Agile Data Recording Architecture for Complex Scientific Simulations

Xiang Li

A report

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington
2024

Reading Committee:

Michael Stiber, Chair
Dong Si, Committee Member
Afra Mashhadi, Committee Member

Program Authorized to Offer Degree:
Master of Science in Computer Science & Software Engineering

University of Washington

Abstract

Agile Data Recording Architecture for Complex Scientific Simulations

Xiang Li

Chair of the Supervisory Committee:

Chair Michael Stiber
Computer Science and Engineering

Simulation development and eScience are driven by the complex questions that scientists and engineers want to answer. A simulation-driven or eScience investigation is an iterative process — as answers are found, new questions are created. Consequently, the development of simulation and eScience software involves rapid iteration, and the data that investigators want to capture from such software frequently changes.

Traditionally, new simulation data characteristics require development of new software modules or modification of existing ones to facilitate the recording of the updated data. This brings two disadvantages. First, scientists and engineers must invest significant time and resources into understanding and addressing data recording nuances with each iteration of their investigation. Second, the procedures developed during each iteration are of limited use in the next. This is particularly problematic in large-scale projects that involve various simulations, where managing multiple data recording systems becomes a significant overhead. To address these issues, we have developed a flexible and scalable data recording architecture that supports a wide range of simulations and data types. This architecture was realized by redesigning the data recording subsystem within the Graphitti simulator, and we assessed the flexibility and reusability of this redesigned system by evaluating the lines of code (LOC) and examining its maintainability. We observed a complete elimination of lines of code (a reduction of 100 percent) in the updated data recording subsystem compared to the old one, specifically in the context of recording various new variables within existing simulations. This result shows that new architecture significantly reduces development needs for saving and updating simulation data across different simulation projects, as well as modifying variables within existing simulation models. Additionally, we demonstrate that this approach can easily record more data types with minimal changes (2 lines of code), thus broadening its ability to support additional fundamental data types that were not previously accommodated by the data recording subsystem. Overall, our new lightweight data-recording architecture met our project goal of supporting various simulations without requiring the development of additional software.

Acknowledgements

I would like to express my deep gratitude to **Dr. Michael Stiber**, my capstone chair, for his invaluable guidance, patience, expertise, and support throughout the course of this research. His insight and feedback were instrumental in shaping the direction of this project, making my research journey not only productive but also enjoyable. My sincere appreciation also goes to committee members **Dr. Afra Mashhadi** and **Dr. Doing Si**.

In addition, I am truly fortunate to have joined the Intelligent Networks Laboratory, creating lasting memories with all lab members. It was a privilege to collaborate with: **Divya Kamath**, **Jardi Martinez Jordan**, **Pooja Pal**, **Mobei (Ben) Yang**, **Natalie Gonzales**, and **Nicholas Posey**. Their willingness to share knowledge and all the engaging discussions have always been helpful.

A special note of gratitude goes to my husband, **Tengda Cheng**, and my sons, **Carlo Cheng** and **Denali Cheng**. Your love, sacrifice, and understanding have helped me realize my academic dreams.

This capstone could not have been completed without the collective support of many. Thank you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scientific simulators are critical tools that replicate and model the behavior of complex systems and phenomena [1, 2]. They enable the study of natural processes, hypothesis testing, and the prediction of real-world problems through simulations [3, 4]. In order to meet the evolving demands of scientific research, simulators cannot be constrained to a specific use case. Instead, a modern simulator needs to have general use simulation capability [5, 6, 7]. This flexibility allows the simulators to adapt to different research contexts and accommodate new simulations, thus saving developers time and effort. Additionally, as science and technology advances and problems become more complex, new simulation models may become available for use in existing fields. In both scenarios, it is essential for simulators to quickly update and record simulation data with new characteristics while maintaining flexibility and adaptability. However, traditionally, the change of data recording requirement requires the development of additional software modules or modification of existing ones. A specific simulation data recording system often can't be reused by other simulation projects. This is time and resource consuming also makes the simulator complex and inflexible. Consequently, this becomes a major problem for large scale simulator projects that involves many simulation models.

1.1 Project Motivation

In response to the evolving demand for general use simulators that can handle a wide range of problems, Graphitti, an open-source high-performance graph-based event simulator, emerges as a versatile solution [8]. Graphitti is designed to be more than just a simulator for a single problem; it is designed to simulate multiple categories of problems, providing a flexible and adaptable platform for various graph-based network simulations [7].

However, Graphitti's adaptability is significantly constrained by the limitation of its current data recording system, the Recorder subsystem. This subsystem struggles to efficiently accommodate/-manage new data types or adapt to changes. Consequently, every new simulation domain and every new simulation type within a domain requires adding (sometimes substantial) specialized code to record that simulation's data, leading to a slow, cumbersome, and inefficient process. Such inefficiency limits Graphitti's adaptability and usability, presenting a critical need to reengineering the Recorder subsystem.

1.2 Project Goal and Scope

This project is to design an agile data recording architecture for Graphitti Recorder subsystem. As an illustrated example figure 1.1 highlights the difference between the existing approach and our proposed new design. When integrating new simulation models, instead of creating specific simulation model data recording component for each simulation, this re-engineered Graphitti data recording component serves as a general use Recorder for different simulation models in Graphitti.

With this more adaptable and scalable Recorder subsystem, Graphitti can become a more versatile tool to support the evolving demands of the scientific research without the need for repetitive,

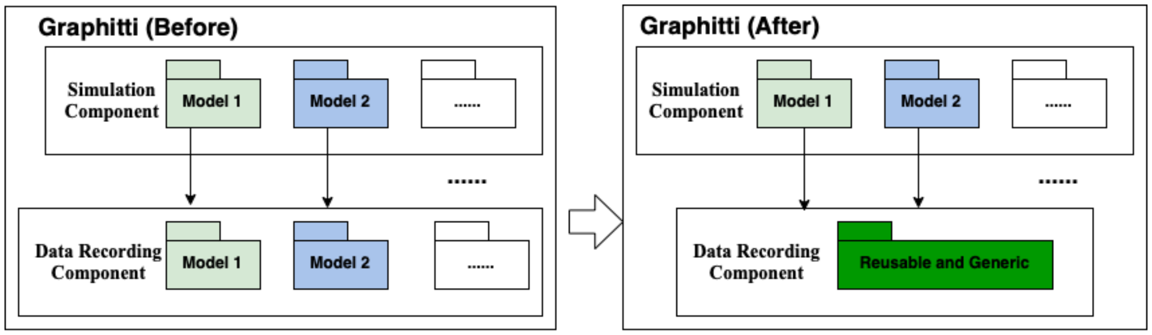

Figure 1.1: The difference between the existing approach and proposed new design illustrating the simplified data recording architecture in Graphitti

specialized code modification. This re-engineered architecture aims to streamline the integration of new simulations, thus enhancing Graphitti's overall maintainability and utility.

The objectives are:

- **Design a flexible Recorder Interface:** Implement a flexible and simple Recorder interface that can effortlessly manage a wide array of data types and adapt to various simulations without requiring additional software modules.
- **Efficient Management of Simulation Data:** Implement data structures and recording techniques capable of automatically managing the dynamic nature of simulation data. This will streamline the process of adding new simulations and modifying existing ones, eliminating the need for extensive custom coding.
- **Cross-Disciplinary Reusability:** Design the system for scalability, ensuring it can easily evolve with and support emerging simulations. This data recording architecture is built for cross-disciplinary reusability, making it applicable and supportive in diverse scientific research areas.

The scope of this project is focused on the Recorder subsystem's architecture to addresses Graphitti's Recorder key weaknesses. To accomplish the project objectives, this scope of the work plan includes:

- **Redesign of the Graphitti Recorder Subsystem Architecture:** The project started with a analysis of the current Recorder subsystem to identify its limitations. The key part of this project is to reengineer the Recorder's architecture, emphasizing modularization to boost the system's adaptability and ease of maintenance [9]. The focus is on the reduction of code complexity for recording diverse variables and standardizing the data reordering procedure.

- **Implementation with Existing Models Across Various Scientific Domains:** An important aspect of this project is ensuring the newly designed Recorder subsystem seamlessly integrates with Graphitti's existing simulation models. This includes neural networks simulations of the Growth (Neural Growth) model , Spike-timing-dependent plasticity (STDP) model, and the Next-Generation 911 (NG911) emergency response network model. [10] [11] [12] This phase aims to show the Recorder's capability to support many different real-world simulation scenarios without requiring extensive customizations or modifications in the codebase.

- **Validation Effort Reduction in New Data Recording:** We use quantified metrics to show the decrease in complexity and effort required to accommodate new and varied simulation data. This involves measuring the reduction in lines of code (LOC), [13, 14] assessing maintenance improvement, and analyzing the ease of integrating additional data formats or simulation variables. We also use workflow comparisons as measurement of improvement in how the Recorder manages the recording, processing, and storage of simulation data.

The Recorder design project significantly benefits researchers and developers by enhancing the system's adaptability and efficiency. Researchers gain from the system's ability to accommodate diverse recording needs and the ease of using the same recording procedures in different simulation projects, which streamlines the research process. For developers, the project shortens the development cycle, simplifies use and allows for straightforward adaptations to new requirements. This redesigned approach ensures the Recorder system effectively supports the needs of both research and development activities, emphasizing practicality and programmer-friendly design.

The new Recorder is intended for general use and could work in use cases outside the Graphitti simulations. In potential future work, the Recorder could be independent from Graphitti and become a stand-alone data Recording package.

1.3 Document Roadmap

This paper is organized in the following manner: following the introduction chapter, the subsequent chapter covers the theoretical background and the methodology for reengineering the Recorder subsystem. Chapter 3 examines the existing Recorder subsystem architecture and workflow within the Graphitti framework to point out its limitations and constraints. Chapter 4 and 5 detail the planning, design, implementation consideration for the new Recorder subsystem, including redesigned architecture and workflow, optimization techniques and testing strategies. The result chapter evaluates the improvements of the reengineered Recorder subsystem through system metrics. The final chapter summarizes the project achievements and explores future research opportunities.

Chapter 2

Background

This chapter lays the groundwork for understanding the reengineering tasks of Graphitti's Recorder system, focusing on essential concepts and technologies. This chapter aims to provide a foundational grasp of key C++ techniques, modern software development practices, all of which are key elements of this project.

2.1 Template and Instantiation Relationship

Templates in C++ offer a versatile approach to create functions and classes that are capable of working with diverse data types. This flexibility allows the implementation of algorithms and data structures without specifying the exact data type beforehand. This adaptability is pivotal in our project, particularly in constructing a Recorder system for Graphitti that accommodates a range of data types encountered in complex simulations.

To use templates in a C++ project, we can define a class or function with parameters representing types (`template <typename T>`) or non-type values (`template <int N>`). The actual types or values are then specified when using the template, allowing the code to be instantiated for specific data types or values. This method treats data as an unknown entity, introducing types only during actual usage. The instantiation relationship is one of C++ relationships and it exists if entity A is an instance of template B [15]. Instantiation relationship refers to the connection between a generic template and specific instances created using that template with particular data types or values.

Using templates in the Recorder Subsystem in Graphitti that is adaptable to various data types encountered in complex simulation brings several advantages. Firstly, templates facilitate code reuse, as a single template class or function can be used with various data types without the need for redundant implementations. This promotes maintainability and reduces the possibility of errors introduced by duplicating code. Additionally, templates promote type safety by enabling the compiler to perform type checking at compile time. This helps catch type-related errors early in the development process, leading to more robust and reliable software. However, it's essential to acknowledge the potential downsides. Complex template programming can introduce challenges in understanding and debugging, and there may be an increase in executable code size due to template instantiation for different types. Therefore, in designing the Recorder Subsystem, if we use template classes and functions, careful consideration is necessary to balance the benefits of flexibility and code reuse against potential complexities and code size implications.

2.2 C++17 Feature

While templates offer versatility, we have explored additional C++17 features in this project[1]. In this project, we use a new C++17 feature, `std::variant`, which is capable of handling different variable types [16].

[1]Enabled by MSCSSE student Divya Kamath's MS project, in which she modernized the Graphitti code base to C++17.

2.2.1 Feature `std::variant`

In C++17, `std::variant`, as a new versatile feature, is a type-agnostic container that can hold a single value of any type, offering flexibility in scenarios where the exact type is unknown at compile time. The simplicity of `std::variant` lies in its ability to accommodate a broad range of types without the need for explicit template parameters, encapsulating the ability to store one of several specific types at a time as a type-safe union. The `std::variant` keeps track of the type of value it currently holds, providing a safer and more controlled way to work with various data. The basic syntax involves specifying the potential types the variant can hold within angle brackets. For example:

```cpp
std::variant<int, double, char> myVariant;

myVariant = 42; //assigning an int
std::cout << std::get<int>(myVariant) << std::endl;

myVariant = 3.14; //assigning a double
std::cout << std::get<double>(myVariant) << std::endl;
```

In this code example, `myVariant` is a variant that can hold values of types `int`, `double`, or `char`. The `std::get()` function is used to retrieve the value of a specific type from the variant.

The specific advantages of `std::variant` include its commitment to type safety, as it keeps track of the type currently held. This is particularly useful in scenarios where a variable's type may change, and the specific type needs to be handled at run time. The performance benefits of `std::variant` are notable, as it avoids the overhead associated with type-erasure and dynamic polymorphism, providing an efficient alternative for storing and processing different data types within the simulation system. Ease of use is enhanced by a robust set of helper functions, such as `std::get()`, simplifying the process of accessing held values and applying operations.

2.2.2 The Case for `std::variant`

Choosing `std::variant` in the simulation recording system is a strategic decision driven by task-specific requirements. `std::variant` is selected for its type-safe storage and return of diverse data types through a single function/class. This eliminates the need for runtime type checking and casting, ensuring the data type returned is always correct and supported, which significantly reduces the risk of type-related errors and enhances system reliability.

In the Graphitti framework, which manages various data types including integers, floating-point numbers, and custom structs, `std::variant` streamlines data handling. It allows for the support of multiple data types through a single method or interface, thus improving the Recorder subsystem's adaptability to different simulation requirements. The explicit nature of `std::variant` in declaring supported data types improves code readability and maintainability, allowing developers to easily identify the data types handled by the system without reviewing implementation details. Another significant advantage of adopting `std::variant` is the future-proofing of the Recorder subsystem. It simplifies the addition of new data types to the system without modifying the method's interface or its data processing logic. This capability ensures the Recorder remains versatile and meet future simulation demands effortlessly.

2.3 Reengineering Software Architectures

Reengineering software architectures involves a systematic approach to transform an existing software system in a way that reconstitutes its structure and design to improve its maintainability, extensibility, efficiency, functionality, or better alignment with current needs or future needs. [17, 18]

There are many key principles that guide software reengineering. The first principle is understanding the existing system. Comprehensive analysis and documentation of the current system's architecture are crucial. It provides insight into the system's strengths and limitations and identifies the components that require reengineering [19]. Second, it is important to define clear objectives to the project. These could include improving system performance, scalability, maintainability, or adapting to new technological standards. The third principle is choosing to use incremental or Big Bang approach for the reengineering process. An incremental approach modifies the system piece by piece, while a Big Bang approach replaces or significantly reworks large portions of the system at once [20]. Here, the initial plan is to use the Big Bang approach, but in later development this project switches to the incremental development approach. The next principle is to preserve functionality during reengineering the system. The core functionality of the original system should be preserved to the greatest extent possible. Reengineering should enhance the system without sacrificing the features that users depend on [21]. Lastly, ensuring quality and reliability of new systems is an important principle [22]. Reengineering should pass rigorous testing to ensure that the new architecture is reliable and that the transition does not introduce new issues.

In the context of the Graphitti project, reengineering the Recorder subsystem means:

- **Assessing Current Recorder Limitations:** Identifying the limitations of the current Recorder architecture which includes inflexibility, complexity, and the inability to easily adapt to new types of simulator or simulation data.
- **Modular Design:** Creating a modular design that separates concerns, making the Recorder system more adaptable and easier to modify or extend [23, 24].
- **Leveraging Modern C++ Features:** Utilizing advanced C++17 features [16] to introduce type safety and flexibility into the Recorder subsystem.
- **Testing and Validation:** Ensuring that the reengineered Recorder retains all necessary functionalities and integrates well with other subsystems without introducing new issues.

When reengineering the Recorder subsystem within Graphitti, there are several considerations:

- **Backward Compatibility:** Ensuring the new system is compatible with existing data and simulation models to prevent disruption to ongoing work.
- **Validation and Documentation:** The project is broken into a list of smaller subtasks. Updating documentation and providing testing and validation for each subtask facilitates a smooth transition to the new system.
- **Planning for the Future:** Anticipating future needs and technological advancements to ensure the continued usefulness of the reengineered architecture.

Chapter 3

Related Work

This chapter provides an overview of the Graphitti simulator and its various subsystems, which is important for understanding the architecture of the Recorder subsystem and the role of the Recorder subsystem within the larger Graphitti simulation framework.

3.1 Graphtti and its Subsystems

BrainGrid [25] is a specialized biological neural network simulator developed in the Intelligent Networks Lab under the guidance of the University of Washington Bothell, Professor Michael Stiber. Graphitti, the successor of BrainGrid, aims to serve as an open-source, versatile, high-performance simulator supporting various graph-based network simulations [7]. Graphitti extends the capabilities of BrainGrid beyond the Neural Growth model and the STDP model [10]. Graphitti retains existing neural network simulation capabilities and can also be adapted for additional graph based network simulation tasks, such as 911 emergency response systems. The 911 application is the first non-neural network simulation to be developed using Graphitti. The Graphitti NG911 application represents the emergency response network as a graph-based network, demonstrating

Graphitti's versatility as a multidisciplinary simulator [12].

Graphitti has six core subsystems:

1. Core Subsystem: The Core subsystem orchestrates the simulation's execution. It manages the simulation across discrete time intervals, termed epochs. Within each epoch, it updates the states of vertices and edges, leveraging data from the Layout and Connections subsystems. Additionally, the Core subsystem uses the Recorder to log the network's state and behavior during each epoch.

2. Layout Subsystem: This subsystem is responsible for managing the spatial arrangement and structural attributes of the network. It oversees the Vertices subclass, detailing individual vertex characteristics separately from the overall network structure. This includes managing 2D coordinates and other spatial properties of vertices within the graph.

3. Connections Subsystem: Tasked with defining the network's connectivity, this subsystem interprets specifications from the input configuration file. It maintains and updates the state of edge connections, outlines network topologies, and modifies connections as the simulation progresses. This subsystem also governs the Edges subclass, differentiating edge properties from their behavior within the simulation.

4. Vertices Subsystem: This subsystem focuses on individual vertices, the primary units within the graph. It manages their distinct properties and states, such as type, coordinates, and current state, playing a critical role in the emergent behavior of the overall system.

5. Edges Subsystem: Representing connections between vertices, edges in Graphitti can exhibit various attributes like weight and directional orientation. The Edges subsystem is tasked with creating, modifying, and managing these edges, thus shaping the network's topology and interaction dynamics.

6. Recorder Subsystem: Integral to documenting the simulation process, this subsystem captures and stores both the results and intermediate states of the simulation. It records various aspects such as vertex location and edge weight. These recorded variables are associated with the Layout subsystem, Connections subsystem, Vertices subsystem and Edges subsystem. In the subsequent sections, we refer to classes that contain the recorded variable as the variable owner classes.

Graphitti's modular architecture, with its distinct interdependent subsystems, establishes a robust framework for simulating complex graph-based networks. Given the project's focus on enhancing the Recorder subsystem, we can simplify Graphitti's workflow by grouping these subsystems based on functionality: Recorder and Simulation Components. The Simulation components contain variable owner classes (encompassing Layouts, Connections, Vertices, and Edges) and Core. This grouping clarifies the roles and interactions.

3.2 The Recorder's Role in Graphitti Simulations

By reviewing the Graphitti workflow, we can see how the Recorder subsystem interacts with other subsystems and records simulation results. This workflow includes three phases:

Phase 1: Initialization and Configuration

- The initialization stage sets up the simulation environment, creating object instances and configuring simulation components.
- A Simulator class object is instantiated, which then creates a Model object appropriate for the simulation mode (CPU or GPU).
- The Model object oversees the instantiation of Edges, Vertices, Layout, Connections, and Recorder objects through their respective factory classes.

- Layout and Connections classes guide the creation of vertices and edges, with Layout generating vertex maps and Connections establishing edges between vertices using deserialized data if provided.

Phase 2: Simulation Execution and Recording (During simulation)

- Vertices and Edges are updated at every time step.
- Graph Modifications: Between epochs, the graph may undergo modifications based on the Connections class in use, such as creating or destroying edges or adjusting weights.
- Recorder Interactions: following a static recording configuration, the Recorder interacts statically with other subsystems:
  - Simulation Components: Components such as Vertices, Edges, or any other entities involved in the simulation expose/public certain variables that are relevant for recording. The accessibility of these variables is determined during the initialization phase, ensuring that the Recorder knows what data is available for recording.
  - Recording Process: The parameters of what is recorded (specific variables from variable owner classes) are all defined upfront. The Recorder accesses the variables in the variable owner classes and continuously captures the updated simulation data.
- Recording Procedure Difference: `XmlRecorder` Versus `HDF5Recorder`
  - `XmlRecorder`: This approach is straightforward. It accumulates all recorded data in memory during the simulation. Only after the simulation completes, it writes them to a file.
  - `HDF5Recorder`: It writes data directly to an Hierarchical Data Format version 5 (HDF5) file as the simulation progresses. It doesn't require holding large volumes of data in memory before writing it to a file.

Phase 3: Termination and Output

- As the simulation concludes, the Core subsystem instructs the Simulator to clean up resources, leading to an shutdown of the program.

3.3 Graphitti Recorder Subsystem Architecture

To better understand the Recorder subsystem's role in Graphitti, it is also necessary to examine the subsystem's functionality and architecture. The Graphitti Recorder subsystem is responsible for managing simulation results according to the requirements of specific simulations. This may include recording the state of vertices and edges in the graph, the numbers of vertices, and capturing network level metrics. In particular, these tasks include: identify recorded variables, collect data, possibly perform desired computations on the fly, based on requirements from the scientists, then output the result to the output file. The existing Recorder subsystem architecture is displayed in Figure 3.1.

The original Recorder subsystem contained the following classes:

- `IRecorder`: This is a foundational interface that is common to all `Recorder` classes within the system.
- `RecorderFactory`: This is a classic example of the Factory Design Pattern, a creation pattern in software engineering. The `RecorderFactory` creates instances of concrete Recorder classes at run time, depending on the current simulation's configuration.
- Specific Concrete Classes: Each of these classes extends the `IRecorder` interface to record different variables from different models into either Extensible Markup Language (XML) or Hierarchical Data Format version 5 (HDF5) data formats:
    1. Concrete Recorder Classes for neural network Simulation: In the neural do-

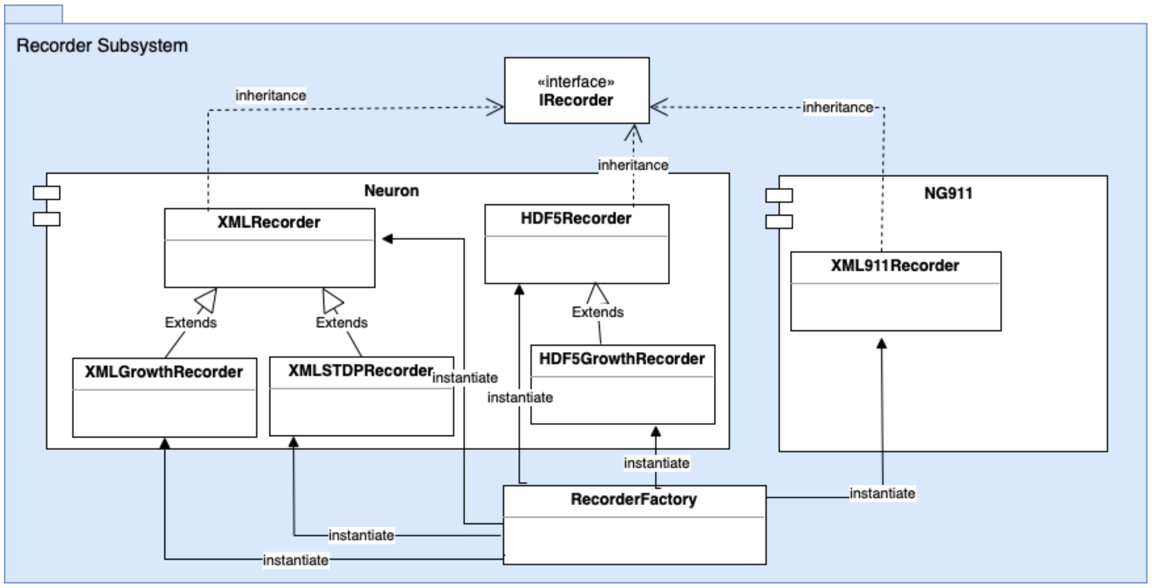

Figure 3.1: Graphitti Recorder subsystem UML diagram showing its hierarchical structure design with neural network and NG911 simulation models

main, there are two models that are extended: Neural Growth model and STDP model. To record the data from Growth model, `HDF5GrowthRecorder` and `XmlGrowthRecorder` are added. The `XmlSTDPRecorder` is added for STDP model results (no `HDF5GrowthRecorder` has yet been written — a good example of the impact of software development overhead on scieitific investigations). Each class caters to different aspects of neural simulation, capturing the simulation result for analysis and further processing. These concrete classes record data by hard coding the variables and computations to collect and save that data during simulation. `XmlRecorder` and `HDF5Recorder` records the neuron's layout, spikes history and compile history information. They can output network wide spike counts in 10ms bins. `XmlGrowthRecorder` and `HDF5GrowthRecorder` record the neuron's radius history in every epoch. `XmlSTDPRecorder` and `HDF5STDPRecorder` record neurons' weight histories.

2. Concrete Recorder Classes for NG911 Simulation: Since the NG911 is not fully implemented yet, currently it only includes the `XmlNG911Recorder`. Similar to the neural network simulation Recorders, this class specifically caters to NG911 simulations. It inherits from the `IRecorder` interface, ensuring consistency in the data recording process across different simulation types.

3.4 Problems in the Current Recorder Subsystem

The existing Graphitti Recorder implementation has several limitations, primarily stemming from its specificity to each simulation type. When integrating a new graph-based simulation into Graphitti, the addition of concrete Recorder classes derived from the Recorder interface is required. Over time, this approach has increased the complexity of the Recorder subsystem. Furthermore, the current Recorder design lacks flexibility to adapt to new computation requirements. Even for simulations of the same type, the need to add new Recorder classes for saving distinct variables of interest poses a significant inflexibility. This not only involves adding new concrete classes but also requires changes and effort to the input file, `Factory` class, and variable owner classes. Moreover, to enable Recorder classes to access member variables, the design requires making these variables public or establishing friendships between variable owner classes and the Recorder, which could lead to tighter coupling and reduced encapsulation.

These identified limitations highlight inherent design flaws in the current Recorder subsystem:

- **Overloaded Responsibilities:** The Recorder currently has multiple responsibilities. On one hand, it captures simulation data, and on the other, it manages the data based on specific use case requirements. This includes identifying interest variables, evaluating the need for additional computation, and defining how to store the required data in the appropriate format.
- **Lack of Isolation:** The Recorder subsystem is not isolated from other components in the

Graphitti framework. Consequently, adapting Graphitti for new simulation types requires simultaneous modifications to both the Recorder subsystem and related components.

- **Steep Learning Curve:** Modifying the Recorder demands a deep understanding of the computation aspects of recorded variables in different simulations. This is a challenging, particularly when the developer lacks familiarity with the specific scientific domain, preventing agile development.

Chapter 4

Methods

System change proposals drive system evolution. These proposals may be based on existing requirements that have not been implemented, requests for new requirements, and new ideas for software improvement from the system development team [26].

The existing Recorder architecture doesn't fulfill the needs of our current project and future requirements. As a result, we need to propose changes and redesign the Graphitti Recorder subsystem. In this chapter, we address the main design aspects of a new Graphitti Recorder subsystem. Initially, we examine the updated requirements for the Recorder. Then we explain the tasks for the new Recorder. Following this, we provide details about the architecture of the newly implemented Recorder subsystem. Finally we cover the Graphitti simulation workflow with the new Recorder subsystem.

4.1 Requirements and Tasks Analysis

The Graphitti framework is built to handle complex, graph-based network simulations. However, the current Recorder subsystem, which is crucial for capturing and storing simulation data, has struggled to keep pace with the Graphitti's broader vision of versatility and adaptability. Its limitations—rigid design, complexity in integrating new simulation types, and a steep learning curve for modifications—highlight a mismatch with the agile and generic aspirations of Graphitti. Recognizing these challenges, we started a strategic redesign of the Recorder subsystem to ensure that it can meets the evolving needs of diverse simulations. The redesign focuses on three key requirements essential for transforming the Recorder into a subsystem that matches the vision of Graphitti as a general use platform for network simulation:

- **Broad Compatibility for Variable Recording:** At the heart of the redesign is the need for a Recorder that effortlessly captures a wide variety of data types across different simulation scenarios. This means creating a system that can support not only current simulations like neural networks or emergency response networks but also future simulation use cases. The Recorder will support an extensive range of data types and formats, making it a versatile tool for researchers and developers working on many different simulation projects.
- **Built-in Extensibility for New Simulations:** As scientific research progresses, so do the requirements for simulation frameworks. The new Recorder is designed with the future in mind, offering easy ways to add new types of simulations, variables, and data output formats. This built-in flexibility ensures that the Graphitti framework can grow and adapt without the need for constant overhauls, making it a lasting solution for the scientific community.
- **Quick and User-Friendly Data Acquisition:** A key goal for the redesigned Recorder is to make it as programmer-friendly and efficient as possible. It will have the capability to quickly gather data from different parts of the Graphitti system. This agility makes it easier

for users to collect the data they need, when they need it, enhancing the practicality and effectiveness of their simulation studies.

By focusing on these three requirements, the redesigned Recorder subsystem aims to resolve the existing challenges and unlock new possibilities for the Graphitti framework. This chapter outlines our approach to create a Recorder that is not only more compatible and extensible but also simpler to use.

4.2 A Task-Based Approach

Based on the refined requirements analysis, we identified some important tasks for the new Recorder subsystem in the Graphitti framework (more detailed explanation of each task will be provided in Chapter 5). Here is the approach to each task:

- **Streamlined Recording Architecture:** Design a more efficient recording architecture consisting of versatile classes so that only one class must be written for each file type. Currently, our Recorder only targets two file formats: XML and HDF5. This simplified architecture will reduce complexity and improve the overall performance of the Recorder. If there is a new file format required in the further development, a concrete class can be easily derived from the Recorder interface.
- **Migration of Variable Registration:** Move the task of registering variables out of the Recorder subsystem. Instead, variable registration will be performed in the relevant simulation components, specifically in the classes that own the variables to be recorded. This approach will distribute the responsibility and improve the overall organization of the system.
- **Design of a Generic Variable Interface:** Design a common generic interface that can be used for all types of variables, enabling the Recorder to easily adapt to different simulation

scenarios. The newly introduced `std::variant`, a type-safe union in C++17, will be used in the implementation of this interface. Initially, the plan was to create a `Recordable` template. However, considering the need for developers to instantiate templates for each specific variable type in the Recorder classes, the module was redesigned. A `RecordableBase` interface was introduced, along with a `Recordable` template as the base class for recorded variables.

- **Creation of a Variable Information Table:** The new Recorder design uses composite, not inheritance. Instead of creating concrete Recorder subclasses, the Recorder adds a new variable element to a variable table in the existing Recorder class when a new variable needs to be recorded. We developed this Variable Information Table within the Recorder classes to store all the necessary information about the variables to be recorded. Each element in this table includes the variable name, its address, an internal buffer for data accumulation, the base type and other necessary metadata. By iterating through this table, the simulator is able to record all the required variable information.

4.3 The New Recorder Subsystem Architecture

To address the limitations and issues identified in the existing Recorder subsystem, We redesigned the Recorder subsystem architecture. Fig 4.1 shows the updated Graphitti Recorder subsystem UML diagram.

Here's a breakdown of the key components:

There are two interfaces:

- `Recorder`: The `IRecorder` interface in the old Recorder has been changed to `Recorder`, to be consistent with interface names in Graphitti's other subsystems. It serves as the base

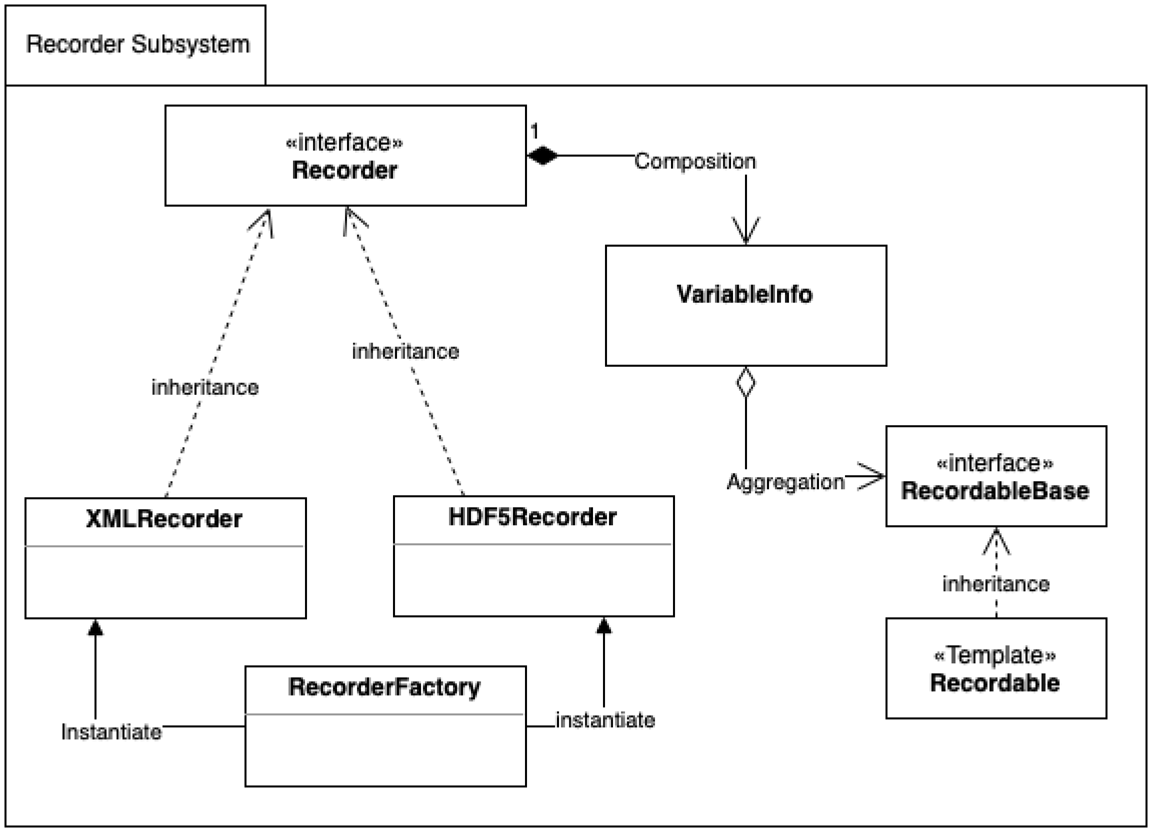

Figure 4.1: Redesigned Graphitti Recorder subsystem UML diagram

class for the recording mechanism, maintaining a list (variableTable) of variables that need to be recorded. It outlines methods essential for the recording process, such as capturing data and saving the data.

- `RecordableBase`: This interface defines methods and properties that make an object capable of being recorded.

There are several other important classes:

- `VariableInfo`: This struct is defined in Recorder to hold common variable information and attributes for a single variable to be recorded, such as variable reference, variable name, the basic type and the update frequency of this variable.
- `Recordable<T>`: A template class derived from `RecordableBase` interface, designed to handle variables of any type (T). This class makes it possible to record variables of different data types while using a common interface.
- `XMLRecorder`: A concrete class that implements the Recorder interface and handles the recording of data into XML files.
- `HDF5Recorder`: Another concrete class that also implements the Recorder interface and records data into HDF5 file formats.
- `RecorderFactory`: This class is responsible for creating instances of the two Recorder classes (`XMLRecorder` and `HDF5Recorder`). It uses the Factory design pattern to instantiate objects without specifying the exact class of the object that will be created until run time.

As the diagram (Fig 4.1) outlines, the redesigned, lightweight Recorder system contains only two types of Recorders — `XmlRecorder` and `HDF5Recorder` — created via a `RecorderFactory`. The latest Recorder subsystem introduces a Recordable variable module, enhancing its ability to uniformly manage and integrate diverse variables and recording methods. The new design allows developers to concentrate on simulation essentials rather than recording tasks. The use of interfaces implies that the system is designed for extensibility and polymorphism, allowing for new types of Recorders in addition to `XmlRecorder` and `HDF5Recorder` without modifying the core system structure in the future.

4.4 Detailed Workflow Analysis of the Redesigned Recorder Subsystem

The Recorder subsystem's operational workflow, as depicted in the sequence diagram in Fig. 4.2, gives us a step by step look at how the new Recorder subsystem keeps track of variables during the simulation. It shows the complex interactions between SimulationComponent, Recorder, and RecordableBase.

This diagram is broken into distinct phases: variable registration, data capture during simulation epochs, and data saving.

1. Setting Up (Variable Registration): Operation begins when a part of the simulation, the SimulationComponent, informs the Recorder of what to record. It gives the Recorder all the details including variable name, address, type, and (optionally) update frequency. This is how the Recorder receives the key information and organizes it into an internal table. Within this process, the Recorder may interact with the `RecordableBase` to determine the data type of the variable being registered. This is done through the `getDataType()` method, which returns a string representation of the variable's type.

2. During the Simulation (Simulation Epoch Loop): The diagram shows that as the simulation goes on, for dynamical variables, their data has been updated in each epoch. Every time

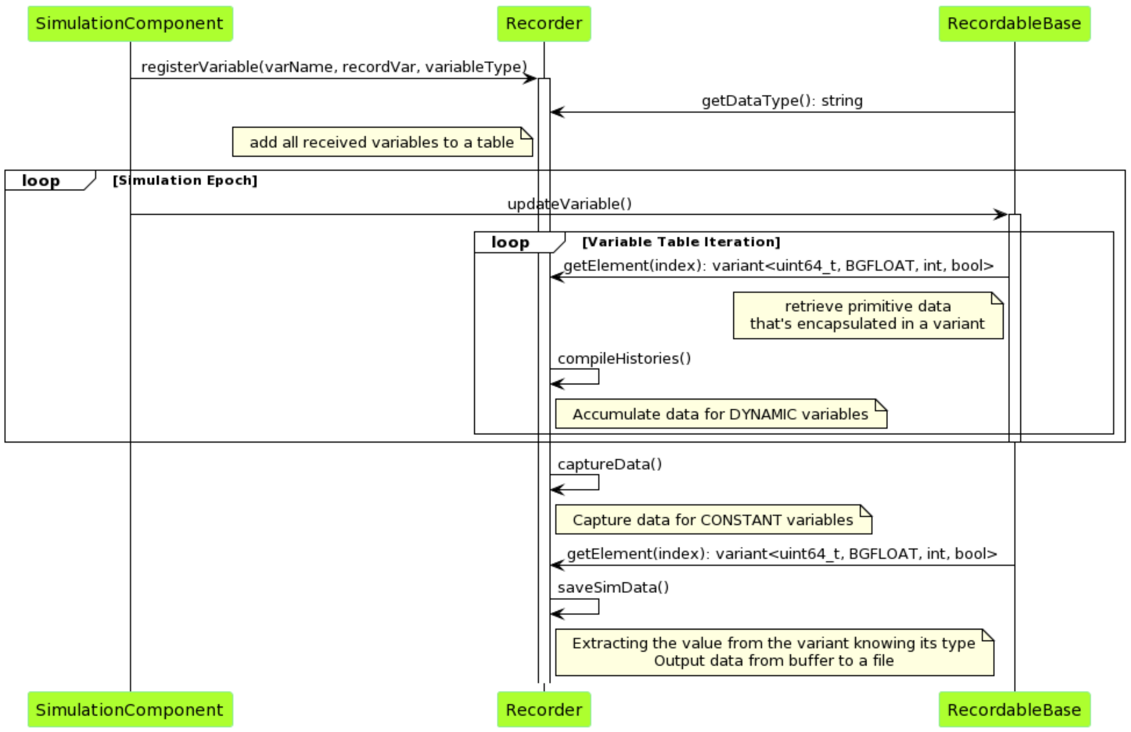
SimulationComponent
Recorder
RecordableBase
registerVariable(varName, recordVar, variableType)
getDataType(): string
add all received variables to a table
loop
[Simulation Epoch]
updateVariable()
loop
[Variable Table Iteration]
getElement(index): variant<uint64_t, BGFLOAT, int, bool>
retrieve primitive data that's encapsulated in a variant
compileHistories()
Accumulate data for DYNAMIC variables
captureData()
Capture data for CONSTANT variables
getElement(index): variant<uint64_t, BGFLOAT, int, bool>
saveSimData()
Extracting the value from the variant knowing its type
Output data from buffer to a file
SimulationComponent
Recorder
RecordableBase

Figure 4.2: Redesigned Graphitti Recorder subsystem sequential diagram

there is a changes, the SimulationComponent updates this information. Then, the Recorder iterates its variable table and checks each variable's updates through interactions with the RecordableBase. It uses a special method, `getElement(index)` to retrieve the current value of the variable. This method returns a variant that encapsulates the primitive data. This variant can hold various types (such `uint64_t`, `BGFLOAT`, `int`, `bool`), demonstrating the system's ability to handle various data types. This process ensures that the Recorder is always up-to-date with the latest information.

3. Wrapping Up (Data Capture and Saving): After the simulation ends, the Recorder first captures the data for constant variables — things that don't change. This is done by retrieving the variable's value using the same `getElement(index)` method. Then, it extracts values from the variant based on their type and saves all this information by outputting data from an internal buffer into a file. This step is crucial because it moves the data from the Recorder's memory to a file.

Chapter 5

Implementation and Testing

In this section, we describe the implementation details, discussing specific issues and the optimization efforts. We start with key implementation elements. Subsequently, we cover how we implemented those key elements.

The class diagram in Fig. 5.1 presents the relevant elements, their relationships, and the important functionalities within the simulation recording system, illustrating a more focused view of the interactions and data management strategies involved.

5.1 Migration of Variable Registration

To enhance flexibility and establish a more logical structure for the Graphitti Recorder subsystem, a significant change is proposed: transferring the responsibility for variable registration from the Recorder subsystem to specific simulation components, particularly the variable owner classes. The previous approach, where the Recorder subsystem manages variable registration, lacks flexibility and is not developer-friendly. Additionally, this method requires owner classes to make their

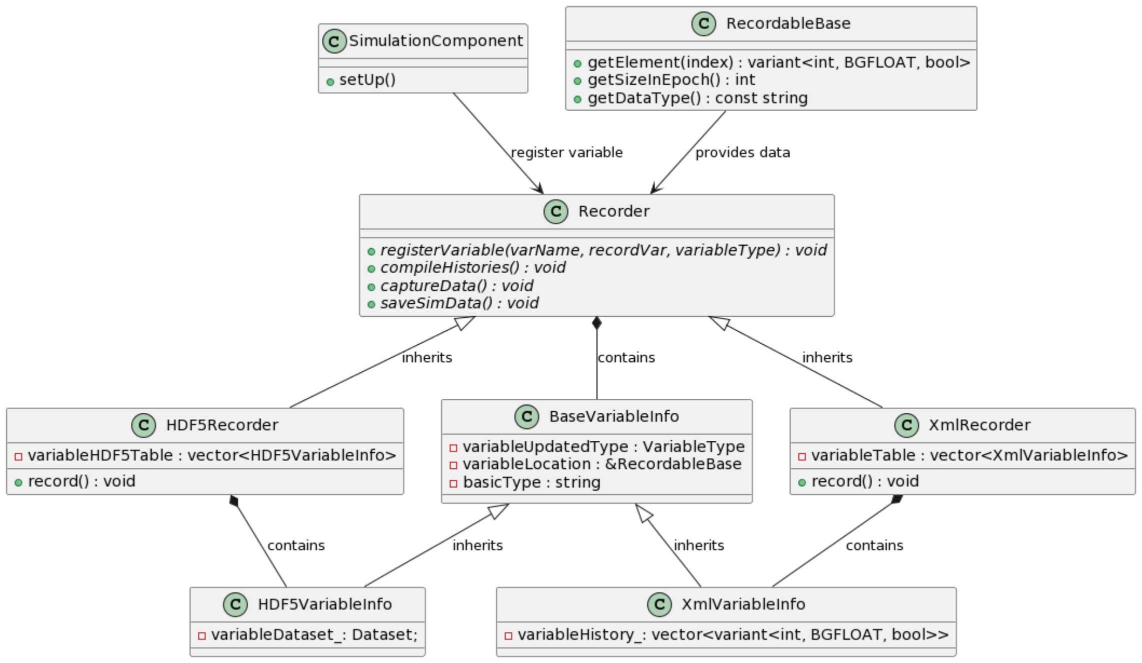
SimulationComponent
+ setUp()
RecordableBase
+ getElement(index) : variant<int, BGFLOAT, bool>
+ getSizeInEpoch() : int
+ getDataType() : const string
register variable
provides data
Recorder
+ registerVariable(varName, recordVar, variableType) : void
+ compileHistories() : void
+ captureData() : void
+ saveSimData() : void
inherits
contains
inherits
HDF5Recorder
- variableHDF5Table : vector<HDF5VariableInfo>
+ record() : void
BaseVariableInfo
- variableUpdatedType : VariableType
- variableLocation : &RecordableBase
- basicType : string
XmlRecorder
- variableTable : vector<XmlVariableInfo>
+ record() : void
contains
inherits
inherits
contains
HDF5VariableInfo
- variableDataset_: Dataset;
XmlVariableInfo
- variableHistory_: vector<variant<int, BGFLOAT, bool>>

Figure 5.1: Redesigned Graphitti Recorder Classes diagram illustrating the structured approach to record data

variables public for Recorder access, which is not ideal for data encapsulation. By allowing each simulation component to independently register its variables, the system becomes clearer, more adaptable, and better organized. Simulation components, familiar with their variables, can now customize and register them according to specific needs, thereby introducing agility to manage various and diverse scenarios and making the system more adaptable and easier to understand. Consequently, developers can focus on variable owner classes without the need to navigate Recorder component complexities. Moreover, this shift ensures enhanced data protection by allowing variables to remain private within their owner classes.

The diagram 5.2 illustrates the variable registration process in the updated Graphitti Recorder subsystem. The updated Recorder introduces a `registerVariable` method to facilitate the registration of variables. When a variable owner class decides to record a variable in its class, it simply invokes this Recorder method, providing the necessary variable information. The `registerVariable` function is designed with function overloading so that the Recorder can handle diverse registration requirements efficiently. The variable owner class calls the `registerVariable` in its `setUp` method, just after memory has been allocated for the Recordable variable. This ensures that the Recorder subsystem is aware of the existence of the variable and will properly handle recording and storage during the simulation. Essentially, the new Recorder allows variable registration by letting the variable owner class communicate its intent through the `registerVariable` method, and the Recorder stores the received variables in a table, handles the storage and output automatically to a file in a generic manner.

This revised methodology ensures that the registration of variables is closely tied to the context in which these simulation variables are used, creating a more modular and easy-to-maintain system. This decentralization makes the system more adaptable to various simulation scenarios and simplifies the development process.

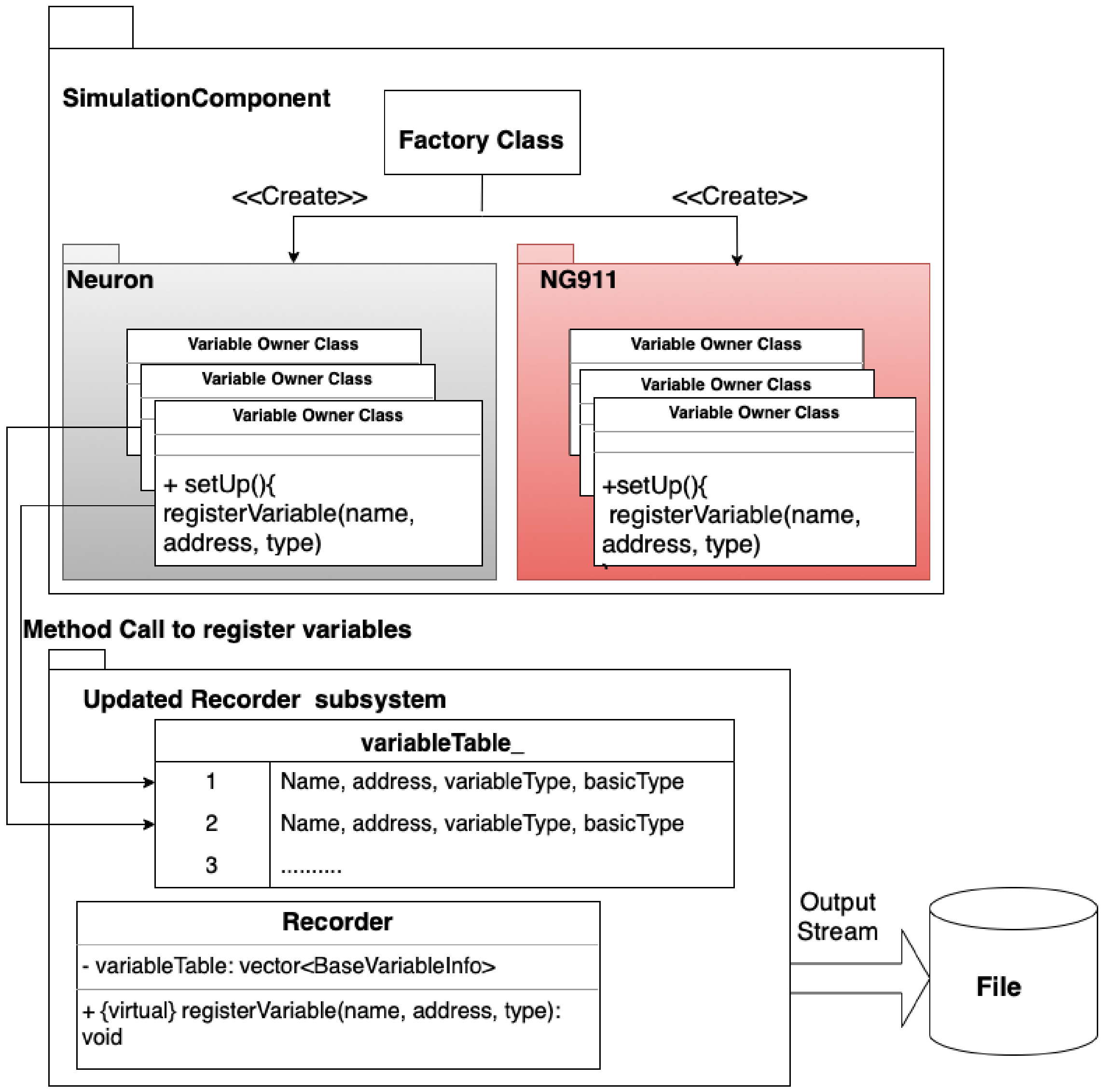
SimulationComponent
Factory Class
<<Create>>
<<Create>>
Neuron
NG911
Variable Owner Class
Variable Owner Class
Variable Owner Class
+ setUp(){ registerVariable(name, address, type)
Variable Owner Class
Variable Owner Class
Variable Owner Class
+setUp(){ registerVariable(name, address, type)
Method Call to register variables
Updated Recorder subsystem
variableTable_
1 Name, address, variableType, basicType
2 Name, address, variableType, basicType
3 ....
Recorder
- variableTable: vector<BaseVariableInfo>
+ {virtual} registerVariable(name, address, type): void
Output Stream
File

Figure 5.2: Redesigned Graphitti Recorder subsystem illustrating the variable registration process

5.2 Recordable Component Design

During the reengineering process, initial stages often include exploring various design patterns and implementations. Analyzing each design pattern and comparing them with metrics helps guide decision making [26]. In the development of the Graphitti Recorder subsystem, an important design decision is the base class for recorded variables, which involve a diversity of data types and data structures during simulation recording. We expected it to simplify the handling of complex data within a single framework, thereby minimizing code duplication and enhancing maintainability.

Various variable types for possible simulations demand a unified recording approach so that the Recorder can automatically process these different variable types in the same way. We considered two main design approaches: 1) employing a `Recordable` template directly within the Recorder, or 2) using an abstract `RecordableBase` interface alongside `Recordable` templates. We made the decision based on these criteria: code duplication, performance, and extensibility.

5.2.1 Two Approaches Comparison

To provide a concrete analysis, we utilized the existing neural network simulation scenario, focusing on recording spike events of vertices within the `XMLRecorder` class. This scenario served as a benchmark to compare the two design options across various metrics. The assessment included performance evaluation, measured by build and runtime across three different sizes of output files (categorized as tiny, small, and medium). Additionally, we provide some code snippets to examine code duplication in each design approach, and evaluate extensibility by estimating the effort (in lines of code) needed to incorporate a new basic data type into the Recorder. Table 5.1 lists the results of the comparison when implementing these two options within the `XmlRecorder` class when simulator records spike history in a neural network simulation.

Table 5.1 shows that the approach utilizing the `Recordable` template has a slightly higher run-

Table 5.1: Performance and Code Comparison of the two Recordable Component Design Approaches in the `XmlRecorder`

Approach	**Build Time (seconds)**	**Run Time (seconds)**		
—		Tiny (24K)	Small (292K)	Medium (1.5M)
Template	0.2837	6.401	56.469	282.945
Interface	0.2759	6.323	55.796	279.138
Reduction (%)	2.7%	1.2%	1.2%	1.3%

time performance cost compared to the approach that employs the `RecordableBase` interface. Additionally, the compilation time for the simulation program is longer when using the template-based approach, mainly due to the necessity of type checking for each basic data type at compile time.

To clear explain the design ideas and highlight the distinctions between the two approaches in the implementation of the new Recorder subsystem, we provide illustrative and simplified code snippets. These snippets illustrate how the `Recordable` template and `RecordableBase` interface are utilized in the `XmlRecorder` class respectively, although they are simplified for explanation purpose and not directly what the implementation in the `XmlRecorder` class.

Considering the Recorder supports two basic data type, `int` and `BGFLOAT`:

- `Recordable` Template Utilization: The approach has to maintains separate variable tables and implement overloaded `registerVariable` functions for each basic data type. It also needs additional function templates for each virtual function within the `XmlRecorder` since C++ forbids virtual template functions. This design complicates the code base, especially when integrating new data types, potentially conflicting with principles of code stability and extensibility. This approach has lower maintainbility due to repetitive code for similar operations.

```
///Recordable Template utilized in the XmlRecorder:

/// a variable table with basic data type BGFLOAT for recording.
vector<singleVariableInfo<BGFLOAT>> variableTableBGLOAT_;
/// a variable table with basic data type int for recording.
vector<singleVariableInfo<int>> variableTableInt_;
/// add more variable tables if need.
///........

/// Variable owner classes call it and pass variables with int type through
    Recorder interface
///received a registered variable and add it to the variable table
// Overload for int type variables
virtual void registerVariable(const string &varName, Recordable<int> &
    recordVar) {
     variableTableInt_.push_back(singleVariableInfo<int>(varName, recordVar))
        ;
}

// Overload for BGFLOAT type variables
virtual void registerVariable(const string &varName, Recordable<BGFLOAT> &
    recordVar) {
    variableTableBGLOAT_.push_back(singleVariableInfo<BGFLOAT>(varName,
        recordVar));
}
/// add more overloaded registerVariable function if need.
///........

///capture variable history information in every epoch and accumulate data
virtual void compileHistories() override
```

```
{
  compileHistoriedTemplate(variableTableInt_);
  compileHistoriedTemplate(variableTableBGFLOAT_);
  //......add when new table created
}

/// this template implements the compileHistories function.
/// capture all information for each variable table
template <typename T>
void compileHistoriesTemplate(vector<singleVariableInfo<T>> &table){
  for (int rowIndex = 0; rowIndex < table.size(); rowIndex++) {
     if (table[rowIndex].variableLocation_->getSizeInEpoch() > 0) {
        for (int columnIndex = 0;
           columnIndex < table[rowIndex].variableLocation_->getSizeInEpoch()
               ;
           columnIndex++) {
           table[rowIndex].variableHistory_.push_back(
              (*(table[rowIndex].variableLocation_))[columnIndex]
                 );
        }
     }
     table[rowIndex].variableLocation_->startNewEpoch();
  }
}
```

- RecordableBase Interface Utilization: This unified approach introduce feature `std::variant` to streamline the handling of a list of possible data types. It allows for a singular variable table and a single `registerVariable` function supporting variables of various basic data types through the `RecordableBase` interface. The virtual functions in the `XmlRecorder` class do not require extra template functions since common functionalities are abstracted in the interface. The duplicated code in the `XmlRecorder` class are entirely eliminated when using this approach. When considering extensibility, new types can be simply added to variant(only 1 line of code). This simplification minimize the potential for errors, increase code maintainability and makes extending support for new basic data type more straightforward. As changes to common behavior are centralized, the design has higher maintainability.

```
///RecordableBase Interface utilized in the XmlRecorder:

/// support a list of basic data types: int, BGFLOAT //add more if need
using multipleTypes = variant<int, BGFLOAT>;

/// a single variable table support different basic data type
vector<singleXmlVariableInfo> variableTable_;

///received a registered variable and add it to the variable table
virtual void registerVariable(const string &varName, RecordableBase &
    recordVar) {
     variableTable_.push_back(singleVariableInfo(varName, recordVar));
}

///capture variable history information in every epoch and accumulate data
void compileHistories( ){
  for (int rowIndex = 0; rowIndex < variableTable_.size(); rowIndex++) {
      ......
      variableTable_[rowIndex].variableLocation_->startNewEpoch();
  }
}
```

Table 5.2: Enhanced Maintainability and Extensibility through a comparative analysis of the two Recordable design approaches in the `XmlRecorder`

Metric	**`Recordable` Template**	**`RecordableBase` interface**
Performance Cost	High	Slightly low due to abstraction
Code Duplication	High	Low
Extensibility	Limited	High
Maintenance	More complex	Simplified

5.2.2 Decistion Rationale

Table 5.1 and code example presents a comparison between two approaches to recording data in a simulation system: using a `Recordable` Template directly versus using a `RecordableBase` interface. The comparison spans four key metrics: performance, code duplication, extensibility, and maintenance.

After a thorough comparison (Table 5.2), the decision to adopt the `RecordableBase` interface alongside `Recordable` templates has been chosen in the Recorder system. This decision is based on multiple considerations, including a slight enhancement in performance, improved maintainability, a reduction in code duplication, and increased system extensibility (potential expansions in variable type).

5.3 Testing Principles and Strategies

In the development of the Recorder subsystem project, a foundational testing principle was adopted through comprehensive unit and regression testing. This approach was essential in validating the functionality and correctness of the newly designed components within the Graphitti simulation environment. Emphasis was placed on creating a suite of tests that could cover a wide range of scenarios, from the simplest variable recording to complex, dynamic simulation states. This testing framework aimed to not only catch and correct immediate errors but also ensure that any changes to the codebase would not accidentally introduce regressions or disrupt existing functionalities. For unit testing, particularly the Recorder class, the Google Test framework was employed due to its comprehensive set of features and ease of integration with C++ projects. Testing the Recorder class presented unique challenges, notably in simulating diverse simulation states and variable updates to verify the accuracy and efficiency of the recording process. A significant challenge was ensuring the Recorder accurately captured and stored variable states across different data types and structures. The solution involved adding an extra set of testing methods such as constructor, getter methods, crafting mock objects and simulation components to feed a controlled set of data and updates to the Recorder, thereby isolating the Recorder's functionality for precise verification. This method facilitated the validation of the Recorder's behavior in a variety of simulated conditions, enhancing the reliability of the tests and the robustness of the Recorder implementation.

The transition to the new Recorder system introduced specific challenges in regression testing, especially given the alterations in data processing and output formats. The old Recorder system processed compiled data with additional computations, a requirement that was eliminated in the new design to streamline data handling. To validate the new Recorder's output against the legacy system, a methodical approach was taken by comparing the unprocessed raw data from the old Recorder against the formatted output of the new system. Despite the disparities in output format—where the old system presented a unified view of spike history data and the new system offered segmented, neuron-specific data chunks—a detailed comparison of corresponding numerical data was conducted. This comparison was carefully executed to ensure accuracy in the recorded data, despite the structural differences in output. For constant variables, such as vertex's locations, where data and format remained unchanged, direct comparisons were straightforward, serving as a benchmark for the reliability of the new system. This rigorous testing strategy was instrumental in confirming the new Recorder’s capability to accurately capture and represent simulation data, bridging the gap between new efficiencies and legacy accuracy.

Chapter 6

Results

This chapter evaluates how well the redesign aligns with the expected outcomes. It first illustrates the Recorder subsystem's workflow change within the simulation. Then it quantifies the impact of re-engineering on the codebase by comparing the Lines of Code between the old and new Recorder subsystems that were implemented for different simulation models, demonstrating reusability of the redesigned Recorder subsystem. Later, this chapter covers the assessment of qualitative aspects, focusing on extensibility and maintainability.

6.1 Evolution of the Recorder Subsystem in Graphitti: Overall Comparison

To illustrate the changes, Table 6.1 compares key aspects of the old and new Recorder subsystem designs across three operational phases: 1) Initialization and Configuration, 2) Simulation Execution and Recording, and 3) Termination and Output.

As we can see from the comparison, the redesigned Recorder subsystem enhances the Graphitti

Table 6.1: A Comparison of Recording Simulation Variables Before and After the Recorder Update

Phase	Aspect	Before Update	After Update
1	Class Utilization for Variables/Simulations	Required separate concrete classes for each variable and simulation type, leading to increased complexity.	Implements a RecorderFactory pattern, minimizing changes for new integrations with only two core concrete classes, regardless of simulation type.
1	Variable Registration Process	Utilized a dedicated configuration file for recording setup, necessitating hard-coded variables.	Enables direct variable registration by variable owner classes, eliminating the need for a separate recording configuration file.
2	Handling Different Data Types	Each new data type or recording need required adding new classes, making the system rigid.	Utilizes a generic interface with `std::variant` for flexible handling of various data types, avoiding the need for multiple specific data members.
2	Adaptability to Simulation Use Cases	Limited by a static recording configuration, reducing adaptability.	Adopts a universal protocol, streamlining the recording workflow for various use cases.
3	Data Output and Storage	Required hardcoded specifications for variables to store and output, leading to a lack of flexibility.	Automatically iterates through the variable table, saving accumulated data for each variable to a file, thus enhancing flexibility and automation.

simulation environment's adaptability and maintainability. By transitioning from an inflexible, use-case-specific framework to a more generalized, streamlined approach, the new Recorder system significantly simplifies the simulation workflow. This transformation not only reduces the complexity inherent in managing diverse data types and simulation scenarios but also aligns the recording process more closely with many simulation use cases' evolving needs. The introduction of direct variable registration by owner classes, coupled with a flexible recording strategy that can accommodate changes easily, ensures that the Recorder subsystem can better support the requirements of various scientific research and analysis. This redesign observes the agile principle, ensuring that the Recorder subsystem can respond to the evolving demands of complex simulations quickly.

6.2 Metrics

6.2.1 Quantitative Measures

In evaluating the Graphitti Recorder subsystem's improvement, the Source Lines of Code (SLOC) metric, also referred to as Lines of code (LOC), plays a crucial role. SLOC helps assess the system's maintainability and development effort by quantifying the codebase size [13]. This quantitative measure assesses the improvements brought by the updated Recorder subsystem across different simulation models within Graphitti, including Neural Growth model and STDP model, and the NG911 simulation model.

We compared four use cases. The first use case, adding variables to record, is a very common need in simulation research. The second and third use cases are about recording data when Graphitti is extending model coverage in the existing neural network domain. The last use case is about recording data when Graphtti expands to a new research domain.

Table 6.2 presents a comparative analysis of the lines of code required inside the Recorder subsys-

Table 6.2: Enhanced Maintainability and Reusability through a Comparative Analysis of the Lines of Code Required for Recording Simulation Variables Before and After the System Updated

Use Case	**Variable**	**Data Format**	**Old Recorder(line)**	**Updated Recorder(line)**
Adding variables to record	Spike, vertex location	XML + HDF5	71	0
Extending a Neural Growth Model	Radii	XML + HDF5	191	0
Extending the STDP Model	Weight	XML	119	0
New NG911 simulation	Vertex Type	XML	63	0

tem for recording simulation variables before and after the system redesign. Please note that the table only counts the relevant lines of code within the Recorder subsystem; therefore, the lowest possible number of lines is 0 if the function call code resides in another subsystem. The improvement of the redesigned Graphitti Recorder subsystem is demonstrated through three key simulation scenarios, showcasing its enhanced efficiency and flexibility:

1. Adding a new variable to record using an existing Recorder class:

   - Before the update, developers needed to configure and hard code the recording process for new variables, resulting in additional lines of code.

   - In the redesigned subsystem, adding new variables like "Spike History" or "Vertex Location" no longer requires extra coding in the Recorder subsystem, leading to a 100% reduction in coding effort on the Recorder subsystem.

2. Creating a new concrete Recorder class for registering variables of new simulations:

   - Previously, developers needed to add over 100 lines of code for each new simulation model, leading to unnecessary time and effort.

   - The redesigned subsystem simplifies this process by streamlining complex coding tasks into a single function call within variable owner classes, such as ConnGrowth, thereby enhancing the system's capacity to handle diverse data types effortlessly on the Recorder subsystem.

3. Integration of the NG911 simulation into the new Recorder architecture:

    - Previously, integrating NG911 simulation variables required substantial coding efforts and "hack code" to fit the older Recorder subsystem.
    - With the new Recorder, integration is seamless, requiring no additional lines of code in the updated Recorder subsystem, and simplifying the process for developers, highlighting the system's adaptability.

This table illustrates a 100% reduction in lines of code (LOC) within the Recorder subsystem across all scenarios, indicating the elimination of the need for additional coding. It's also worth noting that the redesigned Recorder no longer requires the creation of separate C++ files for each scenario.

The improvements in the redesigned Recorder subsystem go beyond just the significant reduction in lines of code and the number of files. They minimize the effort needed to record simulation variables, accommodating frequent updates to requirements—whether this involves adding new variables for new simulations or altering variables within existing ones. With the updated subsystem, the variable owner classes utilize a unified variable registration function to communicate with Recorder without resorting to hardcoding, which significantly reduces coding efforts and enhances the system's adaptability. This shift towards a more streamlined and modular recording process is in harmony with Graphitti's objectives of enhanced flexibility, reusability, and a reduction in coding workload. Moreover, it demonstrates that this redesigned Recorder can be utilized across different simulation models, making Graphitti more versatile and adaptable, and simplifying the entire process of integrating new simulations.

6.2.2 Qualitative Analysis: Extensibility and Maintainability

In addition to the quantitative comparison detailed earlier, in this section we look at the qualitative improvement.

First, in the scenario where further adaptation of the Recorder subsystem to accommodate new basic data types not currently supported, it takes significantly less time and effort due to the enhanced extensibility. For example, the system currently handles various fundamental types including `int`, `uint64_t`, `BGFLOAT`, and `bool`. Should there be a need to incorporate additional data types like `uint32_t` for recording new simulation data, the process involves two simple steps. First, expand the variant structure's data type list to include `uint32_t` within the Recorder's framework. Second, modify how values are retrieved from the variant for data output purposes. These straightforward changes can easily adapt the Recorder subsystem to evolving simulation requirements.

Second, in terms of maintainability, the redesigned Recorder subsystem has also seen improvements. Firstly, the removal of hacky code, exemplified by the NG911 simulation, has contributed to a cleaner and more streamlined codebase. The elimination of such ad-hoc solutions enhances the overall maintainability of the system. Additionally, the reduction in the proliferation of Recorder classes, as observed when adding new variables or simulations, represents a significant improvement. In the old Recorder subsystem, the continuous addition of concrete classes for various recording scenarios resulted in a complex and bulky structure. The redesign addresses this issue by introducing a more unified and extensible approach, mitigating the challenges associated with a growing number of Recorder classes. This streamlined design not only simplifies the addition of new variables or simulations but also positively impacts the overall maintenance of the codebase, ensuring a more efficient and manageable system.

Chapter 7

Conclusion and Future Work

In this project, we present the process of redesigning a modern Recorder subsystem for Graphitti to establish an agile and generic data recording architecture. This agile data recording architecture enables the flexible recording of different variables across different simulation scenarios, addressing Graphitti's complexity and inflexibility when it involves various simulation models. Our enhancements have streamlined simulation integration, automated data management, simplified data collection, and provided effortless management of data types. This redesign results in dramatic reduction in the lines of code required within the Recorder subsystem for adding new variables or modifying existing ones in various simulation models. The 100% reduction in necessary lines of code compared to the previous architecture demonstrates the success of transformation from a domain specific recording subsystem to a reusable purpose subsystem. The redesign is not just about fewer lines of code inside the redesigned Recorder subsystem, it also makes Graphitti more adaptable and efficient, simplify the process of integrating new simulations or extending existing simulations. Thereby the new design reduces the development burden on scientists and engineers. Furthermore, our redesigned recording architecture demonstrates the ability to easily accommodate additional fundamental data types with minimal changes in code, thus expanding its capability to support diverse simulation scenarios.

There are additional research opportunities. A key aspect of our future goals involves exploring the feasibility of configuring the input file to specify which variables should be recorded. This will allow the Recorder to automatically capture and store simulation data based on the provided configuration, which contains variables of interest. This will make Graphitti more dynamic and responsive to simulation requirements. Another opportunity involves exploring the `HDF5Recorder` implementation. As this project has primarily concentrated on the architectural design and implementation aspects of the `XmlRecorder` within the Recorder subsystem, the `HDF5Recorder`, with its potential for handling large datasets more efficiently and its capability to support complex data hierarchies, presents an interesting contrast to the XML-based recording approach. To sum up, future efforts concerning the Graphitti Recorder subsystem should prioritize expanding its capabilities and optimizing performance so it can support the evolving needs of scientific simulations even better.

The significance of this agile data recording architecture extends beyond its application in Graphitti and the various graph-based simulation models; it also has the potential to function independently from Graphitti. In future work, it could become a standalone data recording package, reusable across different simulation projects and promoting cross-disciplinary usability. This architecture provides researchers with a more adaptable and efficient platform, ideally suited for continuous scientific exploration with less coding required. Its widespread applicability and possibility for standalone operations encourages further exploration of its utility in diverse areas that require reliable data recording solutions. Moreover, the agility and flexibility of the agile data recording architecture mean it is well-equipped to keep pace with the future progress of scientific simulation technologies across a broad range of domains.

Bibliography

[1] Marc-Oliver Gewaltig and Markus Diesmann. 2007. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430.

[2] Dan FM Goodman and Romain Brette. 2008. Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics*, 2:350.

[3] Wikipedia. 2023. Simulation. https://en.wikipedia.org/wiki/Simulation.

[4] Armin R. Mikler, Johnny S.K. Wong, and Vasant Honavar. 1998. An object oriented approach to simulating large communication networks. *Journal of Systems and Software*, 40(2):151–164.

[5] Juan M. Durán. 2019. Computer simulations in science and engineering - concepts - practices - perspectives.

[6] Michelle Aebersold. 2016. The history of simulation and its impact on the future. *AACN Advanced Critical Care*, 27(1):56–61.

[7] Victoria Salvatore. 2021. *Demonstrating Software Reusability: Simulating Emergency Response Network Agility with a Graph-Based Simulator*. Ph.D. thesis, University of Washington.

[8] UWB-Biocomputing. 2023. Graphitti. `https://uwb-biocomputing.github.io/Graphitti/`.

[9] Nien-Lin Hsueh, Peng-Hua Chu, and William Chu. 2008. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, 81(8):1430–1439.

[10] Yi-Hsin Emily Hsu. 2020. *Extending a Neural Simulator to Combine Growth and Spike-Timing-Dependent Plasticity*. Ph.D. thesis, University of Washington.

[11] Snigdha Singh. 2021. *Graph Analysis For Simulated Neural Networks With STDP*. Ph.D. thesis, University of Washington, University of Washington.

[12] Victoria Jo Salvatore. 2021. *Demonstrating Software Reusability: Simulating Emergency Response Network Agility with a Graph-Based Simulator*. Ph.D. thesis, University of Washington, University of Washington.

[13] Wikipedia. 2023. Lines of code. `https://en.wikipedia.org/wiki/Source_lines_of_code`.

[14] H.E. Dunsmore. 1984. Software metrics: An overview of an evolving methodology. *Information Processing & Management*, 20(1):183–192. Special Issue Empirical Foundations of Information and Software Science.

[15] R.W. Lafore and Waite Group. 2002. *Object-oriented Programming in C++*. Kaleidoscope Series. Sams.

[16] N.M. Josuttis. 2019. *C++17: The Complete Guide*. Nicolai Josuttis.

[17] Roger S Pressman. 2005. *Software engineering: a practitioner's approach*. Palgrave macmillan.

[18] Dennis Mancl and W. Havanas. 1990. A study of the impact of c++ on software maintenance. *Proceedings. Conference on Software Maintenance 1990*, pages 63–69.

[19] Elliot J. Chikofsky and James H. Cross II. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17.

[20] Keith Bennett. 1995. Legacy systems: Coping with success. *IEEE Softw.*, 12(1):19–23.

[21] Vaclav Rajlich. 2006. Changing the paradigm of software engineering. *Commun. ACM*, 49(8):67–70.

[22] V. Côté, P. Bourque, S. Oligny, and N. Rivard. 1988. Software metrics: An overview of recent results. *Journal of Systems and Software*, 8(2):121–131.

[23] D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 2002. Design patterns: Abstraction and reuse of object-oriented design. In Manfred Broy and Ernst Denert, editors, *Software Pioneers: Contributions to Software Engineering*, pages 701–717. Springer Berlin Heidelberg, Berlin, Heidelberg.

[25] UWB-Biocomputing. 2016. Braingrid. `https://uwb-biocomputing.github.io/BrainGrid/`.

[26] Ian Sommerville. 2015. Software engineering. 10th. *Book Software Engineering. 10th, Series Software Engineering*, 10.

[27] The HDF Group. 2010. *HDF5 C++ Reference Manual*.

Appendix A

Appendix Backward Compatibility

The redesigned data recording architecture allows the `XmlRecorder` class to seamlessly record various variables with different data structures and types in the existing simulation models in Graphitti. Since this design doesn't require changing any code in other subsystems, compatibility issues within the Recorder subsystem are easily addressed.

The redesigned Recorder subsystem is capable of handling different data structures of existing variables. In the redesigned Graphitti Recorder subsystem, we've introduced a `RecordableBase` Interface and a `Recordable` Template that inherits from the interface, as part of the `Recordable` component. These two classes serve as the base class for recording variables.

Three distinct data structures within the variable owner classes have been developed in Graphitti to record different simulation variables:

1. `EventBuffer`: This data structure is used in Neural Network simulation models. It functions as a circular array for queuing event time steps produced by vertices, encapsulating vertex event buffering for events occurring in previous epochs.

2. `VertexMatrix`: It is an self-allocating and de-allocating 1D array optimized for numerical computations, recording static simulation variables like vertex locations.

3. Standard C++ STL vectors (`Vector<int>` or `Vector<BGFLOAT>`): These vectors with different data types are used by simulation variables.

The objects of these data structure classes represent the simulation variables that need to be recorded. Table A.1 outlines how these data structures are integrated into the `Recordable` component design. This table illustrates the adaptability of the redesigned Recorder's architecture to diverse data structures in existing simulation models, showing the backward compatibility of the reengineered Recorder subsystem. Specifically, it showcases the implementation strategies for `EventBuffer`, `VertexMatrix`, and standard C++ STL vectors. Through the `RecordableBase`, the system now supports various simulation variables, thus demonstrating a move towards achieving a flexible and robust recording mechanism in the Graphitti simulation framework.

Table A.1: Solutions for Adopting the Recordable Component Design with Different Data Structures in Graphitti

Data Structure	Data Type	Solution
`EventBuffer`	`uint64_t`	The `EventBuffer` class directly inherits from the `Recordable` template (Instantiation relationship).
`VertexMatrix`	BGFLOAT	`VertexMatrix` has an interface `Matrix`. The `RecordableBase` interface is added as the base class of `Matrix`.
`Vector<int>`	`int`	`RecordableVector` template has multiple inheritance and it inherits from `RecordableBase` and `Vector`.
`Vector<BGFLOAT>`	BGFLOAT	The `RecordableVector` template is directly utilized by substituting the vector in the variable owner class.

When implementing this `RecordableBase` interface, an analysis of the simulation variables within the Recorder shows that the Recorder needs to extract three important pieces of information from each simulation variable. Firstly, it captures the variable's value, which reflects the current state or data at a particular moment in the simulation. The return type for this value is a variant to allow flexible handling of multiple data types. Secondly, the Recorder determines the size of data items within the variable, a crucial detail for gauging memory requirements and planning storage allocation. Lastly, understanding the variable's basic data type is essential, as it ensures the recorded data can be stored, analyzed, and interpreted accurately.

The core of unified different simulation variable recording design lies in the `RecordableBase` interface. Serving as a common contract, this interface outlines the essential methods that any Recordable variable must implement. It introduces a standardized way for different variable types to interact with the Recorder. In this `RecordableBase` interface, it introduces std:: variant to provide flexibility in storing different types of data that a Recordable variable may have. By using `std::variant`, the interface provides variables with different basic types such as integers or float types to Recorder for storing. Additionally, the `Recordable` template class, derived from `RecordableBase`, caters to the specifics of each variable type. This template embraces the diversity of data types, providing a unified Recordable interface tailored to the needs of EventBuffers, Matrices, and beyond.

The redesigned Recorder subsystem is also capable of handling various variables with different update frequencies. Some simulation variables are dynamic, with their values updated in each simulation epoch. Beyond dynamic simulation data, we have identified another category of data in existing simulations: constant simulation data. Our redesigned Recorder subsystem is equipped to handle these two distinct types of simulation variables.

A significant change in the Recorder subsystem is the introduction of a variable table, which holds a list of variables registered for recording. Each entry in this variable table contains important information about the recorded variable, including its name, reference, update frequency (variable type), basic data type, and an internal buffer for accumulating data over time (for `XmlRecorder`).

The Recorder engages with this variable table in a two-phase process during simulation runs. The first interaction with the table occurs during each epoch of the simulation run. The Recorder iterates through the variable table, checking the update frequency (variable type) of each recorded variable. For variables marked as `Recorder::Dynamic`, the Recorder captures the value stored in the recorded variable that its reference or pointer points to and stores these values. The second iteration happens at the end of the simulation, where the Recorder iterates through the variable table again. In this iteration, if the variable is marked as `Recorder::Constant`, it first captures its value and then outputs the data to its own buffer, subsequently outputting the simulation results to the destination. If the variable is not marked as constant, meaning the data has already been captured, it directly outputs the simulation results.

By organizing the variables in the variable table and capturing their values at appropriate intervals, the Recorder subsystem efficiently records and outputs the simulation results.

Appendix B

Appendix Data Storage: HDF5

HDF5 (Hierarchical Data Format version 5) [27] is a data format used to store and manage large and extensive datasets in scientific research, which makes it an ideal choice when storing various kinds of simulation data. Our project's requirement to store large volumes of dynamic simulation data efficiently has led us to choose HDF5 .

A significant change in the redesigned architecture ensures that each Recorder concrete class corresponds only to a file type. As a result, the updated Graphitti Recorder subsystem now supports two types of Recorders: `HDF5Recorder` and `XmlRecorder`, and these two Recorder classes can be reused by different simulations. The `XmlRecorder` saves data in a plain text file with a ".xml" file extension, making it human-readable and easily processed and analyzed with other tools or programming languages. Conversely, the `HDF5Recorder` saves data in a binary file format using a ".h5" file extension.

There are notable differences in the recording procedure between these two Recorders in the previous version of the Recorder subsystem. The `XmlRecorder` captures all data in Vectors during the simulation and writes them to an XML file once the simulation is complete, whereas the

`HDF5Recorder` writes data directly to HDF5 library routines during the simulation, eliminating the need to store the entire dataset in memory at once.

As a result of those differences, when reengineering the updated architecture for the `HDF5Recorder`, specific implementation details require attention. For dynamic variables, the `XmlRecorder` captures the stored value in the recorded variable and places it in its container using a Vector<std::variant> at appropriate intervals, allowing it to accumulate data over time. After the simulation concludes, the `XmlRecorder` outputs all accumulated data at once. In contrast, the `HDF5Recorder` operates differently: each variable is stored as a Dataset (since an HDF5 file comprises a list of groups or Datasets, each being a multidimensional array of values of the same type). In the variable table design, each variable entity contains a Dataset named after the variable. The `HDF5Recorder` captures and stores data for dynamic variables in the Dataset associated with each variable. Since the Dataset is extendable, the `HDF5Recorder` does not need to accumulate data.

Given the specific requirements for the `XmlRecorder` and `HDF5Recorder`, including unique elements like Dataset for HDF5 and vector<std::variant> for XML, a single common base structure might not fully address the distinct needs of each Recorder type. However, we can design a common base that supports shared attributes while also accommodates the specialized needs of each derived Recorder class. Figure 5.1 outlines some common elements of each variable table entity, `baseVariableInfo`, for both the `XmlRecorder` and `HDF5Recorder` classes.

We have designed an agile data recording architecture for Graphitti, and although the design phase is complete, so far we have only implemented the XML Recorder component. Future work includes developing and integrating the HDF5 Recorder implementation, thereby completing the envisioned enhancements to the Graphitti simulation framework and ensuring comprehensive support for diverse data recording needs.