Demonstrating Software Reusability: Simulating Emergency Response Network Agility with a Graph-Based Simulator

University of Washington

A Thesis submitted in partial fulfillment of
Master of Science in Computer Science & Software Engineering

Victoria Jo Salvatore

Dr. Michael Stiber, Committee Chair
Dr. William Erdly, Committee Member
Dr. Afra Mashhadi, Committee Member

July 29, 2021

University of Washington

Abstract

Demonstrating Software Reusability:
Simulating Emergency Response Network Agility
with a Graph-Based Simulator

Victoria Jo Salvatore

Chair of the Supervisory Committee:
Michael Stiber
Department of Computing & Software Systems

This research validates the re-engineering of a neural network simulator to implement other graph-based scenarios. Most of the simulator's components were abstracted to increase reusability and maintainability through strategic refactoring decisions. This paper demonstrates how the simulator [1], developed at the University of Washington Bothell, can be adapted for other graph-based problems. By separating the neurospecific components from the core architecture of the simulator, this research verifies its functionality as reusable software. The scenario used to test the new architecture is the resilience of the US's Next-Generation 911 (NG-911) system in the face of a crisis [2]. Existing research acknowledges that both neural networks and emergency response networks are complex networks that exhibit self-organizing behavior [3]. Initial results from this small-scale test-case demonstrate that when a crisis destroys critical parts of emergency response infrastructure, NG-911 can reroute calls to keep communities connected with resources. This supports the conjecture that self-organizing patterns will emerge from the interconnected events of a full-scale network simulation. The success of this configuration provides evidence that the simulator can serve a broad spectrum of graph-based scenarios. Its growth potential is further substantiated by simulator's improved long-term maintainability and overall software quality.

Contents

Acknowledgements

I would like to express deep gratitude to my research chair, **Dr. Michael Stiber**, and my committee, Dr. William Erdly and Dr. Afra Mashhadi. This research would not be possible without the contributions of the research team: Vivek Gandhi, Chris O'Keefe, Kyle Dukart, Snigdha Singh, Emily Hsu, and Lizzy Presland.

Also, I want to acknowledge the NG-911 research contributions of Dr. Barbara Endicott-Popovsky, M. Scott Sotebeer, the National Emergency Number Association (NENA), the Seattle Police Department (SPD), and Seattle Fire Department (SFD).

Funding Acknowledgement: This work was supported by National Centers of Academic Excellence in Cybersecurity (NCAE-C) Research Grant awarded to the Biocomputing Lab (BCL) at University of Washington Bothell— jointly sponsored by The Department of Homeland Security (DHS) and the National Security Agency (NSA)[NCAEC-00302020]. Project sponsored by the National Security Agency under Grant Number H98230-20-1-0314. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notation herein.

1 — Introduction

Graph-based network simulations are among the most versatile tools for modeling event-driven scenarios. Neural networks, natural disasters, crisis communication, and population dynamics are among the systems that follow this paradigm, and are thus viable candidates for graph-based network simulations [4]. In order to accurately model any real world system, simulated models must be thoroughly optimized and able to manage large, multi-attribute graphs. This motivated the reuse of a proven domain-specific simulator as the framework for building a generalized version. *Graphitti* is the simulator introduced in this paper, and is redeveloped from the specialized neural simulator, *BrainGrid* [1]. This paper delineates between the legacy version and the reconstructed version of the simulator by referring to them as *BrainGrid* and *Graphitti*, respectively.

By leveraging the optimizations designed for large neural simulations, BrainGrid offers a unique set of capabilities not yet developed across other disciplines of science. Graphitti was developed to represent different systems that can utilize the same underlying abstractions. One of these systems is the Next-Generation 911 (NG-911) emergency response network to replace Enhanced 911 (E-911) across the United States [5]. This system is the first non-neural network to be simulated with Graphitti and is the demonstration of its viability as a general-purpose simulator adaptable to many disciplines. Graphitti sets itself apart from BrainGrid not only by improving its adaptability and configurability, but by reducing the life-cycle cost of development.

1.1 Research Motive

The similarities between neural connectivity and other graph-based complex networks motivated investigating how to decouple the simulator's abstract functionality from its

neuroscience roots. The BrainGrid simulator boasts high-performance and its ability to model both discrete and continuous (*a.k.a.* hybrid) events. These favorable qualities propelled its reconstruction and abstraction for other areas of study.

1.2 Project Scope

The software improvements that set Graphitti apart from BrainGrid are the principle elements of this research. The predominant goal of this paper is to demonstrate the value of rearchitecting a system through dissecting the design, development, and refactoring process. Through identifying system requirements and implementing them, this project adapts BrainGrid to become Graphitti, which can now model any graph-based network. The simulator is capable of modeling large-scale, long duration scenarios and uses a graph-based structure to represent event communications and behaviors of the whole network.[1] This robust graph-based simulator offers a novel way for scientists to evaluate the behavior of systems in an abstract form [6].

The Next-Generation 911 test case exposes the amassed value of making Graphitti accessible to all domains. This serves as verification that Graphitti can be used to implement a larger test-bed for the NG-911 system. The current implementation of the NG-911 network is a small-scale testing model. Chapter 6 outlines the next steps in realizing the full-scale NG-911 model.

1.3 Graph-Based Network Simulations

Anything that can be translated into a graph-based network can be modeled within such a simulator, such as the world wide web, weather patterns, natural disasters, brain activity, or phone call activity [1], [4], [7], [8]. These are examples of complex networks, defined as systems with many differing components that have varying internal complex structures and

[1] In this paper, the word *event* is used to describe both (1) the nodes of a network and (2) the instigating catastrophe of a definitional crisis.

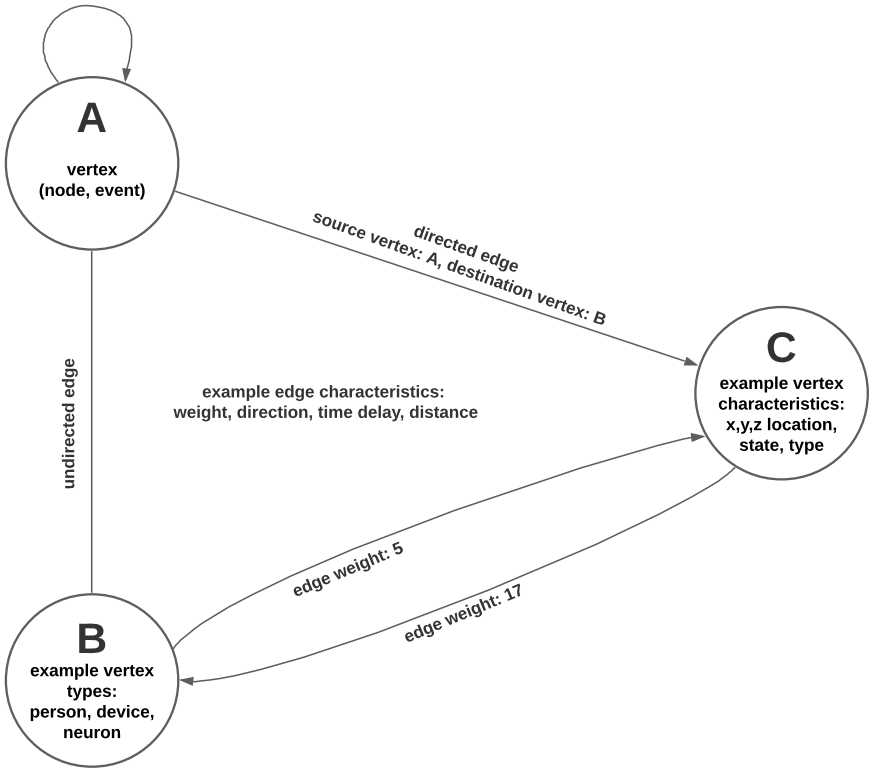

Figure 1.1: In this example, A, B, and C are vertices connected by edges. The relationship between A and B is shown with an undirected edge, which means either could be the source or destination vertex; it is a two-way path. The directed edges, conversely, show that there is a path from A to B through vertex C, but not a path from B to A through vertex C. When C is the source vertex, B and C (itself) are the only destination vertices.

behaviors of their own. The parameters and equations that characterize a real-world scenario distinguish each model [7]. They produce nonlinear behavior which affects the network both locally and globally [9].

Many of the underlying mechanisms used to model complex networks are identical. The main elements of graph-based networks are vertices (*a.k.a.* nodes, events) and edges [10]. Vertices can communicate with each other via edges, as seen in figure 1.1. By using a network of vertices and edges, this graphical paradigm enables simulations for any scenario with state variables and events.In directed graphs, the source vertex influences the state of the destination vertex. In weighted graphs, edges have different values assigned to them. The

addition of edge delays and event conditions can enable a simulation to successfully model real-world events, often using a pseudorandom number generator to account for variations or noise observed in the scenario's real-world counterpart. Vertices and edges have many representations and states that enable simulations of complex networks. Chapter 3 explains how vertices represent neurons and edges represent synapses in BrainGrid in order to simulate neural connectivity. In Graphitti, this neural representation is assigned to a subdomain to allow for other subdomains. The NG-911 subdomain uses vertices to represent caller, dispatch, and resource pools and uses edges to represent the connectivity between them.

1.4 Self-organizing Complex Networks

A system is self-organizing when order develops in a system experiencing a chaotic state. Self-organized critical systems are associated with fractal laws, power law, avalanche behavior, and $1/f$ noise [4]. These universal observations have enabled scientists and mathematicians to rationalize confounding events throughout history and nature. Phenomena exhibiting SOC behavior include the nervous system, crisis communication, the spread of infectious disease, natural disasters, populations, and, more broadly, the "game of life" [4], [8]. Self-organizing networks can be modeled with complex graph-based networks that simulate the same underlying mathematical behavior [4]. Since graph-based simulation is a scale-invariant modeling tool, it provides the optimal research platform for evaluating these systems. BrainGrid, for example, observes self-organizing behavior in large-scale biological neural network simulations [11]. Graphitti ultimately intends to utilize the infrastructure from BrainGrid to evaluate self-organized criticality and other complex network behavior.

1.5 Crisis and Communication

Unforeseeable events, such as a disease outbreak or an earthquake, are perfect specimens for graphical event modeling. In a disease outbreak, for example, each individual would be a vertex, their infection status would be the vertex's state, and an edge would connect a

newly infected individual with the infection source. The patterns of events emanating from a causal event reverberate through regions, compounding issues that result in, what is by definition, a crisis [3]. Emergency communication finds itself in a state of disorder during a crisis due to the increased demand for responders, damage to infrastructure, and level of emergency preparedness of a region [3]. The aftermath of these events varies in severity and duration in different scenarios.

In a crisis, a large event results in a collection of smaller events. For example, a winter storm could lead simultaneously to icy roads and power outages [12], [13]. Those consequences lead to events that affect individuals, such as car accidents and a freezing population. Inherently, emergency aid is not dispatched to respond to the storm itself, but to the individual victims of, for example, the car accident. When communication is compromised during these events, effects can be fatal. The inundation of these events during a large-scale crisis have unintended consequences that affect the entire ecosystem [3]. This is the motivating factor behind modeling the success of increased connectivity in crisis response.

1.5.1 Next-Generation 911

Next-Generation 911 (NG-911) is a national initiative to update emergency response infrastructure [5], [14]. It proposes leveraging new technology in an effort to better respond to crises. The State of Washington's NG-911 system must be robust enough to endure a major adverse event, such as an earthquake, that could compromise critical infrastructure. A robust system design must also tolerate a concurrent cyber-attack to the emergency response pipeline [15].

1.5.2 Cybersecurity Threats in Crisis

Without a robust emergency response network, opportunists can cause or exacerbate a major event by compromising inbound 911 communication, preventing citizens from getting through to emergency responders in time. This threatens national security and becomes an increasing

risk as infrastructure becomes more out-of-date [15]. In section 2.2.1, Mirsky and Guri [15] explain how quickly an emergency response network can be disrupted.

1.6 Research Overview

Other relevant research is presented in Chapter 2, followed by the background on BrainGrid in Chapter 3. This provides context for the architectural and design improvements made in Graphitti, summarized in Chapter 4. The NG-911 case-study is reviewed in Chapter 5. By implementing a small-scale emergency communications scenario, this research proves Graphitti's capability to model graph-based systems outside the realm of neuroscience. Chapter 6 contrasts Graphitti with other work, outlines project limitations and future work, and explores relevant research applications.

2 — Related Work

This chapter explores relevant literature on simulation modeling, emergency response, and self-organizing behavior. It contrasts two other graph-based simulators, discusses findings from Intelligent Networks Lab research, and examines crisis communication and self-organized criticality. Section 2.1 introduces a wide-lens context of graph-based simulators before framing Graphitti via more relevant literature.

2.1 Simulators

Surveying the simulation software landscape in its entirety is beyond the scope of this paper, so such surveys are left to other works [16]. Regardless, it is important to understand the broader functional characteristics of both BrainGrid and Graphitti. Simulations artificially duplicate a set of actual conditions, activities, and processes, by definition. Another quality is their ability to model discrete events, continuous events, or both [17]. BrainGrid and Graphitti are both hybrids of continuous and discrete-event simulation.

Simulation environments manifest themselves in various intersecting formats. The diagram in figure 2.1 contextualizes BrainGrid and Graphitti within the narrow field of simulation software for scientific research, which is used in many scientific fields to represent real-world systems. This diagram categorizes simulation software examples as having either a fixed or programmable functionality (shown in the second tier). They range from single-use functionality to flexible applications designed for reuse and adaptability. If extended left, this diagram could provide the example of a graphing calculator plot as an extremely simplified single-scenario, fixed-functionality simulation. Section 2.2.1 reviews a multi-scenario, fixed functionality simulator that is relevant to the NG-911 implementation of Graphitti. Graphitti and BrainGrid are both programmable-functionality simulation software, which can be

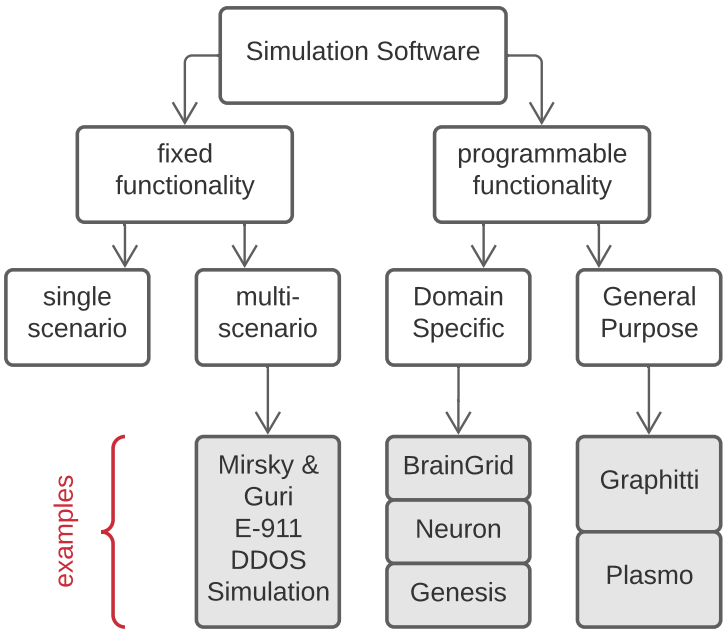

Figure 2.1: A type-map for simulators relevant to this paper, with examples.

divided into domain-specific and general-purpose categories (shown in the third tier of figure 2.1). These examples are not an exhaustive representation of all simulators or their qualities [18]. If the diagram positioned programming languages such as MatLab, Python, or C++ on the diagram, they would be added under a new third-tier branch for programmable-functionality development environments.

2.1.1 Simulating Complex Networks

Specialized simulators can be more efficient than general-purpose simulators by making better design decisions for complex, graph-based problems. This type of simulator is desirable when speed and memory cannot be spared. BrainGrid, Neuron, and Genesis are all domain-specific, programmable simulators specialized for problems within specific fields; in their case, neuroscience [19]–[21]. These simulators can strip away unnecessary components in order to maximize efficiency and produce large-scale models in their application field.

When computer scientists in the 1990s modeled a large communication network, this domain-specific, discrete-event simulation was resource-bound, meeting speed and memory challenges familiar to large-scale simulations [22]. These challenges continue as simulations become larger and more complex with the improvement of technology. Without meeting limits of hardware available today, the same large communication network simulation could run numerous times over. This issue remains relevant today despite the physical constraints differing by orders of magnitude. Graphitti's NG-911 model is also similar in content and setup to the simulator described in Mikler, Wong, and Honavar [22]. It models network traffic, transmitting calls via edges, and manipulates the connectivity of the network by deleting resources. While the full-scale NG-911 model will be larger and more complex than the communication network in the literature, it will encounter similar speed and memory challenges once complete.

Domain-specific simulation software are not as reusable and configurable as general-purpose software. Flexibility and adaptability are traits required for general-purpose simulation software. These software, such as `Plasmo.jl` and Graphitti, can serve a variety of simulation types. They will inherently contain a surplus of capabilities for any given scenario [18], [22].

In recent years, `Plasmo.jl` was introduced to manage scalable graph-based modeling abstractions for complex systems [18]. Like Graphitti, its intent is to simulate various types of networks. This package facilitates both optimization models and decision-making algorithms. The investigators demonstrated the success of this turnkey framework to construct, solve, and analyze three separate case studies: (i) how a gas pipeline network optimizes pipeline controls upon sudden demand decrease, (ii) how the number of CPUs and implementation of delays affect the behavior of Bender's decomposition,[1] and (iii) how a reactor-separator system is impacted by communication and computation delays. These case studies indicate that the package can simulate many different complex networks through its graph-based

[1]Bender's decomposition is an algorithm for complex algebraic graphs by solving subproblems that contribute to solving the master problem [18]

abstraction model [18]. While `Plasmo.jl` is comparable to Graphitti in its ability to model many different complex networks, it is only designed to work on CPUs at present, although there is a proposal to introduce a parallel computing version in the future. Its scaffolding does not host the scale of scenarios Graphitti is designed to accommodate, nor does its programming language, `Julia`, provide the same speed that `C++` offers, regardless of hardware platform.

2.1.2 Neural Simulation

BrainGrid and Graphitti implement initial value, graph-based simulation models for scientific computing. BrainGrid, much like Genesis and Neuron, is domain-specific, while Graphitti is general-purpose [19]. Specialized neural simulators like Neuron and Genesis remain the most proportionate comparisons for large-scale complex networks that intend to model, measure, and record many attributes [20], [21]. This, however, comes at the sacrifice of reusability and generalizability; it is this shortcoming that Graphitti intends to overcome.

2.2 Crisis and Response

Emergency response infrastructure is designed to accommodate varying types of events on a regular basis, but is vulnerable to failure when unforeseen catastrophic events occur. These crises can be manufactured by an adversary, or result from natural causes [8], [15]. For example, Distributed Denial of Service (DDoS) [2] attacks are desirable for adversaries who want to maximize their impact for a low cost, and often without internal access to the targeted software.

2.2.1 Emergency Response Network Threats

An example of software that could be targeted is 911 software. Other recent research simulated the impact of DDoS attacks on E-911 infrastructure (the current generation of 911

[2] DDoS attacks are just one of many types of adversarial attacks. Other common forms include phishing, malware, and ransomware attacks.

infrastructure) [15].[3] This fixed-functionality, multi-scenario simulator shown in figure 2.1, tested various types of DDoS attacks via simulation of cellular networks in North Carolina and the United States. They used a discrete event simulator (DES) to demonstrate that a relatively modest DDoS attack that could generate resource starvation could massively impact large regions within the nation's emergency response network. This work utilized simulated mobile phone botnets to flood the network which, both legally and technically, cannot deny service to callers. These researchers modeled call waiting time, duration, and Public Service Answering Point (PSAP) call capacity, telcom routers, phone type, and adversarial botnet flooding. State-level simulations for North Carolina found that as few as 6,000 bots could deny 20% of wired and 50% of wireless callers. Deploying 50,000 bots could prevent nearly 90% of callers from ever reaching a dispatcher. They conclude that it would only take 200,000 bots to disrupt nation-wide 911 services by flooding PSAP facilities with fraudulent calls. This number of bots could prevent one-third of legitimate callers from reaching a dispatcher by keeping the lines too busy for them to get through. This research is valuable in Graphitti's future scenario development. Our goal is to further understand how to best protect the NG-911 network from events, both adversarial and natural. At present, Graphitti's scenario simulates the aftereffects of a natural disaster, which destroys infrastructure and bottlenecks remaining resources.

2.2.2 The Science behind Crisis Communication

The study of crisis communication spans the disciplines of public health, engineering, sociology, design, communication, and defense. A corpus of literature from a group of disaster social scientists collects and analyzes cross-disciplinary data from crisis communication during previous catastrophes [3], [8]. Three notable characteristics of a crisis are identified as (i) surprising, (ii) usually threatening, and (iii) requiring a short response time [3]. They describe the uncertainty of catastrophic events and the intra-agency preparation

[3]This is an example of purpose-written software for specific simulations, as opposed to simulation software, which acts as predefined infrastructure for many simulations

required to mitigate negative outcomes. Because these events are sudden and unpredictable, the ecosystem affected is thrust into chaos, only returning to order after a period of reorganization. This period of time is usually critical in saving lives and recovering physical infrastructure. The emergency communication that occurs during this period of reorganization and recovery follows the patterns of self-organized criticality (SOC), which is an important pattern to understand in designing resilient networks [8]. While Sellnow, Seeger, and Ulmer [8] does not directly examine simulations, this research is valuable to understanding emergency response network patterns and organizational behavior.

2.3 Self-Organizing Criticality

SOC, as introduced by Per Bak, is the commonality between many analogous laws and complex systems observed in nature [4]. A notable example of SOC is the sandpile model, which demonstrates that spilling grains of sand, such as in an hourglass, will form an organized cone shape until the cone is too steep to handle more grains. When masses of sand suddenly and chaotically travel downward to form a wider base, this state is called an avalanche. This chaotic moment of reorganization occurs until the cone reaches a new and settled state. The sand returns to falling in a gradual fashion until the next critical steepness at which the system observes a tipping-point and thus another avalanche. As the sandpile grows, there is no way to predict which grain of sand will cause the next avalanche, or how large the next avalanche will be. What is known is the probability of avalanche size follows power-law behavior; smaller avalanches are more frequent than larger ones. This is one example of the recurring, scale-invariant law of self-organized criticality. Graph-based abstractions are ideal tools to capture and analyze the behavior of self-organized criticality (SOC) in physical systems [18].

2.3.1 Observed Avalanche Behavior

Self-organizing network activity, including avalanche behavior, is observed in BrainGrid's simulations. Larger whole-network events are identified as bursts in BrainGrid's research [11]. A power law, expressed as a $P \propto f^{-\beta}$, describes the relationship between avalanche probability P, and avalanche size or duration, f. Avalanche behavior is seen in a wide range of other self-organizing systems.

3 — BrainGrid: Graphitti's Predecessor

This chapter provides the necessary background of BrainGrid, the legacy simulator on which Graphitti was based. BrainGrid's framework is applied to Graphitti because of its utility and large-scale capacity. BrainGrid's purpose, architecture, design patterns, advantages, and limitations are outlined to provide context for the decisions made in Graphitti's development.

3.1 A Graph-Based Neural Simulation

The BrainGrid simulator is inherently entangled with neuroscience. It was built to model the network behavior of neurons and synapses by using the structure of a directed graph. It models network growth, internal state evolution of neurons and synapses, spike-timing dependent plasticity, spike production by neurons, spike propagation (with delays) along connections to synapses, and the effects of spikes on neuron and synapse state [1]. The nomenclature used within the BrainGrid simulator is neuroscience-specific, as are the parameters defined for individual simulations.

For the simulator to yield valid results, it is important that all of the inputs reflect the physical model accurately and at a scale large enough to reflect real-world neural behavior. While BrainGrid runs many sizes of models, an example model introduced in 2014 recurs in the lab's official simulations: 10,000 neurons arranged in a 100×100 grid, with up to 500,000 synapses (edges) created during the simulation progress. This setup uses 600 million time steps with 0.1 millisecond time steps [1], [11]. A multi-threaded version of the simulator has been tested with up to 40,000 neurons and 2 million synapses per simulation; expansion beyond that exceeded the available memory.

3.2 High Performance and Quality Assurance

BrainGrid's central goal is to facilitate simulations on two different execution platforms, Central Processing Units (CPUs) and Graphics Processing Units (GPUs), while maintaining validity and maximizing computational efficiency [19]. Utilizing GPU hardware reduces the completion time of large simulations from several months to 1-2 days. With this hardware improvement, it is possible to observe network bursting behavior and patterns of connectivity that do not occur in small network simulations [1], [19].

BrainGrid's code is written specifically for compatibility across GPUs and CPUs, which drove certain design decisions. For example, BrainGrid uses a single `AllNeurons` object instead of separate objects for each neuron [19]. If BrainGrid were designed only to run on a CPU, a different design choice, such as making each neuron its own object, could have been implemented. Although the CPU version is slower for full-scale simulations, BrainGrid maintains this code for its debugging and validation capabilities, as well as for running small simulations, which builds investigator confidence in computational correctness. (This implementation also allows BrainGrid to be run on machines without suitable GPUs.)

3.3 Simulator Design

BrainGrid is an object-oriented, graph-based discrete event and continuous (hybrid) simulator. The spikes and individual events that occur during the simulation are what make this a discrete-event simulator. The evolution of vertices' and edges' internal states are described mathematically in continuous time. At the implementation level, however, they are represented as time-steps and thus must be discrete.

3.3.1 Subsystems of the Simulator

The foundational elements of the simulator are housed in six main subsystems, shown in Figure 3.1: *Neurons, Layout, Synapses, Connections, Core, and Recorders.*

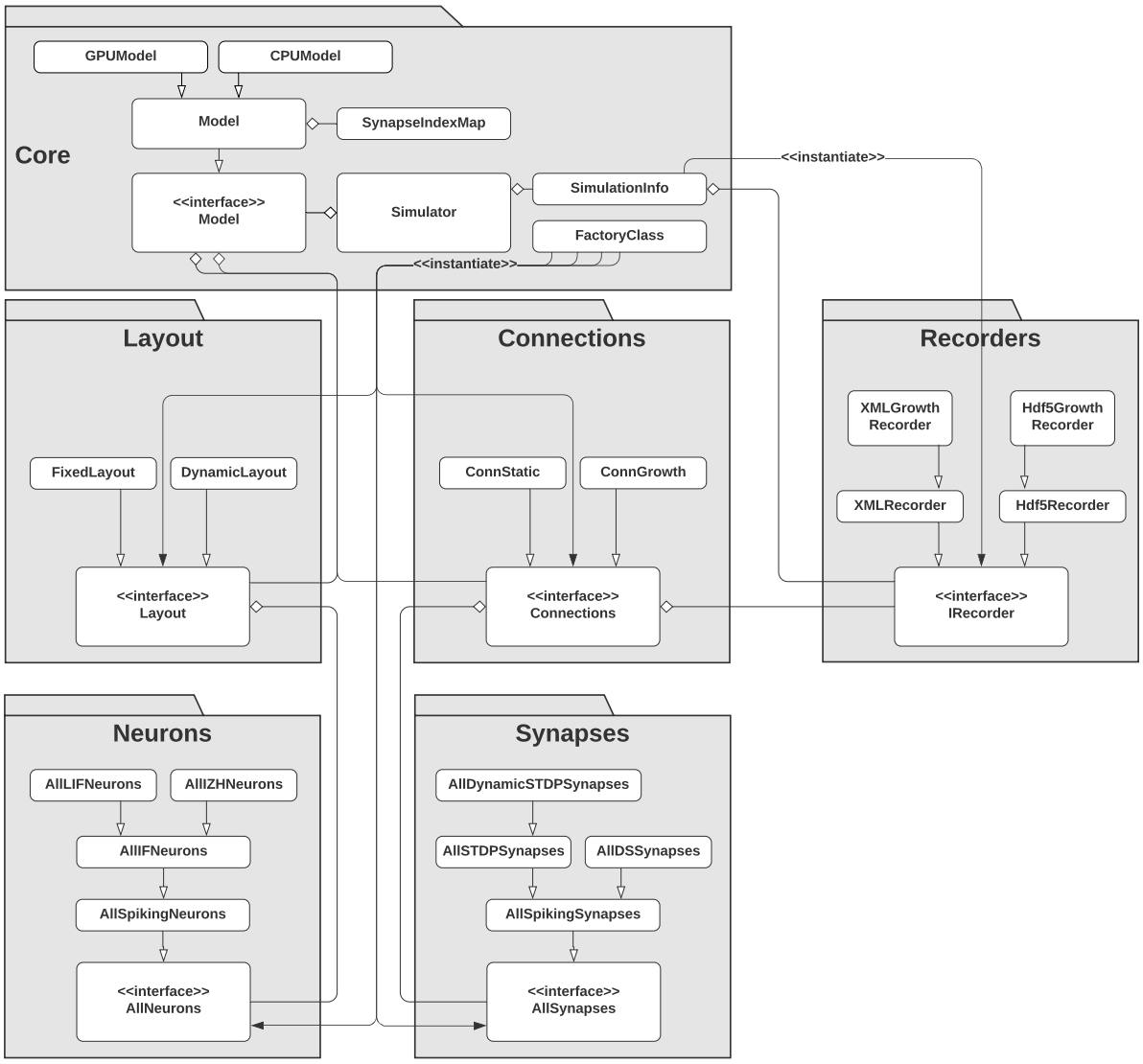

Figure 3.1: BrainGrid UML Diagram showing the six main subsystems: *Core, Layouts, Connections, Neurons, Synapses*, and *Recorders*

Neurons. The *Neurons* subsystem houses the data for the neurons (vertices). It includes an interface class and all of the subclasses, such as `AllLIFNeurons`. All neuron and their properties are instantiated before the simulation begins. They each have a type, an (x, y) location, and a state. *Neurons* is managed by *Layouts* in order to separate the individual neuron properties from the neuronal structure.

Synapses. The *Synapses* subsystem houses all synapse parameters, including the source and destination neurons associated with each synapse. This subsystem includes an interface class and allocates memory for all potential synapses at the beginning of a simulation so it does not have to allocate space during a simulation. *Connections* manages *Synapses*, separating synapse properties from their simulation behavior.

Layouts. The *Layout* subsystem manages the neuron subsystem and is the conduit between the simulator and the neurons. It creates and maintains the maps of neuron types and locations for the model throughout the simulation.

Connections. *Connections* manages the *Synapses* subsystem and is the conduit between the simulator and the synapses. It maintains the states of synaptic connections in the network during the simulation. *Connections* can be either static or dynamic, depending on the simulation. If the simulation is static, connections are initialized at setup. If they are dynamic, they will change as the network evolves.

Core. *Core* is where the simulator's main attributes live, and houses the main simulation operations. It takes in parameters, registers classes, builds the correct graph model, and allocates and deallocates memory. It is a collection of the entities that orchestrate the entire simulation. This includes the interface class, `IModel`, which sets up the network for running

on a CPU or GPU, and maintains the structure of the network via `Layout`, `Connections`, `IAllNeurons`, and `IAllSynapses`.

Recorders. The *Recorders* subsystem provides a way for data to be collected from each epoch of the simulation. It sends this data to an output destination.

3.4 Limitations and Weaknesses

BrainGrid lacks the reusability and maintainability addressed in Graphitti. Like all software, BrainGrid is susceptible to software decay, or technical debt, which is an effect of many developers, unfamiliar code, tight deadlines, and gradual changes. Technical decisions were made to accommodate short term goals and performance. BrainGrid reached a limit where the architecture was no longer modifiable. Over the course of nearly a decade, BrainGrid's incremental changes accumulated until BrainGrid outgrew its original, and, at the time, pragmatic design decisions.

Specialization. Architecturally, BrainGrid was not designed for cross-domain usability, which is a justified design decision for a neural simulator. From the highest to lowest layer, design decisions were made to accommodate the behavior of neurons and synapses in biological neural simulations. Functions, classes and variables are all written with this requirement, which is too specific for other network applications. Reusability for other domains is constrained by the entanglement of specialized information. In this environment, neurons and synapses never need delineation from their abstract counterparts, vertices and edges, and thus never get defined as such. While the nomenclature is interchangeable, inhibitory and excitatory neuron types and their pairs (synapse types) are defined and utilized at the top-level classes. This specialization also means that BrainGrid lacks features essential to other scenarios. For example, BrainGrid does not at this point in time define

the behavior of a vertex or edge based on its relative location although it is capable of doing so. This would hinder modeling geographic systems such as NG-911.

Serialization and Deserialization. Serialization and deserialization is the process by which an object is stored and recovered when the object's original data structure is not suitable for transmission or preservation. BrainGrid's architecture utilizes the Cereal library for this process, but implements serialization and deserialization on only four select objects [23]. This is a symptom of BrainGrid's accruing technical debt. The structural changes necessary to implement serialization and deserialization on the entire network are summarized in section 4.6.

Universal Factory Class. Another symptom of code decay is the gradual degrees of latitude given to each class. Originally, BrainGrid implemented one factory class and used a visitor pattern for the parameters and extracted the names of classes it needed to instantiate. It made sense to add another factory method to the existing class when necessary. This organic growth led to one large factory as BrainGrid added more categories of classes. The simulator reached a point where each category needed its own factory class.

Misnamed and Dormant Code. BrainGrid houses a collection of many researchers' contributions over a period of many years. These individual projects get compiled into one repository that uses components from these many contributors. One side-effect of these individual contributions is the gradual evolution of a method or class. Eventually, a class or method's name misrepresents its role. Another side-effect is dormancy. Placeholders for future work, abandonment of an old function, and other refactoring decisions cause code to become obsolete, causing confusion for future contributors.

3.5 BrainGrid to Graphitti

Summarizing the purpose and function of BrainGrid provides a lens through which to see the Graphitti reconstruction. In order to maintain its coupling with published papers, the legacy simulator, BrainGrid, remains accessible in its original repository [24].

Chapter 4 showcases the methods employed to build Graphitti from BrainGrid's original code base. It outlines the long-term goals for Graphitti, BrainGrid's reusable code, and areas in need of restructuring and refactoring.

4 — Methods for Developing Graphitti

Improving maintainability and providing abstraction for reusability are the two broad requirements for Graphitti's development. This project separates the graph-based network framework from the neuro-specific components [25]. This will facilitate modeling of graph-based networks in other fields. The new architecture had to maintain or better BrainGrid's characteristics in performance, quality of results, multi-platform capability, and minimal coding for new neural models. While BrainGrid exhibits some valuable qualities, its technical debt made the cost of any change or new application difficult. Graphitti preserves BrainGrid's positive characteristics and addresses its shortcomings.

This chapter outlines the process of deriving Graphitti from BrainGrid. To archive BrainGrid in its legacy form, a separate repository was created to begin the refactoring project [26]. Reusable components were incrementally brought over, refactoring improvements were made, and new components were added. Table 4.1 shows what changes were made and how they improved the software's non-functional requirements.

4.1 Graphitti Non-Functional Requirements

Reusability and maintainability are omnipresent priorities of this work. They are the drivers of all design decisions made during the development process [27]. Other non-functional requirements of Graphitti are adjacent to or nested within these requirements, shown in table 4.1. For example, Graphitti's testability is improved from organic development of unit-tests during the transferring of code.

	BrainGrid Design	Graphitti Improvement	NF Requirements
Consolidate	`IModel` and `Model` were both at the top level.	Both classes were combined into `Model`.	Maintainability
	`SimulationInfo` and `Simulator` were both at top level.	Both classes were combined into `Simulator`.	Maintainability
	`IAllSynapses` and `AllSynapses` were both at the top of the `Synapse` subsystem.	Now `Edges`, both classes were combined into `AllEdges`.	Maintainability
	`IAllNeurons` and `AllNeurons` were both at the top of the `Neurons` subsystem.	Now Vertices, both classes were combined into `AllVertices`.	Maintainability
Separate	BrainGrid primarily used raw pointers.	Graphitti transitioned to smart pointers, furthering the transition to modern C++ implementation.	Supportability Maintainability Reusability
	`SimulationInfo` was passed into every operation that needed a parameter.	A singleton design pattern instantiates `Simulator` object is now accessible via `getInstance()`.	Scalability Interoperability
	Method calls that were defined at the lower-level were shared via the SimulationInfo object.	The chain of responsibility design pattern is implemented through the operation manager, which uses the singleton design pattern.	Interoperability Maintainability
	Nomenclature consisted of neurons and synapses.	Top-level nomenclature changed to vertices and edges.	Scalability Interoperability
Replace	Domain-specific functionality was intermingled with graph-based functionality.	Every class was audited to separate any top-level graph-based functionality from domain-specific classes, or move responsibilities to appropriate classes.	Maintainability Reusability
	Subsystems were all neuro-specific.	Neuro-specific classes were relegated to their own subsystem to accommodate more domains.	Scalability Interoperability
	Flat organization of subsystems at top level.	A new subsystem layer delineates between Simulator classes and Third-Party tools.	Maintainability Maintainability
	All parameters were parsed from the config file via visitor methods.	A parameter manager enables Graphitti to load config parameters once with a set of standard methods.	Supportability Maintainability Reusability
	Core managed the instantiation of `Neurons` and `Synapses`, in addition to `Layouts` and `Connections`.	`Core` now instantiates `Layouts` and `Connections`, which each respectively instantiate `Vertices` (prev. `Neurons`) and `Edges` (prev. `Synapses`), truly using their manager class abilities.	Reusability
	`FClassofCategory` was a universal factory class.	`Factory` classes are sequestered to their respective subsystems.	Supportability Maintainability Reusability
	BrainGrid only simulated neuro-specific models.	Graphitti allows for all types of models, and implements an NG-911 test model.	Reusability Interoperability

Table 4.1: Comparison of BrainGrid and Graphitti Design Elements and Non-Functional (NF) Requirements Graphitti supported

4.2 New and Existing Tools

BrainGrid and Graphitti both run on Linux (and, CPU-only, MacOS), are written in C++11, and use C++11-compliant libraries. We carried over the embedded libraries, Mersenne Twister, Cereal, and ParamContainer, from BrainGrid and introduced TinyXPath, Log4cPlus, and Google Tests [28]–[33]. We carried over the external tools NVCC and Doxygen, and introduced CMake [26], [34]–[36].

4.3 Reusing Subsystem Architecture

The six main subsystems from BrainGrid (see Section 3.3.1) became the scaffolding for Graphitti. *Core, Layout, Connections*, and *Recorders* kept their names; *Vertices* and *Edges* replaced *Neurons* and *Synapses*, respectively, to disentangle the graph-based abstraction layer that sits atop the implementation level. *Layout* still manages *Vertices* and *Connections* still manages *Edges*.

4.4 Relegating Specialized Implementation and Elevating Graph-Based Abstraction

An abstract layer devoid of any neural specialization sets Graphitti apart from BrainGrid. The top level of each subsystem only houses general, graph-based elements. With this new paradigm, improvements to the overall simulator will *mostly* occur at the highest level. Neuro-specific nomenclature was abstracted and neural functionality was moved. `Neurons` became `vertices` and `synapses` became `edges`. Every class within each subsystem, *Core, Layout, Connections, Vertices, Edges*, and *Recorders*, was modified. Some classes were renamed to represent their abstract behavior instead of their neural equivalent and interfaces were modified to match the newly abstract operations. Figure 4.1, Graphitti's new UML diagram, reflects these changes.

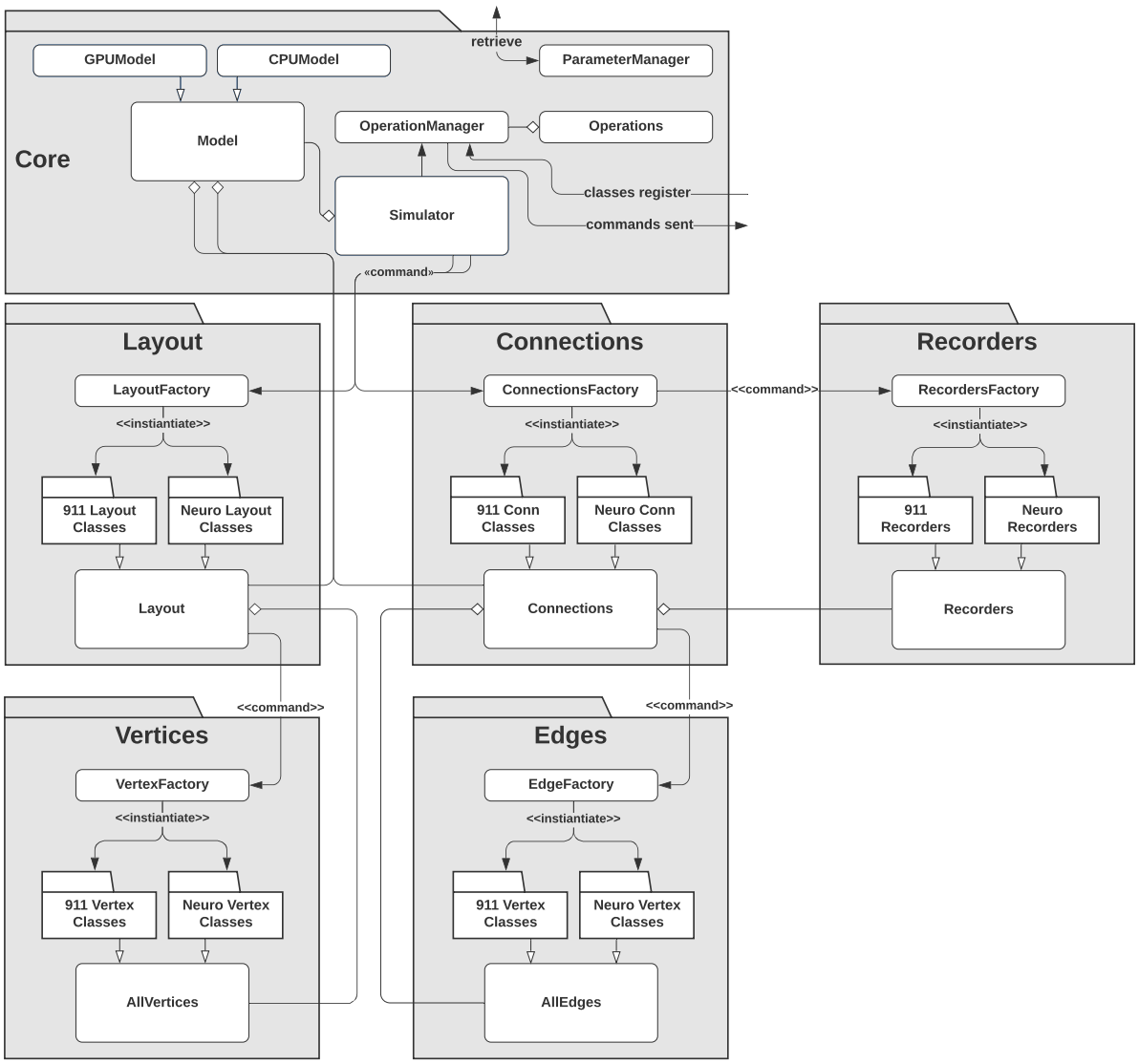

Figure 4.1: Graphitti UML diagram showing (1) the six main subsystems: *Core, Layouts, Connections, Vertices, Edges,* and *Recorders*; and (2) the discipline-specific subsystems

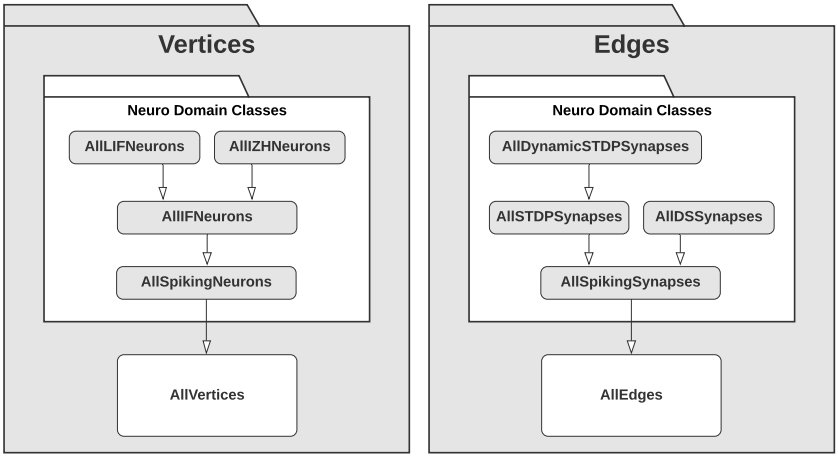

Figure 4.2: This diagram illustrates the relegation of neurospecific classes within the vertices and edges subsystems. No new classes or domains are represented.

Neuro-specific implementation was relegated to a lower level. This new domain-oriented paradigm enables future neuroscience contributions to occur in a subdomain labeled `Neuro`. Figure 4.2 details the delineation between abstract classes and domain-specific implementation within the vertices and edges subsystems prior to adding new classes or domains.

4.4.1 Compressing Redundant Polymorphism

Once the neuro-specific content was removed from top-level classes, there was no need for multiple layers of abstract classes above the implementation level. In *Core*, `IModel` and `Model` were combined. In *Vertices*, `IAllVertices` and `AllVertices` were combined. In *Edges*, `IAllEdges` and `AllEdges` were combined. `Simulator` and `SimulationInfo` were also combined, though they are not polymorphic. These classes were compressed in order to simplify the abstraction layer, as shown in Figure 4.3. The interface-class separation is a form of software decay, as it reflects a design pattern that was implemented before so many other components and functionalities were added. This refactoring decision does not change the observable behavior of the simulator, but improves the internal structure.

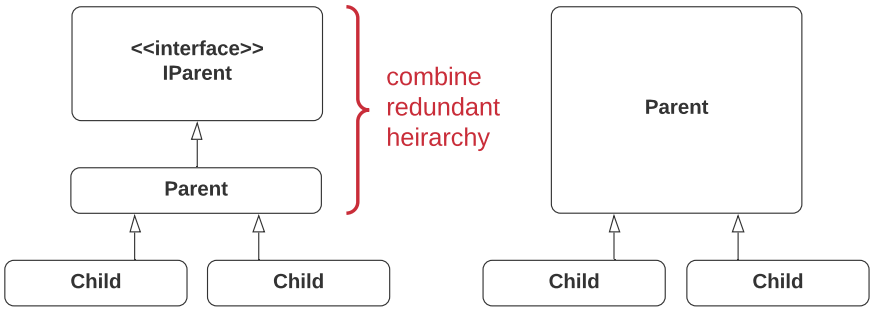

Figure 4.3: The top-level classes are compressed into a single class. Function signatures and implementations are the same, combined into one base class.

4.5 Identifying Reusable Design Patterns

Reducing the technical debt within any software is a multi-faceted process. We identified a number of design patterns that address broad issues. We describe the steps taken to pay down the technical debt outlined in Section 3.4. We employ new design patterns and topology to support the long-term maintainability of Graphitti.

4.5.1 Top-Level Factories

Individual factory classes per subsystem, reflected in figure 4.1, were developed in order to create a singleton instance of each subsystem: `Layout`, `Connections`, `Edges`, `Vertices`, and `Recorders`. The predecessor to this design pattern was an all-inclusive factory class that created the lower level classes defined for different simulations. Much like `SimulationInfo`, this class grew over time, accumulating technical debt that made changes difficult (and whose control flow was more convoluted than necessary). This new design pattern enables the top level to create the correct instances of each of the lower-level classes, specified by simulation configuration information loaded at runtime.

4.5.2 Singleton Simulator Class

Prior to implementing singleton classes, BrainGrid created the simulator object in `main()`. It uses a separate class that was accessible throughout the simulator, `SimulatorInfo`, to house parameters and share pointers to the high level subsystems. The new singleton design pattern creates a static local variable within the Simulator class itself that holds the instance, accessible via a public static method. In the simulator's case, `getInstance()` returns this single object, with accessors and mutators for the parameters that used to be accessed via `SimulatorInfo`. This greatly contributed to the simplification of top-level abstraction in Graphitti.

4.5.3 Chain of Responsibility

The chain of responsibility design pattern was implemented to perform high-level operations that are defined in lower level classes. This approach resolves the need for method calls to be shared via objects. It uses a singleton class, the `OperationManager`, to register and execute these operations [26]. The chain of responsibility design pattern is only successful for operation calls that do not rely on sequence. The `OperationManager` works closely with the `ParameterManager` by calling `loadParameters()` and `printParameters()` in all of the lower level classes at the beginning of runtime. In Graphitti's current state, `loadParameters()` and `printParameters()` are deployed. Other operations will follow in future phases of development, such as `serialize()` and `deserialize()`.

Code 4.1: Graphitti parameter loading

```
1  void AllIFNeurons::loadParameters()
2  {
3      ParameterManager& pm = ParameterManager::getInstance();
4
5      pm.getBGFloatByXpath("//Iinject/min/text()", IinjectRange_[0]);
6      pm.getBGFloatByXpath("//Iinject/max/text()", IinjectRange_[1]);
7
8      pm.getBGFloatByXpath("//Inoise/min/text()", InoiseRange_[0]);
9      pm.getBGFloatByXpath("//Inoise/max/text()", InoiseRange_[1]);
10
11     pm.getBGFloatByXpath("//Vthresh/min/text()", VthreshRange_[0]);
```

```
12      pm.getBGFloatByXpath("//Vthresh/max/text()", VthreshRange_[1]);
13
14      pm.getBGFloatByXpath("//Vresting/min/text()", VrestingRange_[0]);
15      pm.getBGFloatByXpath("//Vresting/max/text()", VrestingRange_[1]);
16
17      pm.getBGFloatByXpath("//Vreset/min/text()", VresetRange_[0]);
18      pm.getBGFloatByXpath("//Vreset/max/text()", VresetRange_[1]);
19
20      pm.getBGFloatByXpath("//Vinit/min/text()", VinitRange_[0]);
21      pm.getBGFloatByXpath("//Vinit/max/text()", VinitRange_[1]);
22
23      pm.getBGFloatByXpath("//starter_vthresh/min/text()", starterVthreshRange_[0]);
24      pm.getBGFloatByXpath("//starter_vthresh/max/text()", starterVthreshRange_[1]);
25
26      pm.getBGFloatByXpath("//starter_vreset/min/text()", starterVresetRange_[0]);
27      pm.getBGFloatByXpath("//starter_vreset/max/text()", starterVresetRange_[1]);
28  }
```

4.5.4 Parameter Manager

We implemented the `ParameterManager` class to provide an interface for accessing the XML parameter file (simulator configuration information) at runtime [26]. This class uses XPath to wrap a (key, value) pair interface around the TinyXML file representation. By providing this manager, all classes can load their parameters with a uniform set of standard methods, as shown in code snippet 4.1 from the `AllIFNeurons` class. This eliminates the need for each such class to implement visitor methods to structurally parse its parameters from the XML, greatly reducing the amount of coding for each class that has parameters. Code snippet 4.2 shows how the `AllIFNeurons` class requires around 90 lines of code, compared to around 20 lines in Graphitti's implementation.

Code 4.2: BrainGrid parameter loading

```
1  bool AllIFNeurons::readParameters(const TiXmlElement &element) {
2      if (element.ValueStr().compare("Iinject") == 0 ||
3          element.ValueStr().compare("Inoise") == 0 ||
4          element.ValueStr().compare("Vthresh") == 0 ||
5          element.ValueStr().compare("Vresting") == 0 ||
6          element.ValueStr().compare("Vreset") == 0 ||
7          element.ValueStr().compare("Vinit") == 0 ||
8          element.ValueStr().compare("starter_vthresh") == 0 ||
9          element.ValueStr().compare("starter_vreset") == 0) {
10         nParams++;
11         return true;
12     }
13
14     if (element.Parent()->ValueStr().compare("Iinject") == 0) {
15         if (element.ValueStr().compare("min") == 0) {
```

```
16              m_Iinject[0] = atof(element.GetText());
17          }
18          else if (element.ValueStr().compare("max") == 0) {
19              m_Iinject[1] = atof(element.GetText());
20          }
21          return true;
22      }
23
24      if (element.Parent()->ValueStr().compare("Inoise") == 0) {
25          if (element.ValueStr().compare("min") == 0) {
26              m_Inoise[0] = atof(element.GetText());
27          }
28          else if (element.ValueStr().compare("max") == 0) {
29              m_Inoise[1] = atof(element.GetText());
30          }
31          return true;
32      }
33
34      if (element.Parent()->ValueStr().compare("Vthresh") == 0) {
35          if (element.ValueStr().compare("min") == 0) {
36              m_Vthresh[0] = atof(element.GetText());
37          }
38          else if (element.ValueStr().compare("max") == 0) {
39              m_Vthresh[1] = atof(element.GetText());
40          }
41          return true;
42      }
43
44      if (element.Parent()->ValueStr().compare("Vresting") == 0) {
45          if (element.ValueStr().compare("min") == 0) {
46              m_Vresting[0] = atof(element.GetText());
47          }
48          else if (element.ValueStr().compare("max") == 0) {
49              m_Vresting[1] = atof(element.GetText());
50          }
51          return true;
52      }
53
54      if (element.Parent()->ValueStr().compare("Vreset") == 0) {
55          if (element.ValueStr().compare("min") == 0) {
56              m_Vreset[0] = atof(element.GetText());
57          }
58          else if (element.ValueStr().compare("max") == 0) {
59              m_Vreset[1] = atof(element.GetText());
60          }
61          return true;
62      }
63
64      if (element.Parent()->ValueStr().compare("Vinit") == 0) {
65          if (element.ValueStr().compare("min") == 0) {
66              m_Vinit[0] = atof(element.GetText());
67          }
68          else if (element.ValueStr().compare("max") == 0) {
69              m_Vinit[1] = atof(element.GetText());
70          }
71          return true;
72      }
73
74      if (element.Parent()->ValueStr().compare("starter_vthresh") == 0) {
75          if (element.ValueStr().compare("min") == 0) {
76              m_starter_Vthresh[0] = atof(element.GetText());
77          }
78          else if (element.ValueStr().compare("max") == 0) {
79              m_starter_Vthresh[1] = atof(element.GetText());
80          }
81          return true;
82      }
83
```

```
84      if (element.Parent()->ValueStr().compare("starter_vreset") == 0) {
85          if (element.ValueStr().compare("min") == 0) {
86              m_starter_Vreset[0] = atof(element.GetText());
87          }
88          else if (element.ValueStr().compare("max") == 0) {
89              m_starter_Vreset[1] = atof(element.GetText());
90          }
91          return true;
92      }
93      return false;
94  }
```

4.6 Improvement Accumulation

BrainGrid documented issues as they arose, which exposed a portion of the emerging software decay. This documentation was a valuable asset in forming the paradigm for building Graphitti. The transition itself further exposed previously unidentified weaknesses that influenced prioritization during the development process. While their individual influence on the project is unremarkable, the sum of their outcomes is noteworthy.

One improvement during this project concerns the refactoring of all raw pointers to smart pointers. A prior initiative in BrainGrid's software development was to fully implement serialization and deserialization: converting objects to and from a byte stream for transmission and reconstruction [23]. For this, BrainGrid and Graphitti utilize the Cereal library. In order to fully implement this process with Cereal, BrainGrid's raw pointers needed to be converted to smart pointers. BrainGrid only implemented serialization and deserialization on a handpicked selection of objects in order to demonstrate its viability. The remainder of implementing serialization and deserialization on all objects will occur when the GPU version of Graphitti is underway. These processes will then take place in the `OperationManager`.

Disentangling the code had the unforeseen advantage of making small improvements that had previously not been in the scope of the project. A computational improvement was made in the `advance()` heirarchy. Previously, BrainGrid cast `allEdges` in `advanceNeuron()`,

a neuro-specific advance method that is called by `advanceNeurons()`, now named `advanceVertices()`. This was an expensive decision, because the class was cast in a method that is repeatedly called. Graphitti now casts `allEdges` in `advanceVertices()` above the `advanceNeuron()` method to save the number of unnecessary times it casts. This decision significantly reduced computation time dependent on the size of the simulation.

4.7 Development Challenges and Limitations

Some challenges and limitations during Graphitti's development process were foreseen, and others were exposed during the process. Changing scope, uncovering further software decay, and managing resources all contributed to the limitations of this project.

- Individual contributor availability influenced sprint planning and prioritization. Many individual features of Graphitti were developed asynchronously, periodically merging with other work. For example, `ParameterManager` and `Serialization/Deserialization` were both implemented early in the design phase of Graphitti, before the new repository was created.

- The chain of responsibility design pattern cannot work on operations that are sequence-dependent. This caused a design change for certain operations, such as `setup()`, which is sequentially designed.

- This refactor had to consider the successive nature of several sub-projects; one part must be completed before the other begins. Some standalone sub-projects are yet to be implemented if they required significant code preparation, and are summarized in section 6.3. For example, the `OperationManager` class only implements two of eight planned operations.

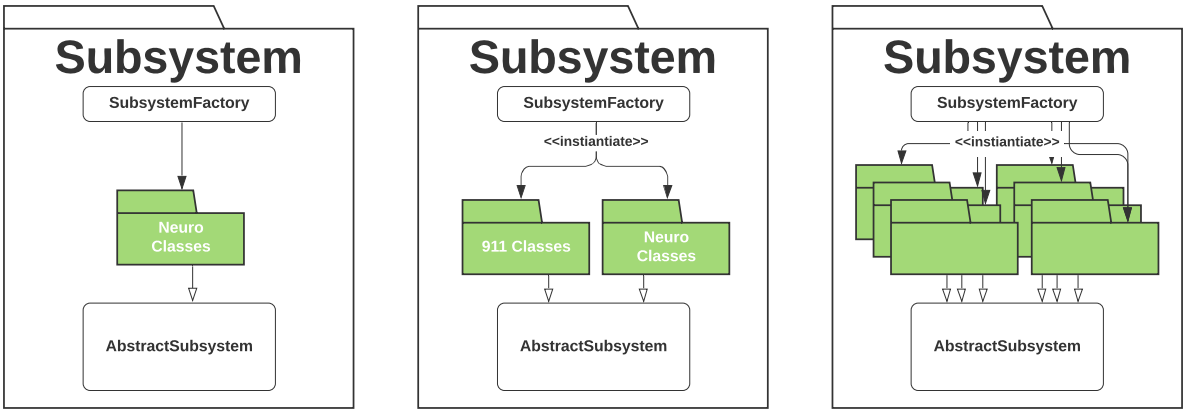

Figure 4.4: Graphitti was designed to have a uniform structure within each subsystem regardless of how many domains are added. Prior to implementing the *NG-911* domain, only the *Neuro* domain existed. This topology allows for new domains to be added without affecting the greater system functionality, increasing configurability.

4.8 Designing for New Scenarios

Preparing to demonstrate a specific scenario for a non-neural network required a significant amount of development. This chapter summarizes the steps taken to separate domain-specific code into lower level classes and preserve abstract graph behavior at a higher level. Figure 4.4 represents three versions of each subsystem's structure. The single Neuro subdirectory reflects the structure before NG-911 was added. The current form of Graphitti is characterized in the center, which corresponds to the uml diagram in figure 4.1. Graphitti can keep adding domains and use the same architecture.

The steps to build out any new subdomain are:

1. Identify a characterizing domain title.

2. Create a domain structure at same level as Neuro and NG-911 subdirectories within Layouts, Connections, Recorders, Vertices, and Edges.

3. Use domain title to create subclasses (.h and .cpp). This can get as detailed as needed. (Example: `Connections/DomainTitle/AllDomainTitleHereConnections.{h,cpp}`)

4. Create a configuration file and modify it to fit a small test scenario. Start small for testing.
   (Example: `configfiles/test-small-domainTitle.xml`)

5. Identify and customize the domain and scenario's parameters.

6. Write the `advance()` methods within Vertices and Edges, and the `updateConnections()` method within Connections. This will likely require helper methods.

Chapter 5 outlines the process for defining and building the NG-911 model by utilizing the abstract layer.

5 — Case-Study: Modeling Next-Generation 911 Connectivity

This chapter summarizes the design, implementation, and validation of the NG-911 test-case. The results of this test-case are *not* the conclusions of the NG-911 research, but a step towards building a full-scale implementation. The goal of this test-case was to establish a minimum-viable-product with which to move forward in NG-911 research.

5.1 Scenario Development

While working on building the abstract layer of Graphitti and separating the neuro-specific elements in a lower level, we had to determine what aspects of the real-world NG-911 system should be modeled in a graph-based network, and what would be safe to omit. We held meetings with groups from the Washington State's NG-911 department, NORCOM, Seattle Fire Department, and King County. By gaining access to individuals within IT, dispatch, first-response, and government, our team was able to gain firsthand knowledge of their needs for the simulator.

Stakeholder meetings focused on determining the exact scenarios that threaten the existing emergency response network and how improvements would impact response capabilities for real-time events. During the design phase, we maintained a document of potential questions and scenarios to explore. The scenarios were classified based on what was feasible to model, the availability of relevant data, and the audiences that would be interested in the results. The questions were labeled as one or more of six different categories: exploratory, growth, structure, cybersecurity, knowledge, or performance. This document was refined to four questions and seven scenarios reflected in Figure 5.1. All four final questions were

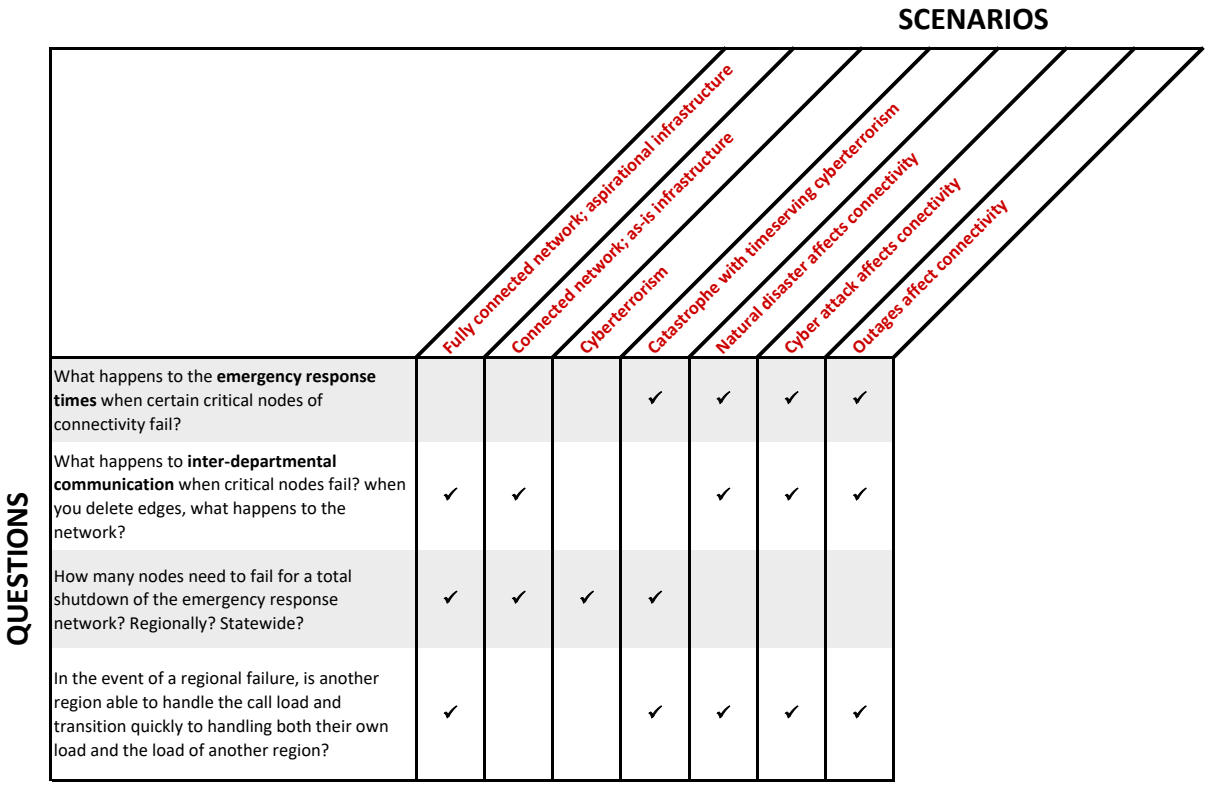

Figure 5.1: Collaboratively-developed scenario to question map to arrive at the optimal first test for Graphitti

concerned with performance and structure. *Performance* represents the ability of the NG-911 emergency response network to perform effectively in events of varying scales. *Structure* refers to the design of the NG-911 operations and infrastructure. These questions helped us define what each node and edge would represent and helped us establish an initial scenario to model.

5.1.1 Modeling the Real-World

It is important to note that emergency response cannot, by definition, precede certain events. The 2021 Texas Polar Vortex, a case study described in Appendix A, spawned unpredictable crises that serve as a valuable road-map for what stakeholders should expect in an unprecedented catastrophe. The emergency response network was only able to use what remained of Texas' existing infrastructure and regional preparedness through the storm. Any number of systematic strains and stressors brought on by crises must be anticipated through this type of event.

Stakeholder meetings influenced the decision to model one specific scenario: the resilience of NG-911 in the long-anticipated scenario of a Cascade Fault-Line earthquake [37]. The Cascadia region would be largely impacted by an event like this, and it is vital that the emergency response network of the region is prepared. The network is reliant on physical infrastructure that, in the scenario of a destructive earthquake, would be massively compromised if not upgraded and maintained. A disaster at this scale has the potential not only to compromise both physical response infrastructure and technological connectivity. Much like the Texas Polar Vortex, it would push rerouting, connectivity, and redundancy to its limits.

5.2 Model Setup

We implemented a small-scale model to demonstrate the destruction of emergency resources during the Cascade Earthquake. At the abstract level, this entails deleting vertices that

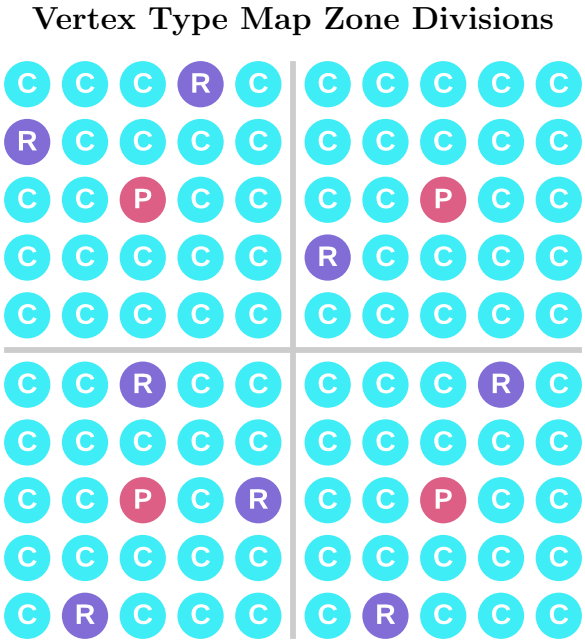

Figure 5.2: 10×10 grid of vertices displaying PSAPs (P), responders (R), and callers (C)

represent PSAPs and resource pools of emergency responders being removed. We configured a simulation that, despite its relative simplicity, shares many key connectivity and response features that we need to study in the real world.

5.2.1 Vertices

The product designed for the first iteration of the NG-911 simulation during a crisis is comprised of a 10×10 grid of caller pools, responder resource pools, and PSAP dispatcher pools, as shown in Figure 5.2. We started only with the essential elements of the model, as it is much easier to add complexity once the simplified version is successful [10]. After testing at this stage, the addition of more complex features will enhance the simulator's capabilities to model more intricate scenarios.

5.2.2 Edges

The grid is divided into four 5×5 zones, with each PSAP placed at the middle vertex of each zone. Each caller is connected to their zone's PSAP. PSAPs have a cap on how

many callers can come in. If responders are fully occupied in that zone, the call will be rerouted to the next closest PSAP in order to reach the next closest responders. If there are available responders within the zone, the caller waits.[1] The edges connecting the responder to the caller are precalculated by Euclidean distance. This helps the dispatcher identify which responders to dispatch. There are between two and three responder vertices per zone, totaling nine responder vertex pools throughout the network. These responder vertex pools are connected to every caller, in order to respond to any emergency sent their way. The NG-911 implementation of `createEdgeIndexMap()` sorts edges from closest to farthest.

Algorithm 1: `All911Vertices::advanceVertices()`: called at each timestep

```
for i ← 0 to all_911_Vertices do
    if vertex_type = PSAP then
        advancePSAP();
    end
    if vertex_type = RESP then
        advanceRESP();
    end
    if vertex_type = CALR then
        advanceCALR();
    end
end
```

Algorithm 2: `Connections911::updateConnections()`: called at epoch/crisis

```
for i ← 0 to PSAPs_to_erase do
    if Number of PSAPs > 1 then
        erasePSAP();
    end
end
for i ← 0 to RESPs_to_erase do
    if Number of RESPs > 1 then
        eraseRESP();
    end
end
```

[1] This is a preliminary implementation that does not reflect the final PSAP-Responder connection in a full-scale implementation

5.2.3 Implementation

Graphitti's top-level implementation is uniform across simulations. Each timestep calls `advance()`, which calls `advanceEdges()` and `advanceVertices()`, and each epoch calls `updateConnections()`, summarized in Alg 2. The NG-911 scenario built `advanceVertices()` to call `advancePSAP()`, `advanceRESP()`, and `advanceCALR()`, as shown in Alg 1. `advanceEdges()` uses the `EdgeIndexMap` to reach the state of each edge at each timestep.

Algorithm 3: `Connections911::erasePSAP()`

```
pick random PSAP from vertexTypeMap;
for each edge connected to PSAP do
    record edge;
    mark connected vertex as orphaned;
    delete edge;
end
delete PSAP;
for each orphaned vertex do
    find closest PSAP by absolute distance;
    connect vertex to PSAP;
end
```

The first scenario design is comprised of two epochs: one before a crisis and one after. Prior to the first epoch, the simulator model is configured with the network connectivity that would exist prior to the interference of a crisis, as shown in Figure 5.3. Upon the next epoch, `updateConnections()` is called to represent the catastrophe that deletes a predefined number of responder and PSAP vertices. `eraseEdge()` calls `erasePSAP()` and/or `eraseRESP()`, summarized in Alg 3 and Alg 4, which deletes the edges associated with the deleted vertices in order to simulate their unavailability in a catastrophe. Figure 5.4 represents the network fragmentation that would occur if callers were not connected to new resources. A new edge is created between a caller and the closest remaining PSAP in the epoch representing the aftermath of the catastrophe, shown in Figure 5.5. Section 5.4 summarizes these findings and contextualizes their value to the overarching project.

Algorithm 4: `Connections911::eraseRESP()`

pick random RESP from `vertexTypeMap`;
for *each edge connected to RESP* **do**
    record edge;
    delete edge;
end
delete RESP;
`// RESP-CALR edges are fully connected and do not need rewiring`

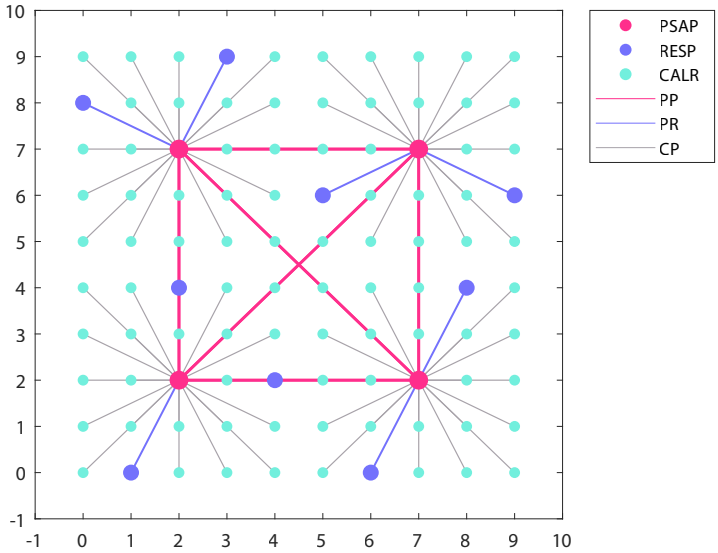

Figure 5.3: 10×10 grid of vertices displaying connections between PSAPs, responders, and callers

5.3 Contrasting NG-911 and BrainGrid

BrainGrid and Graphitti's NG-911 implementation only share the abstraction layer which facilitates the simulation. The setup, parameters, low-level functions, classes, and behaviors within each model are unique to each implementation. Some of these differences are notable to discuss due to the impact they have on transforming the simulator from one domain to another (and, conversely, what interface is common to all domains).

5.3.1 Imposing Behavior into the Scenario

The crisis in the NG-911 model is an external event that is injected into the system during the simulation. BrainGrid's neural simulations model the internal stimuli and response of a brain, thus the graph itself generates avalanches and bursts. This model requires manually stimulating the system with inputs. This injection of chaos is representative of a catastrophe, like the Cascade Fault-Line earthquake, in an otherwise organized environment. Results and observations from this setup will emerge in a full-scale model.

5.3.2 Leveraging Scale Invariance

Graphitti is scale-agnostic, which allows it to simulate behaviors of any scale and any granularity of time. An abstract, hybrid, graph-based network can model any domain of varying spatiotemporal characteristics, given the appropriate parameters. For example, the size of timesteps in the *Neuro* model, is equivalent to 0.1 milliseconds. For the observation of neural behavior, this is proportionally appropriate. Milliseconds are unnecessary to model NG-911, and therefore, we can use a larger time-step to observe behavior.

5.4 NG-911 Testbed Results

The simulation described in section 5.2.3 was run and the edge information was recorded. This was visualized with MATLAB to produce figures 5.3–5.5. The results of this small-scale test-bed reflect the functionality of each method and demonstrate the model's speculative scalability for larger simulations. Figure 5.3 represents the simulation setup to model a small scale emergency response network with four jurisdictions and resources within each. It displays the connections between vertices, stored in the `edgeTypeMap`, before a crisis which takes place at the first epoch. Since that epoch deletes vertices, edges that connect callers to resources must be addressed. The changes that the simulation makes to the network are stored in the edge type map.

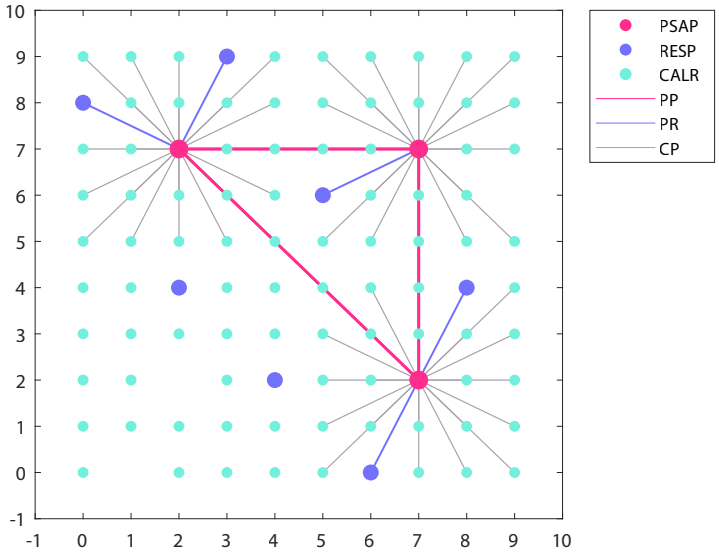

Figure 5.4: 10×10 grid of vertices displaying connections between PSAPs, responders, and callers

Figure 5.4 reflects the isolation of callers to resources if they are not provided new connections. Figure 5.5 reflects the implemented rewiring of edges that ensures connectivity for callers. According to previous research, these simulations at a larger scale will demonstrate the patterns of SOC [8]. The model design should demonstrate how improvements in connectivity would drastically improve the ability of the system to respond to such an event.

5.5 Limitations and Issues of NG-911 Implementation

The limitations described in this section relate to challenges during the development process that had to be overcome in order to deliver a minimum viable product. Some limitations will be addressed in future research, outlined in Section 6.3.

- One of the most noted limitations of this setup is its size relative to the intended model. The GPU implementation of this scenario will enable a larger, more representative model that will allow an in-depth analysis of results. The CPU implementation of a

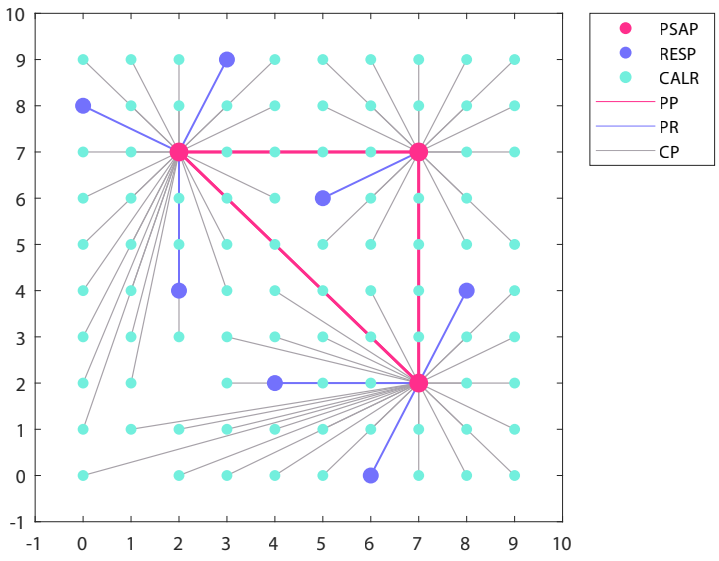

Figure 5.5: 10×10 grid of vertices displaying connections between PSAPs, responders, and callers

small-scale test case provides a foundation for this work, but is not large enough to observe system behaviors.

- Data collection for real-world catastrophic events was unsuccessful up to this point. E-911 data has only recently been collected in select local jurisdictions, and the data provided varies. When we were able to source any datasets, none included large-scale catastrophes. After reaching out to jurisdictions in Texas for data from the Polar Vortex, it was apparent that our data requests would take a significant amount of time. Future work will include collecting this data for a proper comparison of results.

- Due to this lack of data, it was impossible to validate the model's behavior with scientific methods. Stakeholder interviews, surveys, news articles and publications provided a patchwork validation method usable for this stage of implementation.

- Because of the external stakeholders in this project, it is important to deliver a model with unambiguous representations of real-world entities. The simulator's inputs and

outputs must be understandable and usable for a non-technical audience. This limited the level of abstraction within the NG-911 model.

6 — Discussion

This report presents the value of graph-based network simulation for a variety of uses. It introduces BrainGrid, a neural simulator, which was the foundation for Graphitti. The findings in this paper prove the viability and benefits of abstracting an existing, domain-specific simulator. BrainGrid experienced software decay that made it difficult to modify and maintain, which was addressed in the development of Graphitti. The top abstraction layer introduced in Graphitti also provided a paradigm to implement scenarios in other domains, as exemplified with the NG-911 model. This improved the adaptability and reusability of the software.

Because the behavior of every model is unique, it requires its own implementation layer below the abstraction layer. The NG-911 model was implemented as a sibling layer to BrainGrid's neuro-specific architecture. The test scenario demonstrates the functionality of the top level abstraction to setup, build, run, and complete a simulation. These results should not be interpreted as the culmination of the research, but as a demonstration of success within the test-case.

6.1 Graphitti's Potential for Complex Modeling

This section revisits the literature supporting the similarities between complex systems discussed in Chapter 2. These scenarios are unified by their potential to be modeled via graph-based network abstractions.

Early research by Mikler, Wong, and Honavar [22] explains how communication networks are typically very large with a high level of connectivity. Due to the complexity of such

networks, any simulator must be highly computationally efficient to create a successfully realistic model.

6.1.1 Approaches to Crisis Modeling

Existing simulations of the E-911 system provide evidence that the NG-911 system can be modeled with a graph-based network. Research done by Mirsky and Guri [15] demonstrating an E-911 DDoS attack meets the criteria of a crisis as it is surprising, threatening, and requires a short response time [3], [15]. The NG-911 model is similar to the E-911 DDoS simulations in its demonstration of resource paralysis. Previous simulations do so by flooding the system with calls, while this model removes the actual response resource. Unlike the model developed in Chapter 5, the E-911 DDoS simulations are not set up to model the recovery behavior of a network after a debilitating event [15].

6.1.2 Self-Organizing Behavior in NG-911

Based on evidence in previous research, we hypothesize that a full-scale implementation of the NG-911 scenario will bear self-organizing behavior similar to the findings of neural simulations [4], [8], [11]. Developing a large-scale model of the emergency response network that includes the recovery period after a catastrophe will allow us to confirm this.

In the case of the Texas Polar Vortex, for example, a 911 call that derives from the 133-car pile-up depends on weather causing icy roads, which causes an accident [12]. This obfuscates the delineation between dependent and independent variables as separate entities in a system of interdependent, self-organizing cause and effect. With this perpetual stimulus-response feedback loop, the behavior can only describe independence of a variable at the time of a certain event [4]. In self-organizing systems, this is called*allometric growth*, which describes the proportional relative growth of systems within a larger system. This same assumption would conclude that there would not be accidents of the same magnitude and frequency under normal conditions.

We anticipate the observation of the emergent patterns of SOC in a large scale simulation. Confirmation of these patterns in future simulations in Graphitti would greatly improve the theoretical foundations of our propositions for the NG-911 system. At this point in time, these foundations are all based on other research [4], [8].

6.2 Consequential Non-Functional Improvements

While improving maintainability and reusability were the main objectives of developing Graphitti, other non-functional requirements were introduced or improved over the course of the project. Four main categories of non-functional requirements were improved during this process: (1) development requirements, (2) testing requirements, (3) scenario requirements, and (4) performance requirements. These altogether improve software quality and lead to unanticipated outcomes that positively impact the architecture [38]

6.2.1 Development Requirements

As the maintainability improved, the life-cycle cost of development was greatly reduced. This was evident in developing the NG-911 implementation. These benefits are closely related to the modifiability of the code and project management during the development life-cycle. Code readability and documentation quality are side effects of the project's broad scope. These improvements will likely influence the kind of support new contributors are able to provide in the INL.

6.2.2 Testing Requirements

While testing was beyond the scope of this paper, it is worth mentioning the merits of testability in the context of software requirements. During the rigorous undertaking of rebuilding the entire simulator, tests were concurrently written to ensure faults and failures were mitigated along the way. The scale of design changes were so immense, that the

importance of this process contributes to improved fault tolerance and failure management in Graphitti.

6.2.3 Scenario Requirements

By abstracting the code for reusability and implementing new design patterns throughout the software, Graphitti is more configurable and flexible than BrainGrid. This will aid in the extensibility of the software over time, and contribute to a robust suite of capabilities. It could also ease the development of a multi-GPU version of the simulator, which would allow a speed-up at runtime for large simulations.

6.2.4 Performance Requirements

During the development process, refactoring decisions were made across the simulator that cumulatively improved both run-time and memory usage. While the analysis needed to validate performance improvements is yet to be completed, they will be documented at a later stage of Graphitti's development and are already underway. The observations currently influencing this quality improvement are individual lines of code being refactored or moved and the replacement of `SimulationInfo` across all classes with `Simulator.getInstance()`. These changes alone have optimized memory allocation, efficiency, and effectiveness of the simulator.

6.3 Present Limitations and Future Work

Many of the limitations relevant to this stage of development will be addressed in future work. Building the abstract layer and NG-911 layer exposed many areas for improvement and incomplete implementations. Despite the need for future improvement, Graphitti's software development life cycle costs are already greatly reduced from BrainGrid's. Upcoming improvements are more manageable with the quality improvements brought fourth by this project.

GPU Implementation. While this paper only demonstrates the CPU-based implementation of the NG-911 model, the INL maintains a GPU infrastructure to yield over a 50× run-time speedup. The primary benefit of repurposing the BrainGrid framework is its implementation speed [19]. Networks with thousands of vertices and potentially hundreds of thousands of edges will rely on software performance to be as optimal as possible. The GPU-based implementation of the NG-911 model is part of an upcoming phase of ongoing research.

Runtime Comparisons of BrainGrid and Graphitti. Because Graphitti can still run all of the scenarios developed for BrainGrid, it would be valuable to break down certain processes and measure their runtime differences. Section 4.6, for example, addresses the computational improvement of a simple casting placement decision that improves runtime, but the improvement was not tested in a side-by-side comparison at full-scale. The change was made before completion of Graphitti, which prevented an official comparison.

Serialization and Deserialization. Serialization and deserialization is only implemented on a selection of objects. Full implementation of these actions is slated for future development.

Operation Manager Completion. At this point, `OperationManager` is only implemented on two operations. Once serialization and deserialization are complete, the operation manager will also handle them.

Geospatial Modeling. A GIS-compatible version of Graphitti would enable a variety of capabilities not currently in the simulator. This implementation would apply to many possible domains of the simulator, making it a crosscutting component that could sit in

between the abstraction layer and implementation layer. This implementation could be two- or three-dimensional, as well. Currently, the simulator is two-dimensional.

Provenance. Data provenance is a large component of scientific research. Workbench [19] is the provenance software designed to work with Graphitti and requires comparable updates in order to work with the new architecture.

Large-Scale Model Development. Related to the GPU implementation is the large-scale NG-911 model that would accompany it. This would require design decisions such as how long each time-step would equate to in real time, how many callers, responders, and PSAPs would be modeled, how many zones would be implemented, and how many epochs would be required to fully model recovery of the system. Further complexities might include simulating an entire region with border concerns. What happens in a chaotic region when connectivity is increased on one side of the dividing line? A large-scale model will enable researchers to pursue such questions that cannot be explored in the smaller model.

Delays and Noise in NG-911. The NG-911 model is still in an elementary phase of development. It must implement delays between responders and dispatches to reflect travel time, among other various forms of noise. This also provides an opportunity to implement the chance of a DDoS attack during the simulation. Likelihood of the attack could increase during the catastrophe epoch.

DDoS in NG-911. Graphitti does not yet model telcom routers, phone type, and adversarial botnet flooding. Developing a version of the model that includes an adversarial DDoS attack and varying levels of defense against it could be modeled in larger simulations.

NG-911 Recorders Classes. This project did not fully implement recorders classes for the NG-911 model. This will be included in an upcoming phase of development.

NG-911 Results and Metrics. The scope of the project did not focus on the results or metrics of the NG-911 model. Therefore, we have not yet developed a way to quantify the connectivity improvements of the network. This will be an important part of future NG-911 research. It relies on real world data, full-scale modeling, and full implementation of the recorders classes.

Vertex Mobility. At is time, the model does not enable vertices to move during a simulation. In order to realistically model the NG-911 system with hyper-accuracy, vertices, especially responders and callers, must be able to move around. This would require recalculation of distances in connections that would multiply the computational demands significantly.

6.4 Research Applications

Building a top-level abstraction allows Graphitti flexibility in designing new models for different domains. While the first two domains represented in Graphitti are neuroscience and emergency response, there is now the opportunity to implement an infinite number of scenarios.

6.4.1 Emergency Response Research

The NG-911 application of the simulation can be used as a tool in improving regulation and legislation to meet the emergency response needs of the population. The simulator has an opportunity to be an ongoing resource by exposing the need for a periodic re-calibration of emergency response infrastructure. By the output of even the small-scale scenario where the crisis cuts communication (Figure 5.4) and a scenario where the crisis

redirects communication (Figure 5.5), we can see that the increase in connectivity of a response network can save lives.

7 — Conclusion

This research demonstrates the success of reusing a domain-specific simulator, BrainGrid, to build a general purpose simulator, Graphitti. Graphitti leverages its capabilities for other types of complex networks by separating the neuroscience-specific code from the simulation code. The simulator is now capable of modeling scenarios from any domain, expanding its usability to a wide range of researchers. It preserves the valuable components of BrainGrid, such as its computational efficiency, GPU-CPU flexibility, and graph-based implementation. In addition to expanding its modeling abilities, Graphitti also deploys new design patterns, and addresses the compromising software decay accumulated in BrainGrid. These improvements allow for a streamlined approach to implementing new scenarios.

Its new configuration was demonstrated by modeling the NG-911 emergency response network. Although its demonstration was a small-scale prototype, the NG-911 results achieved the originally stated goal of representing the value of connectivity during catastrophe-driven resource depletion. Graphitti's NG-911 implementation demonstrates how influential a node's presence, connectivity, and behavior is on the network. This model simulates the real-world implications of emergency response connectedness in a small-scale test scenario. It verifies the reusability of the simulator for different domains.

While the first implementations are neural networks and emergency response networks, Graphitti's goal is to be accessible to all large complex graph-based networks. This work presents the initial phases in pursuit of realizing this objective.

Appendices

A — Example Crisis: 2021 Texas Polar Vortex

The February 2021 deep freeze in Texas is a good demonstration of how a regional crisis results from a large event causing many small emergencies, which demand even more from the already strained response network. In early February, a weather forecast warned the US of extremely low temperatures across the country due to a latitude drop in the polar vortex, which usually remains above the arctic. Below-freezing temperatures caused a myriad of issues across a region not typically prepared for such conditions, including icy roads, blackouts, and a frozen water supply.

Cases like this can provide insight for how to improve the preparedness of a region for catastrophic events. This case study, illustrated in figure A.1, yields an excellent real-world example to use for node deletion. Resources fell offline gradually and the region became more dependent on emergency response. The volume of well-documented 911 data for various sub-categories within the larger crisis makes this an excellent reference for our study.

Texas experienced some major incidents as a result of the polar vortex, including a 133-car pile-up and hundreds of carbon monoxide poisoning cases [12], [13]. The 133-car pile-up itself was, in the perspective of emergency dispatchers, a collection of smaller accidents. Around 7 a.m., icy roads caused large long-haul trucks to lose control and thus cause this fatal mega-crash. By 10 a.m., the Grand Prairie Police Department responded to over a dozen major accidents and over four dozen minor ones. Six people died and at least 36 were hospitalized due to injuries.

For the majority of the country, energy infrastructure is interconnected and thus most states are able to borrow energy from other sources on the grid if theirs fails. Texas, however, had privatized its grid and separated it from the rest of the country. This lack of connectivity was deadly to many. There were over 500 reports of carbon monoxide poisoning in Houston alone from people using gas-powered heat typically unsafe for indoor use. This was a direct result of infrastructure failure stemming from a lack of connectivity to the rest of the nation's power grid, and is a perfect case study for how connectivity influences disaster response.

The diagram in figure A.1 delineates the point at which emergency responders can be contacted. Unfortunately, response teams have no way of prematurely knowing the precise severity and locations of emergencies before they happen. This means that all preparation for catastrophe is precautionary. 911 cannot be called on a polar vortex, or on an icy road. It can only be called once individuals become victims to these circumstances.

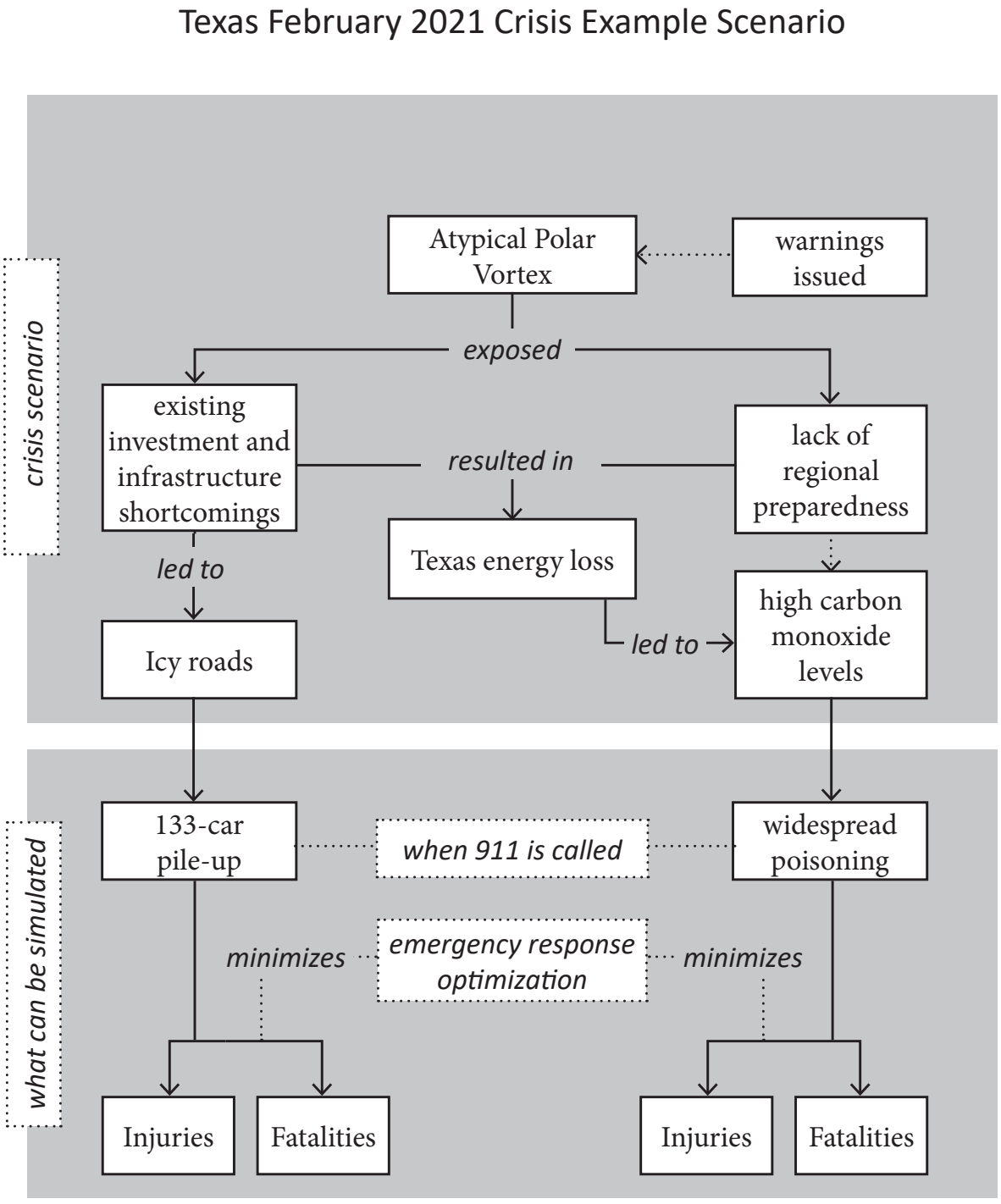

Figure A.1: Emergency Preparedness Failure Examples from Texas: February 2021

B — Contributions List

B.1 Personal Contributions

My personal contributions to the Intelligent Networks Lab fall under two main categories: the transition from BrainGrid to Graphitti, and the development of the NG-911 test-bed. Since I distributed this project over a six-quarter timeline, I held a project management and coordination role throughout the project and was unbounded by any particular area of focus. I was able to unify the team over a broad period of time, smooth transitions between project points, and provide context across different areas of the project. By gaining a deep understanding of neuroscience components, architectural components, and needs of the NG-911 project, all design decisions were made with an inclusive consideration of softeare requirements.

Each contribution is scored to rank time and complexity on a scale of 1 to 5. Time (T) score of 1 = less than a week; 3 = about a quarter; 5 = several quarters. Complexity (C) score of 1 = repetitive and simple tasks; 3 = required some critical thinking; 5 = collaborative, complex critical thinking and decision making.

Graphitti Development

- Prior to Graphitti's existence, I cataloged BrainGrid's technical debt with Dr. Stiber to understand what design decisions will maximize future maintainability of the simulator. [T=3, C=2]
- I worked with Dr. Stiber and Chris O'Keefe on developing a framework for new design pattern implementation in Graphitti. We researched various design choices and

weighed their cost and benefit prior to any implementation. One of the most important factors in these considerations was impact on memory and run-time. [T=3, C=3]

- I abstracted every variable, class, and method that was relevant to general graph-based networks. All neuro-specific code was then stored in neuroscience sub-systems. [T=5, C=2]

- I worked with Chris O'Keefe and Vivek Gandhi to eliminate some top-level interface classes like `IModel`, `IAllEdges`, and `IAllVertices`. We condensed the abstract classes in order to reduce the number of levels at the top of each subsystem. This included the elimination and refactoring of `SimInfo` and `IModel` and the inclusion of a new paradigm for accessors and mutators in the `Simulator` class. [T=5, C=4]

- I combined the `SimulationInfo` and `Simulator` classes to facilitate the future singleton class paradigm that was implemented by Chris O'Keefe. This class was shared with every class than needed access to parameters. [T=1, C=4]

- Prior to the completion of the Doxygen revival by Kyle Dukart, I worked on rewriting and condensing much of the documentation for Graphitti and began standardizing comments. I also developed a document that outlined the Doxygen commenting standards for the lab to use as an internal resource. [T=3, C=2]

- Although Workbench adaptation was not included in this project, section 6.3 discusses the potential for adapting it to Graphitti. After recording issues identified from running it, I shared my findings with current lab members who have addressed this part of the project. [T=2, C=2]

NG-911 Project

- Before building the NG-911 test-bed, I, along with Dr. Stiber and Vivek Gandhi, worked with representatives from the state of Washington, the Seattle Fire Department,

and the city of Seattle on answering the question, "What are we going to model?" We needed to gain an understanding of the necessary modelling capabilities that we needed to build into the simulator for the development of a practical model. I then authored the scenario development report for a document that was submitted for the 6-month research grant update. Following this, we translated the physical scenario into a graph-based model. [T=4, C=4]

- In collaboration with Vivek Gandhi, I developed the NG-911 minimum viable product. This included creating the necessary subsystems unique to the NG-911 model and developing a small test case that would demonstrate the behavior of the graph. [T=4, C=5]

- Also in collaboration with Vivek Gandhi, I analyzed the results of the minimum viable product. These results are preliminary and do not reflect a real-world scenario, but do lay the groundwork for a full-scale simulation. [T=2, C=3]

B.2 Contributions from Other Researchers

This project would not be possible without these primary collaborators. Rebuilding and repurposing a project of this scale required the hard work and commitment of the ten individuals recognized below.

- Dr. Michael Stiber: Intelligent Networks Lab (formerly Biocomputing Lab) founder and Principal Investigator to NG-911 Grant

- Emily Hsu: serialization and deserialization

- Lizzy Presland: parameter manager

- Chris O'Keefe: chain of responsibility, singleton design pattern implementation, operation manager, CMake, Google Tests

- Vivek Gandhi: NG-911 product development
- Kyle Dukart: Doxygen implementation
- Dr. Barbara Endicott-Popovski: Co-Investigator for NG-911 Grant
- M. Scott Sotebeer: resource liaison for NG-911 Grant
- Dr. William Erdly: committee member and early-stage research counsel
- Dr. Afra Mashhadi: committee member

References

[1] F. Kawasaki and M. Stiber, "A simple model of cortical culture growth: burst property dependence on network composition and activity," *Biological Cybernetics*, vol. 108, pp. 423–443, 2014. DOI: 10.1007/s00422-014-0611-9.

[2] M. Stiber and Endicott-Popovsky, "NCAE-C Research Grant Proposal: An Intelligent Testbed for Critical Infrastructure," Tech. Rep., 2020.

[3] R. R. Ulmer, *Effective Crisis Communication: Moving from Crisis to Opportunity*. Thousand Oaks: SAGE Publications, 2007, ISBN: 1-4129-1418-3.

[4] P. Bak, *How Nature Works: The Science of Self-organized Criticality*. Springer, 1996.

[5] FCC, *FCC Releases TFOPA Final Report*, 2016.

[6] L. Schruben and E. Yücesan, "Modeling paradigms for discrete event simulation," *Operations Research Letters*, vol. 13, pp. 265–275, 1993. DOI: 10.1016/0167-6377(93)90049-M.

[7] W. Aiello, F. Chung, and L. Lu, "A Random Graph Model for Massive Graphs," in *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, Association for Computing Machinery, 2000, pp. 171–180.

[8] T. Sellnow, M. Seeger, and R. Ulmer, "Chaos Theory, Informational Needs, and Natural Disasters," *Journal of Applied Communication Research*, vol. 30, pp. 269–292, 2002. DOI: 10.1080/00909880216599.

[9] D. Marković and C. Gros, "Power laws and self-organized criticality in theory and nature," *Physics Reports*, vol. 536, no. 2, pp. 41–74, Mar. 2014. DOI: 10.1016/j.physrep.2013.11.002.

[10] L. Schruben, "Simulation modeling with event graphs," *Communications of the ACM*, vol. 26, pp. 957–963, 1983. DOI: 10.1145/182.358460.

[11] J. Y. Lee and M. Stiber, "Development of spatiotemporal activity patterns in cultures of cortical neurons," unpublished.

[12] C. Marfin, J. Jimenez, N. Keomoungkhoun, C. Scudder, and T. Steele, *At least 6 dead in 133-car pileup in fort worth after freezing rain coats roads*, 2021.

[13] G. Wu, "More than 500 cases of carbon monoxide poisoning have now been reported in Houston," *Houston Chronicle*, 2021.

[14] The National 911 Program, "Next generation 911 (ng911) standards identification and review," Tech. Rep., Aug. 2020.

[15] Y. Mirsky and M. Guri, "DDoS Attacks on 9-1-1 Emergency Services," *IEEE Transactions on Dependable and Secure Computing*, 2020. DOI: 10.1109/TDSC.2019.2963856.

[16] K. Schmucker, "A Taxonomy of Simulation Software: A work in progress," *Learning Technology Review*, 1999.

[17] P. A. Fishwick, "A taxonomy for simulation modeling based on programming language principles," *IIE Transactions*, vol. 30, 1998. DOI: 10.1080/07408179808966527.

[18] J. Jalving, Y. Cao, and V. M. Zavala, "Graph-based modeling and simulation of complex systems," *Computers & Chemical Engineering*, vol. 125, pp. 134–154, 2019, ISSN: 0098-1354. DOI: 10.1016/j.compchemeng.2019.03.009.

[19] M. Stiber, F. Kawasaki, D. B. Davis, H. U. Asuncion, J. Y.-H. Lee, and D. Boyer, "Braingrid+workbench: High-performance/high-quality neural simulation," in *2017 International Joint Conference on Neural Networks (IJCNN)*, IEEE, May 2017, pp. 2469–2476. DOI: 10.1109/IJCNN.2017.7966156.

[20] *Neuron— empirically-based simulations of neurons and networks of neurons*. [Online]. Available: https://www.neuron.yale.edu/neuron/.

[21] *The GENESIS Simulator*. [Online]. Available: http://www.genesis-sim.org/.

[22] A. R. Mikler, J. S. K. Wong, and V. Honavar, "An object oriented approach to simulating large communication networks," *Journal of Systems and Software*, vol. 40, pp. 151–164, 1998, ISSN: 0164-1212. DOI: 10.1016/S0164-1212(97)00007-1.

[23] Y.-H. E. Hsu, "Extending a neural simulator to combine growth and spike-timing-dependent plasticity,"

[24] *UWB-Biocomputing/BrainGrid*, https://github.com/UWB-Biocomputing/BrainGrid, Dec. 2020.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," in *ECOOP '93*, Springer-Verlag, 1993, pp. 406–431.

[26] *UWB-Biocomputing/Graphitti*, https://github.com/UWB-Biocomputing/Graphitti, 2021.

[27] L. Chung, B. Nixon, and E. Yu, "Using non-functional requirements to systematically support change," in *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, 1995.

[28] *Cereal*, https://uscilab.github.io/cereal, 2017.

[29] Y. Berquin, *TinyXPath*, http://tinyxpath.sourceforge.net, 2013.

[30] *log4cplus*, https://github.com/log4cplus/log4cplus, 2021.

[31] *Mersenne twister*, https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine.

[32] *ParamContainer*, https://www.codeproject.com/Articles/8089/ParamContainer-easy-to-use-command-line-parameter, 2005.

[33] *Google test*, https://google.github.io/googletest/.

[34] N. Developer, *Cuda toolkit 11.2*, https://developer.nvidia.com/cuda-11.2.0-download-archive, 2020.

[35] *Doxygen 1.9.1*, https://www.doxygen.nl/index.html, 2021.

[36] *Cmake 3.19.6*, https://cmake.org/cmake/help/git-stage/release/3.19.html, 2021.

[37] J. J. Clague, "Evidence for large earthquakes at the cascadia subduction zone," *Reviews of Geophysics*, vol. 35, pp. 439–460, 1997. DOI: 10.1029/97RG00222.

[38] N. Subramanian and L. Chung, "Relationship between the whole of software architecture and its parts: An nfr perspective," in *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*, 2005.

[39] S. Singh, "Independent Study Report: Application of Random Graphs to analyze brain connectome simulated by BrainGrid network simulator," 2021.

[40] M. Guri, Y. Mirsky, and Y. Elovici, "9-1-1 DDoS: Attacks, Analysis and Mitigation," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, Apr. 2017, pp. 218–232. DOI: 10.1109/EuroSP.2017.23.

[41] T. Huang, H. Yang, H. Zhang, X. Cong, and G. Pan, "Diverse self-organized patterns and complex pattern transitions in a discrete ratio-dependent predator–prey system," *Applied Mathematics and Computation*, vol. 326, pp. 141–158, Jun. 2018. DOI: 10.1016/j.amc.2018.01.012.

[42] *Software rot: Definition, causes, threats, mitigation methods*, https://heimdalsecurity.com/blog/software-rot, 2020.

[43] *Sourcemaking*, https://sourcemaking.com/.

[44] I. Sommerville, *Software engineering*. Addison-Wesley, 2011.

Acronyms

BCL Biocomputing Lab. iii

CPU Central Processing Unit. 10, 15, 18, 42, 49, 53

DDoS Distributed Denial of Service. 10, 11, 46, 50

DES discreet event simulation. 11

GPU Graphics Processing Unit. 15, 18, 30, 42, 49, 50, 53

INL Intelligent Networks Lab. 7, 47, 49

NCAE-C National Centers of Academic Excellence in Cybersecurity. iii

NG-911 Next-Generation 911. i, ii, 1, 2, 4–7, 9, 11, 19, 32–34, 36–42, 44–51, 53, 58–61

PSAP Public Service Answering Point. 11, 37–40, 42, 43, 50

SOC self-organized criticality. 4, 7, 12, 42, 47