2026 IEEE/SICE International Symposium on System Integration (SII)
January 11-14, 2026. Cancún, México

A ROS-Based Hardware Abstraction Layer for Multifunction End-Effector Integration*

Victor R. Passos[1] and Wesley R. de Oliveira[2]

Abstract— **This paper brings the architecture for a ROS-based Hardware Abstraction Layer (HAL) to interface with the devices of a mechatronic end-effector. A HAL is a middleware between an application and the hardware, responsible for abstracting low-level operations for the client applications. A key feature of this approach is the scalability and modularity delivered by the ROS framework. To illustrate this, the paper presents a case study related to the modeling and integration of a critical-safety gripping function of the given robotic end-effector. This function is modelled and simulated using the timed automata formalism in UPPAAL, which is then implemented within the HAL using the proposed architecture. The system's HAL controller is implemented and tested directly on the Toradex Verdin hardware. The results highlight how this design approach enables the early verification of critical mechatronic functions in robotic systems.**

I. Introduction

The integration of sensors and actuators in a mechatronic system — particularly in multifunctional robotic end-effectors — becomes increasingly complex as the number and variety of components grow. Each component may convey distinct signal types and communication protocols, so encapsulating the hardware interfaces through abstraction layers streamlines the development of supervisory applications for the end-effectors [1] [2].

This study tackles the systems-integration challenge for a multifunctional robotic end-effector specifically designed to manipulate aircraft cabin interior components during the final assembly process [3]. While the detailed design of the mechatronic device itself is not the primary focus of this work, its key features define the requirements for the software entities, which are the main aspects considered herein.

The end-effector is composed of a set of pneumatic actuators for object manipulation, including two sets of vacuum cups and one linear actuator, as detailed in the schematic in Fig. 1. Integrated sensors provide pressure feedback from the pneumatic functions, ensuring a secure grip — a critical safety function that must preserve object integrity. Furthermore, position sensors allow for safe interactions within the constrained assembly environment.

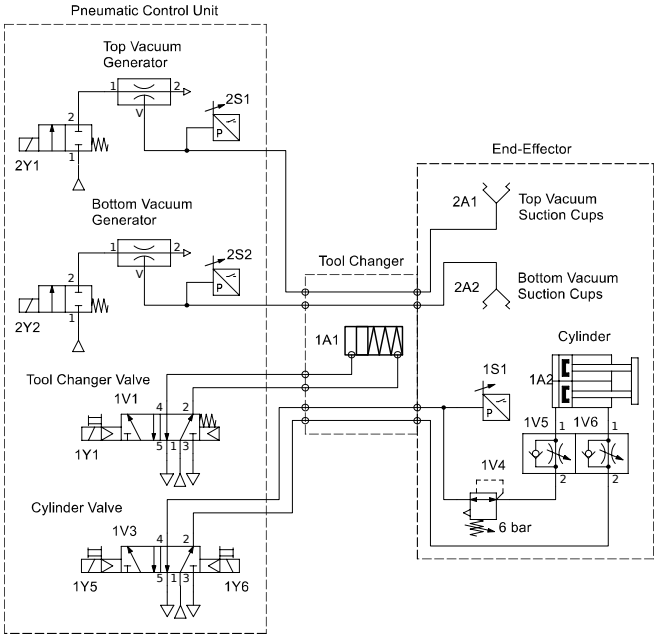

Fig. 1. Pneumatic schematic of the end-effector. The diagram highlights the key valves and actuators relevant to the implementation discussed in this work.

To ensure seamless communication between a high-level control system and the end-effector's hardware, this research defines a Hardware Abstraction Layer (HAL) architecture that acts as middleware, consolidating sensor data and standardizing actuator commands to optimize application-level operations. A HAL manages configuration, communication, message composition, signal validation, resource allocation, and safety interlocks [4], providing a secure, high-level interface to the supervisory system.

This HAL can be implemented using dedicated firmware and hardware, but an alternative approach is to leverage widely adopted middleware with the necessary features [5]. The Robot Operating System (ROS) is an open-source framework [6] that provides extensive libraries and packages for integrating sensors and mechatronic devices [7]. Its modular architecture accelerates development, simplifies maintenance, and enhances scalability [8]. Due to its inherent abstraction capabilities, ROS is often used as a layer interfacing with hardware devices in robotics. However, it's not uncommon to implement an HAL using the ROS framework to abstract end-effectors in general [9].

The objective of this work is to demonstrate the archi-

*Acknowledgements to the São Paulo Research Foundation (FAPESP), by means of FLYMOV (Process No. 2021/11258-5) and the Brazilian Financier of Studies and Projects (FINEP), by means of the Project CAPTAER III (No. 01.22.0313.00).

[1]Victor R. Passos is with Competence Center in Manufacturing, ITA – Aeronautics Institute of Technology, Praça Marechal Eduardo Gomes, 50, ZIP 12228-900 São José dos Campos, São Paulo, Brazil victorpassos@ita.br

[2]Wesley R. de Oliveira is with Competence Center in Manufacturing, ITA – Aeronautics Institute of Technology, Praça Marechal Eduardo Gomes, 50, ZIP 12228-900 São José dos Campos, São Paulo, Brazil wesleyro@ita.br

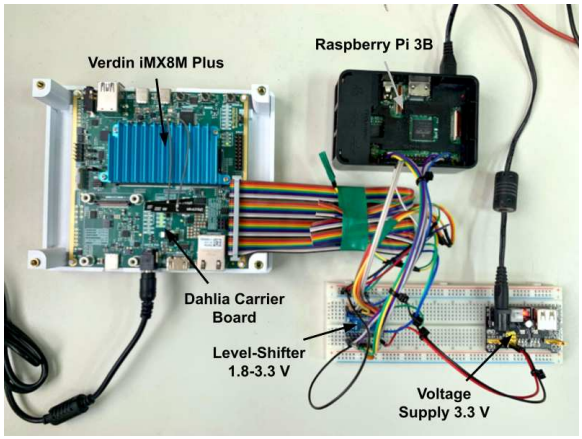

Fig. 2. Hardware testbench showing the connection between the Toradex Verdin and the Raspberry Pi.

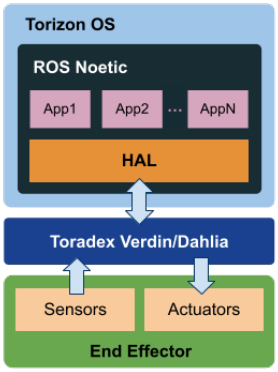

Fig. 3. System architecture depicting the HAL's role in abstracting end-effector hardware (sensors and actuators) from ROS applications running on Torizon OS via the Toradex Verdin platform.

tecture of a Hardware Abstraction Layer (HAL) in ROS for controlling a given multifunctional mechatronic end-effector, managing sensors and actuators, and performing a full safety-critical griping function.

II. Materials and Methods

This section details the hardware and software resources used for the HAL design, centered on a testbench environment defined over Toradex hardware.

A. Hardware Resources: Testbench for Software Development

The mechatronic device is controlled by a Toradex Verdin iMX8M Plus module mounted on a Dahlia Carrier Board. This hardware setup runs Torizon OS, an open-source industrial Linux distribution. To facilitate software development prior to the physical assembly of the end-effector, a dummy circuitry testbench was established around the main controller, as depicted in Fig. 2. The Dahlia Carrier Board provides the necessary I/O interfaces (GPIO, UART, SPI) to connect with external sensors and actuators [10].

Software maintenance and development utilize a Docker container approach [11] on top of Torizon OS. Specifically, the ROS Noetic Ninjemys environment [12] is executed within a container. This ROS version was selected to ensure driver compatibility with a specific end-effector component. Leveraging ROS modularity [7], software entities are programmed in Python 3.8, utilizing community-maintained libraries for hardware abstraction [13], [14].

B. HAL Development Approach

The HAL architecture was designed by mapping the control requirements based on the end-effector functions introduced in Section I. Sensing and actuation functions are converted into ROS nodes, establishing the necessary communication interfaces. A distinct step in this methodology involves modeling the safety-critical gripping function using UPPAAL [15]. UPPAAL is an integrated environment for modeling, validation, and verification of real-time systems using timed automata. This framework allows for simulating events and model checking, ensuring the end-effector's behavior aligns with safety requirements during the design phase.

Experimental validation is performed on the testbench shown in Fig. 2. A Raspberry Pi 3B runs a dedicated Python script to simulate the end-effector hardware: it reads GPIO signals to emulate actuator activation (e.g., solenoids) and responds via UART to simulate sensor feedback (e.g., pressure sensors). This setup verifies the discrete controller model and HAL communication, including failure handling, ensuring full integration with the external application as illustrated in Fig. 3.

III. Implementation of a ROS-Based HAL Architecture

The HAL operates as a primary ROS node that encapsulates low-level interactions with the end-effector circuitry, such as sensor readings and actuator control. This abstraction layer serves as the sole interface for high-level applications, as depicted in the hierarchical view in Fig. 3.

A. HAL Entities: ROS Nodes and Topics

The architecture comprises five nodes (Fig. 4). The primary node, `HALMain`, manages external communication via two action pairs: `/LockToolChanger`//`UnlockToolChanger` for the tool changer, and `/TriggerGrip`//`Drop` for object manipulation. Internally, three nodes handle hardware communication: `Grip` manages the gripping sequence; `ToolChange` controls the manipulator locking mechanism; and `Pressure` processes sensor signals. A fourth internal node, `Position`, was designed for position feedback but was not implemented in this scope.

Regarding data flow, `HALMain` subscribes to sensor topics from `Position` and `Pressure` to publish the aggregated `/EndEffectorState`. For actuation, `ToolChange` hosts the `/SetToolChanger` service, while `Grip` manages `/SetBottomVacuum`, `/SetCylinder`, and `/SetTopVacuum` for individual actuator control.

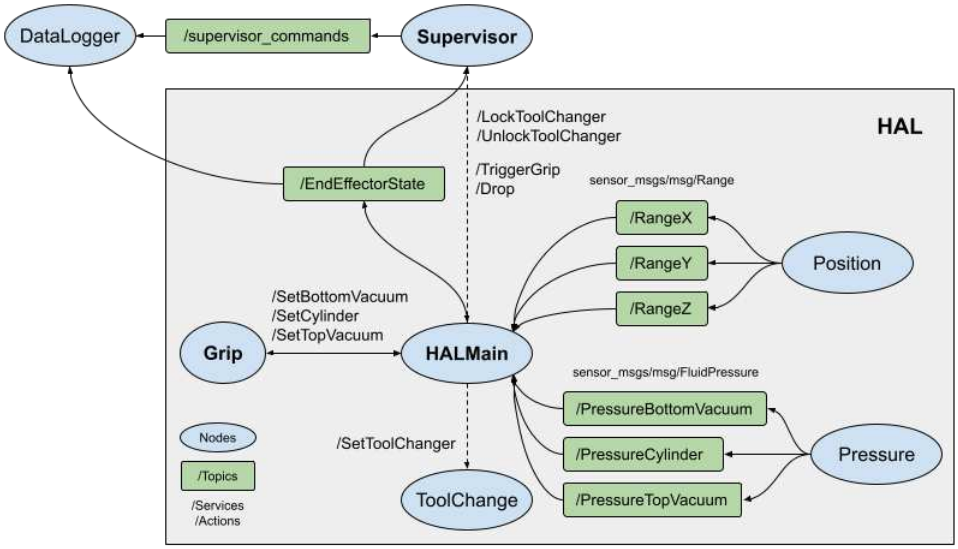

Fig. 4. HAL nodes with their topics, services, and actions. The dashed line indicates the auxiliary node for plotting and its topics.

TABLE I

PIN MAPPING FOR THE HAL FUNCTIONS.

Node	ROS Entity	Communication	Dahlia X20 Header Pin	Signal
Pressure	/PressureVacuumBottom /PressureCylinder /PressureVacuumTop	Topic Topic Topic	12, 13	UART
ToolChanger	/SetToolChanger	Service	27	GPIO
Grip	/SetBottomVacuum /SetCylinder /SetTopVacuum	Service Service Service	28 29 30	GPIO GPIO GPIO

B. Signal Abstraction: Pin-Mapping to Messages

Actuation relies on two Toradex Verdin interfaces: GPIO pins for directly driving valve solenoids and vacuum generators, and a UART connection for receiving sensor feedback. In this work's validation scope, sensor signals are simulated and transmitted via UART by the Raspberry Pi testbench. Table I maps these physical resources to their corresponding ROS entities.

C. Safety-Critical Gripping Function Model

The gripping function is modeled using the automata framework defined in Section II (Fig. 5). It composes four automata: one controller and three representing actuated devices. Actuator behaviors include `OFF` (or `RETRACTED`), `ON` (or `EXTENDED`), and a transition state `PENDING_ON` (`PENDING_EXT`).

The controller automaton oscillates between `STANDBY` and `GRIPPING` states. Upon a grip command, it triggers actuators and awaits sensor feedback. A 5-second timeout guard monitors this process: if a sensor fails to respond, the actuator resets (off/on cycle). After three failed attempts, the sequence aborts, and all previously triggered actuators are deactivated in reverse order.

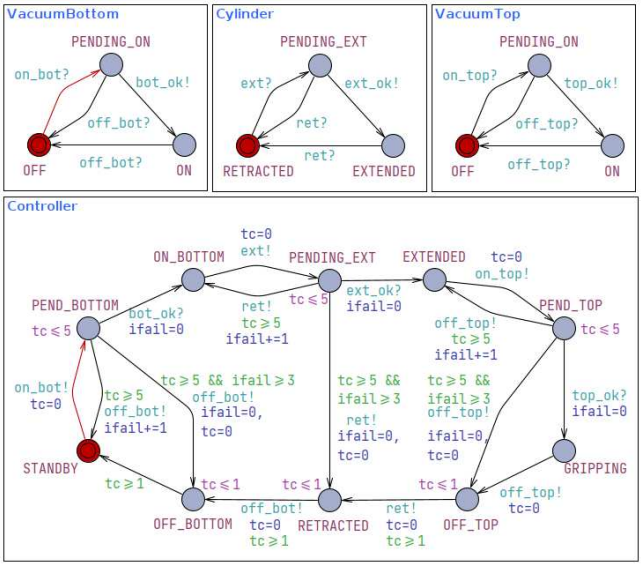

Fig. 5. UPPAAL automaton model for the gripping function. In this notation, circles represent states and arrows represent transitions (events). Channel variables suffixed with `!` denote an event call, while those with `?` denote a response. The variable `tc` represents the system clock, and `ifail` serves as a counter for actuation failures.

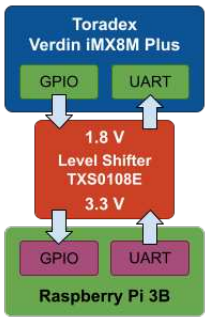

Fig. 6. Diagram of the communication interfaces for the simulation. It shows the data transfer from the Toradex to the Raspberry Pi via GPIO, and the data transfer from the Raspberry Pi to the Toradex via UART.

IV. Case Study - HAL Embedded Gripping Function

The HAL nodes were implemented based on the automaton idealized for the controller. The gripping function, which is critical for the HAL, with its sequence of actuations, feedbacks, and timeouts in case of failure, was created and tested. This section describes how the verification experiment was planned and developed, and the simulation results.

A. Simulation Setup

The primary objective of this experiment is to verify the suitability of the proposed ROS architecture for the end-effector actuation. A secondary, yet equally critical objective, is to validate that the gripping function adheres to the idealized behavior modeled in the automata. To achieve this, the experimental validation utilizes the testbench described in Section II. In this specific configuration, the simulation script on the Raspberry Pi is initialized to emulate distinct operational scenarios. Crucially, this setup mirrors the physical system's data topology: GPIO signals drive the actuation flow, while the UART interface streams the sensor feedback flow. Fig. 6 depicts the resulting data flow for these experiments.

The script running on the Raspberry is capable of simulating three scenarios. In the first scenario, the sensors for each actuator function normally. The objective here is to simulate the proper functioning of the end-effector. In the second scenario, a sensor is adjusted to not function correctly, simulating that the actuation of that actuator failed for some reason. The third scenario simulates a permanent actuator failure, where the sensor will never respond.

The objective of these three simulation scenarios is to observe if the controller for the gripping function responds correctly. In the first scenario, it responds sequentially during the gripping movement and releases sequentially during the releasing movement. In the second scenario, it should receive a timeout, turn off the problematic actuator and re-actuate, succeeding in this subsequent attempt. In the third scenario, the problematic actuator will receive three timeouts and then the controller aborts the gripping function, turning off the actuators in sequence.

The implementation was carried out with the main node (`HALMain`) receiving commands from the Supervisor node. To capture experimental data, a dedicated auxiliary node named `DataLogger` was developed. This node subscribes to the `/EndEffectorState` and `/supervisor_commands` topics, logging the numerical states of the end-effector, supervisor, and actuators directly into a CSV file. Table II details the integer mapping used for these states. Finally, the recorded datasets were processed and visualized using a custom MATLAB script.

B. Results

In the presented curves, the operational state of each element is represented by the integer values defined in Table II. The results for the successful gripping scenario are illustrated in Fig. 7. To avoid visual overlap among the signals, a vertical offset was applied to each curve. Consequently, while the vertical axis provides signal separation, the specific state of each component is identified by text annotations directly on the transitions to facilitate interpretation.

In the first plot (Fig. 7), the nominal gripping scenario is simulated. As observed, the sequence initiates when the `Command State` transitions to `Grip` (integer value 2). Consequently, the actuators first move to the `Pending On` state (1). Upon receiving positive feedback from the sensors, each actuator advances to the `On` state (2), triggering the subsequent device in the chain.

In Fig. 8, the failure and subsequent recovery of the second actuator are simulated. The plot shows the actuator remaining in the `Pending On` state until the 5-second timeout occurs. The system then performs a reset cycle—switching the actuator to `Off` and then back on after 1 second. Upon this second attempt, the correct sensor response is detected, allowing the gripping sequence to complete and the end-effector to reach the `Gripped` state (3).

The third plot (Fig. 9) illustrates the permanent failure scenario, simulated by a non-responsive cylinder sensor. As observed, the cylinder remains in the `Pending Extend` state until the timeout limit is reached. Following each timeout, the system executes a reset cycle, turning the actuator off and then retrying. Upon the occurrence of the third consecutive failure, the operation is definitively aborted. Consequently, the command transitions to `Release`, triggering the sequential deactivation of the actuators to the `Off` state. Finally, the end-effector returns to the `Stand-By` state.

These results demonstrate that the controller successfully implements the UPPAAL model, responding correctly to the simulated success and failure scenarios. The ROS-based architecture proved sufficient, validating the HAL's ability to abstract hardware and execute high-level commands from the application layer (Supervisor node). Furthermore, the experiments confirm that the HAL correctly publishes all internal actuator states and the overall end-effector state, which was essential for logging and analysis.

V. Conclusion

This paper proposed a HAL architecture implemented in ROS Noetic to abstract the hardware devices of a

TABLE II

STATES FOR THE END-EFFECTOR, COMMAND AND ACTUATORS.

Data	End-Effector	Command	Vacuum	Cylinder
-1	-	NONE	-	-
0	OFF	LOCK	OFF	RETRACTED
1	GRIPPING	UNLOCK	PENDING ON	PENDING EXTEND
2	GRIP ERROR	GRIP	ON	EXTENDED
3	GRIPPED	RELEASE	-	-
4	REALEASING	CANCEL	-	-
5	RELEASE ERROR	QUIT	-	-
6	STAND-BY	-	-	-

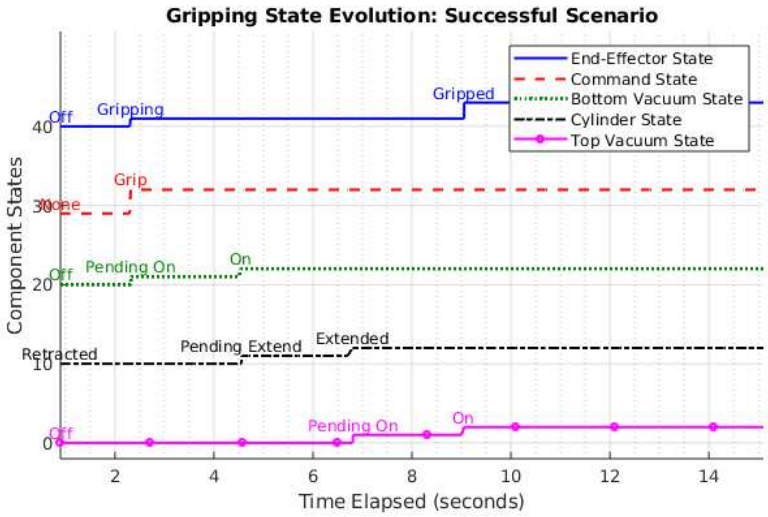

Fig. 7. Time-series analysis of the component states during a successful grip. The specific states are identified by text annotations and correspond to the definitions in Table II.

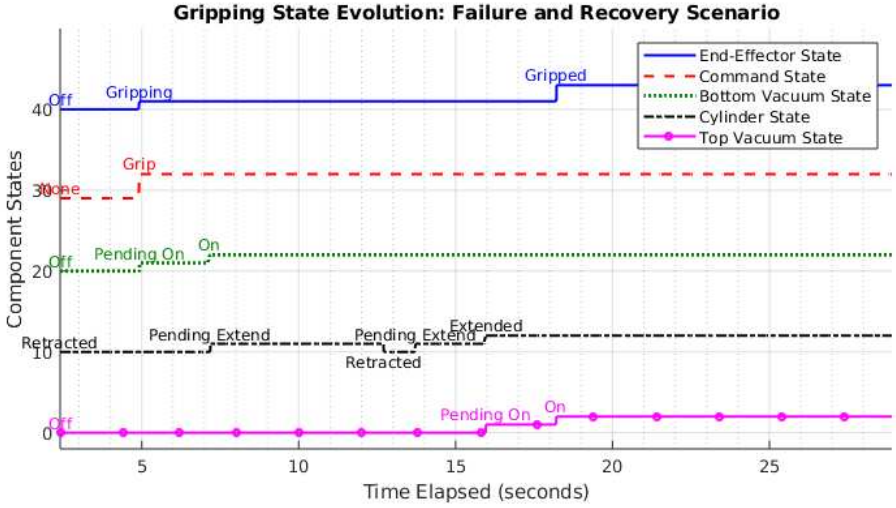

Fig. 8. Gripping state evolution: Failure and Recovery scenario. The plots illustrate the system's response to an initial actuation failure, followed by a timeout, a reset cycle, and a successful recovery sequence.

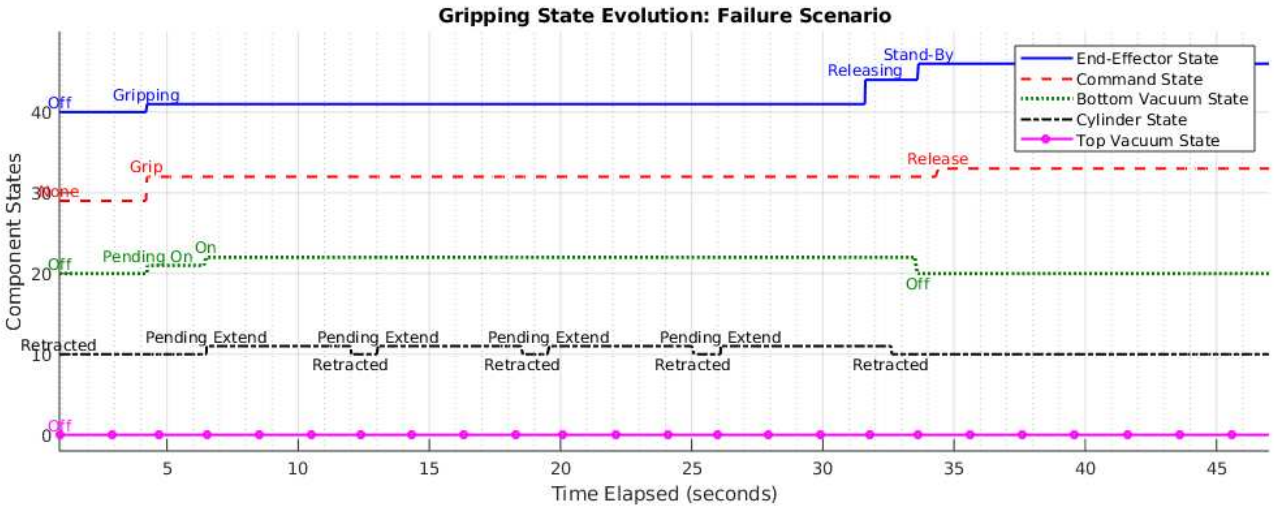

Fig. 9. Gripping state evolution: Permanent failure scenario. The plots illustrate the system's behavior when a sensor remains non-responsive.

mechatronic end-effector. The architecture's suitability and abstraction capabilities were validated through simulations covering nominal, recoverable-failure, and permanent-failure scenarios. Results based on the HAL's state publications demonstrate the system's robustness in handling critical gripping functions. Furthermore, the structure of dedicated hardware nodes successfully isolated the physical layer from the application, ensuring both scalability and modularity.

REFERENCES

[1] G. F. Rigano, L. Muratore, A. Laurenzi, E. M. Hoffman, and N. G. Tsagarakis, "Towards a Robot Hardware Abstraction Layer (R-HAL) Leveraging the XBot Software Framework," in *Proc. of 2nd IEEE International Conference on Robotic Computing, IRC 2018*, pp. 175-176, 2018.

[2] Yves Bergeon, Václav Krivánek, and Jean Motsch, "Raspberry pi as an interface for a hardware abstraction layer structure of software and extension of the TurtleBot 2 - Kobuki protocol," in *ICMT 2019 - 7th International Conference on Military Technologies, Proceedings*, pp. –, 2019

[3] Kiara Ottogalli, Daniel Rosquete, Javier Rojo, Aiert Amundarain, Jos'e Mar'ia Rodr'iguez, and Diego Borro, "Virtual reality simulation of human-robot coexistence for an aircraft final assembly line: process evaluation and ergonomics assessment," *International Journal of Computer Integrated Manufacturing*, vol. 34, no. 9, pp. 975–995, 2021.

[4] Stefan Jorg, Jan Tully, and Alin Albu-Schaffer, "The Hardware Abstraction Layer - Supporting control design by tackling the complexity of humanoid robot hardware," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 6427-6433, 2014

[5] Ricardo C. Mello, Sergio D. Sierra M., Wandercleyson M. Scheidegger, Marcela C. Múnera, Carlos A. Cifuentes, Moises R.N. Ribeiro, and Anselmo Frizera-Neto, "The PoundCloud framework for ROS-based cloud robotics: Case studies on autonomous navigation and human–robot interaction," *Robotics and Autonomous Systems*, vol. 150, pp. 103981, 2022.

[6] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *Proc. of 2009 IEEE ICRA Workshop on Open Source Robotics*, Kobe, Japan, 2009.

[7] Y. Bergeon and V. Krivanek, "Design of an open hardware bridge between robots and ROS/non-ROS environments," *Advances in Electrical and Electronic Engineering*, vol. 16, no. 2, pp. 199-204, 2018.

[8] L. Muratore, A. Laurenzi, E. M. Hoffman, and N. G. Tsagarakis, "The XBot Real-Time Software Framework for Robotics: From the Developer to the User Perspective," *IEEE Robotics and Automation Magazine*, vol. 27, pp. 133-143, 2020.

[9] Davide Torielli, Liana Bertoni, Fabio Fusaro, Nikos Tsagarakis, and Luca Muratore, "ROS End-Effector: A Hardware-Agnostic Software and Control Framework for Robotic End-Effectors," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 108, no. 4, pp. 1-18, 2023.

[10] Toradex, "Verdin iMX8M Plus System on Module," [Online]. Available: `https://www.toradex.com/products/system-on-modules/verdin-imx8m-plus`, (2023).

[11] Docker, "Docker: The World's Leading Software Container Platform," [Online]. Available: `https://www.docker.com/`, (2008).

[12] Open Robotics, "Robot Operating System (ROS) Noetic Ninjemys," [Online]. Available: `https://www.ros.org`, (2020).

[13] E. Cervera and A. P. Del Pobil, "ROSLab: Sharing ROS Code Interactively with Docker and JupyterLab," *IEEE Robotics & Automation Magazine*, vol. 26, no. 3, pp. 64-69, 2019.

[14] S. Pütz, T. Wiemann, and J. Hertzberg, "The Mesh Tools Package - Introducing Annotated 3D Triangle Maps in ROS," *Robotics and Autonomous Systems*, vol. 138, pp. 103688, 2021.

[15] UPPAAL, "UPPAAL Tool," [Online]. Available: `https://uppaal.org/`, (2023).