Modeling and design of neural network architectures for neural artificial-biological hybridization based on synchronous approach

Marino Rasamuel

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Modélisation et conception par approche synchrone d'architectures neuronales hybrides biologique-artificiel

Marino RASAMUEL

Laboratoire d'Électronique, Antennes et Télécommunications (LEAT)

Présentée en vue de l'obtention du grade de docteur en Électronique **d'**Université Côte d'Azur

Dirigée par : Benoît MIRAMOND, PR, LEAT, Université Côte d'Azur
Co-encadrée par : Daniel GAFFÉ, MCF, LEAT, Université Côte d'Azur
Soutenue le : 18-07-2023

Devant le jury, composé de :
Timothée LEVI, PR, Université de Bordeaux
Pierre BOULET, PR, Université de Lille
Bertrand GRANADO, PR, Sorbonne Université
Dumitru POTOP-BUTUCARU, CR, INRIA
Gilles BERNOT, PR, Université Côte d'Azur

Modélisation et conception par approche synchrone d'architectures neuronales hybrides biologique-artificiel

---

Modeling and design of neural network architectures for neural artificial-biological hybridization based on synchronous approach

Marino RASAMUEL

⋈

Jury :

Président du jury
Timothée Levi, PR, Université de Bordeaux

Rapporteurs
Pierre Boulet, PR, Université de Lille
Bertrand Granado, PR, Sorbonne Université

Examinateur
Dumitru Potop-Butucaru, CR, INRIA

Directeur de thèse
Benoît Miramond, PR, LEAT, Université Côte d'Azur

Co-encadrant de thèse
Daniel Gaffé, MCF, LEAT, Université Côte d'Azur

Membre invité
Gilles Bernot, PR, Université Côte d'Azur

Université Côte d'Azur

Marino RASAMUEL

Modélisation et conception par approche synchrone d'architectures neuronales hybrides biologique-artificiel

xiii+199 p.

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : output.tex – 7/9/2023 – 13:41

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

Je dédicace ce travail à ma mère, Jeanne, car c'est pour elle que la bataille en vaut la peine.

Modélisation et conception par approche synchrone d'architectures neuronales hybrides biologique-artificiel

Résumé

Alors que les Réseaux de Neurones Artificiels (RNA) continuent de progresser dans des domaines tels que l'apprentissage automatique, la robotique, les véhicules autonomes et le diagnostic de santé, un nouveau cadre d'application gagne du terrain à la fois dans les secteurs académique et industriel : la Neurobiohybridation. Ce domaine cherche à établir des connexions entre des neurones artificiels et biologiques dans le but de comprendre et potentiellement de réparer ou remplacer des fonctions cérébrales perdues suite à des maladies ou des accidents. Dans cette perspective, le développement de réseaux de neurones artificiels inspirés biologiquement, souvent appelés Réseaux de Neurones à Spikes (SNNs), est essentiel pour améliorer la compatibilité entre les systèmes neuronaux artificiels et biologiques. Cette thèse s'inscrit dans ce contexte en utilisant l'approche synchrone pour modéliser, mettre en œuvre et simuler des SNNs bio-inspirés et biomimétiques. En utilisant des vérificateurs de modèles, qui permettent de prouver ou d'extraire des propriétés des systèmes de manière formelle, notre objectif est d'acquérir une compréhension plus complète des comportements biologiques dans le future. Pour la première fois dans ce contexte, nous utilisons le langage Light Esterel pour atteindre nos objectifs. Nous démontrons son potentiel dans la mise en oeuvre de modèles neuronaux, initiant une bibliothèque de modèles pour explorer différents types de SNNs. Tout au long de cette thèse, nous avons développé un cadre complet basé sur Light Esterel pour modéliser, simuler et mettre en oeuvre divers modèles de SNNs. Pour aborder les expériences de neurobiohybridation, nous avons développé notre propre architecture matérielle, SynchNN, capable d'exécuter en temps réel des SNNs récurrents en utilisant notre bibliothèque de modèles. L'environnement de modélisation que nous avons développé est complété par un framework de simulation, en cours de développement, visant à réaliser des expériences de neurobiohybridation à l'avenir.

Mots-clés : Réseaux de neurones impulsionnels, Approche Synchrone, Langage Light Esterel, FPGA, neurobiohybridation

Modeling and design of neural network architectures for neural artificial-biological hybridization based on synchronous approach

Abstract

As Artificial Neural Networks (ANNs) continue to advance in fields like machine learning, robotics, autonomous vehicles, and healthcare diagnostics, an application domain is gaining attraction in both academic and industrial sectors : Neurobiohybridization. This domain seeks to establish connections between artificial and biological neurons with the goal of understanding and potentially repairing or replacing lost brain functions due to disease or accidents. In pursuit of this, the development of biologically inspired artificial neural networks, often referred to as Spiking Neural Networks (SNNs), is essential to enhance compatibility between artificial and biological neural systems. This thesis fits into this context by using the synchronous approach to model, implement, and simulate bio-inspired and biomimetic SNNs. Leveraging model checkers, that allow to prove or extract properties in systems in formal manner, our aim is to gain a more comprehensive understanding of biological behaviors in the future. For the first time in this context, we utilize the Light Esterel language to achieve our objectives. We demonstrate its potential in implementing neural models, initiating a library of models for exploring different types of SNNs. Throughout this thesis, we developed an entire framework based on *Light Esterel* in order to model, simulate and implement various SNN models. To address neurobiohybridization experiments, we developped our own hardware architecture, SynchNN, capable of executing recurrent SNNs in real-time using our library of models. This framework we developed is completed with an on-going simulation framework aiming to conduct neurobiohybrid experiments in the future.

Keywords: Spiking Neural Networks, Synchronous Approach, Light Esterel language, FPGA, neurobiohybridization

Remerciements

Je tiens à exprimer tout d'abord ma sincère gratitude envers mon directeur de thèse, Benoît Miramond, pour l'opportunité exceptionnelle qu'il m'a offerte de m'engager dans la recherche. Son expertise, sa passion pour son domaine et ses conseils scientifiques ont été une source constante d'inspiration tout au long de ce voyage académique. Cependant, ce qui distingue véritablement Benoît va au-delà du rôle traditionnel d'un directeur de thèse. Il a été là dans les moments les plus difficiles, continuant à me soutenir malgré tout, avec une compréhension profonde et une amitié sincère. Il est un modèle pour sa force et sa générosité. Pour tout cela, je lui en suis infiniment reconnaissant.

Je souhaite également adresser mes remerciements les plus chaleureux à mon encadrant de thèse, Daniel Gaffé. En master, il a cru en mes capacités et m'a poussé à donner le meilleur de moi-même. Son rôle dans mon évolution académique a été fondamental. Les discussions parfois animées que nous avons eues ont enrichi cette expérience académique de manière significative, et je suis reconnaissant d'avoir pu les partager avec lui. Merci, Daniel, pour le temps que tu m'as consacré, ton soutien inébranlable et pour m'avoir poussé jusqu'à la fin de ce voyage.

Ensuite, je tiens à exprimer ma sincère gratitude envers tous les membres du Jury pour avoir consacré leur précieux temps à l'évaluation de mes travaux. Je voudrais adresser mes remerciements les plus sincères à Pierre Boulet et Bertrand Granado d'avoir accepté le rôle de rapporteurs pour ma thèse. Je tiens également à exprimer un remerciement tout particulier à Timothée Levi pour sa bienveillance et son accueil chaleureux au Japon. Je suis honoré de l'avoir eu en tant que président de mon Jury, ce qui a couronné magnifiquement la conclusion de mon travail.

Évidemment, ces années passées au LEAT m'ont permis de rencontrer et de tisser des liens forts avec plusieurs personnes exceptionnelles que je tiens à remercier du fond du cœur. Leur amitié et leur soutien ont rendu ces années académiques particulièrement agréables. Je tiens à exprimer ma reconnaissance spéciale à Roland Kromes, un ami extraordinaire, un précieux collègue de bureau, et une personne formidable avec qui chaque instant passé au bureau était empreint d'une dose supplémentaire d'amusement. Je m'excuse sincèrement de ne pas pouvoir citer tout le monde ici, mais je vous remercie d'avoir partagé cette aventure avec moi et d'avoir contribué à en faire une expérience mémorable.

Enfin, je tiens à exprimer ma profonde gratitude envers ma famille, qu'elle soit près ou loin, pour leur soutien inébranlable qui a été une force constante dans ma vie. Ma sœur, Mahefa, a apporté un soutien précieux, parfois différent de ce qui est standard, mais toujours inestimable. Mon frère, Matio, pour sa précieuse présence. Mes nièces, Mélina et Laurina, ont été des rayons de soleil, apportant des rires et de la joie même lors des moments les plus difficiles. Spécialement à ma mère, Jeanne, mes mots ne peuvent que gratter la surface de son sacrifice extraordinaire. Aujourd'hui, alors qu'elle lutte courageusement contre la maladie d'Alzheimer, son amour inconditionnel continue de briller comme une lumière dans nos vies. Je lui suis infiniment reconnaissant, et c'est désormais à notre tour de prendre soin d'elle, avec la même tendresse et le même dévouement qu'elle a toujours eu à notre égard.

Contents

CHAPTER 1

Introduction

1.1 Project context

With digital systems becoming an integral part of our daily lives, providing services that closely align with human needs, such as sports or medical monitoring through connected devices, the future holds the potential for digital systems to be regarded as "digital prostheses". These prostheses would not only assist in specific tasks but also enhance human capabilities, augmenting cognition and sensory functions. This raises the questions of their learning, appropriation and integration into the everyday environments of those who use them. The partners of the ARTEFACT project proposed that this topic, often imposed by American giants or simplified by the media, has to be finally addressed and reappropriated by the University in an interdisciplinary manner. The ARTEFACT project (Université Côte d'Azur (UCA), 2018-2021) is a collaborative effort involving researchers and professors from various disciplines such as electronics, sociology, psychology, computer science and neuroscience. The project aims to encompass the entire process, from the design of new digital system devices to the support of their implementation and user acceptance. The ARTEFACT project addresses three specific work packages (WPs):

- WP1 - *Spatial cognition*: This WP focuses on the development of a triangulation-based localization system for monitoring and surveillance of elderly people.
- WP2 - *Connected glasses and sensory augmentation*: The objective of this WP is to establish a system capable of delivering visual information from connected glasses to impaired people through tactile feedbacks.
- WP3 - *Neural hybridization and extended cognition*: This WP, which is directly relevant to this thesis, aims to develop neural architectures using the synchronous approach to enable communication between artificial and biological neurons for the design of future neuroprostheses.

This thesis, titled "**Modeling and Design of Neural Network Architectures for neural using the Synchronous Approach**", specifically addresses WP3, the interconnection between biological and artificial neurons. The collaboration involves several laboratories: the Laboratory of Electronics, Antennas and Telecommunications (LEAT) for the study of the synchronous approach and the design and development of neuromorphic systems; the Institute of Industrial Science (IIS) at the University of Tokyo, Japan, for providing the experimental setup for communication between biology and artificial systems; the NeuroMod Institute for biological discussions, and the Laboratory of Computer Science, Signals and Systems (I3S) for computer science expertise.

1.2 Thesis context

The brain, the most complex organ of the human body, performs an astonishing array of functions crucial to our survival and well-being. It represents one of the main components of the nervous system. Within this system, neurons are the basic structural and functional units. These neurons form a vast network, transmitting nerve impulses - a series of electric signals also known as action potentials or spikes - amongst themselves and to different body parts. By doing so, neurons are responsible for coordinating all the body's activities, from basic functions like heart rate and breathing, to complex processes like thoughts and emotions.

Neuroscience is a scientific discipline dedicated to study the nervous system in its entirety. It seeks to understand how the nervous system is structured, how it operates, how it develops, how it changes over time, and how it can be repaired. This discipline covers a wide range of studies, extending from molecular and cellular exploration of the nervous system to the investigation of complex cognitive phenomena such as memory, learning, emotions, perception and decision-making.

How do neurons enable the emergence of complex cerebral functions only through the exchange of electrical impulses?

The brain contains nearly 100 billion neurons (Herculano-Houzel, 2009) that communicate in a multitude of interconnected networks and sub-networks in an extremely intricate way. Despite technological advancements and intensive research, the answer to this question still holds a large share of unknowns. However, there are theories that propose models to explain these complex interactions. Among these theories, there are for example the "neuronal assemblies" and "polychronous networks" theories. The *neuronal assemblies* theory suggests that groups of neurons connected together can generate a sustained pattern of electrical activity. These assemblies are thought to represent specific cognitive or perceptual entities, thus offering a possible way to link neural and cognitive processes (Hebb, 1949b; Buzsáki, 2010). The *polychronous networks* theory, on the other hand, proposes that precise timing of action potentials within the neural network is crucial. It suggests that information is stored in the timing of spikes and the network's time-dependent structure, rather than the rate of neuron firing. This idea allows for an immense capacity for information storage and processing within the brain's neural networks (Izhikevich, 2006).

Studying, testing, or verifying these theories using observations from real neurons can quickly become complex and requires advanced technological means. Precisely and simultaneously recording activities from multiple neurons can be challenging, and the dynamic nature of biological networks activities adds further complexity. To overcome these challenges, it is advantageous to turn to *in silico* approaches, *i.e.* computer-based simulations, modeling or analysis.

Computational neuronscience and neuromorphic engineering

In silico approaches are made possible through the computational neuroscience discipline, a specialized interdisciplinary branch within the neuroscience field. It aims to develop and use mathematical models, algorithms, computer simulations, and theories to comprehend and predict the functioning of the nervous system. It seeks to elucidate how nervous systems process information, generate behavior, and how they can be modeled. These models focus on various aspects of the nervous system, from biochemical processes within individual neurons to the behavior of large neuronal networks. They provide controlled and highly detailed platforms for investigating neural activity, exploring network dynamics, and manipulating various parameters to gain insights into complex neural processes, impossible to perform *in vivo*. There are numerous models of neurons with diverse characteristics. These models are distinguished based on their level of abstraction and modeling objectives. In this thesis, we will focus specifically on bio-inspired or biomimetic (bio-plausible) models and other models of biological mechanisms. These models are governed by a system of differential equations to describe the mechanisms underlying the bioelectrical or functional activity of neurons.

In parallel, a different but related *in silico* branch, the neuromorphic engineering, derives inspiration from the structure, functioning and processes of the human brain to design electronic systems, algorithms and hardware architectures. The neuromorphic engineering aims to create systems capable of mimicking brain-like learning to tackle tasks requiring human intelligence, systems capable of real-time processing of vast amounts of information with energy efficiency compared to traditional computers, or building adaptable systems capable of reorganization.

Toward the development of neuroprotheses

By developing models and architectures based on Artificial Neural Network (ANN) that replicate the functions and structures of the nervous system, while also surpassing the speed of biological neurons (Mahowald & Douglas, 1991), there is a rapidly growing interest in the field of biomedical domain with the development of neuroprostheses. Neuroprostheses are devices that interact directly with the nervous system to restore or replace lost brain functions due to diseases or accidents. Devices that communicate directly with the nervous system already exist, such as cochlear implants, which are inserted into the ear of a deaf or hard-of-hearing person to enable sound perception; motor prostheses, which use brain or muscle signals to control movement of a prosthesis; retinal implants, which restore some form of vision in blind individuals; etc. However, the mentioned implants are not specifically based on ANN. Therefore, ANN-based devices are believed to have the potential to interact with biological neurons, paving the way for a new generation of neuroprostheses.

The integration of biological neurons with artificial ones is referred to as *biohybridization* or, more specifically, *neurobiohybridization* (Vassanelli & Mahmud, 2016). Neurobiohybridization is a challenge being pursued by numerous research teams wordwide. This exploration began in 2001 with the pioneering work of (Jung, Brauer, & Abbas, 2001) and has continued to the present day (Chiappalone et al., 2022). In parallel with the goal of replacing or repairing lost functions, neurobiohybridization also offers the potential to gain a better understanding of biological processes and address questions such as how neurons encode information. This interdisciplinary approach provides an opportunity to bridge the gap between neuroscience and artificial intelligence, leading to advancements in both fields.

Synchronous approach

In our methodology, we adopt the synchronous approach. This allows to leverage the numerous options it offers in the context of modeling neuromorphic systems and understanding the biology. Specifically, we have chosen to work with *Light Esterel*, a synchronous language developed within the INRIA institute (Annie Ressouche) and the LEAT laboratory (Daniel Gaffé). The language and its compilation environment provide advantageaous features we wanted to take advantage of, as illustrated in figure 1.1. The compilation environment includes high-level codes specifications, efficient codes generation and the use of formal methods to check specific behavioral properties.

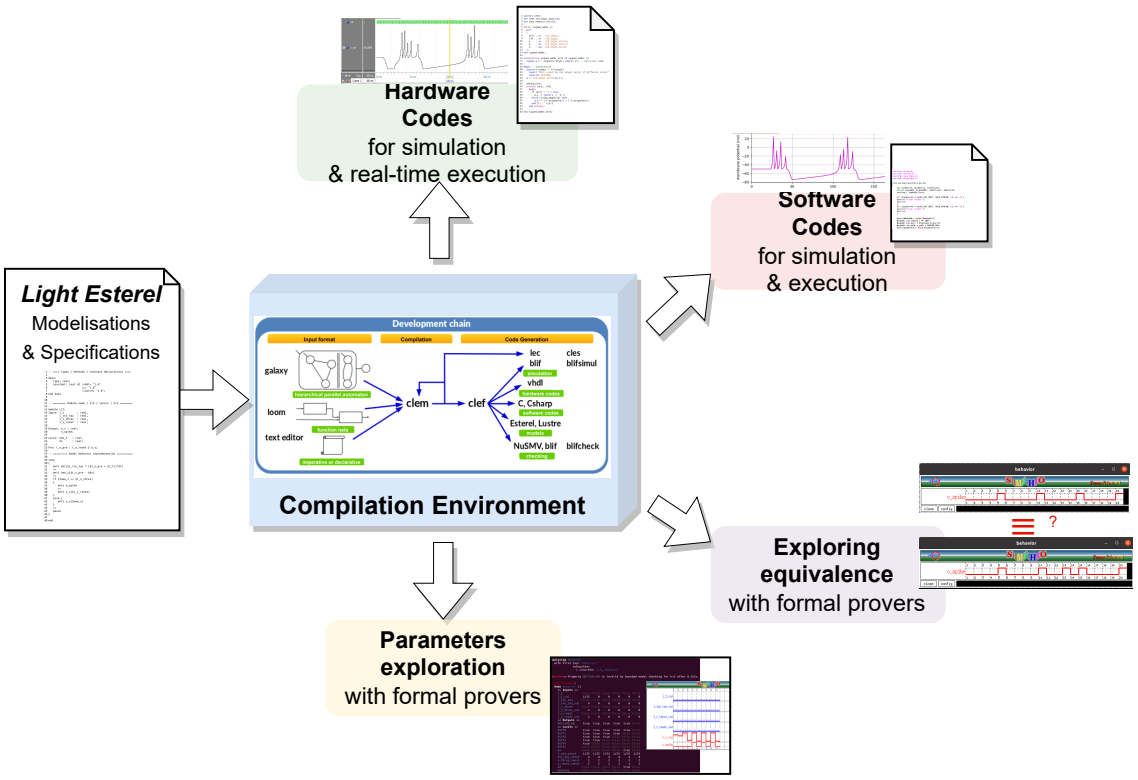


Figure 1.1: *Light Esterel compilation environment and options*

One important feature of the synchronous approach is the assurance it provides that the specifications are faithfully preserved throughout the entire compilation process, up to the code generation phase. This is achieved through the underlying paradigm of the synchronous approach. This allows the developpers to focus on the design and expression of their high-level specifications, to generate multiple target codes seamlessly, without low-level coding burden.

Finally, due to the rigorous and well-defined nature of synchronous languages, it becomes feasible to apply formal methods to verify and reason about the behavior and properties of the specified systems.

So, this thesis is positioned at the intersection of these fields, with the aim to exploit the synchronous approach for modeling and studying neuronal networks, for the development of biologically plausible neuromorphic models and framework for neuroprostheses future applications.

1.3 Problematics

The use of formal methods in verifying software correctness and ensuring that the system respect the given specifications is well-established, particularly in critical software systems. These methods employ mathematical logic and provide rigorous reasoning techniques for system analysis. While formal methods have demonstrated success in various domains, such as the study of dynamic biological systems like genetic and metabolic networks (mentionned in (Guinaudeau, 2019)), their application in the field of neuroscience is limited and relatively unexplored.

This work marks the first application of the *Light Esterel* language, our main tool, in the study of ANNs. Specifically, we will focus on Spiking Neural Networks (SNNs), a category of ANNs that closely mimic the behavior of biological neurons by using spike-based communication.

All these points raise questions regarding, firstly, the use of *Light Esterel* for developping biologically plausible SNN models; secondly, the exploration of the potential of formal methods to gain a comprehensive understanding of the dynamics of BNNs; and thirdly, the types of neurobiohybrid experiments that can be conducted in this context.

1.4 Objectives

The objectives of this thesis concern the application of the *Light Esterel* language in the modeling of bio-inspired and biomimetic neural architectures, with a specific focus on neurobiohybridization.

The first objective is to demonstrate the capability of *Light Esterel* to model neural systems that closely resemble the behavior of their biological counterparts. We aim to develop accurate and biologically plausible models and validate them through experimental simulations and implementations.

The second objective involves exploring the feasibility of employing proof techniques on our models, and to conduct two distinct experiments. The first experiment focuses on behavior equivalence, where we try to formally prove that our modeled neural systems exhibit behavior equivalent to biological observations. This would provide evidence of the fidelity of our models to biological reality. The second experiment involves parameters exploration, where we aim to utilize proof tools to identify sets of parameters that enable our neural models to exhibit specific desired behaviors, similar to those observed in biological systems.

The third objective of this thesis is to leverage the developed neural models and biological mechanisms in *Light Esterel*, to facilitate the generation of neural networks with ease in different codes. We aim to develop a framework that simplifies the process of creating neural networks based on our models and their corresponding implementations. Especially, we aim to test the generated neural networks for neurobiohybrid experiments.

1.5 Contributions

The contributions of this thesis are as follows:

1. The utilization of the *Light Esterel* language for the first time in the domain of SNNs. This required modifications to the initial compilation environment of the language. Specifically, an input format was developed as front end of the *Light Esterel* compilation process for specifying neural networks. Additional instructions and operators were added to the language to facilitate the modeling of neural networks. Furthermore, two tools, "*create_gln*" and "*genlenet*" were developed. These tools enable the generation of high-level specifications for random or specific neural network's configurations, and the rewritting of the high-level specifications, respectively. The compilation tool "genlenet" allows to choose either to generate codes or to produce the configuration files for the developed hardware architecture.

2. The development of a specific hardware architecture called *SynchNN*, designed to execute recurrent SNNs in real-time with the neural models and biological mechanisms developed

in *Light Esterel*. This architecture provides a solution to the scale limit encountered with the *Light Esterel* compilation environment.

3. The use and study of the chosen symbolic proof tool : *KIND2*, within the context of our application.

4. The ongoing development of a simulator for biohybrid or neurobiohybrid experiments, aiming to complement the *SynchNN* architecture. This simulator serves as a tool for conducting simulations, exploring different scenarios of biohybrid experiments, refining the developed models and optimizing the parameters to ensure compatibility with the *SynchNN* platform.

These contributions collectively enhance the field of SNNs and neurobiohybridization by introducing the *Light Esterel* language, developing a specialized hardware architecture, and providing tools for modeling, simulation and analysis. They also contribute to the understanding of the applicability and limitations of symbolic proof techniques in the context of neural modeling.

1.6 Manuscript reading guide

The rest of this manuscript consists of six chapters.

Chapter 2 provides an overview of the basics in neuroscience, including the anatomy of a neuron, the biological mechanisms involved in the generation, transmission, and reception of an action potential, the principles and various models considered in this study, simulation methods for SNNs, and an introduction to the application of neurobiohybridization.

Chapter 3 presents the synchronous approach, briefly discussing different existing languages and providing a detailed description of the *Light Esterel* language chosen for this thesis. We also introduce model checkers in this chapter.

Chapter 4 discusses existing works that combine neuroscience and the synchronous approach. We explain how we described our models in *Light Esterel*, the validation process, and discuss the limitations encountered with the language that led to the development of the *SynchNN* architecture. Additionally, we present the results obtained from model checking experiments.

Chapter 5 provides a detailed description of the *SynchNN* hardware architecture developed, including our approach to configure the architecture, our validation method and performance analysis.

Chapter 6 presents the neurobiohybrid experiment conducted in Japan with the collaboration of the Institute of Industrial Science (IIS) at the University of Tokyo, Japan, and discusses the simulator developed for our neurobiohybrid application.

Finally, in Chapter 7, we present our conclusions and outline future perspectives of this thesis.

CHAPTER 2

Neural Networks

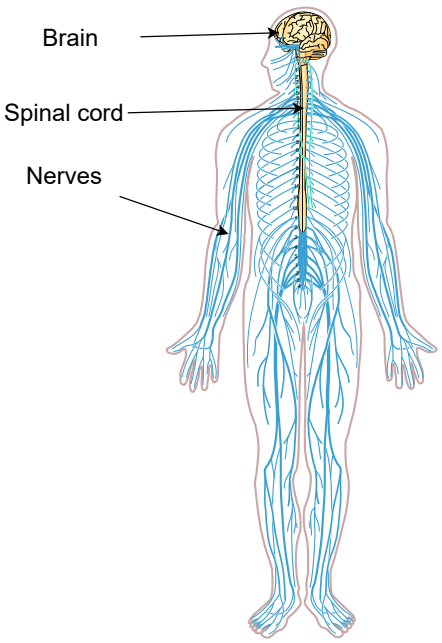
In this chapter, we explore the world of neurons and their interconnected networks, as well as the efforts to replicate these foundation of the brain's computing system. We start by looking at the structure of a neuron and how neurons communicate with each other using electrical signals known as action potentials or spikes. We then discuss different spiking neuron models that aim to mimic the behavior of real neurons.

Next, we talk about neural properties. We discuss AMPA and GABA dynamics, the idea of plasticity and axonal delay and the role of biological noise. Each of these topics gives us more information about the complex workings of neural networks, but also carries significant importance when it comes to the simulation and implementation of more bio plausible Spiking Neural Networks (SNNs).

Then, we discuss about the simulation of SNNs. We present 2 different approaches, hardware and software, and talk about their most-known framework. We discuss about their respective advantages and disadvantages, and the applications they address. Amongst the framework presented, we define the simulation framework we chose in this work's context.

Finally, we focus on the context of neurobiohybridization, the context within which this thesis is situated. We discuss the objectives, the technologies, the methodologies and challenges of this application.

The nervous system is a complex, highly organized network of specialized cells, that transmit and process information throughout the body. It is responsible for coordinating and controlling a wide range of functions, including sensation, movement, emotion, thought, etc. The nervous system can be broadly divided into two main components : the central nervous system (CNS) and the peripheral nervous system (PNS). The CNS consists of the brain and the spinal cord. It is the control center for the entire body, processing the sensory information, generating motor commands, and regulating higher cognitive functions such as learning, memory or decision-making. The PNS encompasses all the nerves outside the CNS. The nerves connect the brain and the spinal cord to the rest of the body, responsible of the transmission of the sensory/command information from/to the CNS.



2.1 Neuron

At the core of the functioning of the brain lies the neuron. A neuron is a unit cell that serves as the basic building block of the nervous system. It was discovered by *Santiago Ramón y Cajal*, a Spanish histologist and neuroscientist in the late 19th century (Llinás, 2003; De Carlos & Borrell, 2007). The main function of a neuron is to receive, process and transmit information called *nerve impulses*, and it has evolved to transmit and process this information rapidly and efficiently. The number of neurons was estimated to 86 billion neurons in the human brain (Azevedo et al., 2009). The connections and communications between such a vast number of cells are responsible of the multiple, various and complex brain functions.

2.1.1 Structure

The neuron is composed of 3 main parts : the soma, the dentrites and the axon, as illustrated in figure 2.1.

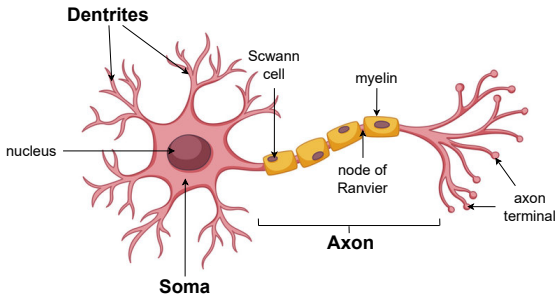


Figure 2.1: *Neuron's structure.*

Dentrites

The dentrites are the branching extensions of a neuron that receive input from other neurons and convey this information toward the cell body, or soma. Dentrites are characterized by their complex branching patterns, which increase their surface area and enable them to form numerous connections. The synapse is the junction between two neurons, where the axon terminal of the presynaptic neuron (the neuron sending the signal) comes into close proximity with the dentrite or cell body of the postsynaptic neuron (the neuron receiving the signal).

Soma

The soma, also known as the cell body or *perikaryon*, is the central part of a neuron that houses the nucleus and other essential cellular components (Golgi apparatus, mitochondria, ...). The nucleus contains the genetic material (DNA) that encodes for the synthesis of proteins and other molecules necessary for the neuron's structure, function and survival of the neuron. The main role of the soma is to integrate signals it receives from the dentrites, and in case the amount of signals reaches a certain threshold, it generates an action potential (or nerve impulse) that will be sent to other neurons.

Axon

The axon is a long and unique projection of a neuron that extends from the soma. Its function is to transmit electrical signals, known as action potentials, from the soma to other neurons or cells such as muscles. The axon facilitates rapid and precise transfer of information between neurons, enabling communication within the nervous system. Most axons in the human nervous system are wrapped in a fatty insulating layer called the myelin sheath (Baumann & Pham-Dinh, 2001). Myelin is produced by specialized glial cells such as Schwann cells in the peripheral nervous system. Its role is to enable faster conduction of action potentials along the axon by reducing electrical resistance and allowing the impulse to "jump" between gaps in the myelin called *nodes of Ranvier*. This is particularly important when the distance between two neurons is long (from microns to more than 1 meter), as it prevents the electric signal from decreasing or disappearing during the transfer.

Neurons share some common features such as the 3 main parts mentionned previously. However, they also exhibit a vast diversity in size, shape and function, as illustrated in figure 2.2. Neurons can be categorized into various types (Masland, 2004; Nelson, Sugino, & Hempel, 2006; Stuart, Spruston, & Häusser, 2016), such as sensory neurons, motor neurons and inter-neurons, each with specific roles in the nervous system. However, there are no established consensus rules to precisely categorize neurons (Markram et al., 2015).

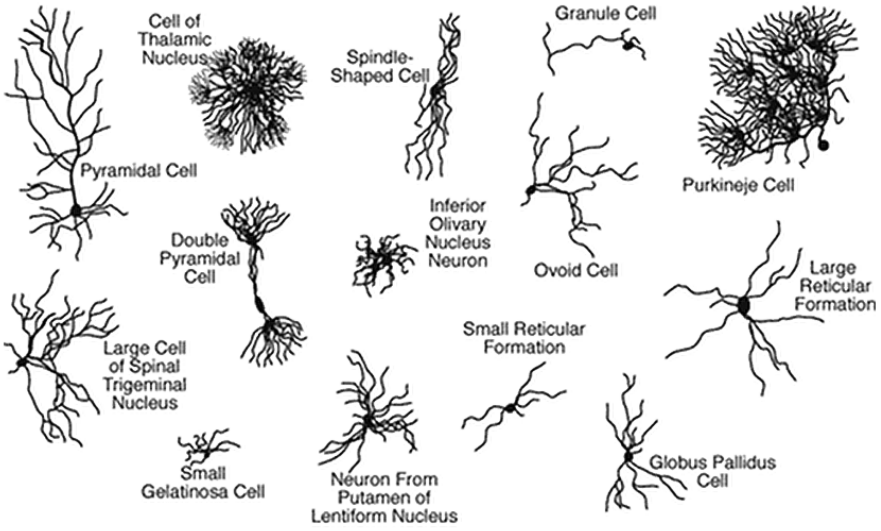


Figure 2.2: *Example of different types of neurons, taken from (Stufflebeam, 2008) based on drawings made by Santiago Ramón y Cajal. It shows the diversity of neurons in shape and size, in different brain areas.*

2.1.2 Action potential

The action potential is the information exchanged between neurons, also known as the *nerve impulse* or *spike*. It is a rapid, temporary change in the electrical potential across the membrane of a neuron, and propagates from the soma, through the axon, to other neurons. An action potential is generated and propagated through a series of well-coordinated movements across the neuronal membrane as shown in figure 2.3.

Membrane potential of a neuron

The membrane potential of a neuron refers to the difference in electrical charge between the inside and the outside of the cell membrane. A difference exists due to an unequal distribution of ions, particularly sodium (Na+) and potassium (K+) (and chloride (Cl-)) between the inside and outside of the membrane. The concentration of ions are regulated thanks to the selective permeability of the cell membrane and to the action of ion pumps. The selective permeability of the membrane to K+ ions means that it allows K+ ions to move more freely across the membrane, more precisely it allows the K+ ions to diffuse out of the cell, which leads to only negatively charged ions inside the cell.

Resting state: at rest, the neuron maintains a resting membrane potential, typically around -70 mV (millivolts) (step 1 in figure 2.3), with the inside of the neuron being more negatively charged than the outside. To compensate the K+ leaving out the cell, sodium-potassium pump (Na+/K+ pump) actively transports 3 Na+ ions out of the cell and 2 K+ ions into the cell, using energy from one ATP molecule (not illustrated in figure 2.3). This process allows to maintain a higher concentration of Na+ ions outside and a higher concentration of K+ ions inside, hence a negative resting membrane potential.

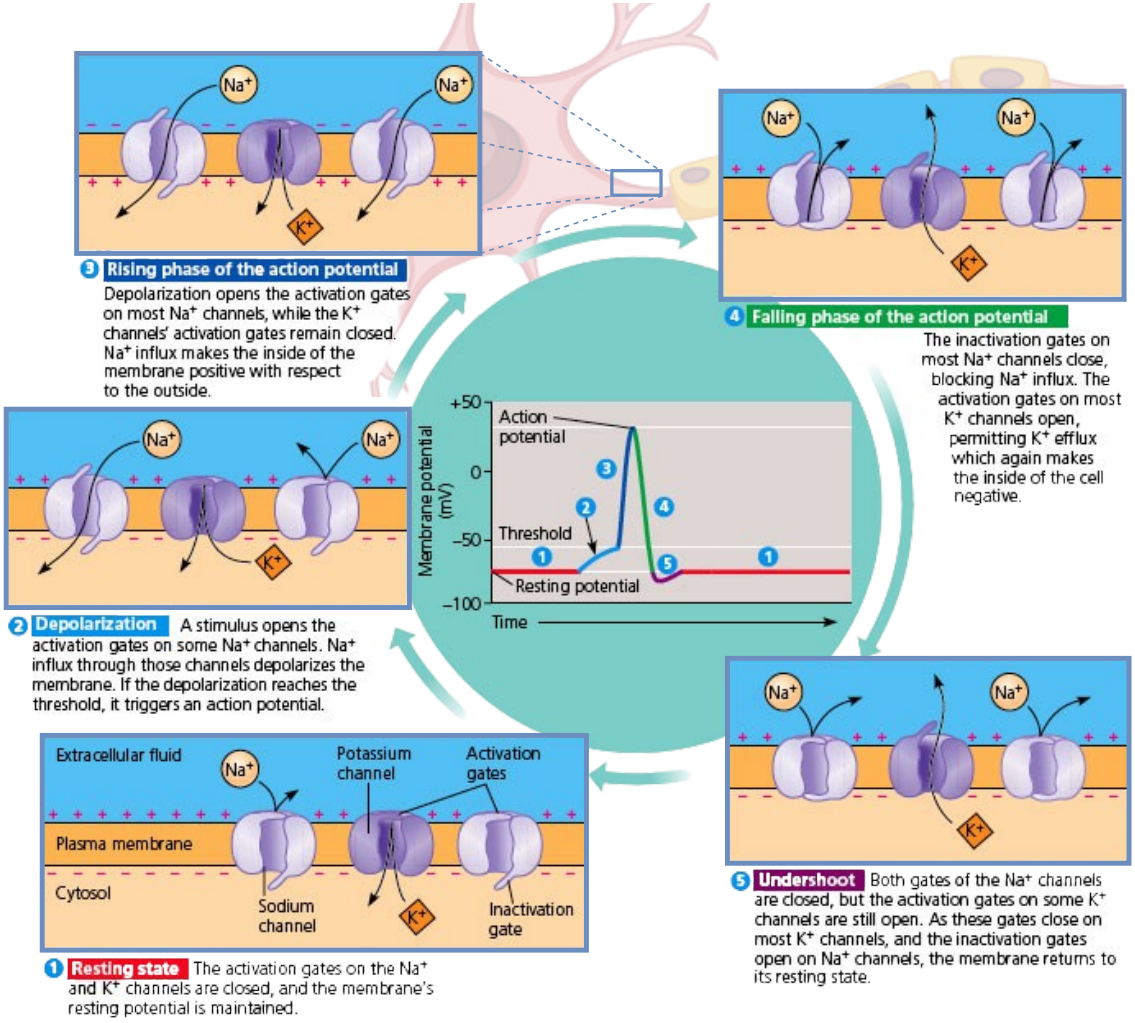


Figure 2.3: *Generation of an action potential. Modified figure from (dundeemedstudentnotes, 2012).*

Generation of an action potential

The capacity of a neuron to generate an action potential is due to a significant concentration of particular ions channels (or ions pumps for Na+ and K+) located at the axon's initial segment (where the axon originates from the soma), also known as the *axon hillock*. These channels are *voltage-dependent*, meaning they open when the membrane potential crosses a specific threshold or a depolarization threshold.

Depolarization: when the neuron receives excitatory stimulation or action potentials from others neurons, the membrane potential becomes less negative, moving toward the depolarization threshold, usually around -55 mV (step 2 in figure 2.3). Excitatory input provokes the opening of some voltage-gated sodium (Na+) channels.

Hyper depolarization: when the threshold is reached, more Na+ channels open, allowing an influx of Na+ ions into the cell. This rapid influx of positive ions causes the membrane potential to become positive, leading the membrane potential to reach its peak, typically around +30 to +40 mV (step 3 in figure 2.3).

Repolarization: as the membrane potential reaches its peak, the voltage-gated Na+ channels close, and the voltage-gate potassium (K+) channels open. This allows the K+ ions to flow out of the cell, returning the membrane potential back toward its negative resting state (step 4 in figure 2.3).
Hyperpolarization: during the repolarization, the K+ channels may remain open slightly longer, causing a brief period of hyperpolarization (step 5 in figure 2.3), where the membrane potential becomes more negative than the resting potential. The sodium-potassium pump (not illustrated on the figure) then restores the original ion concentrations and reestablishes the resting membrane potential.

Immediately after the action potential, it is impossible or extremely difficult to generate another action potential as the Na+ channels responsible for the depolarization of the membrane potential are still inactive and can not be reopened until the membrane potential returns close to its resting state : it is called the *absolute refractory period*. After the absolute refractory period, when the membrane potential is getting back (closer) to its resting state, it is possible to generate another action potential, but it requires a stronger stimulus to reach the depolarization threshold : it is called the *relative refractory period*. The refractory period ensures that action potentials propagate unidirectionally along the axon, from the cell body to the axon terminal, and prevents them from moving backward or continuously firing in a loop.

Propagation of an action potential

The propagation of an action potential refers to the travel of the action potential along the axon of a neuron, from the axon hillock to the axon terminal. The influx of Na+ ions during the depolarization phase creates a local current that flows passively along the axon. This local current depolarizes the adjacent membrane, opening voltage-gated Na+ channels in the neighboring region and allowing more Na+ ions to enter the cell. This depolarizes the adjacent membrane, generating a new action potential at that location (see figure 2.4). The hyperpolarization prevents the action potential to propagate backward or to generate a new action potential, by inactivating the previous Na+ channels. In conclusion, the propagation of an action potential is actually a repeating process of generation of an action potential along the axon.

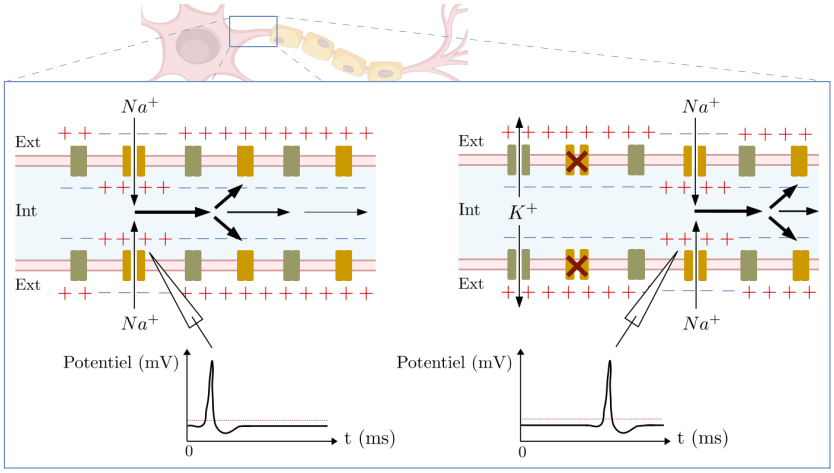


Figure 2.4: *Propagation of an action potential along the axon. Modified figure extracted from (Guinaudeau, 2019). The red cross means that the channel is inactive, which happened during the refractory period, preventing the action potential from propagating backward and ensures unidirectional propagation.*

In myelinated axons, the myelin sheath insulates the axon and increases the speed of action potential propagation. The myelin sheath is interrupted at regular intervals by small unmyelinated regions called nodes of Ranvier, which are rich in voltage-gated ion channels, therefore a good area to generate action potentials. The action potentials "jump" from one node to the next, a process called *saltatory conduction*, significantly increasing the speed of transmission and can reach up to 150 m/s (Purves et al., 2001).

Termination : synaptic transmission

The termination phase occurs when the action potential reaches the axon terminal, which is the endpoint of the axon where it makes connections with other neurons or target cells. The gap connection, also referred as the junction, between the neuron and the target neuron/cells, is called *synapse* (see "b" in figure 2.5). Generally, the junction forms between the axon terminal of the presynaptic neuron and the dentrites of the postsynaptic neuron, called axodentritic synapse. However, there are other types of synapses including axosomatic, axoaxonic, dentrodentritic synapses, etc.

There are two kinds of synapses : chemical synapses and electrical synapses. Most common ones are the chemical synapses. At a chemical synapse, the action potential transmission occurs through the release of neurotransmitters from the presynaptic neuron into the synaptic cleft, then the neurotransmitters diffuse and bind to specific receptors on the membrane of the postsynaptic neuron or target cell. In electrical synapse, signal transmission occurs through the direct passage of the action potential (electrical current) between neurons via gap junctions, which connect the cytoplasm of adjacent cells. In the following, we will focus only on chemical synapses.

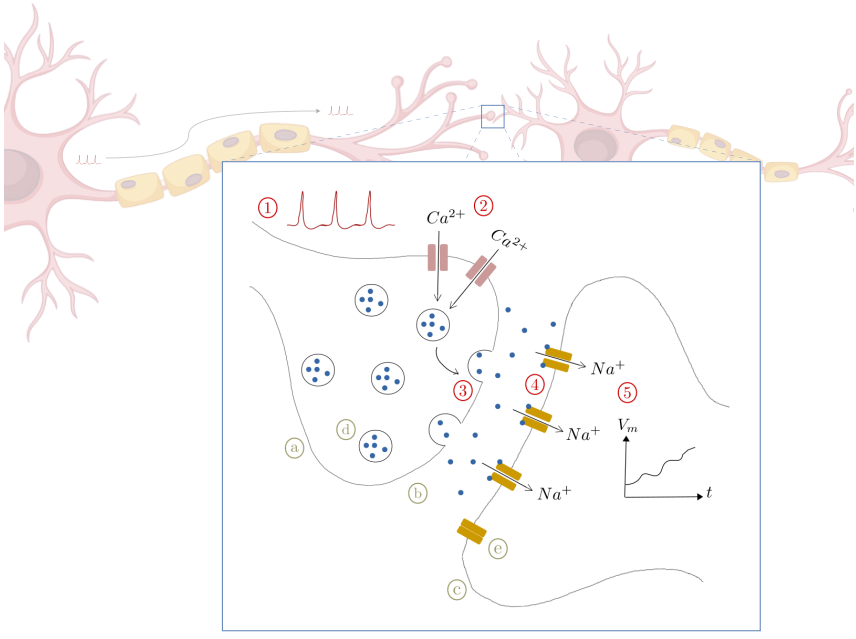


Figure 2.5: *Illustration of an excitatory synaptic transmission. Modified figure extracted from (Guinaudeau, 2019). (1) Arrival of action potentials. (2) Calcium currents. (3) Exocytosis of vesicles. (4) Binding of neurotransmitters to receptors. (5) Depolarizing currents and temporal summation of postsynaptic potentials. (a) Presynaptic neuron. (b) Synaptic cleft. (c) Postsynaptic neuron. (d) Neurotransmitter vesicle; (e) Ion channel receptor.*

When the action potential arrives at the axon terminal, it triggers the release of neurotransmitters that initiate a response in the postsynaptic neuron or target cell. In the process, first the action potential ("1" in figure 2.5) triggers the opening of voltage-gated calcium. This allows Ca2+ ions to enter the terminal, causing a rapid increase in the intracellular calcium concentration ("2" in figure 2.5). The influx of Ca2+ ions prompts the synaptic vesicles, which are small membrane-bound structures containing neurotransmitters, to fuse with the presynaptic membrane ("3" in figure 2.5). This process, known as exocytosis, releases the neurotransmitters into the synaptic cleft. The released neurotransmitters diffuse across the synaptic cleft and bind to specific receptors on the membrane of the postsynaptic neuron or target cell ("4" in figure 2.5). The binding of neurotransmitters can cause the opening of ion channels, and therefore the flowing of ions into the target cell ("5" in figure 2.5), or the activation of intracellular signaling pathways, leading to either excitatory or inhibitory effects on the postsynaptic cells. In both cases, it causes the variation of the membrane potential of the target cell.

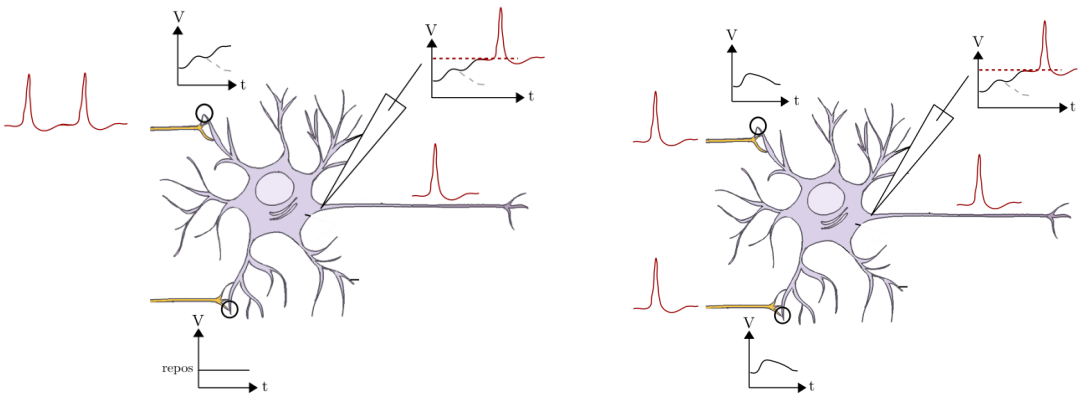
Excitatory synapse involves excitatory neurotransmitters, such as glutamate, and it increases the likelihood of the postsynaptic neuron to generate an action potential by causing depolarization of the postsynaptic membrane. This is due to the influx of positively charged ions, Na+ or Ca2+, which brings the membrane potential closer to its depolarization threshold. In contrast, inhibitory neurotransmitters, such as Gamma-Amino-Butyric Acid (GABA), decrease the likelihood of the postsynaptic neuron generating an action potential by causing hyperpolarization of the target cell's membrane. This is due to the influx of negatively charged ions, like chloride (Cl-), or the efflux of positively charged ions, like K+, which move the membrane potential further away from the threshold. We call *Excitatory Post-Synaptic Potential (EPSP)* when the action potential causes the membrane potential of the target cell to depolarize, else it is called a *Inhibitory Post-Synaptic Potential (IPSP)* which causes the polarization of the target cell's membrane.

Integration of postsynaptic potentials

Both EPSPs and IPSPs can influence the likelihood of generating an action potential in the postsynaptic neuron. Generally, one unique postsynaptic potential is not sufficient to make a neuron generate an action potential. Multiple action potentials are required, and the integration of postsynaptic potentials occurs primarily through two mechanisms : *temporal summation* and *spatial summation* (figure 2.6).

- *Temporal summation* : Temporal summation occurs when multiple postsynaptic potentials (either EPSPs or IPSPs) arrive at the same synapse in rapid succession. If the time interval between these potentials is short enough, their effects can add up or "summate" before the neuron returns to its resting state. In the case of EPSPs, this can increase the likelihood of reaching the threshold potential for generating an action potential (figure 2.6(a)). And in the case of IPSPs, it will make the membrane potential more negative;

- *Spatial summation* : Spatial summation occurs when postsynaptic potentials (either EPSPs or IPSPs) are generated simultaneously or in a short time interval at different synapses on the same neuron. The combined effect of these potentials can influence the membrane potential of the neuron, hence increasing in the case of EPSPs (figure 2.6(b)) or decreasing in the case of IPSPs, the likelihood of generating an action potential.



(a) *Temporal summation* **(b)** *Spatial summation*

Figure 2.6: *Spatio-temporal integration of postsynaptic potentials. The depolarisation threshold of a neuron can be reached through two mecanisms : (a) via a temporal summation or (b) via a spatial summation. Figure extracted from (Guinaudeau, 2019).*

Leakage

The leakage refers to the passive flow of ions through "leak" channels. These ions channels are embedded in the cell membrane, they allow specific ions to flow across the membrane, even when the neuron is at rest. They are distinct from voltage-gated ion channels which open and close in response to changes in membrane potentials. The role of leak channels is to maintain the resting membrane potential of a neuron, or to reach it back when the membrane potential of the neuron is being changed. The most common ions involved in maintaining the resting membrane potential are K+ and Na+ ions. K+ ions flow out of the cell, in more quantity than Na+ flowing in the cell, and it causes the membrane potential to be more negative. The dashed lines in the membrane potentials graphs in figure 2.6 illustrate this phenomenon.

2.1.3 Spiking neuron models

Neuron models are mathematical or computational representations that seek to describe and simulate the behavior of biological neurons. Several neuron models exist, each with their strengths and limitations depending on the level of abstraction, intended application and specific neuronal properties they aim to capture. One abstraction level is for example the type of the model : *single-compartment* or *multi-compartment* model. In a single-compartment model, the entire neuron is represented as a single electrical compartment, which simplifies the neuron's complex structure into a point-like entity. This model ensures that the neuron's membrane potential is uniform throughout the cell, neglecting the spatial distribution of ion channels and the effects of dentrites and axons on the membrane potential. The single-compartment models are generally computationally efficient (but not all of them) and can be used to simulate large-scale neuronal networks. In contrast, in a multi-compartment model, the neuron is divided into multiple interconnected electrical compartments, representing different sections of the neuron, such as the soma, the dentrites and the axon. Therefore, it allows to take into account the influence of the different parts of the neuron on the membrane potential. Although they provide a good insights on the neuron's biological mechanisms,

they are computationally expensive and are not suitable for large-scale simulations or real-time applications.

Single compartments are able to reproduce a variety of biological neuron families, as shown in figure 2.7. In the following, we will present some examples of single compartment neuron models, it is not an exhaustive list but it provides an overview of the variety of models and their applications.

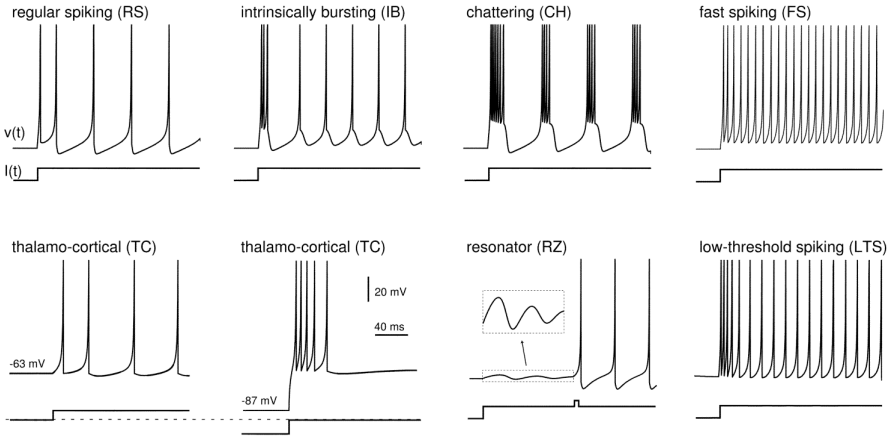


Figure 2.7: *Neurons are classified into different families, here based on their response v(t) to a stimulation I(t). Figure extracted from (Izhikevich, 2003).*

Hodgkin-Huxley (HH)

Developed in 1952 by Alan Hodgkin and Andrew Huxley, this model mathematically describes the biophysical mechanisms underlying the generation and propagation of action potentials from observations in the squid giant axon (Hodgkin & Huxley, 1952). They were awarded for their groundbreaking work in 1963. The model is based on the voltage-dependant behavior of sodium (Na+) and potassium (K+) ion channels and includes a set of differential equations that represent the dynamics of ion flow and membrane potential. The model can be represented using an electrical analog circuit in figure 2.8, to simplify the understanding of the membrane potential dynamics and the role of voltage-dependant ion channels.

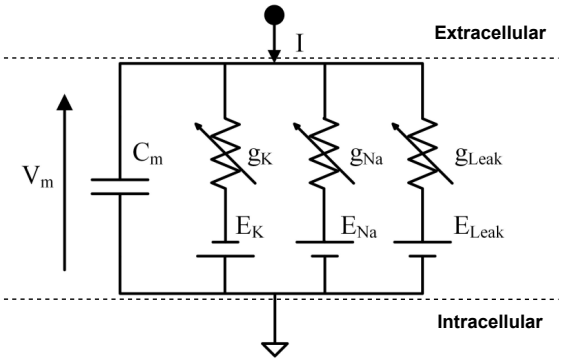


Figure 2.8: *The neuron membrane can be represented as an electrical circuit.*

The resistors, referred here as the conductances $g_{ion} = \frac{1}{R_{ion}}$, represent the ion channels (Na+, K+) in the membrane, the capacitor represents the ability of the neuron membrane to store electrical charge and the ion batteries represent the equilibrium potential of the ions across the membrane. There are additional components to account for the "leak" current, they represent the passive flow of ions across the membrane. Therefore, from the circuit in figure 2.8, the membrane potential (Vm) can be written as :

$$C_m \frac{dV_m}{dt} = I(t) - \sum I_{ion} \tag{2.1}$$

where I is the membrane input current, C_m is the membrane capacity and I_{ion} represents the different ions currents (I_K, I_{Na}, I_{Leak}). Each current can be described in terms of its conductance, the membrane potential and its equilibrium potential :

$$I_{Na} = g_{Na}.(Vm - E_{Na}) \tag{2.2}$$
$$I_K = g_K.(Vm - E_K) \tag{2.3}$$
$$I_{leak} = g_{leak}.(Vm - E_{leak}) \tag{2.4}$$

The conductance evolves in time (only the leakage is constant) and according to the membrane potential. By adding the gating variables n, m, h that represent the probability of activation of the Na+ channel, activation of the K+ channel and inactivation of the Na+ channel respectively, we obtain the differential equations :

$$C_m \frac{dV_m}{dt} = -\overline{g}_{Na}.m^3.h.(Vm - E_{Na}) - \overline{g}_K.n^4.(Vm - E_K) - \overline{g}_{leak}.(Vm - E_{leak}) + I \tag{2.5}$$

with

$$\begin{cases} \frac{dn}{dt} = \alpha_n(V_m).(1-n) - n.\beta_n(V_m) \\ \frac{dm}{dt} = \alpha_m(V_m).(1-m) - m.\beta_m(V_m) \\ \frac{dh}{dt} = \alpha_h(V_m).(1-h) - h.\beta_h(V_m) \end{cases} \tag{2.6}$$

where $\overline{g}$ is the maximal conductance, α and β are the voltage-dependant rate constants that also determine the opening and closing of the ion channels.

The HH model is very close to the physiology of a neuron, as it provides a detailed description of the activation and inactivation of various ion channels at the molecular level. However, this exhaustive representation makes the model complex and computationally expensive, therefore limiting its applicability to large-scale simulations (Catterall, Raman, Robinson, Sejnowski, & Paulsen, 2012).

Integrate-&-Fire (IF) and Leaky-Integrate-&-Fire (LIF)

The Integrate-&-Fire (IF) model is much more simplified neuron model compared to the HH model. It was introduced by Louis Lapicque (Lapicque, 1907), and it captures the overall behavior

of the biological neuron rather than the detailed biophysical process. The membrane potential in the IF neuron is described by one equation :

$$C_m \frac{dV_m}{dt} = I(t) \tag{2.7}$$

where I is the input current, V_m is the membrane potential and C_m is the membrane capacitance. When the membrane potential reaches a predefined threshold value V_{th}, the neuron "fires" an action potential, also referred to as a "spike". After firing, V_m is reset to a resting value V_{rst}, and a refractory period may be imposed, during which the neuron can not fire another spike. This model is used to study the principles of neural computation, coding and information processing in large-scale neuronal networks.

The Leaky-Integrate-&-Fire (LIF) model (Gerstner & Kistler, 2002) is a widely used variant of the IF model, as it includes a leakage current term, making the neuron's behavior more biologically plausible (biologically realistic). The leakage is represented by a resistor through which the charge stored in the capacitance gradually dissipates :

$$C_m \frac{dV_m}{dt} = I(t) - \frac{1}{R_m}.(V_m - V_{rst}) \tag{2.8}$$

where R_m a constant resistance value, also called membrane time constant τ. When the neuron is not excited, the potential V_m returns back to its resting potential state V_{rst}.

Adaptive Leaky-Integrate-&-Fire (ALIF)

The LIF model assumes a constant firing threshold in time. However, experimental evidence suggests that the threshold is not fixed and varies according to the past spiking history of the neuron (Azouz & Gray, 1999). Therefore, (Chacron, Pakdaman, & Longtin, 2003) presents a modified version of the LIF model, known as the Adaptive Leaky-Integrate-&-Fire (ALIF) which incorporates a dynamic threshold variable. The ALIF is described by 2 differential equations :

$$\frac{dV_m}{dt} = I(t) - \frac{1}{\tau_m}.(V_m - V_{rst}) \tag{2.9}$$

$$\frac{dV_{thres}}{dt} = \frac{1}{\tau_{thres}}(V_t - V_{thres}) \tag{2.10}$$

$$\text{if} \quad V_m \geq V_{thres}, \ \text{then} \ \begin{cases} V_m = V_{rst} \\ V_{thres} = V_{thres} + W \end{cases} \tag{2.11}$$

where τ_m is the time constant of the membrane, V_{thres} is the dynamic threshold variable, V_t is the value at which the threshold stabilizes in the absence of firing, and W is the increasing factor of the threshold. When the neuron reaches the threshold value, V_m is reset to its resting potential V_{rst} and V_{thres} is increased by W. W can be constant, a linear or non-linear function.

These new dynamics represent the threshold fatigue of the neuron, meaning that every time the neuron emits a spike, its next emission will require more stimulation or will take longer than the previous emission. After a certain moment of time of not firing, defined by τ_{thres}, the neuron will

recover its initial threshold. The neuron dynamic shows adaptation *i.e.* gradual change in firing rate, a feature that is absent from the standard LIF model : the neuron firing rate decreases over time in response to a constant stimulus.

Izhikevich (IZH)

The Izhikevich (IZH) model (Izhikevich, 2003) is a simplified version of the HH model by using the bifurcation theory. It captures the spiking and bursting of various types of cortical neurons (figure 2.7) while remaining computationally efficient. The IZH model is described by a system of 2 differential equations :

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{2.12}$$

$$\frac{du}{dt} = a(bv - u) \tag{2.13}$$

$$\text{if} \quad v \geq 30mV, \ \text{then} \ \begin{cases} v = c \\ u = u + d \end{cases} \tag{2.14}$$

where v represents the membrane potential, u is the membrane recovery variable and I is the input current. The parameters a, b, c and d are the constants that determine the specific properties and dynamics of the neuron. When the membrane potential v reaches 30 mV, the neuron is considered to have generated a spike, and v and u are reset according to 2.14. The IZH model is designed to reproduce various firing patterns but with a lower computational cost than the HH model for example, making it suitable for large-scale simulations of SNNs.

Digital Spiking Silicon Neuron (DSSN)

The Digital Spiking Silicon Neuron (DSSN) neuron model (Kohno & Aihara, 2007; Nanami & Kohno, 2016) is a qualitative neuron model that can simulate several classes of neuronal activities as the HH model, given the appropriate set of parameters. The model is more precise that the IZH model, and is described by a set of 3 differential equations and 3 functions :

$$\frac{dv}{dt} = \frac{\phi}{\tau}(f(v) - n - q + I_0 + I_{stim}) \tag{2.15}$$

$$\frac{dn}{dt} = \frac{1}{\tau}(g(v) - n) \tag{2.16}$$

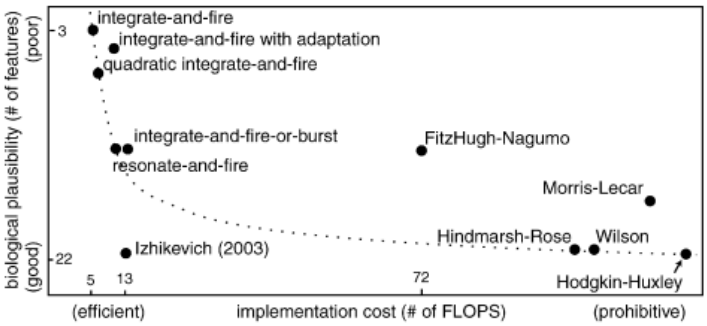
$$\frac{dq}{dt} = \frac{\epsilon}{\tau}\big(h(v) - q\big) \tag{2.17}$$

$$f(v) = \begin{cases} a_{fn}(v - b_{fn})^2 + c_{fn} & (v < 0) \\ a_{fp}(v - b_{fp})^2 + c_{fp} & (v \geq 0) \end{cases} \tag{2.18}$$

$$g(v) = \begin{cases} a_{gn}(v - b_{gn})^2 + c_{gn} & (v < r_g) \\ a_{gp}(v - b_{gp})^2 + c_{gp} & (v \geq r_g) \end{cases} \tag{2.19}$$

$$h(v) = \begin{cases} a_{hn}(v - b_{hn})^2 + c_{hn} & (v < r_h) \\ a_{hp}(v - b_{hp})^2 + c_{hp} & (v \geq r_h) \end{cases} \tag{2.20}$$

where v is the membrane potential, n and q are the fast and slow variables, respectively, that describe the activity of the ion channels. I_0 is a bias constant and I_{stim} is the input current. The parameters ϕ, ϵ and τ control the time constants of the variables. Parameters r_x, a_x and c_x, where $x = fn, fp, gn, gp, hn$ or hp, are constants *that adjust the nullclines of the variables* (Nanami & Kohno, 2016).



Models	biophysically meaningful	tonic spiking	phasic spiking	tonic bursting	phasic bursting	mixed mode	spike frequency adaptation	class 1 excitable	class 2 excitable	spike latency	subthreshold oscillations	resonator	integrator	rebound spike	rebound burst	threshold variability	bistability	DAP	accommodation	inhibition-induced spiking	inhibition-induced bursting	chaos	# of FLOPS
integrate-and-fire	-	+	-	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	5
integrate-and-fire with adapt.	-	+	-	-	-	-	+	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	10
integrate-and-fire-or-burst	-	+	+		+	-	+	+	-	-	-	-	+	+	+	-	+	+	-	-	-		13
resonate-and-fire	-	+	+	-	-	-	-	+	+	-	+	+	+	+	-	-	+	+	+	-	-	+	10
quadratic integrate-and-fire	-	+	-	-	-	-	-	+	-	+	-	-	+	-	-	+	+	-	-	-	-	-	7
Izhikevich (2003)	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	13
FitzHugh-Nagumo	-	+	+	-		-	-	+	-	+	+	+	-	+	-	+	+	-	+	+	-	-	72
Hindmarsh-Rose	-	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	120
Morris-Lecar	+	+	+	-		-	-	+	+	+	+	+	+	+		+	+	-	+	+	-	-	600
Wilson	-	+	+	+			+	+	+	+	+	+	+	+	+		+	+					180
Hodgkin-Huxley	+	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	1200

Figure 2.9: *Comparison of spiking neuron models in terms of bio plausibility and the number of FLOPS (floating point operations such as addition, multiplication, etc.) needed to simulate the model during a 1 ms time span. Extracted from (Izhikevich, 2004).*

There is a wide variety of other neuron models that differ in their complexity, level of abstraction and biological plausibility. The choice of the appropriate model depends on the specific application and the trade-off between the model accuracy and computational efficiency. For example, (Izhikevich, 2004) reviews various neuron models in terms of their biological plausibility, *i.e.* number of neuron class behaviors the model can reproduce, and their computational efficiency (figure 2.9).

In this work, we do not choose a specific neuron model. Instead, we aim to model various models using the synchronous approach to simulate and apply verification tools to these models. As we explain in later chapters, one objective is to simulate and to implement SNN in hardware with the option to choose a neuron model and the neural properties to apply to the neural network.

2.2 Models of neural properties

Neural properties are essential characteristics of neurons and neural networks that enable them to dynamically process, transmit and store information. These properties include neural dynamics such as AMPA or GABA dynamics, short-term and long-term plasticity, axonal delay, synaptic noise, all of which play a role in the functioning and dynamics of neural networks. We will examine these neural properties, focusing on their underlying mechanisms and implications for the modeling and simulation of more biologically plausible SNNs. Understanding these properties and implementing these properties not only make the SNN more biologically realistic, but also give insights of the cognitive processes such as the learning or memory.

2.2.1 AMPA and GABA dynamics

AMPA (α-amino-3-hydroxy-5-methyl-4-isoxazolepropionic) is a receptor for glutamate neurotransmitter, an excitatory neurotransmitter. When a neuron receives an excitatory signal through AMPA receptors, it increases the likelihood that the neuron will trigger an action potential. GABA (γ-aminobutyric acid) is an inhibitory neurotransmitter. GABA receptors are responsible for receiving inhibitory signals, which decrease the likelihood that a neuron will trigger an action potential. Biological observations have shown that when receiving an excitation (or an inhibition), the effect on the membrane potential tends to decay exponentially toward zero (Ben-Ari, Khazipov, Leinekugel, Caillard, & Gaiarsa, 1997). This decay follows a time constant, referred to as τ_{exc} (for excitatory processes) or τ_{inh} (for inhibitory processes), which determines the rate at which the excitations or inhibitions diminish.

In (Ambroise, 2015), they modeled the post-synaptic changes with an exponential equation :

$$I_{syn}(t) = I_{max}.e^{-\frac{t}{\tau_{syn}}} \tag{2.21}$$

where I_{syn} is the post-synaptic current, I_{max} is the maximum post-synaptic current and τ_{syn} is either the time constant for excitatory or inhibitory synapse.

2.2.2 Plasticity

The Short-Term Plasticity and Spike-Timing-Dependent Plasticity are fundamental properties of neural networks that allow them to adapt and change in response to incoming stimuli. These forms

of plasticity specifically affect the post-synaptic potential in time, and are for example important for learning and memory formation. In addition, there are other learning techniques used in the field of neural networks, such as offline learning methods like the *back-propagation* technique (Schuman et al., 2017; Sze, Chen, Yang, & Emer, 2017), an off-line learning algorithm (LeCun et al., 1989). However, we will focus on the biologically-inspired methods, as they correspond to online algorithms for spiking neural networks model.

Short-Term Plasticity (STP)

Short-Term Plasticity (STP) is an algorithm that refers to the synaptic strength modifications that occur on a short timescale (milliseconds to seconds) (Zucker & Regehr, 2002; Beierlein, Gibson, & Connors, 2003). It is associated to the changes in the amount of neurotransmitter released by the presynaptic neuron (see section 2.1.2), the sensitivity of the postsynaptic receptors, or both, following an action potential. There are two main types of short-term plasticity: *facilitation* and *depression*. Facilitation occurs when the synaptic strength increases temporarily due to repeated presynaptic stimulation. It corresponds to an increased probability of neurotransmitter release or enhanced postsynaptic receptor sensitivity. Depression refers to a temporary decrease in synaptic strength following high-frequency stimulation. This reduction in synaptic efficacy is usually related to the depletion of neurotransmitter vesicles in the presynaptic terminal or desensitization of postsynaptic receptors. The transient nature of these changes allows neural networks to adapt rapidly to incoming stimuli. It is for example used to model Central Pattern Generator (CPG), a neural network responsible for generating rhythmic patterns (Hill, Lu, Masino, Olsen, & Calabrese, 2001; Hill, Masino, & Calabrese, 2002) of motor activity, such as those involved in locomotion, breathing, etc. The CPGs will be addressed in chapter 6. The STP algorithm, proposed in (Izhikevich & Edelman, 2008), is described in table 2.1.

When a spike is received	**When no spike has been received**
$I_{syn}[n+1] = I_{syn}[n] + W_{syn}[n]$ with $W_{syn}[n] = x_{syn}[n].W_s$	$I_{syn}[n+1] = I_{syn}[n] - \frac{1}{\tau_I}.\ I_{syn}[n]$
$x_{syn}[n+1] = P.\ x_{syn}[n]$	$x_{syn}[n+1] = x_{syn}[n] + \frac{1}{\tau_x}(1 - x_{syn}[n])$

Table 2.1: *Equations that described the STP algorithm. The two cases are : the postsynaptic neuron receives an action potential or not. Table extracted and modified from (Ambroise, 2015).*

In table 2.1, I_{syn} is the input current or stimulation to the postsynaptic neuron receiving the spike. The scalar factor x_{syn} indicates the state of the synapse (facilitation or depression) by modulating the synaptic weight. The value of x_{syn} always tends toward 1. A percentage P is multiplied by the x_{syn} factor each time a presynaptic spike is received. If the percentage is greater than 1, the synapse will exhibit short-term facilitation, otherwise the synapse will exhibit short-term depression. The constants τ_I and τ_x are the time constant of I_{syn} and x_{syn}, respectively, and they control how fast these variables return to their initial values. The figure 2.10 shows the comparison of STP recorded *in vitro* (see section 2.4.2) and the model described in table 2.1.

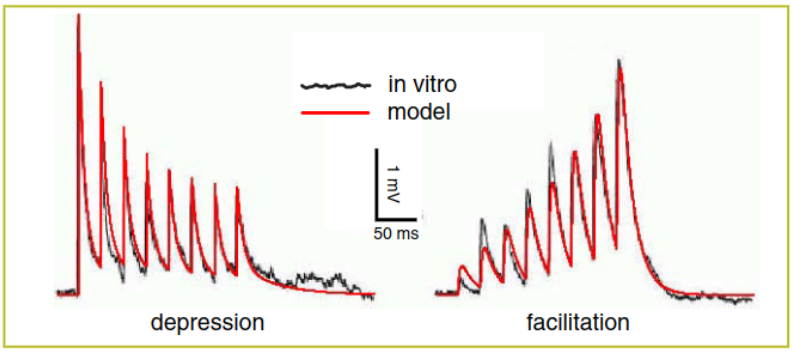


Figure 2.10: *Comparison of STP recorded in vitro (black noisy curve) and simulated (red smooth curve) by the model. On the left, high-frequency of input spikes decreases the effect of the postsynaptic potential on the postsynaptic neuron. On the right, high-frequency input spikes increases the effect of the postsynaptic potential on the postsynaptic neuron. Extracted from (Izhikevich & Edelman, 2008).*

Spike-Timing-Dependent Plasticity (STDP)

Long-term plasticity, also known as Spike-Timing-Dependent Plasticity (STDP), is a form of synaptic plasticity that occurs on a long timescale of minutes, hours, days or years (Feldman, 2012). It is characterized by long-lasting changes in synaptic strength, which can be either potentiation (strengthening) or depression (weakening) of synaptic connections. The STDP rule is based on the timing of pre- and post-synaptic spikes, and can explain mechanisms such as memory or learning.

Basically, the STDP reinforces synapses that have contributed to the postsynaptic neuron reaching its threshold while weakening less active synapses. Hebb's postulate suggests that "if neuron A persistently or repeatedly stimulates neuron B, causing B to generate an action potential, cellular or molecular changes occur to increase the efficiency of A's action on B" (Hebb, 1949a). This principle forms the basis of understanding the Long-Term Potentiation (LTP) and the Long-Term Depression (LTD). LTP is the long-lasting increase in synaptic strength following high-frequency stimulation, LTD is the long-lasting decrease in synaptic strength following low-frequency stimulation. Recent studies have emphasized the importance of precise timing between pre- and postsynaptic action potentials (Markram, Lübke, Frotscher, & Sakmann, 1997; Bi & Poo, 1998; D'amour & Froemke, 2015). A general rule has emerged in the context of STDP: LTP is induced when the presynaptic neuron fires an action potential before the postsynaptic neuron, while LTD is induced when the presynaptic neuron fires an action potential after the postsynaptic neuron. The amplitude of plasticity depends on the delay between pre- and postsynaptic action potentials, with the modification of synaptic efficacy being stronger when the action potentials are closer together. This amplitude decreases gradually with increasing time difference and becomes null after a few tens of milliseconds (see figure 2.11).

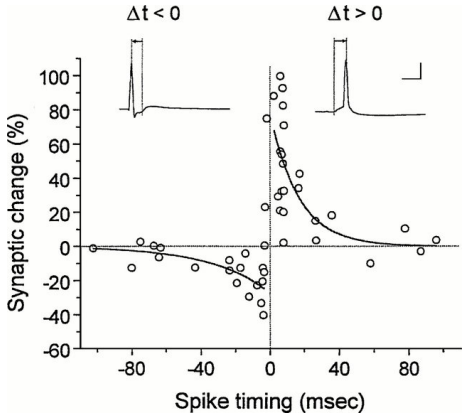


Figure 2.11: *Temporal window for the induction of LTP and LTD. Each circle denotes the percentage change in the amplitude of the post-synaptic potential, as a function of the delay (Δt) between the post-synaptic neuron's action potential and the pre-synaptic neuron's action potential. The 2 curves are fitted to the experimental data (circles). Figure extracted from (Walter et al., 2016), which is a modified version of (Bi & Poo, 1998).*

The formulation of the original STDP proposed in (Song, Miller, & Abbott, 2000) is described by the equation :

$$F(\Delta t) = \begin{cases} A^+ e^{\Delta t/\tau^+} & \text{if } \Delta t < 0 \\ -A^- e^{-\Delta t/\tau^-} & \text{if } \Delta t > 0 \end{cases} \tag{2.22}$$

where A^+ and A^- are the amplitudes of the potentiation and depression, respectively. The τ^+ and τ^- are the time constants related to the potentiation and the depression, respectively. And Δt is the time difference between the pre-synaptic spike and the post-synaptic spike.

2.2.3 Axonal delay

Axonal delay refers to the time it takes for an action potential to propagate along the axon from the soma of the presynaptic neuron to the soma of the postsynaptic neuron (including axon terminal + dendrites propagation). This delay plays a crucial role in the overall timing and synchronization of neural activity within the brain.

The speed at which action potentials travel along axons is influenced by several factors, such as axon diameter, myelination, and the density of ion channels (Waxman, 1980). Larger axons generally exhibit faster conduction velocities due to reduced internal resistance to the flow of electric currents. Myelinated axons can transmit action potentials more rapidly compared to unmyelinated axons, as myelination allows for the phenomenon of saltatory conduction, in which action potentials effectively "jump" between nodes of Ranvier (Hartline & Colman, 2007).

Axonal delay plays a significant role in various neural processes such as sensory perception, motor control, and learning. For example, it has been shown that the precise timing of action potentials in the auditory system is crucial for sound localization (McAlpine, Jiang, & Palmer, 2001). Similarly, axonal delays are essential in the cerebellum, where they contribute to the precise coordination of motor actions and the formation of internal models for motor control (Dean, Porrill, Ekerot, & Jörntell, 2010).

Incorporating axonal delays into the models can lead to more biologically plausible and potentially more efficient networks (Izhikevich, 2006). Understanding and accounting for axonal delays in the design of neuromorphic hardware and software could help improve the performance and functionality of these systems, bringing them closer to the computational capabilities of biological neural networks.

2.2.4 Biological noise

Biological noise is an inherent feature of neurons and neural networks, which can result from various sources, such as stochastic ion channel behavior, random neurotransmitter release, and fluctuations in membrane potential (Faisal, Selen, & Wolpert, 2008). Although noise is often considered a detriment to signal processing and information transmission, it has been suggested that it might play a functional role in neural computation and encoding (McDonnell & Ward, 2011).

In the context of biological neurons, the main sources of noise can be divided into two categories: intrinsic and extrinsic noise. Intrinsic noise arises from the inherent randomness of cellular processes, such as the stochastic opening and closing of ion channels and the probabilistic release of neurotransmitters at the synapses (Stein, Gossen, & Jones, 2005). Extrinsic noise, on the other hand, originates from fluctuations in the external environment, such as variations in sensory input or synaptic activity from neighboring neurons (Destexhe & Rudolph-Lilith, 2012).

Recent studies have shown that the presence of biological noise can have significant consequences for neural computation. For instance, stochastic resonance is a phenomenon in which the presence of an optimal level of noise can enhance the detection and transmission of weak signals in neural systems (McDonnell & Abbott, 2009). Moreover, some computational models have shown that noise can facilitate the generation of complex and irregular spiking patterns, which can contribute to the efficient encoding and representation of sensory information (Faisal et al., 2008).

By incorporating noise into the design of spiking neural networks, researchers can potentially improve the robustness and adaptability of these models, allowing them to better mimic the behavior of biological neural networks (Izhikevich, 2007; Lansner, 2009).

There are different approaches to model and incorporate noise in neural network. For example, by adding random fluctuations in input signals such Gaussian noise or other type of noise distributions (Bishop & Nasrabadi, 2006). Another example is by adding the noise into the neuron model (Levitan, Segundo, Moore, & Perkel, 1968; Tuckwell, Wan, & Rospars, 2002; Brette & Gerstner, 2005). In this work, we implemented a noise model based on the Ornstein-Uhlenbeck (OU) (Brette & Gerstner, 2005), that has been implemented and presented in (Ambroise, 2015; Grassia, Kohno, & Levi, 2016; Khoyratee, Grassia, Saïghi, & Levi, 2019) for digital simulation of biologically plausible neural networks. The OU model is described by the following equations :

$$\frac{dI_{noise}}{dt} = \theta(\mu - I_{noise}) + \sigma\frac{dW}{dt} \tag{2.23}$$

where I_{noise} is a noisy current, θ and σ are parameters to control the noise, μ is the mean value and W is a variable of the Wiener process that represents the random fluctuations. By modifying these parameters, the process can be tuned to capture various levels of noise and dynamics. The discretized version of the process is given by :

$$I_{noise}[n+1] = I_{noise}[n] + \theta(\mu - I_{noise}[n])\Delta t + \sigma\Delta W[n] \tag{2.24}$$

Note that, we have implemented this model, but the integration of this model in the global architecture is still ongoing.

2.3 Spiking Neural Network (SNN) behavorial simulation

Spiking Neural Networks (SNNs) are computational models inspired by the organization and the functionality of biological neural networks, and they consist of interconnected neurons that process and transmit information. SNNs are specifically designed to deal with information in the form of action potentials or spikes, hence their name. The simulation of neural networks is really important for understanding the principles governing neural information processing, testing hypotheses and developing novel computational architectures for artificial intelligence applications.

There are 2 main different approaches to modeling and simulating SNNs, software-based and hardware-based solutions.

2.3.1 Software approaches

Software-based approaches to SNN simulation have become increasingly popular due to the flexibility they provide for exploring various aspects of neural dynamics, network topologies and learning algorithms. Multiple frameworks and tools have been developed, and we will list the most popular.

The *NEURON* environment (Hines & Carnevale, 1997) is a simulation environment for modeling individual neurons and networks of neurons. It provides a high-level interface for defining neuron models and network connectivity and supports a wide range of neural models, including point neurons (simplified models that consider the neuron as a single point without spatial extent or internal structure), compartmental models and cable theory-based models. NEURON is particularly well-suited for simulating biophysically detailed models that incorporate realistic membrane dynamics and synaptic mechanisms (Carnevale & Hines, 2006).

The *NEST* simulator is specifically designed for large-scale SNNs with a focus on the dynamics, size, and structure of neural systems rather than the exact morphology of individual neurons (Gewaltig & Diesmann, 2007). It offers an efficient and highly-scalable simulation framework that can handle networks containing millions of neurons and billions of synapses. NEST supports a variety of neuron and synapse models and includes advanced features such as parallel and distributed computing, allowing researchers to simulate large-scale SNNs on multi-core processors and computer clusters.

BRIAN is a flexible and user-friendly Python-based simulator for SNNs (Goodman & Brette, 2008). It provides a simple and intuitive interface for defining neural models using mathematical equations and supports a wide range of neuron and synapse models. Brian is designed to be easily extensible, allowing users to develop custom models and simulation methods. It also includes built-in support for parallel computing and GPU acceleration (Stimberg, Goodman, & Nowotny, 2018). However, it is less optimized for large-scale simulations compared to other tools (Stimberg, Brette, & Goodman, 2019).

SpiNNaker is a software and hardware platform designed for simulating SNNs in real-time (Furber, Galluppi, Temple, & Plana, 2014). It features a custom-built massively parallel processor architecture optimized for neural network simulations and a flexible software environment that supports various neural models and learning algorithms. SpiNNaker has been used to simulate

networks containing up to a million neurons and billions of synapses, demonstrating its potential for large-scale SNN research (Furber et al., 2012).

SpikingJelly (Feng, Xu, Yang, & Shi, 2020) is an emerging deep learning framework designed for spiking neural networks (SNNs) based on the popular deep learning framework PyTorch (Ketkar, Moolayil, Ketkar, & Moolayil, 2021). It aims to provide an efficient and user-friendly platform for developing and training SNNs on both CPUs and GPUs. SpikingJelly offers a variety of neuron models, learning rules, and encoding schemes, enabling researchers to experiment with different SNN architectures and algorithms. By leveraging the flexibility and powerful features of PyTorch, SpikingJelly makes it easy for researchers and developers to build, train, and evaluate SNNs for various applications, such as computer vision, speech recognition and robotics.

For the purposes of this thesis, which focuses on developing and testing non-deep learning and bio-realistic models or architectures, the *BRIAN* simulator framework was selected as the software approach. The choice was driven by its user-friendly Python-based interface and the ability to define neural models using mathematical equations. Our use of BRIAN is detailed in chapter 4 (section 4.1.1) and in chapter 6.

2.3.2 Hardware approaches

Hardware-based approaches to SNN simulation, also called Neuromorphic hardware, have gained significant interest in recent years due to their potential for high-speed / real-time simulations. The ability to perform computations in parallel, closely mimicking the natural operation of biological neurons, leads to significant performance improvements and energy efficiency when compared to traditional software-based simulators running on general-purpose processors (Furber et al., 2014).

Hardware-based approaches generally involve specialized computing platforms, such as Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). Hardware solutions for SNN simulation can be broadly categorized into *analog* and *digital* implementations. Analog implementations aim to closely mimic the behavior of biological neurons and synapses using continuous voltage or current signals. These approaches can provide high energy efficiency and real-time performance, but they may suffer from limited precision, variability, and noise susceptibility. Some examples of analog hardware platforms for SNN simulation include Neurogrid (Benjamin et al., 2014) and HICANN (Schemmel et al., 2010). Digital implementations, on the other hand, use discrete signals to represent spikes and neuronal states. They offer greater flexibility, programmability, and scalability compared to analog implementations, but may require more computational resources and power consumption.

FPGAs are re-configurable digital devices that allow users to create custom hardware circuits by programming a large array of logic blocks and interconnects. FPGAs provide a high level of flexibility and can be reprogrammed to implement a wide range of neural models and network topologies. Other main advantages of using FPGAs for SNN simulation are their capability to achieve real-time, low-power, low-surface performances, which can be crucial for certain applications, especially embedded applications. However, FPGA-based solutions may be limited in terms of scalability and power efficiency compared to ASICs. We will not focus on ASICs, but generally they are specialized integrated circuits designed to perform specific tasks. They offer the potential for high-performance and power-efficient SNN simulations, but they are less flexible than FPGAs, as they are tailored for a specific neural model, network architecture or application. Developing an

ASIC can be a complex and time-consuming process, making it less suitable for rapid prototyping and exploration of different SNN models.

Some devices, namely SOCs (System On Chips), include one or several CPUs alongside the programmable logic array, which offers possibilities for both software and hardware programming such as SpiNNaker (Furber et al., 2014). In this work, we have chosen to focus on FPGA-based digital solutions for SNN simulation due to their flexibility and potential for real-time performance. Table 2.2 compares some neuromorphic hardware architectures mainly for deep learning applications, and we will introduce some of them, and discuss their advantages/disadvantages.

Chip	Type	Learning	Programming	Neuron	Neural prop.
SpiNNaker (2010)	D	On & Off	PyNN	program.	conductance-based axonal delay
NeuroGrid (2014)	A&D	On & Off	NGPython	Adapt. IF	conductance-based
TrueNorth (2014)	D	Offline	Corelets	LIF	axonal delay
Loihi (2018)	D	On & Off	Loihi API	program.	axonal delay noise
Minitaur (2014)	D	Offline	RTL	LIF	conductance-based
Fast pipeline (2015)	D	Offline	RTL	LIF	-
HFirst (2015)	D	Offline	RTL	Complex-IF	-
BrainScaleS (2017)	A&D	On & Off	PyNN	Exp-IF	conductance-based axonal delay analog noise
DYNAPS (2017)	A&D	Offline	CHP	Exp-IF	conductance-based analog noise
ConvNode (2018)	D	Offline	RTL	LIF	-
This work (2023)	D	On & Off	RTL	program. (IF, LIF, Adapt. LIF, IZH, DSSN)	conductance-based axonal delay noise

Table 2.2: *Neuromorphic systems comparison. "D" is for digital, "A" for analog, "On" for online, "Off" for offline, "program." for programmable and "prop." for properties. The last column lists the known neural properties the chip is capable to simulate. "Conductance-based" refers to the synapse model, taking into account the dynamics of ion channels and the changes in conductance in time. Otherwise, the synapse strength is represented only by a constant current being added to the post-synaptic neuron. "-" means that there were no information found. Table extracted and modified from (Abderrahmane et al., 2020).*

IBM's *TrueNorth* is a neuromorphic chip developed by IBM that aims to mimic the architecture and operation of the human brain (Merolla et al., 2014). The chip consists of an array of digital neurons and synapses that can be programmed to implement various neural models and learning

algorithms. TrueNorth is designed to be energy-efficient, making it suitable for use in embedded systems and edge computing applications.

More recently, Intel's *Loihi* is a neuromorphic research chip developed by Intel that features a many-core, asynchronous, and event-driven architecture designed for energy-efficient SNN simulation (Davies et al., 2018). It supports on-chip learning, enabling the implementation of various learning algorithms without the need for external computation. Loihi has been used for a wide range of applications, from robotics to natural language processing. And the version 2 of Loihi has been released in 2023.

2.4 Interfacing artificial with biological neurons : Neurobiohybridization

SNNs have been employed in various applications including pattern recognition (Tavanaei, Ghodrati, Kheradpisheh, Masquelier, & Maida, 2019), robotics (Chicca, Stefanini, Bartolozzi, & Indiveri, 2014; Krichmar, 2018), neuromorphic computing (Furber et al., 2014), Brain-Computer interfaces (BCIs) (Kasabov, 2014)(Lorach et al., 2023) and more. As research in neuroscience and neural engineering progresses, another interest is growing in *neurobiohybrid* systems (Vassanelli & Mahmud, 2016). Neurobiohybrid research is an interdisciplinary field that combines living and artificial systems. On the one hand, the living part consists of neurons in the form of individual cells or networks. On the other hand, the artificial part is represented by devices that receives and processes the information from the biological neurons through an interface that ensures the recording and translation from the biology to the artificial, and from the artificial to the biology. For example, the figure 2.12 illustrates the basic concept of interfacing a biological neural network (BNN) and an artificial neural network (ANN) (can be referred as SNN as well).

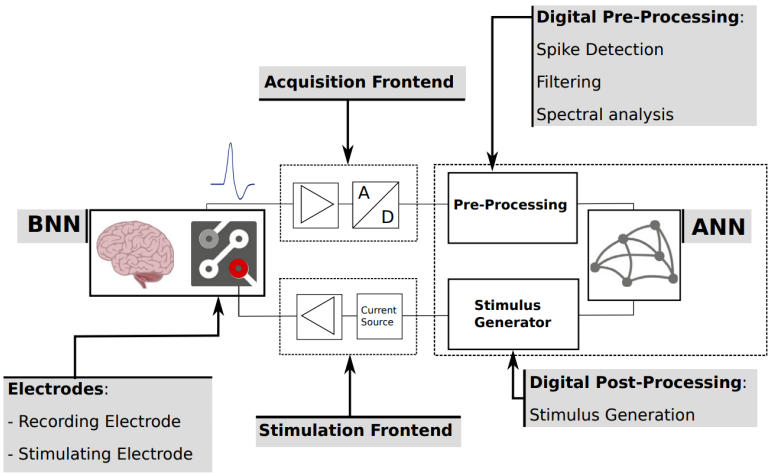


Figure 2.12: *Basic concept of interfacing a biological neural network (BNN) and an artificial neural network (ANN). The physical interface to the BNN is implemented via stimulation and recording technologies. Acquired signals are amplified and digitized in the acquisition frontend, before they are further processed to extract informative features that serve as input to the ANN. ANN output is used to determine a stimulation protocol, which control the current or voltage drivers connected to the stimulation electrodes to close the loop. Figure extracted from (George et al., 2020).*

In this section, we will introduce this application domain in which this thesis is positioned. So we will talk about some experiment technologies and methodologies, existing applications and the challenges in this domain.

2.4.1 Applications and objectives

There are various applications in interfacing artificial and biological neurons, here we introduce briefly a list of some of them :

- *Neuroprosthetics and Brain-Computer Interfaces* : one of the most well-known applications is in the development of brain-computer interfaces (BCIs) and neuroprosthetics (Lebedev & Nicolelis, 2006). Research on neural prosthetics is a contemporary subject with significant advancements in recent years (Jung et al., 2001). Projects like ANR HYRENE contribute to this area of research (Joucla et al., 2016). These devices aim to restore or augment lost motor, sensory or cognitive functions by directly connecting the brain with external devices or artificial limbs (Collinger et al., 2013).
- *Rehabilitation and recovery after brain injuries*: another application area is in developing therapies and interventions for patients with brain injuries or neurodegenerative diseases such as Parkinson, Alzheimer, etc. This diseases are characterized generally by the gradual loss of neurons, leading to cognitive, motor or functional impairments (Erkkinen, Kim, & Geschwind, 2018). By replacing or bridging the damaged neural tissue, it may be possible to facilitate recovery and restore lost functions (Buccelli et al., 2019).
- *Modeling and understanding neuronal mechanisms*: interfacing biological neurons with artificial systems can also provide valuable insights into the mechanisms underlying neural processing, learning and adaptation (Douglas & Martin, 2004; Moxon & Foffani, 2015). By studying the interactions between biological and artificial neurons, researchers can gain a better understanding of neural computation and information processing in the brain.
- *Hybrid Systems for Artificial Intelligence and Robotics*: a promising application is in the development of hybrid systems that combine biological and artificial components for advanced AI and robotic applications (DeMarse, Wagenaar, Blau, & Potter, 2001; Warwick et al., 2010). Such systems can potentially leverage the unique capabilities of biological neurons, such as adaptation, learning and fault tolerance, to create more robust and efficient AI systems.

Indeed, as various applications exist, it is essential to emphasize that the initial ultimate goal of this work is to model artificial biomimetic neural networks capable of interacting with biological neurons, by reproducing their essential properties and mechanisms, to understand neuronal mechanisms.

2.4.2 Technologies and methodologies

In the following, we will discuss about various technologies and methodologies used for interfacing external devices with biological neurons, including the different types of experiments, some techniques and technologies employed for recording biological neurons, and the technologies to collect, process and analyze the recorded data as well as provide stimulation back to the biological system.

Types of experiments : *In vivo*, *ex vivo*, *in vitro*, open-loop, closed-loop

In vivo experiments involve the study of neural interfaces within living organisms. These experiments are particularly valuable for understanding the complex interactions of biological systems in their native environment. However, it sometimes requires *invasive* techniques, meaning surgical operations for recording biological activity. *In vivo* experiments include Deep Brain Stimulation (DBS) for Parkinson's disease treatment (Benabid, Chabardes, Mitrofanis, & Pollak, 2009), optogenetic stimulation of neural circuits (Deisseroth, 2011) and brain-computer interfaces (BCIs) for motor prosthetics (McFarland & Wolpaw, 2008). *Ex vivo* experiments are conducted on living tissues that have been removed from their original organism. These experiments provide a controlled environment to study biological neurons, while still maintaining some of the complexity of the native tissue. An example of an *ex vivo* experiment is the use of brain slices to study the effects of electrical or optogenetic stimulation on neural activity (Mohajerani, McVea, Fingas, & Murphy, 2010). *In vitro* experiments are performed using isolated cells in a controlled environment, such as cultured neurons. These experiments offer a high degree of control over experimental conditions, allowing researchers to study the fundamental mechanisms underlying biological neurons. Examples of *in vitro* experiments include the study of synapse formation between cultured neurons and artificial synapses (Müller et al., 2015).

Experiments can also be categorized into either *open-loop* or *closed-loop* setups. In open-loop experiments, neural signals are recorded and processed without providing any real-time feedback to the biological system. This type of experiment could be used to control the biological part (resp. artificial part) via the artificial part (resp. biological part) (Ambroise et al., 2017). In contrast, closed-loop experiments involve real-time interaction with the biological system, both artificial and biological systems evolving and working together. This type of experiment allows to study for example adaptation or learning mechanisms in real-time (Ambroise et al., 2017; Buccelli et al., 2019).

In this thesis, we have chosen to focus on *in vitro* experiments. Working with *in vitro* experiments allows for precise control of experimental conditions, such as growing and isolating specific cells, as well as the ability to work with various sizes of biological neural networks. This way it is easier to study the behavior of individual cells or growing cells gradually forming networks. Furthermore, *in vitro* studies help avoid many ethical issues, particularly those associated with animal experimentation. However, it is important to note that cell cultures are less complex than neural networks found in the brain, and their random growth limits their physiological relevance compared to biological systems with established structures and functions. Like other approaches, *in vitro* experiments also present challenges in replicating the exact same conditions due to the random nature of cell growth or behavior.

Recording biological neurons

There are different technologies for recording the activity of biological neurons. Some of them are shown in figure 2.13. In this figure, the techniques differ from their temporal resolution (ability to capture changes in neural activity over time), spatial resolution (ability to distinguish and localize the source of neural activity), the physical scale (size of the region being recorded) and the depth scale (the depth at which neural activity can be detected).

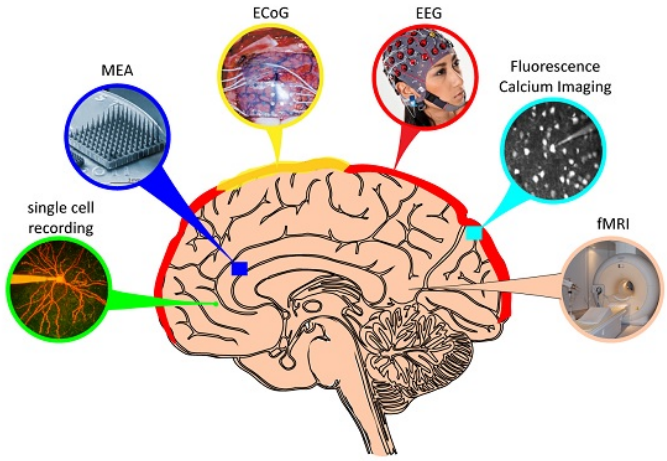


Figure 2.13: *Commonly used recording techniques of biological neuron activities in the brain. From left to right, temporal resolution decreases, from <1 ms for single cell and Micro-Electrode Array (MEA) recordings to ~1 sec for functional Magnetic Resonance Imaging (fMRI). The colors indicate the approximate physical scale of the activity that can be recorded with each approach, as well as the approximate depth limits of each technique. ECoG, EEG and fluorescence imaging are limited to recording from the brain's outer surface. Extracted from (The University of Queensland, 2023).*

Some techniques are not limited to one type of experiment, so briefly :

- *Single-Cell Recording* : it involves the use of fine-tipped electrode to measure the electrical activity of an individual neuron (Neher & Sakmann, 1976; Broccard, Joshi, Wang, & Cauwenberghs, 2017). It has then a high temporal and spatial resolutions, providing detailed information about the neuron's membrane potential and its response to various stimuli. This technique can be used for *in vivo* or *in vitro* experiments, and provides precise investigation on neuron properties.

- *Micro-Electrode Array (MEA)* : it consists of arrays of small electrodes that are used to record the extracellular activity of multiple neurons simultaneously (Obien, Deligkaris, Bullmann, Bakkum, & Frey, 2015). MEAs can be employed in *in vivo*, *ex vivo* or *in vitro* experiments, providing an invasive or non-invasive means to study the activity of neural networks. *In vitro* MEA experiments are commonly used to investigate the behavior of cultured neurons, allowing researchers to monitor the growing and the activity of large populations of cells over time. MEAs technology is capable of recording and also stimulating back the biological neurons.

- *Electroencephalography (EEG)* : it is a non-invasive technique for *in vivo* experiments, to directly record the brain's electrical activity through electrodes placed on the subject's scalp (Niedermeyer & da Silva, 2005). EEG does not capture action potentials, instead, it records the cumulative activity of hundreds of thousands or millions of neurons in the form of oscillatory activity. While the specific information carried by these oscillations is not entirely understood, different oscillation frequencies have been linked to various behavioral states (Buzsaki & Draguhn, 2004).

- *Electrocorticography (ECoG)* : it is similar to EEG for recording cumulative activity of neurons, but differs by its invasiveness level as it involves electrode arrays being implanted

beneath the scalp. Compared to EEG, it allows to better identify the source of the activity and to record higher frequency electrical activity (Crone, Sinai, & Korzeniewska, 2006).

- *Fluorescence calcium imaging* : it is a technique that employs fluorescent indicators to visualize changes in intracellular calcium levels, which are associated with neural activity (Stosiek, Garaschuk, Holthoff, & Konnerth, 2003). This method can be used both *in vitro* and *in vivo*, offering a high-resolution view of neural activity at the cellular level.

- *Functional Magnetic Resonance Imaging (fMRI)* is a non-invasive imaging technique that measures changes in blood oxygenation levels in the brain, indirectly reflecting neural activity (Ogawa, Lee, Kay, & Tank, 1990). fMRI is primarily used *in vivo* and provides information about the brain's functional organization at a macroscopic scale.

With the growing interest from the industry, more technologies are being developed to improve the communication between natural neurons and external devices. For example, Neuralink (Neuralink, 2023), a company founded by Elon Musk, introduced a novel brain-machine interface (Musk et al., 2019) consisting of arrays of small and flexible electrode "threads" (3072 electrodes per array distributed across 96 threads) that can be inserted directly into the brain with micron precision using a neurosurgical robot. This new technology aims to have precise and minimally invasive implantation, to increase electrodes count, to improve biocompatibility and durability, to simplify the implementation procedure, etc.

In this thesis, we had the opportunity to work with MEAs technology for recording neuronal behavior at a small scale, and the experiment will be detailed in chapter 6.

SNN to process biological data

After the biological recording, there are multiple processes that occur which include amplification, denoising (to remove unwanted noise), spike sorting (to identify and classify individual neuron spikes), and more, in order to obtain filtered biological spikes. However, we will only focus here on the neuromorphic system that simulates SNN models and processes the obtained biological spikes. Neuromorphic systems capable of running SNNs represent the next-generation of neuroprosthetic devices, offering advantages such as energy efficiency, capability to perform real-time data processing and the ability to mimic neurobiological computation for an improved synergy between the technological and biological counterparts (Chiappalone et al., 2022).

The pioneer work was provided in (Le Masson, Le Masson, & Moulins, 1995) with the first time interaction between single artificial and living neurons. After this first work, only small neural networks like central pattern generators (see chapter 6) have been used to study locomotion (Joucla et al., 2016; Jung et al., 2001; Sorensen, DeWeerth, Cymbalyuk, & Calabrese, 2004). Then larger and real-time SNNs have been interfaced with biological neurons (Bonifazi et al., 2013; Keren, Partzsch, Marom, & Mayr, 2019; Serb et al., 2020) for closed-loop applications.

In (Ambroise et al., 2017), they used SNNs implemented in a neuromorphic board, based on IZH neuron model to communicate in an open- and closed-loop experiments with *in vitro* biological neural network. The main result of this study is the successful modification of biological dynamics during neurobiohybrid experiments using biomimetic SNN. A similar application to synchronize biological and artificial neural networks was conducted using optogenetic stimulation controlled by SNN dynamics (Mosbacher et al., 2020).

An innovative neuroprosthesis has been designed in the framework of the European Project "BrainBow" (Bonifazi et al., 2013). The project's main results are reported in (Buccelli et al., 2019), where the authors proposed a neuromorphic system that mimics the behavior of biological neurons and synapses, to restore communication in damaged neuronal networks or replace one entire neuronal sub-network. They demonstrated that by successfully stimulating and recording neuronal activity using MEAs, where SNNs process the biological data and control the stimulation parameters. The SNN was designed with the IZH neuron model, the STP model, AMPA and GABA-ergic models, noise model and axonal delay (Ambroise et al., 2017).

2.4.3 Some limitations and challenges

Although this domain is quite interesting and allows researchers to directly communicate with real neurons to understand biological functions and mechanisms, various challenges and limitations must be addressed. It concerns the interface between artificial and biological neurons, and also the experimental setups and technologies involved. There are not all the focus of this work, but it is important to be aware of how difficult is the domain.

In vitro experiments, neuron culturing requires specialists to ensure sterile conditions, controlled environment, culture substrate, etc., to obtain healthy cultures to test on. It can be complex to maintain the viability and functionality of neurons over time, since they are sensitive to changes in their environment such as temperature, pH and nutrient availability. Variability between cultures can arise due to differences in their environment, therefore it impacts the reproductibility of experiments (Buccelli et al., 2019). Finally, neuron cultures provide a simplified model system for studying neuronal function and behavior, as they don't fully exhibit the complexity of neural circuits found *in vivo*.

Neuromorphic systems require computational models that can mimic the behavior of biological neurons and synapses. These models must be able to process information in real-time with low power consumption. Although various models exist, the parametrization of SNNs often requires fine-tuning to effectively mimic the behavior of biological neural networks. Selecting appropriate models and adjusting parameters to accurately represent neural dynamics can be a complex and time-consuming task, and it depends on the application being studied. In (Buccelli et al., 2019), the authors propose a method to create a library of SNNs based on the bursting activity of the network, *i.e.* the frequency at which neurons in the network fire all together.

For future integration with biological cells, the development of neural interfaces that are biocompatible, stable and capable of long-term use is a critical challenge. According to the review article by (Kozai et al., 2012) the long-term stability of neural interfaces is affected by several factors, including the mechanical mismatch between the device and the brain tissue, the inflammatory response to the device, and the formation of glial scars around the device.

2.5 Conclusion

In conclusion, in this chapter, we discussed the biological basics of neurons and how they communicate together. Then we talked about spiking neuron models that reproduce the behavior of real neurons in artificial systems, described by mathematical equations. We also explored various neural properties, such as AMPA and GABA dynamics, plasticity, axonal delay and biological noise, all observable in biological neural networks, and we also discussed on the implementation of computational models that will be able to make artificial systems closer to biological behavior.

We examined different ways to simulate SNNs, focusing on software and hardware approaches. This led us to the main topic of the chapter, which is the connection between artificial neurons and biological neurons through neurobiohybridization. We discussed the goals and applications of this field, as well as the various technologies and methods used to develop these systems.

Despite actual limitations and challenges, neurobiohybrid systems have great potential. They can help us learn more about the brain and its organization, which can lead to better understanding and treatment of brain-related disorders, as well as the development of advanced brain-computer interfaces and neuromorphic computing technologies. As research in this area continues to grow, we can expect exciting progress that will further connect artificial and biological systems, giving us a better understanding of the brain.

CHAPTER 3

Synchronous approach

In this chapter, we introduce the synchronous approach, the paradigm upon which synchronous languages are based, and which is the base of this work. We discuss the origins and objectives of this approach, providing a brief overview of several well-known synchronous languages.

Our primary focus is on Light Esterel, the language that we have chosen to use throughout this thesis. We detail its compilation environment and associated tools, along with previous works that have utilized this language.

Additionally, we discuss model checking, an important tool in the domain of the synchronous approach, which we aim to explore within the context of the neurobiohybridization application.

3.1 Paradigm

3.1.1 Real-time reactive system

"Reactive" systems refer to systems that interact with an environment and are designed to continuously adapt and respond to possible changing conditions while ensuring their intended functions effectively and efficiently. The term "interact" refers to the system's ability to react to events and changes in its environment, including changes it brings itself through its own reactions, creating a possible feedback loop, as shown in figure 3.1. They differ from *transformational systems*, which perform a specific set of computations or transformations on the inputs and produce outputs without interacting with the environment. For example, filters are transformational systems, there is no notion of events, they only operate on input data.

"Real-time" systems refer to systems that have predictable response times as illustrated in figure 3.1. Especially, these systems are subject to timing constraints, so they must guarantee a reaction time with respect to deadlines. They differ from *interactive systems* which respond to environment requests at their own speed, instead of the speed required by the environment *e.g.* web browser. Respecting the time constraints is essential, particularly in critical systems. For example, a flight control system must be able to process the user input (from the pilot) and calculate flight commands in less than a few milliseconds to ensure the stability of the aircraft. Failure to meet the deadlines can result in catastrophic consequences *e.g.* involving human lives, huge amount of money, etc.

Real-time reactive systems (Harel & Pnueli, 1985) combine the two previous definitions, they process and respond effectively and efficiently to events from the environment, at the speed determined by that environment. There are 2 types of event : "external" events *e.g.* input from sensors or the user; or "internal" events *e.g.* interruptions between processes within the system during scheduled tasks. These systems are commonly found in a variety of application domains : electrical equipment, medical devices, automotive industry, aviation, military system, etc.

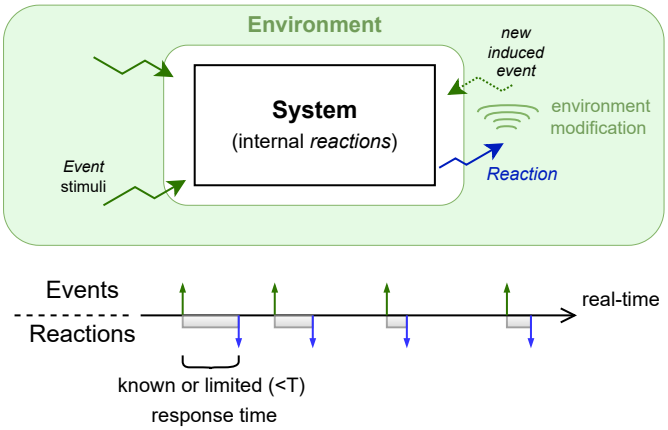


Figure 3.1: *Principle of a real-time reactive system. The system's time reaction to input event is known or bounded. Input events trigger a sequence of internal reactions, that result in an output reaction or output events. These output events may then modify the environment, leading to the production of new possible events the system has to react to.*

Furthermore, according to (Gaffé, 1996), a real-time reactive system is characterized by :

- *Strong timing constraints* : the system has to meet specific deadlines for processing events and responding to them. In some cases with "soft" timing constraints, the non-respect of the deadline is not catastrophic, but it still needs to be detected and addressed.
- *Determinism and safety* : the system produces consistent and predictable results - the same sequence of inputs gives the same sequence of outputs. When facing (external) changing conditions or unknown inputs, the system is still predictable. Deterministic systems are generally easier to understand, analyze, and diagnose. It is important to preserve determinism throughout the design to the implementation process.
- *Reliability* : the system should perform its intended functions consistently and correctly, even when facing (internal) failures or errors.
- *Robustness* : the system should perform correctly or switch to a safe and degraded mode of operation to complete essential functions, when facing (internal) failures or errors.
- *Parallelism and hierarchy* : most systems are composed of a set of components (processes) that evolve in parallel and communicate between each other, to produce the system's output. Moreover, processes can be included in larger processes where interruptions may occur. It is called hierarchical composition, resulting in concurrency that must be ensured with respect to the deterministic behavior.

The design and implementation of real-time reactive systems can be challenging and critical, because the model needs to meet specific performance requirements such as correctness (the model's behavior is correct) and efficiency (the model performs at optimal speed). Traditional programming methods and approaches can often result in errors, and debugging can be difficult and time-consuming, particularly when time development is typically tight.

3.1.2 Some programming approaches and their limits

Mostly, the real world environment is completely asynchronous. It means that it is populated with unpredictable signals (or input events), *i.e.* that do not occur at the same exact time or with the same exact period of time. These variations in timing can be caused by delays, drifts, and other timing-related issues, which basically make it challenging to design and implement reliable systems that must operate in the real world.

Intuitively, to model or to implement such systems interacting with a dynamic environment, the *asynchronous approach* would be ideal, since processes in asynchronous programs are triggered by the occurrence, at any time, of the input events. In general in this approach, they use event handlers to manage the inputs and the interactions between parts of the system model. So when an event occurs, it is placed in a queue, and the event handler is triggered to process the event. However, to include delays or the detection of the absence of a signal, it requires complex logic and additional programming effort to ensure the system's behavior. For example, waiting for a specific period of time before taking an action, or adding timestamp to the signals to track their age, etc. Moreover, although concurrent processes can be implemented thanks to synchronisation mechanisms such as semaphores, it can lead to non-deterministic behavior. Concurrent processes in asynchronous model are independent and can compete for calculation resources, therefore it is difficult to predict when the processes will execute, how long it will take for a specific event to be processed, etc.

So, in general the time management is complex due to the nature of asynchronism, hence more challenging to express and ensure timing constraints.

Another approach is *transition systems*. Transition systems are mathematical models ideal to describe systems whose behavior changes over time. It is mainly based on Finite States Machines (FSMs) where transitions are triggered by the input events. Although this approach results in deterministic models, concurrency can lead to complex and huge FSM models since all possibilities have to be explicitly expressed. For example, the combination of two concurrent FSMs, with n states and m states respectively, will result in a new a global FSM with $n \times m$ states, with an equivalent behavior.

So programming real-time reactive systems requires ensuring two essential traits : determinism and concurrency, while being influenced by an asynchronous environment. Although there are theoretically efficient solutions, they add significant complexity to the system and in its design, therefore the risks of bugs increase. To address these points, specific languages have been developed for designing real-time reactive systems, based on the *synchronous approach* that was developed in the 80's (Harel & Pnueli, 1985; Berry, 1989). In this thesis, we used the synchronous approach that we will present in the next section.

3.1.3 Synchronous approach paradigm

The synchronous approach aims to make the programmer's task easier (Halbwachs, 2013), by providing specific frameworks or programming languages whose "primitives" and "semantics" have been developed to express the system's behavior in a simple, straightforward and formal way. Primitives refer to the instructions or operators available in the language, and semantics refer to the meaning or interpretation of these primitives. The synchronous approach relies on the *synchronous hypothesis* : the model executes at discrete times and is considered to be infinitely faster than the environment.

Atomic reactions

The evolution of a reactive system is a sequence of reactions triggered by events from the environment. In the synchronous world, the main idea is to consider that "reactions are instantaneous" or they "take no time". We can imagine the model being run by a machine infinitely fast, so that by observing from the outside, input events and the triggered output events are simultaneous, as illustrated in figure 3.2(a). By *atomic reaction*, the approach refers to the evolution of the system based on a screenshot of the input events, and the set of reactions that are triggered by the occurrence of the inputs are instantaneous : reactions are logical tests, computations, events broadcasting, etc. So the model evolves only at discrete times, also called *logical instants*. The global evolution of the model through time is then represented by an ordered sequence of *logical instants*, and apart from these instants, nothing happens either in the model or in its environment.

In the real world, this vision can only be applied to systems that respect the following properties : (1) the system produces the same sequence of output events given the same sequence of input events and (2) the system's time reaction must be shorter that the mean period the environment generates events, as illustrated in figure 3.2(b).

There are numerous advantages to the "instantaneous" idea. One advantage is that temporal semantic is simplified, it leads to clear temporal constructs and easier time reasoning. An other advantage is that because a system evolves in a sequence of discrete steps, and nothing happens

between two successive steps, the design, debugging or verification of the model is easier since the model's behavior is predictable and precise at each instant. It guarantees deterministic behaviors even with concurrent processes, since any changes made by one process will be immediately visible to the other processes.

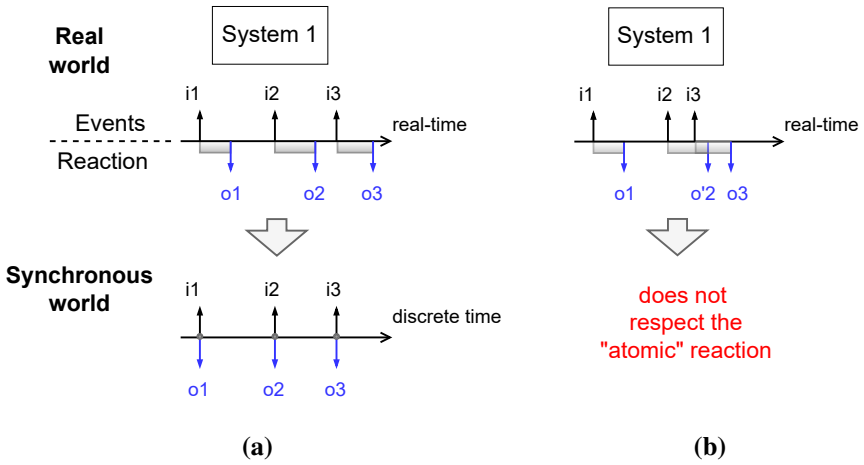


Figure 3.2: *Illustration of atomic reactions in synchronous modeling. (a) In the real world, the system 1 produces a sequence of outputs given a sequence of inputs, with each reaction taking a certain amount of time. In the synchronous world, all reactions takes no time. (b) However, the system must respect the synchronous criteria where the environment can not affect the system during reactions. Figure extracted from Daniel Gaffé's course entitled "the Synchronous language" in Master of Electronics.*

Signals and events

Signals are the mean for the model to communicate with the environment or its components to communicate between each other : the system receives "input" signals from and emits "output" signals to the environment, and "local" signals are for communications between the system's components. Note that local signals are not accessible from outside the system. At each logical instant, a signal must be consistent, meaning that its status and/or value are the same for all read operations. The signal information lasts for the current instant it is emitted. An event is defined as one or multiple *signals* occurring at the same logical instant. An event triggers the system reactions.

In some synchronous languages, such as Esterel which will be described in the next section, a signal conveys two pieces of information at each instant : a *predicate* which refers to the presence status of the signal (either *present* or *absent*), and its *value* of a given type, *e.g.* integer. The predicate allows to handle explicit conditional execution based on the presence or absence of signals. Particular cases are : *pure* signals do not have values and can only be present or absent; *sensor* signals have only values and no predicate. However, in some languages, such as Lustre described in the next section, the presence or absence notion do not exist. Lustre signals are typed, convey only their value and can be undefined at any logical instant. A Lustre signal is "present" only if it is defined.

Causality

By considering that reactions are instantaneous in synchronous models, "causality" is used to track how every event has been triggered in the system. Causality means that there is a sequence of reactions that had led to the emission of event. However, instantaneous reactions can result in temporal paradoxes when programming, and it must be detected during compilation. Generally, temporal paradoxes appear when the internal signals of a system depend on other internal signals or themselves. This is the case of *causality cycle* problem, for example "emit A if and only if A is not emitted". Although it makes sense that A is emitted if it is not present, it would be rejected in synchronous approach. Indeed, a synchronous system is executed on a snapshots of the status/values of the inputs and the internal states at each logical instant. Therefore if the signal A is absent, but the system emits instantaneously the presence of A, then the paradox is that A would have two contradictory status "present" and "absent" within the same logical instant. This also illustrates a synchronous rule which is "a signal can have one and only one value at each logical instant".

Multiform time

In synchronous approach, instantaneous reactions mean that time is no longer related to program execution as in asynchronous approach, but to the occurrence of the input stream. The system only reacts to the occurrence of external events, therefore time is considered as an input event. In other words, to manipulate time, *e.g.* for specifying time constraints, a specific input signal should be produced by a clock. Not only, it also means that any signal can be manipulated as a "time unit", *i.e.* they have their own, which corresponds to the notion of *multiform time*. An example to illustrate this principle is :

"The train must stop within `10 seconds`"
and
"The train must stop within `100 meters`"

The two sentences above specify two constraints based on two different basis of "time", `seconds` and `meters`. In one case, the train must stop after the 10^{th} second, while the other is after the 100^{th} meter. Constraints are therefore easier to manipulate within the synchronous approach.

How to connect the synchronous program to an asynchronous environment : the execution machine

A synchronous program receives synchronous input events, and returns synchronous output events. However, the environment the system is interacting with is asynchronous, for example, different types of sensors can send different data to the system, independently. Therefore, to make the interaction between a synchronous model and an asynchronous environment possible, the *execution machine* is an important architecture that converts asynchronous events into synchronous events (André, Marmorat, & Paris, 1991; Boufaied, 1998; Sarray, 2019). Moreover, the execution machine has to ensure that the real implementation is a good approximation of the ideal synchronous program, *e.g.* by guaranteeing atomic reactions *i.e.* non-overlapping executions. It has 3 main roles : (1) to convert asynchronous input events into synchronous input events while the synchronous program is being executed, (2) to activate the synchronous program on the new created set of input events and (3) to present the generated synchronous output events to the environment. The figure 3.3 illustrates the principle of an execution machine.

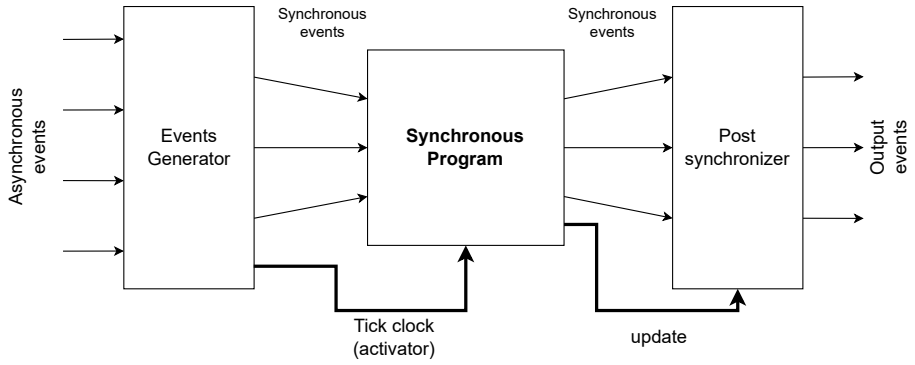


Figure 3.3: *Global architecture of an execution machine.*

The input event processing or *events generator* is responsible for receiving input events from the environment (*e.g.* from different types of sensors) and preparing them for presentation to the synchronous program. Depending on the language and system being modeled, this component may include buffering or sampling of input events to ensure that they are correctly synchronized with the clock-based model of the system. Depending on the event processing strategy, it defines which events should be considered as simultaneous. The *synchronous program* is the core component of the execution machine, it receives the input events from the events generator and performs computation based on the inputs and its internal state, and produces the output events. The synchronous program receives a *tick clock* or activator signal to start the reactions. The tick clock depends on the strategy chosen for the language and the system being modeled. Finally, the output event processing or *post synchronizer* is responsible for updating the output

Then, a *tick* signal (activator) is sent to the synchronous program to execute its reaction(s). While the program is running, the events generator is listening to the incoming events, but the synchronous program inputs are not modified. Finally, at the end of the reactions, the output events are updated by the *post synchronizer* which receives the signal update from the program.

3.1.4 Synchronous languages or models overview

Synchronous programming languages are "model-based" languages, they emphasize the use of models to represent and reason about system behavior. The languages provide operators, tools and techniques for designing and working with these models. There are two families of synchronous languages :

- *Declarative languages* : or "data flow oriented" languages are mainly derived from signal processing techniques that represent systems in an equation-based form. In other words, the programmer gives a description of the constraints and goals without specifying the steps to get there. The most well-known languages are *Lustre* (Halbwachs, Caspi, Raymond, & Pilaud, 1991) and *Signal* (Gautier, Le Guernic, & Besnard, 1987). Each variable is a data-flow expressed as an equation and coupled with a clock, therefore a variable has one unique value at each logical instant when it is defined.

- *Imperative languages* : or "control systems oriented" languages provide operators to describe explicit and step-by-step (or concurrent) instructions on how tasks should be performed with respect to eventual constraints. The most well-known languages are *Esterel* (Berry &

Gonthier, 1992) a textual imperative language, and *SyncCharts* (André, 1996) its graphical formalism.

Synchronous languages are primarily used for specification purposes. They provide rigorous *semantics* that enable direct compilation of the specifications, and eliminate any ambiguity in behavioral interpretation. Semantics in programming languages determine how the language's vocabulary (symbols, instructions, and operators) and syntax (instructions combined) should be interpreted and executed. Semantics are defined using a definition or equivalence in another formalism. For example, *equational semantics* define the meaning or equivalence of the language's vocabulary using mathematical (Boolean) equations.

Synchronous languages ensure that simulation, compilation, and verification results are consistent with each other based on the given specifications. This means that simulation results can be reproduced, generated codes are accurate and efficient in relation to the original model, and verification results on the model can be applied to the generated codes. Generally, the compilation of a synchronous program transforms the program specifications into finite state machines (FSMs) expressed in the form of (Boolean) equation systems.

In the following, we provide an overview of some synchronous languages considered as the most known, historical and some new ones. We especially develop the Esterel and Lustre languages since the first one has inspired the language we chose to work with, and we used the second one for verification experiments.

Esterel

Esterel (Berry, Moisan, & Rigault, 1983) is one the first synchronous languages, it has been introduced and developed in the 80's, through the collaboration of two organizations : the *École Nationale Supérieure des Mines de Paris* (Center of Applied Mathematics (CMA)) and National Institute for Research in Computer Science and Control (INRIA). It is an imperative and modular synchronous language (Berry, 1999), which main objective is to provide high-level constructs or instructions to specify the system's behavior in a clear and concise manner. An Esterel program is called a "module" and the basic structure looks like the following :

```
module <moduleName> :
   input <in1, in2, ...>;
   output <out1, out2, ...>;

   <set of imperative instructions>

end module
```

The structure of an Esterel program consists of the module's name, the input/output interface declaration, the local signals declaration and definition, and the reactions. We refer to as "reactions" the set of instructions that defines the module's behavior. For that Esterel supports usual and specific operators (or instructions), each with well-defined formal semantic. Here are some few operators :

- signal operators : `emit, present, absent`

- control operators : `if... then... else... end, loop... end loop, abort... when,` *etc.*
- arithmetic operators : `:=, +, *,` *etc.*
- logic operators : `or, and, xor,` *etc.*
- sequence and parallel operators : `;, ||`
- temporal operators : `await, every, after,` *etc.*

Signal operators are for manipulating signals and their status : `emit` sets the signal status to *present*, and `present` checks the presence status of a signal. Control operators are conditional branching instructions or loop instructions. The sequence operator "`;`", *e.g.* "*statement1*`;`*statement2*", is to execute the *statement1* until it is finished, then to execute *statement2*, but both belong to the same logical instant. The parallel operator "`||`", *e.g.* "*statement1*`||`*statement2*" is to start and to execute both statements at the same time. The language allows to specify parallelism, and it guarantees the determinism of such constructions since the semantics are well-defined. We just give few examples of instructions in Esterel, but it supports other types of instructions such as *preemption* used to abort sequence of statements when an event occurs.

During the compilation of an Esterel program, first the syntax is checked to ensure that it is correct and conform to the language rules. Then, the program is transformed into a basic automaton model, involving multiple steps. For example, it requires the flattening of the hierarchy : Esterel programs can have multiple levels of hierarchy, with modules calling other modules, or modules in parallel. Each module is transformed into one state machine, and all the state machines are connected together through synchronization signals to form a larger synchronous automaton that represent the entire program. This process simplifies the program and makes it easier to reason and analyze it.

Lustre

The Lustre Language (Halbwachs et al., 1991) is a declarative language, and has been developed in 1984 by Nicolas Halbwachs and Paul Caspi at the laboratory VERIMAG in Grenoble (France). A Lustre program is called a "node" which structure is as follow :

```
node <NodeName> (<in1>: <typeOfIn1>; <in2>: ...)
        returns (<out1>: <typeOfOut1>; <out2>: ...);
   var <var1Name>: <typeOfVar1>;
   var ...;

   let
      <set of equations>;
   tel
```

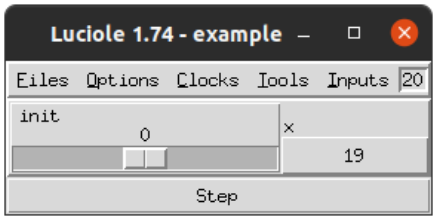
A Lustre program begins with the keyword "node"; followed by its name, then the list of its input/output names and types. Local variables can be declared with their names and types with the keyword "var". Finally the body of the program is written between the keywords "let" and "tel".

In Lustre, every variable or output is a stream, meaning an infinite sequence of values of a given type. As a declarative language, the variables and outputs are expressed as a set of equations which depends on the inputs and the current internal state. Note that the order of the equations is not important. Lustre applies the rewriting rule thanks to the equal operator. Lustre supports :

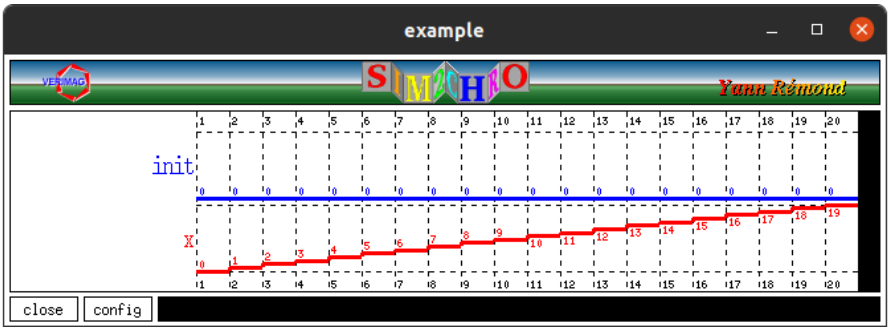
- basic types : `bool, int, real`
- arithmetic operators : `+, -, *, /, mod`
- logical operators : `or, and, not`
- comparison operators : `=, <, >, <=, >=`
- some control operators : `if... then... else...`

There are no loop operators or recursive functions, however Lustre allows for the use of recursive temporal operators such as `pre()` and `->`. The `pre(x)` operator is for accessing the value of a signal `x` at the previous logical instant when it was defined, and the `->` is to define the signal values for all the following instants after its first value at the first instant. For example, a signal `x` of type `int`, defined by `x = 0 -> pre(x)+1`, will take the value `0` at the first instant, then for the following instants `0+1`, then `1+1`, then `2+1`, etc. Where it will be defined. There are more operators (*e.g.* operators for arrays) but we do not give an exhaustive listing since we used only a subset. The previous list is the list of operators we used in Lustre programs in section 4.2 for verification experiments.

Lustre compilation first consists in checking the absence of recursive call of nodes, the absence of signal declarations without initialisation and the absence of definition with cycles. Then the compilation generates a sequential code. The code is encapsulated in an infinite loop, and executed at each logical instant (or clock). The loop is triggered, for example when there are new inputs, or outputs need to be updated. Generally, it reads the inputs, computes the outputs and memorizes the values of variables required for the next instant. The equations order in the Lustre program is not important, however the compilation generates a sequential code where the order of the instructions are important. To order the instructions, a topological ordering is made on the dependence graph of the variables (Halbwachs, 2013). After the compilation, a Lustre code can be simulated with the *Luciole* tool that provides a graphical interface to control the inputs of a node and visualise the outputs accordingly. And *Sim2chro*, provided within Luciole, allows to draw the chronogram of the simulation (see figure 3.4). Next, a lustre program can be translated into C codes thanks to *ec2c* or *drac* tools.



(a) *Luciole simulation*



(b) *Displayed chronogram by Sim2chro*

Figure 3.4: *Simulation of a Lustre node. In this example, the output* `x` *is defined by the equation* `x = init -> pre(x)+1`*, where* `init` *is an input of type "int" to initialize the first value of* `x`*.*

Lustre has also been industrialized by the Verilog campany, its industrial name becomes SCADE (Safety Critical Application Development Environment). SCADE has been used for the design of critical systems in the domain of embedded systems such as aviation.

Signal

Signal (Benveniste, Le Guernic, & Jacquemot, 1991) is a synchronous programming language developed at IRISA Rennes (France) by a team led by Albert Benveniste and Paul le Guernic. It was industrialized by the company TNI through the Sildex 1 environment. Like Lustre, Signal is a declarative language, in which a program expresses the relations between timed sequences of values. The order of equations is not relevant, and signals are defined by equations that specify their properties. Signal manipulates clock trees and does not have a global clock, unlike Lustre.

Signal is modular, and its unit of program is the "process". Signal is a relational language, and every process defines a relationship between its input and output flows. Signal has five basic operators (in addition to arithmetic and Boolean operators) to express the relationships that define the output flows: delay, when, default, |, etc. The Signal language is simple to use for simple data processing but can quickly become complex when programming complex sequences. To prove the static and dynamic properties of programs written in Signal, the model verification tool Sigali was designed.

Argos

Argos (Maraninchi & Rémond, 2001) is a graphical language based on a subset of Statecharts (Harel, 1987) and belongs to the lineage of synchronous languages like Esterel and Lustre. Argos uses operators to create Mealy machines. The language's semantics are synchronous, like Esterel.

Argos has given rise to other graphical formalisms, such as "mode automata" (Maraninchi & Rémond, 2003), a combination of Argos and Lustre to create a powerful synchronous language that allows for the specification of applications combining automata and data flows. The "mode automata" model has been implemented in the latest version of Scade.

SyncCharts

SyncCharts (André, 1996) are a graphical formalism, strongly inspired by StateCharts and Argos. Like StateCharts, SyncCharts use states, initial and final states, transitions, signals, and events to model reactive systems, ensuring hierarchy, modularity, and parallelism. However, SyncCharts differ from StateCharts by introducing synchronous operators, ensuring no ambiguous interpretation and no hidden behavior, guaranteeing the determinism of the model. SyncCharts handle preemption in a more rational way by introducing the notion of abortion and suspension, using a limited set of powerful graphical primitives.

In addition, SyncCharts integrate the simultaneity of events and instantaneous signal broadcasting during communication. The formalism is based on a process calculus that allows for systematic translation into an Esterel program, providing users with the benefits of the software environment developed for synchronous programming. SyncCharts have been used to model critical reactive systems such as automotive control systems. However, to use SyncCharts, users need to have a basic understanding of synchronous programming to understand the notion of instant and transition.

Quartz

Quartz (Schneider, 2009) is an imperative synchronous programming language, developed at the University of Kaiserslautern in Germany. It is designed for the modeling, verification, and implementation of reactive systems. Quartz shares many similarities with Esterel, but includes instructions for asynchronous parallel execution of tasks, enabling explicit implementation of indeterministic systems. Quartz has also been extended with instructions that have a delayed effect on the next instant, enabling the description of both software systems (sequential algorithms) and hardware circuits. Additionally, Quartz introduces the concepts of concurrency and preemption, and uses a four-valued algebra to verify program causality. Quartz also allows for the handling of analog data to accommodate hybrid systems. Programs written in Quartz can be converted to equivalent symbolic transition relations for formal verification, and the language's determinism is important for accurate simulation of observed behavior. In contrast of Esterel and other synchronous languages, variables in Quartz are always present and defined, they only take a single value at each instant.

Zélus

Zélus (Bourke & Pouzet, 2013) is a language to model hybrid systems, which means systems that combine "discrete" behavior and "continuous" behavior. For example, it allows to describe a discrete controller (modeled by FSMs) and its continuous physical environment (modeled by Ordinary Differential Equations (ODEs)). Inspired from Lustre, it supports the combinations of data flow descriptions, ODEs models and hierarchical automata. Synchronous languages in general abstract the environment as a source of discrete sampled inputs, and they respond with discrete outputs in return. A synchronous rule is that nothing can be inferred between two discrete instants. Therefore, they are suitable for the design of discrete controllers. However, a model expressed with

ODEs or Differential Algebraic Equations (DAEs) continues to evolve between two instants, and a variable-step numeric solver is necessary to approximate efficiently and faithfully the solution over time. The compilation of a Zélus program is by source-to-source translation into synchronous code which is then compiled to sequential code and paired with an off-the-shelf numeric solver. Zélus programs can be translated into C codes, *e.g.* for embedded applications.

3.2 The choosen language : *Light Esterel*

Light Esterel (Ressouche, Gaffé, & Roy, 2008) is a synchronous language developed by research laboratories in informatics, electronics and mathematics (INRIA, LEAT and CMA). Its developers are Annie Ressouche, Daniel Gaffé and Valérie Roy. The language is derived from Esterel V5 and is designed for the specification and formal verification of real-time reactive systems. In this thesis, we have chosen to use *Light Esterel* for 3 reasons. First, one of the developers, Daniel Gaffé, is the co-supervisor of this thesis, therefore it is convenient for the understanding of and the programming with the language. Second, as not a lot of studies have been conducted between the synchronous approach and artificial neural networks (see chapter 4), this gives us the opportunity to have access to the language compilers, and to rapidly modify, improve and adapt it for our application. Third, the choice of *Light Esterel* does not limit the use of other languages, as it is possible to generate other synchronous languages, such as Lustre, with the *Light Esterel* environment tool directly from *Light Esterel* programs.

3.2.1 *Light Esterel* program structure

A *Light Esterel* program is referred as a *module*. A module has the following structure :

```
<Data declarations>
module <moduleName>:
  <Interface declaration>
  <Module body : automaton OR mealy machine OR statements>
end
```

Data declarations

The data declaration area is where to declare *types*, *methods* (or functions) and *constants* that will be used in the module's description (interface and body). The data declaration structure is as the following :

```
<Data declarations> structure :
data:
Type: <type1>, <type2>, ...; -- eg: int, real, ...
Method: <returnedType1> <methodName1>(argName11, argName12, ...);
        <returnedType2> <methodName2>(argName21, argName22, ...);
        ...;
Constant: <type1> <constantName1>:=<value1>;
          <type2> <constantName2>:=<value2>;
          ...;
end data
```

In contrast to other synchronous languages, there are no predefined types. User must define specific types or methods or already defined library (see examples in (Gaffé, 2023a)) of the target codes to be generated.

Interface declaration

The interface declaration of a module contains the *input signals* it listens to and the *output signals* it emits. Additionally, it can also contain *local signals*, *pre signals* and the list of already compiled modules that will be used within the current module. Signals can be either pure or typed. In *Light Esterel*, a *pre signal* is used to access (store) the previous information (predicate or value) of a signal. The pre signal declaration contains its name, the signal of interest and its initial constant value at the very first instant. With the `run` operator, *Light Esterel* allows to call independent modules with their own defined interface to form a modular system (a system composed of communicating sub-modules). The interface declaration structure is shown in the following :

```
<Interface declaration> structure :
module <moduleName>:
Input: <signalInName1>,   -- pure signal
       <signalInName2> : <type2>, -- typed signal
       ...;
Output: <signalOutName1> [:<type1>],
        <signalOutName2> [:<type2>],
        ...;
Local:  <signalLocName1> [:<type1>],
        <signalLocName2> [:<type2>],
        ...;
Pre: <signalPreName1> : <signalName1> $ <firstInstantConstantValue1>,
     <signalPreName2> : <signalName2> $ <firstInstantConstantValue2>,
     ...;
Run: "<pathToModule1>" : <fileName1> : <moduleName1>;
     "<pathToModule2>" : <fileName2> : <moduleName2>;
     ...;
```

Module body

The body of the module is where to specify how to produce the outputs from the inputs and eventually the current state of the module. *Light Esterel* unifies different specification styles: a representation in the form of a *hierarchical automaton*, a *declarative syntax* and an *imperative syntax*. The figure 3.5 illustrates a simple application using these 3 different syntaxes. The application is to emit the outputs `O1` and `O3` at the very first instant, then to emit the output `O2` only when the input `I` is present, and finally to stay idle forever.

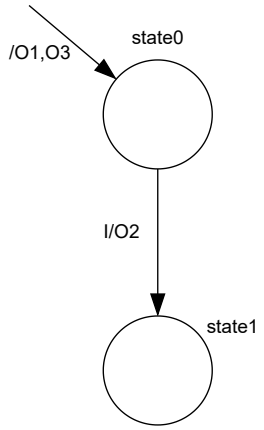
<Module body> structure :

```
module Parallel:
Input: I;
Output: O1, O2, O3;

emit O1
||
{
    wait I
    >>
    emit O2
}
||
emit O3

end
```

(a) *Imperative syntax*



(b) *hierarchical automaton (graphical or textual)*

```
module Parallel:
Input: I;
Output: O1, O2, O3;

Mealy Machine
Register:
  X0: 0: X0next,
  X1: 1: X1next;

X0next = X0 or not X1 and
    I;
X1next = 0;

O1 = not X0 and X1;
O2 = not X0 and not X1 and
     I;
O3 = not X0 and X1;

end
```

(c) *Declarative syntax*

Figure 3.5: *Different syntaxes supported in Light Esterel.*

In figure 3.5, (a) The imperative syntax, a textual language resembling to Esterel, allows to design event driven applications. *Light Esterel* supports diverse operators to manipulate signals, numeric and logical operations, logical instants, etc. Some simple operators are listed in table 3.1 and a complete syntax overview is shown in appendix A. In the given application example, (a) 3 blocs are in parallel separated by the operator `||` : `O1` and `O3` are emitted in parallel at the first instant, while `O2` is waiting for the presence of `I` to be emitted. Note that the operator » is a sequential operator, it specifies the execution order of instructions within the same logical instant. (b) Hierarchical automaton can be described textually or graphically like StateChart. In *Light Esterel*, the graphical format can be implemented using the *Galaxy* tool (Gaffé, 2023b; Gaffé & Ressouche, 2008) developed with the fltk library, the tool is described in section 3.2.2. The application example can be modeled into 2 states, where inputs and outputs are placed on the transition branches. (c) The declarative syntax allows to describe data flow systems with equations resembling to Lustre language. The outputs `O1, O2, O3` are defined by equations that depend on the input `I` and the current state encoded by `X0, X1`.

`nothing`	Do nothing
`emit S`	Immediately set as present the status of S
`present S {P1} else {P2}`	If S is present then P1 is executed, else P2
`P1 >> P2`	P1 is executed then P2
`P1 || P2`	P1 and P2 start simultaneously, the instruction is terminated when P1 and P2 are both done
`abort P when S`	P is executing normally until S is present
`loop {P}`	P is executing and restarts when it is done
`local S {P}`	The scope of S is limited to P (*Locals became a declaration since 2013*)
`run M`	Call of the module M
`pause`	Stops until the next reaction (instant)
`wait S`	Waits for the presence of S

Table 3.1: *Some operators of Light Esterel, extracted from (Ressouche et al., 2008). More detailed operators and syntaxes are given in appendix A.*

3.2.2 Compilation environment and tools

The *Light Esterel* development chain is illustrated in figure 3.6. It shows the main compilers of *Light Esterel* programs, called "Compiler of Light Esterel Modules (CLEM)" and "Compiler of Light Esterel modules Finaliser (CLEF)". The *Light Esterel* compilers have been developed to support modular compilations (Ressouche et al., 2008), *i.e.* to describe independently the sub-modules of a system, then to compile them separately, and finally to assemble them together. This makes modification easier and enables the addition of new modules without having to compile the entire system again.

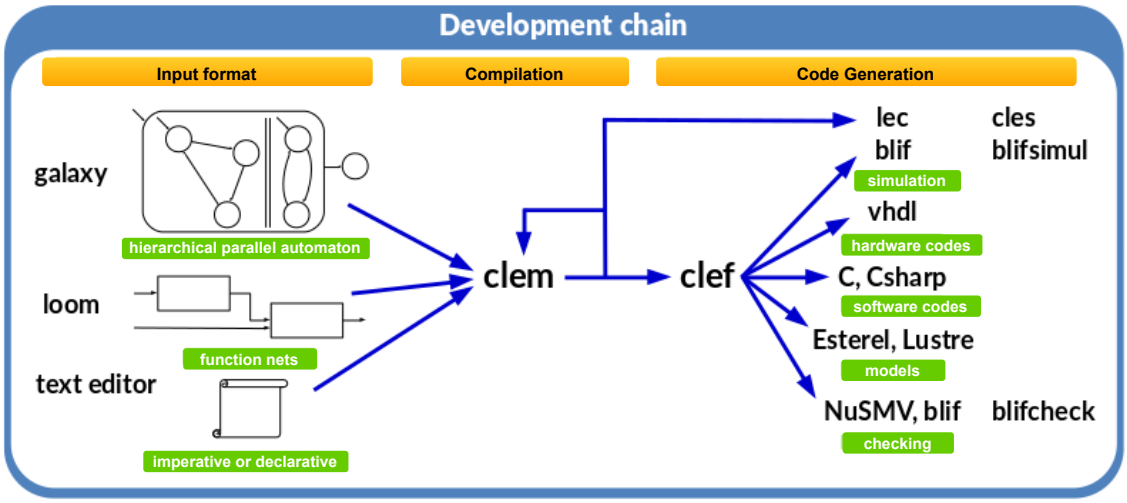


Figure 3.6: *Compilation environment of Light Esterel.*

The development chain or workflow can be decomposed into three main phases :

- the *design* : the programming or specification of the application, and also the "input" of the compiler CLEM;
- the *compilation* : the compilation of *Light Esterel* programs (contained in files with the ".le" extension) into "Light Esterel Compiled codes" (generation of a ".lec" extension file) by the CLEM compiler;
- the *code generation* : the transformation of the ".lec" codes into specific codes (software or hardware) by the CLEF compiler.

Input format : *Light Esterel* model programming

Light Esterel programs are saved into files with the extension "*.le*". A *Light Esterel* program can be manually written entirely using a text editor, and as detailed in section 3.2.1 in the "module body" section, by choosing one of the programming styles : imperative or declarative or textual automaton.

The Linker Of Object Modules (LOOM) tool (Gaffé, 2023b) is to create modular systems graphically by providing a graphical interface to connect the *Light Esterel* compiled modules. Loom then generates the corresponding "*.le*" file. The figure 3.7 illustrates the interface of Loom. It shows 2 sub-modules interconnected to create a main module with 3 inputs and 3 outputs.

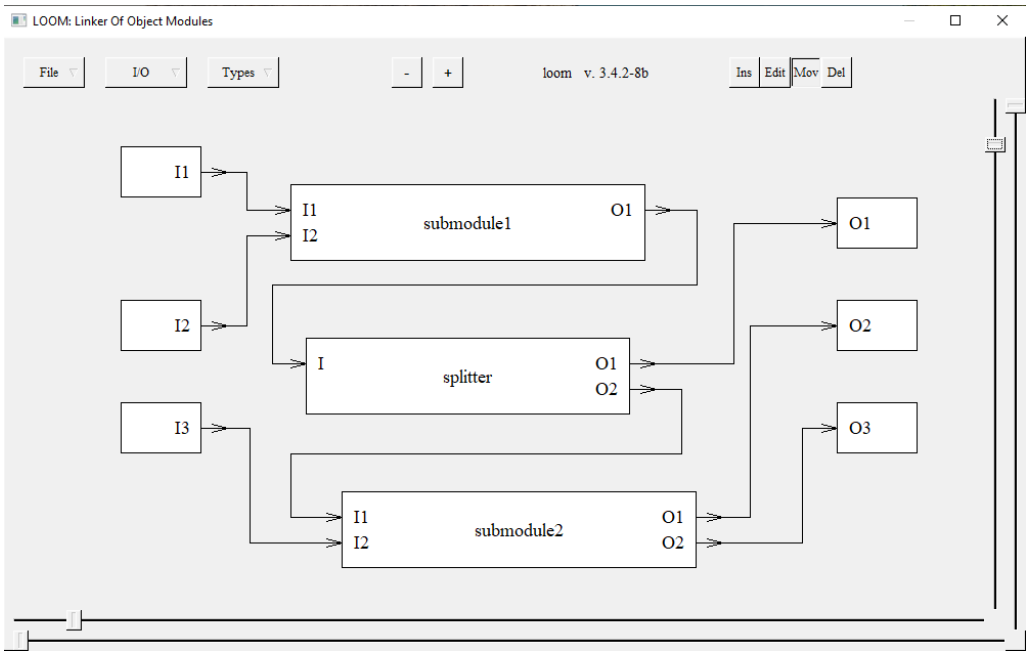


Figure 3.7: *Visualisation of the LOOM tool interface (version 3.4.2). Example of a system with 3 inputs and 3 outputs, and composed of 2 communicating sub-modules through a splitter module. The splitter role is to copy the sub-module1 output value (O1) to the sub-module2 input (I1) and to the system's output (O1).*

As previously mentioned, another tool is the *galaxy* tool. Galaxy is a graphical automaton editor, its interface is shown in figure 3.8, and it has been developed in a research project combining FSMs

and synchronous languages. Galaxy offers the possibility to choose between 4 different models to design an application : for the design of Mealy or Moore state machines, there is a *basic automaton* model; for the design of basic automaton in parallel, there is the *parallel automata* model; for the design of hierarchical automaton, there is the *Light Esterel* model; and finally, it is also possible to create SynchCharts models.

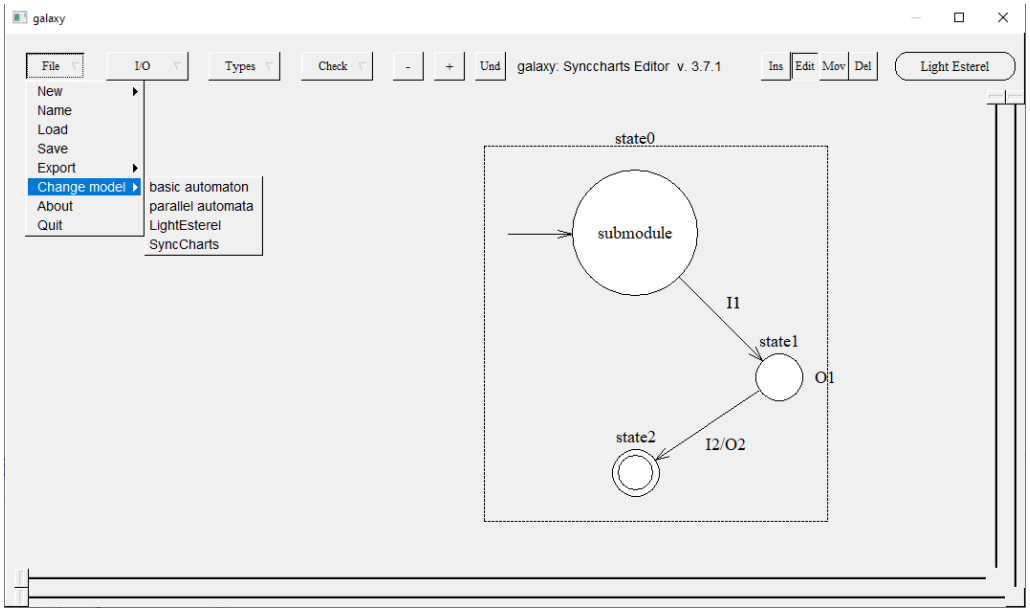


Figure 3.8: *Galaxy tool's interface (version 3.7.1). Example of a hierarchical state machine composed of 3 states. The initial state (state0) is composed of 1 sub-module which is executed when in state0.*

Compiler of Light Esterel Modules (CLEM)

The CLEM compiler (Gaffé & Ressouche, 2008; Gaffé, 2023a) relies on the constructive *equational semantic* of *Light Esterel* (Ressouche et al., 2008). The semantic of a programming language is how the vocabulary of the language (operators and statements) should be interpreted and executed. For example, the equational semantic is the meaning of the *Light Esterel* vocabulary using mathematical equations. The CLEM compilation consists in applying well-established and formal semantic rules to transform each *Light Esterel* program into a ξ equation system. *Light Esterel* uses a set $\xi = \{\bot, 0, 1, \top\}$, a 4-valued algebras inspired from (Ginsberg, 1988), to define the status of a signal and associated operators in *Light Esterel* (Gaffé & Ressouche, 2013), unlike Esterel with 3-valued algebras. When a signal is *absent* its status is 0. When it is *present* its status is 1. When it is not determined then its status is $\bot$ meaning *bottom*. And finally, $\top$ or *error* is when the signal is over-specified meaning that its status has been emitted as "absent" and "present" at the same time by different parts of the program. The use of a 4-valued algebras is to be able to characterize status conflict between subprograms (*i.e.* over-specifications), and therefore to allow separated compilations. ξ is equipped with 5 composition laws (or operations or algebras) : $\sqcup$, $\sqcap$, $\neg$, $\boxplus$ and $\boxdot$ (see figure 3.9). $\sqcup$, for *unification*, performs the unification of the knowledge concerning the signal status : it implements the parallel operator $||$. For example, if a signal status is 0 (absent) from a sub-module and 1 (present) from another parallel sub-module, then the unification is *error* since the signal status can't be both at the same time. The $\sqcap$ is the dual operator of $\sqcup$, useless for

compilation concern but important to prove distributivity properties. The $\neg$ is the negation, but gives identity for $\top$ and $\bot$. The $\boxplus$ is equivalent to the boolean logical OR. The $\boxdot$ is equivalent to the boolean logical AND.

Common

$\sqcup$	1	0	$\top$	$\bot$
1	1	$\top$	$\top$	1
0	$\top$	0	$\top$	0
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	1	0	$\top$	$\bot$

$\sqcap$	1	0	$\top$	$\bot$
1	1	$\bot$	1	$\bot$
0	$\bot$	0	0	$\bot$
$\top$	1	0	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

x	$\neg\, x$
1	0
0	1
$\top$	$\top$
$\bot$	$\bot$

Algebra5

$\boxplus$	1	0	$\top$	$\bot$
1	1	1	1	1
0	1	0	$\top$	$\bot$
$\top$	1	$\top$	$\top$	1
$\bot$	1	$\bot$	1	$\bot$

$\boxdot$	1	0	$\top$	$\bot$
1	1	0	$\top$	$\bot$
0	0	0	0	0
$\top$	$\top$	0	$\top$	0
$\bot$	$\bot$	0	0	$\bot$

Figure 3.9: *Composition laws of ξ. From five different algebras (Gaffé & Ressouche, 2013), the algebra5 was selected, as it allows to satisfy more logical properties related on distributivity, commutativity, associativity, absorption, etc. according to the Ginsberg algebra.*

```
1  module first:
2  Input: I1, I2;
3  Output: O1, O2;
4
5  loop {
6    pause
7    >>
8    {
9      present I1 {emit O1}
10     ||
11     present I2 {emit O2}
12   }
13 }
14 end
```

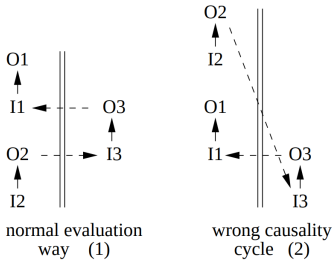
(a) *First module*

```
1  module second:
2  Input: I3;
3  Output: O3;
4
5  loop {
6    pause
7    >>
8    present I3 {emit O3}
9  }
10 end
```

(b) *Second module*

```
1  module final:
2  Input: I;
3  Output: O;
4  Local: L1, L2;
5
6  run first[L2\I1, O\O1, I\I2, L1\O2]
7  ||
8  run second [L1\I3, L2\O3]
9
10 end
```

(c) *Main module*



(d) *Evaluation order*

Figure 3.10: *Modified figure extracted from (Ressouche et al., 2008). Signals `O1`, `O2` and `O3` are independent. In the main module, choosing a total order can introduce causality cycle. With the order (1), by taking into account the renaming, the obtained system is {L1=I, L2=L1, O=L2} which result in a normal evaluation. However, with the order (2), the obtained system is {L2=L1, O=L2, L1=I} resulting in a causality cycle.*

The system specifications is transformed into an equation system, but to generate codes, to simulate or to link with other programs, it requires to find an evaluation order that is valid at each synchronous logical instant. Usually, this order is static in most existing popular synchronous languages. And static order forbids any separated compilation mechanism. For example, 2 scenarios are shown in figure 3.10 with the compilation of a module called `final`. The first order results in a correct evaluation of the equation system, while the second results in a causality cycle.

To avoid such problem, an incremental partial order has to be created. For that, each variable in the equation system is assigned with two integer variables (*CanDate, MustDate*) which indicate when the variable *can* and *must* be evaluated. In other words, the *CanDate* characterizes the earliest date a variable can be evaluated regarding the system's inputs, and the *MustDate* characterizes the date a variable must be evaluated regarding the system's outputs. By considering the dates as *levels*, the level 0 characterizes the variables that are evaluated first because they depend on any other variables. A level *n+1* characterizes the variables that require the evaluation of variables from lower levels (from *n* to 0) to be evaluated. Variables with the same level are independent, therefore they can be evaluated in any order. This method is inspired from the PERT method (Kirkpatrick & Clark, 1966). An example is given in figure 3.11 extracted from (Ressouche et al., 2008). We don't detail here the algorithms underlying the association of the levels for each variable, which is done in (Ressouche et al., 2008). Instead, we only describe basically the main steps. First, a *dependence graph* is first created from the equation system as illustrated in figure 3.11. This graph is composed of the variables as nodes, and the dependency relation between the variables are characterised by arrows, such that $x' \rightarrow x$ means the variable x' depends on the variable x to be calculated. Then, for the *CanDate*, the input variables (x,y and t) are assigned with the level 0 as they don't depend on any other variables. From the graph, the *CanDate* level of a variable is the longest path (or critical path) between this variable and all the other variables. For example, if considering the variable c, starting from x, the dependency path is 2, while starting from t it is only 1, so the *CanDate*level of c is 2. For the *MustDate*, the output variables (b, d and e) are set to the level 0. Then the other levels are assigned in the same way as the *CanDate* levels, only the reasoning on the dependencies is from the outputs to the inputs.

This method allows to detect causality cycles. This is the case when a level is greater than the number of variables of the system.

Equation system example :

$$a = x \sqcup y$$
$$b = x \sqcup (\neg y)$$
$$c = a \sqcup t$$
$$d = a \sqcup c$$
$$e = a \sqcap t$$

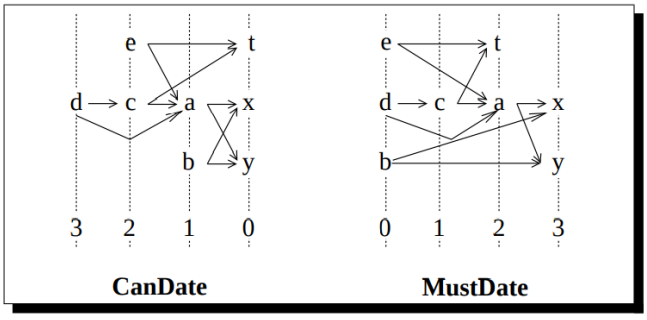


Figure 3.11: *Extracted from (Ressouche et al., 2008), this example illustrates the concepts of CanDate and MustDate. On the left is a ξ equation system where x, y and t are the inputs; b, d and e are the outputs. On the right is the corresponding dependence graph, with the different levels. Note that the $\sqcup$ here represents the $\boxplus$ operator and the $\sqcup$ represents the $\boxdot$ operator since (Gaffé & Ressouche, 2013).*

So, CLEM compiles *Light Esterel* programs into a set of ξ equation systems, and extracts an evaluation order of the equations. The compilation results are saved in a format file "*.lec*". This format is then used to integrate the related program in other modules. The modular compilation performs a *linking operation* (Ressouche et al., 2008) between the compiled modules and the main module. The operation consists in merging the sorted equation systems of the different modules, without reprocessing all the equations and evaluation orders again. The merging concerns for example the equations that computes the same variable, and an adjustment is made for the *CanDate* and *MustDate* of common variables when they are not assigned with the same dates in the different parts of the program (an example is given in (Ressouche et al., 2008)).

Compiler of Light Esterel modules Finaliser (CLEF)

The compiler CLEF is called the "finalizer", it completes the CLEM compilation, in order to generate codes. Indeed, some variables have the $\bot$ status in the ξ equation system, so this specific phase consists in replacing all the $\bot$ status by 0 (absent). Note that any $\top$ status would have thrown an error in the CLEM compilation. Compared to other synchronous approaches (Berry, 1999), only one phase compilation is used instead of two, where signals with $\bot$ status are assumed to be absent. Although this makes the program behaves effectively following a semantic behavior, it has the irreversible disadvantage of preventing any instantaneous connection with other synchronous modules that may emit these signals, thus preventing modular compilation.

The finalisation is made by taking the "*.lec*" produced by the CLEM compilation. The ξ equation system is converted into boolean equation system, so that it can be rewritten into another codes (software or hardware). One ξ equation is translated into 2 boolean equations, based on the mapping translation that encodes each element of ξ with a pair of boolean values, $B : \xi \rightarrow \mathbb{B} \times \mathbb{B}$, as shown in table 3.2. Then, only the first between the 2 generated boolean equations is kept at the end of the finalisation.

Symbol	Coding	Finalisation
$\bot$	00	0
0	01	0
1	10	1
$\top$	11	-

Table 3.2: *Mapping translation of 4-valued algebras to final boolean equations.*

Finally, different codes can be generated : software (C, Csharp, systemC), hardware (VHDL), synchronous models (Esterel, Lustre), or input format for model checking applications (Blif, NuSMV (Cimatti et al., 2002)) using the final equations.

3.2.3 Related works and applications on *Light Esterel*

Light Esterel has been and is still used in the academic field. This language is part of the curriculum for obtaining an electronics engineering degree, specifically in the embedded systems. Some research works have been conducted with the use of the *Light Esterel* language.

In (Abdelmoula, 2014), their objective was to generate exhaustive test sets to verify the correctness of a specified model. Their approach aimed to address the limitations of both industrial testing

techniques, which are typically non-exhaustive, and formal verification in scientific research, which often faces the problem of combinatorial explosion. Their study focused on the "Calypso" communication protocol, which is commonly used in transportation systems, such as in buses to validate transport smart cards. The protocol includes 33 main commands. To create the synchronous model of the Calypso protocol, they used the *Light Esterel* language. Subsequently, a specific compiler was developed for the application, as the CLEM version at that time was not able to compile the model due to the explosion of (hierarchical) states. The synchronous model is then provided as input to a generic compiler they developed, that employs a quasi-flattening algorithm and a compiled internal description to verify security properties and reduce the state space combinatorial explosion problem. As results, they not only succeeded in generating exhaustive test sets, but also they proved that the protocol communication had some bugs and undefined behaviors (unreachable states or unspecified behavior for particular sequence of inputs).

In (Barnes, 2017), they focused on the verification on the fly (in real-time) of the reliability in wireless sensor network systems. They developed a new simulation environment and validation tool to verify the communication protocols used between the nodes or sensors. *Light Esterel* was used to model the properties that needed to be verified, and these properties were given to *observers*. An observer is a module which role is to emit a signal or stop the simulation whenever a protocol property has been violated, similar to a watchdog. This way, it is possible to find the error that caused the violation using a debugger. The *Light Esterel* module is compiled into C codes and inserted in the observer, and the approach allowed to detect and to emit an alarm during the execution in real-time.

In (Sarray, 2019), a new synchronous language called "ADeL" (Activity Description Language (Sarray, Ressouche, Moisan, Rigault, & Gaffé, 2017)) has been developed for a project on activity recognition based on visual perception. The activities can be movements, actions or objectives of mobiles objects (humans, animals or simple artefacts). Activity recognition aims at recognizing a sequence of actions that follows a predefined model. The ADeL language was strongly inspired from the syntax and the compilation method of *Light Esterel*. This language has been developed to create a more accessible syntax especially for the doctors community, but also for non programmers or non computer scientists, in order to describe activities.

3.3 Model Checker

Simulating a program is the easiest way to verify its correctness. However, for complex system with large number of states, it would only cover a portion of the system's behavior, as some possibilities might be omitted. One of the main advantages of the synchronous approach is that the models can be verified through formal verification tools or techniques. Formal verification is the use of mathematical techniques to ensure accurate reasoning on finite states models. It can be the verification of the correctness of the model or the exhaustive exploration of all its possible states. There are three main types of methods : *model checking*, which explores the accessibility of specific states of the system represented by a finite states model (graph or FSM); *abstract interpretation*, which is an approximation theory of the semantics of discrete dynamical systems; and *interactive proof*, which uses tools known as "proof assistants". The first method is adapted only for systems represented with finite states models, while the other two can be used on systems without a finite representation. We chose the first one, since we can apply model checking directly on the compiled synchronous models.

One main advantage of model checking is that it is completely automated and rigorous. It is widely used in the design and verification of complex systems, such as hardware and software systems. It involves three main steps : First, the system's behavior is modeled into a graph or a FSM model, which can be achieved using synchronous languages. Second, a set of specifications or properties is expressed using a temporal logic with a specification language. Finally, an analysis method is selected to verify the system's behavior against the specifications or properties. A model checker works by returning a counterexample if the model fails to satisfy the property, which can be used to identify and fix the issues in the model. There are different types of properties that can be verified :

- *Safety properties* : something bad will never happen;
- *Liveness properties* : something good will eventually happen;
- *Temporal properties* : something will always or eventually hold true over time;
- *Invariant properties* : something will always be true in every state;
- *Reachability properties* : a particular state or a set of states can be reached from a given initial state;
- *Equivalence properties* : two different models behave in the same way.

A model checker mainly executes an exhaustive (but optimised) exploration of the FSM model to determine whether the property is satisfied. So one of the major challenge of model checking is the explosion of the number of states with the size of the system. Some works have been conducted to address this issue with *symbolic model checking* (McMillan & McMillan, 1993) to represent the system's state space using symbolic variables instead of explicitly enumerating all possible states. For example, one symbolic representation is binary decision diagrams (BDDs), which are a data structure for representing boolean functions (Baier & Katoen, 2008) and can result in significant performance gains to reduce large boolean expressions. Enumerating all the states refers to an *explicit* automata, while reducing the states through, for example symbolic representation, refers to *implicit automata*. In this work have chosen to focus on implicit automata since their compexity of O(log2(n)) is lower compared to the complexity of O(n) of explicit automata in verification (where n is the number of states).

Different model checkers exist, for example Xeve (Bouali, 1998) for Esterel programs, Lesar (Raymond, 2008) and Nbac (Jeannet, 2003) for Lustre programs, Sigali (Marchand, Bournai, Borgne, & Guernic, 2000) for Signal, etc. The choice of the model checker to use mainly depends on the type of system and the type of properties to verify. In this section, we specifically present the *Kind2* (Hagen & Tinelli, 2008) model checker for Lustre programs. According to (De Maria, Muzy, Gaffé, Ressouche, & Grammont, 2016), which has an objective close to this thesis work, it is the most powerful model checker for Lustre programs, and therefore we used it in chapter 4.

Kind2 model checker for Lustre programs

Kind2 is an open-source model checker that is based on SMT solvers (Satisfiability Modulo Theories). SMT sovlers are tools that can solve the satisfiability problem for first-order logic combined with theories such as real numbers or data structures. Kind2 was implemented based

on its predecessor *PKind* (Kahsai & Tinelli, 2011). It is designed to prove or disprove "safety" properties expressed in Lustre in the form of invariant properties. A property is related to behaviors the system at issue will never cross/reach in a finite execution *e.g.* deadlock situation.

How Kind2 is working ? We will not go into detail on all the steps, as they are beyond the scope of this thesis, but generally, Kind2 converts the Lustre system model into state transition system, then it relies on SMT based k-induction combined with several resolution engines to verify the property. K-induction is a technique which consists of checking that a given property holds for the initial state, and then proving that if it holds for the current state after k steps, it must also hold for the next state (Gurfinkel & Ivrii, 2017). The value of k is gradually increased until the property is shown to hold for all states, otherwise a counterexample is found. An example of engine is IC3 (Incremental of Inductive Clauses for Indubitable Correctness), an algorithm that generates an inductive strengthening of a property when k-induction fails to prove it (Bradley, 2012). The inductive strengthening involves adding new clauses to the property that help prove it inductively. Kind2 supports multiple SMT solvers (Z3, CVC4, CVC5, ...) or engines (BMC, PDR, ...), but since we didn't extensively study their functions or how they work, we will not detail them.

Kind2 uses an *observer* (Halbwachs, Lagnier, & Raymond, 1993), *i.e.* a lustre program in which the property is expressed and which observes in parallel the behavior of the system receiving sequences of inputs, and it returns if the property is respected at each logical instant or else it returns a counterexample. The counterexample is an execution trace that leads to the violation.

3.4 Conclusion

The synchronous approach is a promising paradigm for modeling real-time reactive systems, as it simplifies programming and makes it deterministic. The growing of interest in the synchronous approach is shown by the development of different synchronous languages we presented in this chapter, both in the academic and the industrial fields, and in a variety of applications.

Specifically, we focused on the *Light Esterel* language which we decided to use. We want to emphasize that the choice of this language does not limit the use of others, as the *Light Esterel* development chain can generate models in different languages. We also summarized some works in which the *Light Esterel* language has been used. At the present time, the language has only been used in academic research to model complex systems or to inspire development projects.

The compilation of a synchronous program generates a sequential, deterministic and generally minimal FSM. So, one of the main advantages of the synchronous approach is the ability to verify the generated FSMs through automatic techniques such as model checkers, *e.g.* to ensure the correctness and reliability of a system, or to prove/disprove properties.

Overall, the synchronous approach has proved its efficiency in multiple domains. In the next chapter, we present our work on the synchronous approach applied to artificial neural networks.

CHAPTER 4

Neuromorphic and synchronous approach

In this chapter, we discuss the intersection of the two fields we've previously discussed, neuromorphic and the synchronous approach, in the context of replicating and understanding biological neural networks. We present two works that are, to our knowledge, closely related to the context that we aim for in this thesis.

We then introduce our methodology for implementing neural models using Light Esterel, and present the validation process of these models. Along with this, we discuss updates and developments we had to make in the language environment to be able to model Spiking Neural Networks (SNNs). Finally, we detail the results we obtained in our model checking experiments with our models.

Furthermore, this chapter highlights the limits and challenges encountered in using Light Esterel and the synchronous approach in relation to the objectives we set out in this thesis. These limitations and challenges led us to focus on solutions that we address in Chapter 5.

As explained in the previous chapter, formal languages exist, as synchronous languages, to rigorously describe complex systems especially real-time reactive systems. They come with formal methods as reasoning approach tools that involves using rigorous mathematical methods to prove the validity or invalidity of propositions, theories or properties upon these formal systems.

On the other hand, SNNs are models inspired by the structure and function of the human brain. They are composed of neurons connected to each other, capable of processing information in a similar way to neurons in the human brain. They are similar to real-time reactive systems since they interact with their environment by receiving input sequences of spikes and returning response sequences of spikes.

Therefore, it is possible to use the formal approach to study SNNs in order to better understand how biological neural networks work. Synchronous languages are ideal for modeling the function of neurons and the interactions between them in a neural network, and then to use formal methods to prove the validity of certain hypotheses about their function. However, there are few studies that have applied formal methods to SNNs, in order to replicate or to understand biological mechanisms. We discuss about 2 studies we are aware of in this area in annexe B.

4.1 Spiking Neural Network modeled with *Light Esterel*

4.1.1 *Light Esterel* neural models

In this thesis, we use the *Light Esterel* language to implement neural models, which is a novel approach in the field. In the following, we explain the methodology and the updates on the *Light Esterel* environment for implementing the models, then we describe the validation of these models by using as reference the *BRIAN simulator* (Goodman & Brette, 2008). As a first step, the *Light Esterel* implementation requires the discretization of the model differential equations.

Euler method : method of solving differential equations

As described in chapter 2, section 2.1.3, neural models are defined by differential equations of order 1, which means they are defined as $f'(t) = F(t, f(t))$. Solving this type of equation involves finding all the differentiable functions $f(t)$ that satisfy this relation. These equations usually have an infinite number of solutions, each of which depends on the value of $f(t_0)$, also known as the initial condition, at $t = 0$. However, these equations are often not able to be solved using analytical methods, and this is especially true for neural models. As a result, numerical methods are used to approximate the solutions. Given an initial condition, the goal is to define a mathematical expression that can be used to calculate the solution at each time step using the previous solution value. This requires discretizing time into steps, with dt being the length between consecutive time steps. There are different numerical methods that can be used, and the choice of which to use depends on the application constraints, considering a trade-off often being made between implementation cost and level of precision. In our case, we need a numerical method that is suitable for simulating neural networks compatible with software and hardware.

In general, when the chosen time step dt is small, the approximation is closer to the true solution, but the calculations take longer as there are more time steps involved. There are 2 categories of numerical methods : dt is either adaptive or fixed (Press, Teukolsky, Vetterling, & Flannery, 1996). Adaptive time steps change according to the error of the previous approximation in order to stay close to the exact solution, but can also allow for larger time steps. However, using an adaptive

time step can increase complexity as it requires calculating different dt values for each neural unit for SNN simulations. For this reason, we have chosen the well-known intuitive simple numerical method of Euler (1768) that uses fixed dt. Let's consider, the general form of a differential equation of order 1 shown in equation 4.1. The definition of a derivative function is given in 4.2 where dt is the time step which tends to 0. By isolating the expression to calculate the solution value at the next time step, we obtain the equation 4.3. Finally, this Euler scheme allows to define the discretized expression for numerical simulation by replacing the continuous time t by discrete steps n as shown in equation 4.4, a recurrence relation with a given initial condition $f(n = 0) = f_0$.

$$f'(x) = F(t, f(t)) \tag{4.1}$$

$$\lim_{dt \to 0} \frac{f(x + dt) - f(x)}{dt} = F(t, f(t)) \tag{4.2}$$

$$\lim_{dt \to 0} f(x + dt) = f(x) + \lim_{dt \to 0} F(t, f(t)).dt \tag{4.3}$$

by replacing t by n and dt by a fixed value, we obtain the discretized expression of f :

$$f_{n+1} = f_n + F(n, f(n)).dt \tag{4.4}$$

Example 4.1.1 – In the Leaky-Integrate-&-Fire model, the membrane potential v is described by the following equations :

$$\tau v'(t) = -v(t) + I$$
$$\text{if} \quad v > v_{theshold} \quad \text{, then} \quad v = v_{reset}$$

The discretized expression of v is then :

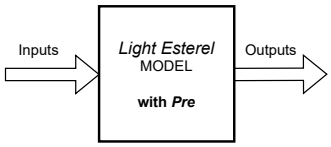
$$v[n+1] = v[n] + \frac{1}{\tau}.(-v[n] + I) \times dt$$
$$\text{if} \quad v[n+1] > v_{theshold} \quad \text{, then} \quad v[n+1] = v_{reset}$$
$$\text{with the initial condition} \quad v[0] = v_0 \quad \text{where } v_0 \text{ is a given value.}$$

The time step dt value

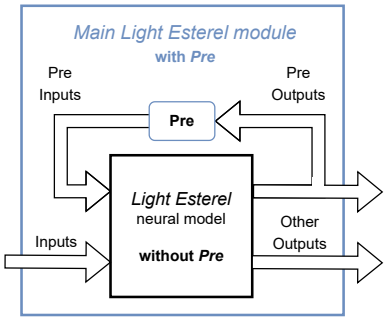
The Euler method has a local error (error per step) that is proportional to the square of dt. Therefore, when dt is small, the precision of the simulation is higher, but it also requires more calculations within a given time period. The dt value may vary between different neural models, depending on their level of biological precision and sensitivity to error in calculations. For example, the *Izhikevich* model uses a time step of 1 ms, while the *Hodgkin–Huxley* model uses a time step of 0.1 ms (Izhikevich, 2004). In the *Light Esterel* implementation, the dt value is configurable, but it must be the same for all units of a same network. In the following, we may refer to a time step as a "logical instant" in the synchronous approach.

Neural implementation in *Light Esterel*

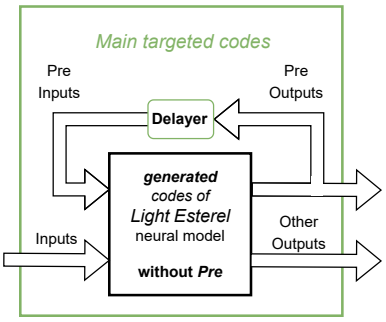
In a neural model, the update of a variable at time step n relies on its previous value at time step $n-1$. However, in *Light Esterel*, a signal value is only available at the logical instant it is emitted and becomes inaccessible afterward. To address this issue, a new instruction has been added to *Light Esterel* called "Pre" signal (short for PREvious). This feature establishes a specific local variable with a designated initial condition value and the signal it tracks, enabling access to signal values at instant $n-1$.



(a) *Model with Pre*



(b) *Model without Pre and Light Esterel main*



(c) *Model without Pre*

Figure 4.1: *3 possible implementations of a neural Light Esterel model with or without the use of pre signals. (a) With pre signals integrated into the neural model implementation e.g. the listing 4.1. (b) Without pre signals within the neural model implementation e.g. the listing 4.2. A Light Esterel main module handles the management of pre signals. (c) Without pre signals, an external module in the target code takes care of the management of the pre signals of the generated neural model, with a delayer for example,.*

There are 3 distinct approaches for implementing neural models in *Light Esterel*, each differing in how they manage storage of previous signal values. The figure 4.1 shows the different approaches :

1. The first approach uses "pre" signals within the module implementation, as illustrated in figure 4.1(a) and example in listing 4.1. Pre signals store specific signal values at each time step, and are inaccessible from outside the module. This approach is used for testing single neural models.

2. The second approach omits pre signals within the module, as shown in figure 4.1(b) and example in listing 4.2. Instead, an external *Light Esterel* main module handles the signal

storage using pre signals on the input/output of the module. We use this approach when creating neural networks that will be compiled into other codes.

3. The third approach is similar to the second one, as show in figure 4.1(c) but in this case, the *Light Esterel* module is compiled and converted into the targeted codes (*e.g.* C codes). Then the management of the signals is done by the main module in the targeted codes (*e.g.* a manually written main in C codes). We use this approach for the hardware we developed in chapter 5.

```
1  -- Types / Methods / Constant
       declarations
2  data:
3    Type: real;
4    Constant: real dt :vhdl= "to_real(1.0)"
5                      :c= "1.0"
6                      :lustre= "1.0";
7  end data
8  -- Module name / I\O / locals / Pre
9  module LIF:
10 Input: i_I       : real,
11        i_inv_tau : real,
12        i_v_thres : real,
13        i_v_reset : real;
14 Output: o_v  : real,
15         o_spike;
16 Local: new_v : real,
17        dv    : real;
18 Pre: v_pre : o_v $ i_v_reset;  -- <<
19 -- Module behavior description
20 loop
21 {
22   emit dv($i_inv_tau*($i_I - $v_pre)*dt)
23   >>
24   emit new_v($v_pre + $dv)
25   >>
26   if ($new_v >= $i_v_thres) {
27     emit o_spike
28     >>
29     emit o_v($i_v_reset)
30   }
31   else {
32     emit o_v($new_v)
33   }
34   >>
35   pause
36 }
37 end
```

Listing 4.1: *LIF implementation with "Pre"*

```
1  -- Types / Methods / Constant
       declarations
2  data:
3    Type: real;
4    Constant: real dt :vhdl= "1.0"
5                      :c= "1.0"
6                      :lustre= "1.0";
7  end data
8  -- Module name / I\O / locals / Pre
9  module LIF:
10 Input: i_I       : real,
11        i_inv_tau : real,
12        i_v_thres : real,
13        i_v_reset : real,
14        i_v_pre : real;   -- <<
15 Output: o_v  : real,
16         o_spike;
17 Local: new_v : real,
18        dv    : real;
19 -- Module behavior description
20 loop
21 {
22   emit dv($i_inv_tau*($i_I-$i_v_pre)*dt)
23   >>
24   emit new_v($i_v_pre + $dv)
25   >>
26   if ($new_v >= $i_v_thres) {
27     emit o_spike
28     >>
29     emit o_v($i_v_reset)
30   }
31   else {
32     emit o_v($new_v)
33   }
34   >>
35   pause
36 }
37 end
```

Listing 4.2: *LIF implementation without "Pre"*

We will explain the implementation of neural models in *Light Esterel*, and we take as example the LIF neuron model. In general for all neural models, the *Light Esterel* implementation structure is globally the same. It is composed of 3 main parts separated by the comment lines starting with the "−−" symbol. The first part, inside the block `data` (line 2) and `end data` (line 7), is where the types, methods and constants are declared. In this example, values are of the "real" type and there is one constant *dt*. A new feature was added that is the use of a specific assignment syntax

": *targetedCode* =", specifically useful when in some target codes, functions such as a conversion are required for the signal/variable definition (*e.g.* "to_real" in vhdl).

The second part is the module and the interface declarations (from line 9 to line 18). It contains the module name, the list of inputs/outputs, the local signals and the pre signals. With the given LIF model example, the inputs of the module listed after `Input`, are the neuron stimulation value (I) and its parameter values ($\frac{1}{\tau}$, $v_{threshold}$ and v_{reset}). We chose to place the parameters of a neural model as its inputs so that the same model can be used to create multiple modules with different parameter values. This allows for flexibility and reusability of the model. Note that division by a constant parameter is replaced by the multiplication by its inverse, such as the τ parameter. This is especially important in hardware generation, where division can be resource-intensive and time-consuming. As outputs, the model returns the membrane potential value v and the spike signal $spike$. Local signals are used for intermediate calculations and are not accessible from outside. Here, dv is the variation of the membrane potential according to the inputs and dt values; new_v represents $v[n+1]$, the new value of the membrane potential. The pre signal v_pre emits the value of v delayed by one logical instant : the syntax "o_v \$ i_v_reset" (line 18) means the signal v_pre emits v values with one logical instant delay, and it is initialized to v_reset at the first instant.

The third part is the description of the module behavior, the step calculations to obtain the outputs from the inputs. The $loop$ instruction is used to execute the module every new logical instant. Otherwise, the module would exit only at the first instant, and thus run only once. Next instructions are related to the LIF model processing steps. First, the variation dv of the membrane potential is calculated using the discretized expression of the differential equation. As a reminder, the "$" sign before a signal accesses the signal value, and without, it returns the signal predicate. Next instruction is the calculation of new_v. The obtained new value is compared to the input threshold, if it is greater, a spike is emitted and the output membrane potential is reset to the input parameter reset value, otherwise the output membrane potential takes the previous calculated new_v.

The listing 4.2 corresponds to the approach of implementing neural models without the use of the pre signals. In this method, the pre signals are removed and turned into inputs instead, *e.g.* the v_pre in the LIF module. Instead of performing the one logical instant delay on the "pre" signals within the module, it is performed outside the module by an external module (*Light Esterel* or other). This external module receives the signals that should be delayed, and sends them back to the right inputs at the next logical instant. Both implementations are equivalent in terms of behavior, and we chose the implementation without pre within the model for reasons we will explain later.

Implemented *Light Esterel* neural models and biological properties

Different neural models and some neural properties have been implemented in *Light Esterel* based on their differential equations and after spike behavior (see chapter 2). They are summarized in table 4.1, on the top left is the list of the neuron models, on the top right is the list of the synapse models, and on the bottom is the list of the neural properties. *Light Esterel* models do not limit to this list. Indeed, it is possible to add more neural models and biological properties.

Neuron models
Integrate & Fire (IF)
Leaky Integrate & Fire (LIF)
Adaptive LIF
Izhikevich (IZH)
Digital Spiking Silicon Neuron (DSSN)
Hodgkin Huxley (HH)

Synapse models
Fixed
Short Term Plasticity (STP)
Spike Timing Dependant Plasticity (STDP)

Neural properties
Exponential decays (AMPA, GABA, ...)
Axonal delay
Noise

Table 4.1: *Neural models and biological properties implemented in Light Esterel.*

Value representation

In software, the float type can represent a wide range of numbers with a reasonable precision. However, it is not supported by most hardware synthesis tools. In general, implementations of floating-point arithmetic in hardware are complex and require a lot of resources, in terms of logic gates and memory. Therefore, the fixed-point type is more suitable and more efficient for hardware implementations. It uses a fixed number of bits to represent a value, it reduces the hardware resources needed, however the precision is limited and must be carefully chosen according to the application. In this work, we use the fixed-point type for hardware implementation, and the float type for software simulation.

The floating-point type uses a *mantissa M* (also called *significand*), a *base B* and an *exponent E* to represent a real number (norm IEEE 754). Similar to scientific notation, a number N is written as : $N = M \times B^E$. The exponent indicates the position of the radix point between the significand digits, and it can be located anywhere thus the "floating" point.

Example 4.1.2 – Here is the floating-point representation of the value 12.345, in two different bases :

$$12.345 = 12\,345 \times 10^{-3} \quad \text{(Base 10)}$$

$$12.345 = 1100\,0000111001 \times 2^{-10} \quad \text{(Base 2)}$$

The fixed-point type, as its name suggests, is characterized by a "fixed" position of the point that separates the decimal part and the fractional part of a real number. In the binary format, the type is defined with the notation $Q_{x.y}$: x is the number of bits on the left before the "point", it encodes the decimal value, and y is the number of bits on the right after the "point", it encodes the fractional value. So $x + y$ is the binary format length or size to represent a value. The x depends on the maximum absolute value to be represented and y depends on the precision required, as shown in the example below.

Example 4.1.3 – In fixed-point representation, the previous example gives the following possible conversions according to two different $Q_{x.y}$ configurations :

$$\begin{aligned} 12.345 \approx 00001100.01011000 &= 2^3 + 2^2 + 2^{-2} + 2^{-4} + 2^{-5} \quad (Q_{8.8}) \\ &= 12.34375 \end{aligned}$$

$$12.345 \approx 00001100.0101100001010010 = 12.345001220703125 \quad (Q_{8.16})$$

With $x = 8$, and considering that the Most Significant Bit (MSB) *i.e.* the farthest bit to the left represents the number's sign ('0' positive, '1' negative), therefore the maximum absolute value (decimal part) that can be represented is $01111111 = 2^{8-1} - 1 = 127$.

Float type can support a much wider range of values than fixed type, with the ability to represent very small numbers and very large numbers. Float type can also be used in hardware, but only for functional simulations as it is not synthesizable for on board simulations. Thus, fixed type are used for hardware to optimize resources utilisation and speed at the cost of range and precision loss. And software simulations are used as reference to ensure that the behaviors are close, when defining the representation size in fixed type.

Validation methodology

In order to validate the *Light Esterel* models, we used the BRIAN2 simulator (Stimberg et al., 2019) which is the new version of the BRIAN simulator (Goodman & Brette, 2008) written in the Python programming language. It is a free, open source simulator for spiking neural networks of single compartment model neurons, and especially flexible to design and simulate bio-inspired and biomimetic neural models. The original aspect of Brian is that neural models can be defined by directly writing their equations in their standard mathematical notation, *i.e.* differential equations and discrete events (the effect of spikes). This allows the user to implement and simulate new models faster instead of learning and using low-level language like in the NEST simulator (Gewaltig & Diesmann, 2007) or in the NEURON simulator (Carnevale & Hines, 2006). Brian is a clock driven simulator which means that all events take place on a fixed time grid $t = 0, 1dt, 2dt, 3dt, ...$ (Goodman & Brette, 2008) as discretization of the models. The simulation of the models requires to chose different numerical integration methods, and to match our implementation method, we chose the *Euler* method.

We illustrate in figure 4.2 the validation methodology of *Light Esterel* language in neural modeling. The validation consists in verifying if the generated codes are correct and functional. We applied the different steps with our first implemented *Light Esterel* neural models (Rasamuel, Gaffé, Levi, & Miramond, 2019). The methodology is divided in 3 levels of validation : functional simulation, hardware synthesis and hardware integration. The "functional simulation" level aims at validating the software and the hardware generated codes. The "hardware synthesis" and "integration" levels are further investigations to validate the hardware generated codes for on FPGA simulations.

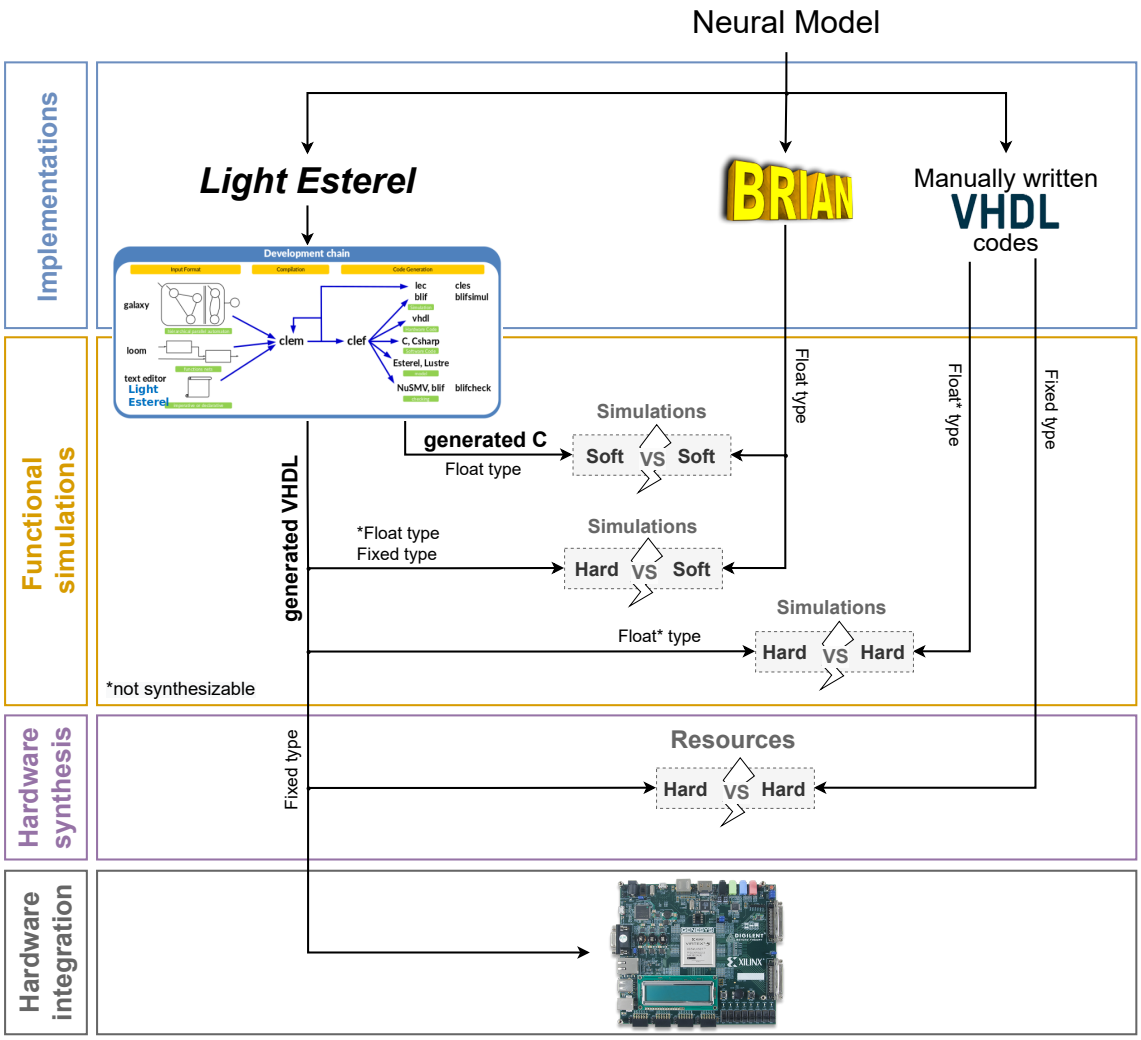


Figure 4.2: *Validation methodology of Light Esterel neural implementations.*

First, the neural model is implemented in *Light Esterel*, in BRIAN and in a manually written VHDL codes. The 3 model implementations have the same parameters with the same numerical *Euler* integration method. From the *Light Esterel* implementation, we generate the related software and hardware codes, C and VHDL respectively, by using the compilation tool presented previously in 3.2.2. In figure 4.2 :

- In the **"functional simulation" level**, each implementation is simulated either with float or fixed type for values. The simulation results of the *Light Esterel* generated codes are confronted to the references BRIAN and the manual codes. There are 3 main simulation comparisons :
  - *Light Esterel generated C vs. BRIAN* : to validate the generation of the C codes, in float type.

  - *Light Esterel generated VHDL vs. BRIAN* : when the VHDL implementation is in float type (not synthesizable), it is to ensure that the generated VHDL codes behave exactly like the reference in simulation. When the VHDL implementation is in fixed type, it is to ensure that the calculation errors induced by the fixed type do not affect too radically the hardware behavior. This comparison also allows to determine the right $Q_{x.y}$ representation in order to minimize the simulation differences.
  - *Light Esterel generated VHDL vs. manual VHDL codes* : manually written and optimized VHDL codes (with no FSMs and no pipelines) are confronted to the *Light Esterel* generated VHDL. The comparison in float type, which can not be synthesized, is to ensure that both implementations have the same behavior before the hardware synthesis validation level.

- In the **"hardware synthesis" level**, we compare the synthesis results between the generated VHDL and a manually optimized written VHDL code. The written VHDL implementation, like in the generated VHDL, doesn't involve pipeline to calculate the output values. The synthesis gives the resources occupation of the model and the maximum clock working frequency (*Fmax*) on the chosen FPGA. Since compilation transforms any *Light Esterel* model specifications into an optimized finite state machines, this comparison evaluates the efficiency of the compilation, *i.e.* the generated hardware model uses less or equivalent resources compared to a manually written VHDL codes. For the synthesis, we use the Register Transfer Level (RTL) synthesis provided in the *Intel® Quartus® Prime Standard Edition version 17.0.0* software tools.

- In the **"hardware integration" level**, we synthesize the *Light Esterel* generated hardware codes to run on the FPGA board. This is the on-board validation to verify that the model is working when simulated on the FPGA in real-time.

Validation results

We show example results in figures 4.3 and 4.4, based on the IZH neuron model, more precisely the chattering neuron class (Izhikevich, 2003). With the same experiment protocol, the simulations show the evolution of the neuron membrane potential when applying a constant stimulation value during a period of time. Note that we chose to not display the simulations of the manually written VHDL codes, but we assure that the simulations are identical to the generated VHDL.

In figure 4.3, the simulations refer to implementations with float type. From top to bottom, the simulations are BRIAN, the generated C, and the generated VHDL. We can see that the 3 simulations are the exactly the same, their values at each time step have been checked and they are equal. Therefore, we can conclude that the compilation and the generation of the *Light Esterel* codes can be validated in software and hardware.

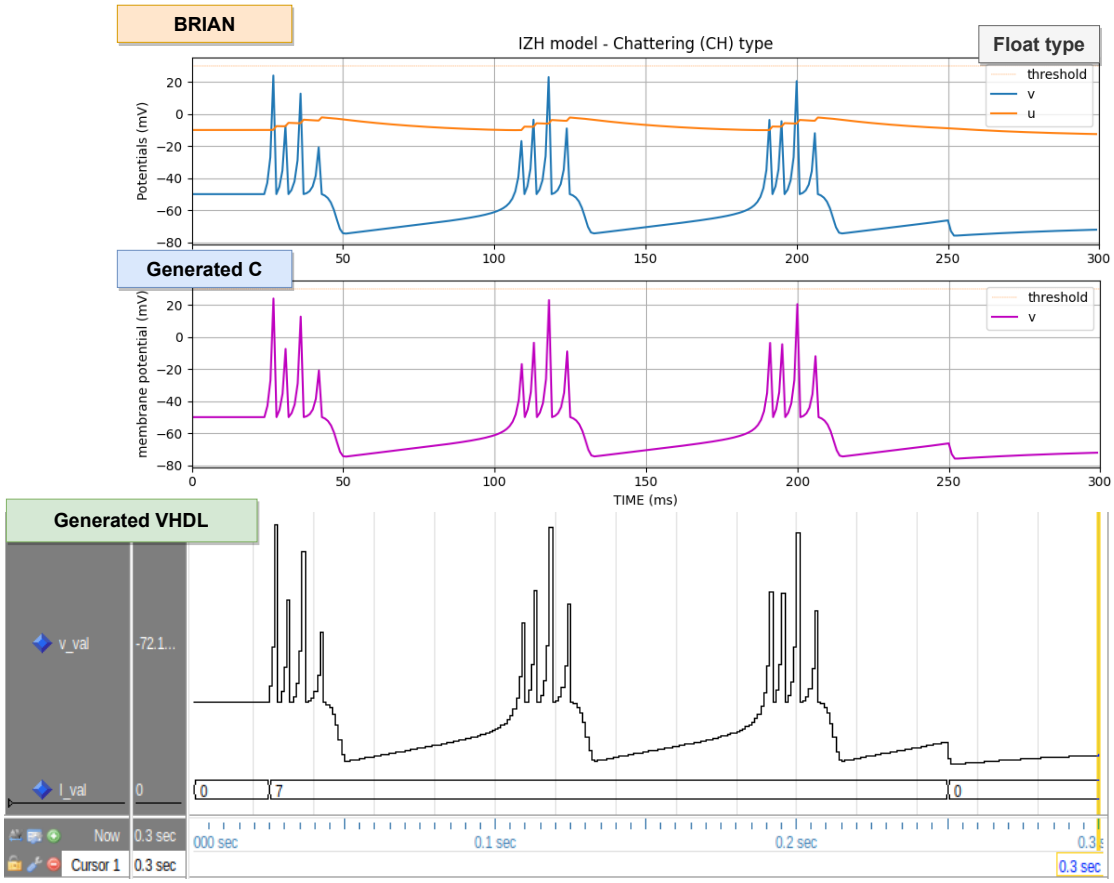


Figure 4.3: *Membrane potential simulations in BRIAN and of the generated Light Esterel codes (C and VHDL) of the IZH neuron model. The neuron class is the chattering (CH) type. On top, the BRIAN simulation. In the middle, the generated C simulation. On the bottom, the generated VHDL simulation. A 7 mV constant stimulation is applied to the neuron from 25 ms to 250 ms in the simulations.*

However, more investigations are needed for the generated hardware codes for FPGA simulations. So in figure 4.4, we simulate the generated VHDL with 3 different configurations of representation size $Q_{x.y}$. As a reminder, fixed point type is more efficient for hardware implementations.

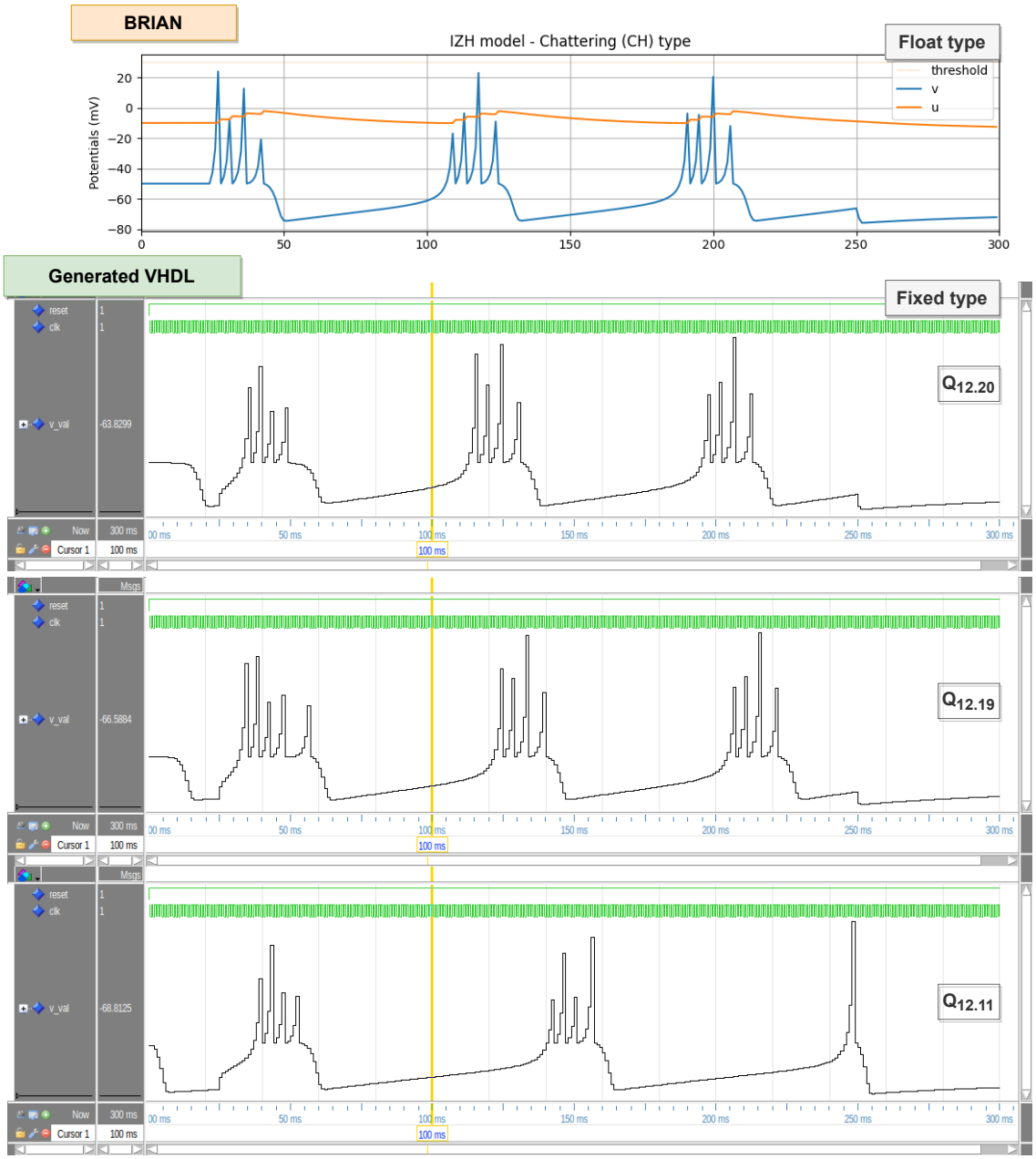


Figure 4.4: *Simulations of BRIAN and the generated Light Esterel codes (C and VHDL) of the IZH neuron model, more precisely the chattering (CH) neuron class model, with float type. On top, the BRIAN simulation. In the middle, the generated C simulation. On the bottom, the generated VHDL simulation. A 7 mV constant stimulation is applied to the neuron from 25 ms to 250 ms in the simulations.*

Globally, we can already notice that the BRIAN simulation (with float type) and the hardware simulations (with fixed type) are not in phase. This right shift in time of the hardware simulations can be explained by the membrane potential fall at the very beginning of each simulation, a consequence of the errors induced by fixed type calculations. As we can see, the larger the size of the value representation, the later the fall. Therefore in hardware, because of this phenomenon

it takes more time to the neuron to reach the threshold value when the stimulation is applied for the first time. Another difference we can notice is that the errors can also cause the emission of less or additional spikes *e.g.* with $Q_{12.19}$, even though only 1 bit has been removed in the value representation (see $Q_{12.20}$). The hardware behavior diverges more from the reference as the representation size decreases (see $Q_{12.11}$), thus it highlights how important it is to chose the right $Q_{x.y}$ configuration. In this case, by referring to the BRIAN simulation, we can assume that $Q_{12.20}$ is the best size to stay accurate but at the cost of using more logics in hardware. Note that the best size can differ with other classes of the same IZH neuron model, *e.g.* with the regular spiking (RS) where the configuration $Q_{12.19}$ is sufficient.

Another investigation on the hardware generated VHDL codes corresponds to the verification of how optimized the implementation is in terms of resources occupation in a FPGA : the "hardware synthesis" level. So we used as target the FPGA device 5CGXFC9E7F35C8 from the Cyclone V family. The available resources on the FPGA are Adaptive Logic Modules (ALMs), registers, Digital Signal Processings (DSPs) and memory. The table 4.2 details the resources utilisation obtained from the *Quartus* synthesis tool, of different implementations of the same neural model in *Light Esterel* and manually in VHDL. We took the same example : the IZH neuron model. In *Light Esterel*, we show two versions : with pre and without pre signals (see figure 4.1). Then, we compared the generated VHDL codes with their equivalent manually written codes. The manual codes are written to use as less resources as possible, the rule being to update the outputs of the neural model at each time step, with no pipeline techniques.

IZH model resources utilization					
		With Pre		*Without Pre*	
VHDL implementation		***Light Esterel***	**Manual**	***Light Esterel***	**Manual**
Resources	**Total available**				
ALMs	113 560	246	**212**	228	**204**
Registers		220	**201**	189	**186**
DSPs	112	11	11	11	11
Memory	12 492 800	0	0	0	0
Fmax		33.8 Mhz	**38.22** Mhz	**220.02** Mhz	-

Table 4.2: *Resources utilization of the IZH model implemented manually in VHDL vs. the Light Esterel generated VHDL, considering $Q_{12.19}$. The FPGA board used is the Cyclone V family - 5CGXFC9E7F35C8. Two versions of implementation are synthetized : with and without the pre signals. The maximum frequency clock (Fmax) is also given for each implementation. Note that the synthesis results "without pre" refer to the neuron model only, it does not include the management of the pre signals as explained in figure 4.1. Therefore, the synthesis of the manual codes without pre gives higher Fmax as the implementations are almost fully combinatorial.*

According to the synthesis results, in general one IZH neuron is only using 0.20% (± 0.013%) of the total available ALMs, and 9.82% of the available DSP blocks. Between the two implementations, the manual one uses the least amount of resources and has a higher Fmax. More precisely, the generated codes use 16.03% more ALMs (respectively 11.76%), 9.45% more registers (respectively 1.61%) compared to the manual codes in the version with pre (respectively without pre). Nevertheless, it does not mean *Light Esterel* implementation is not ideal, because the excess

resources usage is negligible compared to the overall capacity of the FPGA. When comparing both *Light Esterel* version, the "without pre" version uses less resources which is logical as the pre process is outside the model, but the most important is that the maximum frequency is much more higher than the "with pre" version. Note that the Fmax in the manual without pre version is particularly not provided because the manual implementation doesn't involve register to register path, required to calculate Fmax. For these reasons, we chose the version without Pre for the neural implementation in *Light Esterel* as it gives the best performance in the synthesis results.

In conclusion, in this section we detailed how a neural model can be implemented in *Light Esterel*. The simulations have confirmed that the generated *Light Esterel* codes are functional in software and hardware, validated with the BRIAN simulator as reference. In hardware, the synthesis results have led us to chose the implementation without pre in the *Light Esterel* neural module. It has also confirmed that the resources utilisation of the generated VHDL codes is close to an optimized manually written VHDL codes. Therefore, *Light Esterel* compilations and code generations are adapted to neural model design and simulations.

4.1.2 Limits of *Light Esterel* compilation tools

After validating in the previous section the use of the *Light Esterel* language to implement neural models, the next step is to implement SNNs based on them. With the `run` function, the *Light Esterel* language is capable of modular implementation *i.e.* to duplicate *Light Esterel* modules and to connect them together to construct more complex modules. As SNNs are structures of neurons interconnected via synapses, therefore *Light Esterel* is suitable for SNN specification whose structure is illustrated in figure 4.5. However, we reached some limitations with the *Light Esterel* compilation tools for SNN compilations.

Limit 1 : SNN model size

A first limit encountered was the total time needed to compile *Light Esterel* SNN models based on the modular implementation in figure 4.5. As we present in the graph in figure 4.6, the compilation time exponentially increases as the SNN size expends. The graph shows the compilation time of CLEM, the code generation time of CLEF and both summed for different SNN sizes. Even though the codes generation time is close for all target codes, the reported times are the mean time between *C*, *lustre* and *VHDL* codes generation. For the SNN models, as we don't have specific configurations, we decided to test the *Light Esterel* compilation tools on fully-connected models as worst cases. So, the compilation times are based on fully-connected SNNs composed only of LIF neurons, STP synapses and axonal delays equal to one time step $dt = 1ms$. In conclusion, 12 neurons (and 144 synapses) were the maximum size configuration we tested, as it took already more than 3.5 hours in total to compile (CLEM : 1.28 hours) and to generate the related codes (CLEF : 2.28 hours ± 18 seconds). This maximum SNN configuration is small, compared to (Ambroise, 2015) whose application is similar to ours, they modeled SNNs reaching a maximum of 100 neurons and 6200 synapses, with the same neural models and whose maximum axonal delay can go up to 50 ms for each synapse.

```
-- SNN specifications
module SNN:
...
-- neuron runs
loop {
  {
    run IZH[l_a1\a,l_b1\b,...l_spike1\o_spike]
    ||
    run IZH[l_a2\a,l_b2\b,l_c2\c,...l_spike2\o_spike]
    ||
    run IZH[l_a3\a,l_b3\b,l_c3\c,...l_spike3\o_spike]
    ||
    ...
  }
  >> pause
}
||
-- synapse runs
loop {
  {
    run STP[delayed_spike0\i_spike, l_p0\i_P,...\o_Isyn]
    ||
    run STP[delayed_spike1\i_spike, l_p1\i_P,...\o_Isyn]
    ||
    run STP[delayed_spike2\i_spike, l_p2\i_P,...\o_Isyn]
    ||
    ...
   }
   >> pause
}
-- neural properties runs (delay, noise, ...)
loop {...}
||
-- neural parameters
loop {
  emit l_a1(A1_const) >>
  emit l_b1(B1_const) >>
  ...
  >>
  pause
}
end
```

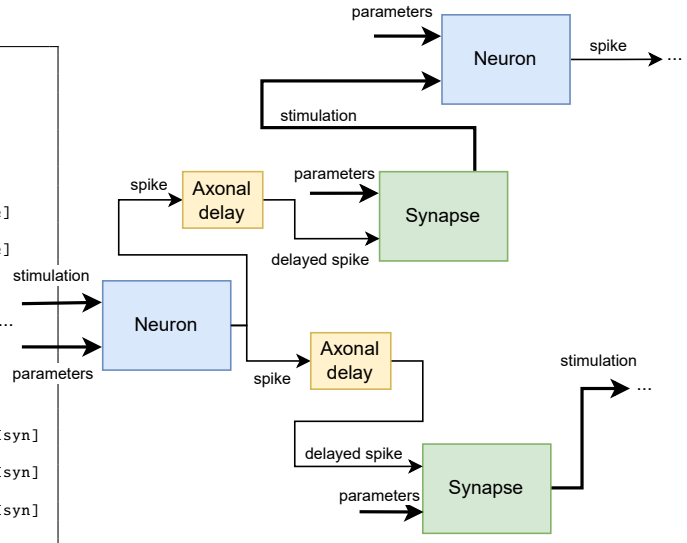


Figure 4.5: *Illustration of modular implementation in Light Esterel. On the right is the illustration of a neural network created by connecting I/O of boxes that represent the Light Esterel neural modules i.e. neuron, synapse, delay, etc. Mainly, a neuron is stimulated by a synapse, its output spike goes to the axonal delay module which outputs the delayed spike to the synapse etc. The modules can have different parameters. On the left is to show the Light Esterel specification format file to create a neural network. Each box is called with the run command.*

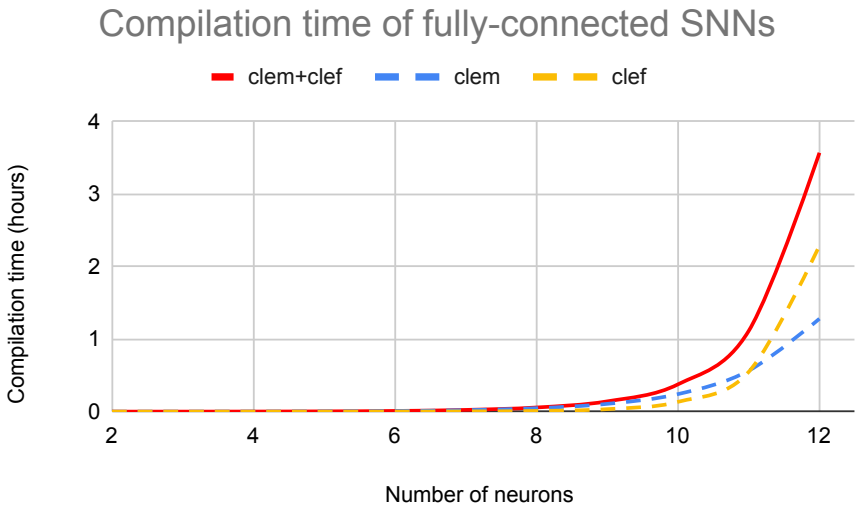


Figure 4.6: *Compilation times of fully-connected Light Esterel SNN models : all axonal delays are 1 ms, neuron model is LIF and synapse model is STP. Fully-connected : the number of synapses is equal to the square of the neurons number. All the targeted codes have approximately the same compilation time with CLEF.*

Limit 2 : SNN model hardware generation

A second limit concerns the SNN hardware codes generation. As *Light Esterel* SNN implementation is modular as illustrated in figure 4.5, each module is then generated individually by CLEF in hardware (VHDL). The resulting hardware codes is especially not ideal for hardware platforms where the resources are limited. For example, in table 4.2 the IZH neuron model uses 11 DSPs and with a total of 112 available DSPs on the FPGA board example, the maximum number of neurons that can be generated and simulated is therefore 10 neurons (without synapses). Evidently, if we consider synapse resources as well, the resulting SNN would be much more smaller. In conclusion, the *Light Esterel* hardware generation, in our case of SNN models, is not optimized for hardware platforms.

Limit 3 : SNN specifications

A third limit, not related to the *Light Esterel* compilers, is the specification of an SNN. Although, *Light Esterel* is modular, the process of creating and specifying an SNN requires the declaration and definition of each neuron and each synapse, including their specific parameters, and can be a laborious task and error-prone, especially with large networks. To overcome this limitation, we developed a specific format file to specify the SNN configurations called "GLN" file. It allows to easily specify the models, connections and parameters. It also allows the specifications of random values for parameters and connections if needed. This new file format is then used within the compilation environment.

The choice of using the *Light Esterel* language is to take advantage of the *Light Esterel* compilation environment for SNN designs. That is to design SNN models from a high level specification language and to generate the corresponding software or hardware or other synchronous codes. To understand biological neurons information processing, we expected not only to use automatic provers on the developed *Light Esterel* models but also to use the models to communicate with biological neurons in real time. However, the 2 limits were a turning point in our work. Indeed, the constraint on SNN size that is supported by the *Light Esterel* environment tool is much smaller than SNN sizes found in other studies (Ambroise, 2015; Buccelli et al., 2019) which also aim to understand biological neurons by communicating with them.

4.1.3 *Light Esterel* compilation environment updates

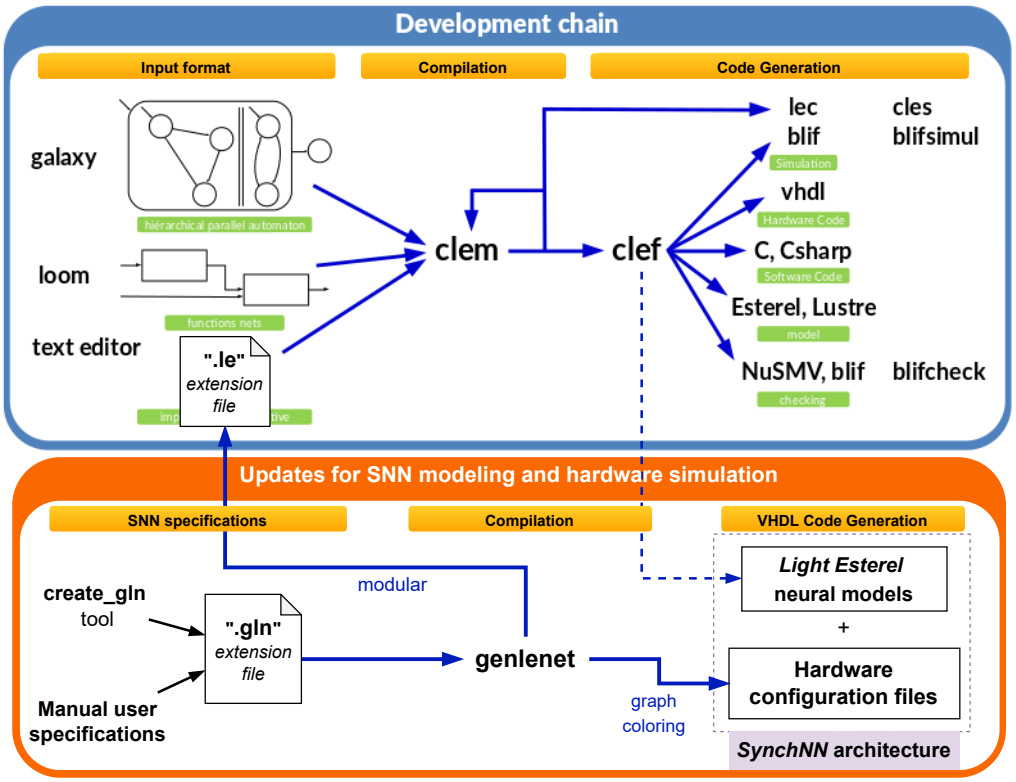


Figure 4.7: *Compilation environment of Light Esterel. On top (blue box) is the native compilation environment of Light Esterel. On the bottom (orange box) is the updates we added to be able to model and simulate SNNs in hardware.*

The first limit is related to the transformation functions of the two compilers CLEM and CLEF. Even though they are still being updated, the compilers are assumed to be already well-optimized and therefore it would require huge and complex modifications to address this first limit. For these reasons, we decided to address the second limit *i.e.* to simulate larger SNNs in hardware based on the same *Light Esterel* neural models we already developed. For that, we updated the *Light Esterel* compilation environment by adding a new specific environment for hardware configurations. This hardware environment configurations is related to the specific hardware architecture we developed, called "***SynchNN***". Its goal is to automatically generate configuration files for the architecture that supports our synchronous neural models. The updated compilation tool is illustrated in figure 4.7 and the developed hardware architecture will be described in the next chapter 5.

SNN specifications : ".gln" file

To model SNNs, we created a new format file called "*gln*" in which the user specifies the SNN configurations. This is a front end configuration file to avoid the laborious work to specify and configure the neural network. This file contains the SNN information such as the number of neurons/synpases, the neuron/synapse models, the parameter values of each neuron/synapse, the connections between the neurons and the connections from the external environment. The "*gln*"

file can be written manually by the user or generated automatically by using the "*create_gln*" tool we developed. The *create_gln* tool takes as inputs the network or file name, the number and model of neurons, the density connection between 0% (just a sequence of connected neurons) and 100% (fully-connected network), and the number of external neurons; then it generates the configuration of a randomly connected SNN in a *gln* file.

create_gln <*file_name*> <*number_of_neurons*> <*neuron_model*> <*connection_density:0%–>100%*> <*number_of_external_synapses*> <*synapse_model*>

Firstly, this configuration file allows to generate the related *Light Esterel* specifications, then to generate software or hardware codes through the initial compilation process. Secondly, it allows to configure the *SynchNN* architecture by generating the necessary configuration files through the updated compilation process. Thirdly, as a future perspective, the *gln* file aims to establish a framework for integrating other software simulators such as the BRIAN simulator with our developed architecture. The idea is to validate simulations in tools such as BRIAN, then to generate the configurations for the hardware architecture, through an after-simulation-generated *gln* file. This work is still in progress.

An example of "*gln*" file is shown in figure 4.8 with the related SNN configuration illustration. We took the IZH model for the neurons and the STP model for the synapses. Only for the sake of understanding, all the units have the same parameters. There are 3 neurons and 6 synapses in total. 3 synapses are "internal" synapses (black arrows) which connect the neurons within the SNN, while the other 3 synapses (red arrows) are "external" synapses that connect the SNN with external spikes stimulations. So, the communication with the external environment is also based on "spike" exchanges. The "`gates`" represent the external neurons or channels (illustrated by the red dashed circles). And as we show, the same channel (gate) can stimulate different "external" synapses or regions in the SNN.

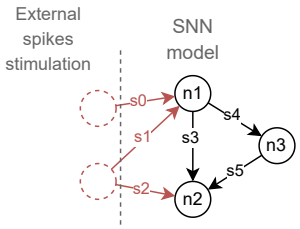
```
-- <file_name>  : <#neurons> : <#synapses> :       <#gates>
Name SNN_example : NEURONS : 3 : SYNAPSES : 6 : GATES : 2;

NEURONS :
-- <IZH> : <a> : <b> : <c> : <d> : <Iconst>
IZH : 0.02 : 0.2 : -50.0 : 2.0 : 10.0, --n1
IZH : 0.02 : 0.2 : -50.0 : 2.0 : 10.0, --n2
IZH : 0.02 : 0.2 : -50.0 : 2.0 : 10.0; --n3

SYNAPSES :
-- <STP> : <n_pre> : <n_post> : <P> : <inv_taux> :  <
    inv_tauWsyn> : <W> : <delay>
STP : 0 : 1 : 4.0 : 0.07 : 1.3 : -1.7 : 1, --s0
STP : 0 : 1 : 4.0 : 0.07 : 1.3 : -1.7 : 1, --s1
STP : 0 : 2 : 4.0 : 0.07 : 1.3 : -1.7 : 1, --s2
STP : 1 : 2 : 4.0 : 0.07 : 1.3 : -1.7 : 1, --s3
STP : 1 : 3 : 4.0 : 0.07 : 1.3 : -1.7 : 1, --s4
STP : 3 : 2 : 4.0 : 0.07 : 1.3 : -1.7 : 1; --s5

GATES :
1, -- line index of s0 in the SYNPASES list above
2 : 3; -- line indexes of s1, s2
```

(a) *"gln" file*



(b) *related SNN configuration*

Figure 4.8: *Example of a "gln" file in (a) which describes the SNN configuration in (b). Neurons are represented by the circles or with prefix 'n' and the synapses are represented by the arrows or with prefix 's'. Red color means the neurons/synapses are external while black color means internal neurons/synapses.*

The connections are specified in the "SYNPASES" list, by using the indexes of the neurons in the "NEURONS" list. It relies on specifying the neuron "pre" followed by the neuron "post" after the synapse model name STP : "pre" is for "presynaptic" neuron and "post" is for "postsynaptic" neuron. The neuron index '0' is reserved for external neurons, therefore synapses with a neuron index "pre" equals to '0' are "external" synapses. The "GATES" list is for specifying which external neurons (channels) are connected to which external synapses. The numbers are the indexes of the synapses in the "SYNAPSES" list. Finally, the other values are the parameters of the models, where the last parameter in the synapse list is the axonal delay.

"Genlenet" compiler

"Genlenet" is the developed compiler that takes as input the "gln" file and generates the according SNN model. There are two options : the compiler can generate the SNN *Light Esterel* model (file ".le") or the configuration files for the *SynchNN* hardware architecture we developed. When generating the *Light Esterel* file ".le", the implementation style is modular as we illustrated in figure 4.5 and where we previously exposed the limits. When generating the hardware configuration files, the compiler runs a coloring graph algorithm on the SNN topology to obtain configurations optimized for hardware platform simulations. In this hardware architecture described in the chapter 5, we made sure to use the same *Light Esterel* neural models listed in table 4.1.

4.2 Model checking experiments

The use of the synchronous approach was to verify if we can apply formal verification tools on SNNs, especially model checking techniques to understand neural mechanisms, *e.g.* to extract

temporal properties. In figure 4.9, we illustrated the model checking principle applied to this work. Basically, model checking is an automated verification technique that takes as inputs a property and a system model, and systematically checks whether the property holds for that model for all the possible accessible states. The possible output are "satisfied" when the model satisfies the property; "violated" when the model doesn't satisfy the property and it returns a counter-example; or "insufficient memory" when the system is too complex *i.e.* the number of states of the model is to wide to be checked. By using model checking, we wanted to confront a SNN model to a given behavior, and conduct 2 experiments : (E1) to verify the equivalence in terms of spiking output behavior and (E2) by taking advantage of the counter-example output, to explore input parameters so that the SNN model behaves in a desired way. From a biological application point of view, (E1) would be used to validate SNN models compared to real neural activity measurements, and (E2) would be used to look for parameters so that the SNN model would replicate biological neural activity.

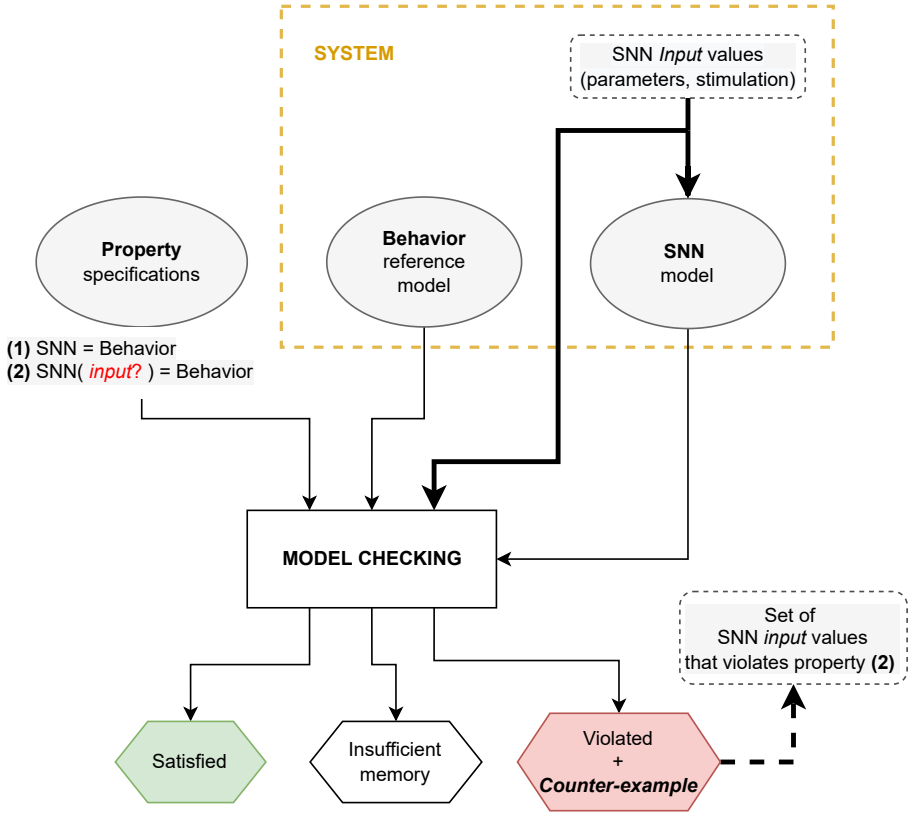


Figure 4.9: *Principle of model checking.*

We took inspiration from the work of (De Maria et al., 2016) described in section B.1. They used the *Kind2* model checker (Hagen & Tinelli, 2008) to extract temporal properties from neuronal "archetypes", and identified this model checker as the most powerful one compared to other existing model checkers for Lustre programs. Even though *Kind2* has been specifically developed for Lustre (see section 3.1.4), the *Light Esterel* compilation tool can generate Lustre codes from *Light Esterel* codes. Therefore, in the following, we have generated the neuronal Lustre nodes from *Light Esterel*

specifications, and we only wrote the Lustre programs containing the observer node and the property.

Related to (E1) and (E2) experiments explained previously, we tested *Kind2* on one single neuron first, instead of an entire SNN. Consequently, we will call (E'1) a derivative of (E1) *i.e.* a comparison of one neuron and one behavorial nodes, and (E'2) a derivative of (E2) *i.e.* the parameters exploration so that a neuron spikes in a desired way. To verify how powerful *Kind2* is, each experiment was conducted on two different neuron models we implemented in *Light Esterel*, a simple model and a more complex model : the LIF model and the IZH model, respectively. In the following, all neuron nodes are the generated lustre program from *Light Esterel* implementation.

4.2.1 Neuron behaviors comparison

The observer node structure for the behaviors comparison experiment is illustrated in figure 4.10. In this experiment, there are 3 main nodes: the neuron node, the behavioral node and the observer node. The two first nodes constitute the system to verify. The neuron node has its input parameters and its input stimulation controlled by the observer using *assertions*, and outputs the resulting spike sequence. *Assertions* are rules, conditions or properties on the system that are assumed to be true at all instants. In this case, we use assertions to set the neuron node input values as constants. The behavioral node outputs a spike sequence we manually expressed, to serve as a reference behavior. And finally, the observer takes as inputs the neuron's inputs, it confronts both output spikes from the system and outputs whether the property it embeds is satisfied or not at each logical instant. In this experiment, we want to verify if "the two system's nodes always behave the same way", therefore the safety property to verify is expressed as "$o1 = o2$" where $o1$ is the output of the neuron node and $o2$ is the output of the behavorial node.

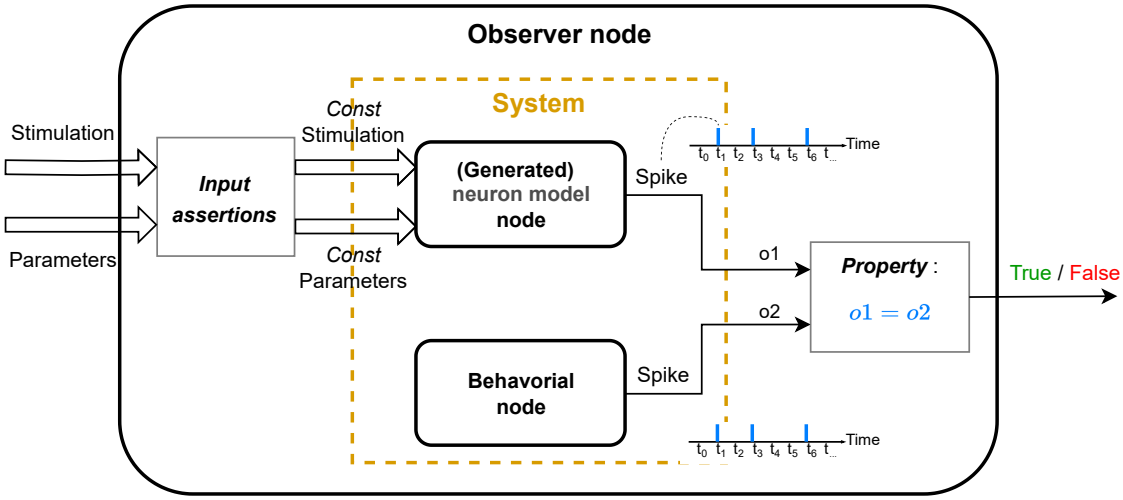


Figure 4.10: *Observer node structure for Kind2 neuron behaviors comparison experiment. The observer node is composed of the system (formed by the neuron and the behavorial nodes), the (safety) property and some assertions on the neuron inputs. The observer verifies if the system satisfies the property at each logical instant for the given inputs.*

As an example, we show the observer lustre implementation of this experiment in the lustre node 4.1 with the LIF model :

Lustre node 4.1: Observer implementation for behaviors comparison experiment, with the LIF model

```
node observer (i_I: bool; i_I_val: real; i_inv_tau: bool; i_inv_tau_val: real; i_v_thres
    : bool; i_v_thres_val: real; i_v_reset: bool; i_v_reset_val: real) returns (SIMILAR:
     bool);
    var useless : bool;
    var o1: bool;
    var o2: bool;
    var o_v: bool;
    var o_v_val: real;
let
    useless = false;
    -- Assertions on inputs --
    assert i_I_val = 30.0;
    assert i_inv_tau_val = 0.2;
    assert i_v_thres_val = 20.0;
    assert i_v_reset_val = 0.0;
    -- Nodes outputs --
    (o_v, o_v_val, o1) = lif(i_I, i_I_val, i_inv_tau, i_inv_tau_val, i_v_thres,
    i_v_thres_val, i_v_reset, i_v_reset_val);
    o2 = behavior(useless);
    -- Property --
    SIMILAR = (o1 = o2);
    --%MAIN;
    --%PROPERTY SIMILAR;
tel
```

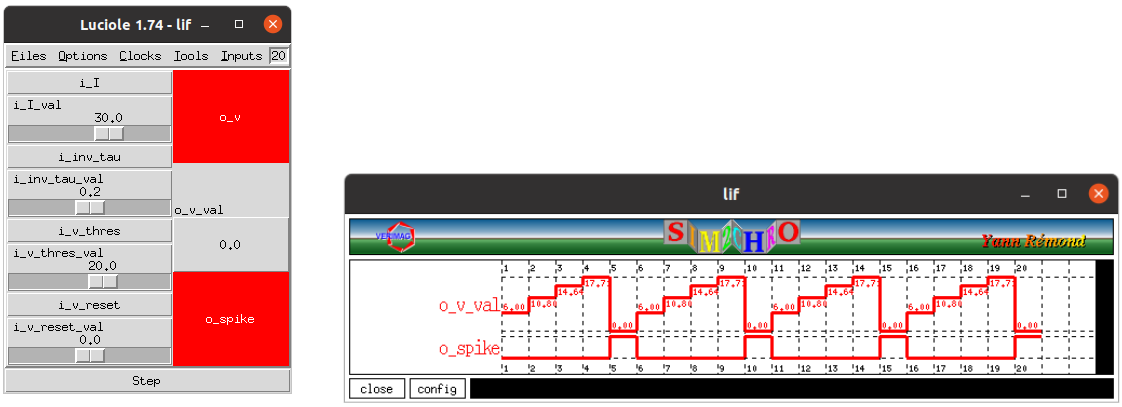
The LIF membrane potential v depends on the stimulation value I and the three constant parameters $\frac{1}{\tau}$, v_{reset} and $v_{threshold}$, as we recall the discretized equation that describes v :

$$v[n+1] = v[n] + \frac{1}{\tau}.(-v[n] + I) \times dt \qquad (\text{with } v[0] = v_{reset})$$
$$\text{if} \quad v[n+1] > v_{theshold} \quad \text{, then} \quad v[n+1] = v_{reset}$$

The node observer takes as input the neuron inputs and output the boolean `SIMILAR`. In the first part, the assertions, preceded by "`assert`", are to specifiy the constant values of the inputs. Next, the two nodes are called and their outputs are assigned to the local variables `o1` and `o2`. Finally, the `SIMILAR` output is expressed as a boolean function of `o1` and `o2`, but also specified as the property to verify with "`--%PROPERTY SIMILAR`".

The figure 4.11 shows the simulation of the lustre LIF node using the *Luciole* tool. This tool allows the user to choose the input values (figure 4.11(a)) and to display the I/O flows at each logical instant. The tool *Sim2chro*, included in *Luciole*, converts flows of values into "viewable" chronogram (figure 4.11(b)). In the simulation, we fixed $I = 30$, $\frac{1}{\tau} = 0.2$, $v_{reset} = 0$ and $v_{threshold} = 20.0$. As a result, the neuron outputs a spike every 5 logical instants.

We confronted the LIF node with the previous parameter values to two behavorial nodes shown in figure 4.12 : one spike sequence similar to the LIF node 4.12(a) and another spike sequence different to the LIF node 4.12(b). The last spike sequence is slightly different to the first one, by emitting one spike at a random logical instant. The behaviors are implemented in *Light Esterel* then generated in Lustre, like the neuron node.



(a) *Luciole window* **(b)** *Chronogram by Sim2chro*

Figure 4.11: *Simulation of the LIF neuron node using the Luciole tool. (a) shows the Luciole simulation window where the user can choose the input states (present/absent) and values, on the left. On the right, the output states (red means "present") and values are displayed at each logical instant. (b) is the chronogram to visualize the outputs : the membrane potential v and the spike.*

(a) *Similar behavorial node* **(b)** *Different behavorial node*

Figure 4.12: *Simulations of two different behavorial nodes to confront the LIF neuron node. (a) Simulation of the behavorial node whose spike sequence is similar to the LIF node in figure 4.11(b) : expressed as "emit a spike every 5 logical instants". (b) Simulation of the behavorial node whose spike sequence is slightly different to the LIF node, one additional spike is emitted at the* 13^{th} *logical instant.*

The *Kind2* results are given in figure 4.13. For the asserted constant input values, when both nodes have the same behavior (figure 4.13(a), *Kind2* validates the property `SIMILAR` after 10 time steps (0.394s). And when the behaviors are different (figure 4.13(b)), *Kind2* refutes the property `SIMILAR` after 13 time steps (0.226s), when the difference occurs between $o1$ and $o2$. In this experiment, we noticed that the model checking analysis time depends on when the spike difference occurs : if the additional spike appears at the 123^{th} time step, it takes ~17s to *Kind2*; if it appears at the 1023^{th} time step, it takes ~413s. This can be explained by the fact that the implementation of the additional spike leads to the increase of the size of the related system transition that *Kind2* generates, hence it tests more states.

```
Analyzing observer
  with First top: 'observer'
                  subsystems
                    | concrete: lif, behavior

<Success> Property SIMILAR is valid by inductive step after 0.394s.

Summary of properties:

SIMILAR: valid (at 10)
```

(a) *Similar behaviors*

```
Analyzing observer
  with First top: 'observer'
                  subsystems
                    | concrete: lif, behavior

<Failure> Property SIMILAR is invalid by bounded model checking for k=12 after 0.226s.

Counterexample:
  Node observer ()
    == Inputs ==
    i_I            false false false false false false false false false false false false false
    i_I_val          30    30    30    30    30    30    30    30    30    30    30    30    30
    i_inv_tau      false false false false false false false false false false false false false
    i_inv_tau_val    1/5   1/5   1/5   1/5   1/5   1/5   1/5   1/5   1/5   1/5   1/5   1/5   1/5
    i_v_thres      false false false false false false false false false false false false false
    i_v_thres_val    20    20    20    20    20    20    20    20    20    20    20    20    20
    i_v_reset      false false false false false false false false false false false false false
    i_v_reset_val      0     0     0     0     0     0     0     0     0     0     0     0     0
    == Outputs ==
    SIMILAR         true  true  true  true  true  true  true  true  true  true  true  true false
    == Locals ==
    o1             false false false false  true false false false false  true false false false
    o2             false false false false  true false false false false  true false false  true
    useless        false false false false false false false false false false false false false

Summary of properties:

SIMILAR: invalid after 12 steps
```

(b) *Different behaviors*

Figure 4.13: *Kind2 results on two sub-experiments on the LIF model : (a) when the neuron node and the behavorial node have the same output and (b) when both nodes have one spike difference.*

In the case of the more complex model IZH, the spike sequence is not strictly regular as with the LIF model. So instead of manually expressing the spike sequence for the behavior node, we used the same IZH model with constant parameters as the behavior node. With the same assertion method on the input values for the neuron node, *Kind2* is also able to prove or disprove the similarity property between the neuron and the behavior. However, the model checker is facing a more complex system hence a bigger transition system to verify. As a consequence, it outputs "error" or "warning" messages even though it succeeds the checking as shown in figure 4.14.

```
Analyzing observer
  with First top: 'observer'
                  subsystems
                    | concrete: izh, behavior

<Error> Runtime failure in property directed reachability: SMT solver failed: line 1768 column 48:
 logic does not support nonlinear arithmetic

<Warning> Child process 233573 (property directed reachability) exited with return code 2

<Success> Property SIMILAR is valid by 2-induction after 3.036s.

Summary of properties:

SIMILAR: valid (at 2)
```

Figure 4.14: *Kind2 results on the behavior comparison experiment on the IZH model.*

4.2.2 Neuron parameters exploration

The observer node structure for the parameters exploration experiment is illustrated in figure 4.15. In this experiment, we tried to take advantage of the "counterexample principle" of *Kind2* when a property is false, to find the constant parameters to apply to a neuron so it outputs a desired sequence of spikes. Therefore, we use the negation on the real property we want to obtain : instead of "both nodes have the same sequence of output spikes", we want *Kind2* to check that "both nodes will *never* have the same sequence of output spikes during a certain number of time steps". We wrote this property as "$o1\{\Delta t_k\} \neq o1\{\Delta t_k\}$". Compared to the previous experiment, first, the observer checks the property according to the outputs on multiple time steps. We define Δt_k the length of the time steps observation window, *i.e.* the $k+1$ "first" time steps over which the property is verified. Second, here there is no imposed input values, but assertions are used to specify that the values assigned to the neuron inputs at the first time step remain the same throughout the next time steps. Otherwise, *Kind2* changes the parameters at each time step to refute the property.

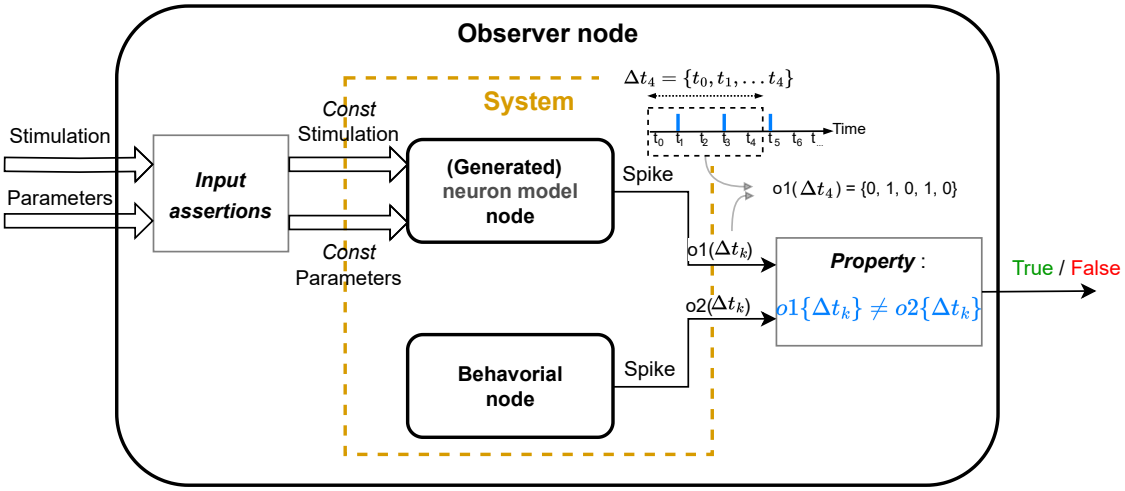


Figure 4.15: *Observer node structure for Kind2 neuron parameters exploration experiment. Δt_k defines the number of time steps .*

The observer lustre implementation for this experiment with the LIF model is shown in the lustre node 4.2, considering a Δt_5 :

Lustre node 4.2: Observer implementation for parameters exploration experiment, with the LIF model

```
node observer (i_I: bool; i_I_val: real; i_inv_tau: bool; i_inv_tau_val: real; i_v_thres
    : bool; i_v_thres_val: real; i_v_reset: bool; i_v_reset_val: real) returns (
    NOTSIMILAR: bool);
    var useless : bool;
    var o1, o2: bool;
    var diff0, diff1, diff2, diff3, diff4, diff5: bool;
    var I_val_const, inv_tau_const, v_thres_const, v_reset_const : real;
    var o_v: bool;
    var o_v_val: real;
let
    useless = false;
    -- Assertions on inputs --
    I_val_const = i_I_val -> pre(I_val_const);
    inv_tau_const = i_inv_tau_val -> pre(inv_tau_const);
    v_thres_const = i_v_thres_val -> pre(v_thres_const);
    v_reset_const = i_v_reset_val -> pre(v_reset_const);
    -- Nodes outputs --
    (o_v, o_v_val, o1) = lif(i_I, I_val_const, i_inv_tau, inv_tau_const, i_v_thres,
    v_thres_const, i_v_reset, v_reset_const);
    o2 = behavior(useless);
    -- Observation window --
    diff0 = true -> pre(diff1);
    diff1 = true -> pre(diff2);
    diff2 = true -> pre(diff3);
    diff3 = true -> pre(diff4);
    diff4 = true -> pre(diff5);
    diff5 = not(o1 = o2);
    -- Property --
    NOTSIMILAR = diff0 or diff1 or diff2 or diff3 or diff4 or diff5;
    --%MAIN;
    --%PROPERTY NOTSIMILAR;
tel
```

We called the property to verify `NOTSIMILAR`, and the `--%` is a Kind2 command to specify the property. Instead of directly linking the observer inputs to the neuron inputs like in the previous experiment, we use local variables : in the assertion part, these local variables are initialized at the first time step with the observer input values, then they keep the same value for the next time steps. The property `NOTSIMILAR` is simply the negation of the property `SIMILAR` as we show in example 4.2.1.

Example 4.2.1 – If `SIMILAR` is the property that means "the two outputs `o1` and `o2` are similar during t_k time steps", then it can expressed as:

$\texttt{SIMILAR} = (\texttt{o1} = \texttt{o2})(t_0)$ *AND* $(\texttt{o1} = \texttt{o2})(t_1)$ *AND* ... *AND* $(\texttt{o1} = \texttt{o2})(t_k)$

Therefore the negation of `SIMILAR`, called `NOTSIMILAR`, can be expressed as:

$\texttt{NOTSIMILAR} = \texttt{not}(\texttt{o1} = \texttt{o2})(t_0)$ *OR* $\texttt{not}(\texttt{o1} = \texttt{o2})(t_1)$ *OR* ... *OR* $\texttt{not}(\texttt{o1} = \texttt{o2})(t_k)$

To memorize whether the property is satisfied or not at each time step i, we use local boolean variables `diff`i : `diff`i is true when "$(\texttt{o1} \neq \texttt{o2})(t_i)$", and false when "$(\texttt{o1} = \texttt{o2})(t_i)$".

One result is given in figure 4.16, with Δt_5. As we can see, *Kind2* doesn't approve the property and gives as counterexample the set of input values of the LIF node that disproves it. Indeed, when simulating the neuron node with these values, it outputs the same sequence as the behavior node, but only for the first 6 time steps, after they become different. It makes sense since we chose Δt_5. When we try to extend the observation window or number of time steps, *e.g.* Δt_{10} to contain 2

consecutive spikes in the behavior node, we reach the limits of *Kind2* : The model checker just runs infinitely on a computer *Dell Intel Core i7-11850H at 2.50 Ghz and 15.4 Go of RAM*.

```
Analyzing observer
  with First top: 'observer'
            subsystems
              | concrete: lif, behavior

<Failure> Property NOTSIMILAR is invalid by bounded model checking for k=5 after 0.265s.

Counterexample:
  Node observer ()
    == Inputs ==
    i_I             false false false false false false
    i_I_val          1/32     0     0     0     0     0
    i_inv_tau       false false false false false false
    i_inv_tau_val        4     0     0     0     0     0
    i_v_thres       false false false false false false
    i_v_thres_val        2     0     0     0     0     0
    i_v_reset       false false false false false false
    i_v_reset_val        1     0     0     0     0     0
    == Outputs ==
    NOTSIMILAR       true  true  true  true  true false
    == Locals ==
    diff0            true  true  true  true  true false
    diff1            true  true  true  true false false
    diff2            true  true  true false false false
    diff3            true  true false false false false
    diff4            true false false false false false
    diff5           false false false false false false
    o1              false false false false  true false
    I_val_const      1/32  1/32  1/32  1/32  1/32  1/32
    inv_tau_const       4     4     4     4     4     4
    v_thres_const       2     2     2     2     2     2
    v_reset_const       1     1     1     1     1     1
    o2              false false false false  true false
    useless         false false false false false false
```

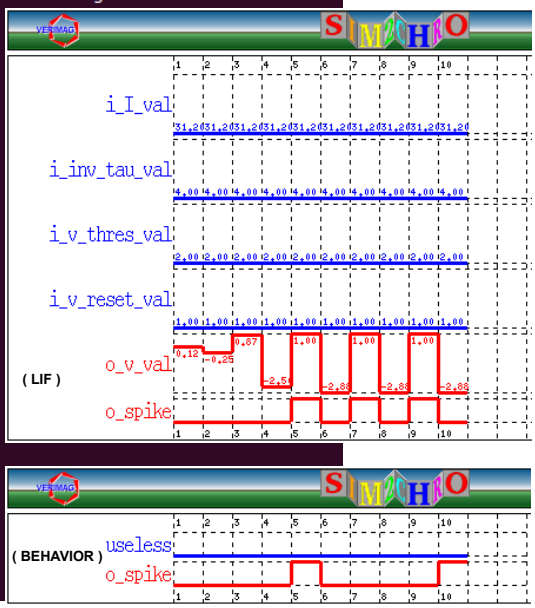
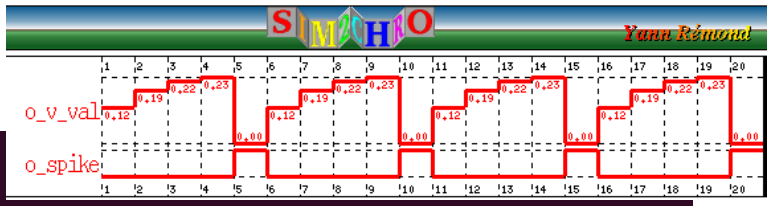


Figure 4.16: *Results of Kind2 parameters exploration with the LIF neuron model, considering Δt_5. On the right, the simulations of the LIF node with the counterexample input parameters compared to the behavior node.*

We succeed to find more adequate parameters to obtain exactly the desired behavior even with Δt_5, by adding more assertions in the observer node. It aims to reduce the domains for the parameters exploration for *Kind2*, but it is possible only when the user imposes the values or knows the intervals of the parameters due to some requirements for example. Nevertheless, with this method the simulation is still required to verify if the desired neuron behavior is obtained with the given set of parameters. An example is shown in figure 4.17, where we added the following assertions in the observer node 4.2 : `i_inv_tau` value must be in the range $]0.0; 1.0[$ and `i_v_reset` is fixed to 0.0. The neuron node simulation with the given counterexample set of parameters is similar to the desired behavior and it shows that different set of parameters is possible.

```
...
-- Additional assertions --
assert i_inv_tau_val > 0.0;
assert i_inv_tau_val < 1.0;
assert i_v_reset_val = 0.0;
...
```




```
Analyzing observer
  with First top: 'observer'
              subsystems
                | concrete: lif, behavior

<Failure> Property NOTSIMILAR is invalid by bounded model checking for k=5 after 0.153s.

Counterexample:
  Node observer ()
    == Inputs ==
    i_I             false  false  false  false  false  false
    i_I_val           1/4      0      0      0      0      0
    i_inv_tau       false  false  false  false  false  false
    i_inv_tau_val     1/2    1/2    1/2    1/2    1/2    1/2
    i_v_thres       false  false  false  false  false  false
    i_v_thres_val 61/256      1      1      1      1      1
    i_v_reset       false  false  false  false  false  false
    i_v_reset_val       0      0      0      0      0      0
    == Outputs ==
    NOTSIMILAR       true   true   true   true   true  false
    == Locals ==
    diff0            true   true   true   true   true  false
    diff1            true   true   true   true  false  false
    diff2            true   true   true  false  false  false
    diff3            true   true  false  false  false  false
    diff4            true  false  false  false  false  false
    diff5           false  false  false  false  false  false
    o1              false  false  false  false   true  false
    I_val_const       1/4    1/4    1/4    1/4    1/4    1/4
    inv_tau_const     1/2    1/2    1/2    1/2    1/2    1/2
    v_thres_const 61/256 61/256 61/256 61/256 61/256 61/256
    v_reset_const       0      0      0      0      0      0
    o2              false  false  false  false   true  false
    useless         false  false  false  false  false  false
```

Figure 4.17: *Results of Kind2 parameters exploration with the LIF neuron model, considering* Δt_5*. Assertions were added on* `i_inv_tau` *and* `i_v_reset` *:* $0.0 <$ `i_inv_tau` < 1.0 *and* `i_v_reset` $= 0.0$*. On top right, the simulation of the neuron LIF with the counterexample set of parameters.*

When applied to the more complex IZH neuron model, the limits of *Kind2* are again reached : the number of time steps upon which the model checker needs to validate the neuron and the behavior outputs is too long, the number of parameters and the neuron equations involve nonlinear arithmetic.

4.2.3 Limits

Kind2 is a model checker that provides an efficient way to automatically check the correctness of a system's model, ensuring that the model satisfies specific properties or requirements. Although we have successfully utilized the model checker in some of our experimental cases, we have also demonstrated that *Kind2* has its limitations. It did not meet our objectives in our initial experiments with reduced cases before moving on to more complex neural network experiments. Our findings indicate that the limitations of *Kind2* are related to the types of models and properties it can handle. Some of these potential limitations include:

- Complexity of models: *Kind2* may struggle with models that are very large or complex, or those with lengthy execution times. The verification process for a large or complex model may take a long time, or it may not be able to complete the verification process at all;
- Non-linear arithmetic: *Kind2* currently does not support the use of non-linear arithmetic in models;
- Finite state space: *Kind2* can only handle models with a finite state space, meaning the number of possible states of the system is limited.

Nevertheless, despite these limitations, *Kind2* remains one of the best model checkers currently available as presented in table B.3.

4.3 Conclusion

In this chapter, we explained our approach for implementing neural models based on differential equations and spike events using the *Light Esterel* synchronous language. We presented the neural models we implemented and described our validation method for the initial models in both software and hardware. As a novel application for the *Light Esterel* environment, we also discussed the updates we made to the compilation environment, enabling the design and simulation of spiking neural networks (SNNs). Although the language is inherently suited for implementing neural models based on differential equations, these updates were essential for creating and simulating SNNs.

However, we encountered several limitations:

1. At the time this thesis was written, the *Light Esterel* compilation tool was constrained in terms of the size of SNNs that could be compiled, with a maximum of 12 neurons and 144 synapses.
2. The SNN hardware code generation was not optimized for hardware platforms with limited resources, such as FPGAs.
3. The hardware division operator had not yet been implemented in hardware, so neural models with division operations, like the HH model, could only be simulated in software.

We chose *Light Esterel* for designing SNNs to leverage automatic provers for extracting or highlighting biological neural properties. Inspired from related works on SNNs, we initially tested the Kind2 model checker on a single neuron rather than an entire neural network. However, we quickly reached the tool's limits due to either the complexity of the neuron model or the properties to be checked. Although other tools and techniques are available, as formal verification is a broad research area applied in various academic and industrial settings, further exploration is needed. Our preliminary results do not close the possibility of using formal methods on SNNs.

The limitations discussed in this chapter served as a critical turning point in our research. Throughout the years of this thesis, we attempted to address each limitation, but time constraints led us to prioritize finding a solution to simulate larger SNNs. This objective was driven by our goal to communicate with biological neurons. Consequently, we decided to design a neuromorphic architecture that could simulate larger SNNs while maintaining compatibility with the previously developed *Light Esterel* neural models. In the following chapter, we provide a detailed description of the hardware architecture we developed for this purpose.

CHAPTER 5

SynchNN : Neural Processing Unit implementation

In this chapter, we address the limitations encountered when using native Light Esterel tools for generating SNNs for software and hardware simulations, discussed in the previous chapter. These limitations led us to develop our own hardware architecture, which is the main focus of this chapter.

We discuss the motivations behind developing our own architecture and the strategies we adopted, to overcome the previous limitations and to address the application constraints in the context of this thesis.

We then provide a detailed description of our hardware architecture, SynchNN, highlighting its various modules and units. The architecture is designed to be flexible and efficient.

Furthermore, we present the validation results of our architecture, demonstrating its effectiveness in simulating and executing SNNs. We discuss the performance of the graph coloring algorithms implemented for the configuration of SynchNN.

Finally, we discuss the future perspectives and potential enhancements for our architecture, highlighting areas for further research and development.

As discussed in section 2.3.2, existing neuromorphic architectures proposed by industry and academia primarily target machine learning applications such as classification or machine vision. These architectures prioritize the simulation of large-scale SNNs in terms of the number of neurons and synapses, often at the expense of flexibility in simulating biomimetic neural models and incorporating various biological properties.

To overcome the limitations of the *Light Esterel* compilation environment and tools (as described in section 4.1.2) and to address the requirements of neurobiohybridation experiments, as explained in the following sections, we have developed our own neuromorphic architecture. This decision not only allows us to continue using *Light Esterel* neural models developed in section 4.1.1, but provides us with complete control over an hardware architecture, enabling us to modify and integrate new features in the future to fit our research objectives.

Time scale constraint

In the context of this work, neurobiohybrid experiments require the artificial part to be able to update the states of all neurons and synapses within a 1 millisecond (ms) time-step (Ambroise, 2015; R. M. Wang, Thakur, & Van Schaik, 2018; Merolla et al., 2014). This time-step represents the minimum sampling period necessary to capture the firing behavior of neural cells, as a neuron can only generate one action potential within 1 ms (Buccelli et al., 2019). Although the development process for hardware platforms can be time-consuming, they offer the advantage of processing at a fast rate, which allows to meet the 1 ms time scale constraint. Note that the time-step is a parameter in our architecture, and it is configurable.

Parallel and time-multiplexed hardware paradigms

In the initial compilation environment of *Light Esterel* (see section 3.6), the hardware generation consists of implementing each neuron and synapse individually. This means that a *neuron module*, described by a set of equations in hardware, is duplicated for each neuron in the network. Similarly, a *synapse module* is duplicated for each synapse. While these modules execute in *parallel* similar to the structural organization in the brain, this approach is rapidly confronted to the limit of available resources on the FPGA.

An alternative approach is to *time-multiplex* the neuron and synapse modules. This leverages the fact that an artificial neuron executes faster than a biological neuron (Mahowald & Douglas, 1991), therefore a single neuron module can be used to compute the state of multiple neurons in the network (Cassidy, Andreou, & Georgiou, 2011; R. Wang, Hamilton, Tapson, & van Schaik, 2014).

In our hardware architecture, we have adopted a strategy that combines parallel processing and time-multiplexing. Each Neural Processing Unit (NPU) consists of a neuron module and a synapse module, and is responsible for processing a specific region of the SNN. This approach allows to time-multiplex the neurons and synapses within a region, while simultaneously processing other regions in parallel using separate NPUs.

Time-multiplexing and parallelism enable acceleration, ensuring compliance with the time-step constraint and allowing for the simulation of a greater number of neurons and synapses. Consequently, our strategy requires the exploration of methods to partition the neural network into regions or groups of neurons, where each group is assigned to a specific NPU.

5.1 Methods for neural network partitioning

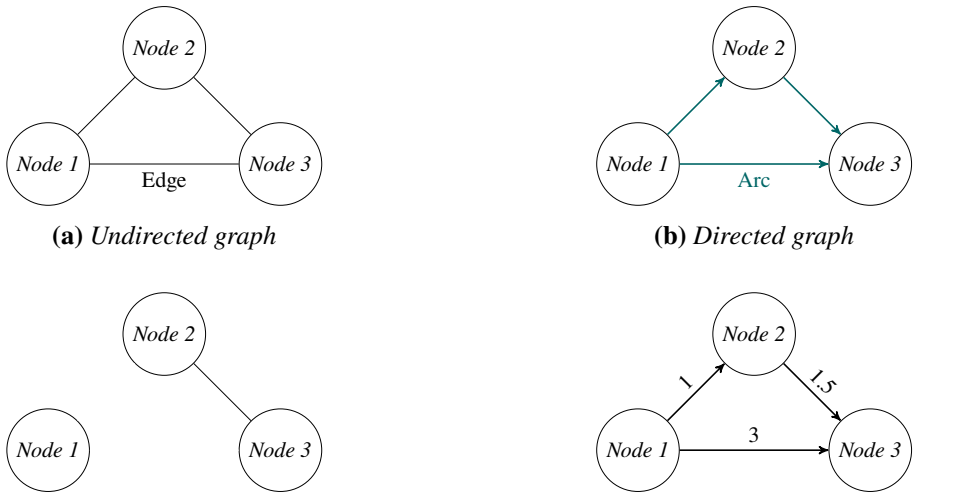
As part of a second-year master's internship that I supervised, we investigated and tested various methods for partitioning neural networks (Chouchane, 2020). In this process, we explored the field of graph theory, which encompasses a wide range of studies related to data representation and partitioning. We will provide an introduction to graph theory to provide context to understand our specific approach, but note that our coverage of the graph theory will be non-exhaustive.

5.1.1 Introduction to Graph theory

Graph theory is a mathematical field that studies graphs, which are structures used to model pairwise relationships between objects. A graph consists of *nodes* (also called "vertices" or "points") that are connected by *edges* (also called "links" or "lines") as illustrated in figure 5.1.

In general, a graph G is represented as an ordered pair $G = (V, E)$, where :

- V is a set of nodes;
- $E \subseteq \{\{x, y\} | x, y \in V \text{and } x \neq y\}$ is a set of edges, which are the connected pairs of nodes.



(a) *Undirected graph*

(b) *Directed graph*

(c) *Disconnected graph*

(d) *Weighted connected directed graph*

Figure 5.1: *4 types of graph : nodes are represented by circles and edges are either lines or arrows. (a) shows an Undirected graph, (b) represents a directed graph, where edges are arrows showing the direction of the information between two nodes. (c) shows a Disconnected graph, where nodes or groups of nodes do not have connections or communication with other nodes. (d) is a combination of type (b) and (c), and additionally, edges are labeled with values.*

Separation problems are a significant area of research in graph theory (Pardalos, Mavridou, & Xue, 1998; Bader, Meyerhenke, Sanders, & Wagner, 2013). We chose to explore this domain as there are notable similarities between graph representation and neural network representation. Nodes can be considered as neurons, and edges as synapses. Graph theory includes various types of graphs (Mondal & De, 2017) as shown in figure 5.1, and in our context, we are particularly focusing on one specific type known as **weighted connected directed graph type** in figure 5.1(d).

A weighted connected directed graph has the following characteristics :

- *Weighted* : each edge is assigned with a value. In the context of neural networks, synapses are defined by weights or a transmission delay, or both;
- *Connected* : there are no isolated nodes or groups of nodes. In the context of neural networks, we focus on neural network where no neurons or groups of neurons are isolated from the global network;
- *Directed* : there is a unidirectional flow of data between pairs of nodes, specified by an arrow. In the case of neural networks, a spike travels from a pre-synaptic neuron to a post-synaptic neuron only.

5.1.2 Two main partitioning approaches

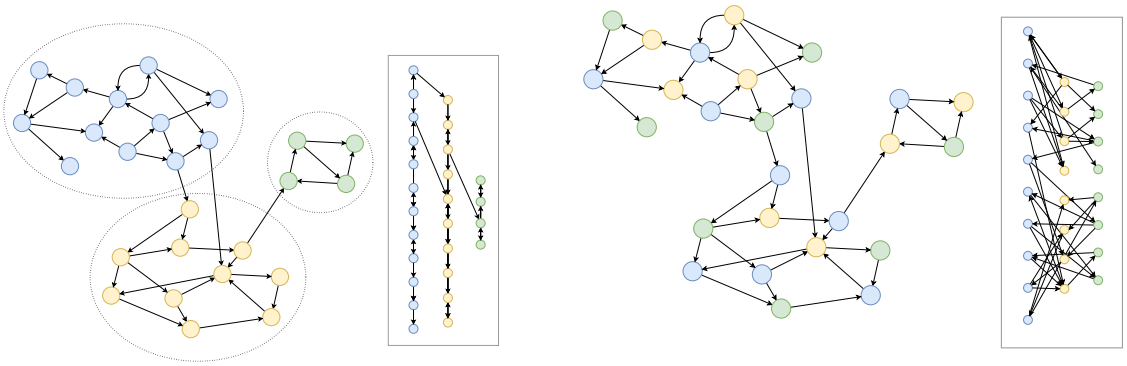
Based on these characteristics, we explored related graph techniques to separate such a graph. And there are two main approaches to divide a weighted connected directed graph into subgraphs: *clustering* separation and *coloring* separation.

Clustering approach

Clustering is a widely used technique in graph analysis that aims to group nodes based on their similarities or proximity. Various clustering algorithms have been developed and applied in different fields. For example, k-means clustering is a popular algorithm that partitions nodes into k clusters based on their distances to cluster centroids (Jain, Murty, & Flynn, 1999). Hierarchical clustering, on the other hand, creates a hierarchy of clusters by iteratively merging or splitting clusters based on similarity measures (Jain & Dubes, 1988). Spectral clustering utilizes the eigenvectors of a similarity matrix to partition nodes into clusters (Ng, Jordan, & Weiss, 2001). These clustering techniques have been successfully applied in various domains, including social network analysis (Fortunato, 2010), image segmentation (Shi & Malik, 2000), etc. By identifying meaningful groups within a graph, clustering provides valuable insights into the underlying structure and organization of complex systems.

Coloring approach

Graph coloring assigns "colors" or labels to the nodes of a graph in such a way that adjacent (or connected) nodes have different colors. This problem has applications in various domains, including task scheduling, timetable planning, resource allocation, map coloring, and more (Jensen & Toft, 2011). For example in task scheduling, graph coloring can be used to assign different time slots (colors) to tasks (nodes) in order to ensure that no conflicting tasks are scheduled at the same time.



(a) *Clustering principle* **(b)** *Coloring principle*

Figure 5.2: *Clustering and coloring methods applied on the same graph. (a) Left, clustering results in 3 clusters : most connected and closest nodes are grouped together. Colors were added to differentiate the clusters. Right, reorganized clusters into vertical layers : 1 layer is 1 cluster. (b) Left, coloring results in 3 colors or groups : nodes that are directly connected have different color. Right, reorganized groups into vertical layers : 1 layer is 1 color.*

Selected approach : coloring

In the context of neural networks, the clustering paradigm limits the connections between clusters of neurons (or inter-group communications), while it increases density of connections within each cluster (or intra-group communications) as illustrated in figure 5.2(a). On the other hand, the coloring paradigm prohibits communications between neurons in the same group, resulting in only inter-group communications as illustrated in figure 5.2(b).

During the early development stages of *SynchNN*, the choice of the partitioning method was influenced by the complexity of the related hardware development. The clustering paradigm requires the hardware to support both inter-group communications between NPUs and intra-group communications within each NPU. In contrast, the coloring paradigm only requires the hardware to support inter-group communications between NPUs.

Furthermore, the layered structure of partitioned colored graphs looks similar to the layered structure of fully-connected feedforward neural networks. In (Abderrahmane et al., 2020), the authors explored various architecture designs for executing fully-connected SNNs on hardware platforms, specifically FPGAs. They compared different designs based on latency, power consumption and resource allocation. Their results show that allocating resources to process each layer by one NPU in the case of fully-connected model can be an interesting choice. Therefore, our intuition led us to adopt the coloring paradigm for our partitioning approach.

5.1.3 Coloring algorithms

In our study, we have chosen Greedy Graph Coloring algorithms, characterized by the careful selection of the next node to be colored (Al-Omari & Sabri, 2006), which significantly impacts the separation solution. The 3 selected algorithms to address our problem are First-Fit (FF), Largest Degree Ordering (LDO) and Welsh-Powell (WP). These algorithms are simple and fast, but they may not be the most optimized amongst existing algorithms. It is important to note that our

partitioning step is not limited to these three coloring methods and can be further optimized in the future, even with completely different approaches.

First-Fit (FF)

FF algorithm (Gyárfás & Lehel, 1988) is a simple and straightforward coloring algorithm. First, it sorts the nodes by their IDs. Then, each node is selected in order, and the algorithm assigns it the smallest available positive integer. In this context, an integer represents a color, and an "available" integer refers to a color that is not assigned to any of the adjacent nodes of the current node. The FF algorithm is described in algorithm 5.1, and a coloring example is shown in figure 5.3. The algorithm can be implemented to run in $O(n)$ (Klotz, 2002) but the coloring result strongly depends on the nodes order.

Example 5.1.1 – In figure 5.3, the FF coloring is applied on a graph composed of 8 nodes randomly connected. Nodes are ordered in a table. (a) The first color blue is applied to the first node in the table, node 1. Then the next node in the table, node 2, is selected for coloring. (b) As node 2 adjacent nodes are uncolored, it can be assigned with blue too. (c) Next, as node 3 is connected to node 1 and node 1 is already colored in blue, therefore a new color yellow is created for node 3. (d) All nodes are processed in order based on the table, and the algorithm gives 3 colors out of the graph.

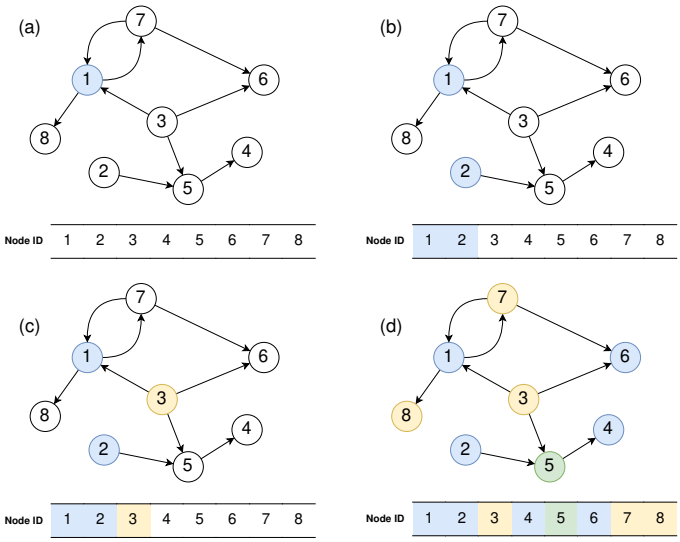


Figure 5.3: *Example of First Fit coloring.*

Algorithm 5.1: First-Fit algorithm

Step 1 :
Order nodes by their IDs in a table *ordered_nodes_tab* by their IDs; *// no node is colored yet, ordering by IDs*

Step 2 :
Create an empty list of *available_colors*;
// e.g. available_colors = {{blue;true}, {yellow;false},...} listing all the created colors and their availability (true or false)

Step 3 :
Create a new *color*;
Add it to *available_colors* as available;
Assign it to the first node in *ordered_nodes_tab*;

Step 4 :
repeat
  Select next *node_i* in *ordered_nodes_tab*;
  foreach *adjacent node_k of node_i* **do**
    if *node_k is colored* **then**
      Mark *node_k*'s *color* as unavailable in *available_colors*;
  if *all colors are unavailable* **then**
    Create a new *color*;
    Add it to *available_colors*;
    Assign it to *node_i*;
  else
    Assign the first available *color* in *available_colors* to *node_i*;
  Reset all colors in *available_colors* to available;
until *All nodes are colored*

Largest Degree Ordering (LDO)

The LDO algorithm (Al-Omari & Sabri, 2006) follows a different approach by ordering the graph nodes based on their degrees. The degree of a node refers to the number of adjacent nodes it is connected to. In each step of the algorithm, the uncolored node with the largest degree is selected. The color for this node is then determined using the same process as the FF algorithm, which involves checking the colors of its adjacent nodes and assigning an available color. Intuitively, the LDO algorithm tends to produce better colorings compared to the FF algorithm due to the specific node selection order. The time complexity of the LDO algorithm is typically $O(n^2)$ (Klotz, 2002). For a more detailed description of the LDO algorithm, please refer to algorithm 5.2, and an example of its coloring result is illustrated in figure 5.4.

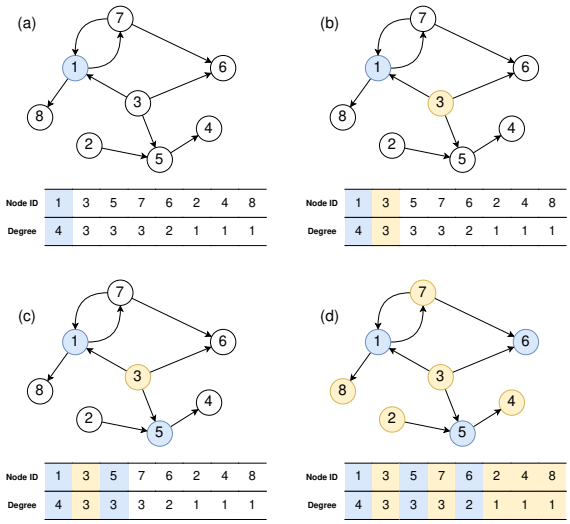


Figure 5.4: *Largest degree ordering coloring example on a graph composed of 8 nodes randomly connected. (a) Nodes are ordered in a table according to their degrees. The first color blue is applied to the node with the largest degree which is node 1. (b) Next, node 3 in the table order is selected. It's adjacent to an already colored blue node, a new color yellow is then created and assigned to node 3. (c) Next, node 5 is connected to the colored node 3 but it can have the blue color. (d) All nodes are processed in the table order, and the algorithm gives 2 colors out of the graph.*

Algorithm 5.2: Largest Degree Ordering algorithm

Step 1 :
Sort the nodes according to their degrees in a table $ordered\_nodes\_tab$ from largest to smallest; *// ordering by degrees*

Step 2 :
Create an empty list of $available\_colors$;

Step 3 :
Create a new $color$;
Add it to $available\_colors$;
Assign it to the first node in $ordered\_nodes\_tab$;
// the node with the largest degree receives the first color

Step 4 :
same as step 4 in FF algorithm;

Welsh-Powell (WP)

The WP coloring algorithm (Welsh & Powell, 1967), as the LDO, uses an ordered nodes table based on their degrees in the graph: from the largest to the lowest. It first assigns one color to the uncolored nodes with the largest degree in the table. Then it selects in the table order, all the uncolored nodes that can be assigned with the same color. Once the color can't be applied to anymore nodes, a new color is created and assigned to the next uncolored node with the largest degree. The process is repeated until there is no uncolored nodes left. The algorithm is detailed in algorithm 5.3, and a coloring example is shown in the figure 5.5.

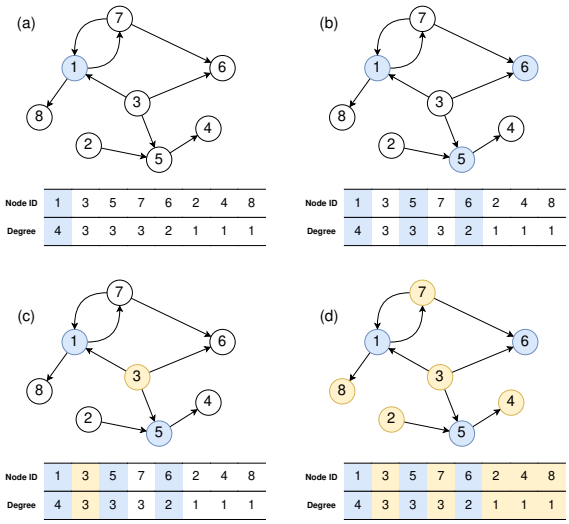


Figure 5.5: *Welsh-Powell coloring example on a graph composed of 8 nodes randomly connected. (a) Nodes are ordered in a table according to their degrees. The first color blue is applied to the node with the largest degree which is node 1. (b) Next, each node in the table order are checked if it can receive the color blue. (c) If the blue color can't be assigned anymore, a new color yellow is affected to the next uncolored node in the table order which is node 3. The processes (b) and (c) are repeated until all nodes are colored. (d) All nodes are processed in the table order, and the algorithm gives 2 colors out of the graph.*

Algorithm 5.3: Welsh-Powell algorithm

Step 1 :
  Sort the nodes according to their degrees in a table $ordered\_nodes\_tab$ from largest to smallest; *// ordering by degrees*

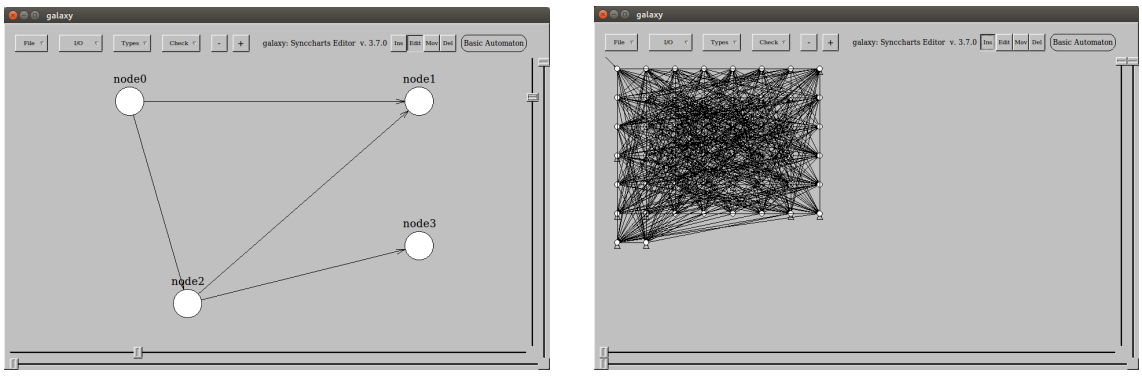
repeat
  Step 2 :
    Select a new $color$ and assign it to the uncolored node with the largest degree in $ordered\_nodes\_tab$;
  Step 3 :
    repeat
      Select next largest degree uncolored node in $ordered\_nodes\_tab$;
      Assign actual $color$ if possible;
    until $color$ *can no longer be assigned to another node*

until *all nodes are colored*

5.1.4 Coloring partitioning selection methods

A comparison was conducted in (Aslan & Baykan, 2016) between performances of well-known coloring algorithms based on the coloring quality and the execution time. A good coloring performance is characterized by minimizing the number of colors while ensuring fast coloring execution. In our context, the number of colors directly corresponds to the number of NPUs to be implemented, making a lower number preferable to conserve hardware resources. However, the coloring execution time is not a significant concern for us, as this partitioning step is a pre-processing step in our approach and does not affect the overall hardware performance. Additionally, we considered another criteria in our partitioning selection process, which is the variation in the number of nodes assigned to each color. It is preferable to select coloring solutions that exhibit a homogeneous distribution of nodes among each color, in order to prevent over-utilization or under-utilization of NPUs.

To evaluate the coloring algorithms previously presented, we used two software tools developed at our laboratory : *Galaxy* (Gaffé, 2022) and *Bing Bang*. These tools are built upon libraries designed for manipulating graphs and automata. *Galaxy* serves as a graphical editor for manually creating and visualizing state machines or graphs, while *Bing Bang* is an additional module used to generate random graphs with customizable characteristics, such as the number of nodes and the density of connections. The connection density determines the number of edges per node. Note that the *Bing Bang* tool serves as the basis of the *create_gln* tool, which is used to generate random networks and to generate the *gln* specification file (see section 4.1.3). We employed both software tools, *Galaxy* and *Bing Bang*, to generate custom or random graphs, as illustrate in figure 5.6, in order to test and evaluate the detailed coloring algorithms discussed earlier. The evaluation results can be found in Section 5.3.



(a) *Custom graph* **(b)** *Random generated graph*

Figure 5.6: *Galaxy interface. (a) An example of customable graph with 4 nodes and 4 edges. (b) An example of random generated graph from Bing Bang and visualized on Galaxy, 50 nodes and 30% of connections per node.*

Now that we have discussed the reasons behind developing our own hardware architecture and the chosen strategies for simulating and executing SNNs, let's proceed to describe in detail the architecture that we have been developed.

5.2 Neural Processing Unit

In this section, we present the neuromorphic architecture we developed, called *SynchNN*. The architecture is illustrated in figure 5.7. It consists of multiple units, which can be categorized into two groups. The first group, referred to as the NPU, is duplicated as needed based on the pre-processings steps that we will explain in the following sections. It includes the "RECEPTION UNIT", the "PROCESSING UNIT" and the "CONTROLLER UNIT". The second group consists of the "BROADCASTING UNIT" and the "TIME STEP MANAGER UNIT", which are generated only once.

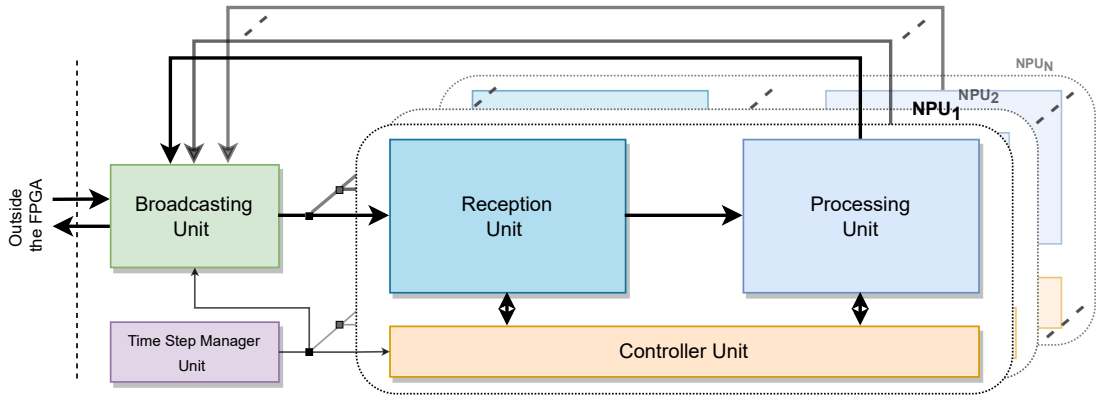
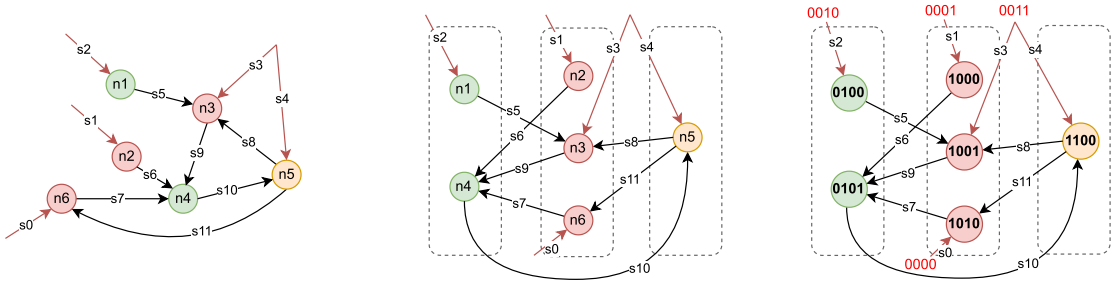


Figure 5.7: *Global architecture of SynchNN. The configuration of SynchNN such as the number of NPUs to be generated, is accomplished using configuration files that are automatically generated from the Genlenet tool (see section 4.1.3).*

- The **RECEPTION UNIT** receives successively neuron identifications (neuron IDs) from the BROADCASTING UNIT. It is a ID-to-spikes converter and includes the delaying of the spikes. It also indirectly handles the matrix connection between neurons outside the NPU and the neurons within the NPU. The converted spikes are then directed to the PROCESSING UNIT.
- The **PROCESSING UNIT** receives the spikes from the RECEPTION UNIT, and performs accordingly the updates of all the synapses and neurons at each time step. When a neuron spikes, its neuron ID is sent to the BROADCASTING UNIT.
- The **CONTROLLER UNIT** orchestrates the RECEPTION and the PROCESSING UNITS functions.
- The **BROADCASTING UNIT** receives neuron IDs from the PROCESSING UNITS and from external devices (outside the FPGA). His role is to broadcast sequentially the IDs to all the NPUs.
- The **TIME STEP MANAGER UNIT** sends a signal "new time step" to the other units, to initiate their processing at each new time step. Note that due to the leakage mechanisms or axonal delays, all units are executed every time step.

We will provide a detailed description of each unit, along with their respective modules and functions. To facilitate understanding, we will also provide an illustrative example of a simple SNN which coloring is shown in figure 5.8. This example will serve as a visual representation of our explanations on the concepts and processes of the architecture.



(a) *Coloring* **(b)** *Neurons reorganization* **(c)** *Definition of the neuron IDs*

Figure 5.8: *SNN example of 6 neurons labeled with "n", 12 synapses labeled with "s" including 7 internal (black arrows) and 5 external (red arrows). (a) The coloring of the SNN topology gives 3 colors. (b) Reorganization of the SNN in layers, so each layer represent one color or one group. (c) Each neuron is assigned a global neuron ID. The first bits on the left are the NPU ID (or color ID) the neuron belongs to. External neurons belongs to a "virtual" NPU which ID is 0. The right side is the local index or local ID of the neuron within a group.*

5.2.1 Pre-processings to configure *SynchNN* from ".gln" file

There are pre-processings that are executed to configure *SynchNN*. It consists of 3 main pre-processings applied during the compilation of the specification ".gln" file (see section 4.8). First one is the partitioning coloring step that determines how many NPUs to generate and which neurons to assign to them. The method of this step has been explained in previous section 5.1.3. The second one consists of assigning a unique neuron ID for each neuron. Finally, the third one involves the extraction of the connectivity matrix for each NPU.

Neuron ID assignment

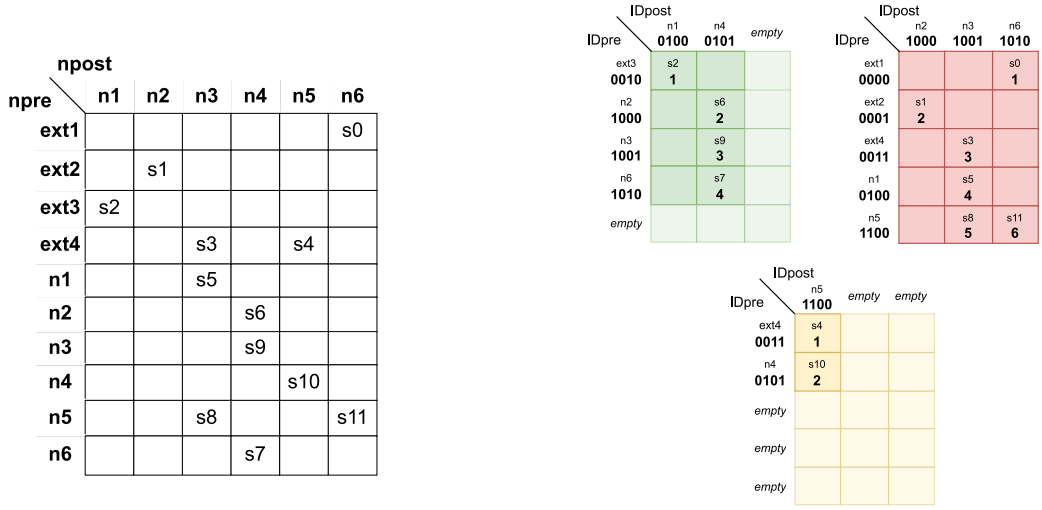
A unique neuron ID is assigned to "external" and "internal" neurons. This neuron ID has the same length for all neurons (see figure 5.8(c)). The neuron ID is obtained by concatenating the ID of the NPU to which the neuron is assigned and its local ID within that NPU. Consequently, the length of the neuron ID depends on two factors : the total number of NPUs and the maximum number of neurons grouped among all the NPUs.

The inputs of *SynchNN* are "events" or "spikes" coming from an outside system to address specific neurons within the SNN. From the perspective of the architecture, these external spikes come from "external neurons", that reach the SNN through "external synapses". Consequently, it is important to note that "external" neurons are considered to belong to a "virtual" NPU with an ID of 0, it distinguishes them from the neurons within the NPUs.

Example 5.2.1 – In figure 5.8, the coloring of the given neural network results in 3 groups. With the "virtual" group, the total number of groups is 4. So 2 bits will be used to code the NPU IDs : "00" for the "virtual" NPU, "01" for NPU1, "10" for NPU2 and "11" for NPU3. Next, the maximum number of neurons grouped within a color is 3 (color red), so 2 bits will also be used to code the local IDs of all neurons. In total, 4 bits are used to code the global neuron ID that is the concatenation of a NPU ID and the local neuron ID : "10 01" is the 2nd neuron within the NPU 2.

Connectivity matrix extraction for each NPU

The global connectivity matrix is a table that describes the pairs of neurons that are connected in the SNN. The table in figure 5.9(a) shows the global connections of the SNN example from figure 5.8. Elements in the table are the synapse labels. They connect the neurons pre ("npre") within the rows to the neurons post ("npost") within the columns. After the coloring, a connectivity matrix is deduced for each NPU (as shown in figure 5.9(b)). The "npre" and "npost" are replaced by their neuron IDs. And finally, the synapse labels are replaced by integers.



(a) *SNN connectivity matrix* **(b)** *NPUs connectivity matrices*

Figure 5.9: *Connectivity matrices of the example in figure 5.8. Rows are the pre-synaptic neurons (npre), and columns are the post-synaptic neurons (npost). (a) Connectivity matrix of the original SNN. (b) Connectivity matrices for the colored groups.*

5.2.2 RECEPTION UNIT

The RECEPTION UNIT has two main functions. First one, it converts neuron IDs coming sequentially from the BROADCASTING UNIT into a vector of spikes. Second one, it incorporates the modelling of axonal delay related to each synapse associated to the NPU. To ensure these functions, the RECEPTION UNIT is composed of 2 modules, as illustrated in figure 5.10 : a "NEURON ID DECODER" and multiple "SHIFT REGISTERS".

The inputs are the neuron ID and the state of the FIFO the neuron ID comes from, *i_neurID* and *i_fifo_empty* respectively. These signals come from the BROADCASTING UNIT. Other inputs are the signal to enable the shift of the registers and the signal to reset the output of NEURON ID DECODER to '0', *i_shiftRegs_en* and *i_rst_spikes* respectively. These signals come from the CONTROLLER UNIT. The output spikes that are transmitted to the PROCESSING UNIT represent the spikes that have traversed the entire array of shift registers or axonal delay lines. Note that the *o_neurID_valid* is a just a debug signal to verify if the input neuron ID is valid and can be converted.

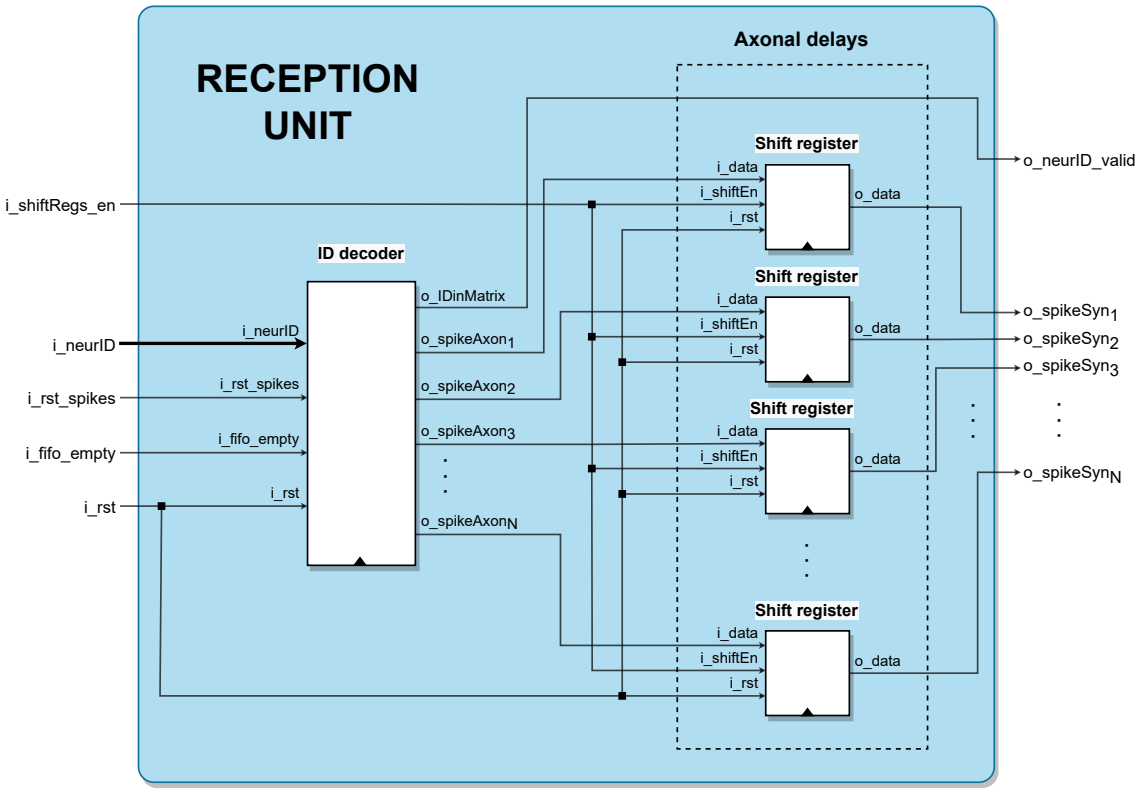


Figure 5.10: *Architecture of the* RECEPTION UNIT.

Neuron ID decoder

The NEURON ID DECODER is responsible for the conversion of incoming neuron ID into a vector of spikes. Its main role is to determine the specific post synapses within the NPU that are targeted by the neuron ID. It consists of 3 modules arranged as illustrated in figure 5.11. These modules are the "NEURON ID TO ADDRESS CONVERTER", the "SPIKES ROM" and the "SPIKES STORAGE".

The inputs are the *neuron ID*, a *i_fifo_empty* signal and a *i_rst_spikes* signal. When '0', the *i_fifo_empty* indicates that the received ID can be converted, otherwise the ID is ignored. The *i_rst_spikes* signal is to reset the output spikes of the module.

- NEURON ID TO ADDRESS CONVERTER : this module converts the neuron ID input into an address for the SPIKES ROM module. It checks whether the received neuron ID belongs to list of pre-neurons in the connectivity matrix of the NPU. If the neuron ID is not found in the list, the module sets the signal *o_IDinMatrix* to '0', indicating that the neuron that emitted the spike is not connected to any neurons within the NPU. On the other hand, if the neuron ID is present in the list, it generates an address that corresponds to the row position of this neuron ID in the NPU connectivity matrix.

  Example 5.2.2 – In figure 5.9(b), if the NPU1 (green color) receives the neuron ID "1100" (*n5*), the ID will be ignored because it is not part of its pre-IDs list. However, if it receives the neuron ID "1001" (*n3*), then it will output the row position 2.

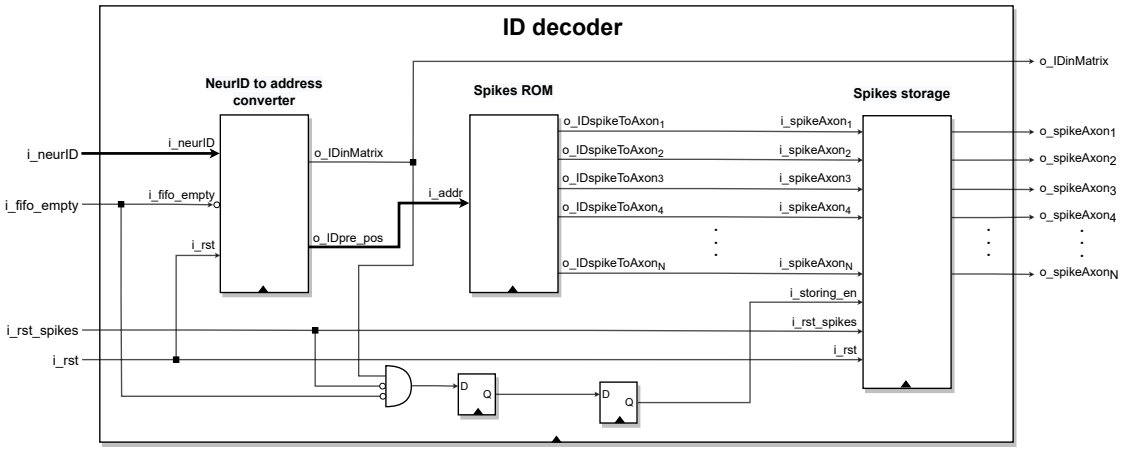


Figure 5.11: *Architecture of the neuron ID decoder module in the reception unit.*

- SPIKES ROM : it is a Read-Only Memory (ROM) where data are initialized once and can only be read and accessed by an address. The SPIKES ROM can be visualized as an array that represents the connections between pre-neurons and post-synapses within the NPU. To generate this array, we transform the NPU's connectivity matrix into an array of '1's and '0's. The number of rows in this array corresponds to the number of pre-neurons, which is the same as the number of rows in the connectivity matrix. The number of columns represents the number of post-synapses.

  For each row in the connectivity matrix, we use the integers within that row as column indexes in the array. We fill '1's at the corresponding row-column positions to indicate a connection between a pre-neuron and a post-synapse. The remaining elements in the array are filled with '0's to indicate the absence of connections. The resulting array is used to initialize the SPIKES ROM.

  This module takes the address of the row position, from the NEURON ID TO ADDRESS CONVERTER module as input. It then outputs the corresponding vector of spikes (a vector of '1's or '0's), stored at that address. Each element in this vector represents the presence or absence of a spike for a specific post-synapse.

  Example 5.2.3 – In figure 5.9(b), for the NPU2 (red color), the number of pre-IDs is 5, the total number of synapses to address is 6. So the size of the memory is 5, and the size of each data is 6 bits. The generated ROM will look like the following table (MSB : Most Significant Bit / LSB : Less Significant Bit) :

SPIKES ROM of NPU2

Address input	Address	Memory data LSB ... MSB	Vector output
2	0	100000	001000
	1	010000	
	2	001000	
	3	000100	
	4	000011	

- SPIKES STORAGE : during the entire time step, the role of this module is to receive and collect the spikes from the SPIKES ROM into internal registers. The "collecting" process involves performing a logical OR operation between the input spikes and the state of the internal registers. The module also has two additional inputs: *i_rst_spikes* and *i_storing_en*. The *i_rst_spikes* input is used to reset the internal registers to '0' when the input signal is '1'. The *i_storing_en* input indicates whether the input spikes array can be stored.

Shift registers : axonal delays

We have chosen to use SHIFT REGISTERS to model the axonal delay. Each synapse is associated with a predetermined axonal delay, which is configured before the generation process and remains fixed thereafter. By setting the general time step to 1 ms, modeling an axonal delay of D ms requires D successive shift registers. If a spike is inputted into this series of shift registers, it will be outputted D ms later if a shift operation is performed every 1 ms. This approach allows for easier management of spike sequences received by post-synapses, compared to a solution involving counters. Finally, we generate a specific series of shift registers for each synapse to minimize resource utilization compared to fixed series of shift registers or memory-based solutions.

5.2.3 PROCESSING UNIT

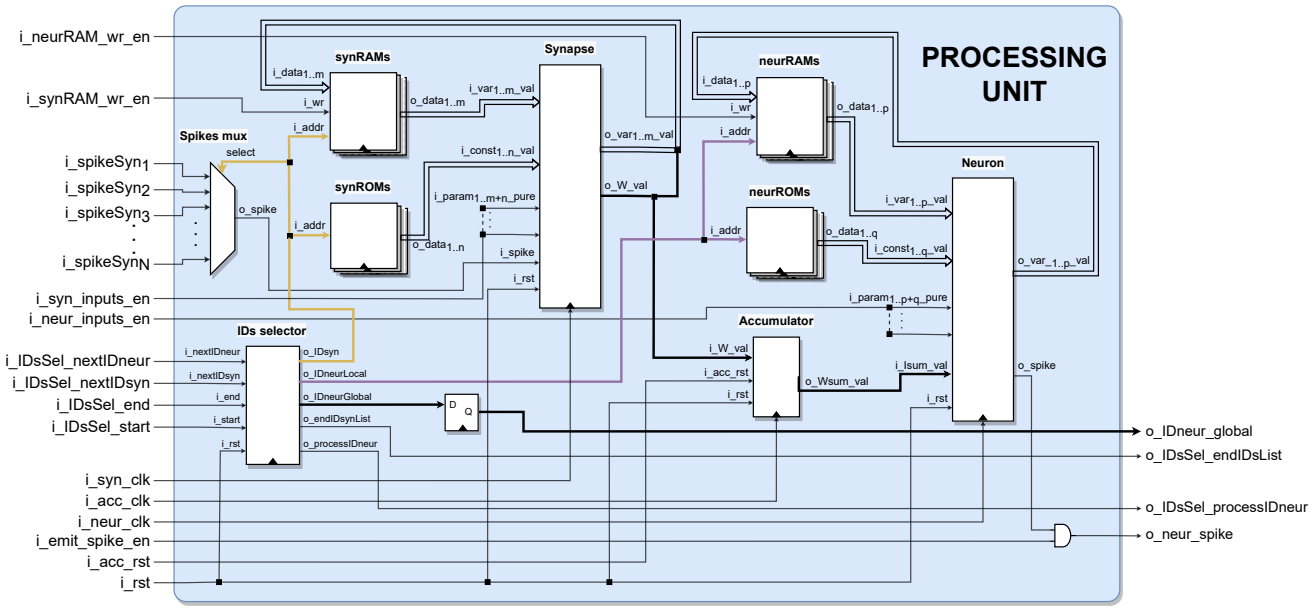


Figure 5.12: *processing unit architecture.*

The main role of the PROCESSING UNIT is to update all the neurons and synapses associated with the NPU. It receives spikes from the RECEPTION UNIT, updates all the synapses, and then updates all the neurons based on the updated synapses. Finally, it emits the neuron IDs of the neurons that have spiked throughout the time step to the BROADCASTING UNIT. To fulfill these functions, the PROCESSING UNIT is composed of several modules and components. These include a "SPIKES MULTIPLEXER", "ROM/RAM MEMORIES", a "SYNAPSE MODULE", a "NEURON MODULE", an

"ACCUMULATOR" and an "IDs SELECTOR". The figure 5.12 shows how these different modules are connected together.

Spikes multiplexer

The SPIKES MULTIPLEXER module takes a vector of spikes as input from the array of SHIFT REGISTERS in the RECEPTION UNIT. Each position in the input vector corresponds to a specific post-synapse and can be either '1' or '0' to indicate the presence or absence of a spike at that post-synapse. The module uses an address selector to choose one of the input spikes as the output. The selection of the address is controlled by the IDs SELECTOR module.

Memories ROM/RAM

We use two types of on-chip memory which are Read-Only Memory (ROM) and Random Access Memory (RAM). These memories are used to store different types of parameters related to the neural models. The ROM is used to store constant parameters of the neural models. These parameters may vary among neurons or synapses but remain fixed over time. An example of a constant parameter is the membrane potential threshold (v_{th}) in most neuron models. On the other hand, the RAM is a type of memory that can be both read and written using an address. It is used to store variable parameters, which describe the current state of a neural model. These variable parameters are updated at each time step based on their previous values and the inputs to the neural model. An example of a variable parameter is the membrane potential (v) of a neuron.

The number of ROM components generated corresponds to the number of constant parameters in the neural models, while the number of RAM components generated corresponds to the number of variables. The length of each ROM and RAM component is determined by the number of neurons and synapses in the neural network, respectively.

Example 5.2.4 – Let's consider an example of a network consisting of 5 neurons using the Izhikevich (IZH) model. The IZH model has 4 constant parameters (a, b, c and d) and 2 state variables (v and u). Each neuron in the network can have different values for these parameters and variables :

(1) In this case, a total of 4 ROMs and 2 RAMs will be generated. Each ROM stores the values of a specific parameter (a, b, c, or d) for all the neurons. Similarly, each RAM stores the values of a specific variable (v or u) for all the neurons.

(2) Since we have 5 neurons in the network, each of the generated ROMs and RAMs from (1) will contain 5 values. To access the parameters or variables of a specific neuron, the same address is used across all the ROMs and RAMs.

The storage strategy is to store all the parameters and variables of a neuron or synapse at the same address in the memory components. This means that only one address is used to retrieve all the parameters and variables associated with a specific neuron or synapse.

Additionally, the data representation format $Q_{x.y}$ impacts the overall memory usage. All data in the memory have the same size, which is preconfigured. If $N_{\text{const\_param}}$, N_{var}, $S_{\text{const\_param}}$, and S_{var} represent the number of constant parameters for the neuron model, the number of variables for the neuron model, the number of constant parameters for the synapse model, and the number of variables for the synapse model, respectively, then the total storage requirement is

$$(N_{\text{const\_param}} + N_{\text{var}} + S_{\text{const\_param}} + S_{\text{var}}) \times Q_{x.y} \text{ bits}$$

Synapse module

The SYNAPSE module implements the set of equations of a synapse model, which is chosen by the user in the *gln* file. The module is one of the synapse models listed in table 4.1 of chapter 4. The synapse module receives input parameter values from the synapse ROMs and RAMs, as well as the output spike ('1' or '0') from the multiplexer. It calculates the updated values of the state variables and sends them back to the RAMs. Additionally, it sends the resulting synapse stimulation value W to the ACCUMULATOR. To ensure the calculations are performed at the correct time, the synapse module is controlled by a specific clock signal. This clock signal is provided by the CONTROLLER UNIT.

Accumulator

The ACCUMULATOR module receives the stimulation value W from the SYNAPSE module. It adds the input value to its internal register when it receives a clock signal. The internal register retains its value until it receives another clock signal and the *acc_rst* signal, at which point it is reset to 0. The module outputs the value of its internal register to the NEURON module. The clock signal for the accumulator module is also provided by the CONTROLLER UNIT.

Neuron module

The NEURON module implements the set of equations of a neuron model. It is selected from the list of *Light Esterel* neuron models in table 4.1 of chapter 4, and specified by the user in the *gln* file. Similar to the SYNAPSE module, the NEURON module takes as input the parameter values from the neuron ROMs and RAMs, as well as the stored stimuli from the accumulator. It performs its calculations and updates the values of the state variables in the RAMs. In case a spike is generated, we use this signal to enable the writing of the neuron ID to the BROADCASTING UNIT. To ensure that the spike signal is only taken into account once, a signal the spike output is set to '1' until the next controller clock signal input, indicating that a spike has been emitted by the neuron. To ensure that the spike is only emitted once, a signal *o_emit_spike_en* is used in conjunction by the CONTROLLER unit. As the SYNAPSE module, the NEURON module operates based on the clock signal provided by the CONTROLLER UNIT and performs its calculations at each clock cycle.

IDs selector

To ensure that synapses are updated before neurons, we have implemented a strategy where one neuron is updated at a time. This is achieved by processing all the input (or pre) synapses associated with that neuron and storing the resulting stimuli in the ACCUMULATOR. While this approach requires more control steps, it eliminates the need for an additional memory in the case of a strategy that updates all the synapses first then all the neurons. To ensure the chosen strategy, the IDs SELECTOR selects a local neuron ID and outputs its related local synapse IDs sequentially. These IDs are used as addresses to access the memories for the NEURON module and the SYNAPSE module, respectively. To optimize the selection of IDs, the IDs SELECTOR module uses a rearranged version of the matrix connectivity, as illustrated in figure 5.13. In the modified matrix version in this example, the local neuron ID is the index of the column, and the pre synapse IDs are the integers within the same column. This new version is synthesized as logic gates and not as a ROM component.

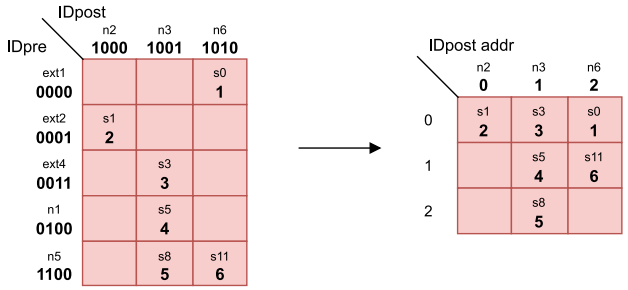


Figure 5.13: *IDs selector : an example of conversion of the connectivity matrix of the NPU 2 in example in figure 5.9. (On the left) The original connectivity matrix of the NPU. (On the right) The modified connectivity matrix, the integers have been moved up to the first lines.*

The IDs SELECTOR input signals are *i_start* to start the selection process, *i_end* to end the selection process and reinitialize the module internal variables, *i_nextIDneur* to ask for the next neuron ID, and *i_nextIDsyn* to ask for the next synapse ID. The output signals are *o_IDsyn* the synapse ID, *o_IDneurLocal* the local neuron ID to address the memories, *o_IDneurGlobal* the global neuron ID that will be sent to BROADCASTING UNIT if a spike is emitted, *o_endIDsynList* a end of list signal to notify that all the neurons and synapses have been processed, and *o_processIDneur* a signal to tell that the neuron module can be executed. More specific explanations are provided in annexe C on the reading process of the modified connectivity matrix.

5.2.4 CONTROLLER UNIT

The CONTROLLER UNIT is responsible for managing and controlling both the RECEPTION UNIT and the PROCESSING UNIT, to ensure that the overall system functions correctly and synchronously. It consists of two parallel FSMs, as depicted in figure 5.14, a RECEPTION CONTROLLER and a PROCESSING CONTROLLER. One controller manages the modules in the RECEPTION UNIT, while the other controls the modules in the PROCESSING UNIT. Both controllers are synchronized using the common signal *shiftRegs_updated* (shown in green in the figure), ensuring that the PROCESSING UNIT does not start until the shift registers in the RECEPTION UNIT have been updated.

The CONTROLLER UNIT has several inputs. But specifically, one input is the signal *i_NEW_dt* from the TIME STEP MANAGER UNIT, which indicates the beginning of a new time step. Another input is the enable signal *i_start_NPU*, which activates the NPUs. The CONTROLLER UNIT will only start its operation if both *i_NEW_dt* and *i_start_NPU* are '1'.

Reception unit controller

The reception FSM has been reduced to only 3 states. Its main roles are to send a shift enable signal to all the SHIFT REGISTERS at the beginning of a time step, then to reset the collected spikes in the NEURON IDS DECODER module, and finally to notify the PROCESSING CONTROLLER that it can start.

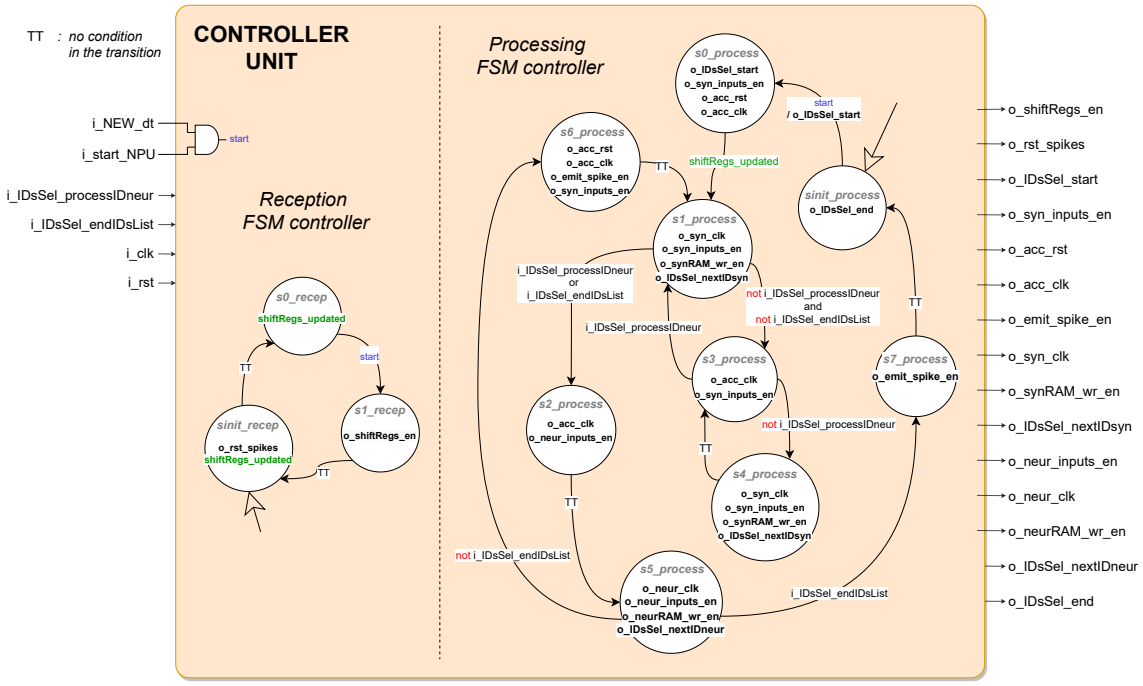


Figure 5.14: *Architecture of the CONTROLLER UNIT. Each state (circle) are individually named in grey. The inputs are identified with "i_" and part of the sensibility list. Local signals are in blue and green and are also part of the sensibility list. The outputs are identified with "o_" and are written in bold. "TT" means that there is no conditions in the transition. Initial state at the reset are the ones pointed by the empty arrows.*

Processing unit controller

The processing FSM orchestrates all the modules in the PROCESSING UNIT. Its main role is to ensure the proper sequence of processing steps for each neuron. This includes processing all the pre synapses, calculating the neuron's response based on the accumulated stimulation, and sending the eventual emitted spike to the BROADCASTING UNIT. The PROCESSING CONTROLLER consists of 9 states :

- **Initital state *sinit_process*** : it's the initial state. In this state, the PROCESSING CONTROLLER outputs the signal *o_IDsSel_end* to stop the operation of the IDs SELECTOR module. It waits for the arrival of the *start* signal, and once it arrives, it triggers the beginning of the IDs SELECTOR module. In the next clock cycle, the new state is *s0_process*.

- ***s0_process*** : In this state, the first IDs from IDs SELECTOR are sent to the SPIKES MULTIPLEXER and the SYNAPSE/NEURON MEMORIES. Before executing the SYNAPSE module, the value of the ACCUMULATOR is reset. When the signal *shiftRegs_updated* from the reception FSM is '1', in the next clock cycle, the new state is *s1_process*.

- ***s1_process*** : All the SYNAPSE module inputs are ready, a clock signal is sent to trigger its operation. A write enable signal is sent to all the synapse RAM memories to store the updated values. At the same time, the next synapse ID is requested, to be available at the next clock cycle. If the input signal *i_IDsSel_processIDneur* to process the neuron is '1' or

if all the synapses have been processed, at the next clock cycle, the new state is *s2_process*. Otherwise, it's *s3_process*.

- ***s2_process*** : The last pre-synapse of the current neuron has been processed, its stimulation output is added to the ACCUMULATOR by sending a clock signal to the module. All the inputs to the NEURON module are ready. Without any conditions, in the next clock cycle, the new state is *s6_process*.

- ***s3_process*** : The calculated stimulation of the pre-synapse is added to the ACCUMULATOR by sending a clock signal to it. At the same time, the new pre-synapse ID is available and the related inputs are present at the SYNAPSE module. If the signal to process the neuron is '1' then the next state will be *s1_process*. Otherwise, the next state is *s4_process*.

- ***s4_process*** : All the SYNAPSE module inputs are ready, its operation is triggered by sending a clock signal to it. The new calculated synapse values are stored in the SYNAPSE RAMs, and the next synapse ID is requested for the next cycle. In the next clock cycle, the new state is *s3_process*.

- ***s5_process*** : all the pre-synapses have been processed and their stimuli have been added to the ACCUMULATOR. The NEURON module is then executed by sending one clock signal to it, and the new calculated values are stored in the NEURON RAMs. At the same time, the next neuron ID is requested for the next clock cycle. If all the neurons have been processed, the next state will be *s7_process*. Else, it will be *s6_process*.

- ***s6_process*** : A new neuron ID has been selected, the ACCUMULATOR is reset to 0. The *o_emit_spike_en* is set to '1' during the one clock cycle to send the global neuron ID to the BROADCASTING UNIT in case a spike has been emitted in the previous clock cycle. The new inputs of the SYNAPSE module are ready. The next state will be *s1_process*.

- ***s7_process*** : This state sets the *o_emit_spike_en* signal to '1' during one clock cycle, to ensure that the *o_neur_spike* lasts only one clock cycle.

5.2.5 BROADCASTING UNIT

The BROADCASTING UNIT receives the neuron IDs of all the neurons being processed from all the NPUs, including the neuron IDs of the external neurons. It stores a neuron ID whenever it receives the spike signal, then it broadcasts them back to all the NPUs one by one. The BROADCASTING UNIT remains active during the entire time-step, allowing neuron IDs to arrive at any moment. This flexibility is important, especially when external inputs may not be bound to a specific time. Only at the beginning of a new time step, the broadcasting is paused to allow the NPUs to update their modules. Note that the BROADCASTING UNIT was added in the latest version of *SynchNN* to enable connections between neurons within the same NPU. The BROADCASTING UNIT is mainly composed of "FIFOs", a "FIFO MULTIPLEXER", a "FIFO READ SELECTOR", a "FIFO ADDRESS SELECTOR" and an "IDs BROADCASTING CONTROLLER".

Each NPU has its own FIFO to store the neuron IDs, and there is an additional FIFO dedicated to the external neurons'IDs. The size of each FIFO is determined by the number of neurons in the respective NPU or the number of external neurons it is associated with. This allows the BROADCASTING UNIT to receive and store the neuron IDs in parallel from all the NPUs.

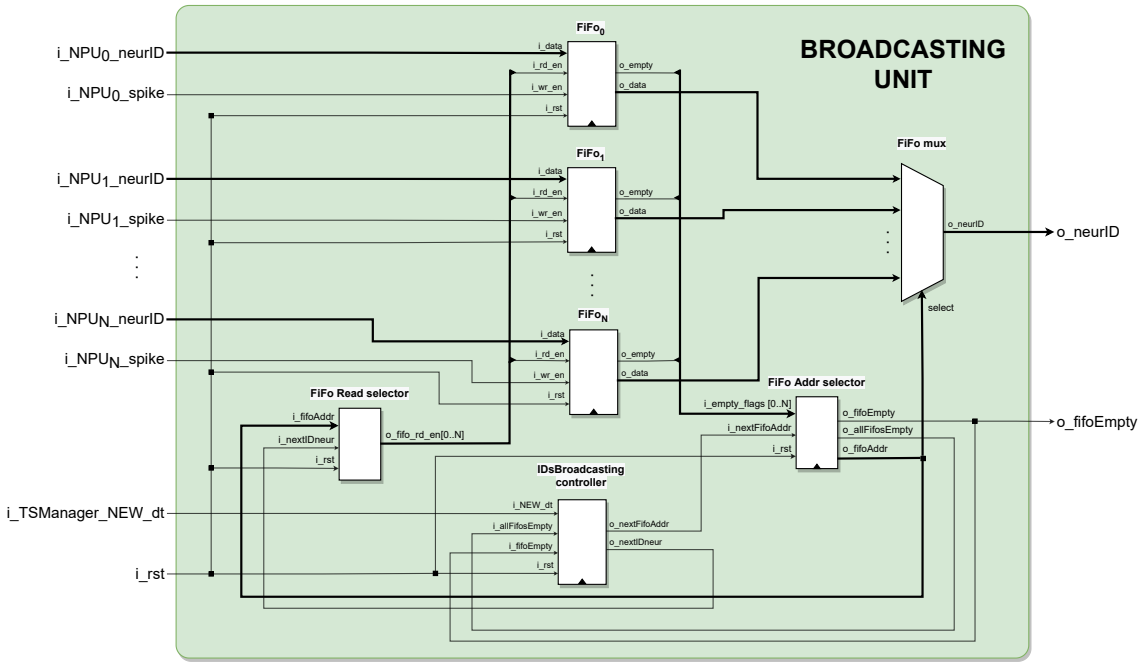


Figure 5.15: *BROADCASTING UNIT architecture.*

To broadcast the neuron IDs, the FIFO MULTIPLEXER selects one of the FIFOs'outputs to be broadcasted to all the NPUs. The selection of the FIFO is controlled by the FIFO ADDRESS SELECTOR, which outputs the addresses of the non-empty FIFOs. This allows for efficient selection and broadcasting of neuron IDs.

The FIFO READ SELECTOR is an asynchronous module which coordinates the read operations of the FIFOs. It sends a read signal to specific FIFO based on the address provide by the FIFO ADDRESS SELECTOR, when its input *i_nextIDneur* is '1'.

Finally, the IDs BROADCASTING CONTROLLER module makes the BROADCASTING UNIT independent. It controls the emptying process and the pause process of the BROADCASTING UNIT.

5.2.6 TIME STEP MANAGER UNIT

There are two main clock signals in *SynchNN*. The first one, *clk*, is the clock from the FPGA that is driving mostly all the unit modules. The second one, *clk_dt*, derived from the first one, is the clock defined by the application constraint time scale. This second clock imposes that the *SynchNN* calculations are bounded by 2 successive rising edges of *clk_dt*.

So the TIME STEP MANAGER UNIT is quite simple but still, it is an important unit. It is configured with the *dt* period constraint and it creates the *clk_dt* by dividing the FPGA *clk*. The unit outputs a signal *o_NEW_dt* during one *clk* cycle to all the other units every new time step.

5.3 Experiments and results

5.3.1 Coloring algorithms results

The 3 algorithms, FF, LDO, and WP, are applied to 3 different graphs. Each graph has the same number of nodes, which is 50, but the number of arcs varies. The number of arcs is determined by the density of connections within each graph. Specifically, we have chosen densities of 20%, 30%, and 50% for the three graphs, respectively. In this experiment, we are looking at the differences in the solutions provided by the 3 algorithms, and the results are shown in figure 5.16.

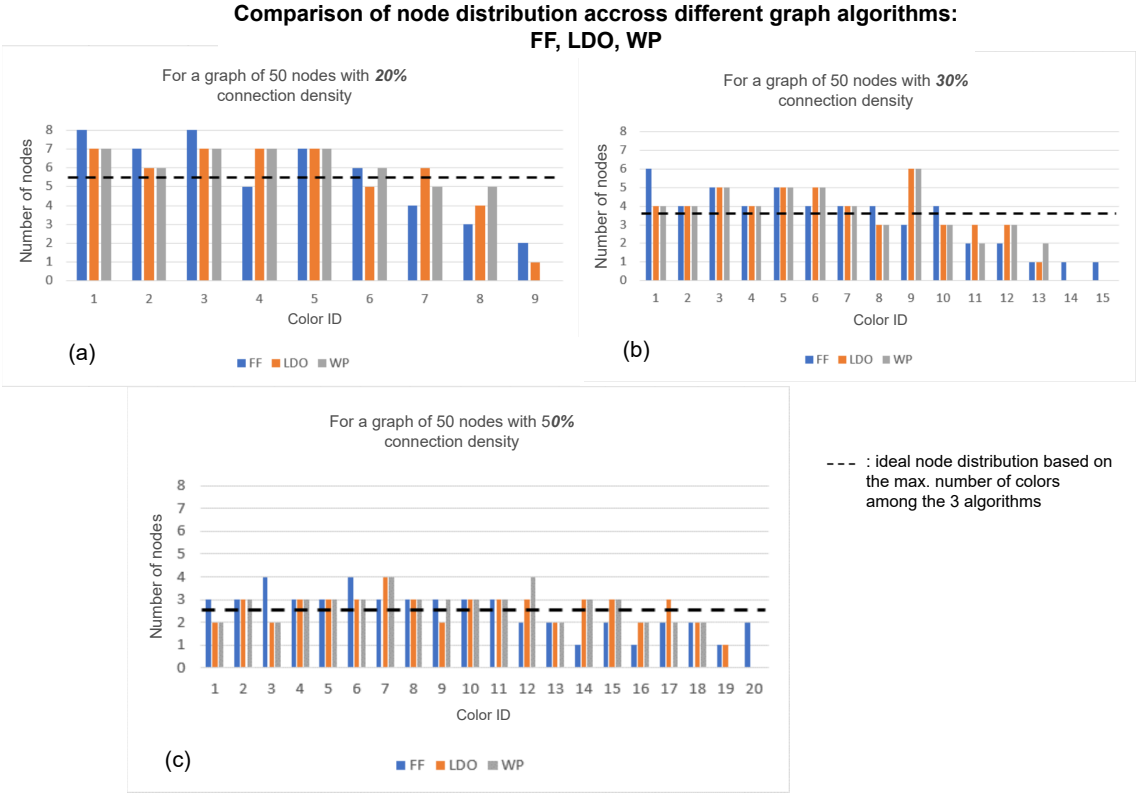


Figure 5.16: *Extracted and translated from (Chouchane, 2020), coloring results of the 3 algorithms FF, LDO and WP on a graph of 50 nodes with (a) 20% connection density, (b) 30% connection density and (c) 50% connection density.*

Each graphical plot represents the coloring results of the 3 coloring algorithms applied on the same graph. The x-axis represents the number of colors used in the coloring, while the y-axis represents the number of nodes assigned to each color. A black dash line represents the ideal distribution of nodes among the colors, which is obtained by dividing the total number of nodes (50) by the maximum number of colors among the algorithms. To decide the best coloring algorithm, it should (1) achieve the minimum number of colors and (2) provides a more even distribution of nodes among the colors to prevent over-utilization or under-utilization of NPUs.

Firstly, it is evident that as the density of connections increases, meaning the number of arcs increases, the number of colors required to properly color the graph also increases. This is intuitive since it becomes more challenging to group nodes that are not connected.

Secondly, the number of colors obtained by each algorithm may vary. In the presented results, the FF algorithm tends to yield more colors compared to the other algorithms. For example, in case (b), it results in 15 colors while the others require only 13 colors. On the other hand, both the WP and LDO algorithms produce fewer colors overall, but the WP algorithm consistently performs better in cases (a) and (c).

In situations where the algorithms yield the same number of colors, the choice of the best coloring depends on an additional criterion, namely the standard deviation. The standard deviation measures how far the actual number of nodes per color deviates from the ideal number of nodes per color represented by the dashed line in the plot. When comparing the LDO and WP algorithms, the WP algorithm minimizes the standard deviation, indicating that the number of nodes per color is closer to the ideal distribution represented by the dashed line.

So in conclusion, in these examples, the WP seems to be the best algorithm, however we keep all 3 algorithms as sometimes the best one is not always the same one. So, the three algorithms will be run on any given graph (or SNN specifications), the best coloring results chosen will be based on again (1) a minimum number of colors and (2) a fair distribution among the colors.

5.3.2 Functional validation

The functional validation consists in ensuring that the architecture developed corresponds to the expected behavior described in this chapter. It is done by running functional simulations on the architecture, and for that purpose we used the *Modelsim* hardware simulation tool. The functional validation is performed before proceeding to hardware synthesis and on-chip operation. As an important debugging step in hardware implementation, functional simulation is performed on the elementary modules that compose the units, then on each unit and finally on the whole architecture.

The simulations have demonstrated that each unit is operational, they are detailed in annexe D. For the hardware performance analysis in section 5.3.3, if N_s is the number of synpases in the NPU and N_n is the number of neurons, to process all the neurons and all the synapses in the PROCESSING UNIT, the number of clock cycles required is :

$$2 + 2 \times N_s + 2 \times N_n + 2 = 2 \times (N_s + N_n + 2) \quad \text{cycles.} \tag{5.1}$$

where : there are 2 clock cycles before the first synapse is processed, the update of one synapse needs 2 clock cycles, the update of one neuron needs 2 clock cycles, and there are 2 additional clock cycles for the FSMs to be all reset.

SynchNN validation

For the global *SynchNN* architecture validation, we compared two simulations of the same SNN : one simulation (software) with the BRIAN simulator and the other simulation (hardware) with *SynchNN* using Modelsim. The main objective is to ensure that the hardware simulation is close to the software simulation taken as reference, given the possible errors with the use of the fixed type (see section 4.1.1).

In the figures 5.17 and 5.18, we show the simulations obtained on the SNN configuration example in figure 5.8. For each simulation, we display the *raster plot* for the global SNN behavior, and we isolate the evolution of the membrane potentials of some neurons for more precise local behavior. The raster plot allows to visualize all the spikes emitted during the simulation by all

the neurons. In a raster plot, a dot represents a spike where its coordinates x and y refer to the time when the spike was emitted and the neuron (index) which has emitted the spike, respectively. As no specific application is demonstrated here, the parameters of the neurons and synapses have been chosen to facilitate the visual validation of the architecture : they are not biologically relevant but they allow to have particular patterns easily comparable. The neural models used and their parameters are given in the table 5.1.

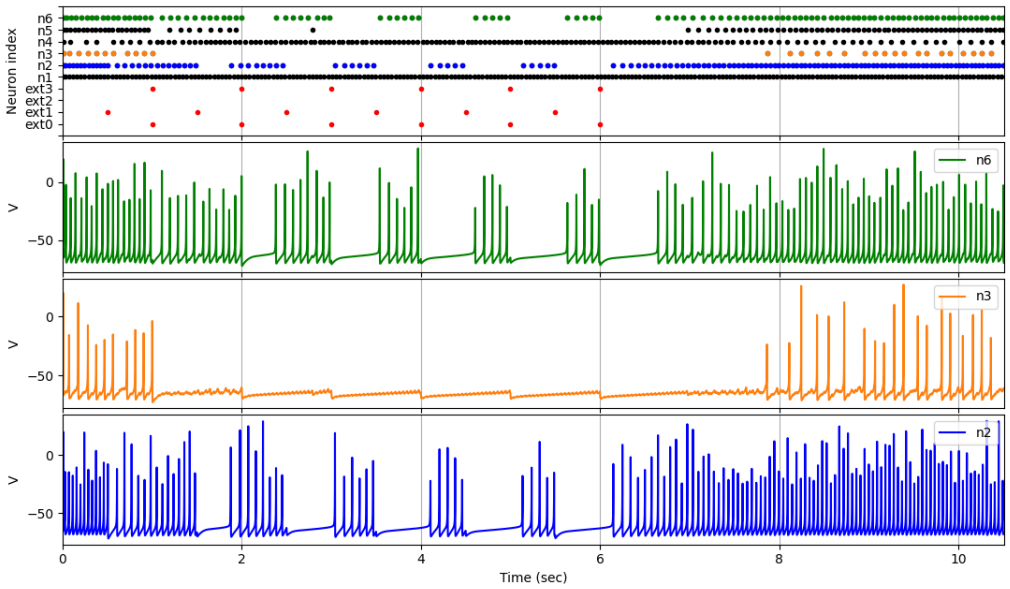


Figure 5.17: *BRIAN simulation of the SNN in example 5.8 with random parameters. (Top) The raster plot of the entire network. Y-axis from bottom to top : ext0, ext1, ..., ext3, n1, n2, ..., n6. The neurons n2, n3 and n6 are highlighted with different colors. (Bottoms) The evolution of the membrane potentials of the 3 neurons in NPU2.*

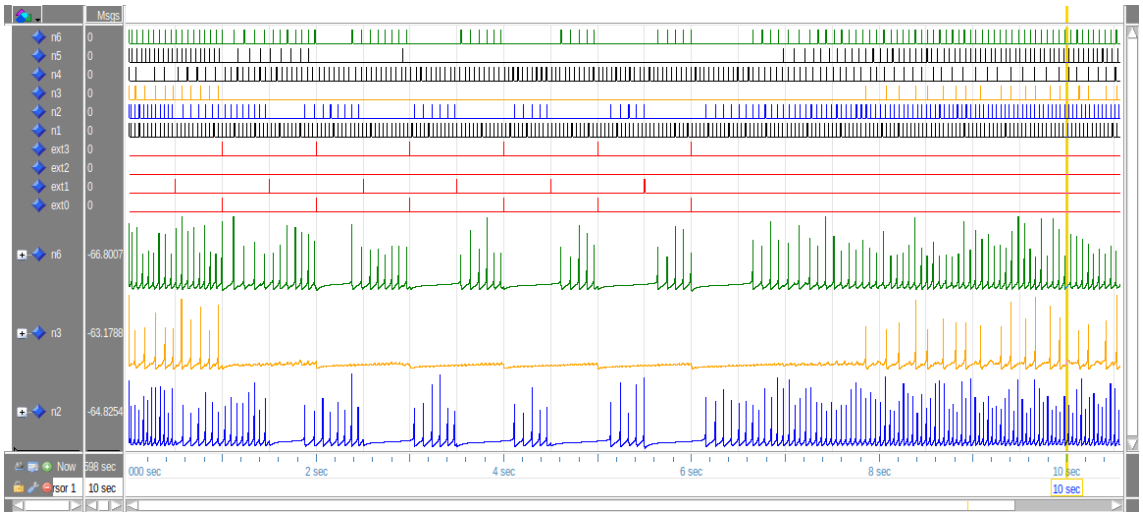


Figure 5.18: *SynchNN simulation in Modelsim of the SNN in example 5.8. Display setup and parameters are the same as in the BRIAN simulation above. The clock frequency is 50 Mhz and* $Q_{x.y} = 12 + 19$ *bits.*

Neuron model "IZH" (for all neurons)				
a	b	c	d	Iconst
0.02	0.2	-65.0	2.0	6.0

Synapse model "STP"			
P	$1/\tau_x$	$1/\tau_W$	W
(*external*) 1.0	0.9	0.0008	-2.0
(*internal*) 1.0	0.9	0.01	-0.5

Table 5.1: *Parameters of the neural models used in the software and hardware simulations for SynchNN validation. All axonal delays are set to 1 ms.*

As detailed in the table, the neuron model is the IZH model and all neurons are the same regular spiking (RS) type with the same parameters. *Iconst* allows the neurons to emit spontaneous spikes without external inputs. The synapse model is the short term plasticity model, and they all are inhibitory with the same axonal delays. Internal and external synapses have different parameters, the external synapses inhibit more the internal neurons on a longer period. On the simulations, we show the membrane potentials of the neurons in the NPU2. With the time step *dt* equals to 1 ms, we run the simulation during 10 seconds, and during the first 6 seconds, external spikes (red dots) inhibit the network. This external inhibition provokes the oscillatory patterns of some neurons (*n2* and *n6*), but also the complete inhibition of others (*n3* and *n5*). When no external stimulation is applied, all the neurons start to fire again with different frequencies as they are inhibiting each other.

At first sight, both simulations look similar: the patterns visually match as well as the number of spikes emitted within the oscillatory patterns. But when we watch closely, there are differences : the spikes are not emitted exactly at the same time, *e.g.* the neuron *n5* (see raster plots) emits one spike a little bit later in hardware around 3 seconds of simulation compared to the software. To be more precise, the percentage of similarity between both raster plots is actually 7.68%. This value represents the number of spikes that have been emitted exactly at the same times by the same neurons in both simulations. This huge difference is due to the fact that calculations in software use float type while in hardware it is fixed size type. Therefore, it leads to errors which explain why the shapes of the action potentials are different. Also error computations cause the spikes in hardware to be emitted with a delay, or even omitted due to the dynamics of the network : by analyzing the spike timings, the delay between hardware and software for the same spike can range from 0 to hundreds of milliseconds. Luckily, the resetting step of the neural models avoid the errors to explode. The errors can be decreased if the time step *dt* is lowered and the fractional representation size in $Q_{x.y}$ is increased. But, a small *dt* will limit the maximum size of the network to simulate (explained in the next sub-section) and a larger representation size will affect the resources utilization. Despite the errors, the spiking patterns are the same but with possible delays between the spikes. Also, in terms of the total number of spikes emitted during the whole simulation, the hardware and the software are similar up to 99.58% (see table 5.2).

TOTAL NUMBER OF SPIKES				
Neuron	Modelsim	Brian	Spikes number similarity	Spike timings similarity
n6	100	101	99.01 %	6.93 %
n5	78	80	97.50 %	3.75 %
n4	153	153	100.00 %	1.96 %
n3	26	24	91.67 %	4.17 %
n2	123	124	99.19 %	4.03 %
n1	228	229	99.56 %	8.3 %
ext3	6	6	100 %	100 %
ext2	0	0	-	-
ext1	6	6	100 %	100 %
ext0	6	6	100 %	100 %
TOTAL	726	729	99.25 %	7.68 %

Table 5.2: *Number of spikes emitted by each neuron in software and hardware during the 10 seconds simulation.*

Hardware simulations cannot totally match with software simulations. But in our case, the most important is that both simulations don't have big difference behavior, and for *SynchNN* we can validate the architecture as the global behavior match with the software behavior.

5.3.3 Performance : SNN maximal size

The total number of neurons N_n and synapses N_s that *SynchNN* supports depends on the available FPGA resources. The maximum size can be determined in 3 steps : (1) compute the maximum number of neurons and synapses respecting the total available memory; (2) compute the maximum number of neurons and synapses one NPU can process with respect to the time step constraint - here, we take into account the possibility of recurrent connections between neurons within the same NPU -; and (3) deduce the minimum number of NPUs needed to respect (1) and (2). The estimations depend on 4 major parameters : the constraint *dt* time step, the chosen working frequency of the FPGA, the available resources on the FPGA (on-chip memory and DSP blocks) and the neural models used. In the following, the equations are general, but the size estimation example is based on the worst case scenario : a totally fully-connected SNN with the same maximum axonal delay across all the synapses, but we consider only one connection from each external neuron.

For the step (1), as the total available memory on a FPGA is given by a total number of memory bits, the *SynchNN* memory usage should not exceed this limit. In *SynchNN*, the memory is used to store the neural parameters and variables (see RAMs/ROMs in the PROCESSING UNIT) which depends on the chosen neural models, to store the spikes in the IDDECODER ROM (see ROM in the RECEPTION UNIT), to model the axonal delays (see SHIFT REGISTERS in the RECEPTION UNIT), and to store the emitted ID of the spiking neurons (see FIFOs in the BROADCATING UNIT). So the following equation is the total memory usage considering one NPU :

$$
\begin{aligned}
& \left[\, Q_{x.y} \times \left(N_n \times N_{\text{nb\_param}} + (N_{s_{\text{int}}} + N_{s_{\text{ext}}}) \times S_{\text{nb\_param}}\right) \,\right]_{\text{RAMs/ROMs}} \\
+ \quad & \left[\, N_{n_{\text{pre}}} \times (N_{s_{\text{int}}} + N_{s_{\text{ext}}}) \,\right]_{\text{IDdecoder ROM}} \\
+ \quad & \left[\, (N_{s_{\text{int}}} + N_{s_{\text{ext}}}) \times \text{max\_delay} \,\right]_{\text{axonal delays}} \\
+ \quad & \left[\, (N_n + N_{n_{\text{ext}}}) \times \text{neuronID\_length} \,\right]_{\text{FIFOs}} \quad \leq \quad \text{max\_memory\_bits}_{FPGA}
\end{aligned}
\tag{5.2}
$$

where $Q_{x.y}$ is the data representation of all the neural values; $N_{\text{nb\_param}}$ and $S_{\text{nb\_param}}$ are respectively the total number of parameters and variables of the neuron model and the synapse model; "max_delay" is the maximum delay of all the synapses; "neuronID_length" is the length of the global neuron ID; $N_{n_{\text{ext}}}$ is the number of external neurons; $N_{s_{\text{int}}}$ and $N_{s_{\text{ext}}}$ are the number of internal and external synapses, respectively; and $N_{n_{\text{pre}}}$ is the total number of pre-synaptic neurons of the NPU.

Example 5.3.1 – We consider a fully-connected SNN case : the chosen data representation is $Q_{12.19}$; the neuron model is IZH ($N^{\text{IZH}}_{\text{nb\_param}} = 7$); the synapse model is STP ($S^{\text{STP}}_{\text{nb\_param}} = 6$); the maximum delay is max_delay = 50 ms; the number of external neuron is $N_{n_{\text{ext}}} = 100$; and each external neuron is only connected to one internal neuron. In this case, $N_{n_{\text{pre}}} = N_n + N_{n_{\text{ext}}}$; as $N_{n_{\text{ext}}} < N_n$ therefore the "neuronID_length" is equal to $\lceil \log_2 N_n \rceil + 1$, where $\lceil\ \rceil$ is the ceil function; and finally, $N_{s_{\text{ext}}} = N_{n_{\text{ext}}}$. On a FPGA Cylcone V family (5CGXFC9E7F35C8), the total memory bits available is 12 492 800 bits. So from the equations above, the total memory usage should be :

$$
\begin{aligned}
& 31 \times (7.N_n + (N_{s_{\text{int}}} + 100) \times 6) \\
+ \quad & (N_n + 100) \times (N_{s_{\text{int}}} + 100) \\
+ \quad & (N_{s_{\text{int}}} + 100) \times 50 \\
+ \quad & (N_n + 100) \times (\lceil \log_2 N_n \rceil + 1) \quad \leq \quad 12\,492\,800
\end{aligned}
$$

$$
318.N_n + 336.N_{s_{\text{int}}} + N_n.N_{s_{\text{int}}} + \lceil \log_2 N_n \rceil.(N_n + 100) + 33700 \quad \leq \quad 12\,492\,800
$$

As a fully-connected case, *i.e.* $N_{s_{\text{int}}} = N_n.N_n$, therefore the couple values $N_n = 158$ and $N_{s_{\text{int}}} = 24\,964$ is the maximum solution of the equation above. So, with the available memory on the FPGA, we estimate a SNN size with $N_n = 158$, $N_s = N_{s_{\text{int}}} + N_{s_{\text{ext}}} = 24\,964 + 100 = 25\,064$.

The step (2) is to determine the maximum number of neurons N_n^{NPU} and synapses N_s^{NPU} only one NPU can process. As a reminder, the total number of cycles needed to update the neurons and synapses is given by the expression in equation 5.1, and the general update must be done within one time step dt. The time step can be converted in a number of cycles dt_{cycles} depending on the chosen FPGA clock frequency. Therefore, the total number of cycles to update the SNN should not exceed the dt_{cycles}, and we obtain the equation 5.4 :

$$
2 \times (N_s^{\text{NPU}} + N_n^{\text{NPU}} + 2) \leq dt_{\text{cycles}} \tag{5.3}
$$

$$
N_s^{\text{NPU}} + N_n^{\text{NPU}} \leq \frac{dt_{\text{cycles}} - 2}{2} \tag{5.4}
$$

Example 5.3.2 – With a chosen working frequency of 50 Mhz and a dt of 1 ms, dt_{cycles} is therefore equal to 50 000 cycles, and we obtain :

$$N_s^{\text{NPU}} + N_n^{\text{NPU}} \leq 24\,998$$

With the fully-connected condition, therefore the couple $N_n^{\text{NPU}} = 157$ and $N_s^{\text{NPU}} = 24\,649$ satisfy the above equation, it represents the maximum capacity of one NPU.

Finally, the step (3) is to find out how many NPUs are required to simulate the SNN size estimation in step (1) with respect to the time step constraint in step (2). The number of NPUs should then fit in the FPGA. This number depends on the number of available dedicated digital signal processing (DSP) blocks. They accelerate typical signal processing tasks and include multipliers. DSP's functions can be implemented directly in logic (LUTs and flip-flops), however it would take significant resources. The DSP usage in *SynchNN* depends on the neural models used and the data representation $Q_{x.y}$.

Example 5.3.3 – On an FPGA of the Cylcone V family (5CGXFC9E7F35C8), the total number of available DSP blocks is 112. With the same example case taken since step (1), one NPU uses 23 DSP blocks. Thus, the maximum number of NPUs that can be generated on the board is 4.

As found in step (1), $N_n = 158$ and $N_s = 25\,064$. However, these values don't satisfy the step (2) constraint. Therefore, one NPU (NPU_1) will be generated and used at its maximum capacity *i.e.* $N_n^{\text{NPU}_1} = 157$ and $N_s^{\text{NPU}_1} = 24\,649$, and another NPU ($\text{NPU}_2$) will process the left over neurons and synapses *i.e.* $N_n^{\text{NPU}_2} = 1$ and $N_s^{\text{NPU}_2} = 415$.

Finally, we can confirm that the maximum SNN size of 158 neurons and 25 064 synapses can be simulated on the chosen FPGA, by generating 2 NPUs in order to respect the time step constraint.

In conclusion, the maximum number of neurons and synapses that *SynchNN* can simulate strongly depend on the application constraint and the FPGA resources constraint. The maximum SNN size changes depending on the SNN configurations and the neural models, and they can be calculated using the equations in step (1) but need to be confirmed with step(2) and step (3). The table 5.3 details the SNN maximum sizes that can be simulated in *SynchNN* depending on the neural models chosen. The neural models presented in this table are the hardware compatible models. This means that not all the implemented *Light Esterel* models can be simulated in hardware for now, *e.g.* the Hodgin-Huxley model due to the division operations that we didn't integrate in hardware yet. With the available resources of the Cyclone V family (5CGXFC9E7F35C8) FPGA, the estimation values are calculated considering a fully-connected network with 100 external neurons and all axonal delays equal to 50 ms.

Datasize	31 bits ($Q_{12.19}$)					
Neuron model choice						
IF	X					
LIF		X				
Adapt. LIF			X			
IZH				X	X	X
Synapse model choice						
Fixe	X	X	X	X		
STP					X	
STDP						X
Fully-connected SNN maximum size						
# neurons	167	167	167	167	158	150
# synapses	27 989	27 989	27 989	27 989	25 064	22 500
# NPU	2	2	2	2	2	1

Table 5.3: *SNN maximum size estimations that can be simulated on SynchNN, according to the neural models used. The estimations are based on the available resources on the Cyclone V (5CGXFC9E7F35C8) with : a fully-connected SNN with the same axonal delay for all the synapses; 100 external neurons; and one connection from each external neuron to internal neuron.*

5.4 Discussions

SynchNN has been developed during the last years of this thesis to address one of the main objectives, to simulate SNN in real-time for neurobiohybrid experiments. Several years are needed to develop an optimized architecture, that's why *SynchNN* still needs optimizations which will be discussed in this section.

As we have seen in the results, the coloring paradigm is not always adapted to reduce the use of FPGA resources. Indeed, when the density connections is high, the coloring tends to generate several NPUs that contains only few neurons. Moreover, the number of NPUs that can actually be generated strongly depends on the available resources on the FPGA target, and it can not be too big. In the last version of *SynchNN*, we integrated the possibility to connect neurons of the same NPU. Therefore, this feature makes the architecture not dependent to the coloring algorithms, so we can investigate on other separation algorithms, for example the clustering method. If properly chosen and set up, clustering seems to be more suitable for hardware resources utilization as it would generate fewer NPUs. Moreover, the clustering paradigm is actually closer to the simulation paradigm of many neuromorphic architectures based on the "*minicolumn*" building block (R. Wang et al., 2014; Merolla et al., 2014; Thakur et al., 2018). Biologically, a minicolumn refers to a vertical volume of cortex, containing a certain number of neurons and synapses (Buxhoeveden & Casanova, 2002). Neurons within a minicolumn can be connected together and are connected to neurons in other minicolumns in the cortex. Neuromorphic accelerators model therefore the minicolumn as a neural core, and several neural cores process portions of the SNN in parallel.

The broadcasting of the events (neuron IDs) to all NPUs was an easy implementation choice. However, first, it increases the number of communication bus lines, and second, an event does not necessarily need to be addressed to all NPUs. For these reasons, a routing method would be more

appropriate for an event to be sent only to specific NPUs while the others wait for events intended for them. This strategy is related to the use of the address event representation (AER) protocol.

Using one same hardware architecture to simulate different neural models in real-time is one contribution of *SynchNN*, while other neuromorphic accelerators restrict the number of their models and focusing on the possibility to simulate large scale network. At the expense of being able to simulate large neural network, *SynchNN* allows to simulate neural models that can be updated in one clock cycle. For models that require more clock cycles, significant modifications would be necessary as the CONTROLLER UNIT is designed to orchestrate only one cycle models.

SynchNN only uses the on-chip memories on the FPGA to store the neural models parameters and variables, and the network connectivity. As the available memories is quite limited, it affects the size of the SNN that can be simulated, especially when the models require multiple parameters to be saved. The use of external memories is a solution to overcome that limit by storing the neural parameters and variables outside the FPGA. However, using an off-chip memory needs flow control to access the data, which will make the architecture significantly more complex, especially if there are multiple off-chip memories or multiple NPUs that access the memories.

Once synthesized, the *SynchNN* architecture can not be reconfigured on-line *e.g.* changing the neural models parameters. To reconfigure the SNN, a new synthesis is required with the new parameters. Therefore it would be more interesting to change the parameters of the SNN on-line without resetting the overall simulation. This can be done by using more RAM memories which will contain the configurable parameters and which values can be changed and accessed by a communication protocol. For the axonal delays, it would require to change the shift register component.

We wanted the models to be configurable as much as possible *i.e.* many parameters are not constants. But to increase the size of the SNN, we can implement other models in which the number of variable parameters are restricted. For example, instead of having different spiking thresholds for all neurons, the use of only one constant threshold for all the neurons will save memory. Another solution to simulate larger SNN is to implement different SNNs on different FPGA boards, then to make them communicate with each other through a protocol that is to be defined. This way, each FPGA is actually processing a portion of a larger network.

5.5 Conclusion

The neuromorphic architecture *SynchNN* presented in this chapter was developed to overcome the limits of the *Light Esterel* native tools which are to be only able to generate and simulate limited sizes of SNN in hardware. This solution is related to our goals of communicating with real neurons in real-time, and understanding how the information is processed in the brain on a small scale using artificial models. The architecture is automatically generated and configured by the developed "Genlenet" software tool that takes as input the user specifications written in a "gln" file (cf. chapter 4).

Without development specifications, we have chosen to design the hardware respecting 2 constraints. The first constraint is related to the real-time artificial-biological communication which imposes the respect of a discrete biological time step processing. The second constraint which is the contribution of *SynchNN* is related to the possibility to simulate different neural models with the same hardware architecture. For the hardware implementation strategies we adopted a parallel architecture mixed with time-multiplexed architecture to improve the time processing and thus to

allow the simulation of larger SNN size. Even if it improves the number of neurons and synpases that can be simulated, the maximum size of the SNNs in *SynchNN* is still limited compared to existing neuromorphic architectures in the literature which aim to simulate large scale SNNs such as the size of the cortex. However the hardware-based neuromorphic accelerators are less flexible and are capable to simulate only few to only one neural models, where *SynchNN* allows to choose various models which were developed in *Light Esterel*. Other neuromorphic accelerators are also flexible but they use processor-based solutions. Developing our own architecture gives us a total control and thus makes it possible in the future to optimize and to improve the current SNN size limit, and especially to add new features and new models.

CHAPTER 6

Neurobiohybrid experiment and simulation

In this chapter, our goal is to connect the work we have developed throughout this thesis with the application field in which it is placed: Neurobiohybridization. Moreover, this chapter serves as a gateway to new perspectives for addressing this domain.

Firstly, we present the neurobiohybrid experiment we conducted in Japan using real biological neurons. We detail the experimental setup and the objectives we aspired to achieve. Furthermore, we discuss the results, the limitations and challenges we encountered during this experiment.

Subsequently, we introduce the development of a simulation framework. This framework is conceived as a solution to overcome the limitations experienced during our experiment, and as a platform for testing proof of concept before conducting real experiments. Moreover, this simulation framework is also intended to complement the SynchNN architecture that we presented in the previous chapter, offering practical insights into the applicability of our work, and extending its reach even into the domain of machine learning.

6.1 Central pattern generator (CPG) experiment

In this section, we present a experiment opportunity we conducted to communicate with real biological neurons using our synchronous models. Although we were unable to achieve the expected results, we explain the preliminary objectives, the methodology, the experimental setup and discuss the challenges encountered. The experiment was conducted in Japan, in collaboration with the Institute of Industrial Science (IIS) at the University of Tokyo, in november 2019.

For this first and only opportunity we had, we tried to reproduce the results obtained in (Ambroise et al., 2017) with our own synchronous implementations. They implemented a CPG with the IZH model and STP model (see chapter 2) for the neurons and synapses respectively (Ambroise, Levi, Joucla, Yvert, & Saïghi, 2013). In this experiment, they proved in an open and closed-loop biohybrid experiments that it was possible to regulate or synchronize the activity of a BNN and a SNN.

6.1.1 Definition

Central Pattern Generators (CPGs) are neural circuits that are characterized by their ability to produce rhythmic patterns of neural activity (Marder & Calabrese, 1996; Hill et al., 2001, 2002) without rhythmic sensory or input from higher-order neural structures. They are responsible for controlling essential functions such as locomotion and respiration, in many organisms. For example, CPGs control walking in salamanders (Ijspeert, Crespi, Ryczko, & Cabelguen, 2007), swimming in tadpoles (W.-C. Li, Merrison-Hort, Zhang, & Borisyuk, 2014), flying in locusts (Stevenson & Kutsch, 1987) and even heartbeats (Hill et al., 2001; Cymbalyuk, Gaudry, Masino, & Calabrese, 2002). The modeling of CPGs has been used in various applications such as robotics to control the locomotion of bioinspired robots (Ijspeert, 2008; C. Li, Lowe, & Ziemke, 2013; Barron-Zambrano & Torres-Huitzil, 2013) allowing for more natural, efficient and adaptive movements. In the field of neurobiohybridization, CPGs have been used in closed-loop experiments with biological neural network (BNN) such as in (Ambroise et al., 2017), where they have shown the possibility to change and control biological dynamics, as a first step toward the realization of innovative neuroprosthesis.

Elementary CPG

An elementary CPG is a minimal neural circuit capable of generating rhythmic oscillatory activity. The simplest model is composed of 2 neurons that reciprocally inhibit each other as illustrated in figure 6.1. In this configuration, when one neuron is active, it exhibits a bursting activity that inhibits the other. The burst refers to a successive spiking activity. However, when considering synaptic fatigue, the inhibition strength gradually decreases over time. This results in the activation of the other neuron, which, in turn, inhibits the first neuron. This mechanism repeats, leading to an alternating pattern of activity.

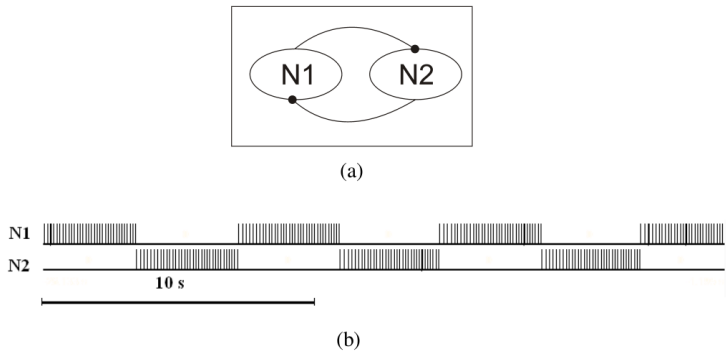


Figure 6.1: *Illustration of an elementary CPG. (a) It is composed of 2 neurons, N1 and N2, of type RS (see figure 2.7) connected with 2 inhibitory synapses. (b) Raster plot describing the behavior of the CPG, an oscillatory activity between the two neurons. In this configuration, a constant input current is applied to both neurons to make them fire. Figure extracted from (Ambroise, 2015).*

In this elementary CPG model, it is possible to change the frequency of the oscillatory activity by playing with the synapses parameters. However, to generate a wide range of frequencies, more complex CPG configurations containing more neurons are required. To create more complex CPGs, elementary CPGs are used as basic building blocks to achieve faster frequency or smaller period of oscillations (Marder & Calabrese, 1996) as illustrated in figure 6.2. This modular approach allows to model and replicate a wide range of biological rhythmic patterns observed in different organisms (Hill et al., 2001).

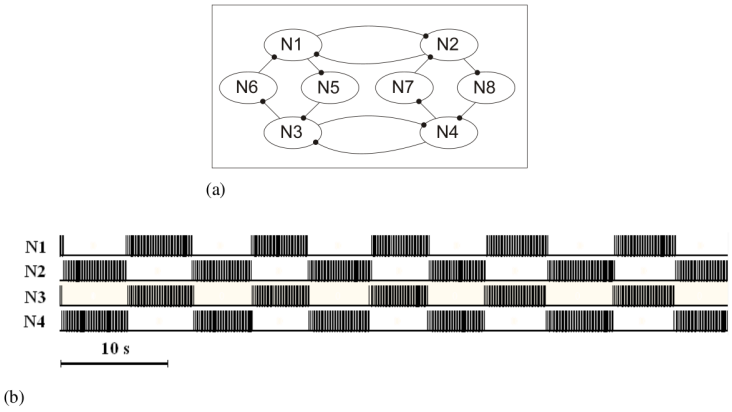


Figure 6.2: *Illustration of a segmental CPG. (a) It is composed of 8 neurons connected with inhibitory synapses. (b) The raster plot shows the behavior of N1, N2, N3 and N4 neurons. Figure extracted from (Ambroise, 2015).*

6.1.2 Synchronous CPG implementations

At the time of this experiment, the framework and hardware architecture presented in chapters 4 and 5 were not developed yet. Therefore, our methodology was to individually model, declare and connect manually the neurons and the synapses as illustrated on top of the figure 6.3. We will refer to this method as the "modular" method.

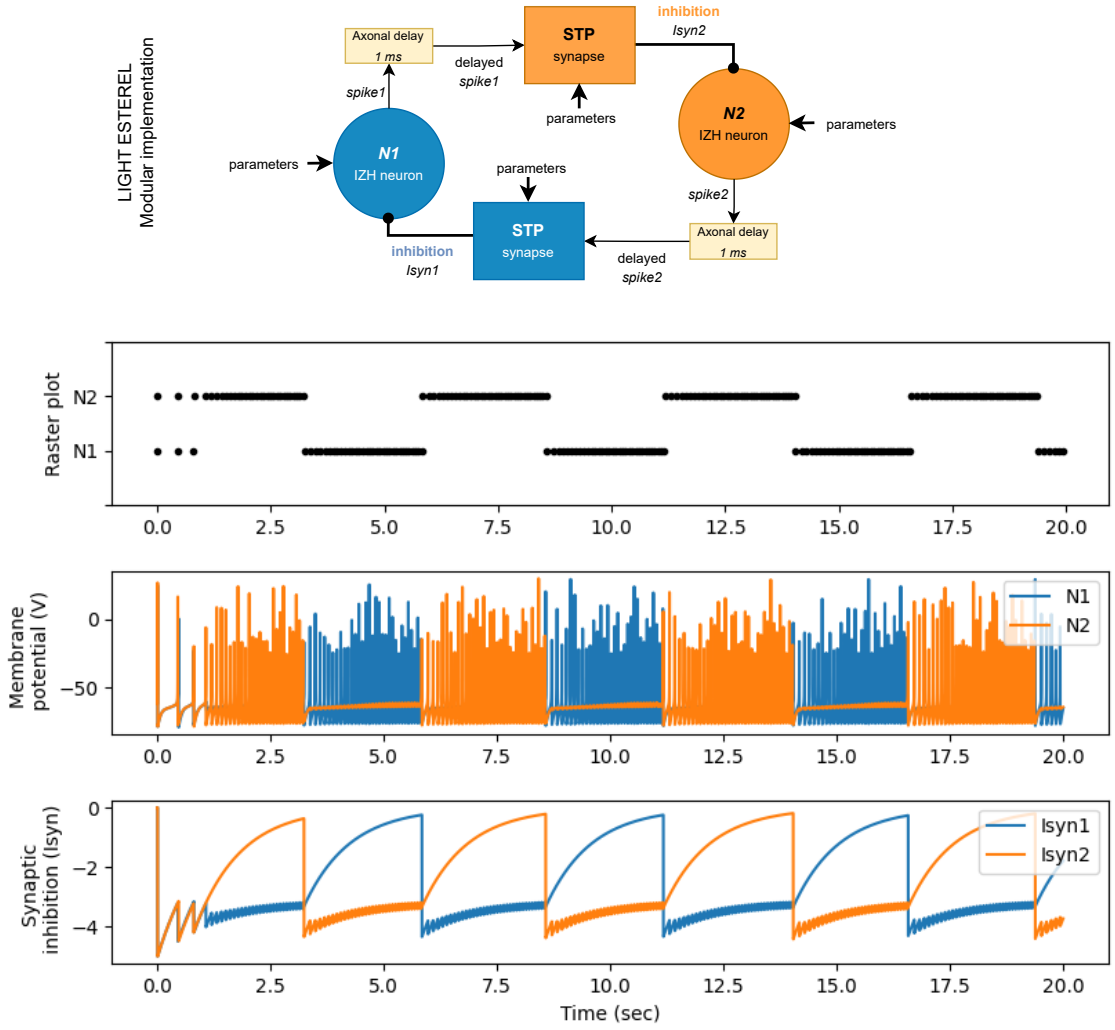


Figure 6.3: *Implementation structure and simulation of the Light Esterel elementary CPG. On top, the modular implementation of the CPG. The following graphs are the simulation of the generated C codes, showing the raster plot, the membrane potential of each neuron and the inhibition currents over time.*

This figure shows the global behavior of the CPG with the *Light Esterel* models. We can observe the inhibition conflict at the beginning of the simulation, then the alternating firing activity due to the inhibition patterns on one another. This simulation allows to validate the model and is obtained by using the parameters in table 6.1.

	Neuron model "IZH"				
	a	b	c	d	Iconst
N1	0.02	0.2	-65.0	8.0	7.0
N2	0.02	0.2	-65.0	8.0	7.025

	Synapse model "STP"			
	P	$1/\tau_x$	$1/\tau_W$	W
Syn1	0.01	0.00065	0.001	-5.0
Syn2	0.01	0.00065	0.001	-5.0

Table 6.1: *Parameters of Light Esterel elementary CPG, with dt=1 ms. Note that a little stimulation constant current is introduced to differentiate the 2 neurons. Iconst is the only parameter we can tune, while the others are inherent to the IZH model. Without this distinction, the two neurons would exhibit the exact same behavior, which would not result in the desired alternating patterns.*

In table 6.2, we compare the synthesis results of the "modular" implementation method used in this experiment with the architecture developed in chapter 5 for the same CPG configuration. The "modular" method consists in generating individually each module of the SNN configuration. For the elementary CPG, this means generating 2 neuron modules, 2 synapse modules and 2 axonal delay modules. With the second method involving the hardware architecture, we generate only 1 neuron and 1 synapse module per NPU (axonal delays are managed differently). We also highlight the impact of using the coloring algorithms. In the case of the elementary CPG, this results in the configuration of 2 NPUs, each managing 1 neuron and 1 synapse.

Resources occupation of elementary CPG				
		Hardware implementation methods		
Resources	**Total available**	**Modular**	**1 NPU**	**2 NPUs**
ALMs	113 560	**438**	503	978
DSP	112	30	**23**	46
Memory	12 492 800	0	0	0
Fmax (Mhz)		24	**138**	95

Table 6.2: *Resources utilization of different hardware implementation methods to simulate the Light Esterel CPG. The first one is the "modular" method generating each modules. The second method refers to the use of only 1 NPU : the 2 neurons and 2 synapses are managed by 1 NPU. The third method refers to the use of the coloring algorithm, and results in the generation of 2 NPUs : 1 neuron and 1 synapse per NPU.*

Even though we did use the modular method for this experiment, we want to highlight in the results reported in table 6.2 on the elementary CPG the need of a specific architecture to simulate and execute SNNs on FPGA, and also to carefully chose the method that uses the lowest resources and provides the best performance. The coloring algorithms are not always the best solution.

6.1.3 Neurobiohybrid experimental setup

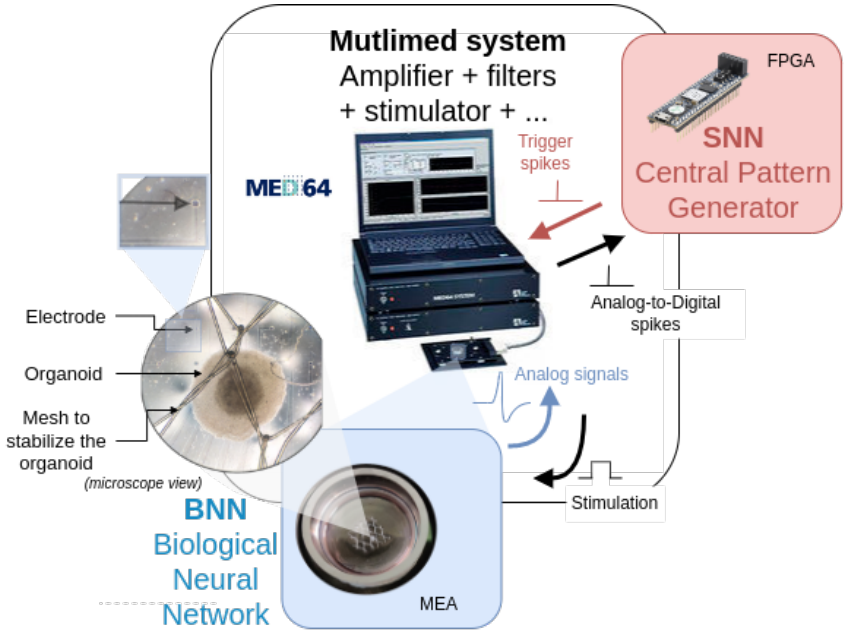


Figure 6.4: *Experimental setup for the neurobiohybrid experiment with CPG.*

Biological neural network

For the biological part in this experiment, we had at our disposal an organoid. The microscope view of the organoid is shown in figure 6.4. Organoids are self-organized, three-dimensional (sphere) tissue cultures derived from stem cells, and mimic the brain structure and its complexity at a small scale. Multiple dissociated neurons grow and begin to form connections with each other through functionality active synapses, establishing a random network and exhibiting spontaneous electrophysiological activity. Brain organoids offer a great potentials of creating neural circuit models that closely mimic brain structure and functionality, surpassing the capabilities of two-dimensional cell cultures.

In our case, we had dissociated cortical rat neurons. These neurons were maintained in culture and grow into an organoid over three months, through methods we do not detail here. Following the cultivation process, the organoid is relocated onto a multi-electrode array (MEA) Petri dish to record neuronal activities. The adhesion of the organoid to the MEA is facilitated by a mesh, among other things. In figure 6.4, we can observe a darker zone in the middle of the culture, it is where the concentration of cells is higher. The MEA used is a 64 electrodes MEA, but only few electrodes were used as the organoid size does not cover all of them.

Spiking neural network

For the artificial part, we implemented the elementary CPG described in the previous section on an FPGA board (*cmodA7*). The FPGA receives the digital spikes from the BNN. The spikes have IDs that allow to target specific neurons in the SNN. In return, the SNN outputs specific spikes that are used as triggers for the stimulation back to the BNN.

Multimed system

Multimed is an acquisition and signal processing board, specifically engineered for real-time analysis of multichannel biological data in open or closed-loop configurations. Its design features a 64 channel acquisition capability with a frequency of 10 kHz and 16-bit precision. The board is equipped with application-specific processing chains that offer sub-millisecond processing latency and decision-making, as well as live display, computer control interfaces, and recording capabilities (Ambroise et al., 2017). In figure 6.5, the processing chains typically include an Analog to Digital converter (ADC), linear band-pass filters, wavelet filters, alongside detection mechanisms for both action potentials and local field potentials. Moreover, the board supports single and network burst detection and the ability to flexibly route events for the creation of intricate feedback rules. To ensure lower processing latency, the described architecture is implemented on an FPGA, including the SD card recording, VGA display and serial control interfaces. All processing functions can be configured via a computer interface, and the data can be accessed and visualized through a commercial software.

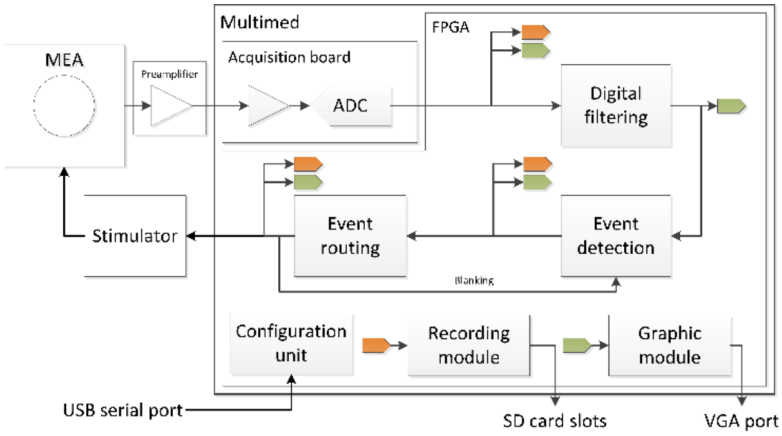


Figure 6.5: *Description of the multimed system. The orange and green ports are the points where the data can be accessed for recording or display purposes. Figure extracted from (Ambroise et al., 2017).*

The MULTIMED system is compatible with MEA, external preamplifier and a multifunctional stimulator. The stimulator can be calibrated to deliver specific current stimulation. It is designed with two digital input channels that are used to trigger the delivery of the electrical stimuli to the BNN. In this experiment, only one digital input channel is configured to receive one extracted action potential from the CPG as a digital trigger. Not every action potential from the CPG is used as a trigger, because rapid sequence of micro-stimulations could potentially cause premature damage to the BNN and the electrodes. Therefore, to avoid this, we only extract the first action potential from each burst of one neuron or specific neurons. In our case, we used as trigger just one neuron of the CPG.

6.1.4 Objectives

One objective was to validate our synchronous models in the communication with biological cells. A second objective, is to synchronize the BNN and SNN activities in a closed-loop experiment (see section 2.4.2). To achieve this, when the CPG begins a burst activity, the first action potential

of one specific neuron triggers an excitatory stimulus to the BNN, on one or multiple electrodes. Conversely, when a significant portion of neurons have spiked (a network burst) at the biological level, an inhibitory signal is sent back to the global CPG.

6.1.5 Expected results and challenges

While we could not complete the full experiment, the expected results based on (Ambroise et al., 2017) are illustrated in figure 6.6. In this figure, we can discern two scenarios. In the first, the BNN and the CPG are not interconnected via the stimulation/inhibition mechanisms, while in the second they are. The first scenario is intended to depict how each entity is supposed to act "naturally", with no interaction between them. The CPG displays an oscillatory pattern of activity, while the BNN exhibits spontaneous activity (random spikes). When the connection is established, the stimulation provided by the CPG, triggered by the first action potential of neuron N1, induces a network burst in the BNN, which in turn inhibits the CPG. This inhibition interrupts the oscillatory activity. The closed-loop experiment shows synchronization between the BNN and CPG, where global activity can be periodic. This is demonstrated in (Ambroise et al., 2017), where the SNN is capable of regulating the activity of the BNN.

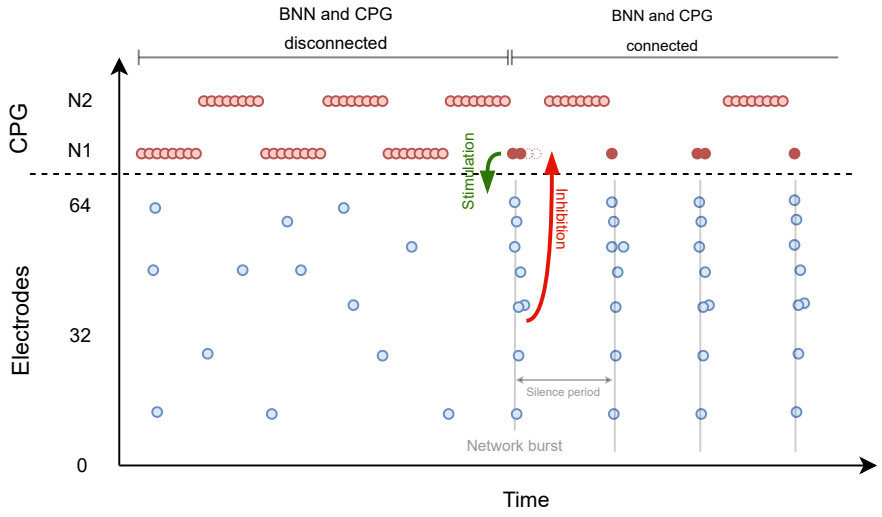


Figure 6.6: *CPG experiment expected results based on (Ambroise et al., 2017) results.*

Regrettably, the neural organoid we were working with was considered non-viable as we were unable to detect any activity on the MEA prior to the experiment. There are numerous potential causes for such failure in cell cultures. For instance, neural organoids lack a vascular system that is crucial for the delivery of oxygen and nutrients to the cells. Consequently, cells in the center of the organoid, which are deprived of these necessary resources, have a higher risk of dying. Additionally, the process of moving the organoid from its original environment to the MEA platform before the experiment requires delicate and sterile handling, which can be stressful for the cells and potentially leading to cell death. Furthermore, having access to new cultures is a long process, the limited time and the conditions that followed after november 2019 did not allow for a second attempt.

6.2 A simulation framework for neurobiohybrid or biohybrid experiments

6.2.1 Objectives

Considering the challenges associated with accessing a physical experimental setup, we decided to develop a software framework for simulating neurobiohybrid or biohybrid experiments. This choice provides various advantages :

- **Proof of concept** : a software framework allows us to conduct preliminary tests and validate application concepts or potential applications, before we move on to live experimentation.
- **Model testing and comparison** : the simulation environment provides a controlled setting where different SNN models can be tested and compared. This supports exploration of model behaviors, performance and characteristics.
- **Topology definition and hyperparameter tuning** : we can adjust and optimize the network's topology and models'hyperparameters. This is crucial for ensuring that the models are well-adapted to the task of interacting with biological neuronal activity.

The development of this simulation framework complements our *SynchNN* hardware architecture. The aim is to determine the global parameters from the software simulation and apply them to the hardware architecture configurations, enhancing the functionality and performance of the hardware network.

6.2.2 BRIAN-based simulator

In (Garg et al., 2021), the authors present their approach for hand gesture recognition using electromyography (EMG) data, *i.e.* electrical activity of muscles. They use the reservoir computing paradigm for their application. After defining the reservoir computing, we will describe their simulator architecture, and discuss about the modifications we incorporated to create our simulator.

Reservoir computing

Reservoir Computing (RC) is a computing paradigm that falls under the umbrella of Recurrent Neural Networks (RNNs). RNNs are characterized by loops in the network model, allowing information to persist or "flow" through the network over time. The loops are characterized by recurrent connections between neurons. This is a distinctive feature in contrast to traditional feed-forward neural networks, where information flows in one direction, from the input layer through hidden layers to the output layer. This structure allows RNNs, and by extension RC, to process sequential data and account for temporal dynamic behaviors, which are essential in various tasks such as natural language processing, speech recognition and time-series prediction (Elman, 1990; LeCun, Bengio, & Hinton, 2015; Graves, Mohamed, & Hinton, 2013).

RC consists of two primary components: a dynamic reservoir (also known as the hidden layer) and a readout layer. The reservoir is a large, randomly generated RNN of sparsely interconnected neurons, which serves as a temporal kernel or memory of past states/inputs. This reservoir transforms the input into a higher-dimensional space where linear separation is easier to achieve. The readout layer is a simple linear combination of the reservoir states. The key feature of RC is that

only the weights of the readout layer are trained, while the weights in the reservoir remain fixed (Jaeger & Haas, 2004; Maass, Natschläger, & Markram, 2002).

The SNN models we used and presented so far in this work can be classified also as RNNs. These SNN models have a randomly connected network architecture, in the case of large neuron numbers, and therefore can be seen as reservoirs in the RC paradigm. Moreover, in neurobiohybrid experiments, the input to the SNN comes in the form of temporal sequence of spikes, from biological neurons. This sequence depends on the current state of the biological network and the stimulations it has received. Therefore the connection between biological input and the SNN models demonstrates the potential of RC paradigm to process and interpret biological neural data.

The use of an open-source code : simulator architecture

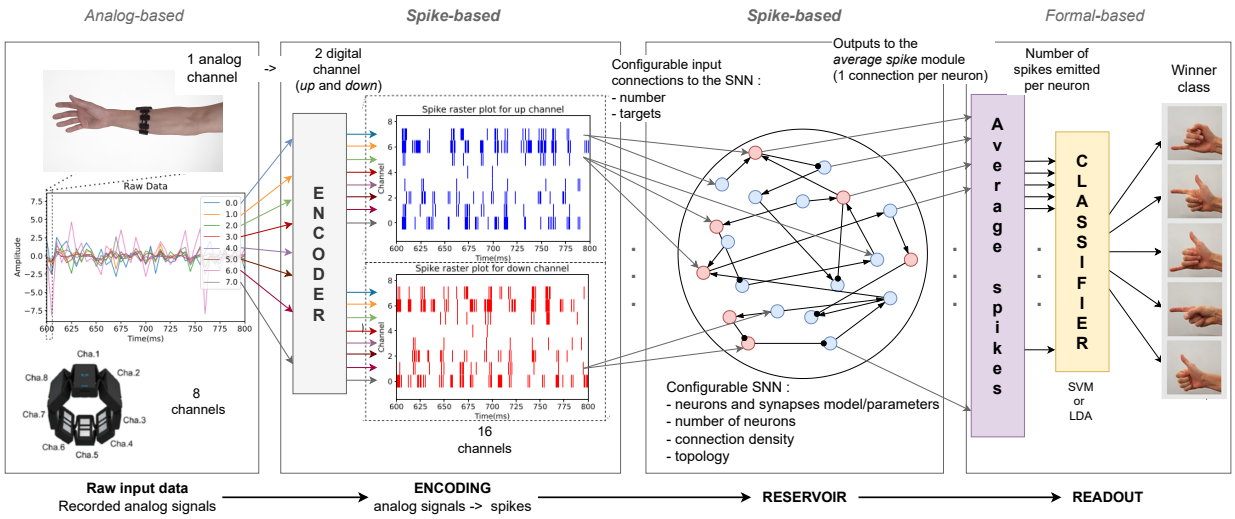


Figure 6.7: *Global architecture of the simulator for hand gestures recognition using Reservoir Computing paradigm. The inputs are hand gesture datasets collected from the Myo armband. The analog signals are converted into spikes (up and down) which are sent to the reservoir as input. In the reservoir, blue (red resp.) circles are excitatory (inhibitory resp.) neurons. Figures extracted and modified from (Garg et al., 2021).*

The architecture of the simulator developed in (Garg et al., 2021) is illustrated in figure 6.7, and the related open-source codes can be accessed through (Garg & Goupy, 2022). Without detailing the specifics of their architecture, we will provide a general overview of the key components.

The simulator was designed for hand gesture recognition, utilizing datasets collected through the *Myo armband*. This is a commercial EMG sensor developed by *Thalmic Labs* that is worn on the forearm and generates a digital EMG signal comprised of 8 channels, each operating at a frequency of 200 Hz.

Their architecture starts with their proposed novel method for converting 8 analog signals into spike sequences for input into a neuromorphic reservoir. Each analog input channel is converted into 2 spike sequence channels (referred to as 'up' and 'down', representing excitatory and inhibitory spikes, respectively). The resulting 16 channels are then used as input for the reservoir. The specific neurons to which each channel is connected can be configured, as well as the SNN that represents

the reservoir. This includes the neuron and synapse models to be used, their parameters, the density of connections, and the network topology. In table 6.3, we list the possible choices of models that can be simulated using their simulator. Although we won't detail the "critical" model, it is also one important approach in their work in this application they targeted. Briefly, either too much or too little activity is not effective in maintaining a working memory inside the reservoir, which is crucial for their implementation. The "critical" learning rule adjusts the weights in a way that guides the reservoir towards a more desirable state, specifically the "edge-of-chaos" state (Garg et al., 2021).

The readout part of the system includes an *average spikes* module, which calculates the number of spikes emitted by each neuron in the reservoir over a specified time period. The resulting vector values is then sent to a classifier, which determines the winning class for the sample time period.

Available models of	**(Garg et al., 2021)'s simulator**	**Our desired simulator**
Neuron	LIF ALIF	LIF ALIF IF IZH DSSN HH
Synapse	fixed STDP critical-STDP	fixed STDP STP
Neural properties	(*None*)	axonal delay exponential decays noise

Table 6.3: *Comparison of the available models within the simulator of (Garg et al., 2021) and the desired simulator we want to develop. Note that the number of models in our "desired" simulator may evolve in the future.*

Adaptations and modifications to create our own simulator

To meet our needs for a simulation environment for neurobiohybrid or biohybrid experiments, we are making some adaptations and modifications to the open-source code provided by (Garg et al., 2021). Our global architecture to work with *SynchNN* is shown in figure 6.8, and the modifications aim at having a more general and configurable simulator, and these include :

- **Adding our neural models** : we want to adapt the simulator to allow selection of neural models (neurons, synapses and properties) as listed in table 6.3 and which are described in chapter 2.
- **Input adaptation** : we want to generalize the input mechanisms to accommodate different types of inputs, not limiting to analog signals or to the number of channels. It will enable a wider range of input types. However, we will keep their approach for spike detection in analog signals.

- **Readout type** : in their architecture, the readout part is designed for formal-based classifiers (SVM or LDA), and we want our simulator to be fully spike-based until the classification.
- **Switching data types** : we have to implement in the simulator the possibility to use fixed-point operations as in hardware. Since floating-point and fixed-point simulations can diverge as developed in chapter 5, this will avoid wrong configurations of the hardware architecture.
- **Export parameters** : we want to add the module to directly export the configurations of the simulated SNN models, in order to configure the *SynchNN* architecture for hardware implementation. This means to generate the "gln" configuration files for the hardware.

Of course, our intention is not to compete with the work presented in (Garg et al., 2021). Instead, we aim to leverage and build upon their open-source contributions to develop a simulator that best fits our needs and objectives, particularly in the context of our SynchNN hardware architecture.

In our current version of the simulator, we have not yet implemented the SNN classifier, acknowledging that this will also require the implementation of a learning algorithm or method related to the classification, which we will not detail here. It's important to note that our simulator is still under development, the architecture presented in figure 6.8 represents the ideal structure we aim to achieve in order to simulate neurobiohybrid experiments.

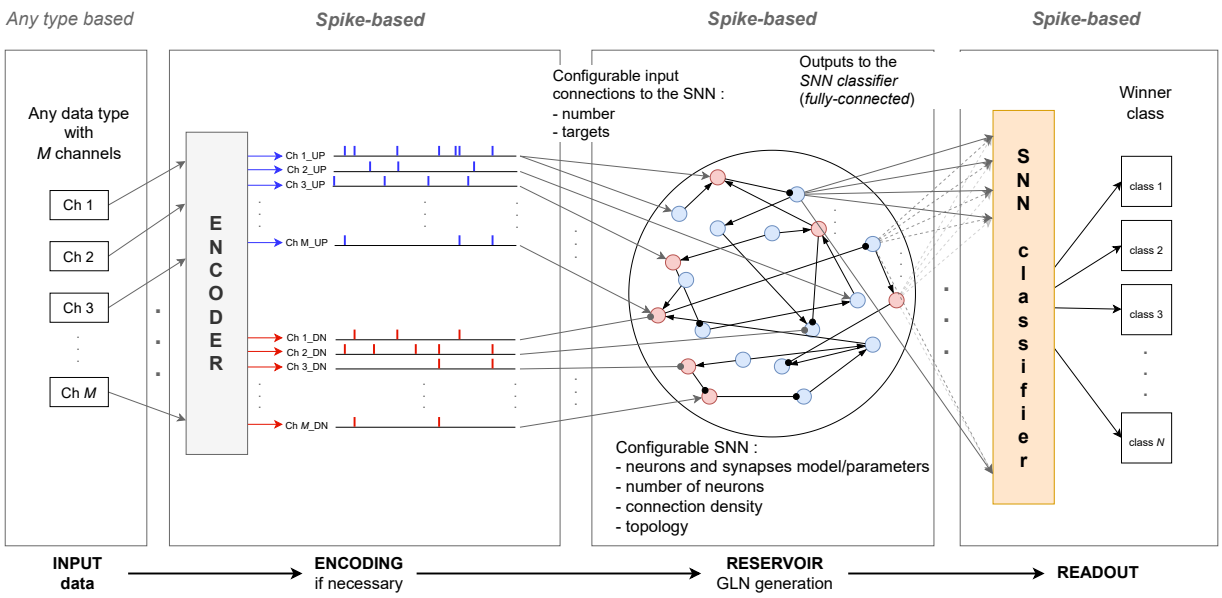


Figure 6.8: *Global architecture of the simulator to work with SynchNN, adapted from simulator in (Garg et al., 2021). It is designed for any applications using Reservoir Computing paradigm for now.*

6.2.3 Early-stage experimental results

Even though we haven't reached yet the stage of developing a complete spike based simulator with our models, we will present some preliminary experiments conducted during the development process of our desired simulator.

The first phase in our development process is to replace the SVM and LDA classifiers, illustrated in figure 6.7, with a Multi-Layer Perceptron (MLP) classifier. The MLP is a well-known type of

neural network model that is widely used for various classification tasks (Taud & Mas, 2018; Rumelhart, Hinton, & Williams, 1986). This initial step allows us to verify that by integrating the MLP classifier, the accuracy of the system, in the context of EMG data recognition, is not significantly compromised while still leveraging the available models in their reservoir. Our goal in the future is to transition from the MLP to a spike-based network by replacing the formal neurons in the MLP with spiking neuron models such as the IF model and adopting spike-based information encoding paradigms. Methods for this conversion have been explored within our laboratory in (Abderrahmane et al., 2020), as well as methods to directly train SNN with backpropagation (Lemaire et al., 2022).

Our initial experiment was conducted using the *EMG subset of the sensor fusion dataset* (Ceolini, Taverni, Payvand, & Donati, 2020), which includes 5 classes corresponding to 5 gestures as shown in figure 6.7. The encoded spikes from the "encoding" part in figure 6.7 were directly connected to the classifiers. We configured all the classifiers with their default parameters provided by the Python-based package *scikit-learn*. The results of this experiment serve as a baseline to compare and evaluate the accuracies provided by the classifiers when the reservoir is added. The results are shown in figure 6.9 comparing the accuracies of 4 different classifiers : LDA, SVM linear, SVM with radial basis function (RBF) and the MLP. It provides two major information. Firstly, among the 3 classifiers used in (Garg et al., 2021), the SVM_RBF is giving the highest accuracy. Secondly, the accuracies obtained with the added MLP are comparable to those achieved by the SVM_RBF classifier. This suggests that the integration of the MLP does not significantly alter the accuracy in the case of this configuration of "input_only". Note that the accuracies presented can be largely improved by optimizing the classifiers, but it is not the pupose of this work.

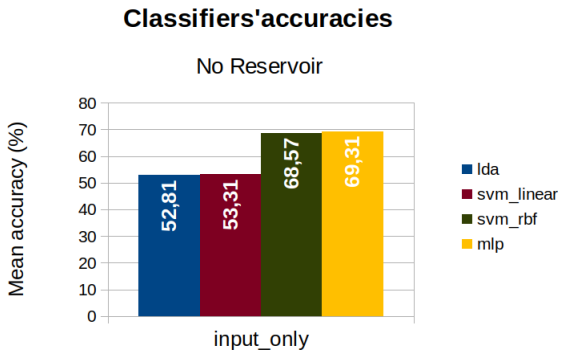


Figure 6.9: *Classifiers comparison when connected directly to the input. This experiment was applied to the EMG subset of sensor fusion dataset, which comprises 5 class dataset (Ceolini et al., 2020), consisting of 5 gestures. This serves as a baseline for evaluating the impact of adding the reservoir.*

In the second experiment, we introduced a reservoir with varying numbers of neurons which are randomly connected. The synaptic weights are uniformly distributed between 0 and 0.25 and remain fixed. This reservoir is placed between the input and the classifier as shown in figure 6.7. This experiment allows to evaluate how different reservoir configurations affect the RC performances. The results are presented in the graph in figure 6.10. As observed, the same classifiers as in figure 6.9 outperform the others in this configuration. However, the RC achieves lower accuracies, meaning that random connectivity may not provide an optimal configuration for this specific task.

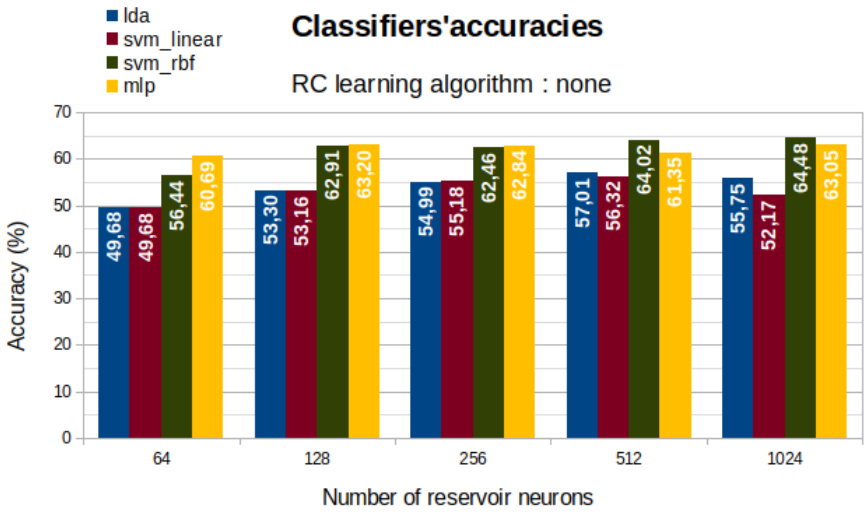


Figure 6.10: *Classifiers comparison depending on the number of neurons in the reservoir. The "none" learning algorithm means the synapses weights are fixed (uniform distribution between 0 and 0.25) as in (Garg et al., 2021)*

Then, we conducted a similar analysis by comparing the accuracies of the best-performing classifier, SVM_RBF, with those of the MLP. This comparison was done based on different configurations of the RC architecture using the available models in their simulator. Again, the results in figure 6.11 highlight the almost similarity in accuracies between the two classifiers, for various configurations of RC. Furthermore, by adding the "critical" learning algorithm, the RC is able to surpass the "input_only" configuration.

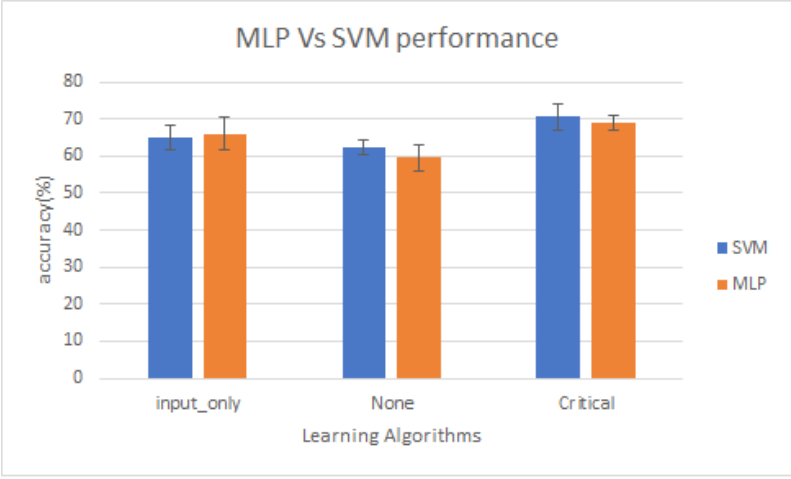


Figure 6.11: *Comparison of the SVM and MLP classifiers using different learning algorithms, considering various sizes of the RC architecture. The reported results are the mean accuracies obtained across the different RC sizes, along with the corresponding mean deviations for each classifier and learning algorithm.*

These findings confirm that the MLP can be effectively integrated with an RC architecture and demonstrate the potential for improving the system accuracy through the exploration and

tests of reservoir configurations. Theses results pave the way for future experiments focused on the conversion of the MLP into a spike-based neural network, aligning with our ultimate goal of developing spike-based simulation and execution capabilities.

The second phase in the development of the simulator involves incorporating different neuron models, synapse models and neural properties. This is done prior to the conversion of the MLP classifier into spike-based networks. The purpose of this phase is to explore the potential of different models in improving the performance of the system in this specific application as a case of study. However, at this stage, we are still in the progress of integrating these models, and we cannot provide conclusive results yet. Our initial experiments with the IZH model and the STDP combined with axonal delay have shown lower accuracies compared to the baseline reference. However, these results were obtained through random testing and require further investigation in terms of adjusting the model parameters, validating the correctness of the codes, optimizing the reservoir configurations, and conducting more systematic experiments. Only then, we will be able to present the results and fully understand their implications.

6.3 Conclusion

In this chapter, we presented our first and only neurobiohybrid experiment we conducted in Japan, with the collaboration of The University of Tokyo. We described the setup of the experiment, and then we discussed the challenges and limitations associated with the culture and experimentation of biological neurons, particularly in the case of organoids.

We also described how we are building a simulator, inspired by the work of (Garg et al., 2021). This simulator will help to test and configure our *SynchNN* hardware architecture for neurobiohybrid or biohybrid experiments. We discussed the need for such a framework, the modifications and adaptations we made to the original simulator.

In the development of the simulator, we started exploring the use of spike-based deep learning methods to classify patterns of activity in biological networks and/with reservoirs. Although we do not detail these deep learning approaches in this manuscript, such an approach is certainly a viable option in the context of neurobiohybrid experiments, particularly when it comes to classifying patterns of activity in biological networks or in the reservoir. We do not have yet concrete results to present, but we wanted to emphasize the need of such simulator for our hardware architecture, and by adding spike-based deep learning classifier, it allows us for future experiments to connect *SynchNN* with *SPLEAT*, a hardware architecture developed in our lab and designed to execute deep SNN models. This connection will allow us to transition fully from simulation to hardware implementation in the future.

CHAPTER 7

Conclusion and Perspectives

7.1 Conclusion

In this thesis, we focused on the exploration, modeling and simulation of bio-inspired and biomimetic neural networks using the synchronous approach. Our ambition was to understand how the biology works and replicate it with the goal of establishing communication with real biological neurons. For this purpose, we developed a framework throughout this thesis. An overview of this framework is provided in figure 7.1.

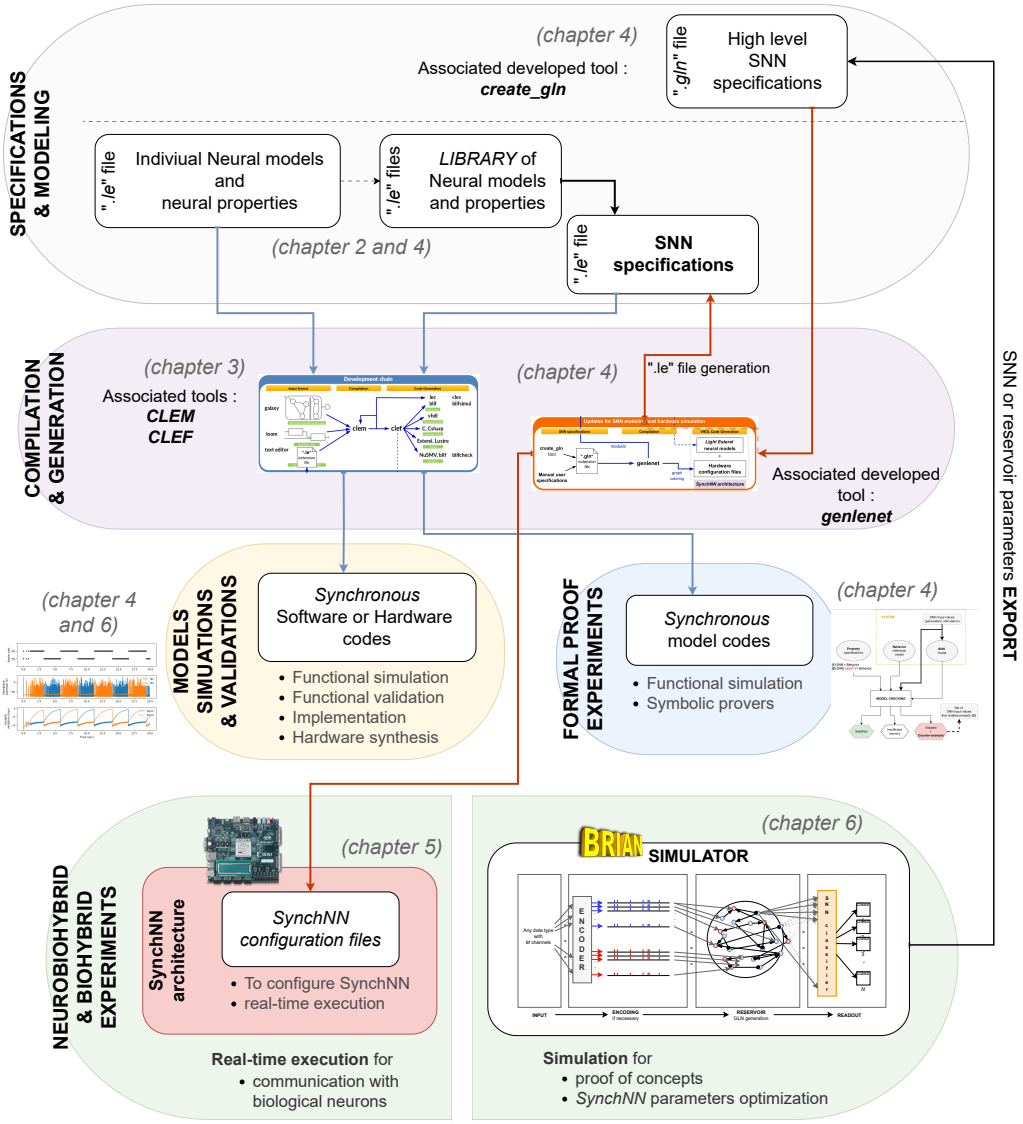


Figure 7.1: *Overview of the developed framework during this thesis. The framework aims to model, simulate and implement SNNs, perform symbolic proof experiments and target real and simulated neurobiohybrid or hybrid experiments. We refer each important point to the relevant chapter in this manuscript.*

We started by studying biological mechanisms and models to simulate and to implement neural networks that closely mirror biological behavior (Chapter 2). Our main tool was the high-level synchronous specification and modeling language, *Light Esterel* (Chapter 3), to achieve our objectives. This marked the first use of this tool in this field.

The first goal of our work, and our first major achievement, was the demonstration of the feasibility of modeling various neuronal models with different levels of abstraction relative to biology, using *Light Esterel* (Chapter 4). We validated our synchronous models by comparing them to reference simulations from the BRIAN simulator. We then created a library containing various models to facilitate simulations and implementations of a wide range of SNN models. Given the complexity and the necessity to specify a neural network configurations, we developed a higher-level specification format, ".gln". This format automatically generates ".le" specifications for large networks with the use of a developed compilation tool "genlenet", allowing potential users to create networks based on the available models without the need to master the *Light Esterel* language. To arrive at these results, it was necessary to make updates to the *Light Esterel* compilation chain (Chapter 4).

Our second goal in the use of the synchronous approach was to leverage automatic provers to study biological neural networks through their artificial counterparts. We thus tested our initial models under symbolic proof experiments such as proving the behavioral equivalence between a model and biology, or exploring parameters to guide a model towards a particular behavior. However, we encountered limitations with the tool we used, KIND2, due to the behavioral complexity of our models, defined by non-linear equations, or the temporal properties requiring too many time steps to be proven (Chapter 4). We remain convinced that more powerful tools exist, and that further exploration is still necessary.

As a third objective, in response to the limit of the network size that can be handled by the *Light Esterel*'s native compilation tools, as well as the efficiency of the generated hardware codes (Chapter 4), we decided to develop a specific hardware architecture, *SynchNN*, capable of simulating in real-time larger networks while remaining compatible with the already developed synchronous neural models (Chapter 5). This hardware architecture can be configured through the updated *Light Esterel* compilation chain with the developed specification file ".gln". This once again allows potential users to focus on the specification without worrying about low-level VHDL coding. The architecture was specifically designed to support various neural models and also potential models that we might add to the neural synchronous library in the future. This differs from certain architectures where the models are limited or even fixed.

Finally, a forth objective in response to the challenges and limitations we encountered in carrying out neurobiohybrid experiments, was to direct our efforts towards the development of a software simulator based on BRIAN (Chapter 6). This simulator has several objectives, including providing proof of concept for future neurobiohybrid or biohybrid experiments, but also to complement *SynchNN* to explore and optimize the parameters to be applied to *SynchNN* for the transition to real experiments. The emergence of the Reservoir Computing paradigm may seem disconnected from our bio-inspired and biomimetic approach, but we consider that the reservoir behaves like BNN. Consequently, we treat the reservoir as a black box, over which we have no control in terms of its internal connectivity and with which we only interact through its inputs and outputs. The interpretation of biological spikes is a complex task, and the encoding of information

in biology still remains an open question. Therefore, we believe that machine learning and deep learning approaches could, in their own way, provide solutions in the context of the interpretation of biological activity, such as they could play a high-level interpreter of the existing correlations between spike sequences. The subject is of course very vast and we cannot make definitive statements, but a major research axis on deep bio-inspired classification methods exists within our team, which would allow us to merge works and investigate the application of RC to biological spikes. Moreover, this would allow neurobiohybrid experiments to go beyond only considering bursts in biological recordings, as we explain with the CPG neurobiohybrid experiment (Chapter 6).

Given the challenges and limitations we encountered throughout this thesis, which were further intensified by the Covid health crisis, we acknowledge that our work was significantly impacted and slowed down. Consequently, the results presented in this manuscript do not completely resolve all these objectives. However, we firmly believe they provide a solid foundation for future research. Our primary contribution to the field of neuronal hybridization is a comprehensive framework that integrates modeling, simulation, implementation, and potentially formal proofs, all based on the *Light Esterel* language.

7.2 Perspectives

Several research directions can be considered for the future to continue to address the different objectives previously mentioned. It would be beneficial to explore more symbolic tools for parameterizing neural models. The field of the synchronous approach is vast, applying to different fields of applications and complexity, and we are convinced that there is still much to test.

In the context of the architecture we have developed, we can consider optimizing *SynchNN* to be able to simulate larger networks and more various neural models. Indeed, it would be interesting to continue developing neural models to supplement our current library. This would offer the possibility to create and test different SNN models in real-time, where other architectures remain quite limited in model choices. To enrich our library, we should also implement a method for the division operation, this would offer us the possibility to run a wider range of models in hardware. Also, in the automatic configuration of the architecture, where we used coloring graph algorithms to partitioning the neural networks into into sub-groups, it would be interesting to study other approaches such clustering algorithms for potential improvements. The broadcasting of events within the architecture should be improved by changing the communication protocol by adopting the address event representation (AER) protocol. This protocol enables specific addressing of neurons instead of the broadcasting strategy. Additionally, to overcome the limitation imposed by the on-chip memory and expand the network size, one solution would be the use of external memory. However, it would suggest making numerous changes in the architecture including the incorporation of a control flow mechanism to facilitate the access to the external memory. Moreover, our present architecture does not allow to configure the SNN parameters online. To enhance flexibility and eliminate the need for recompilation after eventual modifications, exploring strategies for enabling online configuration would be a great benefit for using *SynchNN*.

Finally, the continuous development of the simulator to explore more neurobiohybrid experiments is a promising path. Currently, the development of the SNN part and the integration of

the same models of neurons, synapses and neural properties from the synchronous library are underway. The use of machine learning and deep learning in the interpretation of reservoir activity (biological or artificial) seems to be a promising approach in this field. Additionally, creating a fully spike-based simulator would allow us to generate the hardware model to target 2 hardware architectures developed in our team : *SynchNN* and SPLEAT. SPLEAT is a hardware architecture designed to support the execution of deep spike-based neural networks. This way, we would also be moving closer to the neuroprosthetic solutions advocated in the field of neurobiohybridation.

Author's Publications

Related international conference papers

Rasamuel, M., Gaffé, D., Levi, T., & Miramond, B. (2019, Oct). Synchronous approach for modeling spiking neurons. In *2019 ieee biomedical circuits and systems conference (biocas)* (pp. 1–4). Nara, Japan. doi: 10.1109/BIOCAS.2019.8919084

Unrelated international conference papers

Rasamuel, M., Khacef, L., Rodriguez, L., & Miramond, B. (2019, March). Specialized visual sensor coupled to a dynamic neural field for embedded attentional process. In *2019 ieee sensors applications symposium (sas)* (pp. 1–6). Sophia-Antipolis, France.

Bibliography

Abdelmoula, M. (2014). *Génération automatique de jeux de tests avec analyse symbolique des données pour les systèmes embarqués* (Doctoral dissertation, Université Nice-Sophia Antipolis). Retrieved from http://www.theses.fr/2014NICE4149

Abderrahmane, N., Lemaire, E., & Miramond, B. (2020). Design space exploration of hardware spiking neurons for embedded artificial intelligence. *Neural Networks*, *121*, 366–386.

Al-Omari, H., & Sabri, K. E. (2006). New graph coloring algorithms. *American Journal of Mathematics and Statistics*, *2*(4), 739–741.

Ambroise, M. (2015). *Hybridation des réseaux de neurones: De la conception du réseau à l'interopérabilité des systèmes neuromorphiques* (Doctoral dissertation, Université de Bordeaux). Retrieved from https://hal.science/tel-02527419

Ambroise, M., Buccelli, S., Grassia, F., Pirog, A., Bornat, Y., Chiappalone, M., & Levi, T. (2017). Biomimetic neural network for modifying biological dynamics during hybrid experiments. *Artificial Life and Robotics*, *22*, 398–403.

Ambroise, M., Levi, T., Joucla, S., Yvert, B., & Saïghi, S. (2013). Real-time biomimetic central pattern generators in an fpga for hybrid experiments. *Frontiers in neuroscience*, *7*, 215.

André, C. (1996, July). Representation and analysis of reactive behaviors: A synchronous approach. In *Computational engineering in systems applications, cesa* (Vol. 96, pp. 19–29). Lille, France: IEEE-SMC.

André, C., Marmorat, J.-P., & Paris, J.-P. (1991, July). Execution machines for ESTEREL. In *Ecc'91* (Vol. 2, pp. 1672–1677). Grenoble, France: Hermès.

Aslan, M., & Baykan, N. A. (2016). A performance comparison of graph coloring algorithms. *International Journal of Intelligent Systems and Applications in Engineering*, *4*(Special Issue-1), 1–7.

Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., ... Herculano-Houzel, S. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, *513*(5), 532–541.

Azouz, R., & Gray, C. M. (1999). Cellular mechanisms contributing to response variability of cortical neurons in vivo. *Journal of Neuroscience*, *19*(6), 2209–2223.

Bader, D. A., Meyerhenke, H., Sanders, P., & Wagner, D. (2013). *Graph partitioning and graph clustering* (Vol. 588). American Mathematical Society Providence, RI.

Baier, C., & Katoen, J.-P. (2008). *Principles of model checking*. MIT press.

Barnes, C. (2017). *Verification and validation of wireless sensor network protocol properties through the system's emulation* (Doctoral dissertation, Université Côte d'Azur). Retrieved from https://theses.hal.science/tel-01618142

Barron-Zambrano, J. H., & Torres-Huitzil, C. (2013). Fpga implementation of a configurable neuromorphic cpg-based locomotion controller. *Neural Networks*, *45*, 50–61.

Baumann, N., & Pham-Dinh, D. (2001). Biology of oligodendrocyte and myelin in the mammalian central nervous system. *Physiological reviews*, *81*(2), 871–927.

Beierlein, M., Gibson, J. R., & Connors, B. W. (2003). Two dynamically distinct inhibitory networks in layer 4 of the neocortex. *Journal of neurophysiology*, *90*(5), 2987–3000.

Benabid, A. L., Chabardes, S., Mitrofanis, J., & Pollak, P. (2009). Deep brain stimulation of the subthalamic nucleus for the treatment of parkinson's disease. *The Lancet Neurology*, *8*(1), 67–81.

Ben-Ari, Y., Khazipov, R., Leinekugel, X., Caillard, O., & Gaiarsa, J.-L. (1997). Gabaa, nmda and ampa receptors: a developmentally regulatedmenage a trois'. *Trends in neurosciences*, *20*(11), 523–529.

Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M., ... Boahen, K. (2014). Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, *102*(5), 699–716.

Benveniste, A., Le Guernic, P., & Jacquemot, C. (1991). Synchronous programming with events and relations: the signal language and its semantics. *Science of computer programming*, *16*(2), 103–149.

Berry, G. (1989). *Real time programming: Special purpose or general purpose languages* (Research Report). Sophia Antipolis: INRIA.

Berry, G. (1999). The constructive semantics of pure esterel. *http://www.inria.fr/meije/esterel/esterel-eng.html*.

Berry, G., & Gonthier, G. (1992). The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, *19*(2), 87–152.

Berry, G., Moisan, S., & Rigault, J.-P. (1983). Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In *Proc. ieee real-time systems symposium* (pp. 30–40).

Bi, G.-q., & Poo, M.-m. (1998). Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of neuroscience*, *18*(24), 10464–10472.

Bishop, C. M., & Nasrabadi, N. M. (2006). *Pattern recognition and machine learning* (Vol. 4) (No. 4). Springer.

Bonifazi, P., Difato, F., Massobrio, P., Breschi, G. L., Pasquale, V., Levi, T., ... others (2013). In vitro large-scale experimental and theoretical studies for the realization of bi-directional brain-prostheses. *Frontiers in neural circuits*, *7*, 40.

Bouali, A. (1998). Xeve, an esterel verification environment. In A. Hu & M. Vardi (Eds.), *Computer aided verification: 10th international conference, cav'98 vancouver, bc, canada, june 28–july 2, 1998 proceedings 10* (Vol. 1427, pp. 500–504). Springer Berlin Heidelberg.

Boufaied, H. (1998). *Machines d'exécution pour langages synchrones* (Doctoral dissertation, Université de Nice Sophia-Antipolis). Retrieved from https://www.theses.fr/1998NICE5234

Bourke, T., & Pouzet, M. (2013, April). Zélus: A synchronous language with odes. In *Proceedings of the 16th international conference on hybrid systems: computation and control* (pp. 113–118). Philadelphia, Pennsylvania USA.

Bradley, A. R. (2012, June 17-20). Understanding ic3. In *Theory and applications of satisfiability testing–sat 2012: 15th international conference, proceedings 15* (pp. 1–14). Trento, Italy.

Brette, R., & Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of neurophysiology*, *94*(5), 3637–3642.

Broccard, F. D., Joshi, S., Wang, J., & Cauwenberghs, G. (2017). Neuromorphic neural interfaces: from neurophysiological inspiration to biohybrid coupling with nervous systems. *Journal of neural engineering*, *14*(4), 041002.

Buccelli, S., Bornat, Y., Colombi, I., Ambroise, M., Martines, L., Pasquale, V., . . . others (2019). A neuromorphic prosthesis to restore communication in neuronal networks. *IScience*, *19*, 402–414.

Buxhoeveden, D. P., & Casanova, M. F. (2002). The minicolumn hypothesis in neuroscience. *Brain*, *125*(5), 935–951.

Buzsáki, G. (2010). Neural syntax: cell assemblies, synapsembles, and readers. *Neuron*, *68*(3), 362–385.

Buzsaki, G., & Draguhn, A. (2004). Neuronal oscillations in cortical networks. *science*, *304*(5679), 1926–1929.

Carnevale, N. T., & Hines, M. L. (2006). *The neuron book*. Cambridge University Press.

Cassidy, A., Andreou, A. G., & Georgiou, J. (2011, Mar). Design of a one million neuron single fpga neuromorphic system for real-time multimodal scene analysis. In *2011 45th annual conference on information sciences and systems* (pp. 1–6). Baltimore, USA. doi: 10.1109/CISS.2011.5766099

Catterall, W. A., Raman, I. M., Robinson, H. P., Sejnowski, T. J., & Paulsen, O. (2012). The hodgkin-huxley heritage: from channels to circuits. *Journal of Neuroscience*, *32*(41), 14064–14073.

Ceolini, E., Taverni, G., Payvand, M., & Donati, E. (2020). *Emg and video dataset for sensor fusion based hand gestures recognition.* https://doi.org/10.5281/zenodo.3663616.

Chacron, M. J., Pakdaman, K., & Longtin, A. (2003). Interspike interval correlations, memory, adaptation, and refractoriness in a leaky integrate-and-fire model with threshold fatigue. *Neural computation*, *15*(2), 253–278.

Chiappalone, M., Cota, V. R., Carè, M., Di Florio, M., Beaubois, R., Buccelli, S., ... others (2022). Neuromorphic-based neuroprostheses for brain rewiring: State-of-the-art and perspectives in neuroengineering. *Brain Sciences*, *12*(11), 1578.

Chicca, E., Stefanini, F., Bartolozzi, C., & Indiveri, G. (2014). Neuromorphic electronic circuits for building autonomous cognitive systems. *Proceedings of the IEEE*, *102*(9), 1367–1388.

Chouchane, Y. (2020, September). *Synthèse VHDL optimisée pour des réseaux de neurones artificiels* (Other). Master 2 ESTEL Université Côte d'Azur. Retrieved from https://hal.archives-ouvertes.fr/hal-03780097 (Encadré par M.Daniel Gaffé, et M.Marino Rasamuel)

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., ... Tacchella, A. (2002, July 27–31). Nusmv 2: An opensource tool for symbolic model checking. In *Computer aided verification: 14th international conference, cav 2002* (Vol. 14, pp. 359–364). Copenhagen, Denmark.

Collinger, J. L., Wodlinger, B., Downey, J. E., Wang, W., Tyler-Kabara, E. C., Weber, D. J., ... Schwartz, A. B. (2013). High-performance neuroprosthetic control by an individual with tetraplegia. *The Lancet*, *381*(9866), 557–564.

Crone, N. E., Sinai, A., & Korzeniewska, A. (2006). High-frequency gamma oscillations and human brain mapping with electrocorticography. *Progress in brain research*, *159*, 275–295.

Cymbalyuk, G. S., Gaudry, Q., Masino, M. A., & Calabrese, R. L. (2002). Bursting in leech heart interneurons: cell-autonomous and network-based mechanisms. *Journal of Neuroscience*, *22*(24), 10580–10592.

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., ... others (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, *38*(1), 82–99.

Dean, P., Porrill, J., Ekerot, C.-F., & Jörntell, H. (2010). The cerebellar microcircuit as an adaptive filter: experimental and computational evidence. *Nature Reviews Neuroscience*, *11*(1), 30–43.

De Carlos, J. A., & Borrell, J. (2007). A historical reflection of the contributions of cajal and golgi to the foundations of neuroscience. *Brain research reviews*, *55*(1), 8–16.

Deisseroth, K. (2011). Optogenetics. *Nature methods*, *8*(1), 26–29.

De Maria, E., Bahrami, A., l'Yvonnet, T., Felty, A., Gaffé, D., Ressouche, A., & Grammont, F. (2022). On the use of formal methods to model and verify neuronal archetypes. *Frontiers of Computer Science*, *16*(3), 1–22.

De Maria, E., L'Yvonnet, T., Gaffé, D., Ressouche, A., & Grammont, F. (2017, Dec). Modelling and formal verification of neuronal archetypes coupling. In *Proceedings of the 8th international conference on computational systems-biology and bioinformatics* (Vol. 17, pp. 3–10). Nha Trang, Vietnam: ACM. doi: 10.1145/3156346.3156348

De Maria, E., Muzy, A., Gaffé, D., Ressouche, A., & Grammont, F. (2016, Oct). Verification of temporal properties of neuronal archetypes modeled as synchronous reactive systems. In *International workshop on hybrid systems biology (hsb 2016)* (Vol. LNBI 9957, pp. 97–112). Grenoble, France.

DeMarse, T. B., Wagenaar, D. A., Blau, A. W., & Potter, S. M. (2001). The neurally controlled animat: biological brains acting with simulated bodies. *Autonomous robots*, *11*, 305–310.

Destexhe, A., & Rudolph-Lilith, M. (2012). *Neuronal noise* (Vol. 8). Springer Science & Business Media.

Douglas, R. J., & Martin, K. A. (2004). Neuronal circuits of the neocortex. *Annu. Rev. Neurosci.*, *27*, 419–451.

dundeemedstudentnotes. (2012). *Action potentials.* Online. Retrieved from https://dundeemedstudentnotes.wordpress.com/2012/04/06/action-potentials/

D'amour, J. A., & Froemke, R. C. (2015). Inhibitory and excitatory spike-timing-dependent plasticity in the auditory cortex. *Neuron*, *86*(2), 514–528.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179–211.

Erkkinen, M. G., Kim, M.-O., & Geschwind, M. D. (2018). Clinical neurology and epidemiology of the major neurodegenerative diseases. *Cold Spring Harbor perspectives in biology*, *10*(4), a033118.

Faisal, A. A., Selen, L. P., & Wolpert, D. M. (2008). Noise in the nervous system. *Nature reviews neuroscience*, *9*(4), 292–303.

Feldman, D. E. (2012). The spike-timing dependence of plasticity. *Neuron*, *75*(4), 556–571.

Feng, S., Xu, Y., Yang, S., & Shi, L. (2020). *Spikingjelly: A pytorch-based deep learning framework for spiking neural networks.* Online. Retrieved from https://spikingjelly.readthedocs.io/zh_CN/latest/

Fortunato, S. (2010). Community detection in graphs. *Physics reports*, *486*(3-5), 75–174.

Franzén, A. (2006). Using satisfiability modulo theories for inductive verification of lustre programs. *Electronic Notes in Theoretical Computer Science*, *144*(1), 19–33.

Furber, S. B., Galluppi, F., Temple, S., & Plana, L. A. (2014). The spinnaker project. *Proceedings of the IEEE*, *102*(5), 652–665.

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., & Brown, A. D. (2012). Overview of the spinnaker system architecture. *IEEE transactions on computers*, *62*(12), 2454–2467.

Gaffé, D. (1996). *Le modèle de grafcet: réflexion et intégration dans une plate-forme multiformalisme synchrone* (Doctoral dissertation, Université de Nice Sophia-Antipolis (UNS)). Retrieved from https://hal.science/tel-02514876

Gaffé, D. (2022). *Galaxy: suite logicielle autour des machines d'états finis.* Retrieved from http://www.unice.fr/dgaffe/recherche/outils_automate.html

Gaffé, D. (2023a). *Clem homepage.* Online. Retrieved from http://sites.unice.fr/dgaffe/recherche/site_Clem/index.html (consulted September 7, 2023)

Gaffé, D. (2023b). *Research web site.* Online. Retrieved from http://sites.unice.fr/dgaffe/recherche/research.html (consulted September 7, 2023)

Gaffé, D., & Ressouche, A. (2008, September). The clem toolkit. In *23rd ieee/acm international conference on automated software engineering* (pp. 495–496). Aquila,Italy.

Gaffé, D., & Ressouche, A. (2013, July 1-3). Algebraic framework for synchronous language semantics. In L. Ferariu & A. Patelli (Eds.), *International symposium on theoretical aspects of software engineering (tase 2013)* (pp. 51–58). Birmingham, United Kingdom.

Garg, N., Balafrej, I., Beilliard, Y., Drouin, D., Alibart, F., & Rouat, J. (2021). Signals to spikes for neuromorphic regulated reservoir computing and emg hand gesture recognition. In *International conference on neuromorphic systems 2021* (pp. 1–8).

Garg, N., & Goupy, G. (2022). *Emg experiments.* https://github.com/nikhil-garg/EMG_exp. GitHub.

Gautier, T., Le Guernic, P., & Besnard, L. (1987). Signal: A declarative language for synchronous programming of real-time systems. In *Proc. of a conference on functional programming languages and computer architecture* (pp. 257–277). London, UK: Springer-Verlag.

George, R., Chiappalone, M., Giugliano, M., Levi, T., Vassanelli, S., Partzsch, J., & Mayr, C. (2020). Plasticity and adaptation in neuromorphic biohybrid systems. *Iscience*, *23*(10), 101589.

Gerstner, W., & Kistler, W. M. (2002). *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press.

Gewaltig, M.-O., & Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, *2*(4), 1430.

Ginsberg, M. (1988). Multivalued logics: A uniform approach to inference in artificial intelligence. *Computational Intelligence*, *4*, 265–316.

Goodman, D. F., & Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics*, 5.

Grassia, F., Kohno, T., & Levi, T. (2016). Digital hardware implementation of a stochastic two-dimensional neuron model. *Journal of Physiology-Paris*, *110*(4), 409–416.

Graves, A., Mohamed, A.-r., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 ieee international conference on acoustics, speech and signal processing* (pp. 6645–6649).

Guinaudeau, O. (2019). *Neurone abstrait: une formalisation de l'intégration dendritique et ses propriétés algébriques* (Doctoral dissertation, Université Côte d'Azur). Retrieved from http://www.theses.fr/2019AZUR4001

Gurfinkel, A., & Ivrii, A. (2017). K-induction without unrolling. In *2017 formal methods in computer aided design (fmcad)* (pp. 148–155). doi: 10.23919/FMCAD.2017.8102253

Gyárfás, A., & Lehel, J. (1988). On-line and first fit colorings of graphs. *Journal of Graph theory*, *12*(2), 217–227.

Hagen, G., & Tinelli, C. (2008, 17-20 November). Scaling up the formal verification of lustre programs with smt-based techniques. In *Formal methods in computer-aided design, FMCAD 2008* (pp. 1–9). Portland, Oregon, USA. doi: 10.1109/FMCAD.2008.ECP.19

Halbwachs, N. (2013). *Synchronous programming of reactive systems* (Vol. 215). Springer Science & Business Media.

Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991, September). The synchronous data flow programming language lustre. *Proceedings of the IEEE*, *79*(9), 1305–1320.

Halbwachs, N., Lagnier, F., & Raymond, P. (1993, June). Synchronous observers and the verification of reactive systems. In *Algebraic methodology and software technology (amast'93)* (pp. 83–96). Twente: Springer.

Halbwachs, N., & Raymond, P. (1999). Validation of synchronous reactive systems: from formal verification to automatic testing. In *Annual asian computing science conference* (pp. 1–12).

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming*, *8*(3), 231–274.

Harel, D., & Pnueli, A. (1985). On the development of reactive systems. In *Logics and models of concurrent systems* (Vol. 13, pp. 477–498). Springer.

Harris, K. D. (2005). Neural signatures of cell assembly organization. *Nature reviews neuroscience*, *6*(5), 399–407.

Hartline, D., & Colman, D. (2007). Rapid conduction and the evolution of giant axons and myelinated fibers. *Current Biology*, *17*(1), R29–R35.

Hebb, D. O. (1949a). *The organization of behavior: A neuropsychological approach*.

Hebb, D. O. (1949b). The organization of behavior. a neuropsychological theory. *John Wiley & Sons*.

Herculano-Houzel, S. (2009). The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in human neuroscience*, 31.

Hill, A. A., Lu, J., Masino, M., Olsen, O., & Calabrese, R. L. (2001). A model of a segmental oscillator in the leech heartbeat neuronal network. *Journal of computational neuroscience*, *10*, 281–302.

Hill, A. A., Masino, M. A., & Calabrese, R. L. (2002). Model of intersegmental coordination in the leech heartbeat neuronal network. *Journal of neurophysiology*, *87*(3), 1586–1602.

Hines, M. L., & Carnevale, N. T. (1997). The neuron simulation environment. *Neural computation*, *9*(6), 1179–1209.

Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, *117*(4), 500.

Huyck, C., & Passmore, P. (2013, 04). A review of cell assemblies. *Biological cybernetics*, *107*. doi: 10.1007/s00422-013-0555-5

Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: a review. *Neural networks*, *21*(4), 642–653.

Ijspeert, A. J., Crespi, A., Ryczko, D., & Cabelguen, J.-M. (2007). From swimming to walking with a salamander robot driven by a spinal cord model. *science*, *315*(5817), 1416–1420.

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks*, *14*(6), 1569–1572. doi: 10.1109/TNN.2003.820440

Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, *15*(5), 1063–1070. doi: 10.1109/TNN.2004.832719

Izhikevich, E. M. (2006). Polychronization: computation with spikes. *Neural computation*, *18*(2), 245–282.

Izhikevich, E. M. (2007). *Dynamical systems in neuroscience*. MIT press.

Izhikevich, E. M., & Edelman, G. M. (2008). Large-scale model of mammalian thalamocortical systems. *Proceedings of the national academy of sciences*, *105*(9), 3593–3598.

Jaeger, H., & Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *science*, *304*(5667), 78–80.

Jain, A. K., & Dubes, R. C. (1988). *Algorithms for clustering data*. Prentice-Hall, Inc.

Jain, A. K., Murty, M. N., & Flynn, P. J. (1999). Data clustering: a review. *ACM computing surveys (CSUR)*, *31*(3), 264–323.

Jeannet, B. (2003). Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, *23*(1), 5–37.

Jensen, T. R., & Toft, B. (2011). *Graph coloring problems*. John Wiley & Sons.

Joucla, S., Ambroise, M., Levi, T., Lafon, T., Chauvet, P., Saïghi, S., ... Yvert, B. (2016). Generation of locomotor-like activity in the isolated rat spinal cord using intraspinal electrical microstimulation driven by a digital neuromorphic cpg. *Frontiers in neuroscience*, *10*, 67.

Jung, R., Brauer, E. J., & Abbas, J. J. (2001). Real-time interaction between a neuromorphic electronic circuit and the spinal cord. *IEEE Transactions on neural systems and rehabilitation engineering*, *9*(3), 319–326.

Kahsai, T., & Tinelli, C. (2011). Pkind: A parallel k-induction based model checker. *arXiv preprint arXiv:1111.0372*.

Kasabov, N. K. (2014). Neucube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data. *Neural Networks*, *52*, 62–76.

Keren, H., Partzsch, J., Marom, S., & Mayr, C. G. (2019). A biohybrid setup for coupling biological and neuromorphic neural networks. *Frontiers in neuroscience*, *13*, 432.

Ketkar, N., Moolayil, J., Ketkar, N., & Moolayil, J. (2021). Introduction to pytorch. *Deep Learning with Python: Learn Best Practices of Deep Learning Models with PyTorch*, 27–91.

Khoyratee, F., Grassia, F., Saïghi, S., & Levi, T. (2019). Optimized real-time biomimetic neural network on fpga for bio-hybridization. *Frontiers in neuroscience*, *13*, 377.

Kirkpatrick, T., & Clark, N. R. (1966, March). Pert as an aid to logic design. *IBM journal of Research and Development*, *10*(2), 135–141.

Klotz, W. (2002). *Graph coloring algorithms*. Verlag nicht ermittelbar.

Kohno, T., & Aihara, K. (2007). Digital spiking silicon neuron: concept and behaviors in gj-coupled network. In *Proceedings of international symposium on artificial life and robotics* (Vol. 2007).

Kozai, T. D. Y., Langhals, N. B., Patel, P. R., Deng, X., Zhang, H., Smith, K. L., ... Kipke, D. R. (2012). Ultrasmall implantable composite microelectrodes with bioactive surfaces for chronic neural interfaces. *Nature materials*, *11*(12), 1065–1073.

Krichmar, J. L. (2018). Neurorobotics—a thriving community and a promising pathway toward intelligent cognitive robots. *Frontiers in neurorobotics*, *12*, 42.

Lansner, A. (2009). Associative memory models: from the cell-assembly theory to biophysically detailed cortex simulations. *Trends in neurosciences*, *32*(3), 178–186.

Lapicque, L. (1907). Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. *Journal de physiologie et de pathologie générale*, *9*, 620–635.

Lebedev, M. A., & Nicolelis, M. A. (2006). Brain–machine interfaces: past, present and future. *TRENDS in Neurosciences*, *29*(9), 536–546.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., & Jackel, L. (1989). Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, *2*.

Lemaire, E., Cordone, L., Castagnetti, A., Novac, P.-E., Courtois, J., & Miramond, B. (2022). An analytical estimation of spiking neural networks energy efficiency. In *Neural information processing* (pp. 574–587). Springer International Publishing. Retrieved from https://doi.org/10.1007%2F978-3-031-30105-6_48 doi: 10.1007/978-3-031-30105-6_48

Le Masson, G., Le Masson, S., & Moulins, M. (1995). From conductances to neural network properties: analysis of simple circuits using the hybrid network method. *Progress in biophysics and molecular biology*, *64*(2), 201–220.

Levitan, H., Segundo, J., Moore, G., & Perkel, D. (1968). Statistical analysis of membrane potential fluctuations: relation with presynaptic spike train. *Biophysical Journal*, *8*(11), 1256–1274.

Li, C., Lowe, R., & Ziemke, T. (2013). Humanoids learning to walk: a natural cpg-actor-critic architecture. *Frontiers in neurorobotics*, *7*, 5.

Li, W.-C., Merrison-Hort, R., Zhang, H.-Y., & Borisyuk, R. (2014). The generation of antiphase oscillations and synchrony by a rebound-based vertebrate central pattern generator. *Journal of Neuroscience*, *34*(17), 6065–6077.

Llinás, R. R. (2003). The contribution of santiago ramon y cajal to functional neuroscience. *Nature Reviews Neuroscience*, *4*(1), 77–80.

Lorach, H., Galvez, A., Spagnolo, V., Martel, F., Karakas, S., Intering, N., ... Courtine, G. (2023). Walking naturally after spinal cord injury using a brain-spine interface. *Nature*. Retrieved from https://doi.org/10.1038/s41586-023-06094-5

Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, *14*(11), 2531–2560.

Mahowald, M., & Douglas, R. (1991). A silicon neuron. *Nature*, *354*(6354), 515–518.

Maraninchi, F., & Rémond, Y. (2001). Argos: an automaton-based synchronous language. *Computer languages*, *27*(1-3), 61–92.

Maraninchi, F., & Rémond, Y. (2003). Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of computer programming*, *46*(3), 219–254.

Marchand, H., Bournai, P., Borgne, M. L., & Guernic, P. L. (2000). Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic Systems*, *10*, 325–346.

Marder, E., & Calabrese, R. L. (1996). Principles of rhythmic motor pattern generation. *Physiological reviews*, *76*(3), 687–717.

Markram, H., Lübke, J., Frotscher, M., & Sakmann, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, *275*(5297), 213–215.

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., ... others (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell*, *163*(2), 456–492.

Masland, R. H. (2004). Neuronal cell types. *Current Biology*, *14*(13), R497–R500.

McAlpine, D., Jiang, D., & Palmer, A. R. (2001). A neural code for low-frequency sound localization in mammals. *Nature neuroscience*, *4*(4), 396–401.

McDonnell, M. D., & Abbott, D. (2009). What is stochastic resonance? definitions, misconceptions, debates, and its relevance to biology. *PLoS computational biology*, *5*(5), e1000348.

McDonnell, M. D., & Ward, L. M. (2011). The benefits of noise in neural systems: bridging theory and experiment. *Nature Reviews Neuroscience*, *12*(7), 415–425.

McFarland, D. J., & Wolpaw, J. R. (2008). Brain-computer interface operation of robotic and prosthetic devices. *Computer*, *41*(10), 52–56.

McMillan, K. L., & McMillan, K. L. (1993). *Symbolic model checking*. Springer.

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., ... others (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668–673.

Mohajerani, M. H., McVea, D. A., Fingas, M., & Murphy, T. H. (2010). Mirrored bilateral slow-wave cortical activity within local circuits revealed by fast bihemispheric voltage-sensitive dye imaging in anesthetized and awake mice. *Journal of Neuroscience*, *30*(10), 3745–3751.

Mondal, B., & De, K. (2017). An overview applications of graph theory in real field. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, *2*(5), 751–759.

Mosbacher, Y., Khoyratee, F., Goldin, M., Kanner, S., Malakai, Y., Silva, M., ... others (2020). Toward neuroprosthetic real-time communication from in silico to biological neuronal network via patterned optogenetic stimulation. *Scientific reports*, *10*(1), 1–16.

Moxon, K. A., & Foffani, G. (2015). Brain-machine interfaces beyond neuroprosthetics. *Neuron*, *86*(1), 55–67.

Müller, J., Ballini, M., Livi, P., Chen, Y., Radivojevic, M., Shadmani, A., ... others (2015). High-resolution cmos mea platform to study neurons at subcellular, cellular, and network levels. *Lab on a Chip*, *15*(13), 2767–2780.

Musk, E., et al. (2019). An integrated brain-machine interface platform with thousands of channels. *Journal of medical Internet research*, *21*(10), e16194.

Nanami, T., & Kohno, T. (2016). Simple cortical and thalamic neuron models for digital arithmetic circuit implementation. *Frontiers in neuroscience*, *10*, 181.

Neher, E., & Sakmann, B. (1976). Single-channel currents recorded from membrane of denervated frog muscle fibres. *Nature*, *260*, 799–802.

Nelson, S. B., Sugino, K., & Hempel, C. M. (2006). The problem of neuronal cell types: a physiological genomics approach. *Trends in neurosciences*, *29*(6), 339–345.

Neuralink. (2023). *Neuralink company site web.* Online. Retrieved from https://neuralink.com/ (consulted September 7, 2023)

Ng, A., Jordan, M., & Weiss, Y. (2001). On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, *14*.

Niedermeyer, E., & da Silva, F. L. (2005). *Electroencephalography: basic principles, clinical applications, and related fields*. Lippincott Williams & Wilkins.

Obien, M. E. J., Deligkaris, K., Bullmann, T., Bakkum, D. J., & Frey, U. (2015). Revealing neuronal function through microelectrode array recordings. *Frontiers in neuroscience*, *8*, 423.

Ogawa, S., Lee, T.-M., Kay, A. R., & Tank, D. W. (1990). Brain magnetic resonance imaging with contrast dependent on blood oxygenation. *proceedings of the National Academy of Sciences*, *87*(24), 9868–9872.

Pardalos, P. M., Mavridou, T., & Xue, J. (1998). The graph coloring problem: A bibliographic survey. In *Handbook of combinatorial optimization* (pp. 1077–1141). Springer.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1996). *Numerical recipes in fortran 90 the art of parallel scientific computing*. Cambridge university press.

Purves, D., Augustine, G., Fitzpatrick, D., Katz, L., LaMantia, A., McNamara, J., ... others (2001). The organization of the nervous system. *Neuroscience. Purves D, Augustine GJ, Fitzpatrick D, Katz LC, LaMantia AS, McNamara JO, et al, editors. Sunderland, MA: Sinauer Associates*.

Rall, W. (1962). Theory of physiological properties of dendrites. *Annals of the New York Academy of Sciences*, *96*(4), 1071–1092.

Rasamuel, M., Gaffé, D., Levi, T., & Miramond, B. (2019, Oct). Synchronous approach for modeling spiking neurons. In *2019 ieee biomedical circuits and systems conference (biocas)* (pp. 1–4). Nara, Japan. doi: 10.1109/BIOCAS.2019.8919084

Raymond, P. (2008). Synchronous program verification with lustre/lesar. *Modeling and Verification of Real-Time Systems*, 7.

Ressouche, A., Gaffé, D., & Roy, V. (2008, Aug). Modular compilation of a synchronous language. In R. Lee (Ed.), *Software engineering research, management and applications* (Vol. 150, pp. 151–171). Springer. doi: 10.1007/978-3-540-70561-1

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Rümmer, P. (n.d.). *Luke webpage*. Retrieved from http://www.it.uu.se/edu/course/homepage/pins/vt11/lustre

Sarray, I. (2019). *Conception de systèmes de reconnaissance d'activités humaines* (Doctoral dissertation, COMUE Université Côte d'Azur (2015-2019)). Retrieved from https://theses.hal.science/tel-02145417

Sarray, I., Ressouche, A., Moisan, S., Rigault, J.-P., & Gaffé, D. (2017, Oct). An activity description language for activity recognition. In *2017 international conference on internet of things, embedded systems and communications (iintec)* (pp. 177–182). Gafsa, Tunisia.

Schemmel, J., Brüderle, D., Grübl, A., Hock, M., Meier, K., & Millner, S. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *2010 ieee international symposium on circuits and systems (iscas)* (pp. 1947–1950).

Schneider, K. (2009). *The synchronous programming language quartz* (Internal Report No. 375). Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern.

Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., & Plank, J. S. (2017). A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*.

Serb, A., Corna, A., George, R., Khiat, A., Rocchi, F., Reato, M., ... others (2020). Memristive synapses connect brain and silicon spiking neurons. *Scientific reports*, *10*(1), 2590.

Shi, J., & Malik, J. (2000). Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, *22*(8), 888–905.

Singer, W., Engel, A. K., Kreiter, A. K., Munk, M. H., Neuenschwander, S., & Roelfsema, P. R. (1997). Neuronal assemblies: necessity, signature and detectability. *Trends in cognitive sciences*, *1*(7), 252–261.

Singer, W., & Gray, C. M. (1995). Visual feature integration and the temporal correlation hypothesis. *Annual Review of Neuroscience*, *18*(1), 555-586.

Song, S., Miller, K. D., & Abbott, L. F. (2000). Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, *3*(9), 919–926.

Sorensen, M., DeWeerth, S., Cymbalyuk, G., & Calabrese, R. L. (2004). Using a hybrid neural system to reveal regulation of neuronal network activity by an intrinsic current. *Journal of Neuroscience*, *24*(23), 5427–5438.

Stein, R. B., Gossen, E. R., & Jones, K. E. (2005). Neuronal variability: noise or part of the signal? *Nature Reviews Neuroscience*, *6*(5), 389–397.

Stevenson, P. A., & Kutsch, W. (1987). A reconsideration of the central pattern generator concept for locust flight. *Journal of Comparative Physiology A*, *161*, 115–129.

Stimberg, M., Brette, R., & Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife*, *8*. doi: 10.7554/eLife.47314

Stimberg, M., Goodman, D. F., & Nowotny, T. (2018). Brian2genn: a system for accelerating a large variety of spiking neural networks with graphics hardware. *bioRxiv*, 448050.

Stosiek, C., Garaschuk, O., Holthoff, K., & Konnerth, A. (2003). In vivo two-photon calcium imaging of neuronal networks. *Proceedings of the National Academy of Sciences*, *100*(12), 7319–7324.

Stuart, G., Spruston, N., & Häusser, M. (2016). *Dendrites*. Oxford University Press.

Stufflebeam, R. (2008). *Research web site.* Online. Retrieved from https://mind.ilstu.edu/curriculum/neurons_intro/neurons_intro.html (consulted September 7, 2023)

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, *105*(12), 2295–2329.

Taud, H., & Mas, J. (2018). Multilayer perceptron (mlp). *Geomatic approaches for modeling land change scenarios*, 451–455.

Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., & Maida, A. (2019). Deep learning in spiking neural networks. *Neural networks*, *111*, 47–63.

Thakur, C. S., Molin, J. L., Cauwenberghs, G., Indiveri, G., Kumar, K., Qiao, N., ... others (2018). Large-scale neuromorphic spiking array processors: A quest to mimic the brain. *Frontiers in neuroscience*, *12*, 891.

The University of Queensland, A. (2023). *How to measure brain activity in people.* Online. Retrieved from https://qbi.uq.edu.au/brain/brain-functions/how-measure-brain-activity-people (consulted September 7, 2023)

Tuckwell, H. C., Wan, F. Y., & Rospars, J.-P. (2002). A spatial stochastic neuronal model with ornstein–uhlenbeck input current. *Biological cybernetics*, *86*(2), 137–145.

Université Côte d'Azur (UCA). (2018-2021). *Site web of the artefact project.* Online. Retrieved from https://univ-cotedazur.fr/recherche-innovation/structures-de-recherche/academies-dexcellence/academie-dexcellence-homme-idees-et-milieux/artefact-etude-et-conception-dappareils-numeriques-pour-la-substitution-sensorielle-et-la-cognition-etendue (Project coordinated by Daniel Gaffé)

Vassanelli, S., & Mahmud, M. (2016). Trends and challenges in neuroengineering: toward "intelligent" neuroprostheses through brain-"brain inspired systems" communication. *Frontiers in neuroscience*, *10*, 438.

Walter, F., Röhrbein, F., & Knoll, A. (2016). Computation by time. *Neural Processing Letters*, *44*, 103–124.

Wang, R., Hamilton, T. J., Tapson, J., & van Schaik, A. (2014, June). An fpga design framework for large-scale spiking neural networks. In *2014 ieee international symposium on circuits and systems (iscas)* (pp. 457–460). Melbourne, Australia.

Wang, R. M., Thakur, C. S., & Van Schaik, A. (2018). An fpga-based massively parallel neuromorphic cortex simulator. *Frontiers in neuroscience*, *12*, 213.

Warwick, K., Xydas, D., Nasuto, S. J., Becerra, V. M., Hammond, M. W., Downes, J., ... Whalley, B. J. (2010). Controlling a mobile robot with a biological brain. *Defence Science Journal*, *60*(1), 5–14.

Waxman, S. G. (1980). Determinants of conduction velocity in myelinated nerve fibers. *Muscle & Nerve: Official Journal of the American Association of Electrodiagnostic Medicine*, *3*(2), 141–150.

Welsh, D. J., & Powell, M. B. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, *10*(1), 85–86.

Zucker, R. S., & Regehr, W. G. (2002). Short-term synaptic plasticity. *Annual review of physiology*, *64*(1), 355–405.

List of Abbreviations

ALIF Adaptive Leaky-Integrate-&-Fire
ALM Adaptive Logic Module
ANN Artificial Neural Network
ASIC Application-Specific Integrated Circuit
BCI Brain-Computer interface
CLEF Compiler of Light Esterel modules Finaliser
CLEM Compiler of Light Esterel Modules
CPG Central Pattern Generator
CMA Center of Applied Mathematics
DAE Differential Algebraic Equation
DBS Deep Brain Stimulation
DSSN Digital Spiking Silicon Neuron
DSP Digital Signal Processing
EEG Electroencephalography
EPSP Excitatory Post-Synaptic Potential
FPGA Field Programmable Gate Array
FF First-Fit
FSM Finite States Machine
GABA Gamma-Amino-Butyric Acid
HH Hodgkin-Huxley
IF Integrate-&-Fire
INRIA National Institute for Research in Computer Science and Control
IPSP Inhibitory Post-Synaptic Potential
IZH Izhikevich
LDO Largest Degree Ordering
LEC Light Esterel Compiled codes
LEAT Laboratory of Electronics, Antennas and Telecommunications
LIF Leaky-Integrate-&-Fire
LOOM Linker Of Object Modules
LTD Long-Term Depression
LTP Long-Term Potentiation

MEA Micro-Electrode Array
MSB Most Significant Bit
ms millisecond
NPU Neural Processing Unit
ODE Ordinary Differential Equation
RAM Random Access Memory
RC Reservoir Computing
ROM Read-Only Memory
SNN Spiking Neural Network
STDP Spike-Timing-Dependent Plasticity
STP Short-Term Plasticity
WP Welsh-Powell

List of figures

List of tables

List of examples

List of Algorithms

Annexes

A *Light Esterel* syntax overview

In the following, we detail the different syntaxes of the *Light Esterel* language, as the different available statements, operators and expressions.

Module body : Imperative syntax

```
<Data declaration>
<Interface declaration>

Mealy Machine
-- register declaration
Register:
    <reg_name1> : <0 or 1> : <reg_name1_next>,
    <reg_name2> : <0 or 1> : <reg_name2_next>,
    ...;
-- equations definition
<reg_or_signal_name1> = <quadrival expression1>;
<reg_or_signal_name2> = <quadrival expression2>;
...;
end

```

where :

- `<reg_name> : <0 or 1> : <reg_name_next>` : syntax declaration of boolean registers. First, the register's name must be declared, then its first value (0 or 1), and finally the register "next" value name.
- `<reg_or_signal_name> = <quadrival expression>` : equations of the signals (outputs or locals) or registers ("next"). The equations are boolean expressions based on the current inputs or the registers value.

Note that the "register declarations" part is not mandatory, when registers are not needed.

Module body : Automaton textual syntax

The *Galaxy* tool offers the possibility to model automaton graphically. It gives the choice between different models : basic automaton, parallel automata, *Light Esterel* or SyncCharts. The *Galaxy* tool generates the according *Light Esterel* file. However, it is also possible to use the textual syntax to describe automaton.

```
1  <Data declaration>
2  <Interface declaration>
3
4  automaton
5  -- states declaration
6  state:
7     <state1_id> [run <module_name1>[<new>\<old>,...]] [/<signals_to_emit>]
       [geometry: <val1,val2> val3];
8     <state2_id> [run <module_name2>[<new>\<old>,...]] [/<signals_to_emit>]
       [geometry: <val1,val2> val3];
9     ...;
10    <stateN_id> [final] [run <module_nameN>[<new>\<old>,...]] [/<signals_to_emit>]
       [geometry: <val1,val2> val3];
11 -- transitions definition
12 transition:
13   initial -> [<quadrival_expression>] [/<signals_to_emit>] <transition_type>
       <next_state_id> [geometry: <val1,val2>];
14   <state_id> [<quadrival_expression>] [/<signals_to_emit>] <transition_type>
       <next_state_id> [geometry: <val1,val2>];
15   ...;
16 end
17
```

In the states declaration :

- `<state_id>` : refers to the name or identification of the created state.
- `run <module_name>[<new>\<old>,...]` : an external module can be associated with a state. That module (sub-module) will be executed when inside the given state. The "run" operator is used to link the sub-module to the state. And to link the main module's signals to the sub-module interface, in the bracket is the list of signals where "new" refers to the name of signals in the main module and "old" refers to the sub-module's input/output signal names.
- `/<signals_to_emit>` : is the list of signals that are emitted continuously when in the given state (as in a Moore machine style). The list of signals are separated by the coma character ",".
- `geometry: <val1, val2> val3` : is to specify the coordinates of the state (<val1, val2>), and the size of the circle representing the state (val3), for graphical representation. The values are in float.
- **`final`** : this keyword is to specify that the declared state is a terminal state, *i.e.* the automaton computation is over when the terminal state(s) is (are) reached.

In the transitions definition :

- /<signals_to_emit> : is the list of signals (separated by ",") that are emitted when the transition is satisfied (like Mealy machine style). The signals are emitted for only 1 logical instant.
- geometry: <val1, val2> : for graphical representation, <val1, val2> are the coordinates of the transition label.
- [<quadrival_expression>] : boolean expression or conditions that must be satisfied for the transition to occur between two states. If specifying the quadrival constant "TT" as condition, it means the transition is always true and occurs.
- <transition_type> : there are 3 types of transition, which define how the instructions being executed in a state are treated when the transition is triggered. Let's consider an instruction *P* being executed when in a particular state. The transition to a new state can be :
  - strong-> : refers to a *strong transition*, it forces the instruction *P* being executed to terminate when the transition's condition is satisfied.
  - weak-> : refers to a *weak transition*, it lets the instruction *P* ends when the transition's condition is satisfied.
  - normalterm-> : refers to a *normal transition*.

Statements

The module behavior is expressed in the module body using a set of *control operators*. There are two kinds of operators : usual programming language operators and operators devoted to deal with logical time, as presented in (Ressouche et al., 2008).

Non temporal statements

nothing	Does nothing and terminates instantaneously
emit <signal>	Sets <signal> status to "present" during the current logical instant
noemit <signal>	Sets the <signal> status to "absent" during the current logical instant
sustain {<signal1>} **until** <signal2>	Continuously emits <signal1> and stops when <signal2> occurs
present <signal> {P1} **else** {P2}	If <signal> is present then P1 is executed, else P2
if <condition> {P1} **else** {P2}	If <condition> is true then P1 is executed, else P2
P1 >> P2	P1 is executed then P2
P1 || P2	P1 and P2 start simultaneously, the instruction is terminated when P1 and P2 are both done
abort P **when** <signal>	P is executed normally until <signal> is present
strong abort {P} **when** <signal/expression>	Kills P when <signal> occurs or <expression> is satisfied
weak abort {P} **when** <signal/expression>	Executes P one last time before killing it
loop {P}	Infinite loop : P is executed and restarts when it is done
repeat <integer value> {P}	Finite loop : P is executed a number of <integer value> times
local <signal> {P}	The scope of <signal> is limited to P
run M [<new>\<old>,...]	Calls the module M, and links its interface signals (referred as "old") to "new" signals

Table 1: *Overview of Light Esterel non temporal operators.*

Temporal statements

pause	Stops until the next reaction (instant)
wait <signal>	Waits for the presence of <signal>
wait <integer value> <signal>	Terminates with <integer value> presences of <signal>
halt	Stops the module forever

Table 2: *Overview of Light Esterel temporal operators i.e. to manipulate logical instants.*

Expressions

$<signal>	To get the value associated to <signal> at the current logical instant
== != <= < > >=	Values comparison operators
+ - * /	Predefined value operators
FF or **TT**	Quadrival constants
& or **|** or **~** or **and** or **or** or **not** or **abs** or **nand** or **nor** or **xor**	Quadrival operators
;; or --	Comments

Table 3: *Overview of Light Esterel expressions.*

B Formal methods in Neurosciences

B.1 Neural archetypes

(De Maria et al., 2016) present a new discrete version of the LIF neuron. In the classical version, both neuron state and time are possibly infinite. The membrane potential p, at a time $t \in [0; +\infty]$, integrates the values of the action potentials received from its input neurons, and also what remains from previous inputs. The membrane potential can be described by :

$$p(t) = \sum_{e=0}^{+\infty} \sum_{j=1}^{m} r^e x_j(t-e)$$

where $e \in \mathbb{R}^+ \cup +\infty$ is the time elapsed until the current time t; r is the remaining potential coefficient, usually a function $r(t) = exp(-\alpha t)$ defined for $t \in [0; +\infty]$, with α a positive constant; x_j are the neuron's spike inputs $\in \{0, 1\}$ (m is the number of the neuron's predecessors). For simplicity, all the synaptic weights are equal to 1.

As the inputs effect exponentially decrease with time, they assumed that inputs received a long time ago can nevertheless be neglected. So they define a sliding integration time window of length σ, where at each time t, inputs older than σ are not taken into account in the calculation of the membrane potential. σ depends on an arbitrary error ϵ, such that the greater (resp. lower) ϵ, the lower (resp. greater) σ. Therefore, the time dependence of the membrane potential is not anymore infinite but bounded to $[t-\sigma, t]$. Finally, by discretizing each time step with t, the computation of the membrane potential now depends on a finite memory size, and can be defined as a simple sum :

$$p(t) = \sum_{e=0}^{\sigma} r^e \sum_{j=1}^{m} x_j(t-e)$$

This discrete version of the LIF model, characterized by 3 parameters (remaining potential coefficient, the size σ of the integration window and the potiential threshold), can be implemented in the synchronous language Lustre. And by using this model, they encoded neural micro-circuits, referred as *archetypes*, on which they applied model checking to verify temporal properties. An archetype is a specific graph of a few neurons that have biologically significant structures and behaviors. These archetypes correspond to elementary and fundamental elements of neural information processing. From the biological point of view, archetypes constitute the normalized form of potentially bigger and topologically more complicated neuronal circuits. Every micro-circuit, event with many neurons, can theoretically be reduced to one of the few existing archetypes. The figure B.2 shows basic neuronal archetypes they implemented with their discrete model.

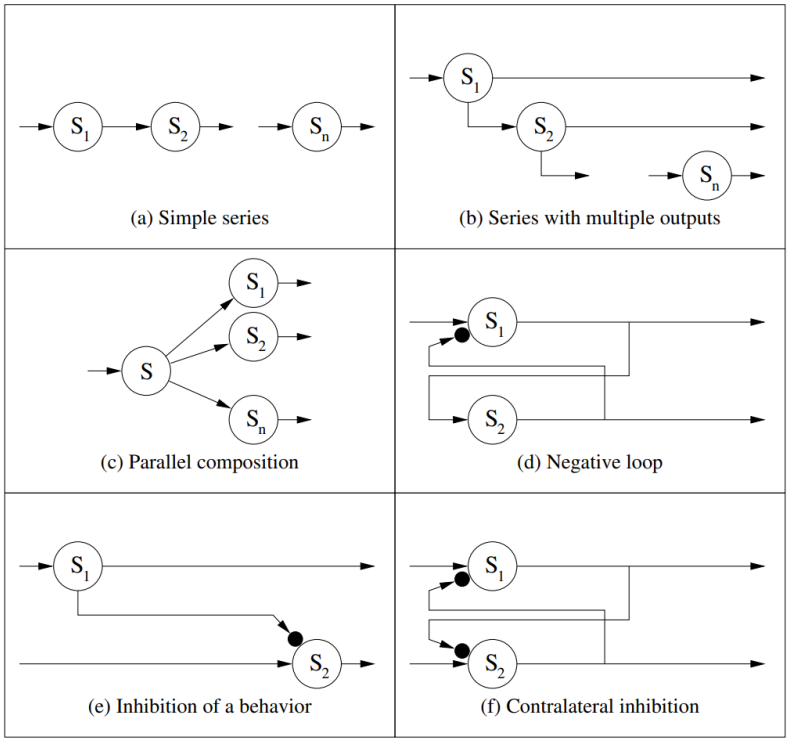


Figure B.2: *Basic neuronal archetypes extracted from (De Maria et al., 2016). Circles represent neurons based on their discrete neuron model and arrows represent the inputs, outputs and connections between neurons. Arrows ending with black dots are inhibitory connections, and excitatory otherwise. (a) The simple series is a sequence of neurons where each neuron receives as input the output of the preceding one. (b) The series with multiple outputs is the simple series where all the neurons output are considered. (c) The parallel composition is multiple neurons in parallel receiving inputs from the same neuron. (d) The negative loop is composed of one neuron exciting another neuron, which in response inhibits the first one. (e) The inhibition of a behavior is two neurons being stimulated while one is inhibiting the other. (f) The contralateral inhibition is two neurons being stimulated while mutually inhibiting each other.*

With the use of model checkers, they automatically verified some temporal properties on these archetypes. For example, two simple series with different neuron parameters and different lengths can always output the same spikes sequence. So with adapted parameters, this example shows the possibility to reduce a micro-circuit to have the same behavior. They compared different Lustre model checkers (see figure B.3), and it turns out that *Kind2* is the most powerful one capable of proving more general properties, compared to other Lustre model checkers : *Lesar* (Halbwachs & Raymond, 1999), *Nbac* (Jeannet, 2003), *Luke* (Rümmer, n.d.) and *Rantanplan* (Franzén, 2006).

	lesar	nbac	luke	rantanplan	kind2
Simple series (prop1)	No	Yes	very long time!	Yes	Yes
Simple series (prop2)	No	exit before!	Yes	very long time!	Yes
Series with multiple outputs	No	exit before!	Yes	Yes	Yes
Parallel composition	No	exit before!	Yes	Yes	Yes
Negative loop	No	exit before!	Yes	Yes	Yes
Inhibition of a behavior	No	Yes	Yes	Yes	Yes
Contralateral inhibition	No	Yes	Yes	Yes	Yes

Figure B.3: *Extracted from (De Maria et al., 2016). Comparison of Lustre model checkers. It lists the model checkers that was able or not to prove properties on Archetypes.*

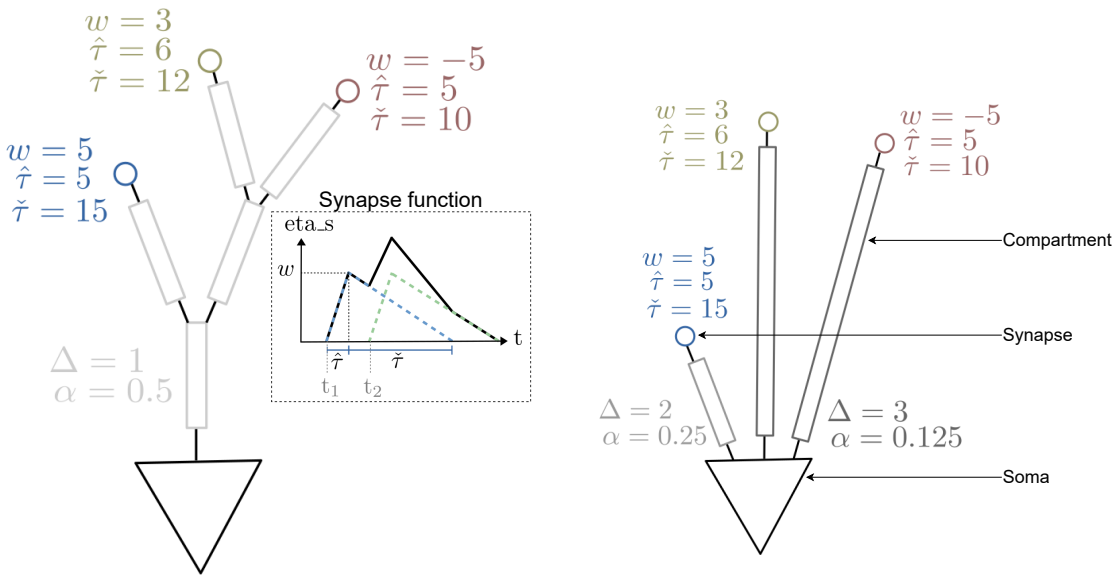
As a next step of their work, in (De Maria, L'Yvonnet, Gaffé, Ressouche, & Grammont, 2017) they study the coupling of these archetypes. If the resulting circuit do not perform a more complex function, then it can be reduced to a smaller archetype. But if a new biologically relevant function is identified, it can lead to the creation of a whole new archetype. In (De Maria et al., 2022), they use theorem provers as a new formal method approach to verify temporal properties compared to the model checking approach. They concluded that both approaches have their pros and cons, and they should be used together in a pragmatic way. Theorem provers are capable to prove more general properties but proofs can be long and an expert is needed to help and drive the results, while model checkers are faster and automatic, but cannot prove properties at the desired level of generality. With the exploration of archetypes coupling, different functions may emerge, like cell assemblies (Hebb, 1949b).

B.2 Abstract neuron model

Despite the evolution of technologies and the multiple neuroscience studies, it is still difficult to understand how complex brain functions emerge from neuron activities. Among several theories, one has been proposed by (Hebb, 1949b) to explain the neural representation of concepts: the brain functions come from neural properties and their ability to constitute neuron assemblies. A neuron assembly is a small set of strongly connected neurons which specificity is to synchronize activity in response to specific stimuli (Singer & Gray, 1995; Harris, 2005; Huyck & Passmore, 2013). The synchronization of activity corresponds to the firing of all the neurons within a small time window of tens of milliseconds or less (Singer et al., 1997). The activity of an assembly would represent the perception of one stimulus, therefore the activation of multiple assemblies would represent more complex stimuli (concepts) as a mix of multiple stimulus. This hypothesis has been increasingly supported by biological, theoretical and simulation data that have proven their existence. However, it is difficult to study neuron assemblies *in vivo* and *in vitro* because, first, actual technologies are not able to record neurons individually and simultaneously in large biological neural network thus to identify assemblies precisely; and second, the dynamic complexity of formation and behavior of assemblies rely on processes that are not fully understood yet. Indeed, they can form and reconfigure, multiple assemblies can activate simultaneously, one assembly can activate other assemblies in a chain manner, one neuron may participate in multiple assemblies, etc. Additionally, the large number of neurons in the brain makes experimental studies on biological neurons even more complex as the combinatorial nature of possibilities explode. Consequently, *in*

silico simulation becomes a good option to overcome experimental limits and to test hypothesis related to the formation of neuron assemblies.

In that context, it is important to choose the right neuron model when studying the behavior and the functions of neuron assemblies. Indeed, complex dynamics of these assemblies emerge from the particular properties of individual neurons. It is particularly important to choose a neuron model that is "simple" so it makes easier the study of the dynamic mechanisms, but also "biologically plausible" enough to make biological accurate predictions. Most existing neuron models do not take into account the morphology of neurons, particularly the dentrites, which can significantly affect the integration of action potentials and the overall function of the neuron. The *Hodgkin-Huxley* (HH) model (Hodgkin & Huxley, 1952) is one example that does incorporate dentrites model derived from the cable theory (Rall, 1962). Even though these models are biologically accurate, they come with a large system of differential equations (see chapter 2 section 2.1.3) that describe multiple biological mechanisms, even on the molecule level. With the numerous parameters to adjust, they would be too complex, when in network, to study assemblies. For these reasons, (Guinaudeau, 2019) proposed a new abstract neuron model that incorporates dentrites integration, while remaining simple in terms of number of parameters. Since their long-term goal is to use computer-assisted reasoning about neurons in networks, they based their work on the formal approach to describe and to analyze each part of their abstract model illustrated in figure B.4. This way, they could use formal methods, such as model checker, to verify properties on their model.



(a) *Abstract neuron* `neuron1` **(b)** *Reduced neuron* `neuron2`

Figure B.4: *Modified figures extracted from (Guinaudeau, 2019). Illustration and example of their abstract neuron (a) and its reduced form (b). In (a), the synapse function is illustrated according to the three parameters w, $\hat{\tau}$ and $\check{\tau}$ when receiving a spike at times t_1 and t_2.*

The abstract neuron model shown in B.4(a) consists of a soma, synapses, and compartments representing the dendrites tree. Each synapse is characterized by a state function `etat_s` with

3 parameters : a stimulation weight (w), a rise period $\hat{\tau}$ to reach w when a spike occurs, and a fall period $\check{\tau}$ to return to 0. When two consecutive spikes arrive at the same synapse, the total stimulation induced at that synapse is the sum between the induced weight values of the first spike and the weight induced values of the second spike. Each compartment is characterized by two parameters : a time Δ required to cross the compartment and an attenuating factor α that reduces the stimulation (weight) as it passes through the compartment. Therefore, the total stimulation of the soma is the sum of all the synapse stimulations, which are themselves affected by the route, delays and attenuations in the dentrite tree. The authors in (Guinaudeau, 2019) used the Kind2 model checker to prove the property of equivalence between the abstract neuron model with a complex dendrite tree (referred to as `neuron1` in figure B.4) and a reduced model with a linear dendrite tree (referred to as `neuron2`), in terms of input/output functions, given the same constant inputs. However, they reported that the model checker's limits were reached when the inputs became more complex.

Both works have used formal methods to describe and design neuron models or neural network models, in order to analyze and extract relevant properties by using model checking. Their works were promising as they confirmed that thanks to model checkers, it is possible to analyze neural models in a systematic and efficient way, to extract useful insights about their behavior and functions. In that context, it inspired our works to model neuron and neural network models using the synchronous language *Light Esterel* that have been developed in the laboratory. In the next sections, we explained the implementation of our models, and we will describe the results we obtained with *Kind2* regarding our applications.

C *SynchNN* : IDs selector flowchart for reading process

The reading process of the modified matrix version is controlled by a state machine, as depicted in the flowchart in figure C.5. The module starts by selecting and outputting the first synapse ID in the first column index as *o_IDsyn*, and the corresponding column index as *o_IDneurLocal*.

If the number of processed synapses reaches the maximum number of synapses in the NPU, the module emits the *o_processIDneur* signal to notify the CONTROLLER UNIT that the last synapse is being processed, and the neuron module can be executed one final time. Additionally, the *o_endIDsynList* signal is emitted to indicate that all synapse IDs have been processed.

If the maximum number of synapses has not been reached yet, the module checks the next synapse ID within the same column. If the next ID is equal to 0, it means that the current synapse ID is the last one in the column. In this case, the module emits the *o_processIDneur* signal and waits for the *i_nextIDneur* signal from the CONTROLLER UNIT.

On the other hand, if the next ID is not equal to 0, the current synapse is processed, and the module waits for the *i_nextIDsyn* signal from the CONTROLLER UNIT to proceed to the next synapse.

This state machine-based approach allows for efficient selection and processing of synapse IDs from the modified matrix version, ensuring that all synapses associated with a specific neuron are properly processed.

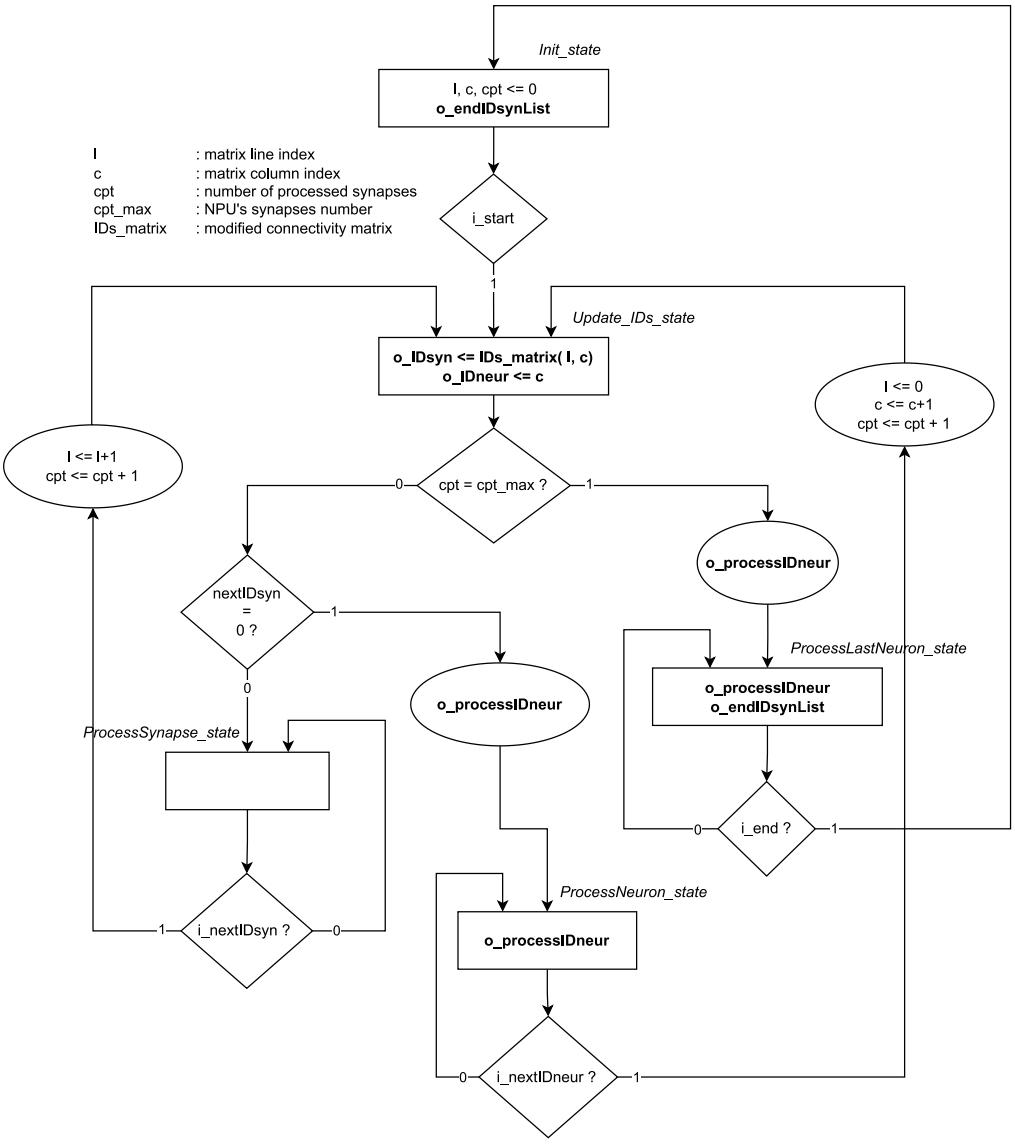


Figure C.5: *Flowchart reading process of the modified connectivity matrix by the IDs selector.*

D *SynchNN* : functional validations

The following is the functional validation of each unit in *SynchNN*. We detail the simulation of each unit to validate their function. The simulations are based on the SNN in the example figure 5.8. We have chosen the Izhikevich (IZH) and the Short-Term-Plasticity (STP) models for the neurons and the synapses, respectively. The neuron model and synapse model parameters are not important as no specific application is aimed at this validation step. So all the neurons are configured the same, as for the synapses.

For the sake of understanding, we reduced the number of simulations to the units in the generated NPU 2 (red) : composed of 3 neurons and 6 synapses. As the total number of signals is huge, only important signals are shown that confirm the function of the unit. Note that the controller unit validation is merged with the other units validation as they are dependants.

Reception unit validation

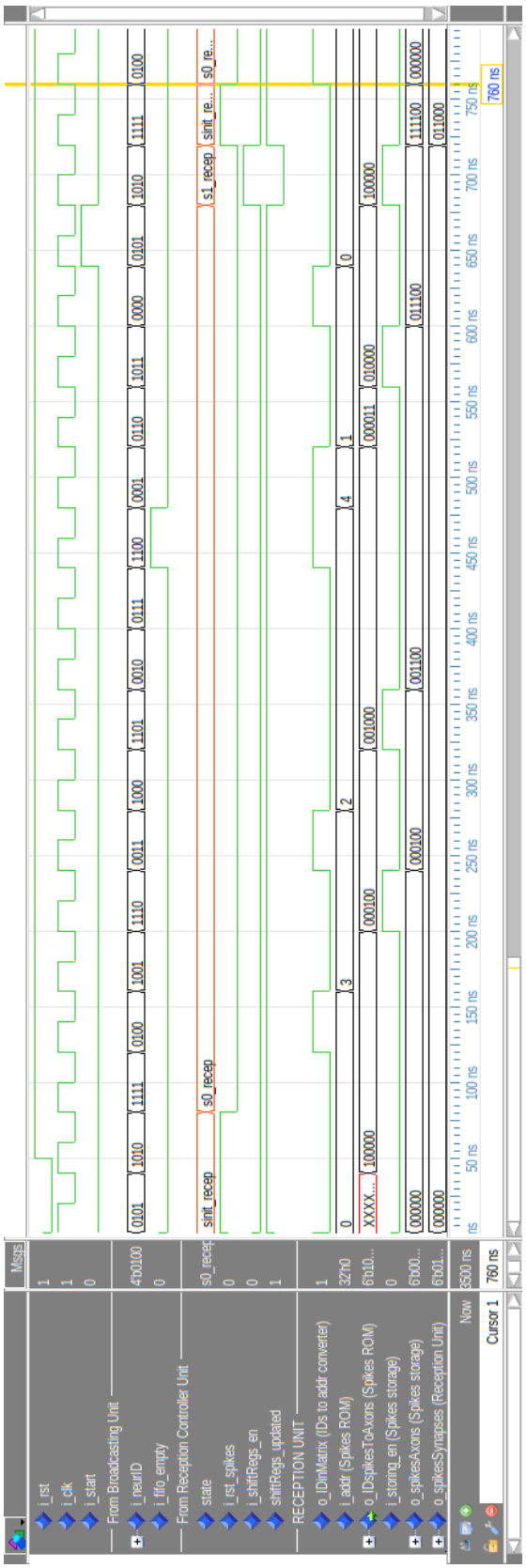


Figure D.6: *Modelsim simulation of the Reception unit based on the NPU 2 configurations in 5.8.*

The simulation in figure D.6 comes from the generated reception unit of the NPU 2 from the exemple in 5.8. We focus on how random input neuron IDs are transformed into spikes within a time-step. So in this simulation, we show the random signals (neuron IDs and *i_fifo_empty*) supposed to come from the broadcasting unit, the signals and states of the reception controller unit and finally the signals of the reception unit modules. Note that the beginning of a time step is flagged by the *i_start* signal when '1'.

The reception unit conversion process directly starts after the reset signal, shown by the transition from the *sinit_recept* state to the *s0_recep* step in the controller. When the neuron ID is part of the pre-IDs list of the NPU, the signal *o_IDinMatrix* is '1'; an example is the ID "0100". One clock cycle later, the *IDs to addr converter* module outputs the address of the spikes to be routed to the correct synapses; the spikes generated by "0100" are stored at the address 3 of the ROM. One clock cycle later, from that address the *spikes ROM* module outputs the stored spikes array to the *Spikes storage* module; the ID "0100" is addressing a spike to the 4th synapse of the NPU. The spikes storage module collects one cycle later the spikes when its enable storing signal is '1'. The collected spikes are shown by the *o_spikesAxons* signal. So in total, the conversion of an ID to the storage of the associated spikes requires 4 clock cycles. In the case the ID is not part of the pre IDs list or the signal *i_fifo_empty* is '1' respectively, the input neuron ID is ignored or the spikes are not stored; see *e.g.* "1100" arround 450 ns. Again, when the signal *i_fifo_empty* is '1', it means the input neuron ID is actually coming from an empty FIFO and the ID has already been sent before. At the beginning of a new time-step, the reception controller unit sends the enable signal *i_shiftRegs_en* to shift all the shift registers once. As a result, we can observe the changing values of the *o_spikes_Synapses* output which is all the spikes that travelled the shift registers (axons) and can be processed by the processing unit. For understanding purpose, the axonal delays of all the synapses have been set to $0 \times dt$, there is no axonal delay involved, the received spikes are directly transmitted by the reception unit at the next time-step. Finally, the signal *shiftRegs_updated* is re-emitted to notify that all the registers have been updated.

Processing unit validation

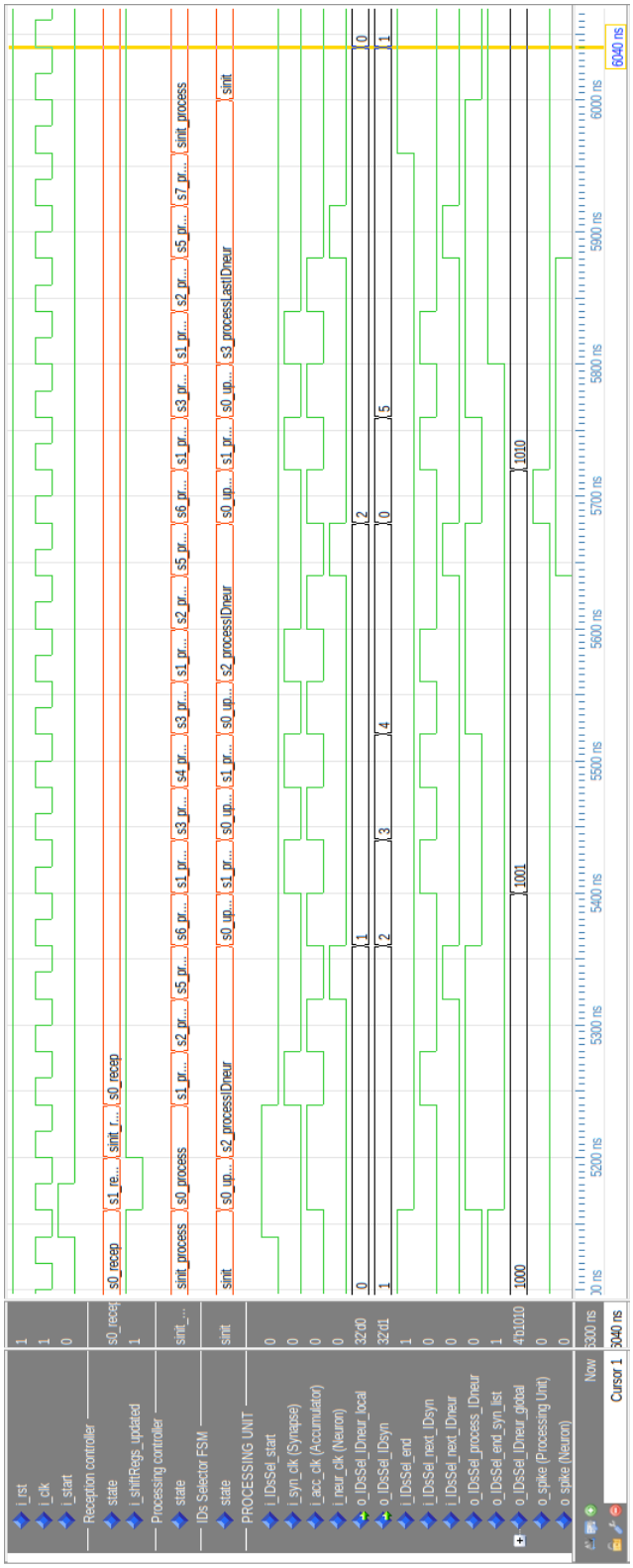


Figure D.7: *ModelSim simulation of the Processing unit based on the NPU 2 configurations in 5.8.*

The processing unit uses 1 neuron module and 1 synpase module to process all the neurons and synapses in the NPU in a time-multiplexed manner. For each neuron, it computes all its excitatory/inhibitory synpases first, and then it computes the neuron. The selection of the correct synapses and their parameters in the memories for a given neuron is made by the IDs selector module. As a reminder, the IDs are used as addresses for the memories. Note that we apply a minus 1 to the synapse IDs for addressing the memories, because they start at "1" instead of "0".

The processing unit is activated only after the *i_start* signal is '1' followed by the *shiftRegs_updated* signal emitted by the reception controller. At the beginning, the processing unit initiates the IDs selector module which outputs the IDs of the first synapse (*o_IDsSel_IDsyn*) and the neuron (*o_IDsSel_IDneur_local*) it stimulates. For each neuron, the accumulator is first reset to 0 by sending an impulsion on both signals *i_acc_clk* and *i_acc_rst* for one clock cycle. The next clock cycle, the synapse module is executed by sending an impulsion on *i_syn_clk*, while the next synapse ID is asked (see impulsion on *i_IDsSel_next_IDsyn*). The next clock cycle, the previous synapse stimulus is added in the accumulator (see *i_acc_clk* again). The two previous clock cycles are repeated until all the synapses related to the selected neuron have been processed. In that case, the signal *o_IDsSel_process_IDneur* is emitted. So the next clock cycle after the last accumulation, the neuron module is executed (see *i_neur_clk*), and the next neuron is asked (see *i_IDsSel_next_IDneur*). Then the process is repeated until all the neurons have been calculated, which means until *o_IDsSel_end_syn_list* is emitted. Note that when a neuron spikes, see arround 5550 ns for *o_neur_spike*, the neuron output module lasts during the whole neuron clock signal. Therefore, we use the *o_spike* signal to control the broadcasting instead, with the corresponding global ID (see *o_IDsSel_IDneur_global*). Finally, all the related FSMs of the processing unit are reset.

Broadcasting unit validation

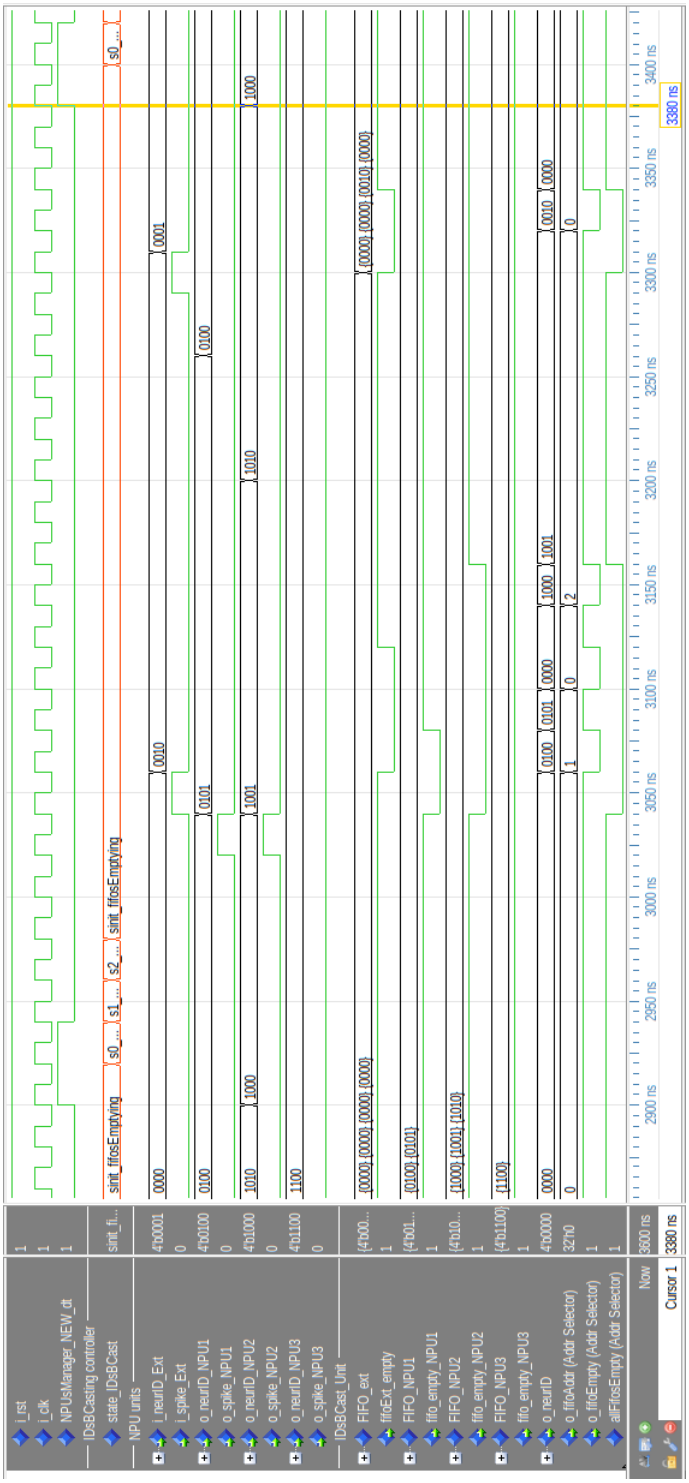


Figure D.8: *ModelSim simulation of the Broadcasting unit based on the SNN example in 5.8.*

The broadcasting unit receives the IDs of the neurons that have spiked in the network, *i.e.* from all the NPUs. In the simulation in figure D.8, the IDs of the neurons being processed by each NPU are received by the broadcasting unit, displayed by $o\_neurID\_NPU_i$. They come with their eventual spike signals $o\_spike\_NPU_i$. When a spike is emitted, it enables the writing of the neuron ID into the FIFO assigned to the NPU. During the simulation, $FIFO\_NPU_i$ displays the list of neuron IDs stored in a FIFO, and an empty signal shows the FIFO's state (empty/not empty). As long as the fifos are not all empty, *i.e. allFifosEmpty* is '0', the IDs of the not empty FIFOs are being read and broadcasted to the NPUs. When a FIFO is empty or when all the FIFOs are empty, the output signal *o_fifoEmpty* is set to '1' which notifies all NPUs that the current output *o_neurID* is from an empty FIFO.

Time step manager unit validation

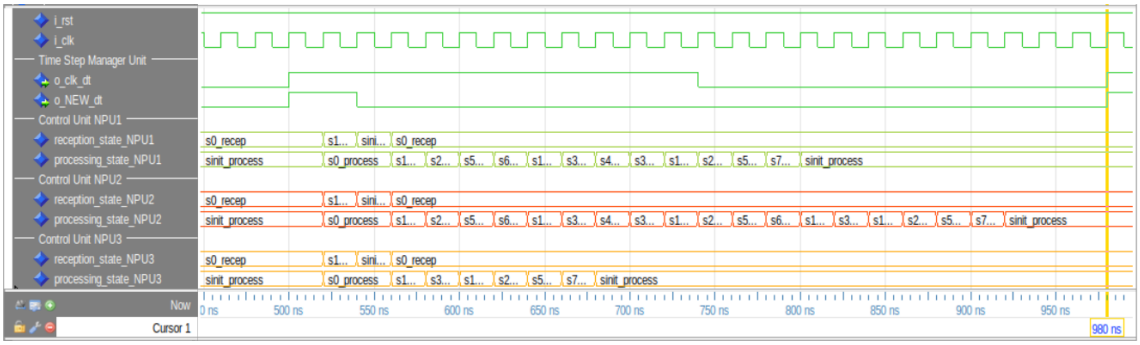


Figure D.9: *ModelSim simulation of the Time Step Manager unit and the Controller unit.*

The *clk_dt* clock signal represents the time constraint, it is configurable and is handled by the time step manager unit. On every rising edge of *clk_dt*, the time step manager unit emits the *o_NEW_dt* signal to all the NPUs which then start their calculations. The figure D.9 shows the global evolution of all the controller units of the 3 NPUs after the *o_NEW_dt* is '1'. Otherwise, they remain in their initial state. The figure also shows the most important rule of the architecture, which is to terminate all NPU's process before the beginning of a new time step.

Modélisation et conception par approche synchrone d'architectures neuronales hybrides biologique-artificiel

Marino RASAMUEL

Résumé

Alors que les Réseaux de Neurones Artificiels (RNA) continuent de progresser dans des domaines tels que l'apprentissage automatique, la robotique, les véhicules autonomes et le diagnostic de santé, un nouveau cadre d'application gagne du terrain à la fois dans les secteurs académique et industriel : la Neurobiohybridation. Ce domaine cherche à établir des connexions entre des neurones artificiels et biologiques dans le but de comprendre et potentiellement de réparer ou remplacer des fonctions cérébrales perdues suite à des maladies ou des accidents. Dans cette perspective, le développement de réseaux de neurones artificiels inspirés biologiquement, souvent appelés Réseaux de Neurones à Spikes (SNNs), est essentiel pour améliorer la compatibilité entre les systèmes neuronaux artificiels et biologiques. Cette thèse s'inscrit dans ce contexte en utilisant l'approche synchrone pour modéliser, mettre en œuvre et simuler des SNNs bio-inspirés et biomimétiques. En utilisant des vérificateurs de modèles, qui permettent de prouver ou d'extraire des propriétés des systèmes de manière formelle, notre objectif est d'acquérir une compréhension plus complète des comportements biologiques dans le future. Pour la première fois dans ce contexte, nous utilisons le langage Light Esterel pour atteindre nos objectifs. Nous démontrons son potentiel dans la mise en oeuvre de modèles neuronaux, initiant une bibliothèque de modèles pour explorer différents types de SNNs. Tout au long de cette thèse, nous avons développé un cadre complet basé sur Light Esterel pour modéliser, simuler et mettre en oeuvre divers modèles de SNNs. Pour aborder les expériences de neurobiohybridation, nous avons développé notre propre architecture matérielle, SynchNN, capable d'exécuter en temps réel des SNNs récurrents en utilisant notre bibliothèque de modèles. L'environnement de modélisation que nous avons développé est complété par un framework de simulation, en cours de développement, visant à réaliser des expériences de neurobiohybridation à l'avenir.

Mots-clés : Réseaux de neurones impulsionnels, Approche Synchrone, Langage Light Esterel, FPGA, neurobiohybridation

Abstract

As Artificial Neural Networks (ANNs) continue to advance in fields like machine learning, robotics, autonomous vehicles, and healthcare diagnostics, an application domain is gaining attraction in both academic and industrial sectors : Neurobiohybridization. This domain seeks to establish connections between artificial and biological neurons with the goal of understanding and potentially repairing or replacing lost brain functions due to disease or accidents. In pursuit of this, the development of biologically inspired artificial neural networks, often referred to as Spiking Neural Networks (SNNs), is essential to enhance compatibility between artificial and biological neural systems. This thesis fits into this context by using the synchronous approach to model, implement, and simulate bio-inspired and biomimetic SNNs. Leveraging model checkers, that allow to prove or extract properties in systems in formal manner, our aim is to gain a more comprehensive understanding of biological behaviors in the future. For the first time in this context, we utilize the Light Esterel language to achieve our objectives. We demonstrate its potential in implementing neural models, initiating a library of models for exploring different types of SNNs. Throughout this thesis, we developed an entire framework based on *Light Esterel* in order to model, simulate and implement various SNN models. To address neurobiohybridization experiments, we developped our own hardware architecture, SynchNN, capable of executing recurrent SNNs in real-time using our library of models. This framework we developed is completed with an on-going simulation framework aiming to conduct neurobiohybrid experiments in the future.

Keywords: Spiking Neural Networks, Synchronous Approach, Light Esterel language, FPGA, neurobiohybridization