

From Gazebo to FMI: A Modelica-Based Cyber-Physical Simulation Architecture for Firmware-Integrated Robotic Vehicles

Minhyun Cho¹ Minjoon Lee¹ Micah K. Condie¹ Michael P. Rooney² Evan R. English² James M. Goppert¹ Arslan Khan³ Dongyan Xu² Inseok Hwang¹

¹School of Aeronautics and Astronautics, Purdue University, United States,
{mhcho, lee5826, condie, jgoppert, ihwang}@purdue.edu

²Department of Computer Science, Purdue University, United States, {rooney13, englis20, dxu}@purdue.edu

³School of Electrical Engineering and Computer Science, The Pennsylvania State University, United States,
{arslankhan}@psu.edu

Abstract

Robotic vehicle simulators such as Gazebo, CARLA, JSBSim, and Isaac Sim provide mature physics, rendering, and sensor environments, but simulation workflows built on these tools are often tied to simulator-specific plugins and asynchronous middleware coupling. This paper presents a Modelica/FMI-based cyber-physical simulation architecture for firmware-integrated robotic vehicles. Our physics-aware rehosting engine, COURBET, models vehicle dynamics in Modelica, exports them as Functional Mock-up Units (FMUs) in FMI-3.0 through the Rumoca toolchain, and couples the resulting FMUs to ArduPilot firmware running inside a QEMU-based firmware-in-the-loop execution environment, FastDyn. The framework uses instruction-counted virtual time and a configurable board tick to synchronize firmware execution, actuator commands, sensor feedback, and mission automation. By separating vehicle-generic dynamics from firmware-specific interfaces, missions, ports, and parameter overrides, COURBET enables interchangeable copter, rover, and plane models to run behind the configurable firmware harness. We describe the Rumoca-to-FMU export flow, the firmware-plant coupling architecture, and simulation workflows for ArduPilot. The proposed approach moves robotic vehicle simulation from simulator-bound plugins toward portable FMI-3.0 artifacts, enabling deterministic execution, controlled fidelity variation, sensor and actuator customization, and scalable cyber-physical testing campaigns.

Keywords: *Modelica, FMI-3.0, Functional Mock-up Unit (FMU), Firmware-in-the-Loop (FITL), Cyber-Physical Systems, Robotic Vehicle Simulation, ArduPilot*

1 Introduction

Open-source Robotic vehicle simulators such as Gazebo (Koenig and Howard 2004), CARLA (Dosovitskiy et al. 2017), JSBSim (Berndt 2004), AirSim (Shah et al. 2018) and Isaac Sim (NVIDIA 2023) have been widely adopted commercial and research platforms for developing, testing, and validating autonomous systems. For instance,

Gazebo provides a physics engine with collision detection, rendering, and actuator/sensor models through an open-source robotics simulation stack, while Isaac Sim provides an NVIDIA Omniverse libraries-based multi-physics environment for high rendering/perception fidelity, and synthetic data generation. These tools are particularly effective when the primary objective is to create realistic scenes, render sensor data, evaluate a specific robot behavior in rich environments, or integrate with robotics middleware (Robot Operating System, ROS).

However, firmware-integrated robotic vehicle simulation introduces requirements that are not fully addressed by conventional simulators. In such settings, the simulator is not merely a visualization or a physics engine; it becomes part of a cyber-physical execution loop that must synchronize plant dynamics, sensor feedback, actuator commands, firmware timing and experimental perturbations. In conventional simulators, much of this coupling is implemented through simulator-specific plugins, message interfaces, and process-level orchestration rather than mathematically rigorous and configurable modeling abstractions. Furthermore, extending simulator capability/fidelity often requires additional custom plugins or backend-specific integrations for sensors, actuators, and environmental effects (Pant et al. 2023; Cacace et al. 2020), making the simulation architecture increasingly dependent on simulator internals. The tight coupling makes it difficult to isolate the mathematical plant model from the simulator backend, to replace model fidelity in a controlled manner, or to reuse the same component artifacts across different environments.

The Functional Mock-up Interface (FMI) offers a complementary physics abstraction layer (Junghanns et al. 2021). FMI is an open standard for exchanging dynamic simulation models as Functional Mock-up Units (FMUs), where the model interface, metadata, binaries, and supporting resources are packaged into a portable artifact. FMI supports Model Exchange and Co-Simulation, and FMI-3.0 further introduces interface concepts such as clocks and scheduled execution that can be utilized for cyber-physical and embedded-system simulation. By sep-

arating the mathematical model from the simulator backend, equation-based modeling languages, such as Modelica (Wetter 2009), can provide strong support for hybrid differential-algebraic systems and event-aware modeling in FMI. This enables mode transitions and constraints to be represented declaratively through equations and events, rather than being manually encoded as explicit branching within simulator plugins or middlewares. As a result, more complex cyber-physical behaviors can be modeled in a more mathematically rigorous, modular, and scalable manner.

This paper presents a Modelica/FMI-based simulation architecture for firmware-integrated robotic vehicles. Transitioning a physics substrate from Gazebo in our framework (Rooney et al. 2026) to FMI, we compile Modelica models into FMI-3.0 FMUs through Rumoca (Condie et al. 2025) and execute them inside our firmware-in-the-loop framework (FITL), FastDyn/COURBET. The resulting architecture decouples vehicle dynamics models from simulator-specific backend implementations and firmware orchestration logic. It allows FMU plants with different vehicle configurations and model fidelities to be automatically generated and exchanged, and synchronizes the firmware and plant through board-timer-driven virtual time rather than relying on wall-clock execution with a real-time factor. The current implementation supports Rumoca-generated FMI-3.0 plants for ArduCopter, ArduRover, and ArduPlane configurations (ArduPilot Development Team 2025), with Gazebo retained as legacy backend. The contribution of this paper is threefold. First, we describe a Modelica/FMI-3.0 backend for firmware-integrated robotic vehicle simulation. Second, we show how Rumoca-generated FMUs can be coupled to a QEMU-based firmware execution framework using board-timer-driven virtual-time orchestration. Third, we demonstrate our approach on multiple robotic vehicle classes, including rover, plane, and copter models, and discuss how the architecture supports model-fidelity substitution and scalable campaign execution for validation/testing.

The remainder of the paper is structured as follows. Section 2 presents the overall architecture of the proposed firmware-integrated simulation framework, including the virtual-time execution model and firmware-physics synchronization mechanism. Section 3 describes the Modelica-to-FMU workflow using Rumoca, including FMI-3.0 export and integration into the firmware-in-the-loop environment. Section 4 demonstrates the framework on ArduPilot, highlighting configurable vehicle dynamics. Section 5 discusses current limitations, implementation considerations, and directions for future work in cyber-physical testing and firmware-integrated simulation.

2 Simulation Architecture

The proposed architecture moves the physical simulation layer from a simulator-centric implementation to an FMI-

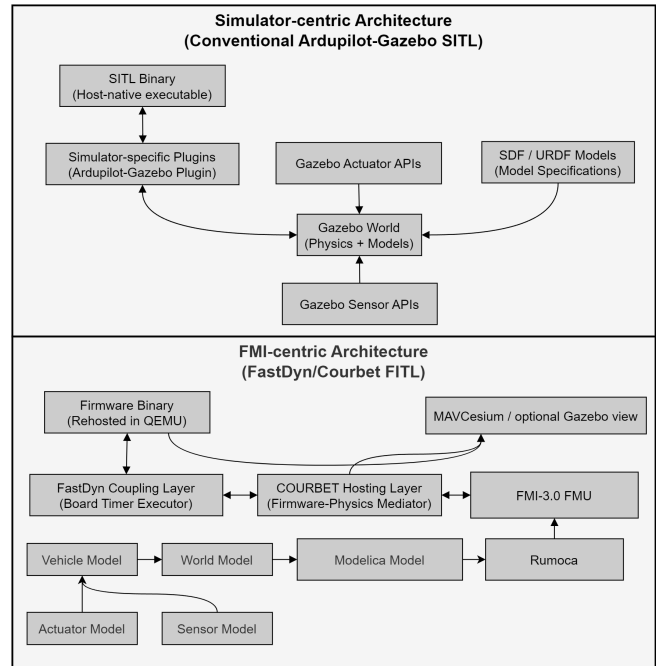


Figure 1. Comparison between a simulator-centric robotic vehicle software-in-the-loop (SITL) and the proposed FMI-centric firmware-in-the-loop (FITL). In the proposed architecture, the physical plant is represented as an FMI-3.0 FMU rather than being embedded in simulator-specific plugins.

centric physics representation. The motivation for this transition is not the absence of physical simulation capability in existing robotic simulators. Simulators widely adopted for robotic vehicle firmware development, such as Gazebo, already provide mature physics engines, rendering capabilities, sensor models, and actuator models. The limitation arises when the simulator becomes the primary location where plant dynamics, sensor behavior, perturbation logic, and firmware interfaces are implemented. In such architectures, the mathematical plant model becomes tightly coupled with simulator-specific backend application programming interfaces (APIs) and plugin implementations. As a result, changing the model fidelity, adding sensor vulnerability models, or replacing vehicle dynamics often requires rewriting simulator plugins rather than exchanging a well-defined component artifact.

Figure 1 illustrates this architectural shift. In the conventional simulator-centric architecture, the vehicle model, actuator interface, sensor interface, and firmware coupling logic are distributed across simulator plugins, Gazebo APIs, and SDF/URDF (Simulation Description Format/Unified Robot Description Format) descriptions. Although this architecture is effective for realistic visualization and robotics middleware integration, the physical plant is not naturally exposed as a portable and replaceable artifact. In contrast, the proposed FMI-centric architecture encapsulates the physical model as an FMI-3.0 FMU. The FMU becomes the authoritative representation of the vehicle dynamics and the physical state from which sensor feedback is generated. The surrounding

FastDyn/COURBET coupling layer handles firmware execution, timing, actuator transfer, sensor injection, and optional visualization. Within this layer, FastDyn orchestrates QEMU to provide a board-level execution environment for running the unmodified firmware binary. Rather than modeling the physical vehicle, QEMU hosts the cyber-side firmware execution, while the coupling interface synchronizes firmware actuator outputs and sensor inputs with the FMU-based physical plant.

The resulting architecture separates the cyber and physical concerns of firmware-integrated simulation. The cyber side consists of the firmware binary, QEMU-based execution environment, board timing model, MAVLink communication, and command handling. The physical side consists of the FMU physics model, actuator response, sensor generation, vehicle dynamics, and disturbance models. The coupling layer maps actuator outputs produced by the firmware into FMU inputs and maps FMU states or sensor outputs back into the firmware. This separation makes it possible to replace the physical model without rewriting the firmware harness, and to modify firmware-specific configurations without changing the underlying vehicle dynamics model.

A key feature of the architecture is that firmware and plant execution are synchronized through board-timer-driven virtual time rather than host wall-clock progression. The firmware runs inside a QEMU environment, while FastDyn maintains an instruction-counted virtual-time model of firmware execution. For fidelity-oriented runs, the framework advances the plant at a configurable board-timer lockstep boundary, such as 1 ms. At each lockstep boundary, the coupling layer reads the most recent actuator commands that the firmware has written to the emulated actuator hardware interface, converts them into FMU inputs, and applies them to the plant. The FMU is then advanced to the next simulation time, and the resulting plant states and sensor outputs are published back to the firmware-facing sensor interfaces. The firmware subsequently observes these updated sensor values through its normal sensor-driver reads. Thus, actuator commands are not generated by the timer tick itself; rather, they are produced asynchronously with respect to the lockstep boundary whenever the firmware control logic writes to the emulated actuator hardware registers. This mechanism allows the firmware, plant, and sensor stream to advance under a common virtual-time semantics, reducing dependence on host scheduling latency and wall-clock real-time factor.

Figure 2 illustrates the execution sequence of the proposed firmware-integrated FMI simulation architecture. The command handler first sends mission items or MAVLink commands to the firmware. QEMU executes a slice of firmware instructions while the FastDyn/QEMU integration layer accumulates instruction-counted virtual time using the QEMU `icount` mechanism. During this firmware execution, the onboard estimator and controller may read sensor-driver values and write actuator com-

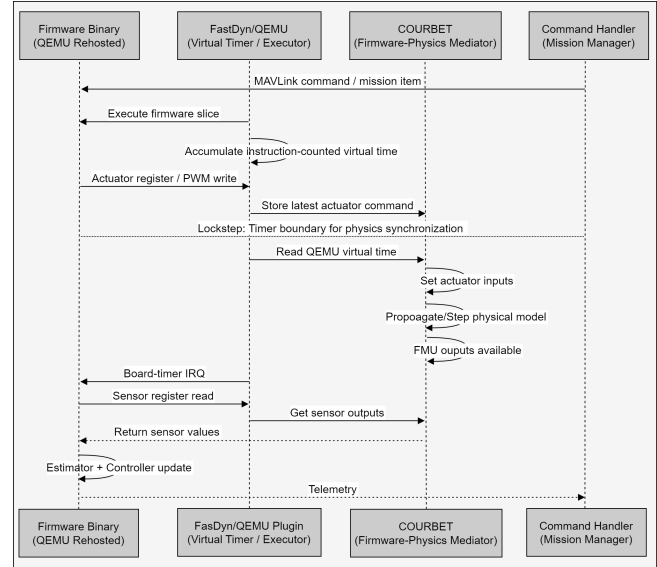


Figure 2. Timer-boundary synchronization between QEMU-hosted firmware, FastDyn/COURBET, and FMU plant. Firmware actuator writes are captured as latest actuator commands. At each board-timer boundary, FastDyn reads QEMU virtual time, advances the plant to that timestamp, and only then raises the board-timer IRQ. Subsequent firmware sensor reads are served from the updated plant outputs.

mands to the emulated actuator hardware interface according to the firmware’s native scheduling and control logic. When the next board-timer lockstep boundary is reached, the coupling layer transfers the most recent actuator commands from this emulated hardware interface to the FMU-based plant model. COURBET then advances the physical model through an FMI co-simulation step using the current simulation time and communication step size. The updated plant states and sensor outputs, including IMU, GPS, barometer, and actuator-state signals, are returned to the coupling layer and made available through the firmware-facing sensor interfaces. The firmware then consumes these updated sensor values through its normal sensor-driver reads and may produce subsequent actuator commands through later writes to the actuator hardware interface, thereby closing the cyber-physical simulation loop. Telemetry and mission-state information may also be forwarded to mission handling utilities or visualization tools.

Listing 1. Example FastDyn/COURBET TOML configuration binding the QEMU-executed ArduCopter firmware binary to an FMI-3.0 quadrotor model

```
[Machine]
platform = "STM32F427"
icount = { shift = 5, sleep = false,
           align = false }
timer_irq_period_ns = 1000000

[CPU]
arch = "arm"
machine = "cortexm"
```

```

cpu = "cortex-m4"
plugin_library = "build/libfastdyn.so"
binary = "virtuals/physics/
    flight_controllers/courbet/bin/
    arducopter_v462"

[FMU]
active = "quadrotor"
auto_build = true

[FMU.models.quadrotor]
model = "FastDyn.Copter"
model_file = "modelica/FastDyn/Copter.mo"
"
output = "out/fmi3/Copter"

[FMU.models.quadrotor.parameters]
# Purdue University Airport (KLAF)
lat0 = 40.414929
lon0 = -86.932387
ground_alg_wgs84 = 149.0

```

Listing 1 shows an example TOML-based configuration used by the proposed FastDyn/COURBET framework. The configuration binds a QEMU-rehosted ArduCopter firmware binary to an FMI-3.0 quadrotor plant model generated by Rumoca from a Modelica source model. The same configuration mechanism orchestrates firmware execution, FMU initialization, virtual timing configuration, mission-handler processes, MAVLink communication utilities, and optional visualization backends such as Gazebo or MAVCesium. QEMU instruction-counted virtual time is configured through the `icount` parameters, while the FMU section specifies the active plant model, Modelica source, output artifact location, and environment-specific initialization parameters.

Within the proposed architecture, Gazebo can be retained as an optional visualization and ecosystem-compatibility backend rather than serving as the primary physics and sensor simulation substrate. The authoritative physical representation instead resides within the FMI-compliant FMU artifact. This separation enables the same firmware execution harness to operate with multiple plant implementations, including low-fidelity kinematic models, higher-fidelity multi-body dynamics, and vulnerability-containing sensor or actuator models, provided that the FMI interface contract is preserved.

3 FMU Export and Execution

Rumoca provides the translation layer between high-level Modelica models and executable FMI artifacts. Since Rumoca itself is described elsewhere (Condie et al. 2025), this paper focuses on how its generated FMUs are integrated into the FastDyn/COURBET FITL architecture rather than on the compiler’s internal details. At a high level, reusable Modelica models describing vehicle dynamics, actuator behavior, sensors, and environmental effects are compiled into FMI-3.0 co-simulation FMUs that can be loaded by the physics hosting of COURBET. Fig-

ure 3 summarizes the resulting integration pipeline.

Listing 2. Rumoca invocation used by FastDyn to export a Modelica model as an FMI-3.0 target. The configuration is resolved into a model file, Modelica class name, and packaging mode, which are then passed to Rumoca’s `compile` command.

```

def cargo_command(build: FmuBuild) ->
    list[str]:
    command = ["cargo", "run"]

    if build.release:
        command.append("--release")

    command.extend([
        "-p", "rumoca", "--",
        "compile", str(build.model_file),
        "--model", build.model])

    for source_root in build.source_roots:
        command.extend(["--source-root", str(
            source_root)])

    command.extend([
        "--output", str(build.output),
        "--target", "fmi3"])

    if build.package:
        command.append("--build")

    return command

```

FastDyn manages FMU generation through a user-defined configuration table shown in Listing 1. When automatic build is enabled, FastDyn resolves the configuration into an internal build specification and constructs the corresponding Rumoca invocation. Listing 2 shows the command construction logic used to generate an FMI-3.0 target from a Modelica source model. The generated FMU is rebuilt automatically when the artifact is missing or stale, allowing simulation configurations to remain reproducible across pinned FastDyn, Rumoca, and Modelica repository revisions.

Internally, Rumoca represents Modelica models as hybrid differential-algebraic equation (DAE) models of the form:

$$\begin{aligned}
 0 &= f_x(v, c), \\
 z &= \begin{cases} f_z(v, c) & \text{at events } (t = t_e), \\ \text{pre}(z) & \text{otherwise,} \end{cases} \\
 m &:= f_m(v, c) \text{ at events } (t = t_e), \\
 c &:= f_c(\text{relation}(v)),
 \end{aligned} \tag{1}$$

where

$$v := [t, \dot{x}, x, y, z, m, \text{pre}(z), \text{pre}(m), u, w, p],$$

and where $x(t)$ denotes the differentiable real states, $y(t)$ algebraic real variables, $z(t_e)$ discrete-time real variables, and $m(t_e)$ discrete-valued event variables. The variables p , t , and t_e represent model parameters, simulation

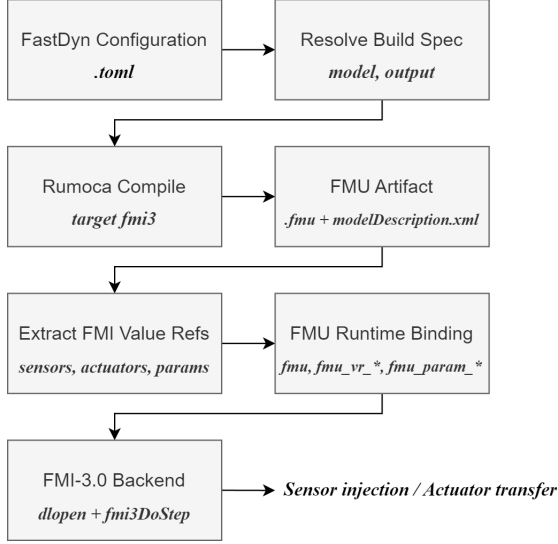


Figure 3. FastDyn FMU export and integration pipeline, from Rumoca-based model compilation to runtime FMI binding for sensor injection and actuator transfer.

time, and time instance at events, respectively. The symbols u and w denote model inputs and outputs. The operator $\text{pre}(\cdot)$ denotes the value of a variable immediately before an event transition, and c represents the conditions generated from relational expressions and converted `if`-expressions, including `when`-clauses.

The Rumoca compiler pipeline parses Modelica source models using the Rust-based Parol parser generator, which automatically generates parsers from an EBNF grammar while allowing customized parse-tree handling during parsing. The resulting abstract syntax tree (AST) is flattened to remove hierarchical Modelica class structures and lowered into the hybrid-system representation shown above. Continuous dynamics, discrete state updates, relational conditions, and event operators such as `when`, `reinit`, and `pre()` are explicitly preserved before backend-specific FMI or symbolic code generation.

The current workflow exports wrapper models such as Copter, Rover, and Plane for ArduPilot firmware. These wrappers encapsulate firmware-specific interfaces while inheriting reusable rigid-body dynamics and subsystem models from a shared Modelica library. As a result, vehicle-generic physical models can evolve over time independently from firmware integration concerns such as PWM mappings, MAVLink ports, mission helpers, geographic initialization, or sensor routing. The exported FMU therefore acts as a replaceable physical plant artifact behind a stable firmware interface. This separation supports fidelity variation of FMU without modifying the surrounding execution framework. A low-fidelity kinematic approximation, a higher-fidelity rigid-body model, or a vulnerability-oriented sensor model can be exported behind the same FMI interface as long as the required inputs and outputs are preserved. Consequently, physical fidelity becomes a property of the FMU artifact rather than a

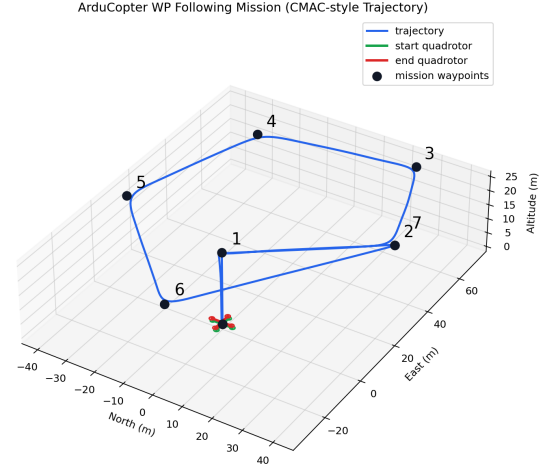


Figure 4. ArduCopter waypoint-following result for the CMAC-style trajectory. The blue curve shows the simulated flight trajectory, black markers indicate mission waypoints, and the green and red airplane markers denote the start and end positions, respectively.

property of simulator-specific plugins or middleware integrations. This allows experiments to vary plant complexity while keeping the firmware, mission logic, and orchestration layer unchanged.

At runtime, the generated FMU is integrated into the FastDyn execution environment through an FMI-3.0 backend implemented within COURBET. The firmware executes using instruction-counted virtual time, while the FMI backend advances the physical plant through `fmi3DoStep()` calls. Actuator outputs generated by the firmware are mapped to FMU inputs, and simulated sensor states are injected back into the firmware using extracted FMI value references managed by COURBET. Additional runtime components, including the mission helper, MAVProxy bridge, and optional visualization tools, are orchestrated alongside the firmware and FMU plant, enabling synchronized firmware-integrated simulation while remaining independent of any single robotics simulator implementation.

4 Firmware-Integrated Simulations

We evaluate the architecture on three robotic vehicle classes: rover, fixed-wing plane, and multi-copter. These vehicles exercise different aspects of the simulation stack. The rover emphasizes ground interaction, steering, throttle mapping, and low-speed navigation. The fixed-wing plane emphasizes aerodynamic response, attitude stabilization, and mission-level guidance over longer trajectories. The copter emphasizes fast attitude dynamics, motor lag, high-rate inertial sensing, and tight coupling between the firmware control loop and the plant.



Figure 5. MAVCesium visualization of ArduCopter FITL. The interface displays the simulated vehicle attitude, position, and flight status using telemetry.

4.1 Arducopter

For the copter case, FastDyn provides a quick-start workflow that invokes Rumoca to generate the quadcopter FMU, executes ArduCopter inside QEMU, launches the mission handler, and exposes a MAVCesium web visualization endpoint. The default configuration initializes the vehicle on the Purdue University Airport tarmac, uses a 1 ms board tick, maps ArduPilot PWM outputs linearly to aerodynamic motor speeds, and incorporates an explicit first-order motor lag within the FMU plant model. The resulting simulation is presented in Figure 4, while the MAVCesium rendering shown in Figure 5 provides real-time visualization of the vehicle trajectory, attitude evolution, and mission execution state.

4.2 Ardurover

For the rover case, FastDyn generates and executes a ground-vehicle FMU coupled with ArduRover running inside QEMU. The default scenario performs an autonomous waypoint-following mission using the same FMI-based execution pipeline and mission orchestration framework. The rover simulation demonstrates that the proposed architecture can support the nonholonomic ground-vehicle dynamics and closed-loop navigation behaviors, while preserving a unified firmware-integrated simulation interface across vehicle classes.

4.3 Arduplane

In the fixed-wing case, FastDyn couples a fixed-wing FMU with ArduPlane executing in the QEMU-based firmware environment. The evaluated mission consists of climb, waypoint-following, and descent to exercise trajectory-tracking as shown Figure 6. The resulting simulation demonstrates that the FMI-centric architecture can support the continuous aerodynamic flight dynamics, mission-level autonomy, and FITL without relying on simulator-specific vehicle plugins. However, we have identified a coordinate-frame inconsistency in broadcast-

ArduCopter WP Following Mission(CMAC-Circuit Trajectory, Window: 30-90 s)

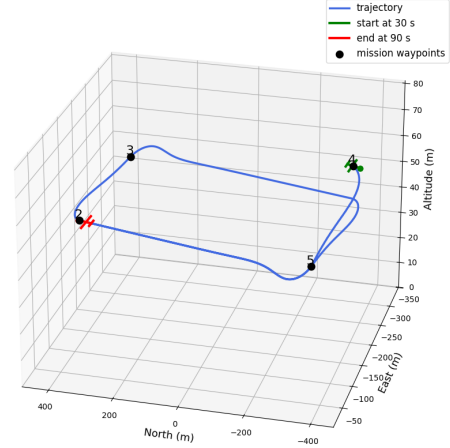


Figure 6. ArduPlane waypoint-following result for the CMAC circuit mission over 60 s time window. The blue curve shows the simulated flight trajectory, black markers indicate mission waypoints, and the green and red airplane markers denote the start and end positions, respectively.

ing navigation data into the firmware, which will be addressed in subsequent development.

4.4 CPS Testing and Implications

Table 1 summarizes the vehicle class, firmware version, FMU model, target board, and mission type. The proposed FMI-based simulation architecture is intended not only for single-mission demonstrations, but also as a reusable cyber-physical testing workflow for firmware-integrated robotic systems. The vehicle simulations presented in this section demonstrate representative execution scenarios within a broader experimentation framework that can be extended to Monte-Carlo simulation campaigns, parameter sweeps, firmware fuzzing, coverage-guided exploration, and falsification-oriented robustness analysis. Because the physical plant is encapsulated behind a stable FMI interface, perturbation inputs, parameter mutations, sensor faults, environmental variations, and model substitutions can be introduced without modifying the firmware execution harness itself. At the same time, the FastDyn/COURBET execution layer provides access to cyber-domain observations including firmware traces, MAVLink traffic, timing behavior, and execution coverage. Consequently, the architecture supports scalable formal-methods-oriented workflows in which physical-system perturbations and firmware-level exploration can be combined within a unified execution environment.

5 Discussion and Limitations

The proposed architecture should be understood as complementary to, rather than a complete replacement for, existing robotic simulators. In our case, Gazebo can remain as a strong choice for contact-rich scenes, perception simulation, synthetic data generation, rendering, and integra-

Table 1. Summary of evaluated vehicle configurations and mission behaviors.

Vehicle	Firmware	MCU	Vehicle Model	Mission
Copter	ArduCopter	STM32F427	Quadcopter (3DR Iris)	TO, WP, LD
Rover	ArduRover	STM32F427	Rover (Aion R1)	WP
Plane	ArduPlane	STM32F427	Fixed-wing (Skywalker X8)	TO, WP, LTR, LD

TO: takeoff, WP: waypoint tracking, LTR: loiter, LD: landing.

tion with robotics middleware. The advantage of the FMI-based architecture is strongest when the experimental objective requires a replaceable mathematical plant model, deterministic firmware-plant synchronization, formal perturbation interfaces, or scalable execution.

The fidelity of an FMI-based simulation is not guaranteed by FMI alone. Accuracy depends on the Modelica equation model, the Rumoca translation, solver settings, event handling, communication step size, actuator and sensor models, and the quality of the firmware interface mapping. Therefore, the contribution of this paper is architectural rather than claiming that our framework is universally more effective than Gazebo. The key claim is that Modelica/FMI provides a cleaner model boundary for firmware-integrated cyber-physical system (CPS) experimentation campaigns.

Another limitation is that high-fidelity terrain, contact-rich environments, and perception models may still be easier to implement in existing robotics simulators. For this reason, the proposed architecture keeps Gazebo as an optional backend and compatibility path. In future work, a hybrid configuration could use FMUs for vehicle dynamics and firmware synchronization, while delegating selected perception or contact-rich subsystems to other simulators, such as Gazebo and Isaac Sim.

6 Conclusion

This paper presented a Modelica/FMI-based cyber-physical simulation architecture for firmware-integrated robotic vehicles. By transitioning from simulator-specific plugins to Rumoca-generated FMI-3.0 FMUs, the proposed architecture decouples reusable vehicle dynamics models from simulator-specific integration logic and platform-dependent interfaces. Our FITL framework, FastDyn and COURBET, couples these FMUs to QEMU-based firmware execution using instruction-counted virtual time, enabling effective mission execution, model substitution, and scalable campaign orchestration.

Our approach was demonstrated on copter, rover, and plane configurations, showing that the same firmware-integrated execution framework can support multiple robotic vehicle classes through FMU configuration changes. The architecture is particularly suitable for experiments that require controlled fidelity variation, custom sensor or actuator vulnerability models, falsification interfaces, and reproducible parallel campaigns. Future work will expand the model library, improve automated vali-

dation against Gazebo and real-flight traces, and investigate hybrid backends that combine FMI-based physics with simulator-based rendering and perception.

Acknowledgements

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) via Contract No.FA8750-24-2-0500. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA and the Government.

References

- ArduPilot Development Team (2025). *ArduPilot*. <https://ardupilot.org>. Accessed: 2026-05-10.
- Berndt, Jon S. (2004). “JSBSim: An Open Source Flight Dynamics Model in C++”. In: *AIAA Modeling and Simulation Technologies Conference and Exhibit*. DOI: 10.2514/6.2004-4923.
- Cacace, Jonathan, Nicola Mimmo, and Lorenzo Marconi (2020). “A ROS Gazebo plugin to simulate ARVA sensors”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 7233–7239.
- Condie, Micah, Abigaile Woodbury, James Goppert, and Joel Andersson (2025). “Rumoca: Towards a Translator from Modelica to Algebraic Modeling Languages”. In: *Modelica Conferences*, pp. 1009–1016.
- Dosovitskiy, Alexey, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun (2017). “CARLA: An Open Urban Driving Simulator”. In: *Proceedings of the 1st Annual Conference on Robot Learning (CoRL)*, pp. 1–16.
- Junghanns, Andreas et al. (2021). “The functional mock-up interface 3.0-new features enabling new applications”. In: *Modelica conferences*, pp. 17–26.
- Koenig, Nathan and Andrew Howard (2004). “Design and use paradigms for gazebo, an open-source multi-robot simulator”. In: *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 3. Ieee, pp. 2149–2154.
- NVIDIA (2023). *NVIDIA Isaac Sim*. <https://developer.nvidia.com/isaac/sim>.
- Pant, Kartik A, Zhanpeng Yang, James M Goppert, and Inseok Hwang (2023). “An open-source gazebo plugin for gnss multipath signal emulation in virtual urban canyons”. In: *AIAA SCITECH 2023 Forum*, p. 2586.
- Rooney, Michael P, Minhyun Cho, Muhammad Hammad Bashir, Evan English, Colin Smith, Ashwin Nambiar, et al. (2026). “OptiFuzz: Finding Cyber-Physical Bugs with Physics Accurate Rehosting of Unmanned Vehicle Firmware”. Manuscript in preparation.

- Shah, Shital, Debadeepta Dey, Chris Lovett, and Ashish Kapoor (2018). “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: *Field and Service Robotics*.
- Wetter, Michael (2009). “Modelica-based modelling and simulation to support research and development in building energy and control systems”. In: *Journal of Building Performance Simulation* 2.2, pp. 143–161.