Simulation and optimisation of a submarine ROV with AADL

Ivan Kovačević+, Jure Antunović+, Tonko Kovačević+, Laurent Lemarchand*, Frank Singhoff*

+University of Split, Croatia; email: kovacevic.ivan0120@gmail.com, jure.antunovic6@gmail.com, tkovacev@oss.unist.hr

**Lab-STICC UMR 6285 - University of Brest, France; email: {lemarch,singhoff}@univ-brest.fr*

Abstract

Keywords: template, journal, Ada.

1 Introduction

When designing underwater robots, reliable simulation is essential to master the complexity and cost of physical trials. Most existing simulations rely on ad-hoc code and are rarely integrated with design methods.

This project bridges that gap by combining model-driven engineering (MDE) techniques with runtime environments like Gazebo and ArduSub SITL. We use the Architecture Analysis & Design Language (AADL) to model the real-time structure and behavior of a Remotely Operated Vehicle (ROV), and then generate implementation code using Ocarina [8].

This code is connected to a Gazebo simulation through a C++ plugin, enabling verification of sensor data, actuator responses, and mission performance in a virtual underwater environment.

2 Background on AADL

AADL (Architecture Analysis & Design Language) is a standardized modeling language developed by SAE (AS5506), used for specifying the software and hardware architecture of embedded real-time systems.

It enables developers to describe components like devices, processors, buses, and their interactions, including timing constraints and behavioral properties.

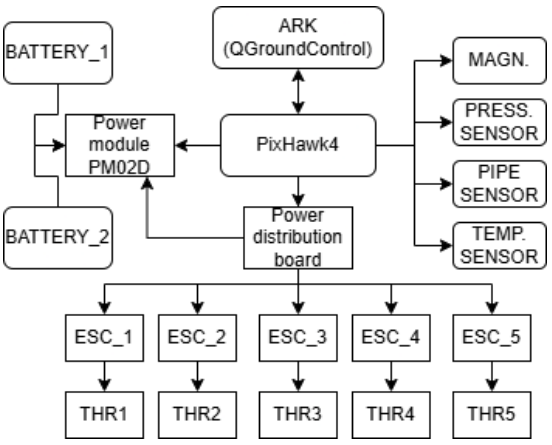
Tools such as OSATE and AADLInspector support modeling, analysis, and verification. For this project, AADLInspector was used for model validation, and Ocarina was used to generate C code for execution.

Through AADL, the ROV's architecture—including sensors, actuators, and control units—was modularly specified and enriched with realistic dispatch protocols, allowing accurate simulation of periodic threads and system scheduling behavior in a real-time context.

3 ROV AADL modeling

3.1 ROV description

A concise description of the ROV would define it as an underwater vehicle propelled by five thrusters, designed to access areas that were previously difficult to reach in submerged environments. The ROV is powered by two Bosch batteries and equipped with a range of sensors used for monitoring both environmental conditions and the vehicle's own performance. An overview of the ROV's general design is shown in Figure 1.



Picture 1: ROV model

3.2 AADL model of the ROV

The ROV modeled using AADL represents a modular, real-time system architecture designed for underwater simulation in environments such as PX4 SITL, Gazebo, and QGroundControl. It mirrors the structural organization and operational behavior of a real-world Unmanned Underwater Vehicle (UUV).

The model is organized into three interdependent layers: sensing, processing, and actuation. The sensing layer includes a magnetometer and a pressure sensor, providing orientation and depth data, respectively. These are represented as AADL device components with data output ports, sampled at realistic update rates (e.g., 100 Hz for the magnetometer).

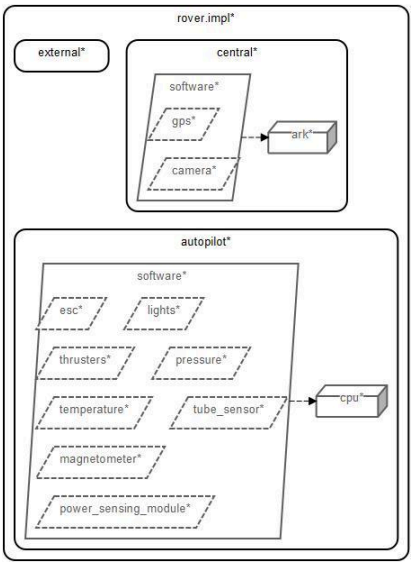
The processing layer consists of a central processing unit defined as a process that hosts a dedicated controller process, which includes a stabilization_thread. This thread receives sensor inputs and computes control signals for the actuators.

The actuation layer receives data of the type ControlCmd, which is defined as a structured data type. In a complete implementation, this data would be used to control multiple thrusters. This design supports a model-based approach, enabling seamless integration into both simulation and real-world deployment environments.

Each key component - controllers, CPU, sensors, and data types - is encapsulated within its own dedicated AADL file, fostering modularity and facilitating reuse across the system. Additionally, the model is fully compatible with the Ocarina toolchain, which translates the AADL architecture into C source code. This integration ensures deterministic behavior throughout the simulation loop.

Figure 2 below shows the general structure of the simulation, without including all the individual

sensors and actuators.



Picture 2: AADL model of ROV structure

4 Simulation with Gazebo

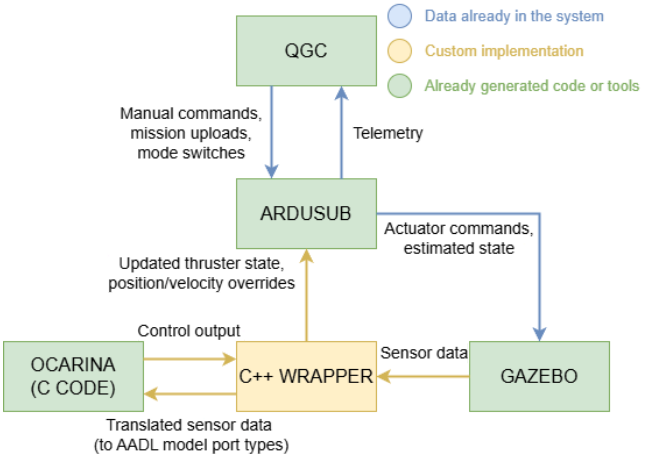
The AADL-modeled ROV system follows a model-driven development process integrated with real-time simulation. The system architecture—comprising sensors, actuators, and control logic—is specified in AADL and compiled into C code using **Ocarina**. This code represents the functional behavior and software structure of the ROV.

To interface with the simulation environment, the generated C code is wrapped in a **C++ wrapper**, allowing it to be embedded into a custom **Gazebo plugin**. This plugin runs within Gazebos simulation loop and provides access to simulated sensor data and actuator control.

In Gazebo, the ROV is simulated with realistic underwater physics and equipped with virtual sensors like GPS, IMU, etc. As the simulation runs, sensor inputs are passed to the plugin, which executes the control logic and returns actuator commands—thus creating a real-time feedback loop.

The system is further integrated with **ArduSub SITL**, which interprets the plugin's outputs and communicates with **QGroundControl (QGC)** over a MAVLink. This setup allows developers to monitor missions, receive telemetry, and control the ROV from QGC as if it were a physical vehicle.

This workflow bridges formal modeling and simulation, enabling early validation, traceability, and consistent deployment behavior. The workflow can be seen in Figure 3:



Picture 3: Logic behind Gazebo plugin

Here is the explanation of the data flow in the graph:

First the sensor data is generated by the simulated environment and passed to the wrapper. **(Sensor data)**

Secondly the wrapper converts Gazebo sensor data into a format matching AADL port types expected by the Ocarina-generated C code. **(Translated sensor data)**

Thirdly, the AADL system model, implemented as C code, computes control outputs which are passed back to the wrapper. **(Control output)**

After that the wrapper sends the control outputs (e.g., desired thrust values, position or velocity corrections) to

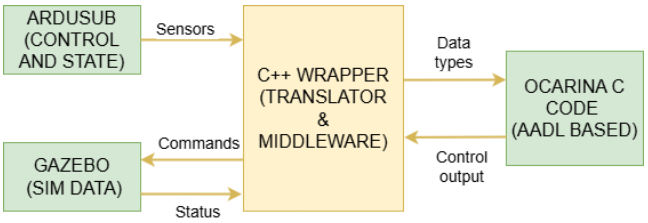
ArduSub as overrides or direct control inputs. **(Updated thruster state, position/velocity overrides)**

ArduSub sends low-level actuator commands to Gazebo's simulated vehicle, along with estimated state outputs (e.g., attitude, velocity) for visualization and synchronization. **(Actuator commands, estimated state)**

ArduSub sends telemetry data (vehicle state, sensor readings, etc.) back to QGC for real-time monitoring and display. **(Telemetry)**

At last, QGC sends user input or mission instructions to ArduSub, including manual joystick commands, pre-defined mission plans, or changes in flight mode. **(Manual commands, mission uploads, mode switches)**

The functionality of the C++ wrapper can be seen underneath in Figure 4:



Picture 4: C++ wrapper logic

The explanation of the data flow in the graph is as follows:

First, ArduSub sends estimated state and control-related sensor data (e.g., velocity, orientation) to the wrapper. **(Sensors)**

Secondly, Gazebo provides simulated sensor data (e.g., GPS, IMU, pressure) to the wrapper. **(Status)**

Then afterwards, the wrapper translates ArduSub and Gazebo data into AADL-typed inputs and passes them to the Ocarina-generated C code. **(Data types)**

The Ocarina C code returns computed control outputs (e.g., thrust or setpoints) to the wrapper. **(Control output)**

At the end the wrapper converts control outputs into simulation-compatible commands and sends them to Gazebo to actuate the simulated vehicle. **(Commands)**

This flow ensures seamless integration between the simulation, the formal model, and the autopilot.

Lastly, it must be said that even though PX4 can be used as far as hardware specifications go, during the simulation inside the Gazebo environment the PX4 has shown several challenges. The biggest challenge is that PX4 SITL doesn't support UUVs and Submarine vehicles. For that reason another SITL was chosen - the ArduSub SITL, used for simulation purposes only.

Maybe in the future an in-depth work can be written on further developing the PX4 SITL that works specifically for UUVs, submarines and other underwater vehicles which are currently unsupported and experimental from PixHawks side.

5 Verification and optimization

Simulations allow to check and measure the behavior of the ROV in a given environment for achieving a targeted mission. Both the internal and external aspects of the ROV mission can be optimized, e.g. scheduling aspects for the embedded tasks, path planning characteristics or mission achievement rate. Alternative versions of the embedded software can be evaluated and alternative mission design (path planning and goals) can be simulated. Metrics such as latency, preemptions can be evaluated for internal aspects. Values reflecting path length, risk, robustness, energy consumption can also be defined and used for evaluating alternatives. All of these architectural and mission design solutions can be explored using Multiple Objective Optimization techniques [5].

Those techniques are often based on Evolutionary algorithms (metaheuristics), leading to MOEA (Multi-Objective Evolutionary Algorithms) such as NSGA-II [6] or PAES [7]. These frameworks have to be customized according to the different metrics used and the design space for both internal and external aspects. They compute a set of trade-offs solutions (a Pareto Set) instead of a single best solution. This set includes solutions for which contradictory objectives are optimized. A solution can be better than another on one aspect (e.g. energy or distance for this mission solution) while it is worse on another (e.g targets achievement rate) and conversely (e.g a more costly mission could allow it to process more targets).

The verification and optimization of the ROV system are critical for ensuring both reliable behavior and efficient performance during underwater missions. Given the complexity of the architecture—spanning sensors, actuators, control algorithms, and middleware—simulation plays a central role in validating system correctness before deployment. Developers are encouraged to rely on the full simulation loop using Gazebo, ArduSub SITL, and QGroundControl in conjunction with the AADL-generated C code to evaluate behavior under realistic environmental conditions. This simulation-based workflow enables early identification of anomalies, such as control instability or synchronization delays, and allows safe iterations without risking physical hardware.

Future improvements should focus on the optimization of both internal system behavior and mission-level performance. Internally, timing parameters such as task periods, execution times, and thread priorities can be tuned to minimize CPU load and communication jitter. Metrics like response latency, number of thread preemptions, and CPU utilization can be used to assess the quality of scheduling decisions and control algorithms. Externally, mission-specific parameters such as energy consumption, trajectory smoothness, and task coverage should be evaluated. Using multi-objective optimization

methods—such as NSGA-II or PAES—developers can explore trade-offs between conflicting goals, generating a Pareto front of feasible system configurations. This allows informed decisions about balancing efficiency, robustness, and control performance.

To support future optimization efforts, simulation scenarios should be recorded and reused systematically. By preserving Gazebo world configurations, QGC mission files, and simulation logs, the test environment can be replicated reliably. Additionally, injecting artificial faults or environmental disturbances into simulations will help assess system robustness and fault tolerance. With a model-based design rooted in AADL, future researchers and engineers are well-positioned to apply formal methods and design space exploration techniques to evaluate alternative architectures and optimize system behavior both at the component and mission level.

6 Experiments

The experimental phase aimed to validate the end-to-end functionality of the ROV system, starting from its modular AADL model to execution in a full simulation loop. The system architecture was modeled using AADL, where sensors and actuators were implemented as periodic threads with well-defined data ports and execution properties. Each thread's behavior was mapped to a user_<thread>_entrypoint() function in C, generated using the Ocarina toolchain.

To simulate sensor feedback without hardware, custom functions (e.g., temperature_spg(), gps_spg(), thrusters_spg()) were implemented to generate realistic or randomized data based on timing and physical characteristics. These, along with the Ocarina-generated code, were compiled into a shared .so library using a CMake-based build system. A Gazebo plugin dynamically loaded this library, allowing direct execution of the logic inside a simulated underwater world

The vehicle model in Gazebo responded to control messages sent through ArduSub SITL and MAVProxy. QGroundControl was used to define missions such as moving forward, right, and returning to launch. During mission execution, simulated sensors sent data at realistic frequencies (e.g., 1 Hz for temperature, 50 Hz for IMU), enabling full feedback loops and verification.

Key experimental outcomes confirmed that sensor data were processed and acted upon correctly, actuator logic matched mission expectations, and control feedback loops operated as intended. The modular structure also enabled quick reconfiguration, such as swapping sensor functions or modifying execution periods for testing timing constraints.

This simulation framework proved effective for rapid prototyping, behavioral validation, and mission testing of the ROV without physical hardware. It lays the groundwork for more advanced tasks like fault injection, performance optimization, and real-time validation of future system extensions.

7 Related work

Several related works explore model-based approaches for robotics, with varying emphasis.

Aloui (2024) focuses on multi-robot behavior using SysML and ROS, emphasizing real-world prototyping over physics-based simulation, contrasting with this work's use of SITL and Gazebo.

Steffano (2022) highlights virtual prototyping and its validation against real-world laboratory equipment, aligning with the goal of bridging simulation and physical systems.

Jasmine (2020) presents high-level SysML modeling of autonomous, fault-aware missions, showing how system-level models can drive resilience. Unlike these, our approach uses AADL and Ocarina to directly generate C code, targeting embedded execution in a fully closed-loop underwater simulation.

Compared to our AADL-based approach, ROS 2 offers a flexible middleware-centric framework ideal for runtime communication and distributed control but lacks the formal architectural modeling and static analysis features of AADL. While ROS 2 excels in real-time data exchange and modular integration, our method emphasizes early design validation, schedulability, and verifiable code generation through Ocarina. Combining both could enhance simulation fidelity while preserving system correctness.

8 Conclusion

This project demonstrated a complete model-to-simulation workflow for an underwater ROV system. Beginning with modular AADL architecture, it leveraged Ocarina for C code generation and integrated with Gazebo and ArduSub SITL for realistic visualization and mission execution. QGroundControl and MAVProxy were used for planning, telemetry, and control, closing the feedback loop in a fully simulated environment.

Future work includes adding underwater dynamics like current flow and sensor noise, extending the vehicle model with richer physics, and exploring integration with PX4 SITL once support matures. Continuous integration pipelines and automated testing can enhance reproducibility and development speed.

Overall, the approach demonstrates how architecture-centric modeling, combined with simulation and code generation tools, can yield a robust development workflow for complex cyber-physical systems like ROVs, even in early design phases.

References

[1] K. Alaoui, A. Guizani (2024). An integrated Design Methodology for Swarm Robotics using Model-Based Systems Engineering and Robot Operating System

[2] S. Steffano (2022). A Model-based Approach for Designing Cyber-Physical Production Systems

[3] R. Jasmine , L. Stephanie ´ b , C. Jean-Charles , V. Nicole (2020). MBSE approach applied to lunar surface exploration elements

[4] Hugues, J., Zalila, B., Pautet, L., & Kordon, F. (2008). From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Transactions on Embedded Computing Systems (TECS)*, *7*(4), 1-25.

[5] Coello, C. A. C., Brambila, S. G., Gamboa, J. F., & Tapia, M. G. C. (2021). Multi-objective evolutionary algorithms: past, present, and future. In Black Box Optimization, Machine Learning, and No-Free Lunch Theorems (pp. 137-162). Cham: Springer International Publishing.

[6] Deb, K., Agrawal, S., Pratap, A., & Meyarivan, T. (2000, September). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In International conference on parallel problem solving from nature (pp. 849-858). Berlin, Heidelberg: Springer Berlin Heidelberg.

[7] Knowles, J. D., & Corne, D. W. (2000). Approximating the nondominated front using the Pareto archived evolution strategy. Evolutionary computation, 8(2), 149-172.

[8] Singhoff, F., Legrand, J., Nana, L., & Marcé, L. (2004, November). Cheddar: a flexible real time scheduling framework. In *Proceedings of the 2004 annual ACM SIGAda international conference on Ada.* pp. 1-8. November 2004. Atlanta. USA.

[9]