ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

Master's Degree in Computer Science and Engineering

Lowering the Reality Gap in Aggregate Programs Validation: Running Collektive Over Unity

Thesis in:
Software Process Engineering

Supervisor
Prof. Danilo Pianini

Candidate
Filippo Gurioli

Co-supervisors
Martina Baiardi
Angela Cortecchia

Graduation Session: IV
Academic Year 2024-2025

Abstract

Max 2000 characters, strict.

To my grandparents and Roberto...

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Modern computing is moving away from the era of powerful and isolated machines toward one composed by massively interconnected ensembles of devices. We can observe this transition everywhere, from global Internet of Thing (IoT) sensor networks to smart city infrastructures. In such scenarios, the focus shifts from 'how to compute' to 'how to coordinate'.

As the number of devices in these systems grows into the thousands or millions, traditional centralized management becomes a bottleneck. The latency, bandwidth constraints, and single-point-of-failure risks of a 'command-and-control' architecture make it unsuitable for the dynamic, often unpredictable environments these systems inhabit. Instead, we must look toward decentralized coordination, where collective intelligence arises from local interactions rather than global oversight.

This thesis explores the intersection of high-level collective programming and high-fidelity simulation. Specifically, it addresses the engineering gap between abstract coordination models, such as Aggregate Computing (AC), and the practical requirements of developing, testing, and deploying these models within realistic 3D environments. By leveraging the power of modern game engines and automated development workflows, this work aims to provide a robust infrastructure for the next generation of collective system design.

1.1 Motivation: Swarm Behaviour

The natural world provides the strongest precedence for the goal of resilient decentralized coordination. From the coordinated flashing of fireflies to the intricate architectural achievements of termite mounds and the smooth collective motion of starling murmurings, biological systems exhibit an efficiency that is frequently difficult for classical engineering to match. These phenomena, which are collectively referred to as Swarm Intelligence (SI), arise from the interaction of many simple agents that follow localized rules rather than from a global supervisor.

In a natural swarm, intelligence is inherently distributed and emergent. Individual agents (be they ants, bees or birds) possess only a partial perception of their surroundings. The collective however can solve high-order problems such as finding the shortest path to a food source or executing rapid evasive maneuvers against predators. From an engineering perspective, these systems offer three indispensable properties:

- the absence of a central controller; the loss of individual units does not compromise the mission.
- The logic governing ten agents often remains functional for ten thousand, as interactions remain local regardless of total population size.
- Swarms autonomously adapt to dynamic environments, re-configuring their behaviour in response to external stimuli.

As we attempt to port these characteristics into the digital and physical domains (specifically through paradigms like AC) we face a significant translation gap. While the mathematical models for collective logic are maturing, the infrastructure to test them in realistic, high-fidelity environments remains fragmented. To truly harness the potential of swarm behaviour in human-made systems, we must develop tools that can simulate the complex interplay between decentralized algorithms and the physical world.

1.2 Problem Statement: Engineering Challanges in Simulation

Simulation has been widely explored in terms of scalability, but not many researches have been done regarding high-fidelity. This field brings into play hard constraints that mathematical rigor often does not consider. Physics collisions, gravity and friction are just examples of what a good high-fidelity simulator could add to a cooperative swarm simulation. Traditional simulators often prioritize the number of agents at the expense of environmental complexity, leading to a 'reality gap' that complicates the deployment of algorithms onto physical hardware. Fortunately, game engines do this work for us; they add physics engines capable of computing the result of physical interactions with rigor. The real problem now becomes only one: bridging these two worlds.

The challenge of bridging high-level coordination with game-engine-driven physics is not merely a matter of data transfer, but one of architectural alignment. In particular:

- synchronism: collective programming models rely on discrete logical steps whereas game engines operate on a continuous, high-frequency tick (e.g. 60Hz, 60 frames per second).
- Abstraction: collective models treat agents like points in space whereas high-fidelity environment represent them as complex entities with mass, inertia and physical bounds.
- Scalability: the simulator should still be able to compete with other collective programming simulators in terms of nodes represented inside the experiments and their interactions.

Chapter 2

Background and State of the Art

To contextualize the contributions of this thesis, it is necessary to establish the theoretical foundations upon which it is built. This chapter explores the evolution of distributed systems toward collective intelligence and examines the formalisms of self-organizing frameworks. By evaluating the limitations of current simulators, this chapter identifies the technical 'reality gap' that this research aims to bridge, providing the necessary background to appreciate the integration of high-fidelity game engines into the decentralized coordination workflow.

2.1 Distributed Systems and Organizational Complexity

2.2 Self-Organizing Frameworks

2.2.1 Aggregate Computing

2.3 Simulation Landscape

2.3.1 Paradigms

2.3.2 The Reality Gap

2.3.3 Reealism vs. Scalability

2.4 Game Engines as Simulators

Chapter 3

Unity-Package-Template: Automated Unity Development Infrastructure

3.1 Requirements

3.2 Features

Chapter 4

Collektive×Unity: Designing a 3D Simulator for Collective Systems

This chapter face the core research project produced for this thesis: a simulator for 3D Complex Adaptive Systems (CAS).

4.1 Goal

The project goal is to bridge the Unity game engine with the aggregate computing library named Collektive.

This communication should be bidirectional, achieve high performance and enable huge customization.

4.2 Requirements

Requirements are split into separated categories.

4.2.1 Business Requirements

- The project should create a communication channel between the Collektive back-end and the Unity front-end.

- The project should extract environmental data from Unity node sensors and share them to the Collektive program.
- The integration must map the output of the Collektive aggregate program to Unity Actuators (e.g., changing position, color, or state of a GameObject).
- The system shall evaluate and implement a low-latency communication or bridge mechanism to minimize overhead between the JVM-based Collektive and the C#-based Unity environments.
- The integration should allow Collektive nodes to perceive Unity's colliders, rigidbodies and spatial triggers as first-class citizens.
- The integration layer should reamin agnostic to the specific CAS case study.

4.2.2 Domain Requirements

Simulator Domain

- The simulator should have customizable node sensors.
- The simulator should have customizable node actuators.
- The simulator should have customizable step duration (i.e. *delta time*).
- The simulator should be able to pause the simulation.
- The simulator should have a centered handling of randomization to enable reproducibility.
- The simulator should support addition and remotion of nodes in the simulation dynamically.
- The simulator should allow nodes to interacts at least with the following unity components:
  - rigid body
  - collider

- The simulator should support addition and remotion of neighbors dynamically.
- The simulator should support any kind of nighborhood discovery.

Communication Domain

- The communication should follow the reactive pattern (i.e. Collektive reacts to Unity's stimuli).
- The data exchanged should be agnostic from the underlying case study.
- Performance should be the driver for choosing the right technology.

Research Domain

- The system should prove the feasability of integrating game engines within CAS frameworks.

4.2.3 Functional Requirements

User Functional Requirements

- The user should treat a Unity scene as the simulation environment.
- The user should treat the Unity Editor as the simulator.
- The user should be able to add and remove nodes from the environment.
- The user should be able to create neighborhood discovery logic.
- The user should be able to inject any kind of collektive program inside the simulation.
- The user should be able to attach many different sensors and actuators to the same node.
- The user should be able to configure each node independently from the others.

System Functional Requirements

- The system should allow users to define custom sensors and acutators without modifying the core integration library.

- The system should allow users to define custom Collektive program without modifying the core integration library.

4.2.4 Non-Functional Requirements

- The system should maintain stable frame rate (> 30 FPS) with at least 500 active collektive nodes in a low budget laptop (ryzen 7 5700U, 16GB DDR4, integrated GPU).

- The system should be implemented with the fastest technology found during exploration.

4.3 Architecture

The integration is designed as a modular bridge between Unity and Collektive, facilitating the bidirectional flow of information as illustrated in diagram 4.1.

Figure 4.1: Diagram showing the bidirectional communication bridge between Unity and Collektive.

4.3.1 Architecture Rationale

Before detailing the individual components, it is necessary to outline the core architecture decisions that ensure the system's scalability and flexibility.

From Application to Package The architectural design of this bridge was shaped by a shift in how the simulation environment is hosted. Early design iterations focused on developing a standalone simulation application. However, this approach led to a redundant replication of complex features already native to the Unity Editor, such as spatial partitioning, inspector-based configuration, and asset management.

To resolve this, the architecture was refactored from a monolithic application into a Unity Package. This shift leverages the Unity Editor as the primary configuration interface rather than a mere rendering target. By distributing the backend and bridge as a modular library, the user can utilize Unity's native tooling to define simulation parameters, modify environment geometry, and inspect node states in real-time. Consequently, the project functions as an extensible framework that transforms a standard Unity project into a specialized CAS simulator.

Delegated Neighborhood Discovery A critical architectural decision was the delegation of spatial logic to the Unity environment. In aggregate computing, determining network topology is fundamental. If this logic were handled in the Kotlin backend, every simulation tick would require an $O(n^2)$ complexity check to evaluate node proximity.

By moving discovery to Unity, the system exploits the engine's highly optimized physics engine and pre-baked spatial partitioning (e.g., grids or Octrees). This allows for efficient neighborhood calculations using native features like Colliders or Raycasting, ensuring the backend remains a pure logic engine that receives a pre-calculated adjacency list.

4.3.2 Component Implementation

The bridge architecture is decomposed into three distinct functional components that realize the design principles mentioned above.

Collektive Parser

To enable communication across a network or inter-process communication (IPC) layer, the Collektive Application Programming Interface (API) requires a robust

serialization strategy. Two primary design choices were made to facilitate this:

- the simulation runs on a single machine; thus, Collektive is configured as `InMemory` to minimize communication latency;
- the generic identifier in Collektive has been reified to an integer to ensure predictable serialization.

The primary challenge lay in the program's input and output, which are both generics. While maintaining these as generics is essential for flexible collective behavior, it complicates data exchange. Rather than the brittle approach of manually replicating data structures and custom serializers on both sides of the bridge, this implementation adopts an Interface Definition Language (IDL). By utilizing an IDL, the system decouples the data definition from the implementation language, allowing 'a program or object written in one language [to] communicate with another program written in an unknown language' [con].

Unity Parser

While the Collektive Parser handles the logic-side data mapping, the Unity Parser translates these definitions into the engine's entity-based structures. Unity operates on an Entity Component System (ECS) inspired architecture, where the state of any object is defined by its collection of attached components. To bridge this with the aggregate computing domain, the *Unity Parser* is implemented as a specialized component responsible for bidirectional data translation.

At the scene level, a central *Simulation Orchestrator* component serves as the gateway. This entity enhances a standard Unity scene into an active simulation environment by managing the communication channel's lifecycle. It interfaces directly with the *Core Bridge*, utilizing the IDL to serialize the environmental state and deserialize incoming commands from the Collektive back-end.

For individual nodes, the parser abstracts Unity's internal state into the language-agnostic format required by the aggregate program. The parser is responsible for neighborhood discovery. Rather than delegating spatial logic to the back-end, Unity identifies neighbors according to any arbitrary logic (e.g., physical proximity, line-of-sight or topological links) and provides them to the bridge as a collection

of identifier pairs. This ensures the system remains agnostic to the specific neighboring criteria, allowing the user to implement any discovery logic within the Unity environment.

Core Bridge

The Core Bridge represents the central link in the system, serving as the high-speed interface between the Unity C# environment and the Collektive Kotlin/JVM runtime. To satisfy the strict non-functional requirement of maintaining high frame rates with over 500 active nodes, the bridge is implemented as a Foraign Function Interface (FFI) layer.

The choice of FFI over more traditional IPC methods was driven by the need to minimize serialization overhead and context-switching latency. By exposing Collektive's core logic as a native library, the bridge allows Unity to perform direct memory-to-memory communication. To ensure data consistency across the two environments, the bridge enforces a synchronous execution model. Each simulation 'tick' triggers a blocking call: the Unity engine pauses its internal loop while the bridge marshals the data and waits for the Collektive engine to complete its computation round.

The Bridge's role is strictly defined by three mechanical operations:

- Data marshalling: The process of transforming high-level language objects into a language-agnostic binary representation, as defined by the IDL schema.
- Function invocation: The execution of the foreign logic via the FFI, passing pointers to the marshalled data buffers.
- Data unmarshalling: The reconstruction of the binary data into the target language's native types, allowing the receiving environment to process the information.

A comprehensive performance comparison and a detailed evaluation of the trade-offs between FFI and Socket-based communication are provided in Chapter 7.1.

Chapter 5

Implementation of Collektive×Unity

This chapters goes into details about what produced in the Collektive×Unity project. Here it will be exposed the detailed design of the application.

5.1 Design

The design discussion will be divided into the 3 main components identified during the architecture phase: Collketive backend, Core Bridge and Unity frontend.

5.1.1 Collektive Backend

As stated in the architecture section (4.3) the communication across the bridge is synchronous. That means, each time Unity asks something to Collektive, it awaits its answer before restart doing its work. To enforce that architecture choice it has been created a central engine in charge of exposing the main capabilities of a CAS simulation. It can be seen as a facade pattern [GHJV94] in which the engine is the middleware from which the communication passes.

As shown in fig. 5.1 diagram this class should have methods to handle the whole simulation like `addNode`, `addConnection` and `step`. The `step` method in particular is the central point that should perform a cycle of the passed node. It is what moves the simulation one step ahead.

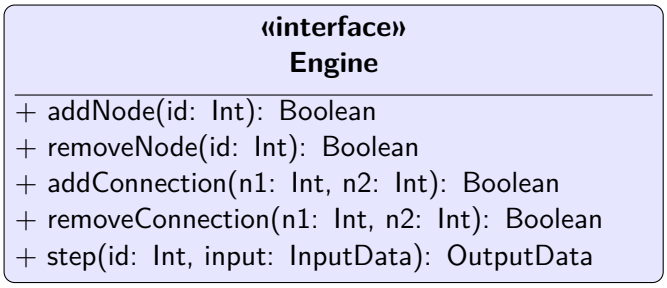

Figure 5.1: The Engine Facade.

```
1 fun Aggregate<Int>.entrypoint(sensorData: SensorData): ActuatorData
```

To follow the Single Responsibility Principle (SRP) [Wik24] it has been created a helper component that is in charge of handling neighborhood and network in general for the simulation. This component is split into 2 objects `Network` and `NetworkManager`. The former is a stateless representation of the network visible from the perspective of a node (i.e. the implementation of the `Mailbox` in Collektive) while the latter is a master that knows the entire network and handles communication. The same `NetworkManager` is injected to each `Network` that will use it in order to send and receive messages from and to the neighborhood. The fig. 5.2 diagram shows the network API and their interaction.

From requirements 'the system should allow users to define custom Collektive program without modifying the core integration library'. To honor that requirement the core collektive program (from now on 'entrypoint') should be somehow injected into the engine, so that it can remain independent from the actual simulation it has being done. This inversion of control is achieved by injecting a lambda into the Engine constructor, as illustrated in Listing 5.1.1. This allows the Engine to remain agnostic of the specific collective logic being executed.

The entrypoint serves as the primary abstraction layer for the simulation designer (i.e. the simulator user), providing a restricted scope where collective behaviors are defined without exposure to the underlying engine orchestration. By isolating this logic into a standalone lambda, the framework ensures that the simulation's behavioral rules remain decoupled from the communication and synchronization mechanics of the backend.

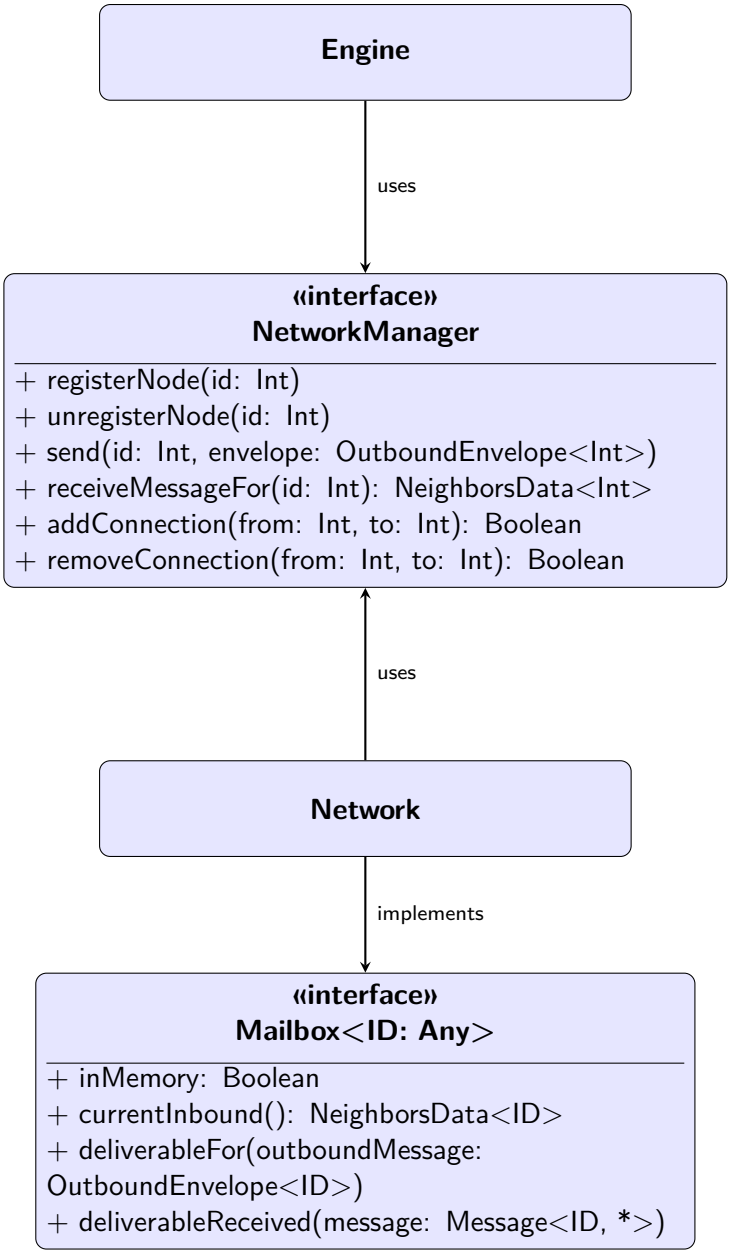

Figure 5.2: How networking is handled in the Collektive backend.

It is important to note that the `SensorData` and `ActuatorData` types utilized in the entrypoint signature represent the boundary between Collektive and Unity. To maintain strict type-safety across this boundary, these structures are centralized in the *Core Bridge*, acting as the common interface through which simulation state and environmental sensors are exchanged.

5.1.2 Core Bridge

The Core Bridge serves as the critical 'glue code' that enables communication between the Collektive and Unity environments. As detailed in Section 4.3, it leverages an IDL to marshal domain-specific data structures into low-level byte arrays. These are transferred via a FFI to the opposing context, where they are reconstructed into native data structures.

The IDL selected for this implementation is Protocol Buffers. While FlatBuffers was initially considered for its superior 'zero-copy' performance [vO14], it was ultimately discarded due to compatibility constraints with Kotlin MultiPlatform (KMP) projects.

Protocol Buffers As stated in the official documentation, Protocol Buffers provide a 'language-neutral, platform-neutral, extensible mechanism for serializing structured data [...] You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data' [Goo24]. This system utilizes a specialized compiler (`protoc`) that processes a `.proto` definition file to generate the required source code for each target environment (e.g., `.kt` for Kotlin and `.cs` for C#).

The adoption of an IDL extends beyond mere serialization efficiency; it enforces the Don't Repeat Yourself (DRY) principle by providing a single source of truth for data definitions. This decoupling ensures that the data schema remains synchronized across both the Java Virtual Machine (JVM) and the Unity engine, reducing the risk of structural mismatches during the development of complex collective behaviors.

Usage The responsibility for defining shared data structures is delegated to the simulation designer. Users must define the following two core structures within the IDL schema:

- `SensorData`: the information captured by Unity components from the 3D environment to be sent to the Collektive backend;
- `ActuatorData`: The resultant data produced by the Collektive program, which is returned to Unity to trigger physical or logical actions.

The decision to delegate these definitions to the user is intentional and ensures the framework remains domain-agnostic. In a monolithic environment, this flexibility would typically be achieved through generics or polymorphism. However, since data must cross a language barrier where native generic types cannot be shared, the IDL serves as a functional substitute. By requiring the user to define these structures, the bridge maintains its genericity, allowing it to support any simulation type without requiring changes to the underlying bridge implementation.

Data Transmission Once the user-defined data structures are generated, the framework manages the lifecycle of their transmission across the bridge. When a simulation 'tick' occurs, the Unity engine serializes the `SensorData` into a compact binary format using the Protocol Buffers library. Lastly, the Core Bridge passes a pointer to the underlying byte array through the FFI channel.

On the receiving end, the Collektive backend reads from this shared memory location and reconstructs the data into native Kotlin objects. This process ensures that while the user works with high-level, type-safe classes in both C# and Kotlin, the actual communication remains a memory-bound operation.

5.1.3 Unity Frontend

The frontend serves as the final integration layer, consolidating the previously described components into a unified simulation environment. To align with Unity's

Component-Based architecture, the system utilizes specialized `MonoBehaviour` entities to bridge engine-level events with aggregate logic. Central to this coordination is the `SimulationManager`.

Acting as the functional counterpart to the Collektive `Engine`, the `SimulationManager` is a singleton-like component attached to a unique orchestrator object within the scene. Its role is twofold:

- environment configuration: it initializes the unity physics engine and global simulation parameters to ensure deterministic execution;
- interface bridging: it reflects the high-level facade of Collektive backend `Engine`. By interfacing with a static service layer that utilizes the Platform Invoke (P/Invoke) mechanism [Mic24], the manager exposes native library capabilities to the Unity domain.

As illustrated in diagram 5.3, this creates a continuous communication pipeline where the `SimulationManager` orchestrates the flow of data from the Unity scene, through the binary serialization layer, and finally into the native JVM execution context.

5.2 Implementation Details

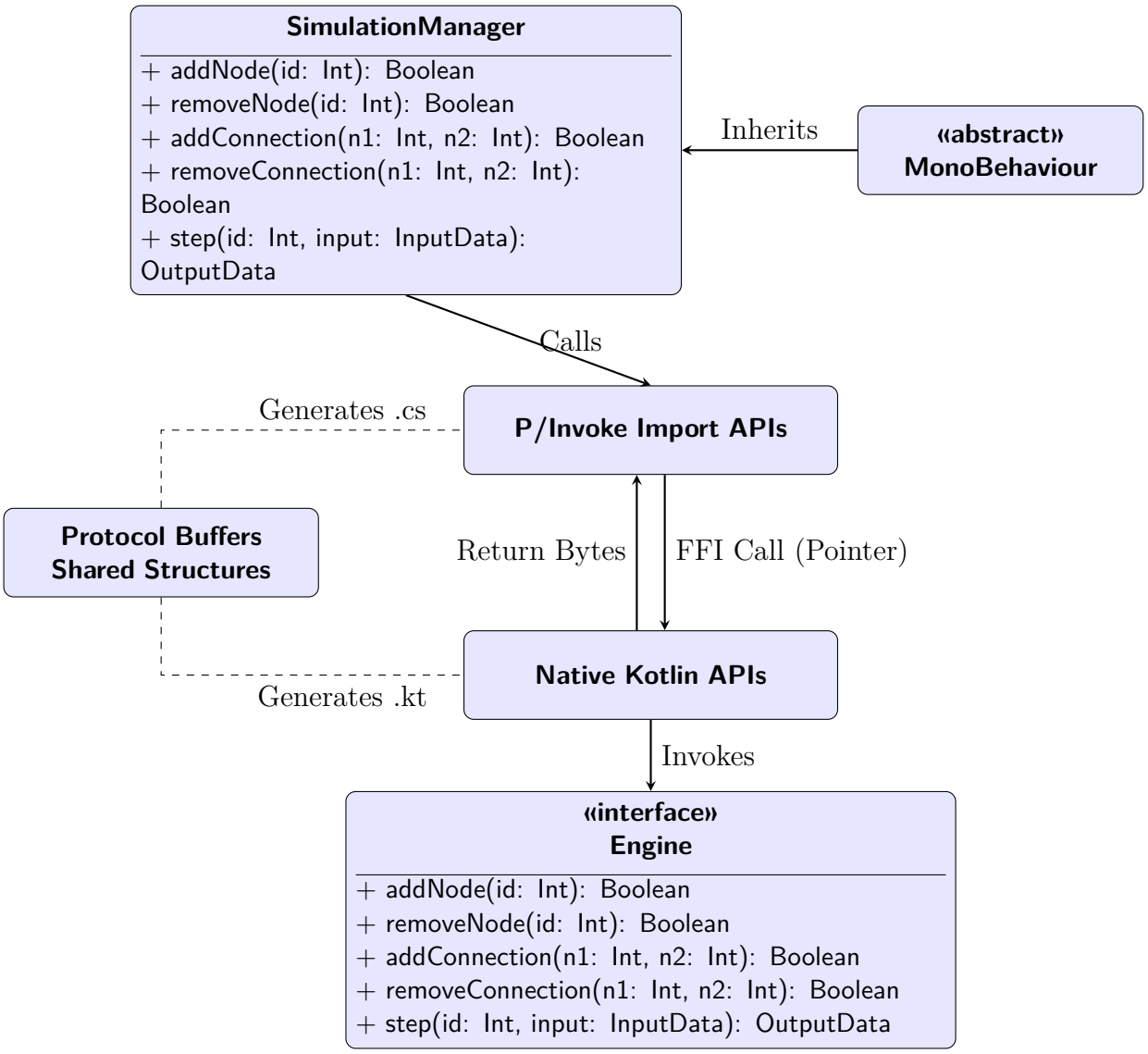

Figure 5.3: Communication channel from the Unity domain down to the Collektive boundary.

Chapter 6

Case Study: Environment-aware Gradient Ascent

Chapter 7

Results

7.1 Comparison with Socket-based Communication

Chapter 8

Conclusions and Future Work

Bibliography

[con] Wikipedia contributors. Interface description language.

[GHJV94] Erich Gamma, Richard F. Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: elements of reusable Object-Oriented software*. 1994.

[Goo24] Google LLC. *Protocol Buffers: Google's Data Interchange Format*, 2024. Accessed: 2026-02-17.

[Mic24] Microsoft. Platform invoke (p/invoke), 2024. Accessed: 2026-02-17.

[vO14] Wouter van Oortmerssen. Flatbuffers: Memory efficient serialization library. Technical report, Google, 2014. Accessed: 2026-02-17.

[Wik24] Wikipedia contributors. Single-responsibility principle — Wikipedia, the free encyclopedia, 2024. [Online; accessed 17-February-2026].

Acknowledgements

Optional. Max 1 page.