2018, doi: https://doi.org/10.1007/978-3-319-73008-0.36.

[7] L. Apvrille, P. de Saqui-Sannes and R. Vingerhoeds, ”An Educational Case Study of Using SysML and TTool for Unmanned Aerial Vehicles Design,” in IEEE Journal on Miniaturization for Air and Space Systems, vol. 1, no. 2, pp. 117-129, Sept. 2020, doi: 10.1109/JMASS.2020.3013325.

[8] L. Gomes and A. Costa, ”Model-driven development in hardware-software co-design of controllers for cyber-physical systems,” 2023 IEEE 10th International Conference on E-Learning in Industrial Electronics (ICELIE), Singapore, 2023, pp. 1-6, doi: 10.1109/ICELIE58531.2023.10313106.

[9] Cinzia Bernardeschi, A. Domenici, Adriano Fagiolini, and M. Palmieri, “Co-simulation and Formal Verification of Co-operative Drone Control With Logic-Based Specifications,” The Computer Journal, vol. 66, no. 2, pp. 295–317, Oct. 2021, doi: https://doi.org/10.1093/comjnl/bxab161.

[10] Bernardeschi, Cinzia, Andrea Domenici, and Sergio Saponara. ”Formal verification in the loop to enhance verification of safety-critical cyber-physical systems.” Electronic Communications of the EASST 77 (2019).

[11] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock, “RoboChart: modelling and verification of the functional behaviour of robotic applications,” Software and Systems Modeling, vol. 18, no. 5, pp. 3097–3149, Jan. 2019, doi: https://doi.org/10.1007/s10270-018-00710-z.

[12] A. Cavalcanti and R. M. Hierons, ”Challenges in testing of cyclic systems,” 2023 27th International Conference on Engineering of Complex Computer Systems (ICECCS), Toulouse, France, 2023, pp. 1-6, doi: 10.1109/ICECCS59891.2023.00010.

[13] Ana Cavalcanti. RoboStar modelling stack: tackling the reality gap. In Proceedings of the 1st International Workshop on Verification of Autonomous and Robotic Systems 2021. Association for Computing Machinery, New York, NY, USA, Article 2, 1–3. https://doi.org/10.1145/3459086.3459628

[14] “Robot operating system,” ROS, http://www.ros.org/ (accessed Feb. 1, 2024).

[15] “Trustworthy autonomous systems verifiability node,” Verifiability Node, https://verifiability.org/ (accessed Feb. 4, 2024).

[16] K. Anwar, I. K. Wibowo, B. S. B. Dewantara, M. M. Bachtiar and M. A. Haq, ”ROS Based Multi-Data Sensors Synchronization for Robot Soccer ERSOW,” 2021 International Electronics Symposium (IES), Surabaya, Indonesia, 2021, pp. 167-172, doi: 10.1109/IES53407.2021.9594029.

[17] Randolph, Charles. ”Improving the Predictability of Event Chains in ROS 2.” PhD diss., Delft University of Technology, 2021.