SAARLAND UNIVERSITY

Master Thesis

Embedded Systems M.Sc.

Modular Full-System Simulation for Cyber-Physical Systems

Author:	Gideon Mohr
First Reviewer:	Dr. Antoine Kaufmann
Second Reviewer:	Prof. Dr. Martina Maggio
Advisor:	Artem Ageev
Submission Date:	07.08.2025

Faculty of Mathematics and Computer Science
Department of Computer Science
Max Planck Institute for Software Systems
Operating Systems Group

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 07.08.2025 _____

(Gideon Mohr)

Abstract

Cyber-Physical Systems (CPS) are frequently used in safety-critical settings, where mistakes in the hardware configuration or software bugs can result in significant costs increases or even threat public safety.

Frequently, such issues stem from the interaction between multiple components and do not appear when each component is tested in isolation. As a result, comprehensive system-level testing is essential to identify those faults early in the development process.

Virtual prototyping using simulation can help avoiding these issues before deployment by accurately modeling all involved components. This technique is already widely used when designing cloud infrastructure or when developing custom hardware components like accelerators.

However, unlike software systems, CPS also require a precise simulation of their physical environment alongside the simulation of the control logic. Existing simulators model this interaction at a low fidelity failing to capture certain errors that are just visible on certain hardware configurations, require specific environmental circumstances or need precise timing to make an impact.

This thesis proposes an extension of the SimBricks simulation framework allowing for a modular full-system simulation of CPS. The new physical interface connects a physical simulator with simulators supported in the SimBricks ecosystem. The synchronization mechanisms in SimBricks allow to keep the physical simulation in sync with the simulation of the software, thus eliminating the dependency of the simulation results on the simulation hardware. This approach is evaluated using the Gazebo simulator with ArduPilot, a widely used controller for unmanned aerial vehicles (UAVs).

Contents

Abstract ..... **v**
Contents ..... **vii**
1 Introduction **1**
2 Background **3**
2.1 Cyber-Physical Systems 3
2.1.1 CPS Control Platforms 5
2.2 Full-System Simulation 5
2.2.1 Full-System Simulation in SimBricks 6
2.3 Physics Simulation 8
3 Motivation **13**
4 Design **15**
4.1 Requirements 15
4.2 An Interface for Physical Simulators 16
4.3 Limitations 18
5 Implementation **21**
5.1 Protocol 21
5.1.1 The `intro` messages 21
5.1.2 The Configuration Phase 22
5.1.3 Exchanging Values 22
5.2 Gazebo Integration 23
5.3 gem5 Integration 25
6 Evaluation **29**
6.1 Setup 29
6.2 RQ1: Evaluating Hardware Configurations 30
6.3 RQ2: Detecting Previously Undetectable Bugs 32
6.4 RQ3 & RQ4: Performance & Scalability 34
6.5 RQ5: Implementation Effort 37
7 Related Work **39**
8 Discussion and Future Work **41**
8.1 Improving the performance of the physical simulation 41
8.2 Simulating external conditions 42
8.3 Validating the accuracy of the simulation 42
8.4 Choice of the right simulator 43
9 Conclusion **45**
Glossary ..... **I**
List of Figures ..... **V**

List of Tables ..... **V**
List of Code Listings ..... **VI**
Bibliography ..... **VII**

1 Introduction

Cyber-Physical Systems (CPS) have become a crucial part of our day-to-day life over the last decades. The ongoing digitalization of our society lead to an increasing amount of electronic devices that feature digital components and interact with the environment using sensors and actuators.

However, beyond consumer electronics, CPS are important for many different, usually safety-critical applications. In aviation correct interaction of sensors, turbines or propellers and the interfaces communicating the current state to the pilot are crucial for safe and secure operation of an aircraft. Similarly, in astronautics many missions rely on the correct behavior of spacecraft or satellites which usually cannot be patched easily in case of errors once the devices are launched. A mission failure due to bugs in software or hardware may result very costly and could cause significant delays in the initial timeline. For example, the Ariane 5 mission failed partially due to an unexpected horizontal velocity [1] which caused a chain of failures leading to a dangerous change of trajectory and a destruction of the spacecraft. Apart from aerial vehicles, other CPS might also operate in safety-critical environments or are hard to patch once widely deployed. For example, bugs in the software of industrial robotic applications may put the life of factory workers at risk when heavy robotic arms behave unpredictably.

Consequently, extensive testing of software, hardware and their interactions is required before the deployment of CPS in safety-critical applications. To test software and hardware individually, there are many different frameworks that have proven to work pretty well at identifying possible problems ahead of deployment. However, some problems might only arise in certain interactions of different components which can hardly be found when testing those components in isolation since those faults might only be triggered by rare and very specific external inputs. In some situations, interfaces between components may not adhere to their specifications or may be interpreted inconsistently. When tests are performed in isolation and based on similar assumptions, such mismatches are unlikely to be detected.

To increase the likelihood of identifying those inputs and uncovering those faults, thorough testing of each component with many different inputs is required. Such a testing strategy might also involve narrowing down the space of possible inputs by using mathematical models of the environment and other component.

Those models must also account for noise possibly introduced by sensors to avoid excluding slightly inaccurate yet realistic inputs. As a result, such approaches must conservatively over-approximate noise levels and input ranges which can sometimes suggest the need for better, less noisy hardware components that are potentially more expensive and thus unnecessarily increase the production cost of the final CPS.

To avoid such issues, it is desirable to develop and use testing frameworks that integrate different components to avoid relying on isolated tests of every component and to improve the accuracy of the simulation and boost the confidence in the results obtained from those tests.

There are various different approaches to test CPS at the software level using Software in the Loop (SITL) testing. However, usually there are some drawbacks with those solutions. Some of those solutions are tailored to test for specific properties of a system and thus cannot easily be reused on other systems or for other properties. Some simulators also operate at a high level of abstraction making it impossible to detect certain bugs since their root cause may no longer appear in the abstraction.

This project introduces a simple yet flexible framework that allows full-system simulation for different kinds of CPS. The framework makes use of prior work on SimBricks extending it to allow interactions with a simulated physical environment. This approach allows to easily integrate new devices and new control software resulting in a flexible environment to prototype CPS, test specific combinations of hardware configurations, environmental conditions and model the interaction of various CPS even for large-scale deployments.

Since all components are simulated in software, no expensive hardware is required while still allowing to test at a high fidelity that previous work fails to provide.

Before motivating this work further in Chapter 3, Chapter 2 introduces the necessary background. Subsequently, Chapter 4 proposes the concrete design of the integration into the existing SimBricks framework and Chapter 5 provides details on how this design has been implemented for the evaluation presented in Chapter 6. Throughout this project, the integration of ArduPilot (an open-source control framework for autonomous vehicles) with the physics simulator Gazebo served as a case study to evaluate this framework.

2 Background

Developing Cyber-Physical Systems (CPS) is a complex task as their design involves developing the software, choosing appropriate hardware while taking the implications of the physical environment into account. This chapter first introduces the challenges that arise when designing CPS (Section 2.1), introduces full-system simulation (Section 2.2) and describes how the physical environment can be simulated (Section 2.3).

2.1 Cyber-Physical Systems

CPS come in many different shapes and are an indispensable part of our daily life. Generally, the term refers to systems with digital components, i.e. a processing unit that interfaces in some way with its physical environment. This is usually done using sensors and/or actuators that are connected to the device. The sensor readings are then used by the digital controller to decide how the actuators should behave. In the majority of implementations, this is achieved by a periodic control loop that queries the current sensor readings, updates the internal state estimation and then decides on certain actions to move the system towards the desired state.

Common examples for CPS include (partially) autonomous vehicles such as rovers, planes, or drones. Those systems use sensors to evaluate their environment to find the correct path of movement for their mission and use their motors to move towards the desired location. However, also "traditional" vehicles such as cars and airplanes are considered to be CPS despite being manually operated. Sensor data needs to be constantly and timely provided to the operator and some systems like airbags need to automatically activate under certain conditions. Industrial robots are another group of CPS which usually complete repetitive tasks in manufacturing or logistics. Apart from the actual manufacturing process which requires sensors and actuators to accomplish the desired result, sensors are also used to ensure worker's safety.

The direct interface with the physical environment comes with many challenges. Most importantly, the control loop needs to operate under real-time conditions. External and environmental changes should have a timely reaction, e.g. a sudden gust of wind should cause immediate countermeasures even if the CPS is currently performing some other task. This means that the control loop should have a very predictable timing behavior and needs to be executed at a sufficiently

high frequency such that external environmental factors cannot destabilize the system.

Furthermore, sensors usually introduce some noise and lose some precision when converting the analog readings to digital values consumed by the control logic. This needs to be taken into account when computing the current state which for those reasons will always be slightly inaccurate.

At the same time CPS often operate disconnected from the power grid on battery or solar power. Therefore, power consumption is another limiting factor that needs to be taken into account.

Those constraints increase the difficulty of developing CPS and introduce some unique challenges. Many CPS use real-time operating systems making it easier to ensure that the frequency constraints are met. Filters are used when handling noisy sensor readings in order to reduce its impact on the internal state. Power models help estimating the available power resources and whether they are sufficient to accomplish the current mission.

In general, high predictability of both the software and the hardware is desirable for CPS and other embedded systems that will be deployed in safety-critical environments. To provide formal guarantees on the execution time, worst-case execution time (WCET) analysis can be used to provide an upper bound on the execution time. In less critical scenarios, the WCET may be estimated by adding a constant factor to the highest observed execution time (HOET) which works in many cases in practice but fails to provide any guarantees[2]. However, to provide guarantees, a pessimistic over-approximation of the actual runtime is used, which means that the actual HOET observed in practice might be a lot lower. Especially microarchitectural optimizations like caches and pipelines which cannot be influenced at the software level make the execution time hard to predict[3]. Specially designed hardware can make use of predictable pipelines and provide software-controlled scratchpad memories which in reality might be slower than other alternatives but provide better guarantees in the worst case.

In the development of CPS over-approximations due to WCET analysis might lead to hardware upgrades which would not have been necessary thus possibly increasing the cost of the entire system. Improving the analysis or being able to show that a lower WCET for specific safety-critical scenarios applies (e.g. controlled shutdown in case of emergencies) might allow to settle on less capable and

possibly cheaper hardware that is guaranteed to be able to handle certain safety-critical scenarios but fails to provide guarantees in the general case.

2.1.1 CPS Control Platforms

Due to the large variety of CPS, there is also a large variety of CPS controllers and control platforms. While some CPS have very specialized software designed for their specific purpose, there are also some well-known platforms that can target different hardware configurations and different use-cases. For robotics, there is the Robot Operating System (ROS) [4] which is broadly used in robots in both research and industry. The Open Robot Control Software (Orocos) [5] project provides a toolchain to develop robots with real-time components and comes with a library of several components that can be integrated directly into this toolchain. For Unmanned Aerial Vehicle (UAV), the two most well-known flight-controllers are PX4[6] and ArduPilot[7]. Both of them are widely used in research and industry including being deployed on commercially available UAV [8, 9]. Both of them use the MavLink protocol [10] to communicate with the ground station and receive mission commands. This protocol defines a lightweight message format for communication between the ground control station and the UAV.

ArduPilot is a platform that can be deployed on unmanned copters, planes, rovers and boats. Its architecture allows to easily integrate different hardware components, schedule periodic tasks to communicate with the hardware and influence the system's internal state. Hardware Abstraction Layers (HALs) are provided for various platforms, including the ESP32, ChibiOS and Linux. The custom firmware builder [11] currently supports 241 different boards for the copter variant only.

2.2 Full-System Simulation

Simulation is nowadays an important aspect of the testing process in software development and systems design since it provides a reproducible and reliable way to evaluate the performance of a given system.

However, not every simulation has the same requirements. In some cases, only the resulting data might be important, while in other cases the runtime matters and in some other cases one might even care about the power consumption during the computation. To accommodate these different necessities, many different, specialized simulators have been developed over time. For example OMNet++[12]

is framework to build domain-specific simulators for network simulations and CARLA[13] is a simulator tailored for research towards autonomous vehicles

While these simulators are very well suited to test individual pieces of software or very specific scenarios, in many cases the interesting results come from the interaction of multiple components, e.g. two hosts that communicate over the network. Simulating all relevant components of a complex system is called full-system simulation. One approach to do this kind of simulation is to develop a specialized simulator that incorporates the whole system into one large simulation. However, this approach requires significant manual effort and provides no flexibility when it comes to small changes to the topology of the system. Another approach is to take existing simulators for off-the-shelf components and interconnect them to recreate the simulated topology.

The latter approach provides more flexibility regarding the simulated topology and makes it easier to integrate a new simulator for a specific component. However, the heterogeneity of simulators and their interfaces creates challenges regarding inter-simulation communication and synchronization to ensure that external events are received at the same time they would be received in the real system. Without synchronization every component would be executed with a variable slowdown causing simulators to drift away from each other in terms of simulated time.

2.2.1 Full-System Simulation in SimBricks

SimBricks [14] is a framework for full-system simulation that overcomes the aforementioned challenges by interconnecting different simulators at natural boundaries using standardized interfaces and by coordinating the progress of simulated time using synchronization messages.

SimBricks provides support for several commonly-used simulators such as gem5 [15, 16, 17] and QEMU [18] for host simulation, ns-3 [19] for network simulation as well as Verilator [20] to simulate custom hardware designs. The simulators are connected using SimBricks adapters that implement one of the available interfaces. These interfaces aim to separate the simulators at natural boundaries. By default, there are interfaces implementing PCIe (to connect hosts with NICs or accelerators), Ethernet (to connect the NICs to the network simulator) and a memory interface (to connect hosts with different kinds of memory devices, e.g. networked memory).

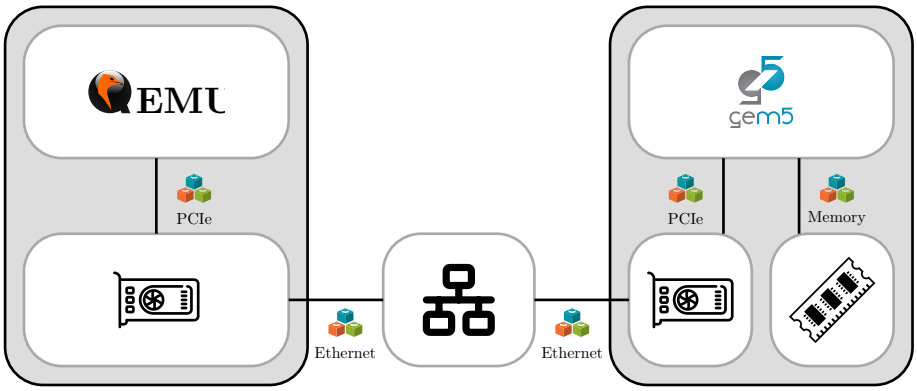


Figure 1: Simulator configuration for a SimBricks simulation involving two hosts connected over the network.

During the simulation, every component is simulated in a different process. Communication is handled through shared memory queues allowing for efficient communication. A configurable parameter `sync_interval` ensures that each simulator is at most `sync_interval` ahead or behind any other directly connected simulator, ensuring that all simulators progress at roughly the same speed.

A simulation in SimBricks is configured using a Python script describing the necessary components and wiring the interfaces up. The orchestration framework then takes care of instantiating the simulators and configuring the shared-memory queues for communication. Figure 1 shows the setup of a small SimBricks experiment consisting of two hosts, one simulated using QEMU and one using gem5. Both feature a NIC that connects them to the simulated network. The gem5 host additionally has a dedicated memory device simulated in another process. Such a setup can be used to simulate a client-server application where the server needs to access memory to fullfil the incoming requests. Tuning the simulations parameters allows to e.g. measure the impact of the memory latency on the latency that the client perceives or limiting the client's bandwidth allows simulating poor network conditions.

To simulate this setup in SimBricks, the setup needs to be specified using the provided orchestration framework as shown in Listing 1. Each required simulator is instantiated and configured with a few important parameters like the IP address or the size of the simulated memory. Then connections between simulators are defined and lastly the experiment is added to the queue. SimBricks provides a script that takes this specification, configures the simulators accordingly and executes the simulation.

```python
from simbricks.orchestration.experiments import Experiment
from simbricks.orchestration.nodeconfig import I40eLinuxNode
from simbricks.orchestration.simulators import Gem5Host, QemuHost, I40eNIC, SwitchNet, BasicMemDev
from nodeconfig import ServerApp, ClientApp

e = Experiment(name='demo')

# create server
server_config = I40eLinuxNode()
server_config.ip = '10.0.0.1'
server_config.app = ServerApp()
server = Gem5Host(server_config)
server.name = 'server'
e.add_host(server)

# attach the server's memory
server_mem = BasicMemDev()
server_mem.name = 'server_mem'
server_mem.addr = 0x2000000000
server_mem.size = 512 * 1024 * 1024
server.add_memdev(server_mem)

# attach server's NIC
server_nic = I40eNIC()
e.add_nic(server_nic)
server.add_nic(server_nic)

# create client
client_config = I40eLinuxNode()
client_config.app = ClientApp()
client_config.server_ips = ['10.0.0.1']
client = QemuHost(client_config)
client.name = 'client'
client.wait = True
e.add_host(client)

# attach client's NIC
client_nic = I40eNIC()
e.add_nic(client_nic)
client.add_nic(client_nic)

# connect NICs over network
network = SwitchNet()
e.add_network(network)
server_nic.set_network(network)
client_nic.set_network(network)

experiments = [e]
```

Listing 1: Configuration script for the SimBricks simulation shown in Figure 1

2.3 Physics Simulation

For a full-system simulation of a CPS, simulating the digital components of the system is not sufficient. To capture all interactions with the environment, a physics simulator is needed to model the dynamics of the environment.

Simulating this environment accurately however is not straightforward. While digital components can be modeled as a sequence of discrete events (e.g. a sequence

of instructions being executed on a CPU or a sequence of function calls), the physical environment is continuous in both space and time. Modeling this can be done using differential equations that allow calculating the state of the environment over time. To integrate this continuous model with a discrete controller, there are two orthogonal options.

On the one hand, one can capture the control logic in mathematic equations that compute the CPS's control outputs for a certain state of the environment. This approach integrates well with the differential equations from the environmental model. This set of equations can be used to precisely compute the behavior of the CPS in the continuous space and over some period of time. The drawback of this approach is that one needs an exact model of the control logic in the form of mathematical equations which is usually hard to obtain. Complex controllers might require thousands of lines of code that cannot easily be transformed into mathematical equations and the timing of those computations will depend on the specific hardware. To simplify the mathematical representation of the control logic, several approximation methods can be applied, e.g. worst-case execution time (WCET) analysis that allows to put an upper bound on the execution time of certain code sequences. While this allows to prove that certain hardware is capable enough for a certain task, the over-approximation might lead to costly and unnecessary upgrades of the deployed hardware. Matlab/Simulink[21, 22] is a popular choice for such a simulation. This tool allows to model both the physical environment and the control logic mathematically at various levels of detail and allows to create user interfaces to interact with the simulation.

On the other hand, the physics simulation can be set up to interact with the control logic in an event based fashion. In this case, the physics simulator proceeds in fixed time intervals during which the control inputs are assumed to be stable. This allows to then compute the differential equations on how the state will evolve during the next interval. Afterwards, the control inputs are sampled again before executing the next step. While this does not model the dynamics as precisely as the first approach, it does not require any model of the control logic. Instead, the actual controller can be used to compute the control inputs in every sampling step.

Independently of the choice of the simulator, a physical simulation consists of two important parts. Most importantly, there is the simulation of the physical dynamics. This simulation considers all simulated entities, their mass and their shape as well as movable joints, friction and collisions of multiple objects to

compute forces, velocities and relative positions of those objects at some point during the simulation. Such a simulation is not only needed in research and development but can also be found in modern games where a 3D model is used to represent the current state of the game which is subject to physical dynamics although instead of focussing on a high precision, rendering movements such that the user perceives them as natural is more important in this case.

Apart from that, one needs to be able to specify the configuration of the environment and all entities. Ideally, certain commonly-used components are predefined such that the user does not need to specify the respective attributes individually. The specification can also include virtual sensors that are placed on certain objects that can be read and fed back into the simulation of the control loop.

Gazebo[23] is a collection of simulation-related libraries forming an ecosystem for a large range of physical simulations using a step-based approach. It uses the ODE[24] backend to compute the dynamic behavior of the simulated entities. The SDF[25] format is used to specify the simulated world as well as all entities present within the environment. The description of the world consists of parameters that affect the entire simulation, e.g. simulation granularity, gravity and weather conditions. These properties alow worlds to mimic the earth's surface as well as water, airspace or even other planets. Inside these worlds, any number of models can be placed describing independent entities present in the world. Models are described by their form factor, mass and aerodynamics and can be composed of multiple joints and links that connect different parts of the model. Dynamic behavior as well as virtual sensors are integrated using plugins which can also be imported directly in the SDF description. Each plugin provides hooks to be called before and after the world's state gets updated. Inside these hooks, it is possible to change the state of the simulation, e.g. by applying a force on a joint emulating a motor. Communication between plugins is achieved using a message passing system with topics. Those messages can contain information about the state of the simulation, e.g. the current simulation time is published as `gz::msgs::Clock` under `/clock`, the state of specific components, e.g. an IMU sensor publishes its readings periodically as `gz::msgs::IMU` or provide commands to specific components, e.g. to control the force applied to a joint. Each plugin can advertise topics and publish messages for each advertised topic. On the other end, it can also subscribe to any advertised topic and receive a callback once a message is published.

```
1  <sdf version="1.8">
2    <world name="simple_world">
3      <!-- max_step_size ensures that plugins are called every 100us -->
4      <physics name="100us" type="ignore">
5        <max_step_size>0.0001</max_step_size>
6      </physics>
7      <!-- Import GPS plugin -->
8      <plugin filename="gz-sim-navsat-system" name="gz::sim::systems::Navsat" />
9      <!-- Add models -->
10     <model name="object_with_gnss_sensor">
11     <!-- Define position in the world -->
12     <pose degrees="true">0 0 1 0 0 90</pose>
13     <link name='base_link'>
14       <!-- Optionally define mass, center of gravity etc. -->
15       <inertial>
16         <mass>1.5</mass>
17       </inertial>
18       <!-- Define shape, size friction -->
19       <collision name='base_collision'>
20         <geometry>
21           <box>
22             <size>0.47 0.47 0.23</size>
23           </box>
24         </geometry>
25         <surface>
26           <friction>
27             <ode>
28               <mu>100000.0</mu>
29               <mu2>100000.0</mu2>
30             </ode>
31           </friction>
32         </surface>
33       </collision>
34       <!-- Add a GNSS sensor that updates every 100 cycles -->
35       <sensor name="navsat_sensor" type="navsat">
36         <always_on>1</always_on>
37         <update_rate>100</update_rate>
38       </sensor>
39     </link>
40   </world>
41 </sdf>
```

Listing 2: The definition of a world in the SDF format. It includes only one object of a certain size that is equipped with a Global Navigation Satellite System (GNSS) sensor.

Listing 2 shows a small sample of how a simple world is defined in Gazebo. A model has three different types of components. Links (lines 13-39) represent static components. Additionally, there can be joints which represent a connection between links and define how two links can move relative to each other. Collisions (lines 19-33) on the other hand define the shape of a link and are used to calculate collisions between models. Line 8 and lines 35-38 show how a sensor can be loaded using a plugin.

3 Motivation

The first chapter highlighted that CPS often have high requirements towards reliability and functional correctness. Additionally, those system might appear in settings where testing on the physical hardware is expensive or infeasible. Software-based simulation can be an alternative to this since it eliminates the requirement to build the hardware beforehand and potential failures cannot result in damaged or destroyed hardware that needs replacement.

Various different simulation approaches already exist modeling the system at different levels of detail. Each of these simulators has its advantages and downsides. The ArduPilot SITL simulator for example works great to test the interaction with the ground control station, i.e. receiving mission commands, changing the flight mode or completing a fixed mission. The approach proposed in [26] using Matlab/ Simulink uses a mathematical model of the flight controller and the aerodynamics which might outperform other simulators in precision given a precise mathematical representation while some aspects of the UAV like computational latencies are not part of the abstraction and thus cannot be tested in such a simulation.

While those simulators help avoiding costly errors while building CPS, many errors cannot be detected when different components are tested in isolation. Additionally, some errors only appear with certain hardware configurations or when specific environmental factors match.

A study[27] analyzed commits in the ArduPilot repository identifying commits likely corresponding to bug fixes. Those code changes were then manually analyzed to identify under which circumstances this bug could have been observed. The authors concluded that a large number of bugs are not detectable using traditional testing methods or simulation using ArduPilot's built-in SITL simulator. This simulator bypasses most of the platform-specific code by providing its own HAL for the simulation. Errors in code for specific hardware configurations, e.g. drivers for different IMU sensors thus cannot be detected in the simulation as these modules are not part of the simulation. Other bugs require certain hardware limitations such as a low memory capacity. The lack of simulators capable of detecting this kind of bugs may turn out to be very costly, especially if the bugs cause the hardware to be destroyed e.g. in a crash.

Therefore, in order to build reliable CPS, simulation at a higher fidelity considering all components of the system is crucial. Simulation not only allows

testing regular operations but the ability to configure environmental factors helps to recreate rare conditions that in reality might not occur very often and are thus difficult to test in reality during development.

Furthermore, simulating under realistic conditions allows to choose appropriate hardware by testing how different components can handle the required workload. This allows to dimension CPU and memory accordingly and testing different sensor models (according to their specification from the respective datasheet) might allow to settle on more affordable hardware that is still capable of providing the required functionality.

Another limitation of many existing simulators is the missing interoperability. First of all, since most simulators do not implement a standardized interface, interconnecting different simulators might be challenging. Apart from that, existing simulators cannot easily be combined due to their lack of synchronization. For example, the ArduPilot SITL simulator needs a ground station to receive mission control commands. Without synchronization, the ground station operates at real time while the SITL simulator has its own simulated time that may face a variable slowdown due to the limited hardware capabilities. A simulation in which e.g. the drone receives a command to land exactly 10 seconds after reaching a certain altitude is therefore infeasible since (without any further modifications) the ground station has no way of determining how much simulated time has passed.

This project enables full-system simulation for CPS by integrating Gazebo into the SimBricks framework. This integration happens transparently to the control software running in a separate simulator which thus does not need to enable special drivers for simulated environments but accesses the hardware as it would do in a real scenario. The simulation takes care of emulating the sensors using data provided from Gazebo. The synchronization mechanisms provided by SimBricks enable precise modeling of the interactions between various components. Every simulated CPS shares the same environment in Gazebo allowing to capture interactions between the devices, e.g. collisions.

To show the feasibility of this approach, ArduPilot is integrated and evaluated (Chapter 6). However, the flexibility of the simulators and the interfaces does not restrict this approach to UAV but this framework can also be used to e.g. simulate spacecraft, to evaluate automated factory designs or to build vehicle-to-infrastructure or vehicle-to-vehicle communication.

4 Design

The previous chapter outlined the importance of full-system simulation for the development of reliable CPS. This chapter focusses on how the existing SimBricks framework can be extended to support simulating CPS. After evaluating the requirements for those simulations in Section 4.1, a new interface for SimBricks is proposed in Section 4.2. Finally, the limitations of the proposed design are discussed in Section 4.3.

4.1 Requirements

In its current state, SimBricks focusses on components that are physically connected and implements the interfaces that those devices usually provide, splitting the simulation at natural boundaries. However, there is currently no interface to integrate a physics simulator which would be needed to fully support CPS. This is the environment where the simulated objects are placed, but since the physical environment is not directly "connected" to the digital components, there is not a single, well-defined connection which a SimBricks interface could mimic. Nevertheless, CPS mainly interact with their environment through specialized sensors and actuators that provide information about a certain property of the environment or affect certain properties of the environment. Thus, it makes sense to place the physical interface at the level of the sensors and actuators.

In line with the existing interfaces in SimBricks, this physical interface should be fully simulator-agnostic, i.e. it should be possible to switch the simulators without any modifications to the interface itself. Furthermore, the interface should not be specialized to a specific use case but rather be flexible in terms of sensor and actuator types that are supported.

Another important aspect is the ability to support multiple devices in the same physical environment that act independently but may influence each other's sensor reading (e.g. imagine a LIDAR sensor that should detect objects nearby). All of those devices in the physical simulation should be accessible through the physical interface. In case that there are multiple identical devices, there should be some way of addressing a specific device and a specific sensor or actuator.

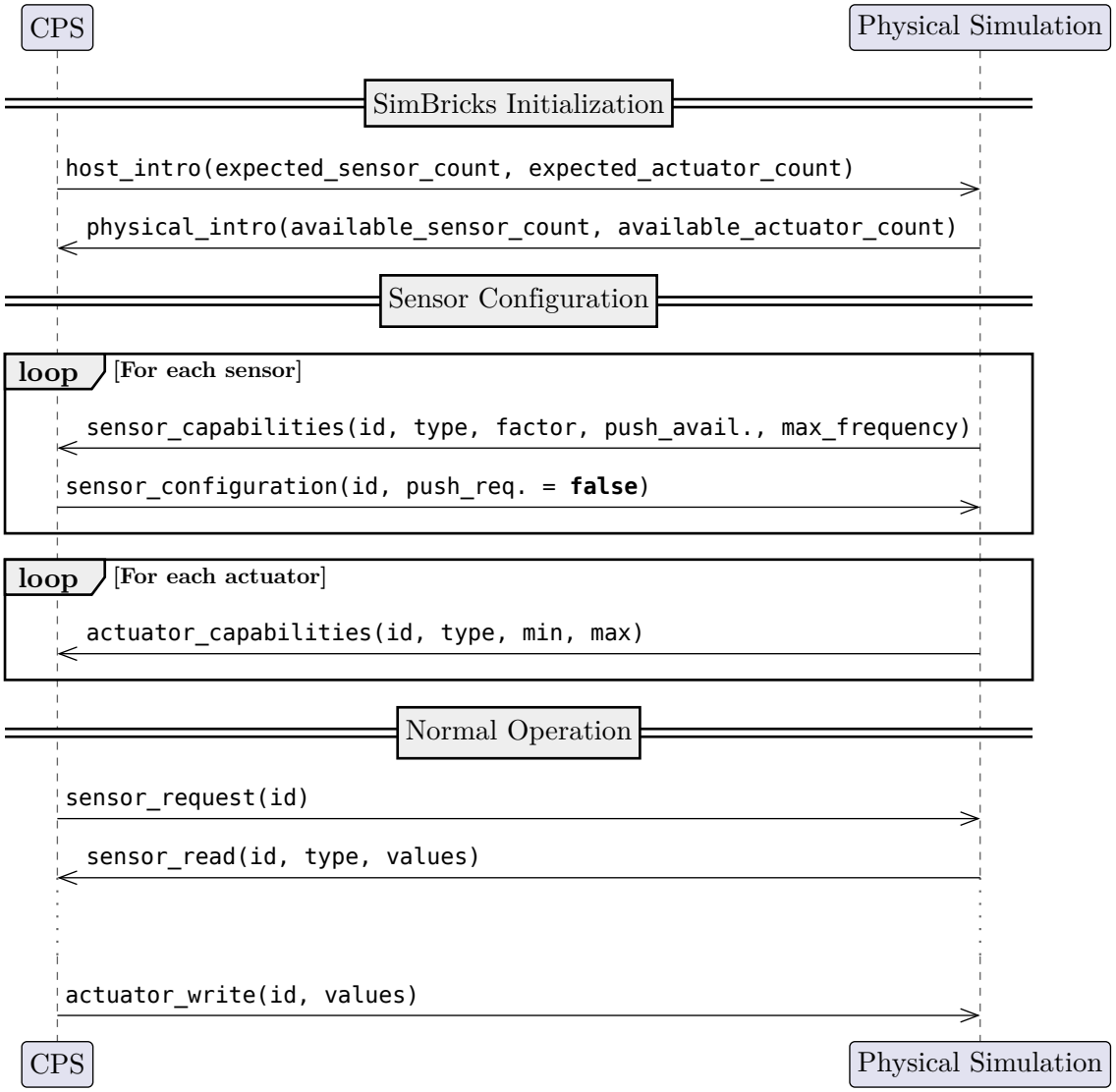


Figure 2: Messages sent during setup and normal operation of the physical interface with polling.

4.2 An Interface for Physical Simulators

All interfaces in SimBricks extend the `Base` interface. This already defines how a connection is established and how messages are kept in order. To extend the interface, one needs to define a protocol, i.e. which types of messages should be available and what the content of these messages should be.

For the physical interface, one important design choice is whether the sensor-side should poll for new data or if the simulator should push new sensor readings at certain intervals. Polling has some advantages when certain sensors allow to dynamically configure their update rate. In this case, one only needs to modify the rate of polling. However, polling might lead to more congestion on the interface

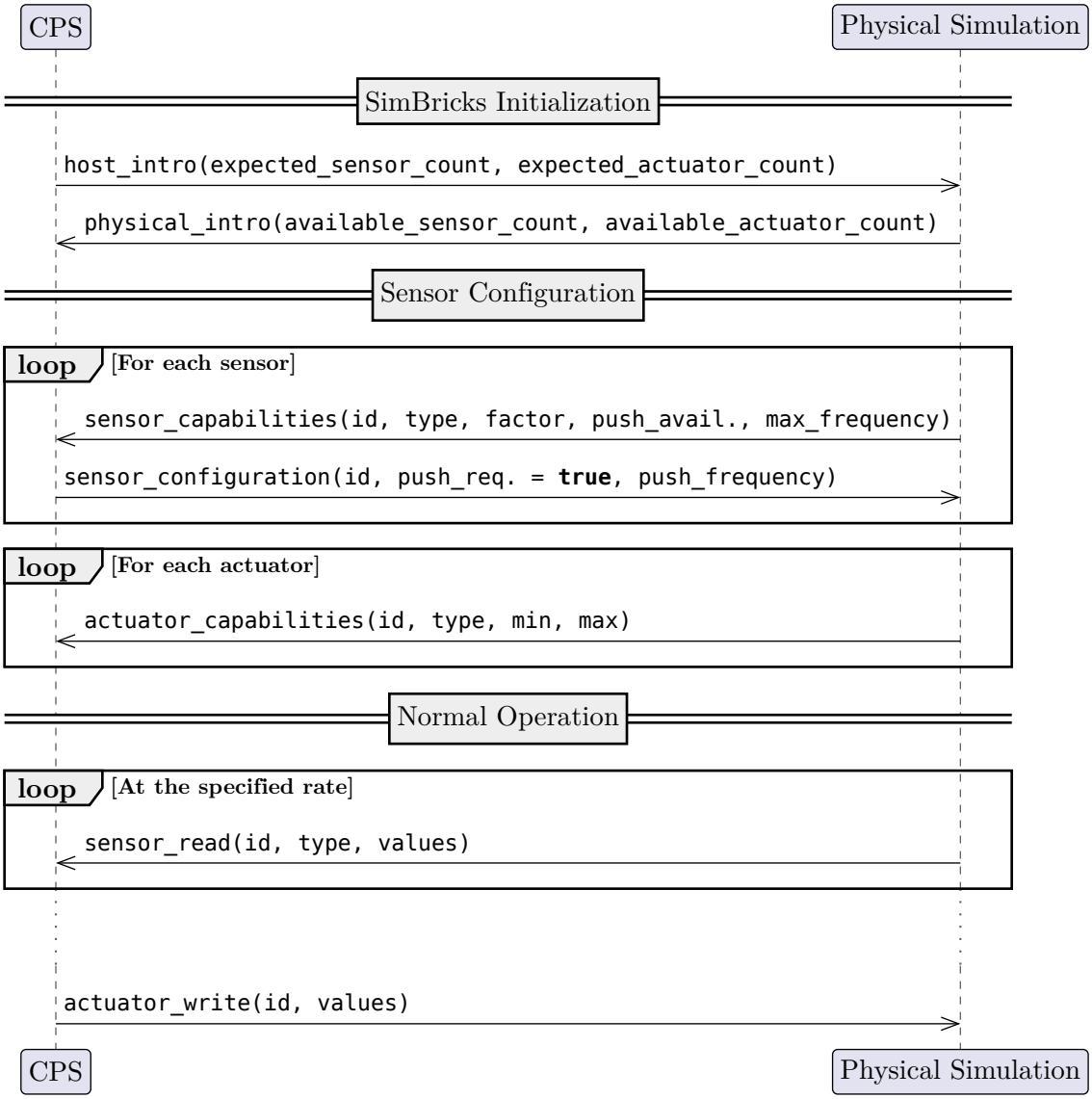


Figure 3: Messages sent during setup and normal operation of the physical interface with pushing

since for every new sensor read two messages need to be exchanged (the request and the response) while with pushing there is no need for those requests. Thus, it depends on the application which approach fits better. The physical interface therefore supports both pushing and polling, configurable during the initial handshake.

Figure 2 shows which messages are exchanged between the CPS and the physical simulation. First, both parties agree on the number of available sensors and actuators. The physical simulation then presents the sensors and actuators and their capabilities, e.g. their type and their value ranges. For sensors, the CPS now configures that the sensors are queried using polling, for actuators no further

configuration is required. During the normal operation, the CPS periodically (or as needed) sends a sensor read request and gets a response with the current sensor readings. For actuators, the CPS sends a write message every time the control values for this actuator need to be updated.

When using pushing for sensor readings, the initial phase is similar as shown in Figure 3. The only difference is that the CPS requests pushing when configuring a sensor. Additionally, it sets the push frequency, i.e. the frequency at which it expects sensor updates. After the initialization, the physical simulation now sends out the sensor readings at the requested rate without a separate read request.

4.3 Limitations

The proposed design allows to use SimBricks to connect a physical simulator to the already supported simulators. While this setup allows to simulate most aspects of CPS, there are still some limitations to consider.

First of all, there is no human in the loop. Thus devices that require manual inputs, e.g. joystick operated robots are hard to simulate in this way. While in theory it is possible to record and replay those inputs during the simulation, timing those inputs correctly may result difficult. Slight variations may already cause a different behavior which would require varying the input which is impossible to achieve with a static input. These applications are considered out-of-scope for this project and are left for future work. A possible solution to this would model the manual inputs depending on the current state, e.g. when moving towards a goal the inputs are chosen in such a way that the robot moves towards the goal.

The issue of slight variations may also arise when the emulated sensors are not modeled perfectly, e.g. depending on the sensor the datasheet might not specify at which precision certain computations are performed internally and slight rounding errors might propagate and lead to small variations in the emulated sensor readings. If necessary, validation of the emulator could be achieved by comparing the sensor readings from a physical sensor in a very controlled environment with the emulated readings. A recent article [28] proposes a methodology to validate sensor models using strategic experimental validation to perform a statistical validation. However, even with this, sensor signals are always affected by noise which is usually assumed to be additive gaussian noise. While it is possible to add noise in the simulation, in reality the signals will always be slightly different.

Furthermore, SimBricks allows to exchange messages of a fixed size. Thus, transmitting large amounts of data, e.g. a high resolution camera stream would come with a significant overhead and would likely slow down the simulation even more. To support transmitting large amounts of data, adding a secondary (possibly unidirectional) communication channel outside the SimBricks environment might yield the best performance.

The design from the previous section also considers sensors and actuators as the only point of interaction of the CPS with its environment. Generally speaking, many other interactions are possible that are not covered in this abstraction. For example the temperature of the environment could influence the clock frequency of the controller. In theory even rain could enter into some connectors creating short circuits or external radiation could cause corruptions of the internal state. Those influences are considered out of scope and would require modeling and simulating the system at a much higher fidelity.

Finally, as always for simulation, the simulation does not provide any guarantees about the correctness of the control logic or any other simulated component. Applications requiring a high reliability thus still need to be verified using formal proofs to avoid unwanted behavior in the field. However, simulations can help to justify certain assumptions that are required for the formal verification, e.g. by ensuring that certain specific scenarios work as expected.

5 Implementation

This chapter provides some details on how the design presented in the previous chapter has been implemented.

5.1 Protocol

The SimBricks `Base` interface already provides useful functionality shared between all interfaces. By extending this interface, the existing synchronization infrastructure as well as message allocation, ordering and destruction is already provided. Thus, what remains to do is to define the available message types and its contents as well as implementing the correct handlers on both sides. However, to simplify message handling, SimBricks requires all messages to have the same size of 64 bytes and certain fields to remain in specific positions. This results in a slightly less flexible structure for the messages. For compatibility reasons, all the values sent over this interface are stored as `int32_t`. Values that have a small range but require a higher precision (i.e. floating point values) are multiplied with a negotiated conversion factor and then converted from floating point to integer. On the receiving side, the original value is restored using the conversion factor.

5.1.1 The `intro` messages

Listing 3 shows the definition of the `intro` messages sent by the physical simulator (Listing 3a) and the host (Listing 3b). Both contain the `device_id` to ensure the communication channel is set up correctly by ensuring both parties agree on the device. Furthermore, the `available_sensor_count`/`available_actuator_count` combined with the `expected_senor_count`/`expected_actuator_count` provide a first sanity check to see if the devices match. Which sensor and actuator types are actually available and used is negotiated in the next phase.

```
1 struct SimbricksProtoPhysicalPhysicalIntro {
2   uint32_t device_id;
3   uint32_t available_sensor_count;
4   uint32_t available_actuator_count;
5 } __attribute__((packed));
```

(a) Intro message sent by physical simulator to host.

```
1 struct SimbricksProtoPhysicalHostIntro {
2   uint32_t device_id;
3   uint32_t expected_sensor_count;
4   uint32_t expected_actuator_count;
5 } __attribute__((packed))
```

(b) Intro message sent by host to physical simulator.

Listing 3: The `intro` messages sent via the physical interface upon initializing the connection.

```c
struct SimbricksProtoPhysicalP2HSensorCapabilities {
  uint64_t req_id;
  uint32_t device_id;
  uint32_t sensor_type;
  uint32_t sensor_id;
  uint32_t conversion_factor;
  uint32_t max_frequency;
  bool push_available;
  uint8_t pad[19];
  uint64_t timestamp;
  uint8_t pad_[7];
  uint8_t own_type;
} __attribute__((packed));
```

(a) Sensor capabilities message providing the sensor type and whether push functionality is available.

```c
struct SimbricksProtoPhysicalP2HActuatorCapabilities {
  uint64_t req_id;
  uint32_t device_id;
  uint32_t actuator_type;
  uint32_t actuator_id;
  int32_t min_value;
  int32_t max_value;
  uint8_t pad[20];
  uint64_t timestamp;
  uint8_t pad_[7];
  uint8_t own_type;
} __attribute__((packed));
```

(b) Actuator capabilities message providing the actuator type and the range of accepted values (`min_value` and `max_value`).

```c
struct SimbricksProtoPhysicalH2PSensorConfiguration {
  uint64_t req_id;
  uint32_t device_id;
  uint32_t sensor_type;
  uint32_t sensor_id;
  bool push_enabled;
  uint8_t pad[3];
  uint32_t frequency;
  uint8_t pad_[20];
  uint64_t timestamp;
  uint8_t pad__[7];
  uint8_t own_type;
} __attribute__((packed));
```

(c) Response sent by the host to configure pushing.

Listing 4: The configuration messages used to introduce the available sensors and actuators and to configure how sensor signals should be transmitted.

5.1.2 The Configuration Phase

During the second phase, the physical simulator sends details about the sensors using the format specified in Listing 4a and about the actuators (Listing 4b) it is simulating and the host has the possibility to enable pushing of sensor signals as well as configuring the frequency using its response (Listing 4c). The conversion factor specifies the multiplier that is used to convert the actual metric into an `int32_t` for transmission. As the number of sensors and actuators has already been negotiated in the initial phase, the number of messages sent during the second phase is fixed and thus allows for an easy implementation on both sides, especially if the received values are just used to verify the configuration.

5.1.3 Exchanging Values

Once the initial configuration is over and the simulation is running, updated sensor readings and actuator outputs need to be transferred between the physical simulation and the host. Depending on whether pushing was configured or not, the host now has to request certain values (Listing 5a) and the physical simulation

```
1  struct SimbricksProtoPhysicalH2PSensorReadRequest {
2    uint64_t req_id;
3    uint32_t device_id;
4    uint32_t sensor_type;
5    uint32_t sensor_id;
6    uint8_t pad[28];
7    uint64_t timestamp;
8    uint8_t pad_[7];
9    uint8_t own_type;
10 } __attribute__((packed));
```

(a) Request to send the latest readings of a specific sensor.

```
1  struct SimbricksProtoPhysicalP2HSensorRead {
2    uint64_t req_id;
3    uint32_t device_id;
4    uint32_t sensor_type;
5    uint32_t sensor_id;
6    int32_t val_1;
7    int32_t val_2;
8    int32_t val_3;
9    int32_t val_4;
10   int32_t val_5;
11   int32_t val_6;
12   int32_t val_7;
13   uint64_t timestamp;
14   int32_t val_8;
15   uint8_t pad_[3];
16   uint8_t own_type;
17 } __attribute__((packed));
```

(b) Latest sensor readings of the specified sensor.

```
1  struct SimbricksProtoPhysicalH2PActuatorWrite {
2    uint64_t req_id;
3    uint32_t device_id;
4    uint32_t actuator_type;
5    uint32_t actuator_id;
6    int32_t val_1;
7    int32_t val_2;
8    int32_t val_3;
9    int32_t val_4;
10   int32_t val_5;
11   int32_t val_6;
12   int32_t val_7;
13   uint64_t timestamp;
14   int32_t val_8;
15   uint8_t pad_[3];
16   uint8_t own_type;
17 } __attribute__((packed));
```

(c) Request to set the actuator.

Listing 5: The messages used to exchange data on the physical interface. The generic `val_1` - `val_8` may hold different values depending on the specific sensor or actuator type.

then responds with the current readings or the simulation will just provide updated values at certain intervals (Listing 5b). For the actuators a different message is sent to update the current outputs (Listing 5c). While no message is sent, the outputs are assumed to remain unchanged.

As explained above, all values are converted to `int32_t` and due to the restricted size of SimBricks messages, up to 32 bytes (i.e. eight 32 bit values) of data can be transferred in one message. Having just one message type for all readings and one message type for any write allows to flexibly add support for different metrics without changing SimBricks core library. By default, the interface defines 9 different metrics and the corresponding units in which values should be exchanged. An overview can be found in Table 1.

5.2 Gazebo Integration

Gazebo provides a flexible interface for plugins to interact with the simulation. This even allows stalling the simulation for an extended period of time, e.g. until the host has caught up. To add a plugin to a simulation it is as easy as adding the plugin to the `sdf` of the currently simulated world (as shown in Listing 1, Line 8).

Metric	**ID**	**Response**	**Unit**	**Default Factor**
Altitude	`0x01`	altitude	m	10^2
Temperature	`0x02`	temperature	K	10^6
Position	`0x03`	$\begin{bmatrix} \text{latitude} \\ \text{longitude} \\ \text{altitude} \end{bmatrix}$	$\begin{bmatrix} ° \\ ° \\ m \end{bmatrix}$	10^6
Ground Velocity	`0x04`	velocity	$\frac{m}{s}$	10^3
Heading	`0x05`	heading	°	10^3
Orientation	`0x06`	$\begin{bmatrix} \text{w} \\ \text{x} \\ \text{y} \\ \text{z} \end{bmatrix}$	$\mathbb{H}$[1]	10^6
Angular Velocity	`0x07`	$\begin{bmatrix} \text{x} \\ \text{y} \\ \text{z} \end{bmatrix}$	$\begin{bmatrix} \frac{\text{rad}}{s} \\ \frac{\text{rad}}{s} \\ \frac{\text{rad}}{s} \end{bmatrix}$	10^6
Acceleration	`0x08`	$\begin{bmatrix} \text{x} \\ \text{y} \\ \text{z} \end{bmatrix}$	$\begin{bmatrix} \frac{m}{s^2} \\ \frac{m}{s^2} \\ \frac{m}{s^2} \end{bmatrix}$	10^6
Magnetic Field	`0x09`	$\begin{bmatrix} \text{x} \\ \text{y} \\ \text{z} \end{bmatrix}$	$\begin{bmatrix} T \\ T \\ T \end{bmatrix}$	10^9

Table 1: Predefined metrics for the physical interface with their respective `ID`, unit and the default conversion factor.

Optional arguments are passed to the plugin, allowing to configure the plugin directly from the `sdf` file.

Plugins can act at different levels, e.g. some may be specific to one simulated entity while others affect the whole simulation. To integrate SimBricks, a plugin at the `system` level is required. These plugins need to implement three methods:

- `Configure` is called once during the setup of the simulation and allows to make changes to the model before starting the simulation.
- `PreUpdate` is called at some point before the simulation advances to the next interval allowing to adjust actuators in time for the next computation.
- `PostUpdate` is called after performing a simulation step.

Furthermore, Gazebo uses a publisher-subscriber system to communicate. New sensor readings are published under a certain topic and actuators can be configured by publishing messages on their respective topics. Thus, during the `Configure`

[1] $\mathbb{H}$ stands for a rotation quaternion which represents a rotation using a four-dimensional vector normalized to a length of 1[29].

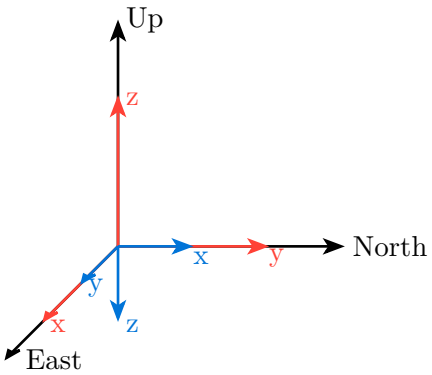


Figure 4: Gazebo's ENU coordinate system vs. ArduPilot's NED system .

phase, the SimBricks plugin subscribes to all relevant sensors and then receives a callback once the sensor values update. Similarly, in order to apply actuator writings, the plugin will publish a message under the respective topics containing the new control value, e.g. the force to be applied to a joint).

There is one caveat when working with Gazebo and ArduPilot. Gazebo uses the East-North-Up (ENU) coordinate system where x points east and z upwards (red in Figure 4) and ArduPilot uses the North-East-Down (NED) coordinate system (blue in Figure 4). The rotational direction is - as per convention - according to the right hand rule, i.e. a rotation is positive when rotating from east to north around an axis facing upwards. In the current implementation, those conversions are done before sending the values over the physical interface, thus all transmitted values are in the NED coordinate system.

5.3 gem5 Integration

To simulate the control logic, the gem5 simulator provides a detailed simulation environment for various CPU types and is already supported in the SimBricks environment. It serves as one of the standard choices to simulate applications in a Linux environment. ArduPilot provides various HAL, one of which supports Linux hosts. Thus, executing ArduPilot inside gem5 running the provided Linux image does not require any modifications to existing SimBricks simulators. However, depending on the configuration, various external devices (sensors and actuators) need to be available to successfully start ArduPilot.

Typically, sensors and actuators are connected via serial interfaces such as Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C) or Universal Asynchronous Receiver Transmitter (UART). Linux provides access to those

```
class Device
{
    public:
        virtual ~Device() {}

        virtual uint32_t getDeviceID() = 0;
        virtual size_t getBase() = 0;
        virtual size_t getMMIOSize() = 0;

        virtual void processSensorReading(volatile struct SimbricksProtoPhysicalP2HSensorRead *msg) = 0;
        virtual size_t processMemoryReadRequest(size_t offset, size_t size, void** data) = 0;
        virtual size_t processMemoryWriteRequest(size_t offset, size_t size, volatile void* data) = 0;

        virtual std::pair<uint64_t, std::list<uint8_t>> nextDeviceReadRequestTime() = 0;
        virtual void deviceReadRequested(uint64_t timestamp) = 0;

        virtual bool hasActuatorWriteMessage(uint64_t timestamp) = 0;
        virtual bool getActuatorWriteMessage(
          uint64_t timestamp,
          volatile SimbricksProtoPhysicalH2PActuatorWrite* msg
        ) = 0;

        virtual bool restoreCheckpoint() = 0;
};

```

Listing 6: The interface implemented by the emulators of sensors and actuators.

devices via the device tree, i.e. serial devices are available as e.g. `/dev/serial0` or `/dev/spidev0.0`. Those are special files that support the `ioctl` system call that allows to send or receive data from a serial device.

For this project, those devices are registered using a custom kernel module that provides implementations of the necessary system calls and redirects them via the SimBricks memory interface to an emulator for those devices. While gem5 does provide support for programmed I/O devices directly, due to a conflicting address range of the gem5 configuration and ArduPilot's configuration, this approach turned out to work better in this case.

The emulator receives the requests from the host and simultaneously connects to Gazebo to keep the sensor readings updated. From those readings, the appropriate control register values are computed and provided to the host upon request.

The emulator currently supports the following sensors:

- The MS5611 digital pressure and altimeter sensor[30]
- The MPU-9250 IMU sensor[31]
- The AK-8963 magnetometer (integrated with the MPU-9250)[32]
- A generic NMEA GNSS sensor[33]

As actuator the emulator supports any PWM controlled device.

While those sensors are not enough to simulate any CPS, given a detailed datasheet describing the interface and the value's encodings, integrating more sensors is straightforward. The emulator provides an interface for Devices (Listing 6) to implement that allows to react to inputs from either the host (`processMemoryReadRequest`, `processMemoryWriteRequest`) or the physical (`processSensorReading`) side, to request sensor readings (`nextDeviceReadRequestTime`) and to send messages to the physical simulation (`getActuatorWriteMessage`). The sensor's implementation then only needs to keep track of the internal state of the sensor. The complexity of this state largely depends on the specific sensor.

For example, for the MS5611 the temperature and pressure are calculated using six calibration values (C_1 - C_6) provided by the factory. As shown in [30, Figure 2], the temperature t is calculated using this formula:

$$t = 2000 + (D_2 - C_5 * 2^8) * \frac{C_6}{2^{23}}$$

where D_2 is one of the two values read from the sensor. To emulate the sensor, D_2 needs to be calculated for a given temperature, i.e.

$$D_2 = \left((t - 2000) * \frac{2^{23}}{C_6} \right) + C_5 * 2^8$$

The emulator uses the calibration values provided as example in the datasheet.

6 Evaluation

To evaluate the approach presented in the previous chapters, multiple experiments have been conducted. The setup of those experiments is described in Section 6.1. Using this setup, the following research questions have been analyzed:

RQ1: Can SimBricks be used to evaluate different hardware configurations of a CPS? (Section 6.2)

RQ2: Is this approach able to detect bugs that are out-of-scope for existing simulators? (Section 6.3)

RQ3: How does this approach scale to multiple simulated CPS? (Section 6.4)

RQ4: How does the slowdown compare to the existing ArduPilot SITL simulator? (Section 6.4)

RQ5: What is the required effort to set up a simulation for a specific CPS? (Section 6.5)

6.1 Setup

An overview of the setup used in the following evaluation can be found in Figure 5. ArduPilot is running on a gem5 host with the default Linux configuration provided by SimBricks. The connection to the network is established using the `i40e_bm` which is one of the provided Network Interface Cards (NICs) in SimBricks. The connection to the network is established using an ethernet interface. This setup does not match a realistic setup since the drone is likely to be wirelessly connected to the ground station with a much lower bandwidth. However, evaluating the

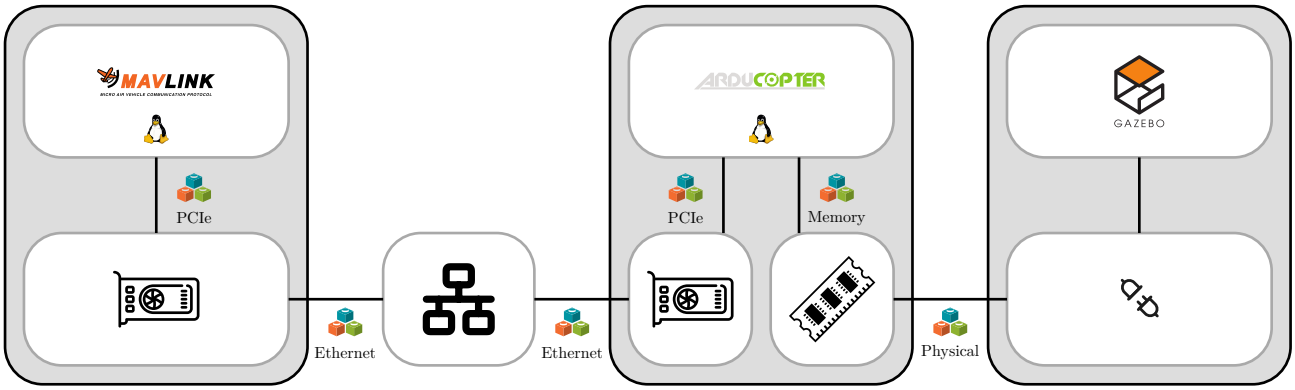


Figure 5: Experimental setup in SimBricks with one ground control station and one ArduPilot host, connected via the network. The ArduPilot instance is connected to the physical simulation in Gazebo.

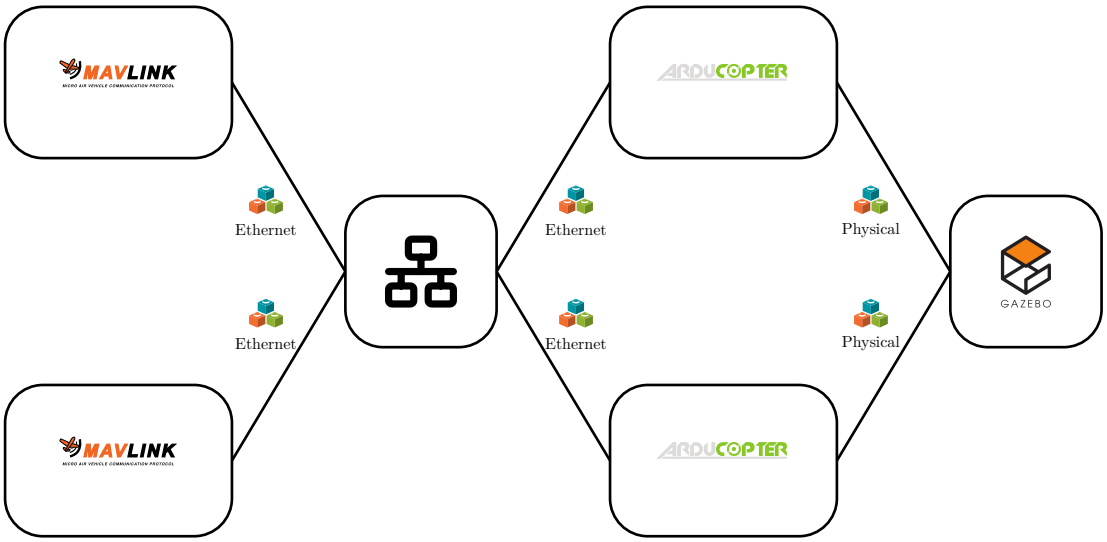


Figure 6: Experimental setup in SimBricks with two ground control stations and two ArduPilot hosts. Both instances are connected via a shared network simulation and are simulated in a shared physical environment in Gazebo.

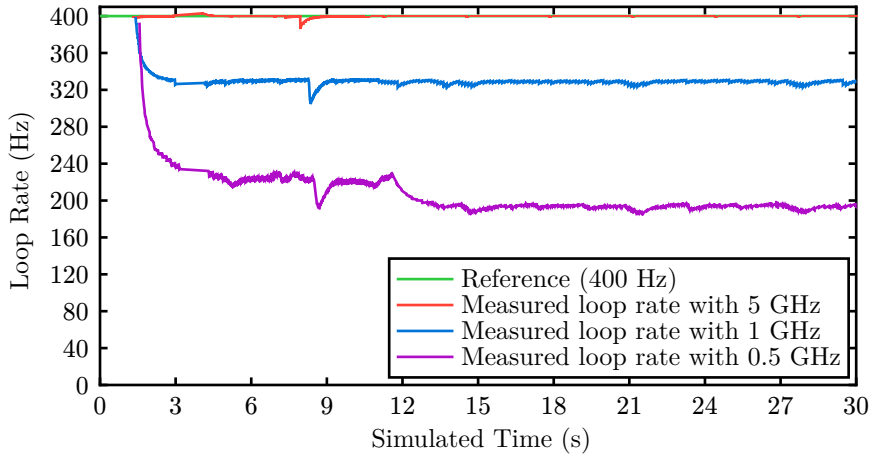
network and the connection to the ground station is left for future work and could easily be integrated given an appropriate wireless NIC simulator in SimBricks. The ground station follows a similar setup with a gem5 Linux node connected via an `i40e_bm` to the network simulator. On this host, a simple ground control station partially implementing the MavLink protocol is listening for messages from ArduPilot and responding with appropriate commands (e.g. `arm`, `takeoff` etc.). On the other side, the ArduPilot host is connected to Gazebo through a memory device emulating the sensors and actuators and providing the correct file handlers for Memory-Mapped IO (MMIO). This emulator connects to the Gazebo plugin to gain access to the physical simulation.

For experiments involving multiple drones, the ground control station and the drone controller have been replicated for each simulated device. They are connected to the same network and to the same physics simulator using additional SimBricks interface instances as shown in figure Figure 6 for two instances.

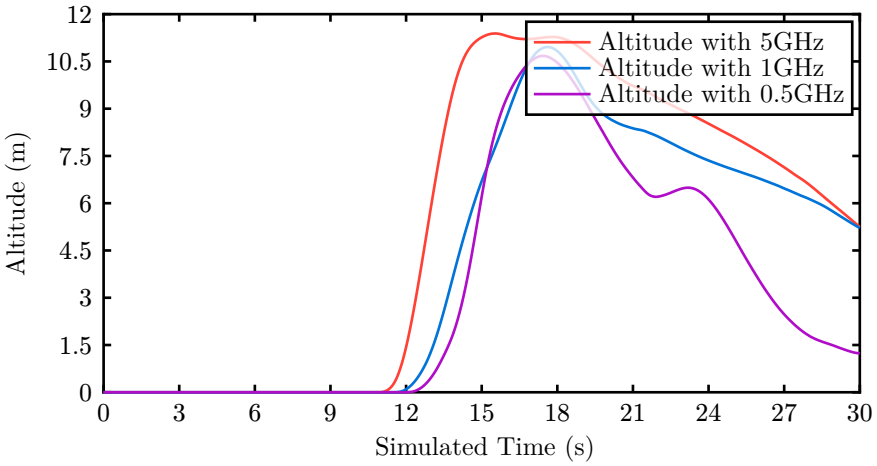
All experiments have been executed on an AMD Threadripper PRO 5995WX with 64 cores (128 threads) and 512GB of RAM.

6.2 RQ1: Evaluating Hardware Configurations

One big advantage of a full-system simulation over simulations with a lower fidelity is that full-system simulators allow to precisely model the hardware which



(a) Loop rate of ArduPilot's main control loop.



(b) Altitude over ground of the simulated drone.

Figure 7: The impact of different CPU frequencies on ArduPilot's loop rate and the observable behavior of the drone.

eventually is to be used in the field. In the setup described in Section 6.1, this permits evaluating the influence of different CPU models or the impact of lower network bandwidth or higher sensor noise caused by different (and most likely cheaper) hardware modules used in the setup.

A recent study [34] has shown that it is possible to tune gem5 CPU models such that their performance closely matches modern CPUs like the `arm Cortex-R8`. For simplicity, for this evaluation only different CPU frequencies for the `TimingSimpleCPU` were considered without a specific microarchitecture in mind.

ArduPilot has one main control loop which ideally runs at 400Hz. In each iteration, a few high-priority tasks and a subset of lower priority tasks are executed.

After each loop, the actual loop duration is used to update the loop rate in order to slowly settle onto a loop rate that allows executing all the tasks:

$$\text{loop\_rate}(t) = \frac{1}{0.99 * \frac{1}{\text{loop\_rate}(t-1)} + 0.01 * \text{last\_loop\_duration}(t)}$$

If the CPU is not fast enough to run ArduPilot, the loop rate will settle below the desired 400Hz. There is a runtime check before arming the rotors to ensure that the loop rate is below 90% of the desired frequency. This check was disabled for this experiment to show the effect of insufficient hardware configurations.

Figure 7a shows how the loop rate evolves over time with different CPU frequencies. Note that these numbers have been obtained on the `TimingSimpleCPU` model which has very few microarchitectural optimizations and thus does not directly compare to modern CPUs. At 5Ghz the CPU is fast enough to maintain a loop rate of 400Hz while at lower frequencies the loop rate drops significantly below the desired 400Hz. This has noticeable effects on the flight behavior as shown in Figure 7b. At a lower loop rate, updates are not processed as promptly, thus changes cannot be mitigated as quickly what may lead to over-corrections, e.g. around second 23.

One of the analyzed bug fixes in [27] commit `52c4715c` was classified as non-detectable using simulation. This bug affected systems with a low memory configuration. When running ArduPilot in gem5, the available memory can be freely adjusted thus allowing to detect this bug.

6.3 RQ2: Detecting Previously Undetectable Bugs

Another shortcoming of existing simulators with lower fidelity is that they usually introduce a new abstraction layer for simulation purposes. Since the sensors are not physically available in the simulation, simulated measurements are usually provided directly to the controller without involving the device drivers which map the measured values to their actual physical meaning.

This limits the ability of the simulation to detect bugs in those device drivers. For example, the error fixed commit `facd8fc` can only be detected when the GPS sensor fails to provide a 3D fix and only provides a 2D fix which is currently not possible in the existing SITL simulator. Therefore, it was classified as non-detectable in [27]. In contrast, the sensors emulated in this project can be easily extended to provide GPS readings of various qualities at certain points in time, e.g. limiting the number of satellites in use to only provide a 2D fix. This can be

achieved by adapting the emulator of the GNSS sensor to change the quality of the signal based on certain circumstances, e.g. the simulated time, the current position or other sensors signals. In the NMEA standard[33] for example, messages include the number of satellites in use and the `GSA` sentence allow to specify whether the current fix is 2D or 3D.

Another example where this can be useful is to detect the bug fixed in commit `28b98a17`. Here, intermittent failures of the compass can trigger an unexpected landing even if the individual failures only last a small amount of time. To detect failures of the compass, a simple counter was used which failed to reset once the compass returned to a healthy state. Thus after several independent and short failures, an emergency landing would be triggered. The SITL emulator does not easily allow to specify various short intervals in which a sensor should fail, a sensor can either be simulated correctly or fail completely. The emulator however allows to simulate transient failures as required to observe the effects of this bug. Again, the emulator could introduce these failures based on the simulated time or other sensors values or randomly with a certain probability.

The patch from commit `36634265` changes how the groundspeed undershoot is calculated. Previously, this data was taken directly from the GNSS system which does not account for movements due to wind, i.e. if the acceleration due to the wind and due to the motors cancel out, the plane will not move according to a GNSS system. However, when using the groundspeed provided by the Attitude and Heading Reference System (AHRS) instead, this can be avoided. To detect this in simulation, the wind needs to be constantly tuned to match the current acceleration of the UAV. While this is not possible in the SITL simulator, the presented approach could be configured to account for this allowing to expose this bug. In such a configuration, the Gazebo plugin could activate the windy conditions in the physical simulation, again as desired based on the simulation time or some other property of the state of the simulation like certain sensor signals.

Generally speaking, the presented approach provides a flexible way to simulate transient and permanent faults in the emulated devices and allows to dynamically adjust the environmental parameters at runtime. While randomly inserting faults is unlikely to reveal the mentioned issues due to the specific requirements to trigger the faulty behavior, this approach might be useful to avoid regressions by adding specific tests for those scenarios. Apart from that, this methodology might be

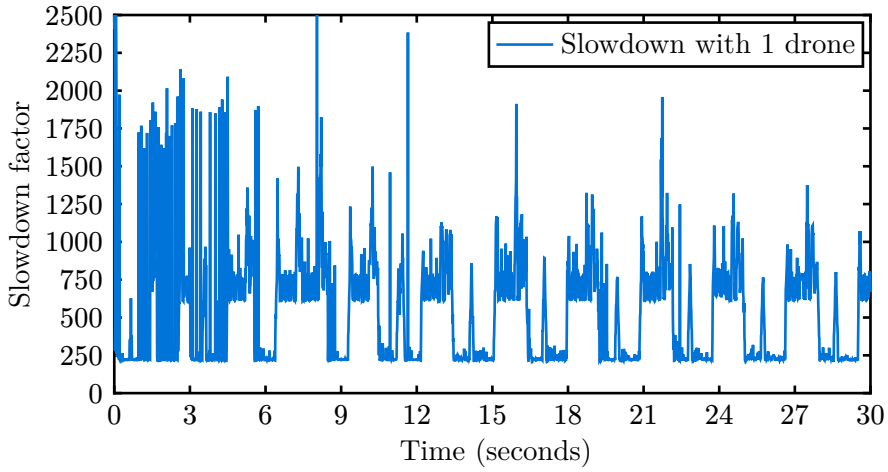


Figure 8: The slowdown factor over time for the experiment shown in Figure 5, sampled at intervals of 1ms.

useful to reproduce behavior reported by users that would be difficult to reproduce in reality due to very specific environmental requirements.

6.4 RQ3 & RQ4: Performance & Scalability

Simulating complex systems at a higher fidelity comes at a cost. One important number to consider is the slowdown factor of the simulation, i.e. the amount of real time it takes to simulate one time interval in the simulation. For example, a slowdown factor of 600 means that for each simulated second, the simulation will take 10 minutes. The slowdown factor determines if certain tests are feasible and how many different tests can be conducted within the required timeframe.

Synchronized simulations in SimBricks are generally slower than other types of simulation since all components are simulated separately and due to synchronization, the slowest simulator determines the simulation speed. On the other hand, each simulator runs in its own process, thus given enough CPU cores, adding more simulators to a simulation does not necessarily slow down the entire simulation.

This slowdown factor is obtained by analyzing the progress of the physical simulation. A separate logger subscribes to the `/clock` topic of the Gazebo simulation and records both the timestamp on the host and the simulated time in Gazebo. The average slowdown is then computed using:

$$\text{average\_slowdown} = \frac{t_{\text{end,host}} - t_{\text{start,host}}}{t_{\text{end, simulation}} - t_{\text{start,simulation}}}$$

With one simulated drone, the average slowdown factor of the setup in Figure 5 is ~500x. As shown in Figure 8, the slowdown factor is not constant but changes over time depending on the necessary computations. The slowdown can get as low as ~250x and as high as ~2500x when sampling at intervals of 1ms. This highlights the importance of constant synchronization as used in SimBricks instead of reducing the simulation speed of certain components to roughly match the speed of the slowest simulator (using the average slowdown over the entire simulation). Although this approach might appear a good option to reduce the overall slowdown and to avoid implementing proper synchronization mechanisms, doing so might cause large differences in the local timestamp of two connected simulators even though the local clocks will eventually converge to the same number.

Unfortunately, SimBricks does not provide any tracing mechanisms to record the messages exchanged on each interface or to visualize the progress of each simulator which would be useful to analyze the bottlenecks of this approach. In this setup, the device emulator has the advantage that it is both connected to the host simulation and the physical simulation and thus can be used for some basic profiling. Before advancing its local clock, the device emulator needs to wait until both interfaces allow to proceed further. By recording which interface advances first and how many more (host) cycles are needed until the other interface proceeds, it is possible to see which side of the simulation slows down the entire system most. For one drone about 27% of the wait time is spent on waiting for the physical interface (Gazebo) while 73% of the time, the delay can be attributed to the host side (gem5 with ArduPilot, the network simulator and the simulation of the ground control station). This indicates that the simulation speed is limited by the software simulation.

Figure 9 shows how the slowdown changes when multiple drones are simulated as shown in Figure 6. This experiment indicates that the slowdown is linear in the number of simulated drones. Again, looking at the wait time of the device emulator allows to break down which part of the simulation causes most delays. Figure 10 summarizes this result for all tested configurations. In contrast to the results from above for one drone, for two drones 67% of the wait time can be attributed to the physical simulation while only 33% of the delay is due to the host simulation. The latter decreases even further with more simulated drones. This indicates that the physical simulator is now slowing down the simulation due to the higher complexity of multiple simulated drones since all devices are simulated

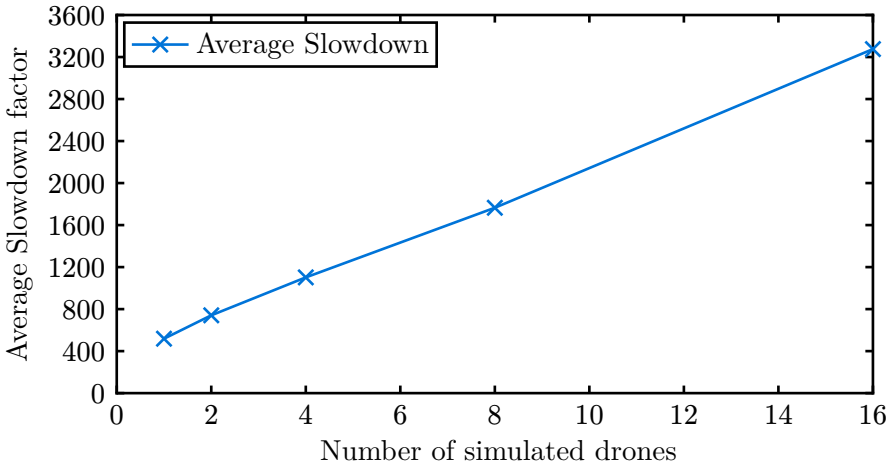


Figure 9: The average slowdown factor for various configurations with different numbers of simulated CPS configured as shown in Figure 6.

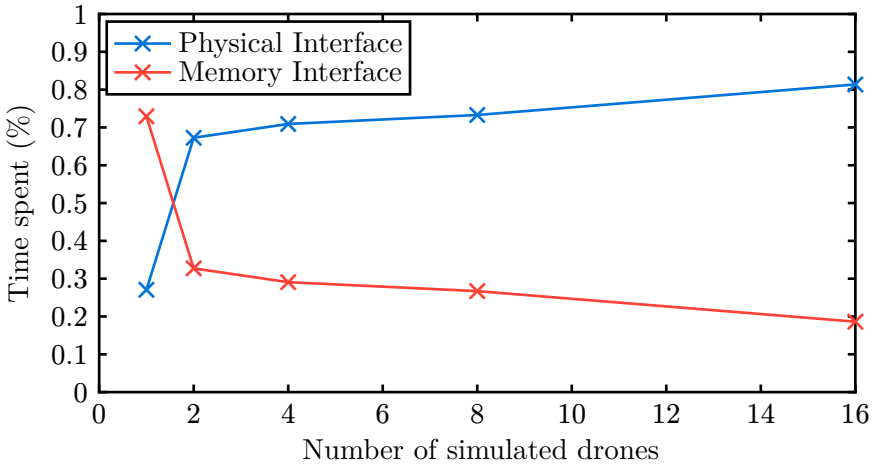


Figure 10: Percentage of time spent waiting in the device emulator for the physical interface and the memory interface to proceed (relative to the total wait time) with different numbers of simulated CPS.

in one physical simulation. Gazebo currently does not provide a multithreaded physical simulation since the dynamics simulator ODE[24] relies on singletons that are not thread-safe[35]. Thus, adding more drones to the simulation decreases the simulation speed since more calculations need to be completed for every simulated time interval.

In comparison with the existing integration of Gazebo into ArduPilot's SITL simulator[36] which does not provide synchronization between the simulated components, the slowdown is (as expected) generally much lower. However, when increasing the number of simulated devices, the slowdown quickly starts to rise even at a higher rate from a slowdown factor of ~4x with one drone to ~580x with

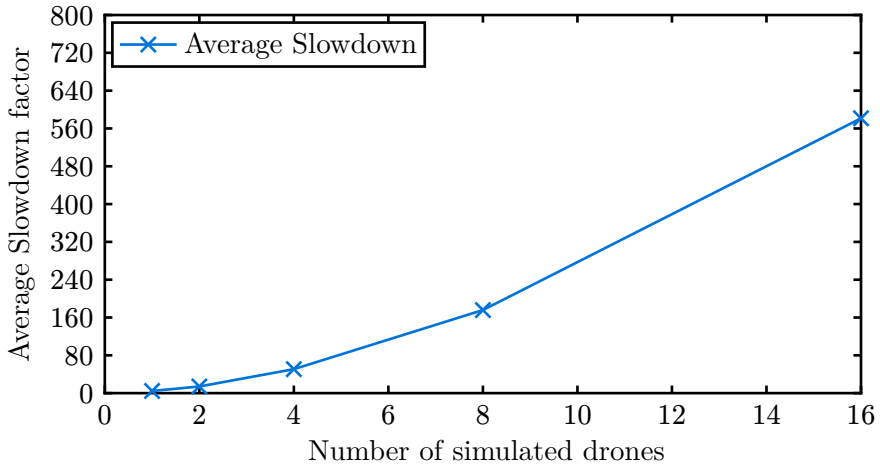


Figure 11: Average slowdown factor for experiments with different numbers of simulated CPS, simulated using the existing ArduPilot SITL simulator and the existing Gazebo plugin[36].

16 drones. Due to the unsynchronized simulation the connection between the SITL driver and Gazebo also frequently times out when simulating multiple drones. If the SITL simulator does not receive at least one message within a certain time interval (real time on the simulation host), the connection times out, even though the physical simulation was just too slow.

6.5 RQ5: Implementation Effort

Several components had to be implemented to do the above evaluation. Some of those implementations are independent of the simulated CPS while other parts were specifically designed to support ArduPilot and the simulated drone in Gazebo.

Table 2 summarizes the size of each component in terms of lines of code. The numbers were obtained using Cloc[37]. The definition of the physical interface should not need to be modified to support other CPS. As described in Chapter 4, the interface was designed such that it supports a large variety of sensors and actuators without any modifications.

The Gazebo plugin and the custom kernel module to make the emulated devices available in gem5 contain some elements that are specific to simulations with ArduPilot, however the overall structure could serve as a basis for future work integrating other types of CPS.

Out of all components, the device emulator requires most implementation effort to support additional hardware. As described in Section 5.3, the current

Component	Programming Language	Lines of Code
Physical Interface Definition	C++	182
Gazebo Plugin	C++	868
Custom Kernel Module	C	547
Device Emulator	C++	2778
Drone Definition[2]	XML/SDF	794
World Definition	XML/SDF	174
Simulator Definition	Python	98
Experiment Script	Python	92
Gazebo Logger	C++	331

Table 2: Lines of code required to implement the evaluated setup from Chapter 6.

implementation provides an interface to easily integrate additional sensors or actuators, however, the main effort consists of correctly implementing every control register according to the specification and ensuring that the required configurations are correctly supported.

On the Gazebo side, each experiment requires the definition of the world and the simulated models. The model for the simulated drone is a slightly adapted version of the `iris_with_gimbal` model from [36]. The world definition then only need to take into account how many drones should be placed where in the simulated world. The complexity of these definitions depends largely on the desired level of detail. Since in this project no comparison with any commercially available drone was made, there was no need for a more detailed model of the CPS. In general, a model is required in other approaches as well. This can be either a mathematical model or a description in a format like SDF (as it is the case for the ArduPilot Gazebo plugin [36]).

Apart from that, a Python class for every simulated entity is required to define the available options and how they translate to command line arguments for the components. Those classes are then used in the experiment script to instantiate the simulators as shown in Listing 1. This code is tailored to the specific setup and thus needs to be adapted for each experiment.

To obtain the slowdown in Section 6.4, an additional logger has been implemented to log relevant data at runtime. This logger is however not required for the simulation.

[2]excluding visuals, slightly adapted from the ArduPilot Gazebo plugin[36]

7 Related Work

Various simulation techniques have been proposed to examine CPS, some focussing more on the control logic while others focus on the physical simulation.

TrueTime[38] is an early example that focusses on embedded control systems modeled fully in Matlab/Simulink. Their work introduces computer blocks in the Simulink model which allow executing C++ functions like interrupt handlers. However, implementing a complex controller like ArduPilot would take a significant amount of work and might introduce inconsistencies.

Mambo[39] on the other hand focusses on accurately simulating PowerPC hardware and included support for UART devices. This simulator provides a graphical interface to debug at the OS level. Unfortunately, the paper does not provide any details on how the UART interface is implemented and whether a similar approach could be used to interactively simulate CPS.

More recent work on CPS-Sim[40] integrated Matlab/Simulink with a network simulator allowing for more precise time synchronization compared to previous work. This approach is mostly focussed on using the network as communication medium between CPS, integration into larger setups with heterogeneous simulators (e.g. to also simulate the ground control station) is currently not possible in this framework.

RoSÉ[41] provides an integration of AirSim[42] with FireSim[43] to evaluate domain-specific accelerators for robotics in simulation. Sensor data of a UAV is provided by AirSim and transferred to FireSim similar to the approach presented in the last chapters. As this paper does not propose a standardized interface, it would be harder to integrate the simulation of other components like the network.

The commercially available Simics[44] simulator does allow devices to be connected via serial ports. However, it currently lacks an integration of a physics simulator and thus in its current state is not able to effectively simulate CPS.

Apart from those general-purpose simulators that (partially) support CPS or other external devices, there are also some specialized simulators for different domains. For simulating UAVs, there is FlyNetSim[45] a simulation platform that focusses on the network communication between UAVs. Realistic flight simulation is also important in pilot training[46] or for video games where users can control different kinds of aircraft like Microsoft Flight Simulator[47] or X-Plane[48]. How-

ever, in the latter simulating visually at a high quality is usually more important than accuracy.

Regarding the emulation of sensors there is MIXED-SENSE[49] where the authors aim to recreate signals including latencies and noise. This approach then allows to emulate physical attacks on CPS such as replay attacks for GNSS signals. There are also various articles focussing on the simulation of IMU sensors [50, 51] for a given trajectory. This exceeds the requirements of this project as Gazebo already provides the IMU data and the emulator then needs to convert it into a sensor-specific format. As mentioned in Section 4.3, [28] proposes an approach to validate those models.

8 Discussion and Future Work

The results presented in the previous chapters highlight both the strengths and that limitations of the proposed framework. While it enables high-fidelity simulation for CPS, several challenges remain, in particular regarding scalability and accuracy. This chapter outlines possible directions for future work that could help address these limitations and broaden the applicability of the presented approach.

8.1 Improving the performance of the physical simulation

The evaluation section has revealed that the physical simulation does not scale perfectly to a large number of devices. However, to apply this approach to some other areas like vehicle-to-infrastructure communication or factory modeling, the concurrent simulation of many different entities inside one environment is required. There are various options on how future work might help to improve the scalability of the physical simulation.

One observation is that some entities only need to be modeled to allow measuring their influence on other entities and the evaluation focusses on a few main devices. In this case good results might still be obtained even if those surrounding entities are modeled with a lower precision. Such mixed-fidelity simulations might allow to decrease the amount of computations required in each step of the physical simulation allowing to improve the overall simulation time.

Additionally, it might be possible to separate the environment into blocks in which the simulated entities can be assumed to act independently. Using this approach it might be possible to split the simulation into various fragments allowing for more parallelism. Though, it is not obvious how those blocks could be determined and how the transition of an object between two of those blocks might look like.

A similar approach was implemented in earlier versions of Gazebo [52]. Thus, it might be possible that this feature is added back in a future release. There are two recent issues ([53], [54]) requesting improvements in this area, however, the development team has not yet detailed whether this is feasible and planned to add in the future.

Currently not only the physical simulation is constraining the feasibility of larger-scale simulations. To maximize the simulation speed, SimBricks interfaces constantly poll for new messages, i.e. even simulators that do not need to complete

complex computations fully use one core of the host system. As soon as the number of simulated devices exceeds the number of available cores, the simulation speed will reduce drastically since a lot of context switches will be required and the scheduler cannot distinguish simulators that are waiting for other simulators to catch up from those lagging behind the other simulators. In order to simulate an environment with many components, the number of processes should therefore be limited. It might be possible to co-locate the simulation of various simple components in one process, especially if those components only require a behavioral simulation. This could allow to reduce the number of required cores to permit simulating more complex systems with reasonable hardware requirements.

8.2 Simulating external conditions

Gazebo allows to specify various external factors like wind or rain using simulated fluids. In order to precisely model the impact of those conditions, the model of the CPS might need to be refined to capture e.g. how the surface's friction changes when wet or whether the weight of the device changes in rainy conditions. Some existing simulators already allow to simulate those conditions [55] with a lower fidelity, however, the missing synchronization does not allow to vary those conditions depending on the current simulation time. It is left to future work to explore the effects of environmental factors on the simulated devices using the presented approach.

8.3 Validating the accuracy of the simulation

One question left open in this project is the validation of the simulation results. In order to draw reliable conclusions from the simulation results, one must have a certain confidence that the simulation results are accurate and match the behavior of a physical testbench. However, such a testbench might not always be available. Even if such a testbench is available, reproducing the results from the simulation might be hard since the environmental conditions need to match closely. Slight variations e.g. in temperature or position might lead to significant observable differences over time. It is left to future work to explore how the simulation results can be validated once a physical implementation is available. In many cases it might be important to also choose the validation scenarios wisely to avoid damaging the hardware especially since the validation likely requires analyzing unusual circumstances that might lead to damaged hardware.

8.4 Choice of the right simulator

Chapter 6 has shown that the presented approach allows to use the simulation for purposes that were previously infeasible to accomplish in simulation. However, especially Section 6.4 has also shown that this increase in fidelity comes at the cost of much higher simulation time.

Thus it is important to carefully choose the simulation approach for a specific project. Existing simulators like the ArduPilot SITL simulators are sufficient for many purposes. Several bugs in [27] are easily detectable in this simulator. For example the bug fixed in commit `a185fa95` caused the UAV to disarm unexpectedly since IDLE checks were performed even while the motors were unarmed. Another example is commit `e583ade6` which fixes an issue where the acceleration was measured in $\frac{\mathrm{m}}{\mathrm{s}}$ instead of $\frac{\mathrm{cm}}{\mathrm{s}}$. Other specialized simulators like FlyNetSim[45] might be more performant compared to this more general approach presented above, especially when the simulation is used to analyze specific components or properties of the system to which this simulator is tailored to, e.g. the network communication.

In cases where the simulation should help answering more fundamental questions like whether a rotor of a certain size or shape is able to lift the weight of an UAV, a mathematical model might be suited best as implementation specific details like delays do not influence the results. In those cases Matlab/Simulink might be a good choice to help with those computations.

9 Conclusion

Simulation of CPS is an important tool in their development. However, existing simulators are only capable to simulate certain scenarios. The previous chapters introduced an approach to provide modular full-system simulations for CPS by extending the SimBricks framework.

The proposed interface allows to integrate physics simulators into the SimBricks ecosystem in order to provide accurate data for sensor signals taking actuator commands into account. This data is then used to emulate specific sensors connected to the simulation of the CPS's control software. This has several advantages over existing simulators such as the ability to run unmodified software and do the emulation transparently. The simulators also allow to easily adjust the simulated hardware (e.g. CPU frequencies, memory sizes etc.) and to easily replace sensors with emulators for different models.

This design has been implemented to support the Gazebo physics simulator and several emulated sensors that allowed an evaluation using ArduPilot. This evaluation has confirmed that the simulation is indeed capable of demonstrating the impact of different hardware choices and of detecting certain bugs that were previously hard to detect in simulation. However, this also comes at the cost of a significantly higher slowdown over realtime causing long running experiments.

Thus, it is crucial to evaluate the capabilities and tradeoffs of various different simulators with different levels of fidelity to choose the best option for a specific use case. Nonetheless, the evaluation has shown that this approach is promising for several use cases. Future work could explore integrating additional physics simulators, supporting more sensor and actuator types, or improving the performance of the simulation framework. Furthermore, the approach could be applied to a broader range of cyber-physical systems, enabling research in areas such as robotics, autonomous vehicles, and industrial automation.

Glossary

Acronyms

AHRS – Attitude and Heading Reference System: A system that combines data from multiple sources to obtain the orientation and acceleration of an aircraft.

CPS – Cyber-Physical Systems: Digital systems that interface with their physical environment through sensors and actuators. Common examples include cars, airplanes, drones and other types of robots.

ENU – East-North-Up: A coordinate system where x corresponds to the eastern direction, y to north and z points upwards.

GNSS – Global Navigation Satellite System: A system used for positioning using signals from multiple satellites providing worldwide coverage. Common examples include GPS (operated by the United States of America), Galileo (operated by the European Union), GLONASS (operated by Russia) and BeiDou (operated by China).

HAL – Hardware Abstraction Layer: An abstraction layer providing a common application interface to access hardware components enabling cross platform compatibility.

HOET – highest observed execution time: The longest execution time of a given program observed while testing or during deployment.

I²C – Inter-Integrated Circuit: A common interface for inter-chip communication. It only has two signals, a clock and a data signal and thus only allows for half-duplex communication.

MMIO – Memory-Mapped IO: A common way of interacting with external devices. The device's control registers are mapped to a specific memory location and can be read or written by accessing this memory location. Those accesses are then forwarded to the physical device using the device-specific protocol.

NED – North-East-Down: A coordinate system where x corresponds to the northern direction, y to east and z points downwards.

NIC – Network Interface Card: A hardware component that provides network connectivity to computers.

SITL – Software in the Loop: As opposed to hardware in the loop, SITL refers to simulators that do not require any hardware components and simulate the entire system.

SPI – Serial Peripheral Interface: A common interface for inter-chip communication. In a 1-to-1 setup, there are three wires, the serial clock and one data signal for each direction allowing for full-duplex communication.

UART – Universal Asynchronous Receiver Transmitter: A device implementing an asynchronous interface for inter-chip communication. Data is transmitted bit by bit with an optional parity bit. UART is also commonly used to refer to the protocol itself instead of the device implementing it.

UAV – Unmanned Aerial Vehicle: A class of CPS autonomously operating in the air.

WCET – worst-case execution time: An upper bound on the execution time of a program, usually by pessimistically over-approximating the actual runtime.

Software

ArduPilot: A control software suite for various types of unmanned vehicles such as copters, planes, rovers and boats, https://ardupilot.org

CARLA: An autonomous driving simulator supporting generated worlds, traffic and other obstacles, https://carla.org

Gazebo: An extensible physics simulation tool developed by OpenRobotics, https://gazebosim.org/home

gem5: A system simulator supporting different levels of fidelity and different CPU architectures, https://gem5.org

Matlab: A commercial platform for numerical computations, https://mathworks.com/products/matlab.html

MavLink: A protocol definition commonly used for communication with UAV and other embedded systems, https://mavlink.io

OMNet++: A framework to build simulators for network applications and networked communications, https://omnetpp.org

Orocos – Open Robot Control Software: A collection of C++ libraries to design robot control systems, https://orocos.org

PX4: A control software for drones developed by the Dronecode Foundation, https://px4.io

ROS – Robot Operating System: A collection of libraries helping to design robot applications developed by OpenRobotics, https://ros.org

SimBricks: A framework that allows virtual prototyping of systems by orchestrating different, specialized simulators for individual components of the simulated system, https://simbricks.io

Simulink: An environment to design and simulate CPS integrated into the Matlab platform, https://mathworks.com/products/simulink.html

List of Figures

Figure 1 Simulator configuration for a SimBricks simulation involving two hosts connected over the network. 7
Figure 2 Messages sent during setup and normal operation of the physical interface with polling. 16
Figure 3 Messages sent during setup and normal operation of the physical interface with pushing 17
Figure 4 Gazebo's ENU coordinate system vs. ArduPilot's NED system . . . 25
Figure 5 Experimental setup in SimBricks with one ground control station and one ArduPilot host, connected via the network. The ArduPilot instance is connected to the physical simulation in Gazebo. 29
Figure 6 Experimental setup in SimBricks with two ground control stations and two ArduPilot hosts. Both instances are connected via a shared network simulation and are simulated in a shared physical environment in Gazebo. 30
Figure 7 The impact of different CPU frequencies on ArduPilot's loop rate and the observable behavior of the drone. 31
Figure 8 The slowdown factor over time for the experiment shown in Figure 5, sampled at intervals of 1ms. 34
Figure 9 The average slowdown factor for various configurations with different numbers of simulated CPS configured as shown in Figure 6. 36
Figure 10 Percentage of time spent waiting in the device emulator for the physical interface and the memory interface to proceed (relative to the total wait time) with different numbers of simulated CPS. 36
Figure 11 Average slowdown factor for experiments with different numbers of simulated CPS, simulated using the existing ArduPilot SITL simulator and the existing Gazebo plugin[36]. 37

List of Tables

Table 1 Predefined metrics for the physical interface with their respective `ID`, unit and the default conversion factor. 24
Table 2 Lines of code required to implement the evaluated setup from Chapter 6. 38

List of Code Listings

Listing 1 Configuration script for the SimBricks simulation shown in Figure 1 . 8
Listing 2 The definition of a world in the SDF format. It includes only one object of a certain size that is equipped with a GNSS sensor. 11
Listing 3 The `intro` messages sent via the physical interface upon initializing the connection. 21
Listing 4 The configuration messages used to introduce the available sensors and actuators and to configure how sensor signals should be transmitted. 22
Listing 5 The messages used to exchange data on the physical interface. The generic `val_1` - `val_8` may hold different values depending on the specific sensor or actuator type. 23
Listing 6 The interface implemented by the emulators of sensors and actuators. 26

Bibliography

[1] M. Dowson, "The Ariane 5 software failure," *ACM SIGSOFT Softw. Eng. Notes*, vol. 22, no. 2, p. 84, 1997, doi: 10.1145/251880.251992.

[2] J. Abella *et al.*, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, IEEE, 2015, pp. 39–48. doi: 10.1109/SIES.2015.7185039.

[3] P. Axer *et al.*, "Building timing predictable embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, pp. 1–37, 2014, doi: 10.1145/2560033.

[4] Open Robotics, "ROS (Robot Operating System)." [Online]. Available: https://www.ros.org/

[5] H. Bruyninckx, "Open Robot Control Software: the OROCOS project," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation, ICRA 2001, May 21-26, 2001, Seoul, Korea*, IEEE, 2001, pp. 2523–2528. doi: 10.1109/ROBOT.2001.933002.

[6] Dronecode Project, Inc., "PX4." [Online]. Available: https://px4.io/

[7] ArduPilot Development Team, "ArduPilot." [Online]. Available: https://ardupilot.org/

[8] ArduPilot Dev Team, "Corporate Partners." Accessed: Jul. 21, 2025. [Online]. Available: https://ardupilot.org/ardupilot/docs/common-partners.html

[9] Dronecode Project, Inc., "Commercial Systems." Accessed: Jul. 21, 2025. [Online]. Available: https://px4.io/ecosystem/commercial-systems/

[10] Dronecode Project, Inc., "MAVLink." [Online]. Available: https://mavlink.io/

[11] ArduPilot Dev Team, "ArduPilot Custom Firmware Builder." Accessed: Jul. 21, 2025. [Online]. Available: https://custom.ardupilot.org/add_build

[12] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, SimuTools 2008, Marseille, France, March 3-7, 2008*, S. Molnár, J. R. Heath, O. Dalle, and G. A. Wainer, Eds., ICST/ACM, 2008, p. 60. doi: 10.4108/ICST.SIMUTOOLS2008.3027.

[13] A. Dosovitskiy, G. Ros, F. Codevilla, A. M. López, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*, in Proceedings of Machine Learning Research, vol. 78.

PMLR, 2017, pp. 1–16. [Online]. Available: http://proceedings.mlr.press/v78/dosovitskiy17a.html

[14] H. Li, J. Li, and A. Kaufmann, "SimBricks: end-to-end network system evaluation with modular simulation," in *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022*, F. Kuipers and A. Orda, Eds., ACM, 2022, pp. 380–396. doi: 10.1145/3544216.3544253.

[15] gem5 Development Team, "gem5 Simulator." [Online]. Available: https://www.gem5.org/

[16] N. L. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011, doi: 10.1145/2024716.2024718.

[17] J. Lowe-Power *et al.*, "The gem5 Simulator: Version 20.0+," *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2007.03152

[18] Fabrice Bellard, "QEMU." [Online]. Available: https://www.qemu.org/

[19] ns-3 Development Team, "ns-3." [Online]. Available: https://www.nsnam.org/

[20] W. Snyder, "Verilator." [Online]. Available: https://veripool.org/verilator/

[21] I. The MathWorks, "Matlab." [Online]. Available: https://www.mathworks.com/products/matlab.html

[22] I. The MathWorks, "Simulink." [Online]. Available: https://www.mathworks.com/products/simulink.html

[23] Open Source Robotics Foundation, "Gazebo." [Online]. Available: https://gazebosim.org/

[24] R. Smith, "Open Dynamics Engine." [Online]. Available: https://www.ode.org/

[25] Open Source Robotics Foundation, "SDFormat." [Online]. Available: http://sdformat.org/

[26] P. Lu and Q. Geng, "Real-time simulation system for UAV based on Matlab/Simulink," in *2011 IEEE 2nd International Conference on Computing, Control and Industrial Engineering*, 2011, pp. 399–404. doi: 10.1109/CCIENG.2011.6008043.

[27] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues, "Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, IEEE Computer Society, 2018, pp. 331–342. doi: 10.1109/ICST.2018.00040.

[28] P. Rosenberger *et al.*, "Towards a Generally Accepted Validation Methodology for Sensor Models - Challenges, Metrics, and First Results," Graz,

Austria, May 2019. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/8653/

[29] J. C. Hart, G. K. Francis, and L. H. Kauffman, "Visualizing quaternion rotation," *ACM Trans. Graph.*, vol. 13, no. 3, pp. 256–276, Jul. 1994, doi: 10.1145/195784.197480.

[30] TE Connectivity / Measurement Specialties, "MS5611-01BA03 Barometric Pressure Sensor, with stainless steel cap." [Online]. Available: https://www.te.com/commerce/DocumentDelivery/DDEController?Action=srchrtrv&DocNm=MS5611-01BA03&DocType=Data%20Sheet&DocLang=English&DocFormat=pdf&PartCntxt=MS561101BA03-50

[31] InvenSense, "MPU-9250 Product Specification Revision 1.1." [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf

[32] AKM, "AK8963 3-axis Electronic Compass." [Online]. Available: https://web.archive.org/web/20190803054413/https://www.akm.com/akm/en/file/datasheet/AK8963C.pdf

[33] National Marine Electronics Association, "NMEA 0183 Standard." 2002.

[34] I. Wang, P. Chakraborty, Z. Y. Xue, and Y. F. Lin, "Evaluation of gem5 for performance modeling of ARM Cortex-R based embedded SoCs," *Microprocess. Microsystems*, vol. 93, p. 104599, 2022, doi: 10.1016/J.MICPRO.2022.104599.

[35] D. Ferigo, "Multithreading in Ignition Gazebo." Accessed: Jul. 21, 2025. [Online]. Available: https://github.com/robotology-legacy/gym-ignition/discussions/363#discussioncomment-936604

[36] ArduPilot Development Team, "ArduPilot Gazebo Plugin." [Online]. Available: https://github.com/ArduPilot/ardupilot_gazebo

[37] A. Danial, "cloc: v2.06." [Online]. Available: https://doi.org/10.5281/zenodo.15734241

[38] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Arzen, "How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 16–30, 2003, doi: 10.1109/MCS.2003.1200240.

[39] P. J. Bohrer *et al.*, "Mambo: a full system simulator for the PowerPC architecture," *SIGMETRICS Perform. Evaluation Rev.*, vol. 31, no. 4, pp. 8–12, 2004, doi: 10.1145/1054907.1054910.

[40] A. Suzuki, K. Masutomi, I. Ono, H. Ishii, and T. Onoda, "CPS-Sim: Co-Simulation for Cyber-Physical Systems with Accurate Time Synchronization,"

IFAC-PapersOnLine, vol. 51, no. 23, pp. 70–75, 2018, doi: https://doi.org/10.1016/j.ifacol.2018.12.013.

[41] D. Nikiforov, S. C. Dong, C. L. Zhang, S. Kim, B. Nikolic, and Y. S. Shao, "RoSÉ: A Hardware-Software Co-Simulation Infrastructure Enabling Pre-Silicon Full-Stack Robotics SoC Evaluation," in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, Y. Solihin and M. A. Heinrich, Eds., ACM, 2023, pp. 1–15. doi: 10.1145/3579371.3589099.

[42] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," *CoRR*, 2017, [Online]. Available: http://arxiv.org/abs/1705.05065

[43] S. Karandikar *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, M. Annavaram, T. M. Pinkston, and B. Falsafi, Eds., IEEE Computer Society, 2018, pp. 29–42. doi: 10.1109/ISCA.2018.00014.

[44] P. S. Magnusson *et al.*, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002, doi: 10.1109/2.982916.

[45] S. Baidya, Z. Shaikh, and M. Levorato, "FlyNetSim: An Open Source Synchronized UAV Network Simulator based on ns-3 and Ardupilot," in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM 2018, Montreal, QC, Canada, October 28 - November 02, 2018*, A. Boukerche, S. S. Kanhere, and P. Bellavista, Eds., ACM, 2018, pp. 37–45. doi: 10.1145/3242102.3242118.

[46] G. M. T. McLean, S. Lambeth, and T. Mavin, "The Use of Simulation in Ab Initio Pilot Training," *The International Journal of Aviation Psychology*, vol. 26, no. 1–2, pp. 36–45, 2016, doi: 10.1080/10508414.2016.1235364.

[47] A. Chansanchai, "As real as it gets: Pilots lend their expertise to the most authentic flight sim on the market." Accessed: Jul. 21, 2025. [Online]. Available: https://news.microsoft.com/source/features/work-life/as-real-as-it-gets-pilots-lend-their-expertise-to-the-most-authentic-flight-sim-on-the-market/

[48] Laminar Reasearch, "Press Kit." Accessed: Jul. 21, 2025. [Online]. Available: https://www.x-plane.com/press-kit/

[49] K. A. Pant, L.-Y. Lin, J. Kim, W. Sribunma, J. M. Goppert, and I. Hwang, "MIXED-SENSE: A Mixed Reality Sensor Emulation Framework for Test and Evaluation of UAVs Against False Data Injection Attacks," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2024,*

Abu Dhabi, United Arab Emirates, October 14-18, 2024, IEEE, 2024, pp. 12414–12419. doi: 10.1109/IROS58592.2024.10802327.

[50] M. Parés Calaf, J. Rosales, and I. Colomina, "Yet another IMU simulator: validation and applications," p. , [Online]. Available: https://www.isprs.org/proceedings/2008/euroCOW08/euroCOW08_files/papers/20.pdf

[51] T. Brunner, J.-P. Lauffenburger, S. Changey, and M. Basset, "Magnetometer-Augmented IMU Simulator: In-Depth Elaboration," *Sensors*, vol. 15, no. 3, pp. 5293–5310, 2015, doi: 10.3390/s150305293.

[52] Open Source Robotics Foundation, "Gazebo Parallel Physics Report." 2015.

[53] YanningDai, "CPU Multithreading in Gazebo Harmonic." Accessed: Jul. 21, 2025. [Online]. Available: https://github.com/gazebosim/gz-sim/issues/2764

[54] cmeng-gao, "How to improve CPU/GPU utilization in Gazebo simulation calculations." Accessed: Jul. 21, 2025. [Online]. Available: https://github.com/gazebosim/gz-sim/issues/2982

[55] ArduPilot Dev Team, "Using Simulation Parameters to Control the Simulation." Accessed: Jul. 21, 2025. [Online]. Available: https://ardupilot.org/dev/docs/SITL_simulation_parameters.html