# Synchronous Neural Networks for Cyber-Physical Systems

Keyan Themba Monadjem

February 2019

Supervisors: Partha S. Roop

Department of Electrical & Computer Engineering
The University of Auckland
New Zealand

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING

# Abstract

Cyber-Physical Systems (CPS), such as autonomous vehicles or smart power grids, use interactive machine learning modules for decision making. Conventional design approaches use multiple machine learning modules, often using Artificial Neural Networks (ANNs), to achieve the desired functionality. The approaches to verification and validation of these ANNs are generally either very difficult, time consuming and/or not fully reliable. A key feature missing is related to the use of ANNs in real-time systems, which demand the capability of worst-case analysis. In this thesis we introduce Synchronous Neural Networks (SNNs) as a new approach to the safe use of ANNs in CPS. SNNs provide synchronous semantics to ANNs. This enables real-time operation and facilitates static timing analysis of individual ANNs. We define these SNNs using the Esterel synchronous language and then implement them on the time predictable platform called T-CREST, which facilitates static timing analysis.

We propose Meta Neural Networks (MNNs) as a framework for the systematic composition of SNNs. This enables compositional system design using multiple SNNs and other synchronous functional components, while maintaining the synchronous semantics of the system. Synchronous MNNs allow for the creation of causal, deterministic, predictable controllers for CPS.

Misclassification is a major issue with input perturbation in ANNs. We combine MNNs with Run-time Enforcers (RE), which enforce a set of desired policies by transforming inputs and outputs when desired. The proposed solution is able to effectively deal with many misclassifications. Finally, we propose a tool that extends the ANN-library Keras to give it a MNN description capability. We then automatically generate synchronous C code, which is shown to perform even better than our earlier MNN implementation using Esterel.

We have developed CPS examples that range from an energy storage system for charging electric vehicles, to a traffic sign detection system for autonomous vehicles. The results of our approach show that the implemented MNNs can meet real-time, safety critical deadlines. Composing MNNs of multiple SNNs and RE provide an increase in the safety of autonomous vehicles, where the MNNs not only increase classification accuracy of the environment, but also detect misclassifications and allow the system to respond safely to these misclassifications. This thesis introduces novel techniques to the timing and functional verification of ANNs.

# Acknowledgements

Firstly, I would like to express my very great appreciation to my supervisor, Partha S. Roop. He expended a lot of time and a lot of effort to get me to the point where I am now, and I am exceedingly grateful for that. He guided me through the tenure of my thesis, showing me what needed to be done, but not doing it for me. I learned many new and interesting concepts and technologies while working with him and was able to produce a great thesis with original work.

Secondly, Hammond Pearce has my utmost thanks. He took the role as a secondary supervisor to me, even with his own work as a PhD student. He helped me many a time when I was stuck, had a question or needed a push in the right direction. We spent many hours covering my current line of work and getting me back on line.

Next is the research group with Partha Roop. Many thanks to Matthew Kuo, Nathan Allen and Jin Ro. All three gentlemen helped me numerous times during my time as a ME student.

A thanks to Yash Raje, who was a great lab and Master's partner, and someone I spent a lot of time working with.

Without Marc Katzef, University of Canterbury, I would not have been able to use the extremely well made MNN2C tool. He did a really great job creating it in his time spent at the University of Auckland, and I would like to thank him for his fantastic work.

Finally, I wish to thank my family; my mom Lynette, my dad Ara, my stepmom Sara, my aunt Mona, my brothers Damian and Liam, and last, but not least, a friend of many years, Alex. They all supported me throughout this degree and gave me the love and encouragement I needed.

# Publications and Planned Submissions

- P. S. Roop, H. A. Pearce, and K. Monadjem, "Synchronous neural networks for cyber-physical systems," *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pp. 1–10, 2018.
- K. T. Monadjem, H. Pearce, P. S. Roop, and S. Pinisetty, "Enhancing the safety of artificial intelligence in cyber-physical systems using run-time enforcement and synchronous neural networks," *IEEE Trans. Cybernetics*, 2019, Unpublished: in preparation.
- K. T. Monadjem, H. Pearce, P. S. Roop, and S. Pinisetty, "Dealing with adversarial perturbation in neural networks using synchronous neural networks combined with run-time verification," *IEEE Trans. Neural Netw. and Learning Sys., 2019*, Unpublished: in preparation.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) refer to a network of distributed controllers that are used to control adjoining physical processes [2] [32]. CPS encompass a wide variety of disciplines, from mechatronics to civil infrastructure (Figure 1.1). CPS operate in the physical world, where environments can be unpredictable and uncontrollable. CPS applications encompass real-time systems, where the systems need to satisfy a set of timing and functional requirements to ensure correct operation. Here, a missed deadline may result in catastrophic consequences, making these CPS highly *safety-critical*. These have strict timing and functionality requirements — any errors in control can result in physical damage, injuries, and/or fatalities [55]. Hence, the hardware and software processes used in CPS must operate in a predictable and reliable [33] [63] manner at all times. This introduces safety concerns for the CPS. This thesis focuses on ensuring such requirements for CPS, especially where Artificial Intelligence (AI) are concerned.

## 1.2 Artificial Intelligence (AI) and Machine Learning (ML)

Machine Learning (ML) is a field of data science where Artificial Intelligence (AI) modules are taught to learn large and/or complex data relationships [48]. AI come in many shapes and forms, from decision trees to Artificial Neural Networks (ANNs) [24]. AI was invented to fill a gap that humans cannot, where they can store a huge number of data relationships and learn

Figure 1.1: Examples of CPS [1] [68] [72] [42], from left to right, top to bottom: A) factories, B) robotics, C) power and electricity, and D) construction

new relationships where a human would struggle. AI is used all over industry, from industrial systems, to cybernetics and robotics (such as Figure 1.2).

AI is increasingly used in CPS, where safety is critical. However, the mathematical nature of AI is non-linear, leaving many AI applications difficult to verify. Additionally, as AI adapt to learn larger data sets and more complex relationships, the size and complexity of the systems increase to unmanageable proportions. While of these are easier to verify than others [36], as they grow in size, these techniques take more time and resources to implement. In CPS, it is essential for all components to be verified and validated, and thus the issue with AI in CPS is made clear: conventional techniques cannot be used to verify AIs for CPS.

The use of AI, notably the ANNs used in this work, in CPS is highly contended, as ANNs are not 100% reliable and this can result in fatal accidents, as reported in [16]. Thus, it is of utmost importance that any ANN used in CPS is verified to be safe for that system in all possible environments. Without that guarantee of safety the use of AI in such systems may pose undesirable risks [13].

Static verification methods do not carry over well to many types of AI, including ANNs. While some research groups can demonstrate successful functional verification methods [21] [26], these are often limited by the type and size of the ANN to be verified; and the timing verification

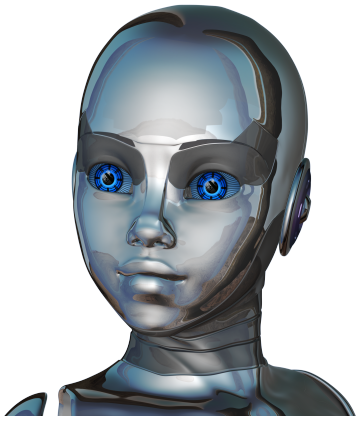


Figure 1.2: AIs are frequently used as the controllers of robots [18]

of ANNs has received scant attention.

The increase in research of AI is causing a convergence of conventional, model-driven approaches to verification and newer, data-driven approaches [69]. The integration of these two approaches is required using novel techniques. The problem posed is this: how can ANNs, implemented in the model-driven design of CPSs and trained using data-driven techniques, be verified as *safe*?

This thesis answers that question by introducing model-driven [69] Synchronous Neural Networks (SNNs) using the synchronous language Esterel. Timing analysis of these SNNs is done using Worst Case Execution Time (WCET) [70] analysis. Static verification techniques do not scale well with the size of system. We address the functional safety of these SNNs using the dynamic verification techniques of Run-time Enforcement (RE) and Run-time Verification (RV) that do scale with the size of the system, as the system is observed as a black box during run-time.

## 1.3 Contribution

This thesis addresses the issue of safe ANNs using synchronous semantics [5]. In this thesis, a new ANN library was created to demonstrate the benefits of synchronous ANNs and a tool chain is introduced that compiles Keras [14] trained ANNs to the created ANN library.

The major contributions of this thesis are as follows:

- A time predictable approach to ANNs is developed using the synchronous language Esterel [7]. These predictable ANNs are termed Synchronous Neural Networks (SNNs) and are defined using formal methods. Using T-CREST's Patmos processor architecture [62], due to its time-predictable nature, timing analysis of these SNNs is possible. These SNNs

provide a safe approach to implementing ANNs in CPS. The results of the SNNs are presented using a set of benchmarks.

- Meta Neural Networks (MNNs) are proposed as a framework for the composition of multiple SNNs. The thesis provides formal definitions for the MNNs and benchmarks are presented that show their efficacy.
- The combination of Run-time Enforcement (RE) and SNNs is investigated, with the intent to modify unsafe events. This provides functional safety for SNNs, something not covered in previous chapters. An Autonomous Vehicle (AV) case study is introduced for the purposes of this work and a simulation is created to demonstrate its efficacy.
- Run-time Verification (RV) of MNNs is proposed as an approach to dealing with input perturbation. This demonstrates a functional verification method for ANNs that have complex inputs, such as image classification Convolutional Neural Networks (CNNs), where RE is not an option. An AV case study is also created for this work, with an AV object detection simulation created to demonstrate the benefits of this proposition.

## 1.4 Thesis Structure

The remainder of the thesis is organised as follows:

Chapter 2 gives provides details of the concepts required to understand this research towards safe ANNs for CPS while reviewing related literature.

Chapter 3 introduces the concept of Synchronous Neural Networks (SNNs) and their timing properties. Formal definitions of SNNs and their related components are provided. Furthermore, the formal compositions of these SNNs, termed MNNs, and the usefulness of such MNNs in CPS is discussed. Lastly, a new Python tool chain that creates these SNNs from Keras [14] is also introduced.

Chapter 4 introduces the concept of Run-time Enforcement (RE) in combination with SNNs. This chapter provides formal definitions for the enforcers and the safety policies that are enforced. An Autonomous Vehicle (AV) case study is made for this chapter to show the efficacy of the RE of SNNs.

Chapter 5 proposes two different methods, used in tandem, to increase the safety of systems with complex inputs, such as object detection for AVs. The first method is the use of MNNs to increase the classification accuracy of SNNs. The second builds on Chapter 4, and introduces the use of Run-time Verification (RV) to increase the safety of CPS where Run-time Enforcement (RE) cannot do so. An Autonomous Vehicle (AV) object detection system is created as a complex MNN and adversarial input perturbation is introduced in this system.

Chapter 6 covers the conclusions drawn on the synchronous approach proposed in this thesis to creating safe ANNs.

# 2

# Background and Related Work

This chapter provides an overview of the terms and techniques used in this chapter. This chapter starts with an introduction on ANNs, their structure and their value in CPS. The aim is to provide a general understanding to the functionality of ANNs and their safety regarding CPS. The next section briefly introduces synchronous programming, and highlights Esterel as the language of choice for this thesis. An example of an Esterel program is given, with a quick run-down of how it functions and its syntax. The chosen method of Worst Case Execution Time (WCET) analysis is discussed. Finally, RE is introduced as a safety mechanism for CPS.

The related work concerning the convergence of formal methods and AIs is introduced. This covers the verification and validation of ANN, and the timing and functional safety of these ANNs.

## 2.1 Background

### 2.1.1 Artificial Neural Networks (ANNs)

ANNs were originally proposed to mimic the functioning of biological neural networks [27], which produce recurrent spatio-temporal patterns [56]. Similar timed activity of neurons in the cerebellum has been reported in [11, 44]. A number of types of Neural Network (NN) which mimic their biological counterparts exist, varying in complexity and accuracy, including the SNN [25, 38], which was designed to model the brain and has been demonstrated to be periodic

and run with discrete time intervals when implemented in software. ANNs are also part of deep learning; Convolutional Neural Networks (CNNs) [59] were introduced with many more layers than is possible on a MLP. CNNs, are widely used in CPS applications such as autonomous vehicles, e.g. in [10], where a CNN is trained to map raw pixel data directly to vehicle steering commands.

### Structure of an Artificial Neural Network (ANN)

Most ANNs do not feature such complex models like those of spiking neural networks, as they are more difficult to use, implement, and train. Instead, they rely on simpler networks, which can be considered as *un-timed non-linear* functions, where the outputs change relative to the inputs, but the timing of the change is not precisely defined. An example of such a network is provided in Figure 2.1, which is using neurons defined in Figure 2.2. This is a type of ANN known as an Multi-layer Perceptron (MLP) [76]. A MLP consists of interconnected layers of artificial neurons, each neuron providing input to neurons in the next layer [73].

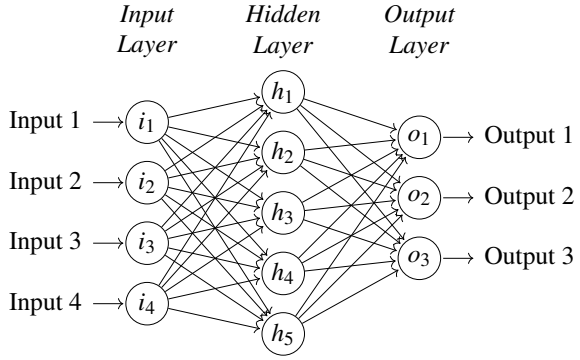



Figure 2.1: Example MLP ANN, showing the layers of artificial neuron and their connections.

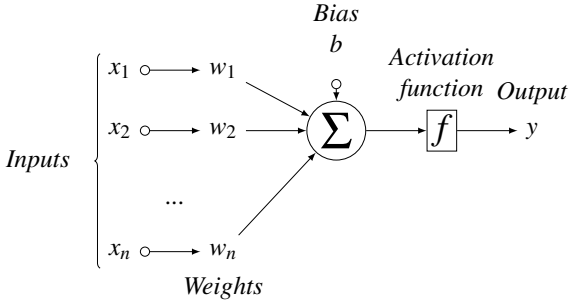



Figure 2.2: A model of an artificial neuron with the inputs, summation function and activation function.

Specialised neural networks, called Recurrent Neural Networks (RNNs) [40], were introduced to classify temporal sequences. These operate in a step by step manner, where the operation in the current time step relies on the context from some previous step.

CNNs were introduced to classify complex, 3-D inputs such as images [73]. These are similar to ANNs such that they are composed of multiple layers, each providing inputs to the following layer. However, CNNs have more complex neurons than MLPs, with varying types of layers such as convolutional layers, pooling layers and dense layers. CNNs, due to their deep structure, are able to learn larger and more complex data relationships than the previously introduced MLP. A figure of a CNN is provided in Figure 2.3.

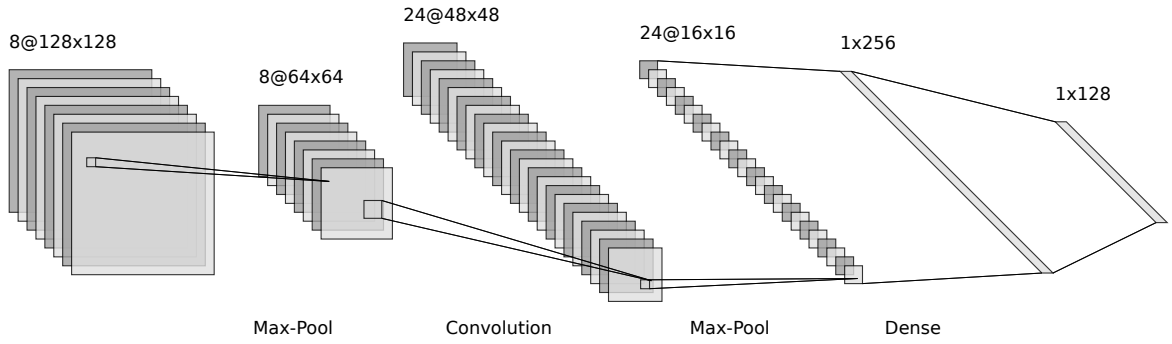



Figure 2.3: An example of a CNN, abstracting some of the different layer types and connections.

ANNs are being increasingly used as controllers in CPS due to their ability to learn data relationships in ways that are difficult to replicate [30]. ANNs can deal with novel inputs to the system and are able to outperform other forms of AI at computational efficiency, pattern recognition, function approximation and image identification [49, 47]. However, it can be very difficult to ensure the safety of a system involving ANNs [13, 30].

### Training of Artificial Neural Networks (ANNs)

ANNs need to be trained to learn data relationships, these cannot just be made up or drawn out by hand [45]. There exist many training techniques for ANNs, all which fall under one of three categories: supervised, reinforcement and unsupervised learning. All techniques used in this thesis fall under supervised and reinforcement learning. Under any training method, the ANN is trained over a training data set multiple times. For every time it trains over a complete data set, it is know to have trained for 1 epoch. ANNs are often trained for thousands of epochs to converge to the best solution.

Supervised learning refers to the training when the desired output set to every input set in the training data is known. The ANN can be trained to converge to the correct output sets for the

trained input sets. The most well-known, and used in this thesis, supervised learning technique is back-propagation with gradient descent [73]. This involves calculating the gradient of the error of an output set and propagating this error back through the ANN, all the way to the inputs. Over time, the ANN converges to the outputs it is trained on.

However, often the desired outputs are not known. In such systems, reinforcement learning is used to train the ANN [52]. Reinforcement learning works by giving the ANN a positive reward when it produces outputs that are good in its environment, and giving it negative rewards when it produces outputs that are bad. Thus, the ANN can learn from good and bad decisions and learn to produce the best outputs in a given situation.

#### Artificial Neural Network (ANN) ensembles

An ANN ensemble is the parallel execution of multiple ANNs working in combination to produce more accurate output [39]. An ensemble can contain different ANNs, with different structures, inputs, outputs and even programming languages. The output of an ANN ensemble represents some combination of all the ANNs in the ensemble. The output of an ANN ensemble is more accurate than any individual ANN in the ensemble. These are used to increase the prediction or classification of a system.

#### Adversarial Perturbation in Artificial Neural Networks (ANNs)

A research group showed that marginal modifications to the input image of a Convolutional Neural Network (CNN), such as the discolouration of a few pixels, could cause the image to be misclassified [21, 78]. A CNN can train to high accuracy, e.g. 99.99%, on the training set, but simple perturbations to the CNN's input can lead to drastically reduced accuracy. Figure 2.4 shows the effect of input perturbation on a ANN trained classifier. Misclassifications by a ANN controller in a CPS can have drastic consequences to the safety of the system.

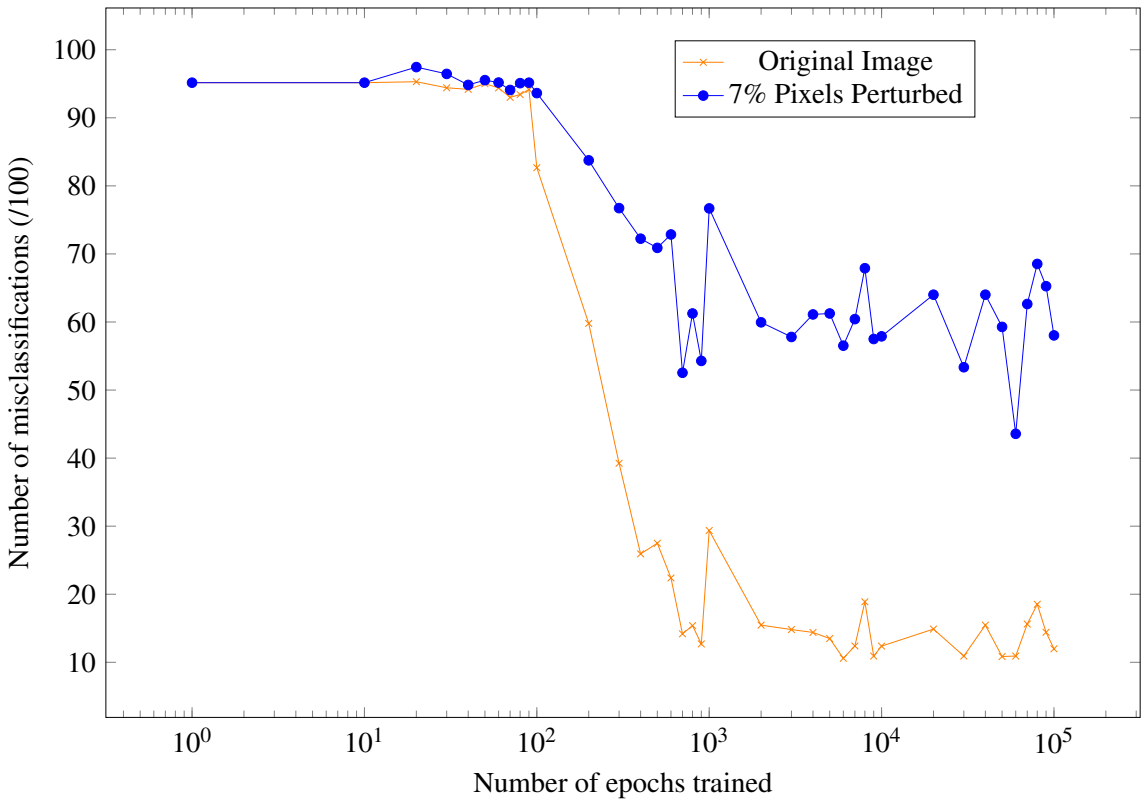



Figure 2.4: Line graph showing the effect of input perturbation on the prediction accuracy of a MNN

#### Artificial Neural Network (ANN) libraries used in this work

Primarily, the our own SNN library was used in this thesis. This library was able to implement and train any MLP ANN as a time-predictable SNN. This library was developed in C in conjunction with the synchronous language Esterel to produce the SNNs. However, this library was unable to handle CNNs, and we use the Darknet [53] C library to implement any CNNs. These CNNs were not time-predictable, but were able to run synchronously as SNNs using Esterel. The Python tool-chain Meta Neural Network to C (MNN2C) was introduced at a later stage. This tool-chain was able to take Keras [14] trained ANNs, compose a Meta Neural Network (MNN) from these ANNs and generate synchronous, time-predictable MNNs in C.

### 2.1.2 Synchronous Languages

Synchronous languages include a variety of different languages, some notable examples being Esterel, Lustre and Signal [6]. When it comes to validating and implementing real-time, embedded software, synchronous languages are the most design friendly and formally sound.

Synchronous languages abide to a set of semantics, that is that they support functional concurrency, have a simple formal model and support synchrony models. To support synchrony and concurrency, synchronous languages divide time into discrete instants according to a log-

ical clock. Each logical tick, the system samples the inputs, takes some action and then emits the outputs.

### Esterel

Esterel is one such synchronous language, created by Gérard Berry in 1991 [5]. Esterel uses a collection of concurrently running threads synchronised to a single, global clock. Pause statements are used to pause the thread execution, with each thread resuming from where it was paused at each clock tick. Communication between threads is done using globally broadcast signals. The communication between Esterel's threads is deterministic, i.e. a single signal cannot be read as both present and not present in any tick. Esterel can, additionally, use *pre()* (pre-emption) statements to check the past presence, or absence, of signals, allowing for logical delays in the execution of an action. Due to the structure of Esterel's loops, unbounded loops are not allowed; it must be guaranteed that a loop will be paused at least once per iteration.

```
1  module ANN:
2
3  procedure runANN()(integer, integer);
4  function getOutput(): integer;
5
6  input A: integer, B: integer;
7  output O: integer;
8
9  loop
10 [
11 [await A || await B]
12 call runANN()(?A, ?B);
13 pause;
14 emit O(getOutput());
15 ]
16 end
17
18 end module
19
```

Figure 2.5: An Esterel module to run a basic ANN using C function calls.

Figure 2.5 shows an ANN run example. This module has 2 defined input signals *A* and *B* and one defined output signal *O*. These signals are declared as *integer* signals, meaning they have an attached integer value in addition to the presence (or absence) of the signal. A *loop* denotes an iterative thread that repeats when the *end loop* statement is reached. Each loop must include a *pause*, making the loop bounded. The **"||"** are used to show parallel actions, i.e. concurrency. This means that *await A* and *await B* are running concurrently, where *await* <*signal*> is synonymous with "pause until <signal> is present".

External functions can be called in Esterel, included from a C file. A *procedure* refers to a C function that does not return any value, while *function* refers a C function that does. The *procedure runANN()(integer, integer)* takes two integers as input, and does not return any output. The *function getOutput()* takes no input, and but returns an integer as output. The *call* statement can be used to run a procedure, and just like in C the parameters are passed in brackets. However, since *A* and *B* are signals, the ***"?"*** is used to fetch the value at that signal, rather than the presence (or absence) of the signal. In Esterel a *function* returns some value, this value can be stored in a variable using the operator ***":="***, or it can be passed straight to the next function or, as with this example, be emitted by passing the parameter to the *emit* statement. The statement *emit O* sets the output signal *O* to be present for one tick, and sets its value according to its parameters.

When combined, the statements in this example would combine to do the following, described in English for convenience:

1. Wait for A and B.
2. When both A and B have been present, run the function *runANN* with the integer values of A and B as its parameters.
3. Pause execution until the next tick.
4. Emit O with the returned value of *getOutput* as its integer value.
5. Repeat from **1**.

### 2.1.3 Worst Case Execution Time (WCET) of Cyber-Physical Systems (CPS)

The timing analysis done in this thesis, notably Worst Case Execution Time (WCET) analysis, is done using the Patmos controller architecture by the T-CREST project [61] [62] [67]. Patmos is a time-predictable processor architecture on which time-predictable software can be run and, more importantly, analysed. Platin [22] is a Worst Case Execution Time (WCET) analysis tool used to calculate the WCET of software implemented on Patmos processors. The Platin tool is given the Patmos processor's parameters and an entry point in the C code that will be analysed. This entry point can be any reachable C function from the *main* function of the given project. The Platin tool then compiles the C function to an intermediary file, upon which the tool uses to calculate the WCET. The resultant WCET is a number of cycles on the given Patmos processor type.

However, this tool does introduce some rules to be followed for the C software to be time-predictable. These are listed as follows:

- All loops must be bounded. Every loop contained or accessed by the entry function must have a minimum and maximum number of cycles within which it will finish executing. Any *for* loops are analysed by the Platin tool for the minimum and maximum, however *while* loops must be given these parameters using a *pragma* in the following format: *pragma loopbound min* <*minimum*> *max* <*maximum*>. These *pragmas* may also be assigned to *for* loops, but this is not necessary.

- Dynamic memory allocation must be avoided. The use of *malloc* and *calloc* statements must be avoided.

- No C *string* manipulation can be handled by Platin. This includes, but is not limited to *printf* and *strcpy* functions.

- While mathematical floating point operations can be handled by Platin, it does so very badly; i.e. a unnecessarily high WCET is generated. As a result, floating point operations should be avoided and integer, or fixed-point, operations should be used instead.

All code generated for this thesis (except for CNNs generated using the Darknet library) adhere to this list of time-predictable rules. Thus, all code generated for this thesis (bar Darknet CNNs) is time-predictable when implemented on a Patmos processor.

### 2.1.4 Run-time Enforcement (RE) of Autonomous Systems

While static verification has its place, more dynamic approaches to verification and safety can be used to pick up anything that the static verification and testing may have missed. Take a standard image classification CNN for example. The CNN can be trained to 99.9% accuracy according to the test cases used to train it. However, this means that the CNN fails 0.1% of the time, and that is only on the tested population, not even taking the entire population into consideration. Having a method to verify this CNN while it is running and pick up any inevitable failures would allow for this CNN to be used in systems where safety is critical.

#### Run-time Verification (RV)

Run-time verification is an extension of run-time monitoring [50]. A run-time verifier monitors the I/O events of a system using a specified safety policy. The verifier provides positive or negative feedback depending on the I/O of the system, providing a verdict for the current state of the automaton. The run-time verifier has no knowledge of the inner workings of the system, regarding it as a black box. This makes it ideal for autonomous systems, where the inner workings are often too complex to be verified. Figure 2.6 shows the basic structure of a run-time verifier.

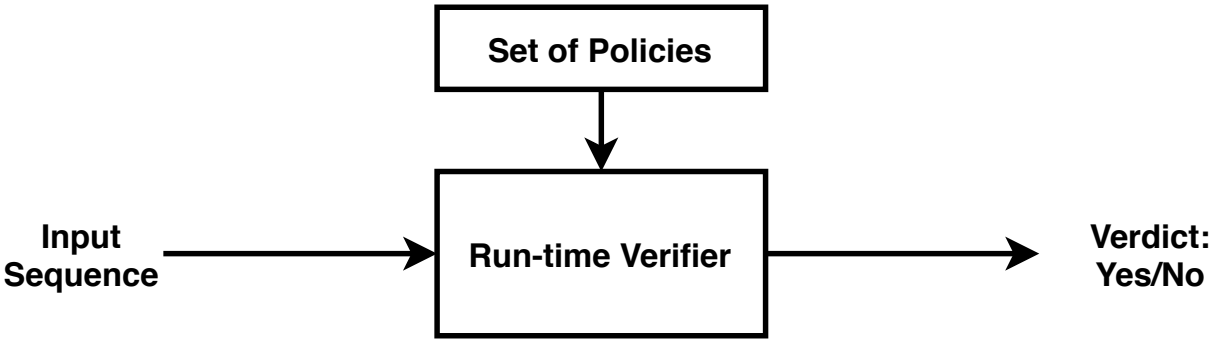



Figure 2.6: Basic view of a run-time verifier.

**Run-time Enforcement (RE)**

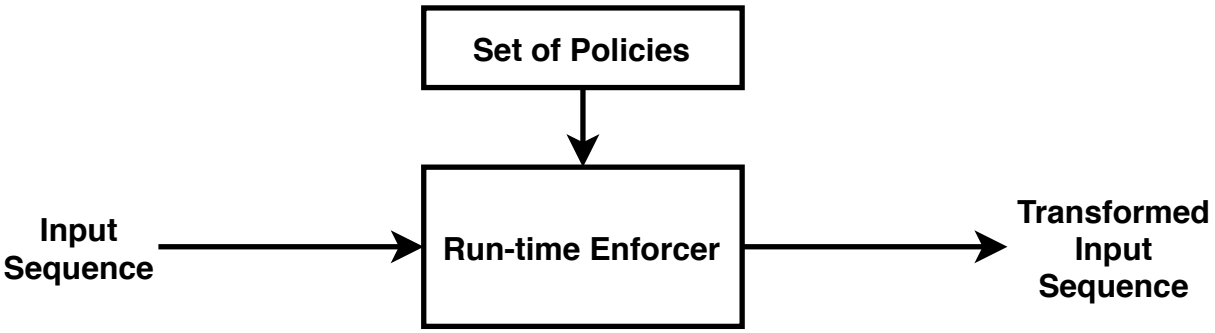



Figure 2.7: Basic view of a run-time enforcer.

Run-time Enforcement (RE) is a subset of Run-time Assurance (RA) that focuses on formal semantics and blocking, delaying, modifying and/or re-ordering of events in a system. RE can be transformation or reactive. Transformational RE uses the delaying, buffering and reordering of event to enforce a safety policy, while reactive RE uses edit functions to edit events and can be bi-directional. This thesis focuses on reactive RE, since ANNs are reactive in nature. Processes that are deemed unsafe can be monitored by an enforcer at runtime to ensure that they obey desired policies and remain in a safe state at all times [35]. Formal runtime verification methodologies mathematically guarantee the detection of improper system behaviour [15]. For example, Security Automata (SA) have been proposed, which formally monitor uni-directional run-time properties only (e.g. outputs only) [60]. Edit automata are a type of SA that can edit, suppress or insert events [34]. DTA have been proposed that can edit *bi-directional* events at runtime [51]. They were designed for reactive CPS demonstrated in a pacemaker environment [51]. Figure 2.7 shows the structure of a run-time enforcer.

**Discrete Timed Automata (DTA) for Run-time Enforcement (RE)**

In order to specify the safety policies to be enforced by the run-time enforcer Pinisetty et al define Discrete Timed Automata (DTA) [51]. These DTA can be used to represent safety policies

φ which can be enforced at run-time using *edit functions* [51].

Run-time enforcers are able to enforce binary I/O events at run-time. I.e. the enforcers can enforce the absence or presence of an input or output signal at run-time. A unique property of DTA is their ability to express timed properties which can be enforced by a run-time enforcer. A DTA can have one or more timers, which increment at each discrete time instance. Guards and transitions can be enforced that check the timer. An example DTA is given in Example 2.1.1 to introduce the syntax used in the DTAs.

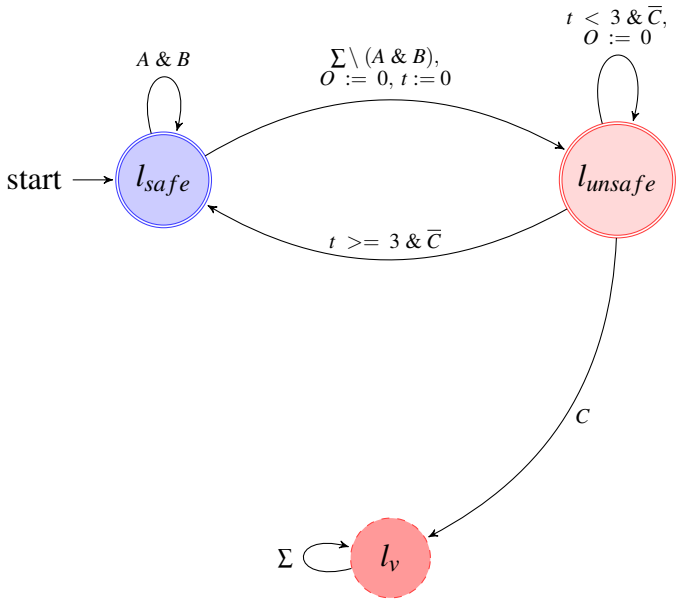



Figure 2.8: DTA example showing the syntax used to describe DTAs in this thesis

**Example 2.1.1** (Syntax of a DTA)**.** *The DTA shown in Figure 2.8 represents a basic safety policy. The policy as inputs A, B and C and output O and starts in the safe state $l_{safe}$. The policy's safe state transitions are described in layman's terms below:*
*While A and B, denoted as A & B and not to be confused with A OR B (A | B), are present simultaneously, transition back to the safe state.*
*However, if A and B are not detected simultaneously, transition to the unsafe state $l_{unsafe}$, suppress the output signal O ($O := 0$) and set the timer to 0 ($t := 0$).*
*The statement "$\Sigma \setminus Z$" reads as "anything except Z". The enforcer actions are indicated by statements using the syntax "$X := Y$", which reads "set X to Y".*

*The unsafe state has the following transitions:*
*If the timer is less than 3 and C is not been present, transition back here while suppressing output 0.*
*If the timer is greater than, or equal to, 3 and C is not been present, transition to the safe state.*
*If C is detected, transition to the violation state.*
*The absence of a signal is detected using the* ***overline****, e.g. "not X" would be $\overline{X}$.*

*The violation state has only a single transition that returns to the violation state on anything.*

*A brief description to the DTA would be to say the A and B must be present together. If they are not, the system is unsafe for 3 ticks, within which if C is present a violation occurs. After 3 ticks, the system returns to a safe state. While the system is in the unsafe state, the output O is suppressed by the enforcer.*

## 2.2 Related Work

### 2.2.1 Artificial Neural Networks (ANNs) for Cyber-Physical Systems (CPS)

In order for an ANN to be used in any capacity within a system where safety is critical, it should undergo rigorous and thorough validation, verification, and testing procedures to ensure that they it is sufficiently safe for its target system [28, 31].

While considerable research effort is starting in the direction of formal verification of AI-based CPS [63, 58], the issue of timing verification has received scant attention. Like the challenges involving functional verification, timing verification of AI-based CPS poses considerable challenges due to the fact that: (1) real-time AI systems could involve many concurrent and interacting AI modules, which need deterministic composition for safety; (2) AI modules are usually developed as untimed systems and the reactive nature of AI algorithms used in CPS are not carefully studied; and (3) Worst Case Execution Time (WCET) analysis [74] of AI-based CPS has received scant attention.

Definitions for this safety vary, but Kurd et. al. [29] provide a generalisation: safe ANNs can be defined as those that:

- tolerate faults and inconsistencies in their inputs,
- do not create hazardous outputs,
- behave in a predictable and repeatable manner,
- and are trained on clean, reliable data.

To achieve these properties, there exist safety measures such as risk management systems that span the entire development process of the ANN [55] and standards with which ANNs can be certified before they are used in systems where safety is critical [4]. These techniques are primarily *proactive* in nature, producing ANNs that are classified as *safe enough* for their role.

However, as ANNs become larger and more full-featured, they become harder to statically analyse. Problematic situations can arise when an ANN exhibits unexpected behaviour that the system is unable to safely respond to, and in CPS these situations can be life threatening.

### 2.2.2 Functional Verification of Artificial Neural Networks (ANNs)

Typical approaches for ensuring that CPSs are safe involve processes to demonstrate that an acceptable level of risk has been achieved [28]. Designers of AI software rely on several validation and verification technique, including, but not limited to, conventional testing, run-time monitoring, static analysis, model checking and theorem proving [41]. Unfortunately, due to the complexity of Artificial Neural Networks (ANNs), techniques such as static analysis, model checking and theorem proving are less valuable in ANN environments. Conventional testing is a common method to test the accuracy of ANNs, but this method is not fool-proof and its efficacy relies heavily on the creator of the test cases. Run-time monitoring is a technique that could greatly benefit the safety of running ANNs, but there has been minimal research done in this field.

There are a variety of pre-existing methods for statically checking the correctness of autonomous (i.e. artificially intelligent) systems. For instance, model checking on systems that use timed automata [46]. Okano et. al. explore the concept of model checking of autonomous systems that use timed automata [46]. Conventional model checking done on this automaton allowed the safety and robustness of the system to be demonstrated. While techniques such as model checking work very well on non-ANN AI, ANNs are not usually able to be simplified to simple automata. This technique is a static technique and cannot be applied to autonomous ANN systems, where the behaviour of the ANN controller cannot be defined by an automaton. For an autonomous system that includes at least one ANN in its controller, novel techniques are required to guarantee the safety properties of the ANNs. However, ANNs are not usually able to be simplified to simple automata.

Deep learning is a widely and extensively researched field with regards to modern machine learning that refers to the learning of data representations, rather than learning task specific algorithms [59]. Deep learning has applicability in a lot of current ANN implementations, such as the autonomous vehicles used by Tesla and Uber. Verification of ANN, specifically, Deep Neural Networks, can be performed for certain properties (such as robustness) using Satisfiability Modulo Theories (SMT) [21, 23, 26]. This is useful, because the robustness of a deep ANN is a critical property of its safety. A robust ANN is one that will provide consistently accurate outputs even when the input to the ANN is noisy, incorrectly coloured or otherwise distorted. However, this approach is not flawless. SMT has issues with scale: as the ANNs to analyse become larger, analysis time grows exponentially [21]. Ergo, they are less efficient on larger, more complex ANNs. In addition, they require the ANNs to fulfil some specific properties, such as specific activation functions and specific ANN variants, i.e. [21] only allows CNNs and MLPs with Recitifed Linear Units (ReLU) activation functions. This limits flexibility, as each ANN must be designed around these restrictions, thus limiting properties of the ANN, such as its activation function and size, could result in an ANN that is inefficient, not robust, slow, etc.

Due to these difficulties, it can be tempting for designers to simply rely on manual testing to

check for the correctness of ANN-based systems. However, this is a time-consuming and error-prone process which cannot provide good guarantees, as it is very difficult to ensure that tests have acceptable coverage of all possible situations [9]. Furthermore, as with the static analysis approaches, as the ANNs increase in size and complexity, verification and validation of these networks becomes increasingly more difficult to achieve [21]; test data is not unlimited, time is a resource and verification is not 100% accurate.

Finally, no matter the chosen methodology, as ANNs increase in size and complexity, verification and validation of these networks becomes increasingly more difficult and resource intensive to achieve [21].

#### Run-time Enforcement (RE) of Artificial Neural Networks (ANNs)

The idea of RE of autonomous systems has received some research attention. De Niz et. al. propose a type of RE they term temporal enforcement, which ensures that the system controller meets timing deadlines where outputs are concerned [17]. While this shares similarities with the work in this paper, their work does not expand to cover ANNs, and does not propose the use of RE for anything other than meeting timing deadlines. Aniculaesei et. al. propose static formal verification and runtime monitoring of autonomous, robotic systems to prevent physical collisions during system execution [3]. While this looks at the enforcement of system outputs, the inputs are not monitored and the timing deadlines of the system are not investigated. Additionally, the case study involves a robot controlled by an automaton, not a highly complex AI such as an ANN.

# 3
# Synchronous Neural Networks

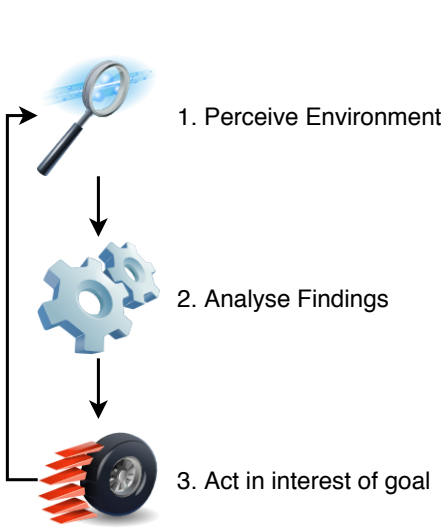
## 3.1 Introduction

CPSs use a set of controllers that are distributed across a network for the control of physical processes [2]. CPS applications encompass real-time systems, where the system needs to satisfy a set of timing requirements to ensure correct operation. Examples include autonomous vehicles and smart power grids. Here, a missed deadline may result in catastrophic consequences, making these CPSs highly *safety-critical*.

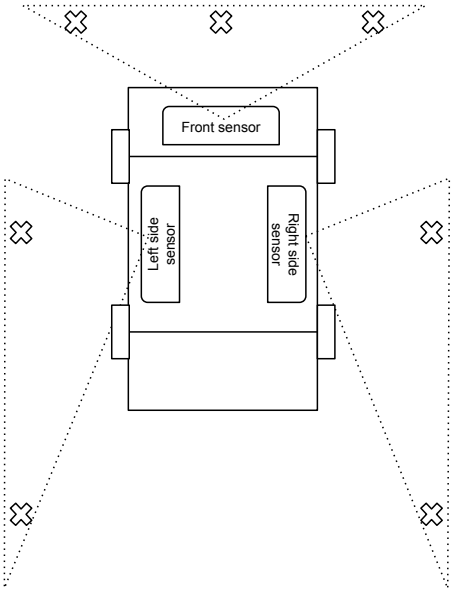
## 3.2 Synchronous Neural Networks

Consider an autonomous vehicle, simplified for pedagogy, as shown in Figure 3.1b as our motivating example. The vehicle uses various sensors to “see” the environment around it, while making *lane changing decisions*. The sensor outputs are interpreted in order to make decisions based on the dynamic environment. The vehicle uses a number of sensors to detect objects in the environment around it. Based on these, it will decide when it needs to change lane to the *left* or *right*, to *continue* in the current lane, and when to *stop* if necessary. The sensors consist of one frontal sensor and a sensor on each side. The frontal sensor is simplified to detect three positions in front of the vehicle. The side sensors work similarly to the frontal sensor, and each "see" two positions on each side of the vehicle.

ANNs used in CPS applications, such as lane changing, are used for decision making tasks




(a) Reactive AI




(b) Autonomous Vehicle Sensors

Figure 3.1: Reactive systems (left) and the sensor system for `AI-BRO` (right)

needed by the controller. Such tasks operate periodically, being reactive in nature, as shown in Figure 3.1a. We start by formalising SNNs using Definition 3.2.1, which are introduced by us.

**Definition 3.2.1.** A Synchronous Neural Network (SNN) operates periodically, where the length of each period is an integral multiple of the *tick length* of a "logical clock" that *tick*s. Tick length is a constant $\in \mathbb{R}^{>0}$.

The periodic behaviour SNNs, as defined in Definition 3.2.1, are ideally implemented using the synchronous paradigm [6] and we will use the Esterel programming language [7] to motivate the design of SNNs. Synchronous programs execute tick-by-tick and during a tick (also known as reaction) the environment inputs are sampled, the overall system reaction is performed and finally emission of outputs are made. Thus, a tick encapsulates the atomicity of reactions, where inputs can happen only at the start of a tick, preventing race conditions arising due to environment non-determinism, which is widely prevalent in event triggered paradigms. A synchronous reaction is often represented as a composition of multiple, interacting *threads* exhibiting *logical concurrency* [6]. These are usually *compiled-away* to produce sequential code that avoid conventional race conditions widely prevalent in the asynchronous setting. Distributed compilation of threads, which is a much harder problem, over multicore platforms is also well developed, though less widely used [77]. The synchronous approach guarantees key properties such as *determinism* and *reactivity*, which is ideal for AI-based CPS. This motivates our adoption of the this paradigm while designing SNNs.

We have adapted Berry's well known ABRO example in Esterel [7] to develop our first example of SNNs. Being an adaptation of ABRO, we term our example, the AI-inspired ABRO,

```
1  module ai-bro:
2
3  % data handling declarations -- omitted
4
5  % interface declarations
6  input start, A, B, R;
7  input fr1: integer, fr2: integer, fr3: integer;
8  input s1: integer, s2: integer;
9  input s3: integer, s4: integer;
10 output O: integer;
11
12 loop
13 [
14 await A;
15 % run ANN A: interpret frontal scanner data
16 present fr1 or fr2 or fr3 then
17 call processFrontSensors()(?fr1, ?fr2, ?fr3);
18 end present
19 ||
20 await B;
21 % run ANN B: interpret side scanner data
22 present s1 or s2 or s3 or s4 then
23 call processSideSensors()(?s1, ?s2, ?s3, ?s4);
24 end present
25 ];
26
27 % run ANN C: decide on best course of action
28 call makeDecision()();
29 % fetch output action
30 emit O(getAction());
31 each R
32
33 end module
34
```

Listing 3.1: Esterel implementation of AI-BRO

as AI-BRO, and present it in Listing 3.1. The original ABRO example performs *the emission of O when both inputs A and B have happened. It resets and restarts this behaviour when the input R happens*. In our setting, the input A is used to control the processing of the frontal sensor, and the input B is used for the side sensors. When both inputs have happened, we start the processing of the decision making network, which outputs the lane changing decision using the output O. R resets and restarts this behaviour.

The AI-BRO implementation in Esterel is done using an Esterel module called ai-bro shown in Listing 3.1 on line #1. A module is a basic programming unit in Esterel and the interface of the module consists of input and output *signals*. We have defined input signals A, B, R, which are *pure signals* i.e. these are Boolean in nature carrying a status — either *present / true* or *absent / false* during a tick. The output signal O is valued and in addition to its status takes a value of type integer. The interface definitions appear in lines #6-10. A variable,

unlike a signal, has no status information.

The main logic of the program is composed of two concurrent threads (denoted using the `||` operator) that await external events `A, B` to happen. This task can be performed concurrently as these two inputs may happen either simultaneously in the same tick or in an interleaved manner in different ticks. Upon occurrence of the respective event, corresponding data processing is performed by the invocation of the appropriate ANN, implemented as the C function `processFrontSensors()` and `processSideSensors()`. Both these functions encapsulate pre-trained neural networks. Thus, when both events have happened and both networks have executed, the two concurrent threads are terminated. The two threads are also grouped using the grouping operators (denoted by `[ ]`), which are put in sequence using the sequence operator (denoted by the "`;`"). Thus the third neural network, called `makeDecision()`, can trigger only after the data from both the front and side sensors have been processed. The output of this decision is subsequently emitted through the valued signal `O`, on line #30, which will indicate the chosen lane changing action.

Finally, the main logic is enclosed in a `loop..each R` construct. This is a *preemption construct* that preempts and restarts the loop on every `R` input, maintaining reactive behaviour with every reset. This will happen after each lane changing decision.

The `AI-BRO` example illustrates several features of SNNs. (1) We are able to create ANNs that trigger periodically (analogous to RNNs) and we can control the timing of these invocations, and (2) within each period, we can precisely synchronise the execution of different ANNs i.e. the sensor processing ANNs are synchronised with the external inputs `A, B` while the decision making ANN is executed immediately after the first two ANNs have finished. To the best of our knowledge, such precise synchronisation mechanisms are lacking in existing literature on ANNs. More importantly, a side effect of the Esterel implementation is that the resultant ANN compositions are either rejected by the compiler as *non-causal* or compiled into a *causal* implementation that is guaranteed to be *deterministic* and *reactive* [6].

**Proposition 1.** *The AI-ABRO implementation in Listing 3.1 is an instance of SNN defined using Definition 3.2.1.*

*Proof.* The proof of this trivially follows from the fact that this program is compiled to a single `tick()` function by the Esterel compiler, where the concurrency is compiled away. Thus, during any tick, we can have a combination of neural networks executing that range from just one network to up to three networks. Assuming that we can execute this program with a fixed tick length (elaborated in section 3.3), the overall period of any network is at most 1-tick. □

## 3.3 Timing Analysis of Synchronous Neural Networks (SNNs)

Time in a synchronous program progresses in logical discrete instants called *ticks*. In each tick, the environment is first sampled, then computation takes places, and finally the results are emitted. In order for safety-critical CPS, such as our `AI-BRO`, to be verified, we need to ensure that the time taken for the execution of any given tick is less than their environmental deadlines. This necessitates the computation of the worst-case tick length via static timing analysis — also known as Worst Case Reaction Time (WCRT) analysis [57].

WCRT analysis needs two things: (1) an architectural model of the underlying processor architecture; and (2) a Control Flow Graph (CFG) [74] representation of binary program execution over the architectural model. The analysis process then computes the longest path through the CFG. While general purpose programs can feature complications which make this process difficult, Esterel is ideally suited to derivation of CFGs due to inherent features: (1) all loops in Esterel are bounded, as there is a need for at least one `pause` statement within every loop; and (2) compilers "compile-away" the logical concurrency to produce sequential code. However, as can be seen in Listing 3.1, Esterel programs can also invoke arbitrary C-functions, which may use features such as function pointers, unbounded loops, and external interrupts. To this end, we provide a set of guidelines and ANN implementation libraries in C, which restrict these features to make each C function call time analysable.

In this paper, we are using the open-source Platin tool [22] for WCRT analysis. Consequently, we have to consider some tool-specific restrictions to avoid the use of generic features in C that are not suitable for static analysis, such as system calls, function pointers, interrupt-driven flow, or unbounded loops [37]. Hence, the restrictions considered in this paper are: (1) all neural network functions are coded as bounded `for` loops; (2) all memory is statically allocated; (3) variable-length array iterator functions are coded using preprocessor macros; and (4) the final binary is executed 'bare-metal', requiring no RTOS or OS support, thus avoiding scheduling dependencies that may be difficult to statically analyse.

Annotating a CFG with execution times is no simple task for many architectures. Enhancements such as out-of-order execution, multi-level caches, and deep pipelines can all cause complications when considering the time taken to execute program code. For instance, in [20], Ferdinand et. al. demonstrate the difficulties of determining the worst-case time in arguably simple avionics code running on a Motorola ColdFire 5307. To avoid these complications, we rely on the Patmos [62] architecture, which is a simple RISC pipeline optimised for static timing analysis. Here, caches have simple replacement policies, and it also has a time-predictable memory hierarchy. It relies on an adaptation of the LLVM compiler that targets the Patmos instruction set. This simplifies the annotation of the nodes of the CFG with timing costs, thus allowing for a straightforward static analysis of any SNN. Consequently, this work is restricted to the single-core Patmos architecture and does not examine execution of SNNs on other plat-

form types, such as Intel processors and GPUs.

### 3.3.1 WCRT algebra

Though Esterel is amenable to timing analysis, it still suffers from state space explosion when considering large programs. Analysing large sequential code generated from Esterel compilers that ignore *infeasible state space* produce large overestimates. WCRT algebra, presented in [70], presents a solution to this problem: large synchronous programs can be expressed as networks of simpler Tick Cost Automata (TCA), which have equivalent timing properties to their Timed Control Flow Graph (TCFG) counterparts. Considering this, we seek to use this framework to develop a compositional timing semantics of SNNs and their compositions.

We consider a discrete max-plus structure over natural numbers $(\mathbb{N}_\infty, \oplus, \odot, \mathbb{0}, \mathbb{1})$ where:

- $\mathbb{N}_\infty =_{df} \mathbb{N} \cup \{-\infty, +\infty\}$
- $\oplus$ and $\odot$ are commutative and associate operators that denote the maximum and addition respectively on $\mathbb{N}_\infty$.
- Neutral elements are $\mathbb{0} =_{df} -\infty$ and $\mathbb{1} =_{df} 0$, respectively, i.e., $x \oplus \mathbb{0} = x$ and $x \odot \mathbb{1} = x$.

In particular, $-\infty \odot +\infty = -\infty$. Addition $\odot$ distributes over max $\oplus$, i.e., $x \odot (y \oplus z) = x + max(y,z) = max(x+y, x+z) = (x \odot y) \oplus (x \odot z)$. However, $\oplus$ does not distribute over $\odot$, for instance, $9 \oplus (5 \odot 7) = max(9, 5+7) = 12$ while $(9 \oplus 5) \odot (9 \oplus 7) = max(9,5) + max(9,7) = 18$. This induces on $\mathbb{N}_\infty$ a (commutative, idempotent) semi-ring. We denote $\mathcal{W}$ as the set of formulae in this algebra.

We use an automaton called TCA to capture the timing behaviour of any SNN. These are mono-cyclic automata, where every state of the automaton is labelled by a WCRT formula that associates the state with its worst case timing cost. We start by defining TCA using Definition 3.3.1

**Definition 3.3.1.** A *mono-cyclic tick cost automaton*, here after referred to as tick cost automaton (TCA) is $M = \langle Q, q_0, \rightarrow, \mathcal{W}, L \rangle$, where $Q$ is a finite set of *states* with a distinguished *entry* state $q_0 \in Q$ and $L : Q \rightarrow \mathcal{W}$ is a labeling function that labels every state with its associated timing cost by a formula in $\mathcal{W}$. The *transition relation* $\rightarrow \subseteq Q \times Q$ satisfying the following:

1. From any state $q_i \in Q$ there is at most one transition to a successor state $q_j \in Q$.
2. The transitions are such that they form a mono-cycle that starts in $q_0$ and and ends in a transition from some state $q_{n-1} \rightarrow q_0$, resulting in a cycle of length $n \in \mathbb{N}_{>0}$.
3. A TCA whose cycle length is 1 is termed *mono-periodic*, while those with cycle length $> 1$ are called *multi-periodic*.

4. The *reaction time* of a TCA is its tick-length, which is always fixed to its WCRT.

5. The *response time* of a TCA is the time taken for the corresponding SNN to produce the outputs given inputs. The response time of a TCA is given by the formula $n \times WCRT$, where $n$ is the cycle length.

## 3.4 Multi-Periodic SNNs in Esterel

The mono-periodic implementation of SNN using the methodology in Listing 3.1, while being time predictable, has limitations. This is because each NN must run to completion before any reaction can complete. For large networks with many hidden layers, this is not ideal — as the overall WCRT of the system could become quite large, causing violations when considering response to some critical inputs which may need immediate attention to ensure safety. We highlight this aspect using an Energy Storage System (ESS) case study in Section 3.6.

There are other approaches for arranging the internals of each network while still meeting Definition 3.2.1, and to illustrate this, we will use a simple XOR (exclusive or) MLP. This network is depicted in Figure 3.2, and as can be seen, it features five neurons in three layers. While we use this simple illustration, the methodology is applicable for any NN.

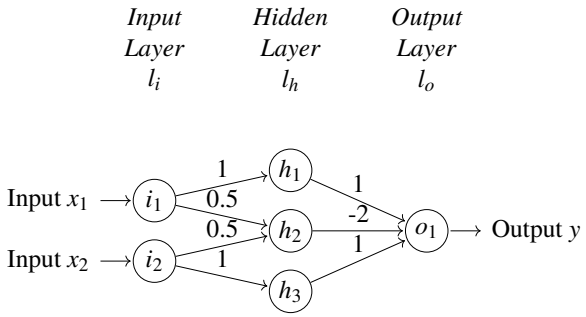



Figure 3.2: Example of an XOR ANN showing its layers and weights.

Each neuron in this network is based on the model in Figure 2.2, with a simple Boolean step activation function and zero bias. The activation function is represented by Equation 3.1, where $y$ is the output, $x \in X$ are the inputs, and $w \in W$ the weights.

$$y = \begin{cases} 1 & \Sigma_i (x_i \times w_i) >= 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

**Definition 3.4.1.** An MLP is formalised as a tuple $M = \langle I, O, N, L, \alpha, \eta \rangle$, where:

- $I$ is a finite collection of input variables with its domain being $\mathbf{I} = \mathbb{R}^n$
- $O$ is a finite collection of output variables with its domain being $\mathbf{O} = \mathbb{R}^m$

- $N$ denotes a set of neurons
- $L$ denotes a set of layers
- $\alpha : N \rightarrow L$ is the neuron mapping function that maps a given neuron to a layer
- $\eta : \mathbf{I} \rightarrow \mathbf{O}$ is the non-linear function (termed the network function) that provides the behaviour of a given network i.e. when provided a vector of input of size $n$ produces an vector of output of size $m$. Internally,

**Example 3.4.2.** *The MLP presented in Figure 3.2 can be presented formally as* $I = \langle x_1, x_2 \rangle$, $O = \langle y \rangle$, $N = \{i_1, i_2, h_1, h_2, h_3, o_1\}$ *and* $L = \{l_i, l_h, l_o\}$. *The function* $\alpha$ *maps neurons to layers e.g.* $\alpha(h_1) = l_1$ *and the network function* $\eta$ *given input* $\langle 0, 1 \rangle$ *produces an output* 1 *i.e.* $\eta(\langle 0, 1 \rangle) = 1$.

Using Definition 3.4.1, we can implement the XOR ANN as a "black-box" function in Esterel, as presented in Figure 3.3. The corresponding mono-periodic TCA is shown beside the code. The WCRT is the time taken to execute the network function $\eta$, i.e. $WCRT(M) = WCRT(\eta)$. In the following, we use $WCRT(\eta)$ and $\eta$ interchangeably.

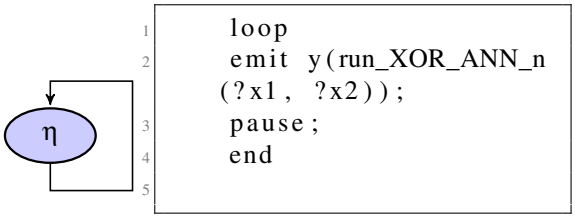


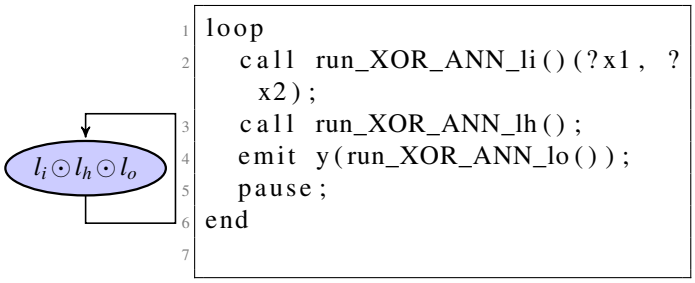

Figure 3.3: Mono-periodic 'black-box' execution, $WCRT = \eta$

It can be noted that by Proposition 1, any ANN implemented using the approach in Figure 3.3 is also an instance of SNN defined using Definition 3.2.1.

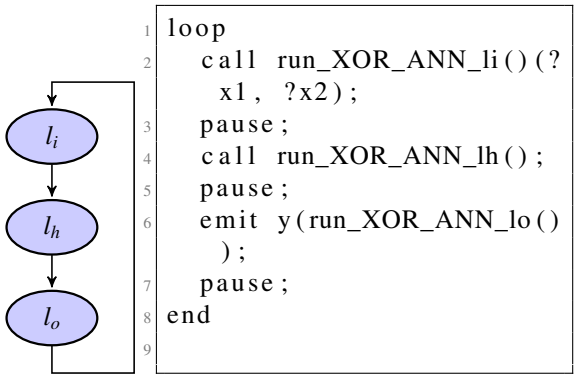
**Theorem 3.4.3.** *(Soundness) A mono-periodic SNN given an input vector* **i** *produces an output vector* **o** *in a given tick iff the corresponding MLP given the same input vector* **i** *produces the identical output vector* **o** *during any execution.*

*Proof.* The proof of Theorem 3.4.3 trivially follows from the fact that synchronous execution of a black-box function does not change the mathematical properties of that function. □

As mentioned, using the approach in Figure 3.3 to implement complex ANNs i.e. CNNs may lead to implementations that violate their timing requirements. The synchronous approach, however, enables efficient implementations via compositional modelling. Esterel supports this via statements such as `pause` that help to create smaller ticks, `;` for indicating sequence operation, and `||`, which enables the modelling of concurrency.




(a) Mono-periodic 'layer by layer' execution, $WCRT = l_i \odot l_h \odot l_o$




(b) Multi-periodic 'layer by layer' execution, $WCRT = l_i \oplus l_h \oplus l_o$

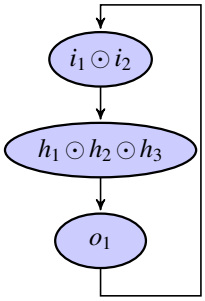
For example, the MLP in Figure 3.3 can also be compositionally modelled by splitting the neural function $\eta$ into its constituent layers $l_i$, $l_h$, and $l_o$. While this maintains the overall functionality of the network, it reduces the WCRT by making the network multi-periodic, where each period is shorter. As previously mentioned, the ESS case study in Section 3.6 examines the benefits of this approach.

**Proposition 2.** *The mono-periodic implementation of the XOR MLP is functionally equivalent to the multi-periodic implementation.*

*Proof.* For any input, output vectors $\mathbf{i}, \mathbf{o}$ respectively, $\eta(\mathbf{i}) = l_o(l_h(l_i(\mathbf{i}))) = \mathbf{o}$ □

Observe that the response time of the multi-periodic implementation is $3 \times T_2$, where $T_2$ is the WCRT of the multi-periodic version. In contrast, the response time as well as the reaction time of the mono-periodic version is always its WCRT $T_1$. However, it is expected that $T_2 << T_1$.

Like the breaking up of the function $\eta$ into its component layers, we can also break each layer $l$ into its component neurons. This option is presented in Figure 3.5a. Unlike in Fig-




```
loop
  [
    call run_XOR_ANN_i1()(?x1);
    ||
    call run_XOR_ANN_i2()(?x2);
  ];
  pause;
  [
    call run_XOR_ANN_h1()();
    ||
    call run_XOR_ANN_h2()();
    ||
    call run_XOR_ANN_h3()();
  ];
  pause;
    emit y(run_XOR_ANN_o1());
  pause;
end
```

(a) Multi-periodic, neuron-by-neuron execution, $WCRT = (i_1 \odot i_2) \oplus (h_1 \odot h_2 \odot h_3) \oplus o_1$

Figure 3.5: Different arrangements for XOR MLP

ure 3.4a, neurons are able to operate concurrently within each layer. This could provide benefits when considering multi-core implementations such as in [77].

A summary of these approaches is presented in Table 3.1.

Table 3.1: Summary of SNN approaches and their relevant figures, WCRT, cycles and response time

| Arrangement | Figures | WCRT | Cycles | Response Time |
|---|---|---|---|---|
| Mono-periodic | 3.3,3.4a | High | 1 | == WCRT |
| Multi-periodic | 3.4b,3.5a | Low | >1 | >>WCRT |

## 3.5 Meta Neural Networks

Complex CPS use several interacting NNs to achieve their functionality. However, to the best of our knowledge, formal modelling and timing analysis of such concurrent NNs is lacking. To this end we propose several alternative architectures, facilitated by the synchronous approach for formalising the composition and timing of several interacting NNs. We term these as *meta neural networks*. We start by formalising the input-output compatibility requirements.

**Definition 3.5.1.** Let $\mathcal{M}$ be a set of SNNs according to Definition 3.4.1. We term $M_1 = \langle I_1, O_1, N_1, L_1, \alpha_1, \eta_1 \rangle, M_2 = \langle I_2, O_2, N_2, L_2, \alpha_2, \eta_2 \rangle \in \mathcal{M}$ input-output compatible, IO-compatible for short, when the following hold:

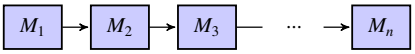
- *Size and type compatibility*: $|I_2| = |O_1|$ and the type of every connection also matches.
- *Timing compatibility*: The outputs of $M_1$ are produced in a tick on or before $M_2$ executes.

Hence, just as we are able to break execution of ANNs over multiple ticks by calling them "layer by layer", we can also structure the synchronous program such that it runs a set of concurrent SNNs "network by network". We term such an arrangement of multiple SNNs as *meta-layers* of SNNs.

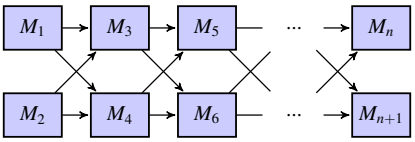
**Definition 3.5.2.** A network of SNNs organised into meta-layers is a tuple $H = \langle I, O, \mathcal{M}, \mathcal{L}, \mathcal{A}, \Delta \rangle$ where:

- We term the first meta-layer as the input meta-layer and the last one as the output meta-layer.
- $I = I_1 \cup .. \cup I_k$, where $I_1..I_k$ are the inputs of the SNNs in the input meta layer. If $K = |I|$ then the domain is $\mathbf{I} = \mathbb{R}^K$
- $O = O_1 \cup .. \cup O_j$, where $O_1..O_j$ are the outputs of the SNNs in the output meta layer. If $J = |O|$ then the domain is $\mathbf{O} = \mathbb{R}^J$
- $\mathcal{M}$ is the set $M \in \mathcal{M}$ of SNNs according to Definition 3.4.1 such any two connected SNNs are IO-compatible.
- $\mathcal{L}$ is the set of meta-layers of SNNs
- $\mathcal{A} : \mathcal{M} \to \mathcal{L}$ is the network mapping function that maps a given SNN to a layer.
- $\Delta \subset \mathcal{M} \times \mathcal{M}$ provides the mapping of IO connectivity information between SNNs of different meta-layers.

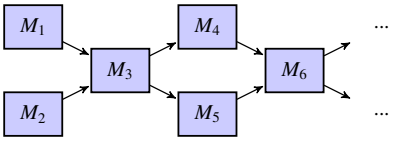
Depending on the application several options may be explored when considering the design space for the concurrent arrangement of multiple networks. Some of these are presented in Figure 3.6. The first approach, in Figure 3.6a, creates a pipeline of networks, where they are simply executed in sequence. Alternatively, SNNs could be arranged in parallel, as in Figure 3.6b. Finally, an arrangement might be more arbitrary, as in Figure 3.6c. Due to the synchronous approach, as long as the *size and type compatibility* of Definition 3.5.1 is satisfied, it is simple to achieve *timing compatibility* based on Definition 3.5.3. Moreover, the worst case timing behaviour of each composition is straight forward to represent as a formula in WCRT-algebra, as shown in Figure 3.6.




(a) Pipeline: $WCRT = M_1 \oplus M_2 \oplus M_3 \oplus \ldots \oplus M_n$




(b) Grid: $WCRT = (M_1 \odot M_2) \oplus (M_3 \odot M_4) \oplus \ldots \oplus (M_n \odot M_{n+1})$




(c) 2x1: $WCRT = (M_1 \odot M_2) \oplus M_3 \oplus (M_4 \odot M_5) \oplus M_6 \oplus \ldots$

Figure 3.6: Some possible concurrent MNN arrangements and the WCRT associated with the arrangements

**Definition 3.5.3.** Timing compatibility between SNNs in Figure 3.6 may be achieved using the following:

- When a SNN $M_1$'s output is connected to $M_2$ and the period of $M_1$ is $n$-ticks, then for timing compatibility, $M_2$ has $n$-`pause` statements at the start.
- When a SNN $M_1$ and $M_2$'s output is connected to $M_3$ and the period of $M_1$ is $n_1$-ticks and that of $M_2$ is $n_2$-ticks, then $M_3$ has $(n_1 \oplus n_2)$-`pause` statements at the start.
- When a connectivity self-loop is provided in $\Delta$, we break it explicitly.

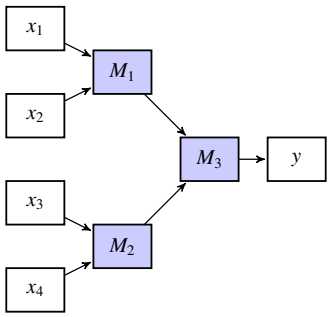



Figure 3.7: MNN implementation of a three-network XOR system

**Example 3.5.4.** *Consider three XOR MLP networks $\mathcal{M} = (M_1, M_2, M_3)$ in a system $H$. They are arranged in the formation shown in Figure 3.7, such that they compute $y = \big(x_1 \; xor \; x_2\big) \; xor \; \big(x_3$*

*xor* $x_4\big)$. *The overall network H has two meta-layers in* $\mathcal{L}$, *an input layer and an output layer. Hence, the global inputs* $I = \langle x_1, x_2, x_3, x_4 \rangle$, *and global outputs* $O = \langle y \rangle$. *The connectivity of* $\Delta \in H$ *is correctly defined, as the internal connections between* $M_1$, $M_2$, *and* $M_3$ *are matching, i.e. they are the same widths and types.*

*Let us execute the networks* $M \in \mathcal{M}$ *using the multi-periodic "layer by layer" approach. Hence, for each network M,* $WCRT = l_i \oplus l_h \oplus l_o$. *Using WCRT algebra, we can compute the system WCRT is of the form* $WCRT = (M_1 \odot M_2) \oplus M_3$, *ergo, the actual system* $WCRT = \Big( \big(l_1^i \oplus l_1^h \oplus l_1^o\big) \odot \big(l_2^i \oplus l_2^h \oplus l_2^o\big) \Big) \oplus \big(l_3^i \oplus l_3^h \oplus l_3^o\big)$ , *where* $l_y^x$ *indicates layer* $l_x \in L_y$ *(in network* $M_y$*).*

The Example in 3.5.5 discusses the issue of timing compatibility.

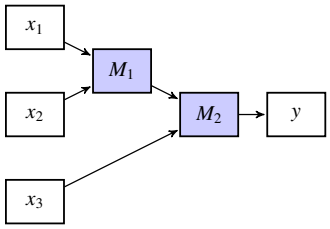



Figure 3.8: MNN implementation of a two-network XOR system

**Example 3.5.5.** *Consider the case of a two-input XOR system H, as presented in Figure 3.8. This will compute* $y = \big(x_1$ *xor* $x_2\big)$ *xor* $x_3$. *As in Example 3.5.4, there are two meta-layers in* $\mathcal{L}$. *Once again, the connectivity of* $\Delta$ *is correctly defined, as the inputs* $I_1 \in M_1$ *are simply in the global inputs I, and the inputs* $I_2 \in M_2$ *are provided as both a global input and as the output* $O_1 \in M_1$.

*However, as can be seen, the sources for* $M_2$ *will take different numbers of logical ticks to arrive. As a result, we must delay the input* $x_3$ *by some number of cycles. If we are using the multi-periodic "layer by layer" execution timing, this number will be* $|L \in M_1|$, *i.e. the number of ticks it will take to execute the layers of* $M_1$ *(in our case,* $|L| = 3$ *cycles). Or, if we are using a mono-periodic black-box approach, then we will need to delay the input* $x_3$ *by one cycle.*

All architectures of meta neural networks discussed thus far are devoid of instantaneous cycles. While such cycles in asynchronous settings are difficult to deal with, the synchronous approach offers two different alternatives. Synchronous compilers can detect such static cycles and flag them as *non-causal* [6] programs, and reject them outright. Alternatively, such instantaneous cycles can be broken by inserting an unit delay (i.e. a `pause` statement) to break the cycle.

## 3.6 Case studies and evaluation

We have developed a set of benchmark applications, written in Esterel, to evaluate the efficacy of the developed approach. These benchmarks are presented in Table **??**, and are also publicly available at *https://github.com/PRETgroup/sann*. All benchmarks in this chapter were developed in Esterel, C, and where appropriate, Python. Hard-real time benchmarks were analysed using the timing analysis tool Platin [22], and executed on a Patmos soft-core processor [62] on an Altera DE2-115 FPGA running at 50MHz. All neural networks were trained offline (i.e. pre-trained).

The benchmarks include an Energy Storage System (`ESS`) inspired by [12], the `AI-BRO` example from Section 3.2, the `XOR` network from Section 3.4, and an `ADDER` developed for pedagogy. These examples all use MLP-based SNNs. We have also developed an example involving a RNN called `HELLO`. These first benchmarks are all statically analysed to compute their WCRT and hence could be used in hard real-time applications. The `ESS` example is the most complex of these benchmarks, as it illustrates the concept of meta neural networks (Section 3.5) involving three neural networks with 74, 23, 18 neurons respectively.

As this is the first work on synchronous neural networks, we had to develop time analysable implementations from scratch. Unfortunately, we were unable to complete the implementation of more complex ANNs such that they met the requirements of the Platin tool. However, we did develop two additional interesting soft-real time examples to illustrate the power of the proposed methodology. The first example is a computer game called `RABBIT`, which was also implemented using Pthreads in Python. This program demonstrates the benefits of the synchronous approach. Lastly, we developed a CNN application called `SENSOR`, by reusing libraries from Darknet [54]. This example involved more than 1000 neurons, and we were able to implement both the "black box" and the "layer by layer" approaches.

**An Energy Storage System `ESS`** Energy Storage Systems (ESSs) are used to lower the costs of charging an electric vehicle (EV) [12]. Simply put, an ESS serves as an intermediary between the electrical grid and a large electrical load i.e. an EV. Usually, it is made up of a connection to the grid, a connection to the load, one or more electrical storage devices (such as batteries), and a controller to manage the system, as depicted in Figure 3.9.

ESSs work by shifting the burden of their electrical load to the electrical grid during periods of low demand, i.e. when the price is the cheapest. Then, as the load on the grid increases along with consequent increase in price, the ESS starts to supply its load from its battery instead. This results in a lower average cost for providing the load current. This system must be functionally correct while also meeting timing deadlines i.e. the system must respond to safety situations within a specified time bound.

Our `ESS` has an associated over-current detector. When this triggers, the system must evade

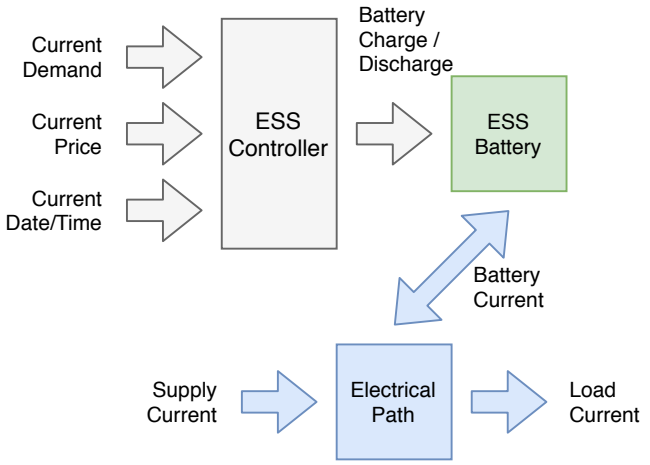



Figure 3.9: Example ESS components, showing IO and connections

this unsafe situation within 30 milli-seconds to activate a circuit breaker, which must take effect within this deadline. However, the response time of the overall ESS controller is much less strict, as it has a soft-real time *quality of service* deadline of one second. This is primarily because the demand input of the ESS does not need to be sampled faster than every second for suitable behaviour. In addition, the price of electricity changes only once every 10 minutes.

We propose a meta neural network, involving three 3-layer MLP SNNs, to decide when to charge and discharge the battery in our ESS. These networks operate synchronously with a over-current detector that preempts the system by activating a circuit breaker. The architecture of this meta neural network is presented in Figure 3.10.

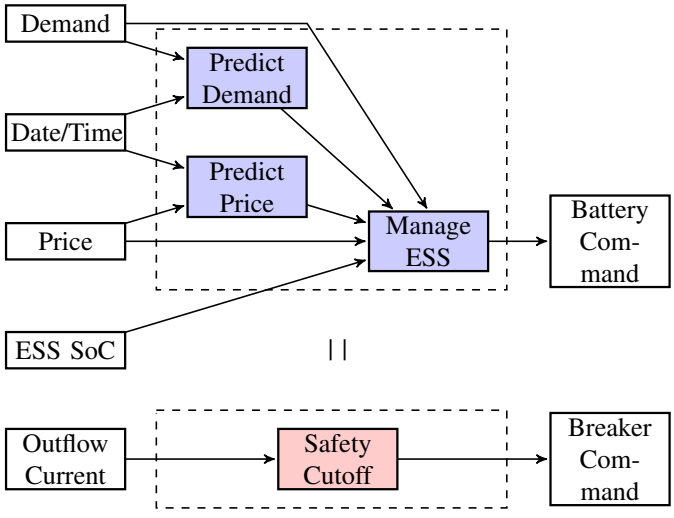



Figure 3.10: ESS SNN / Safety Cutoff arrangement

Training this system is complex; back-propagation with gradient descent cannot be used to train it as the outputs are not known beforehand. Instead, Q-learning [66], a type of reinforcement learning, was used for training the network. Reinforcement learning trains an ANN without knowledge of the best possible output(s). It trains based on the reinforcement of the

reward of any given action $A$, with regard to the current state $S$. An action that is *good* will be given a high reward, and the SNN will get its weights updated in an attempt to preserve the behaviour. A *bad* action, however, will result in the weights being modified so that it will have a reduced chance of choosing that action in that state again. Using this system, Q-learning attempts to train an SNN to follow the path of highest reward.

In order to validate this design, we performed WCRT analysis of both the mono-periodic and multi-periodic implementations. These results are presented in Table 3.3.

Table 3.3: WCRT and response time results for the different ESS executions

| Approach | WCRT (ms) | Response Time (ms) |
|---|---|---|
| Black-box | 14 | 14 |
| Layer by layer | 9.8 | 58.8 |

As can be seen, the WCRT for the mono-periodic "black-box" approach is $14ms$, making the total worst-case cut-off time $28ms$ (one cycle to capture value, one cycle to emit command). This only gives $2ms$ for the hardware to respond to a digital signal to cut the circuit, which may not be sufficient.

Considering this, we developed a multi-cyclic "layer by layer" approach, that results in a WCRT of $9.8ms$, giving a total worst-case cut-off time time of $< 20ms$. This gives sufficient time for the physical hardware to respond and break the circuit. In addition, although changing the implementation to multi-cyclic does increase the ANN response time by 54.8ms, it does not come close to the deadline of the ESS battery command, which has a one second soft-real time deadline.

**AI-BRO** Introduced in Section 3.2, and as depicted in Figure 4.1, AI-BRO uses three different SNNs working together to guide an automatic vehicle. For this case study, the SNNs were simple MLPs. Each was trained using back-propagation with a gradient descent approach [76], as the correct outputs for each of the inputs could be exactly specified. For instance, AI-BRO, a front sensor reading of $(010)_2$ implies an obstacle directly in front of the vehicle. Therefore, the vehicle must either turn left, turn right, or stop. Consequently, it should output $(101)_2$.

Once created, WCRT values were able to be derived for both mono-cyclic and multi-cyclic implementations, and are provided in Table 3.4.

Table 3.4: WCRT and response time results for the different AI-BRO executions

| Approach | WCRT (ms) | Response Time (ms) |
|---|---|---|
| Black-box | 2.8 | 2.8 |
| Layer by layer | 1.9 | 11.4 |
| Neuron by neuron | 1.8 | 10.8 |

**XOR / ADDER** In addition to the exclusive-or MLP XOR used as an example in Sections 3.4 and 3.5, an MLP ADDER was also created that could add two 31-bit numbers together. These two networks were both straightforward to train and analyse due to their simple 3-layer designs, and their WCRTs are presented in Table 3.5.

Table 3.5: WCRT and response time results for the different XOR and ADDER executions

| Approach | WCRT (ms) | Response Time (ms) |
|---|---|---|
| XOR | | |
| Black-box | 0.82 | 0.82 |
| Layer by layer | 0.57 | 1.71 |
| Neuron by neuron | 0.6 | 1.8 |
| ADDER | | |
| Black-box | 0.49 | 0.49 |
| Layer by layer | 0.39 | 1.17 |
| Neuron by neuron | 0.33 | 0.99 |

**HELLO** The HELLO system is a 3-layer RNN which was trained to learn the pattern of the word "hello". We implemented this system to demonstrate that the synchronous approach works with any type of ANNs, not just MLPs. The results of this system are presented in Table 3.6.

Table 3.6: WCRT and response time results for the different HELLO executions

| Approach | WCRT (ms) | Response Time (ms) |
|---|---|---|
| Black-box | 2.22 | 2.22 |
| Layer by layer | 1.33 | 3.99 |

**RABBIT and SENSOR** The RABBIT game was designed to highlight the benefits of synchronous programming while designing meta neural networks. The game is played in turns between two teams, two wolves versus one rabbit, who are all moving in the same grid. Each animal in the game has its own functional ANN, which all operate synchronously, respecting causality and avoiding deadlocks. The same system was also implemented in Python (using Pthreads), without the use of any synchronization primitives, and we found that the outputs of the two versions were strongly correlated with a Pearson's coefficient value of 0.999999802345.

Deep Neural Networks (DNNs) have many industrial applications compared to any other type of ANN. This is largely due to their ability to train to extremely large amounts of data. We implemented a SENSOR application using a DNN. An open source library called Darknet [54] was used to implement this CNN. Darknet provides a platform for training of many types of ANNs in C, including CNNs. Using this system, an Esterel application was developed to recognize images using the black-box and the layer by layer approaches.

Table 3.7: Combined WCRT results comparing the ESS, AI-BRO, XOR, ADDER and HELLO benchmarks, with the added comparison to MNN2C

| Approach | WCRT (ms) | Response Time (ms) | Size (neurons) |
|---|---|---|---|
| ESS | | | |
| Black-box | 14 | 14 | 125 |
| Layer by layer | 9.8 | 58.8 | 125 |
| MNN2C | 4.41 | 4.41 | 125 |
| AI-BRO | | | |
| Black-box | 2.8 | 2.8 | 35 |
| Layer by layer | 1.9 | 11.4 | 35 |
| Neuron by neuron | 1.8 | 10.8 | 35 |
| MNN2C | 2.29 | 2.29 | 35 |
| XOR | | | |
| Black-box | 0.82 | 0.82 | 6 |
| Layer by layer | 0.57 | 1.71 | 6 |
| Neuron by neuron | 0.6 | 1.8 | 6 |
| MNN2C | 0.53 | 0.53 | 6 |
| ADDER | | | |
| Black-box | 0.49 | 0.49 | 5 |
| Layer by layer | 0.39 | 1.17 | 5 |
| Neuron by neuron | 0.33 | 0.99 | 5 |
| MNN2C | 0.37 | 0.37 | 5 |
| HELLO | | | |
| Black-box | 2.22 | 2.22 | 14 |
| Layer by layer | 1.33 | 3.99 | 14 |

## 3.7 Meta Neural Network to C (MNN2C)

Model-Based Designs (MBDs) is a traditional approach to designing systems [69]. As opposed to design by trial-and-error, MBD builds formally defined, safer systems [69]. This approach to creating systems is highly preferred where the safety of the system, e.g. systems that interact with humans, is concerned. This is relevance where AI systems are created that interact with the same environment as humans, such as the Energy Storage System (ESS) shown in Section 3.6.

The ability to generate formally defined system models from existing ANNs allows for potentially unsafe and unpredictable ANNs to be re-implemented in such a way that they are safe and predictable. This allows the use of such ANNs in systems where safety is critical, e.g. Cyber-Physical Systems (CPS).

Keras [14] is an ANN Python library, able to use TensorFlow as a backend. Keras enables the quick, easy and highly adaptable training of various ANNs. The ANNs generated by Keras are not safe: they are not formally defined, and any timing must be done using measurement based timing. This poses a problem to the use of the ANNs in CPS, since CPS have strict rules and protocols that the software must follow.

Meta Neural Network to C (MNN2C) is an ANN compiler that aims to produce Meta Neural Network (MNN) models in C using pre-existing, Keras-trained ANNs. The C code that MNN2C produces is not only time-predictable, but the generated MNNs are also formally defined in 3.5.2. This means that the MNN models generated can be used in CPS with the knowledge that these MNNs are rigorously, mathematically defined. Additionally, MNN2C allows for the simple, easy and quick training of ANNs which can then be implemented in existing C systems.

### 3.7.1 Results

The software generated by MNN2C was not only highly accurate, with a correlation coefficient of 1.0 compared to the original Keras generated ANN, but also had a lower static WCET compared to the code originally created in this chapter. Using the built-in WCET compiler, the timing results were easily found for each time-predictable example from Table 3.7 and are shown in Table 3.8. This table shows that the structure of the MNN being analysed has a large influence on the timing properties of that ANN. The ESS MNN, which has a lot fewer non-linear activation functions (using a lot of Re-Lu activations compared to Tanh) has a much larger decrease in WCET compared to the other benchmarks, which average a 26% decrease. This shows that the MNNs generated by MNN2C better manage the implementation of the activation functions it uses, giving more accurate WCET of the MNN.

Table 3.8: MNN2C static WCRT results for the `ESS`, `AI-BRO`, `XOR` and `ADDER` benchmarks

| Benchmark | Original Black-box WCRT (ms) | MNN2C Black-box WCRT (ms) | % **decrease** |
|---|---|---|---|
| `ESS` | 14 | 4.41 | 68.5 |
| `AI-BRO` | 2.8 | 2.29 | 18.21 |
| `XOR` | 0.82 | 0.53 | 35.37 |
| `ADDER` | 0.49 | 0.37 | 24.49 |

### 3.7.2 Future Work

MNN2C currently only generates time predictable, formally defined C code ANNs based off of Keras ANNs. While this is a huge step, this is not the limit when it comes to using ANNs. MNN2C can be expanded to generate code that is implementable on various hardware platforms, such as Field-Programmable Gate Arrays (FPGAs), graphics cards and many others. This would allow for the implementation of ANNs in various CPS besides those that just use C software.

## 3.8 Discussion

Neural networks are being used in many Cyber-Physical Systems (CPS) to influence real-time decisions with safety implications. Consequently, there has been a convergence of AI and formal methods to ensure that such applications operate safely at all times [63]. This has led to the development of new techniques for functional verification of AI-based CPS. However, the issue of timing verification has received scant attention.

This paper, for the first time, develops neural networks with timed semantics called Synchronous Neural Networks (SNNs). Our approach develops periodic networks, where the response time of a network may be between 1-tick (called *mono-periodic* networks) to $n$-ticks (called *multi-periodic* networks). We propose a timing semantics of such networks using WCRT-algebra [70]. Multi-periodic networks are introduced to reduce the system reaction time, while keeping the response time within the specified deadline (see the ESS case study in Section 3.6).

For complex CPS, we propose meta neural networks for designing concurrent applications where the output from one network influences other networks. We provide several alternative architectures and study such systems in Section 3.6. Overall, this paper develops a new approach for the design of time predictable Cyber-Physical Systems (CPS) involving interacting AI modules.

As this is the first work on synchronous neural networks, there are some limitations and hence several avenues for future research exist. First, all benchmarks use off-line learning. We are already working on developing techniques for on-line learning in Esterel. A second limitation is that we are relying on "compiling away" Esterel concurrency, which is not ideal for many-core processors. We will develop parallel implementations of SNNs in the near future. Additionally, we developed SNNs to ensure timing correctness. However, synchronous semantics could also aid in the validation of functional correctness for NNs, due to their deterministic and reactive nature. This aspect will be examined in the future. Finally, we could have considered data-flow synchronous languages like Lustre [6] for SNNs. However, we selected Esterel due to the nature of CPS applications, which combine data and control-flow, where the Esterel style seems more natural.

# 4

# Runtime Enforcement of Synchronous Neural Networks

## 4.1 Introduction

Chapter 3 presented Synchronous Neural Networks (SNNs), as periodic networks with their systematic composition as Meta Neural Networks (MNNs), to ensure that AI applications operate in a time predictable manner. This chapter deals with the related issue of functional verification of ANNs. We propose the technique of Run-time Enforcement (RE) [51] along with SNNs and MNNs.

## 4.2 Case study: Autonomous Vehicle (AV) braking

Autonomous Vehicles (AVs) are Cyber-Physical Systems (CPS) that are safety critical: any faults in their operation can lead to accidents resulting in injuries or fatalities. For instance, both Tesla's and Uber's have had accidents with their AVs [16] [65]. In this chapter, we take inspiration from such accidents and create a case study involving the braking mechanism of AVs.

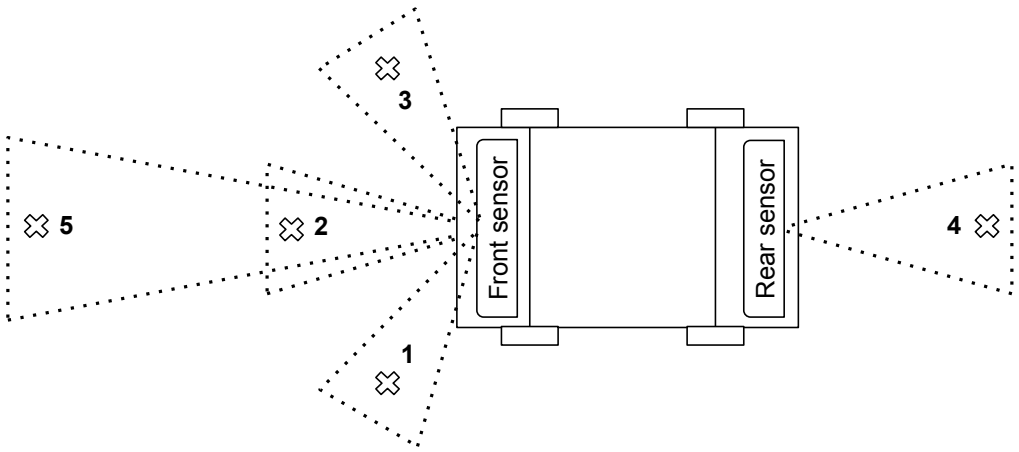



Figure 4.1: Sensor layout for the AV example.

### 4.2.1 Autonomous Vehicle (AV) system

We propose an abstracted Autonomous Vehicle (AV) represented in Figure 4.1 as our case study. We deal with the linear, forward movement of the AV and the braking involved with such movement. The AV system consists of one Autonomous Vehicle (AV), interacting with other vehicles and pedestrians in its proximity, run by an autonomous controller, with 5 directional, sensor inputs (see Figure 4.1). The sensor package consists of five cameras, each with its own, unique field of view and a Light Detection and Ranging (LiDAR) sensor. Each camera picks up a certain area relative to the AV, and each camera has an associated LiDAR reading for the area is is detecting. Each of the five cameras feeds into a Meta Neural Network (MNN) ensemble [39] of SNNs, using the Darknet library [53], while the LiDAR readings are passed directly to the controller. These ensembles are covered in the Chapter 2 of this thesis. These SNN ensembles classify their input image and provide a confidence level for the classified image, before passing this information to the controller. The controller SNN is a MLP, and decides the best course of action given the environment and the status of the vehicle itself. The current status of the vehicle is noted by the current speed of the vehicle ($S$) and the previous speed of the vehicle($S'$), taken from one synchronous tick in the past using the Esterel keyword *pre*. The controller then outputs one of three simple commands to the vehicles actuators which are represented by an accelerator and a brake. These commands are as follows: accelerate ($A$); brake softly ($B_S$); and brake hard ($B_H$). A block diagram of the AV used in this system is shown in Figure 4.2.

The system controller, MNNs and sensor readings are all synchronous and, as such, the system is run using synchronous semantics. The synchronous language Esterel is used to run the entirety of the system controller and its connected components. The entire system is run in two logical ticks: a single tick for the plant and a single tick for the controller MLP. The MNN ensembles are run concurrently with each other, followed by the execution of the controller

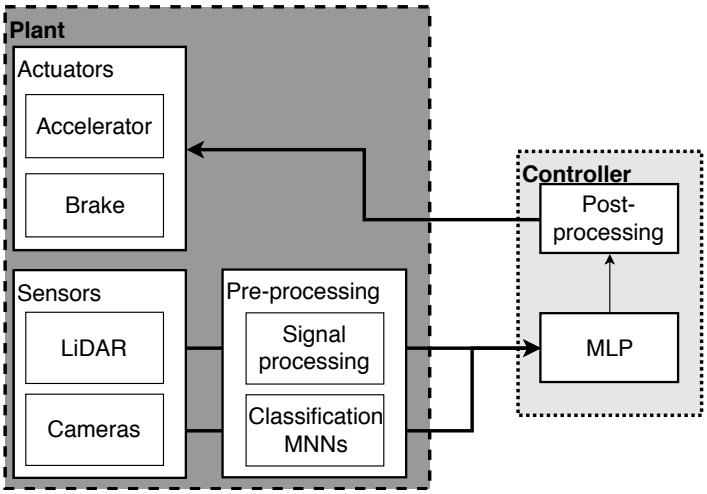



Figure 4.2: Block diagram of the AV system used in the case study.

MLP on the MNN ensembles' outputs. The environment update is run in the same tick as the sensor readings and plant execution, with the environment updating before new sensor readings are taken. Tables 4.1 and 4.2 show the environment encoding of the system.

Table 4.1: Table showing the encoding of each of the AV's sensor detection

| Detected object (O) | Symbol | Numerical Value |
|---|---|---|
| Pedestrian | P | 0 |
| Car | C | 1 |
| Nothing | N | 2 |

Table 4.2: Table showing the encoding of the AV's environment

| Item | Symbol | Possible values | Possible numerical values |
|---|---|---|---|
| Camera Object $x$ | $O_x$ | [P, C, N] | $[0,1,2]$ |
| Camera Object $x$ Speed | $O_{x_S}$ | [Stationary, Slow, Fast] | $[0,1,2]$ |
| Camera Object $x$ Direction | $O_{x_D}$ | [North, East, South, West] | $[0,1,2,3]$ |
| LiDAR Object $x$ | $L_x$ | [P, C, N] | $[0,1,2]$ |
| Speed | S | [Stationary, Slow, Fast] | $[0,1,2]$ |
| Previous Speed | S' | [Stationary, Slow, Fast] | $[0,1,2]$ |

Consider Table 4.2 and Figure 4.1. The environment for this AV system consists of the AV in question, and up to five other "objects" ($O$) in the environment. Each sensor position in the AV system (Figure 4.1) can detect up to one object at that position at any one time, represented by the numbered position at which the object was detected. Each object can be a person ($P$), a car ($C$) or nothing ($N$). Each object detected by a camera is represented by the sensor position it is detected in and the type of object it is. The system used three possible speeds (stationary,

slow, fast) represented by an integer value (0, 1, 2) respectively. The directions were limited to (North, East, South and West), represented by the numerical values (0, 1, 2 and 3) respectively.

**Example 4.2.1.** *A person behind the vehicle would be* $O_4 = 0$*, a vehicle directly in front would be* $O_2 = 1$ *and nothing to the right side of the AV would be* $O_3 = 2$*. Consider speed and direction: object 5 moving quickly would be* $O_{5_S} = 2$*; the object to the left of the vehicle facing westward would be* $O_{1_D} = 3$*.*

The environment updates in such a way that objects move in the direction they are facing at a speed relative to the AV. For example, a pedestrian facing the road and moving quickly would be placed in front of the vehicle in position 2 on the next tick. However, a pedestrian moving away from the road, or not moving at all, would be removed from the sensors detection areas on the next tick. To emulate camera behaviour we rely on the Visual Object Classes (VOC) 2012 dataset [19]. Each tick, each camera will select an image from this dataset for processing by the corresponding MNN ensemble.

Due to this design of this system, it is possible for the vehicle to behave badly in various ways. These include speeding, unnecessarily braking, hitting other vehicles on the road, hitting pedestrians on the road and even not driving at all. All of these scenarios can result in fatalities, thus classifying this system as a safety critical. To have a system that is safe, policies need to be enforced that monitor the system's inputs and outputs and ensure that none of the above scenarios take place under any circumstances.

### 4.2.2 Run-time enforcer for the Autonomous Vehicle (AV) system

We propose the use of a run-time enforcer [51] between the plant and controller (see Figure 4.3), to provide formal guarantees about the controller SNN's functional safety. This enforcer will monitor the I/O events of the controller SNN and *edit* unsafe events so that controller functions safely.

This run-time enforcer is set between the reactive controller and the plant and monitors the I/O of the controller. The controller consists of a decision MLP which feeds into a post-processing unit. This receives inputs from the signal processing unit and MNNs ensembles through the input enforcer. The output from the controller is passed through the output enforcer before being sent to the appropriate actuators. The run-time enforcer for the AV system enforces a set of policies ($\mathcal{V}$), described and defined in English here:

$\mathcal{V}_{ped}$ (Figure 4.4): Ensure that the car does not behave unsafely around pedestrians.

The automaton starts in state $l_{drive}$, with a transition to itself if the vehicle is not braking and no pedestrian is in front. If the vehicle attempts to drive into a pedestrian or suddenly slam on the brakes for no reason, the violation transition to $l_v$ is taken as this is a potentially dangerous situation.

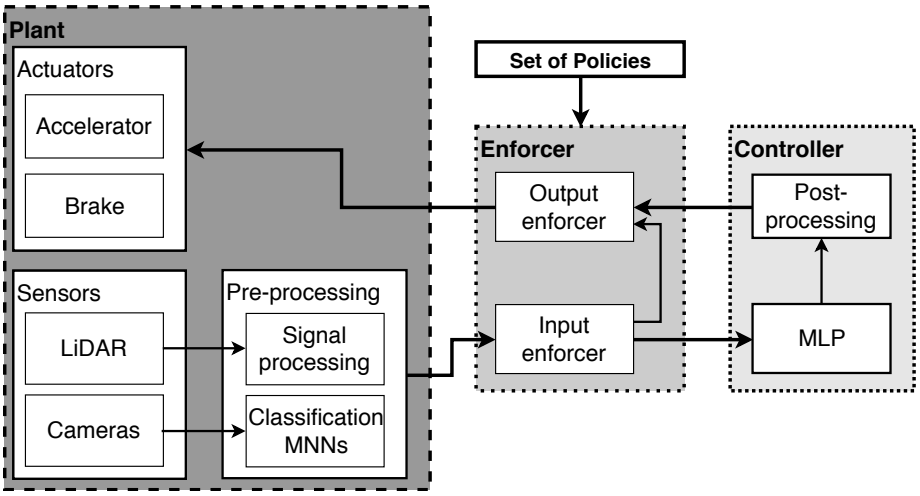



Figure 4.3: Block diagram of the AV system, with run-time enforcer, used in the case study.

When a braking action is started and a pedestrian is in front of the vehicle, the transition from $l_{drive}$ to $l_{brake}$ is taken and the timer $t$ is set to 0. From the $l_{brake}$ state, there is a transition back to $l_{brake}$ if $t$ is less than $T_{lim}$ ($t < T_{lim}$ where $T_{lim} = 3$), or the vehicle is still braking, or there is a pedestrian in front, and the vehicle must not be accelerating. If the timer has reached (or passed) $T_{lim}$ ($t >= T_{lim}$), the vehicle has stopped braking and there is no pedestrian ahead, then the transition back to $l_{drive}$ is taken because it is safe to continue driving. If the vehicle attempts to accelerate before the timer reaches $T_{lim}$ ($t < T_{lim}$) or stops braking then the violation transition is taken to prevent potentially harmful actions.

If a pedestrian remains off to the side of the vehicle, then either the vehicle should cruise, since pedestrians keeping to the side-walk are safe from the vehicle on the road.

$\mathcal{V}_{car}$ (Figure 4.5): Ensure that the car does not drive into other vehicles. This automaton only has a single safe state $l_{drive}$ and a violation state $l_v$. The transition to the violation state occurs when the vehicle is going faster that the car in front of it ($O_{2_S} < S \mid O_{5_S} < S$) and not braking appropriately. This could result in a situation where the AV would be forced to try overtake or slam on the brakes, neither of which are safe actions. If the vehicle is going slower than the car behind it ($S < O_{4_S}$) and not accelerating, and there is no pedestrian in front of the vehicle, the transition to the violation also occurs. This could result in unsafe circumstances where the car behind is forced to slow down or take over. The transition to the safe state occurs if there is no violation.

$\mathcal{V}_{drive}$ (Figure 4.6): The vehicle may not exceed the safe speed limit. This automaton only has a single safe state $l_{drive}$ and a violation state $l_v$. This transitions to a violation when the vehicle attempts to accelerate while driving at the maximum speed (in this case when $S = 2$), since speeding is against the law.

$\mathcal{V}_{cnn}$ (Figure 4.7): The MNN ensembles and the LiDAR should agree on the detected objects. This automaton only has a single safe state $l_{drive}$ and a violation state $l_v$. Considering

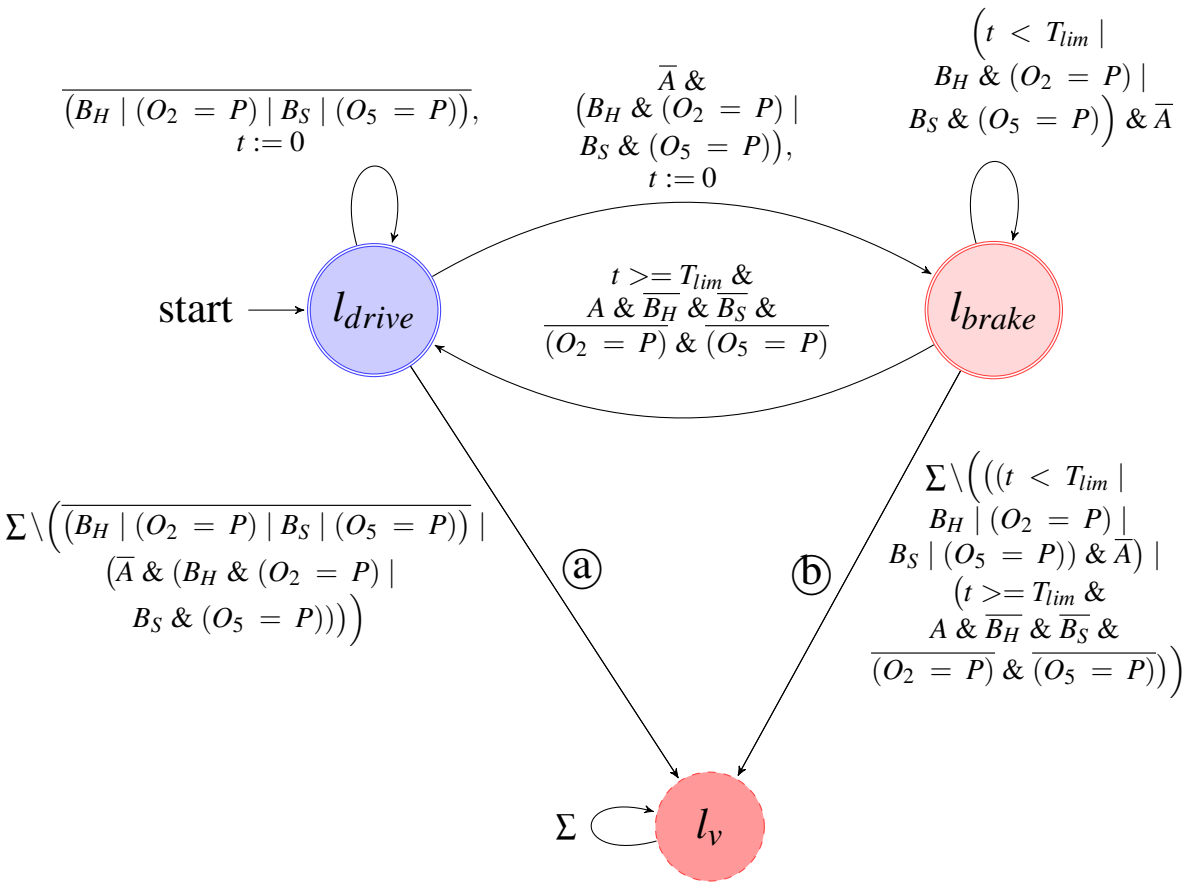



Figure 4.4: VDTA for describing the safety policy for the pedestrian detection $\mathcal{V}_{ped}$

each sensor $x \in [1,5]$, the output of the CNN ensemble network $x$ ($O_x$) must match the corresponding LiDAR value ($L_x$). If they do match the transition back to the safe state is taken, and if they do no match ($\overline{O_x = L_x}$) then the violation transition is taken. If the confidence is high, and there is a mismatch, the output should be classified as a pedestrian ($O_x = P$) to consider the worst case and treat the unknown object with utmost care.

It is possible to specify policies as VDTA which encompass multiple safety rules. Though we have presented one safety rule per VDTA, it is possible to specify policies as VDTA which encompass multiple safety rules.

### 4.2.3 Artificial Intelligence (AI) for the Autonomous Vehicle (AV) system

The AIs in this case study include the MNN ensembles in the plant and a single MLP SNN for the controller.

In this case study, each MNN ensemble consisted of three CNNs feeding into an ensemble function. Each CNN in the ensemble provided its output to an ensemble function, which then produced higher grade output using a custom averaging function based off the CNNs' class scores. Due to the synchronous nature of the CNNs, each CNN was run in synchronous

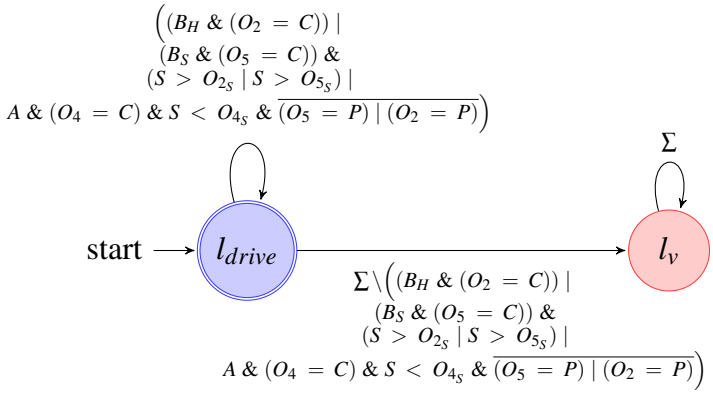



Figure 4.5: VDTA for describing the safety policy for the car detection $\mathcal{V}_{car}$

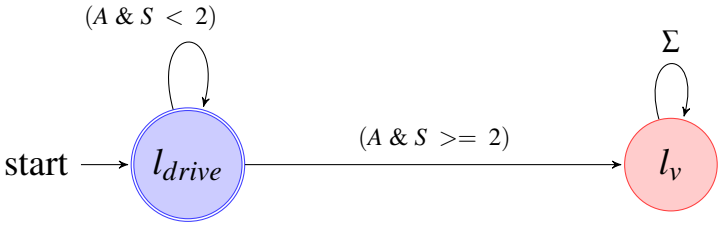



Figure 4.6: VDTA for describing the safety policy for driving according to the rules $\mathcal{V}_{drive}$

concurrency with the others to provide output to the ensemble function.

Each CNN consists of 10 layers: a combination of convolutional, maximum pooling and average pooling layers. Each CNN was trained for up to 100,000 epochs on the Visual Object Classes (VOC) 2012 data set, using the Darknet C library [53] to perform back-propagation with gradient descent. The CNNs took an input image of 28x28 pixels and output three class probabilities; one for a person, one for a car and one for nothing.

The controller MLP consisted of three layers of artificial neurons, with seventeen input neurons, ten hidden layer neurons and three output neurons. The implemented system and environment were complex enough that back-propagation was not an applicable method to train the controller, as it was not possible to map out every possible input set with a known output set. The controller MLP was trained used a type of reinforcement learning called Q-learning [66]. This training processes works by reinforcing good actions with positive rewards, and removing bad actions with negative rewards. Like the CNNs, this acANN was also trained for 100,000 epochs using this technique. The controller took the seventeen different inputs, as an array of integers, in the following order (using the previously defined labels in Table 4.2): $(S, S', O_1, O_2, O_3, O_4, O_5, O_{1_D}, O_{2_D}, O_{3_D}, O_{4_D}, O_{5_D}, O_{1_S}, O_{2_S}, O_{3_S}, O_{4_S}, O_{5_S})$. The outputs of the controller were a binary array representing the actions that could be taken: $(A, B_S, B_H)$, where each value could be 1 or 0. An absence of all the outputs is seen as a "cruise" action, where the vehicle does not accelerate.

Figure 4.8 is an expanded diagram of the AV system, showing the ANN layout for this

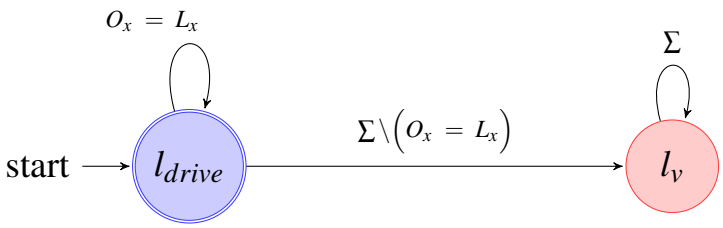



Figure 4.7: VDTA for describing the safety policy for the CNN ensembles $\mathcal{V}_{cnn}$

system. Each MNN ensemble is labelled by its corresponding sensor number and shows three CNNs feeding into a single ensemble function for each MNN. The ensemble outputs are then passed to the controller SNN MLP via the input run-time enforcer. The controller's decision is made by its MLP, and then passed back to the vehicle via the output enforcer.

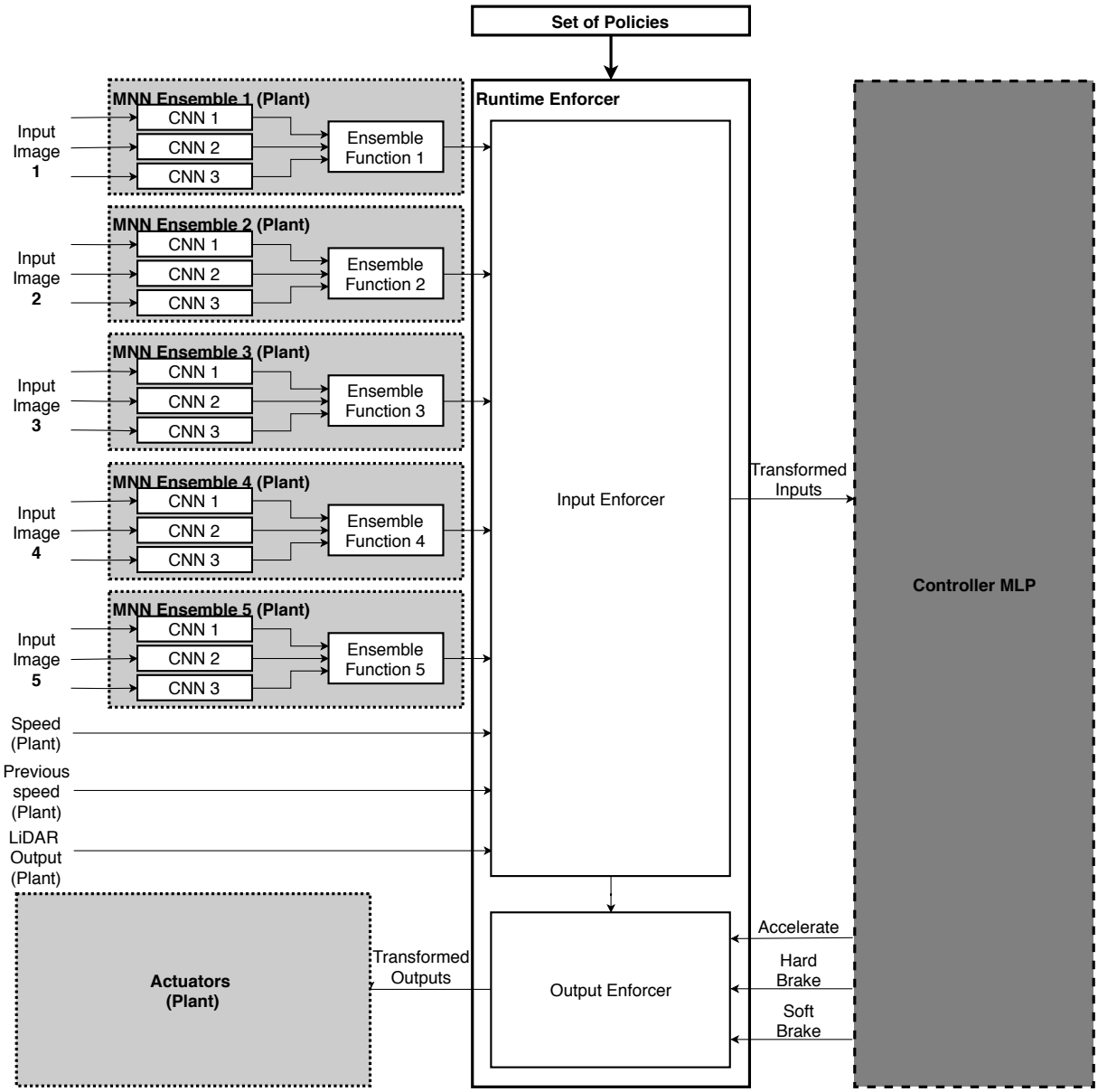



Figure 4.8: Diagram showing the SNN for the AV, and its interaction with the plant via the enforcer.

## 4.3 Defining Safety Specifications for Synchronous Neural Networks

Chapter 3 introduced SNNs as time-predictable ANNs to address the timing verification of ANNs for CPSs. The functional verification of ANNs has not yet been addressed. This chapter tackles the problem of the functional verification of ANNs by proposing the combination of the previously defined SNNs and Run-time Enforcement (RE) [51].

We propose the use of bi-directional run-time enforcers [51] to enforce the I/O events of SNNs at run-time. These run-time enforcers are introduced in more detail in Chapter 2. How-

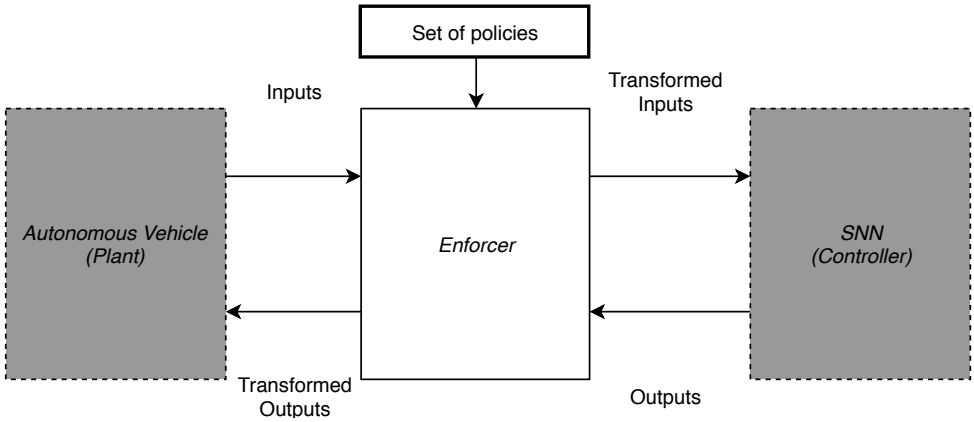



Figure 4.9: Run-time enforcer between a SNN and the Autonomous Vehicle (AV), to monitor the I/O events of the SNN.

ever, with the complex I/O of SNNs, an enforcer that edits only binary events is not sufficient. Additionally, the edit functions introduced by [51] are not sufficient with more complex DTA. Consider Figure 4.9: we wish to enforce the inputs and outputs events of the SNN, defined by some timed safety policy φ. However, the outputs from the SNN are 32-bit integer values, but the previously defined DTAs can only express binary signals. Thus, we cannot use a DTA to to specify the safety policies for this SNNs, nor can we enforce valued guards and transitions. Instead, we propose the use of Valued Discrete Timed Automatas (VDTAs) [43] to enforce valued I/O events of SNNs.

### 4.3.1 Valued Discrete Timed Automata (VDTA): Defining Safety Policies for CPS

We consider our industrial CPS systems to have finite ordered sets of valued input channels $I = \{i_1, i_2, \ldots i_n\}$ and valued output channels $O = \{o_1, o_2, \ldots o_n\}$. A VDTA can be seen as an automaton with a finite set of locations, a finite set of discrete clocks used to represent time evolution, and external input (resp. output channels) called "external variables" which are used for representing system data. They also have internal variables which are used for internal computation, compared to the external variables which model the data carried by the actions from the monitored system (resp. environment). In a VDTA, a transition consists of an action carrying values of external variables, a guard on internal variables, external variables and clocks, and an assignment of internal variables, and reset of clocks.

Before we look into the formal definition of VDTA, let us consider an example.

**Example 4.3.1.** *The VDTA for the pedestrian safety policy, $\mathcal{V}_{ped}$, is used for this example, and refers to the sensors in Figure 4.1. This VDTA specifies that driving towards a pedestrian in-front of the vehicle (sensor 2) and not braking is a violation, and approaching a pedestrian from a distance (sensor 5) and not starting to slow down is also violation. This policy has been*

*described in more detail in the previous section.*

*This is encoded as a VDTA with a set of locations* $L = \{l_{drive}, l_{brake}, l_v\}$ *and accepting locations* $F = \{l_{drive}\}$*, with the initial state being* $l_{drive}$*. The set of external variables* $C = \{O_2, O_5, O_{2_v}, O_{5_v}, S, A, B_S, B_H\}$ *are all 32-bit signed integers, where* $O_2$*,* $O_5$*,* $O_{2_S}$*,* $O_{5_S}$ *and S are input channels and A,* $B_S$ *and* $B_H$ *are output channels. The set of actions* $\Sigma = \{O_2, O_5, O_{2_S}, O_{5_S}, S, A, B_S, B_H\}$*, and the set of internal variables* $V = \{T_{lim}\}$*, where* $T_{lim}$ *is a 32-bit signed integer. X, the finite set of discrete clocks, is limited to t for this example:* $X = \{t\}$*.*

*In this VDTA there are two violation transitions, labelled ⓐ and ⓑ in Figure 4.4. ⓐ occurs when the vehicle does anything else other than braking when there is a pedestrian in-front or not braking when there is no pedestrian ahead of the vehicle. ⓑ can also occur when the vehicle has not braked enough within a certain period of time* $T_{lim}$*, or when the vehicle remains braking longer than is safe or necessary. This represents the vehicle taking further unsafe actions when already in an unsafe state.*

Let us now consider in more detail the formal syntax and semantics of VDTAs. For a variable (resp. channel) $v$, $\mathcal{D}_v$ denotes its domain, and for a tuple of variables $V = (v_1, \ldots, v_n)$, $\mathcal{D}_V$ is the product domain $\mathcal{D}_{v_1} \times \cdots \times \mathcal{D}_{v_n}$. A valuation of the variables in $V$ is a mapping $\nu$ which maps every variable $v \in V$ to a value $\nu(v)$ in $\mathcal{D}_v$. Let $X = \{x_1, \ldots, x_k\}$ be a finite set of integer variables representing discrete clocks. A *valuation* for $x$ is an element of $\mathbb{N}$, that is a function from $x$ to $\mathbb{N}$. The set of valuations for the set of clocks $X$ is denoted by $\chi$. For $\chi \in \mathbb{N}^X$, $\chi + 1$ (which captures the ticking of the digital clock) is the valuation assigning $\chi(x) + 1$ to each clock variable $x$ of $X$. Given a set of clock variables $X' \subseteq X$, $\chi[X' \leftarrow 0]$ is the valuation of clock variables $\chi$ where all the clock variables in $X'$ are assigned to 0.

**Definition 4.3.2** (Syntax of VDTAs)**.** A VDTA is a tuple $\mathcal{A} = (\Sigma, L, l_0, X, V, C, \Theta, F, \Delta)$ where:

- $\Sigma$ is a non-empty finite set of actions, and an action $a \in \Sigma$ has a signature $sig(a) = (t_0, t_1, \ldots, t_k)$ which is a tuple of types of the external variables,
- $L$ is a finite non-empty set of locations, with $l_0 \in L$ the initial location, and $F \subseteq L$ the set of accepting locations;
- $X$ is a finite set of discrete clocks;
- $V$ is a tuple of typed internal variables;
- $C$ is a tuple of external variables, where $C = I \cup O$, where $I$ is the set of input channels, and $O$ is the set of output channels;
- $\Theta \subseteq \mathcal{D}_V$ is an initial condition which is a computable predicate over $V$;
- $\Delta$ is a finite set of transitions, and each transition $t \in \Delta$ is a tuple $(l, a, c, G, A, l')$ also written $l \xrightarrow{a(c), G(V,c), V' := A(V,c)} l'$ such that,
  - $l, l' \in L$ are respectively the origin and target locations of the transition;
  - $a \in \Sigma$ is the action, and $c = (c_1, \ldots c_k)$ is a tuple of external variables local to the transition;

- $G = G^D \wedge G^X$ is the guard where
  - $G^D \subseteq \mathcal{D}_V \times \mathcal{D}_{sig(a)}$ is a computable predicate over internal variables and external variables in $V \cup c$;
  - $G^X$ is a clock constraint over $X$ defined as a Boolean combinations of constraints of the form $x \sharp f(V \cup c)$, where $x \in X$ and $f(V \cup c)$ is a computable function, and $\sharp \in \{<, \leq, =, \geq, >\}$;
- $A$=$(A^D, A^X)$ is the assignment of the transition where
  - $A^D : \mathcal{D}_V \times \mathcal{D}_{sig(a)} \to \mathcal{D}_V$ defines the evolution of internal variables.
  - $A^X \subseteq X$ is the set of clocks to be reset.

A word is a sequence $\sigma = e_1 \cdot e_2 \cdots e_n$ where $\forall i \in [1,n] : e_i = a_i(\eta_i)$ where $a_i \in \Sigma$ is an action and $\eta_i \in \mathcal{D}_V$ is a vector of values of a tuple of variables $V$.

Policy VDTA are required to be *deterministic*, i.e. for any given state, the conjunction of any guards of any other outgoing transitions may not be satisfiable; and *complete*, i.e. for any given state at any given time and any given event, at least one transition guard is satisfied.

### 4.3.2 Semantics for VDTA

Let $\mathcal{A} = (\Sigma, L, l_0, X, V, C, \Theta, F, \Delta)$ be a VDTA. The semantics of $\mathcal{A}$ is a timed transition system, where a state consists of a location, and valuations of internal variables $V$ and clocks $X$, and actions associated with values of external variables in $C$.

**Definition 4.3.3** (Semantics of VDTAs)**.** The semantics of $\mathcal{A}$ is a timed transition system $[\![\mathcal{A}]\!] = (Q, q_0, Q_F, \Gamma, \to)$, defined as follows:

- $Q = L \times \mathcal{D}_V \times \mathbb{N}^X$, is the set of states of the form $q = (l, \nu, \chi)$ where $l \in L$ is a location, $\nu \in \mathcal{D}_V$ is a valuation of internal variables, $\chi$ is a valuation of clocks;
- $Q_0 = \{(l_0, \nu, \chi_{[X \leftarrow 0]}) \mid \Theta(\nu) = \mathsf{true}\}$ is the set of initial states;
- $Q_F = F \times \mathcal{D}_V \times \mathbb{N}^X$ is the set of accepting states;
- $\Gamma = \{a(\eta) \mid a \in \Sigma \wedge \eta \in \mathcal{D}_{sig(a)}\}$ is the set of transition labels;
- $\to \subseteq Q \times \Gamma \times Q$ the transition relation is the smallest set of transitions of the form $(l, \nu, \chi\rangle \longrightarrow a(\eta)\langle l', \nu', \chi')$ such that $\exists (l, a, c, G, A, l') \in \Delta$, with $G^X(\chi + 1) \wedge G^D(\nu, \eta)$ evaluating to true, $\nu' = A^D(\nu, \eta)$ and $\chi' = (\chi + 1)[A^X \leftarrow 0]$.

A *run* $\rho$ of $[\![\mathcal{A}]\!]$ from a state $q \in Q$ over a *trace* $w = a_1(\eta_1) \cdot a_2(\eta_2) \cdots a_n(\eta_n)$ is a sequence of moves in $[\![\mathcal{A}]\!]$: $\rho = q \xrightarrow{a_1(\eta_1)} q_1 \cdots q_{n-1} \xrightarrow{a_n(\eta_n)} q_n$, for some $n \in \mathbb{N}$. The set of runs from the initial state $q_0 \in Q$, is denoted $\mathrm{Run}(\mathcal{A})$ and $\mathrm{Run}_{Q_F}(\mathcal{A})$ denotes the subset of those runs *accepted* by $\mathcal{A}$, i.e., ending in an accepting state $q_n \in Q_F$.

**Example 4.3.4** (Run of a VDTA)**.** *An example run of the VDTA presented in Figure 4.4 is presented here. The functionality of this VDTA is explained in the previous section, and the AV*

*sensor positions are shown in Figure 4.1. Assume that the time the vehicle has to be braking is $T_{lim} = 3$. The starting state is $l_{drive}$, and the first I/O event is $(\{0,0,0,0,2\},\{\langle 0,0,0\rangle\})$. The inputs show that nothing is detected in-front of the vehicle it is cruising at a fast speed km/h ($S = 2$) and the vehicle does not accelerate or brake. Thus, the automaton remains in state $l_{drive}$. Next tick, $(\{0,1,0,1,2\},\{\langle 0,1,0\rangle\})$ occurs, i.e. a pedestrian ($O_5 = 1$) is detected quite far ahead of the vehicle, moving across the road at a slow pace ($O_{5_S} = 1$), and the vehicle is taking a soft braking action ($B_S = 1$) from a fast speed ($S = 2$). Since the outputs $O = \{\langle 0,1,0\rangle\}$, which is a soft brake $B_S$, and sensor 5 picks up a pedestrian a pedestrian $O_5 = 1 = P$ the system enters the unsafe state $l_{brake}$ and $t := 0$ is set. Then, $(\{1,0,0,0,1\},\{\langle 0,0,0\rangle\})$ is received, i.e. the pedestrian has stopped moving in the road and is now close to the vehicle ($O_2 = P$), however the vehicle is not taking any action, but rather cruising at a slow speed ($S = 1$). Since no braking is detected ($B_H = 0$), the violation transition $l_v$ is taken. As such, this run was non-accepting.*

### 4.3.3 Enforcing Non-accepting I/O Events

Enforcers are designed to prevent a system from generating an input/output trace that is non-accepting, such as Example 4.3.4. [51] proposed DTA semantics with two possible methodologies for editing non-accepting I/O events. These are *random* and *minimum* edits; a random edit chooses a random, accepting event from a list of accepting I/O events and minimum edit chooses the closest accepting event to the current non-accepting event. However, neither of these edits always useful for problems in real scenarios. Take Example 4.3.4, when the transition $l_{brake} \rightarrow l_v$ is taken because the vehicle does not slow down when approaching a pedestrian, $O_2$ can be edited to be nothing ($O_2 = 2$) so that $l_v$ is not entered, however this will not remove the danger that initially posed this transition since there really is a pedestrian in front of the vehicle. However, if the action in output $O$ was changed, e.g. the cruising action ($\{\langle 0,0,0\rangle\}$) in the example that caused the non-accepting trace was changed to a hard braking action ($\{\langle 0,0,1\rangle\}$), then the transition back to $l_{brake}$ would have been taken *and* the pedestrian in the road would have been safe.

In general, then, the designer of any given policy should also select their preferred edit actions out of the list of possible safe edits for each violation transition, thus ensuring practical runtime enforcement.

**Example 4.3.5** (Selected Edit Actions for a VDTA)**.** *In Figure 4.4, there are many different actions in $\Sigma$ that, when taken in a specific location $L$ result in a violation transition. In this example, a situation where the current location is $l_{drive}$ and the action is $\Sigma = \{1,0,1,0,1,1,0,0\}$, i.e. there is a pedestrian directly in front ($O_2 = 1$) of the vehicle moving slowly, while the vehicle is moving directly forward slowly ($S = 1$) and accelerating ($A = 1$). Thus, the violation transition ⓐ occurs.*

• *Transition* ⓐ*:* $A := 0$*,* $B_S := 0$ *and* $B_H := 1$

*The recovery for this violation transition is to suppress the accelerate signal (*$A = 0$*), and set the hard brake signal to present (*$B_H = 1$*). This instead changes the transition to* $l_{drive} \rightarrow l_{brake}$*, which is a safe transition that slows the vehicle down before it hits the pedestrian.*

## 4.4 Results

We have developed a set of benchmark applications to evaluate the efficacy of the developed approach. These benchmarks are presented in Table 4.3. All benchmarks were written in Esterel and C and run on Ubuntu 16.04, using an 4 core Intel i7-6700HQ processor at 2.6GHz and 4GB of RAM. The safety automata implemented according to the VDTA introduced earlier in this chapter.

Table 4.3: Table giving a brief description of each benchmark used in this chapter.

| Name | SNN Type(s) | #SNNs | #Enforced Policies |
|---|---|---|---|
| AV | MLP, CNN | 28 | 4 |
| ESS | MLP | 3 | 2 |
| RABBIT | MLP | 6 | 2 |

Benchmarks include (Table 4.3): an Autonomous Vehicle (AV) braking system, described in Section 4.2; an Energy Storage System (ESS) inspired by [12]; and a simple, two-team, zero-sum game called RABBIT. The ESS (described in Section 4.4.1) has two different policies; ensuring that the system battery's State of Charge (SoC) never drops too low or rises too high ($\varphi_{soc}$) and ensuring that the power by which the battery is charging or discharging is never too high ($\varphi_{charge}$). RABBIT (described in Section 4.4.3) has two policies to increase the efficiency of the players, rather than increasing safety in the system and the system is not safety critical. Each benchmark is analysed with and without RE in place, to show the improvement in each system's safety when using RE. Additionally, the overhead generated by each enforcer is also calculated. Table 4.4 lists every policy and provides a brief overview of each policy and it's measured overhead.

The AV system's policies had the highest overall overhead, compared to relatively low overhead generated by the ESS and RABBIT game. This was expected, as the policies for the AV system were a lot larger and more complex than the other two benchmarks' policies. The average overhead for all the benchmarks' policies was just over 6%.

### 4.4.1 Energy Storage System (ESS)

The ESS is inspired by [12], in which an Electric Vehicle (EV) charging station has a separate battery that it uses to assist the charging of the EVs, with the general idea that the battery is

Table 4.4: Design and overhead of the policies used in ESS, AV and RABBIT

| Policy | States | Transitions | Timed | Execution Time (us) | Overhead (%) |
|---|---|---|---|---|---|
| ESS | | | | | |
| *None* | N/A | N/A | N/A | 2.70 | 0 |
| $\varphi_{soc}$ | 1 | 7 | No | 2.836 | 4.9 |
| $\varphi_{charge}$ | 1 | 4 | No | 2.71 | 0.346 |
| $\varphi_{soc} + \varphi_{charge}$ | 1 | 11 | No | 2.84 | 4.935 |
| AV | | | | | |
| *None* | | | | 736 | 0 |
| $\varphi_{cnn}$ | 1 | 15 | No | 764 | 3.8 |
| $\varphi_{drive}$ | 1 | 4 | No | 740 | 0.54 |
| $\varphi_{car}$ | 1 | 7 | No | 774 | 5.1 |
| $\varphi_{ped}$ | 2 | 56 | Yes | 767 | 4.2 |
| $\varphi_{cnn} + \varphi_{drive} + \varphi_{car} + \varphi_{ped}$ | 2 | 99 | Yes | 803 | 9.1 |
| RABBIT | | | | | |
| *None* | | | | 90.1 | 0 |
| $\varphi_{bound}$ | 2 | 16 | No | 89.45 | 0.0 |
| $\varphi_{score}$ | 2 | 112 | Yes | 94 | 4.3 |
| $\varphi_{bound} + \varphi_{score}$ | 2 | 126 | Yes | 93.8 | 4.1 |

charged when power is cheap, and discharged when demand is high or power is expensive. The ESS uses a pre-trained, three-layer MLP, previously introduced in Chapter 3, to decide the action for the battery in the next tick. While the AV system is a hard real-time system the ESS system is a soft real-time system, as a missed deadline will not result in fatalities. Failures in this system could cause damage to electrical components and potentially cause fires (from overcharging or overcurrent) and maybe even customer property, i.e. the charging station and the EVs, and thus appropriate policies can be put into place to make the system safer. RE can be used to formally guarantee that the controller will behave safely at all times. Systems with ANNs are difficult to verify. RE can ensure that they behave safely even in the event of unexpected inputs. The enforced policies monitor the controller's MLP outputs and ensure that (1) the battery levels never reach critical levels, and (2) that too much power is never given or taken from the battery in too short a period of time. Running this system with enforced policies shows that the battery's SoC never exceeds critical values and that the battery never charges or discharges too much power. The charge of buying and selling electricity to the customer's EV does not change by any significant amount when the enforcer is in place, i.e. it does not affect the performance of the system while it keeps the system safe. The results of the system with and without the policies enforced are shown in Table 4.5, while the policies' structure and overhead are shown in Table 4.4.

Table 4.5: Comparison of ESS with and without enforced policies

| Enforcer used | No enforcer | Enforcer | Ideal |
|---|---|---|---|
| Times high charge/discharge power was detected | 52 | 0 | 0 |
| Lowest battery SoC | 0.21 | 0.25 | 0.25 |
| Minutes of critically low SoC | 87 | 0 | 0 |
| Highest battery SoC | 0.905 | 0.905 | 0.95 |
| Minutes of critically high SoC | 0 | 0 | 0 |
| Daily cost of buying power ($) | 335 | 336 | $< 380$ |

Table 4.5 shows the instances of dangerous or unsafe behaviour by the system over an entire run of the system. The ESS, without the enforcer in place, operated under safe levels for a total of 87 minutes in one day. This is a large amount of time for the battery to spend undercharged and can cause damage to the battery. However, with the enforcer, the ESS was as safe as an ideal system: no time was spent under- or over-charged. Additionally, the cost of buying power, accrued over one day, did not change much from the enforced to the unenforced system. The $1 difference is due to the floating-point to integer conversions taken when using the run-time enforcer.

### 4.4.2 Autonomous Vehicle (AV)

The ANNs in this system were trained for varying amounts of epochs, notably 1, 10, 100, 1,000, 10,000 and 100,000 epochs. A partially trained ANN will exhibit behaviour that is unexpected, and perhaps unsafe, because ANNs gain robustness through training, a less trained ANN will be less robust. This is also true for an over-trained ANN, these are too used to training in a certain environment and are not able to respond well to being used in an untrained environment. This partially trained system is used to show how RE can prevent unsafe events in a system that does not behave as expected.

The AV system's performance was measured by the amount of time the AV was able to run before an accident occurred within the system. At every level of training, the whole system was run 10,000 times, where one run consisted of running the system for 100 ticks, noting if and when an accident occurred and what caused the accident. The simulation for these runs worked by randomly generating objects (vehicles and people) in the sensor positions of the vehicle (see Figure 4.1). As a result, the AV had limited time to respond to each generated object, occasionally ending up in situations where an accident was unavoidable, such as being rear-ended by the vehicle behind when braking to avoid hitting a pedestrian ahead. The results show this, as the number of incidents per run is high, shown in Figure 4.10. This also affected the time the system was able to run before an accident occurred, since being rear-ended by

Table 4.6: Results of the AV system with and without the enforced policies

| Number of epochs trained | Runs resulting in accidents (%) | Average minutes to first accident per run (/100) | Average speed (km/h) | Unnecessary braking incidents per run (%) |
|---|---|---|---|---|
| Not enforced | | | | |
| 0 | 100 | 3.21 | 98 | 19 |
| 1 | 100 | 3.3 | 93 | 19 |
| 10 | 100 | 3.8 | 81 | 57 |
| 100 | 100 | 5.35 | 59 | 24 |
| 1000 | 100 | 5.31 | 58 | 27 |
| 10000 | 48.5 | 73.9 | 18 | 63 |
| 100000 | 91.5 | 39 | 31.5 | 65 |
| Enforced | | | | |
| 0 | 75.7 | 75.6 | 45 | 0.7 |
| 1 | 70.2 | 82.8 | 47 | 4.2 |
| 10 | 88.5 | 58.9 | 37 | 20 |
| 100 | 79.5 | 72.6 | 41 | 17 |
| 1000 | 80.5 | 71.0 | 40 | 18 |
| 10000 | 64 | 97.07 | 27.5 | 27 |
| 100000 | 57 | 97.14 | 29 | 28 |

another vehicle would end the run as soon as it happened. With the more trained ANNs, most of the incidents were due to a collision with a vehicle behind it.

Figure 4.11 compares the amount of training of the AV system's ANNs with the performance of the system at that amount of training, as measured by the average number of ticks the system ran for before an unsafe incident occurred. There is a direct relationship between the amount of training and the duration before an accident occurred, the difference between a less trained and more trained system is noticeable. However, the difference between a less trained and more trained system is negligible when the system has enforced policies; both were able to run for almost an entire run.

Over-training is a well known phenomenon that ANNs suffer. Over-training occurs when ANNs become too well trained on the data set they are trained with and, as a result, perform poorly on extrapolated data sets. Over-training was observed in the AV system. After approximately 10,000 epochs of training, the performance of the system began to drop. Due to the nature of the training algorithm and the data used, the system began to make more unsafe decisions in lieu of increasing the average speed of the vehicle. However, the results also showed that the enforced system behaved just as safely when over-trained as it did under-trained, or even perfectly trained. This has applicability in many autonomous systems, where the ideal amount of training for the system is unknown.

Fewer accidents were noted when the system used RE (see Figure 4.10), and the few acci-

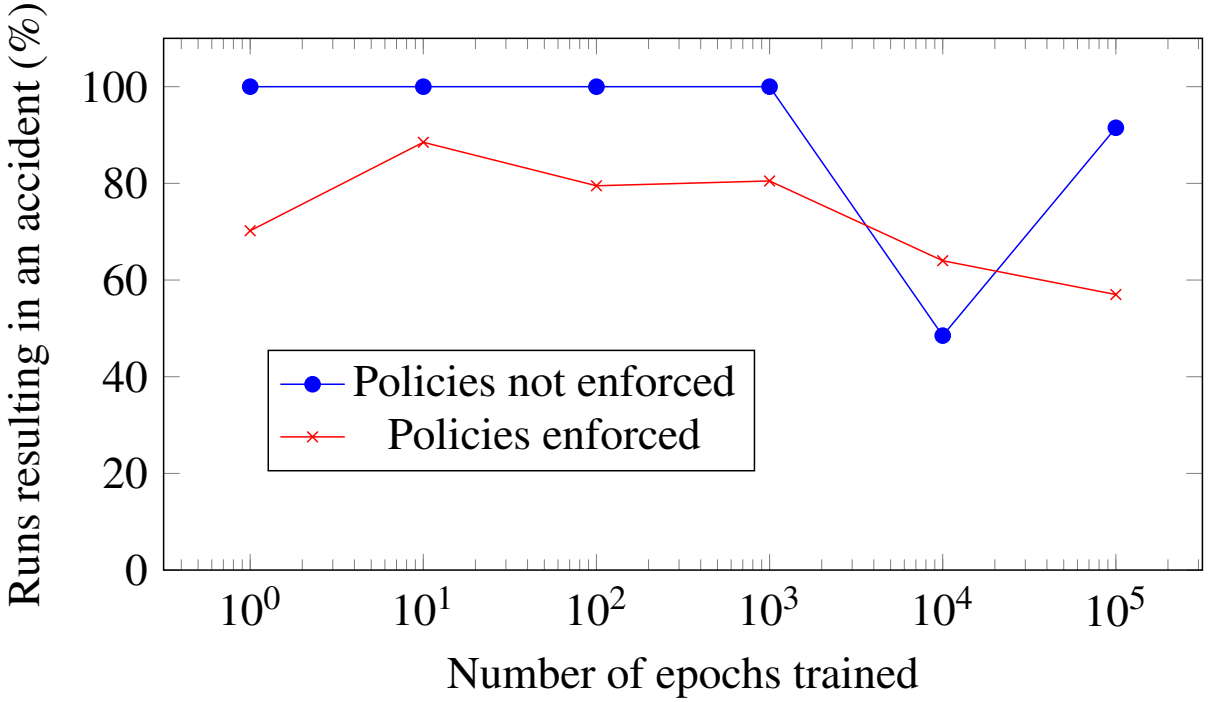



Figure 4.10: Graph showing the number of accidents of the AV system with and without enforced policies and at different stages of ANN training.

dents that did occur were the result of a vehicle hitting the AV from behind when it slammed on the brakes to avoid a collision in front of it. As seen in Table 4.6, the enforcer not only reduced the number of accidents, but also reduced the speed of the vehicle to safe levels. An additional measure of performance was the percentage of brake actions the vehicle took that were unnecessary, i.e. brake actions when there were no obstacles on the road or pedestrians in sight.

### 4.4.3 RABBIT

RABBIT is a simple game that is played between two teams: a team with one rabbit on it, and an opposing team with two wolves. This benchmark was introduced in Chapter 3. RABBIT has no safety implications to humans, as it is a game played between two teams of AI, but policies can be enforced to increase the efficiency of the animals and reduce the errors made by the animals. The policies ensure that the animals never leave the playing field, and that their movements are more refined, e.g. not getting stuck in one place, not moving away from their objective, etc. The results of the policies are shown in Table 4.7, and the structure and overhead of these policies are described in Table 4.4. It is shown that, without the enforcer in place, the "animals" in the system are unable to score many points, on either side. However, with the enforcer in place, they are able to score significantly more points as the enforcer prevents undesirable outputs, i.e. no points being scored.

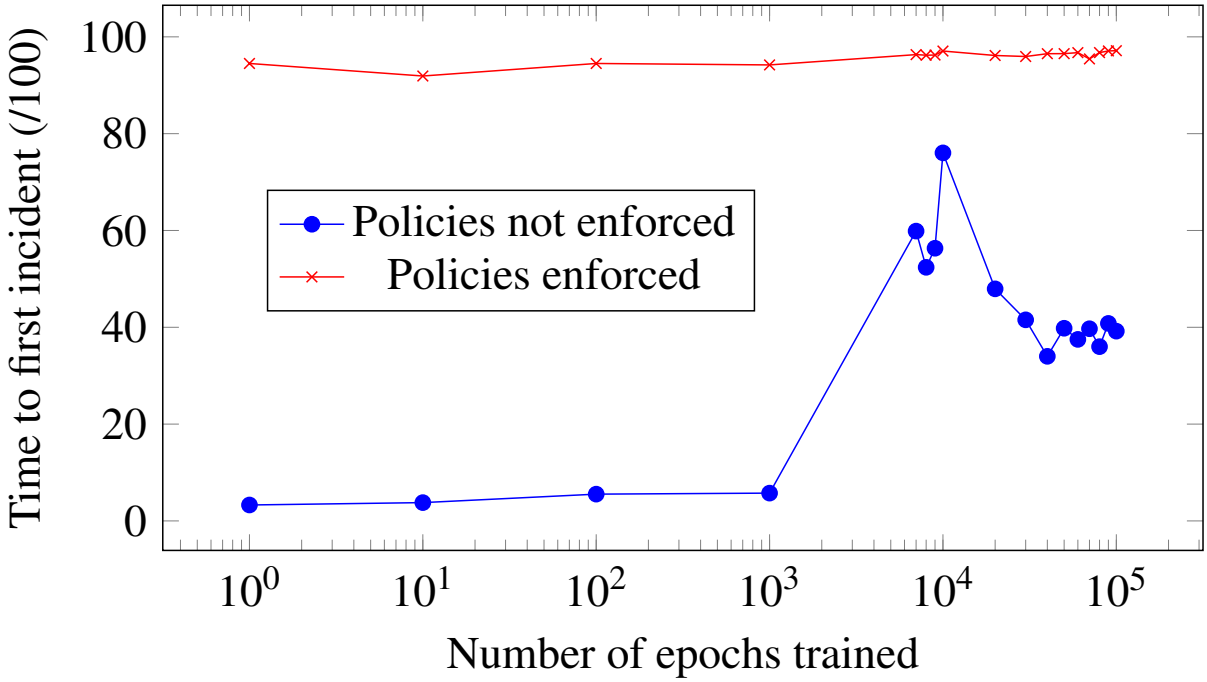



Figure 4.11: Graph showing the performance of the AV system with and without enforced policies and at different stages of ANN training.

Table 4.7: Comparison of RABBIT system with and without enforced policies

| Enforced | Rabbit wins (%) | Wolves wins (%) | Neither animal wins (%) |
|---|---|---|---|
| No | 3 | 17 | 80 |
| Yes | 36 | 59 | 5 |

## 4.5 Discussion

The ESS (Table 4.5) showed that the enforcer increased safety by preventing dangerously low State of Charge (SoC). However, with RABBIT game (Table 4.7) the enforcer increased the efficiency of the players in the game, as the game had no safety requirements.

The AV system, when using RE to enforce some safety policies, showed a large increase in safety. Without the enforcer, at best the system was able to run for 73 ticks before an accident occurred, according to Table 4.6. At lower epochs of training, the system was able to run for at most 3 ticks before an accident occurred. However, when the enforcer was implemented, the system was able to run for up to 97 ticks before an accident occurred. At its worst trained epoch, the system was still able to run for 58 ticks. It is worth noting that not all accidents are avoidable, for instance the AV can be hit by a vehicle from behind while it is slowing down to prevent an accident with a pedestrian in front. These types of unavoidable accidents make up the majority of the total accidents for the more trained MNNs.

Table 4.6 also shows that the number of the accidents over all runs in the system decreased

as the system was trained, even with the enforcer in place. An untrained SNN would have an accident on 75% of the runs, while the same SNN trained for 100,000 epochs would only have an accident 57% of the time. This shows that the enforcer, while increasing the safety of the system, does so proportionally to the level of training of the SNN.

While the enforcer allowed the vehicle to drive safely for longer periods of time, it did not decrease the number of incidents in the system proportionally to the time the vehicle stayed safe. Likewise, the enforcer did not result in the vehicle driving at a reasonable speed. With or without the enforcer, the vehicle braked more often and drove more slowly. We can conclude that the enforcer did efficiently increase the safety of the system by allowing the system to run for longer, however other aspects of the systems safety, like the total number of accidents, were dependant on the amount of the training the ANN had. This shows that the amount of training the SNN has is relevant and makes a difference to the overall safety of the system.

As the SNN was trained, it exhibited more careful behaviour; the vehicle's average speed decreased at and the vehicle performed more braking actions than necessary. Even with the enforcer in place, the speed decreased in proportion to the number of epochs trained, while the vehicle still was still braking unnecessarily. While the enforcer does help the ANN act more carefully, the a less trained ANN will still behave less carefully than a more trained ANN.

In this work, we have presented an approach for synchronous composition of Run-time Enforcement (RE) with Synchronous Neural Networks (SNNs) for CPSs. Our case study and benchmarks demonstrate that policies specifying safe I/O behaviours for ANNs can be enforced. These enforced policies are shown to increase the safety and efficiency of the systems within which they are placed, without adding a large overhead to the system and without negatively influencing the functionality of the system.

Being able to reactively verify ANN properties allows the use of ANN that are difficult, or impossible, to verify in CPS. This solves a lot of problems that the static verification of ANN has not yet solved. Gehr et al [21] introduce a very effective static verification method for the robustness of ANNs. However, this method does scale with the size of the ]acANN well, however it would still take longer periods of time to verify larger networks. Additionally, this method only works for ANNs using the ReLu activation function and does not handle more complex, non-linear activation functions. Finally, this method is not fool proof, not always being able to identify and prove all the specified robustness properties. Our reactive method of verifying ANNs using RE allows the reactive verification, finding and correcting faults that may have been missed during static analysis.

# 5

# Runtime Verification of Synchronous Neural Networks

## 5.1 Introduction

This chapter discusses the issue of adversarial perturbation [21] in Artificial Neural Networks (ANNs) and introduces a new method for dealing with these. We propose the use of reactive Run-time Verification (RV) [50] to functionally verify the I/O events of an ANN controller at run-time. Additionally, we use ANN ensembles [39] to passively increase the classification accuracy of an ANN controller.

## 5.2 A Object Detection Case Study for Perturbed Inputs

Research in Autonomous Vehicles (AVs) systems is rapidly expanding, with companies such as Uber and Tesla the biggest contributors to this field. With the growing use of AVs comes an increase in accidents related to these vehicles. The majority of these accidents have one thing in common: a misclassification occurred right before the accident. Input perturbations greatly decrease accuracy, thereby increasing the chance of a misclassification and, by proxy, the chance of an accident.

### 5.2.1 Object Misclassification in Autonomous Vehicle Systems

The inspiration for this case study was taken from the more recent Uber and Tesla AV accident, such as [16]. In these accidents, a misclassification, or series thereof, precedented the crash. In the case of the Uber accident [16] a pedestrian crossing the road (at night) was misclassified a minimum of three times before the accident occurred.

Theoretically, this accident could have been prevented in two ways: the first being that no misclassifications occurred and the second being that the system recognised the misclassifications sooner and acted accordingly. The misclassifications could have occurred due to adversarial perturbations of the input image due to the darkness. A system can be implemented to increase prediction accuracy regardless of the state of the inputs, perturbed or not. By implementing a runtime verifier and a Valued Discrete Timed Automata (VDTA) (discussed in Chapter 4) to define the safety policy, misclassifications can be detected quickly and effectively.

### 5.2.2 Runtime Verification of Convolutional Neural Networks (CNNs)

Runtime Enforcement, as a solution to the verification of ANNs and introduced in the previous chapter, only works when the inputs to the ANN are known. This works very well with feed forward ANNs, as their inputs are, generally, defined and known. Take the AI-BRO ANN discussed in Section 3.2; this ANN has a vector of defined inputs representing the objects “seen” by the vehicle. Since dangerous situations are defined by both inputs and outputs, the outputs of this ANN can be enforced because both the inputs and outputs are known at all times.

RE can only function with well-defined inputs and outputs. For instance, the ESS system is well-defined, because we can trust current and voltage measurements, and we know that a given model of battery has a minimum state of charge. We can enforce behaviour over a system relying on components such as these because constraints can be well defined.

However, systems that rely on ill-defined inputs, such as camera images, are not so straightforward. How should a rule that says "stop the car if you detect a pedestrian" be implemented? In the previous section, we used RE to enforce a controller MLP with well-defined inputs and output, but this isn’t directly applicable in a real-world situation. CNNs are used as image classifiers because we don’t have a guaranteed method for doing this analysis. An enforcer has no way of detecting if any one output from a CNN is wrong, as it has no baseline of truth to operate from nor has any way of knowing what to change a value to in the event of a violation.

Run-time Verification (RV) provides the formal methodology for detecting faults that Runtime Enforcement relies upon. By utilising this framework, rather than RE, we can instead define, detect, and flag "anomalous" or "unsafe" situations. For an AV system, this would correspond to the system then returning control to the human driver. The solution proposed is to use a run-time verifier to verify the outputs of a complex classifier MNN.

### 5.2.3 Sensor Fusion and Runtime Verification for an Autonomous Vehicle (AV) [75]

Sensor fusion is a technique for increasing the accuracy of object detection CNNs. Sensor fusion is the combination of two or more different sensor types to increase the overall sensor detection accuracy of the system. In Autonomous Vehicles (AVs) safety is of utmost importance, and we consider the issue of input perturbations in such systems.

The proposed approach to safe deep ANNs combines Run-time Verification (RV) [50] and sensor fusion to safely increase the detection accuracy of CNNs in AVs. The sensors in question are a 360° LiDAR sensor and three front-facing cameras. The sensor outputs are combined synchronously using a synchronous runtime verifier. The runtime verifier checks the integrity of the detected outputs, and in the case of a possible misclassification the system is put into a safe mode where control of the vehicle is returned to the driver. Using a Synchronous Neural Network (SNN) as the CNN in the AV system, the system is kept synchronous, time predictable and easy to formalise.

Additionally, to reduce the impact of perturbations to the CNN's inputs, a MNN composed of three MNN ensembles is used, each with three synchronous CNNs. This MNN aims to greatly increase the prediction accuracy by splitting the prediction process amongst multiple, different CNNs such that the weakness of each CNN is addressed by the other CNNs.

### 5.2.4 An Autonomous Vehicle (AV) Object Detection System

Using the Tesla and Uber incidents as inspiration, we construct an environment to emulate a crash situation where adversarial inputs could cause a CNN-controlled AV to misclassify a roadside obstacle and crash. As such, a simulation of an AV was made to try and capture the intricacies of such a system where safety is concerned, while showing the efficacy of the proposed techniques. Unlike the previous chapter, this AV system focuses only on the object classification module of an AV controller. This AV controller consisted of 3 MNN ensembles, similar to the ensembles used in Chapter 4. Each MNN ensemble aims to classify a different aspect of the input image: the first classifies the colour of the object; the second classifies the shape of the object; and the last classifies the type of object. These MNNs were trained and implemented using the VOC 2012 [19] and German Traffic Sign Recognition Benchmark (GTSRB) [64] datasets. Each ensemble aims to increase the overall classification accuracy and the run-time verifier provides an overall verdict of the object type being detected. Using these three different classifications, a run-time verifier can more accurately give a verdict on the type of object classified by the third MNN ensemble.

## 5.3 Simulating a Traffic Sign Recognition AV System

The system designed for this case study was made to reflect an Autonomous Vehicle (AV) and its object detection mechanisms. The system used multiple techniques to tackle the inherent issues of the AV system, i.e. weakness to perturbed inputs and misclassification of detected objects. The system's sensors included an overhead, 360° Light Detection and Ranging (LiDAR) apparatus, and a single set of front-facing cameras. Using only front-facing cameras was sufficient to demonstrate the efficacy of this solution, however it is to be noted that AV systems generally use cameras facing multiple, different directions, so that the controller can make properly informed decisions. The architecture of the system used can be seen in Figure 5.1.

### 5.3.1 Architecture of a Traffic Sign Recognition AV System

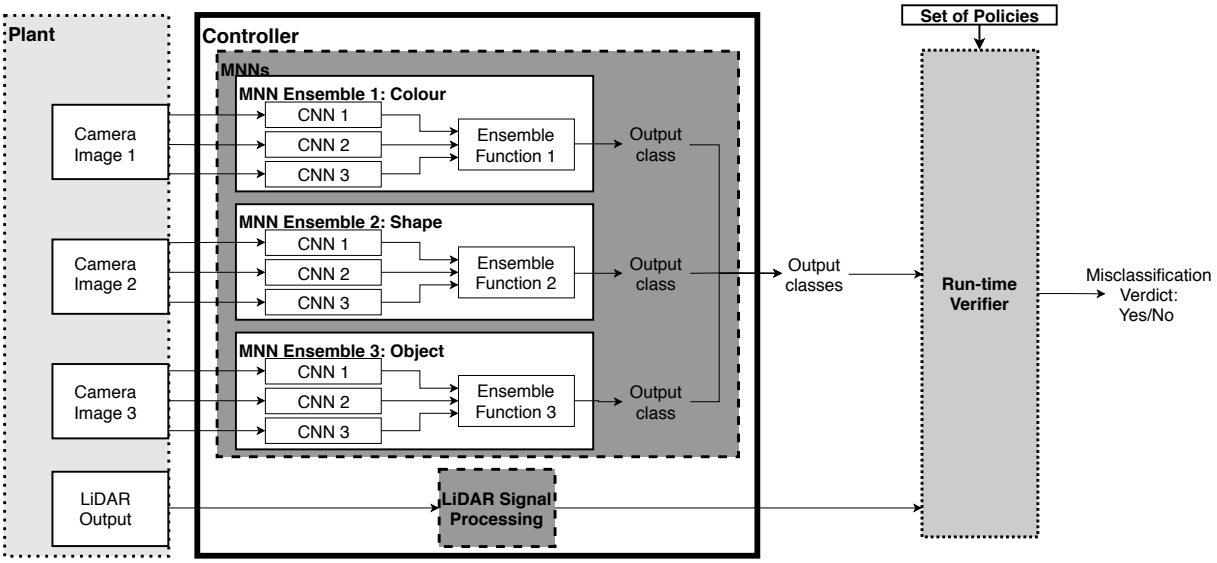



Figure 5.1: Block diagram showing the Meta Neural Network (MNN) ensembles and run-time verifier used in this case study

Utilising synchronous semantics [6], a Meta Neural Network (MNN), containing three other MNN ensembles, was created. This structure can be seen in Figure 5.1. A MNN is a synchronous composition of multiple Synchronous Neural Network (SNN). Each of the three MNN ensembles were used to classify shape, colour and object type of each object being output by the camera. Each ensemble synchronously combined the outputs of three different convolutional SNNs, providing increased prediction accuracy for shape, colour and object type. Each SNN was implemented in the synchronous language Esterel [8] using the Darknet library [53] to train and implement the CNN. After the introduction of Meta Neural Network to C (MNN2C), this system was trained in Python using Keras [14] and converted to time-predictable SNNs in C. These ensembles ran in synchronous concurrency with each other, forming a single, larger MNN.

The individual CNN outputs were passed to an ensemble function that combined the classifications of each CNN together, resulting in a more accurate overall classification. Each ensemble used custom averaging function that worked out the most likely class of the classified object using the top 2 most likely classes output by each CNN. The outputs of each ensemble were then combined into a batch of outputs forming the *predicted output* for the colour, shape and object type of the detected object.

### 5.3.2 Sensor Fusion for a Traffic Sign Recognition AV System

Two sensors were used for the purpose of sensor fusion; LiDAR and cameras. The LiDAR controller for this system used the 93% accuracy of the research group in [71], to closely simulate a system using real LiDAR. The simulated camera outputs consisted of test images from both the VOC 2012 [19] and GTSRB [64] datasets, in a combination of people, vehicles and various traffic signs. The LiDAR and camera outputs were handled by different parts of the controller. The camera outputs were fed into a MNN (see Figure 5.1) where they were classified by shape, colour and object type. Sensor fusion happened after classification occurred and was done using by the run-time verifier.

### 5.3.3 Run-time Verification (RV) [50] for a Traffic Sign Recognition AV System

The system controller was encapsulated by a run-time verifier [51] that used sensor fusion to check for misclassifications made by the MNN. A run-time verifier gives a verdict on every I/O sequence given to i by the controller. The RV can be seen in Figure 5.1, between the controller and the plant, monitoring the outputs of the MNN controller. The run-time verifier was given a Valued Discrete Timed Automata (VDTA), introduced in Chapter 4, as the safety policy to be used. This VDTA is shown in Figure 5.3. The syntax for this figure is explained in Chapter 2.

This policy uses a complex algorithm to determine if an object has been misclassified or not. For simplicity, this algorithm has been transferred to a flow-chart (Figure 5.2), rather than being shown on the VDTA.

The automaton started in a safe state, where control of the vehicle was autonomously handled by the system controller. If a misclassification was detected, the verified policy entered an unstable state, still under autonomous control. Once enough time passed without further misclassifications, the vehicle entered the safe state again. However, if another misclassification was detected while unstable, the verified policy entered a violation state and control of the AV was passed to the driver by the system. The vehicle would not enter autonomous mode again until the system was restarted.

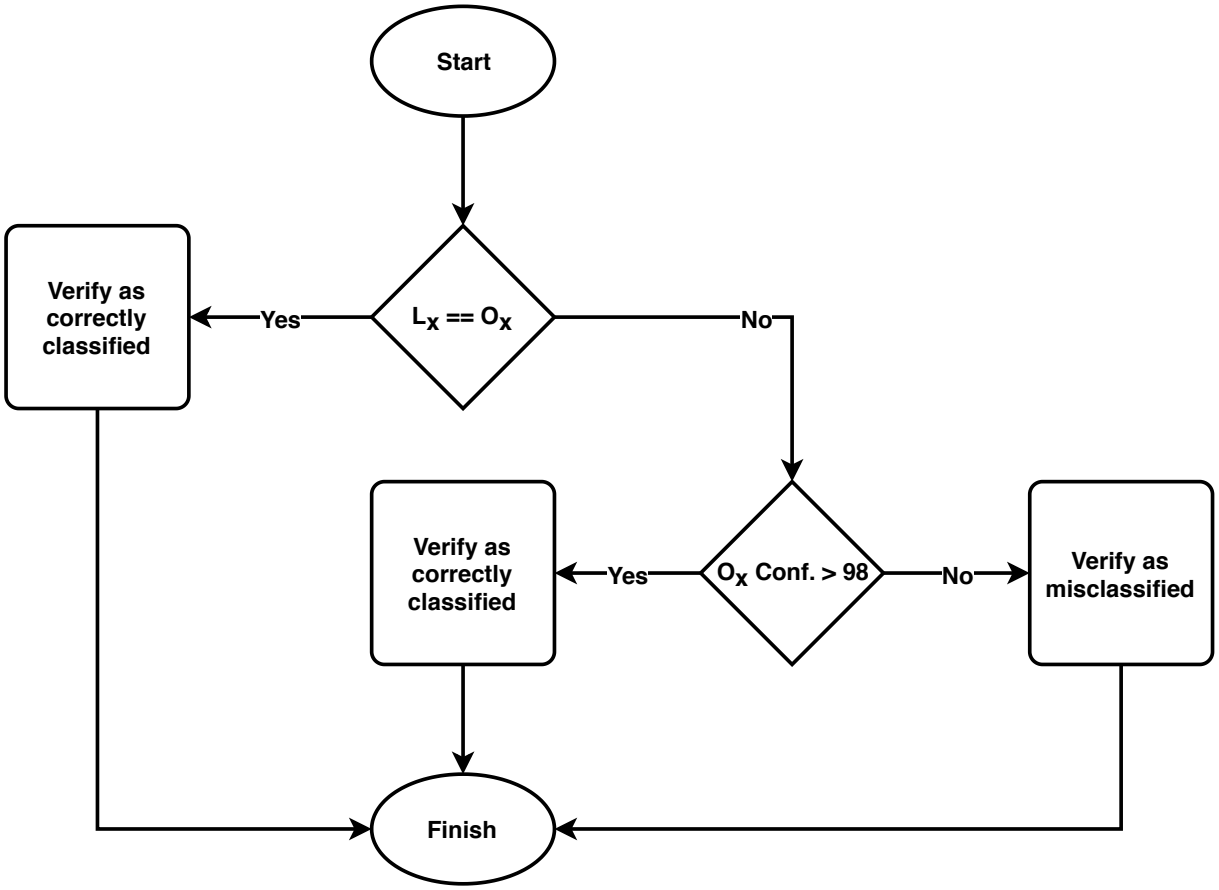
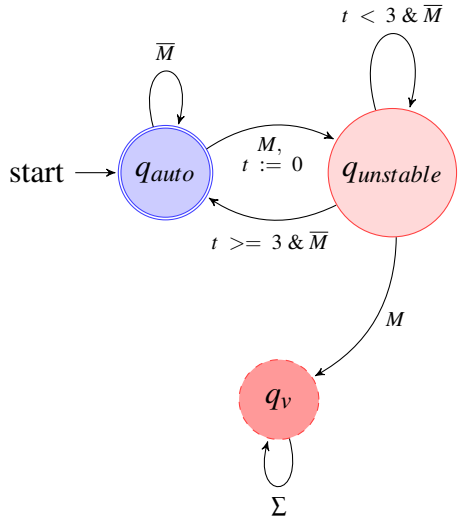



Figure 5.2: Flow-chart simplifying the algorithm used to detect misclassifications made by the MNN controller.

## 5.4 Results of the Runtime Verified AV System

This research provides a solution for two aspects of Autonomous Vehicle (AV) systems: predicting accurately with perturbations to the system's inputs and safely dealing with misclassifications by the system. The issue of input perturbations was addressed using a Meta Neural Network (MNN) of different convolutional Synchronous Neural Networks (SNNs), each SNN working in tandem to predict more accurately. Misclassification by the system's controller was addressed by implementing sensor fusion between cameras and LiDAR. This was done using a run-time verifier that verified a safety automaton. The benchmark was written in Esterel and C and run on Ubuntu 16.04, using an 4 core Intel i7-6700HQ processor at 2.6GHz and 4GB of RAM.

A graph was generated to show the affect of adversarial input perturbation, first presented as Figure 2.4 in Chapter 2. It shows that the input perturbations decreased the accuracy of the classifiers by as much as 50%. However, the aim of this approach is not only to increase the classification accuracy of the MNNs but rather catch misclassifications made by the MNNs, i.e. verify that the classifications made by the MNNs are valid. As such the results displayed do not show an increase in accuracy, but rather that the verifier is fully capable of catching




- $M$: Binary misclassification verdict: no misclassification (0) or misclassification detected (1).
- $t$: Timer for the unstable state.

Figure 5.3: Enforcer policy for the Autonomous Vehicle (AV) prediction system

misclassifications made by the MNNs.

To test the MNN's ability to deal with perturbations and misclassifications, the input images (taken from the VOC 2012 [19] and GTSRB [64] datasets) were perturbed by randomly replacing approximately 7% of the image pixels with randomly coloured pixels. The system was then run for over 10,000 ticks. Each tick the classifier was presented an image to classify. The results of the overall classification was noted: did the MNN controller make a misclassification and if so, was it caught by the run-time verifier.

### 5.4.1 Results of the MNN ensembles

To test the efficacy of MNN ensembles, three SNNs were trained on the data set. These ANNs were then tested, individually, on both the original images and the same perturbed images and the results were noted in Table 5.1. The overall average accuracy of three SNNs was also noted. An ensemble made up of the three SNNs was then run on the exact same images and its classification accuracy noted.

Table 5.1 shows the impact of MNN ensembles on the classification accuracy of images. Without being in an ensemble, the overall classification accuracy of the ANNs was 88.64% for the original images and 60.84% for the perturbed images. However, this average was increased to 91.17% and 62.92% for the original and perturbed images respectively. Additionally, the ensemble performed significantly better than the "worst" SNN.

While the MNN ensemble did not perform better than the "best" SNN, it did not perform

Table 5.1: Table showing the results of the MNN ensemble

| ANN | Classification accuracy (%) |
|---|---|
| Original Inputs | |
| SNN 1 | 84.24 |
| SNN 2 | 92.05 |
| SNN 3 | 89.62 |
| SNN Average | 88.64 |
| MNN Ensemble | 91.17 |
| Perturbed Inputs | |
| SNN 1 | 56.52 |
| SNN 2 | 63.16 |
| SNN 3 | 62.85 |
| SNN Average | 60.84 |
| MNN Ensemble | 62.91 |

significantly worse either. This does not mean that the ensembles are useless. When an ANN is trained, it is not know before deployment whether that ANN is a "worst" or a "best". And while the ensemble does not increase the effectiveness of a "best" ANN, it significantly increases the accuracy of a "worst" ANN. When ensuring timing and, in this case, functional safety of a system, the worst case must always be considered, and these ensembles will turn a worst case ANN into a best case ANN.

### 5.4.2 Results of a Darknet [53] implemented AV system

The MNN used in this system were trained for 10,000 epochs using the Darknet library [53]. Table 5.2 shows the results of the Darknet generated classifier. The first column shows the total number of misclassifications made, normalized to 100. The second column shows the total number of misclassifications caught by the verifier, also normalized to 100. The final column shows the relative percentage of the total misclassifications caught by the verifier.

Using the original inputs the verifier caught more than 65% of the total misclassifications. More than half of all the misclassifications made were detected by the verifier, and the safety of the system turned over to the driver. This same table shows that when the inputs are perturbed, the verifier picked up more misclassifications than with the original images, catching more than 76% of all misclassifications. These results are summarised in Figure 5.4.

Table 5.2: Table showing the results of the AV prediction MNN

| Epochs trained | No. of misclassifications (/100) | No. of caught misclassifications (/100) | % of total misclassifications caught |
|---|---|---|---|
| Original Inputs | | | |
| 0 | 95.16 | 95.16 | 100 |
| 10 | 95.16 | 95.16 | 100 |
| 100 | 82.67 | 61.09 | 73.90 |
| 1000 | 29.36 | 21.39 | 72.85 |
| 10000 | 12.38 | 8.55 | 69.06 |
| 100000 | 11.98 | 7.79 | 65.03 |
| 6000 (best) | 10.59 | 7.32 | 69.12 |
| Perturbed Inputs | | | |
| 0 | 95.16 | 95.16 | 100 |
| 10 | 95.16 | 95.16 | 100 |
| 100 | 93.63 | 71.89 | 76.78 |
| 1000 | 76.69 | 63.71 | 83.07 |
| 10000 | 57.89 | 45.89 | 79.27 |
| 100000 | 58.03 | 45.72 | 78.79 |
| 7000 (best) | 60.42 | 49.13 | 81.31 |

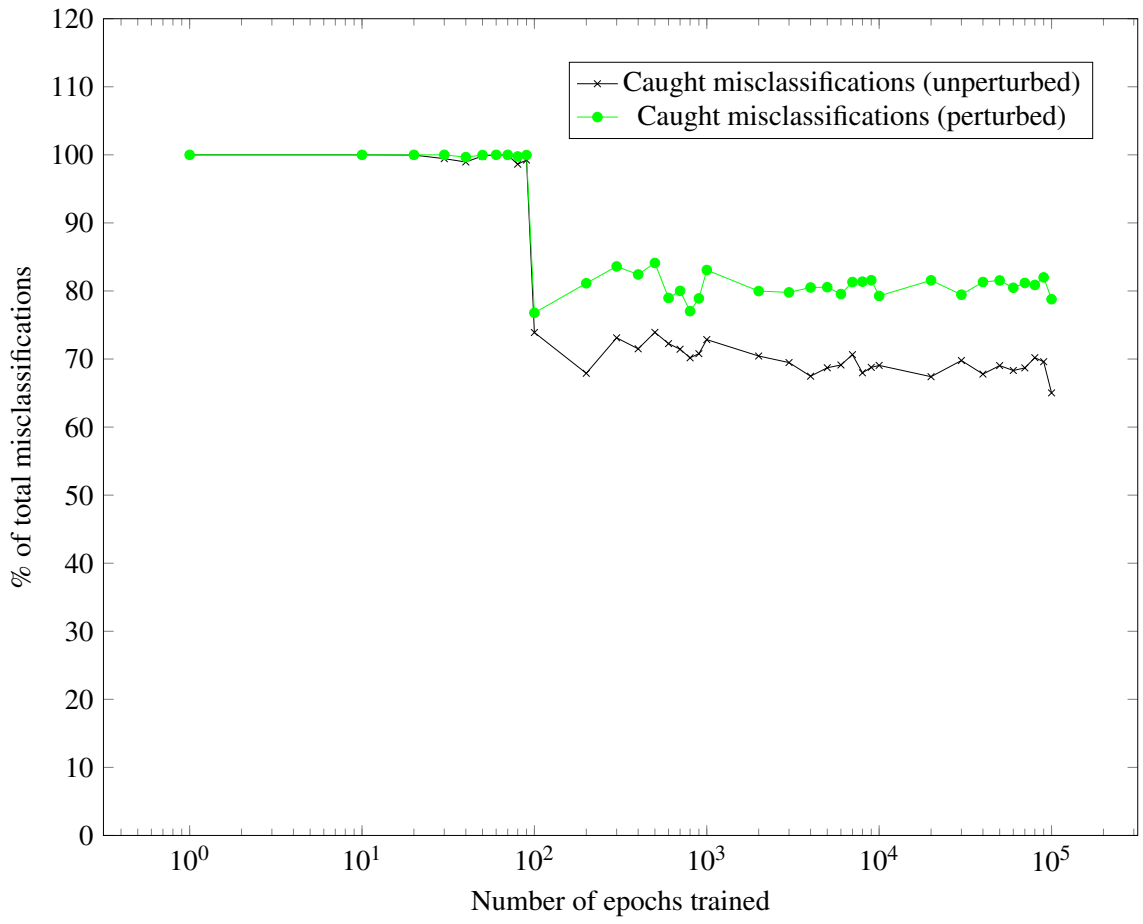



Figure 5.4: Line graph showing the number of misclassifications caught by the verifier for the Darkent MNN classifier.

Where input perturbations are concerned, the verifier responded even better and picked up the majority of the total misclassifications. The verifier caught more misclassifications in untrained ANNs due to the untrained ANNs having no confidence about their output. However, with more trained ANNs the verifier caught a lower percentage of the total misclassifications due to the ANNs having a higher confidence that they were correct, even when they were not.

### 5.4.3 An AV System Using Meta Neural Network to C (MNN2C)

MNN2C, introduced in Section 3.7, creates time-predictable, modular Meta Neural Networks (MNNs) for C from existing Keras (with Tensorflow) trained ANNs. This compiler makes implementing Meta Neural Networks (MNNs) in C easy and safe. For the purposes of testing and demonstration, the complex MNN used in this chapter, shown in Figure 5.1, was trained in Python, using Keras and the exact same images used to train the original, Darknet system. This MNN was then described in the MNN2C format, and modular C code was generated to initialise, run and incorporate the MNN. To show the efficacy of MNN2C, the generated MNN was implemented and tested in an identical system to the original. MNN2C generates outputs identical to the Keras trained ANNs with a one hundred-thousandth tolerance, so the output of each individual SNN is not being tested here, rather that the system as a whole runs as the original does. As with the Darknet trained system, the MNN2C system was implemented in Esterel and C and run on Ubuntu 16.04, using an 4 core Intel i7-6700HQ processor at 2.6GHz and 4GB of RAM.

#### Results of a MNN2C generated AV system

The MNN2C system was trained in Python using Keras [14] on Windows 10, using a 4 core Intel i7-6700HQ processor at 2.6GHz and 16GB of RAM. As a result, Keras was able to use more resources more efficiently than the Darknet library was and, as a result, was able to train ANNs more efficiently and more quickly than Darknet. Since these MNNs were trained in Keras, they did not need to go through same, extensive training to get to a reasonable level of classification accuracy, only needing 100 epochs, as opposed to the 10,000 epochs of training the Darknet MNNs were trained for.

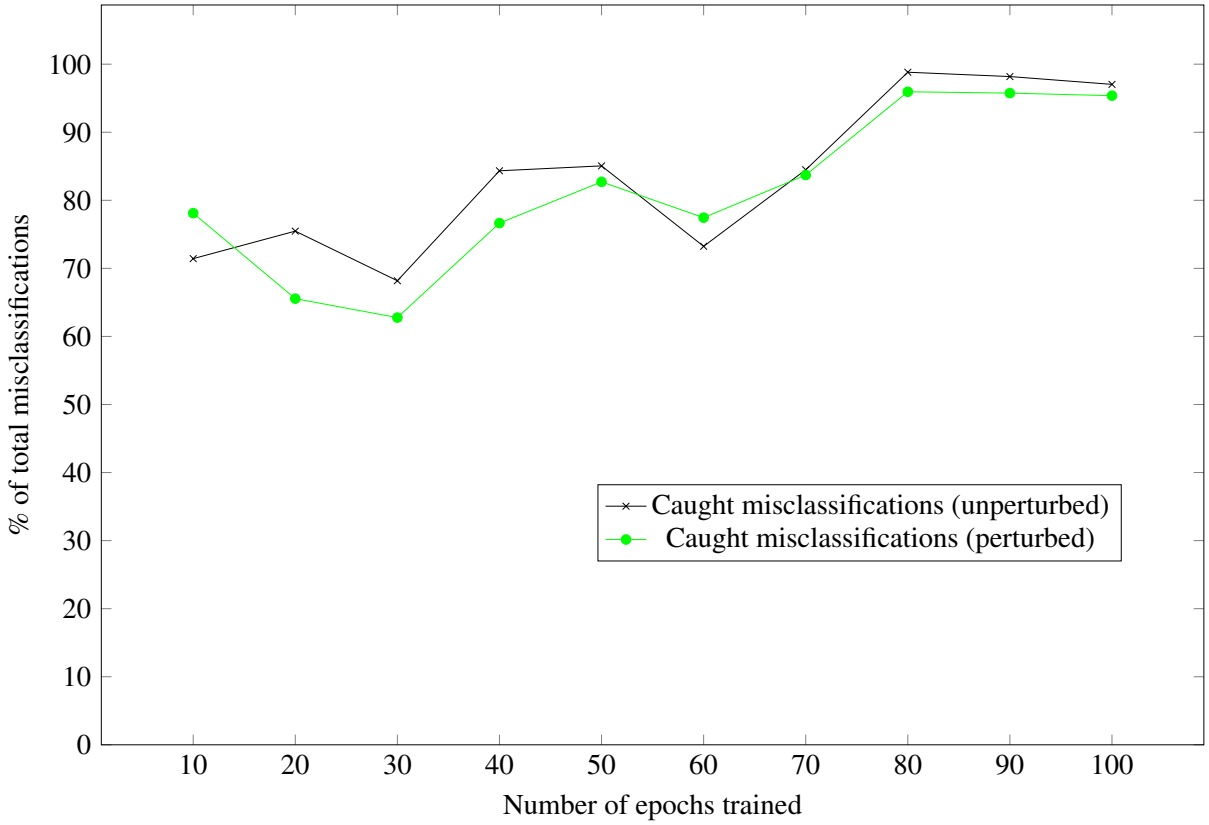



Figure 5.5: Line graph showing the number of misclassifications caught by the verifier with MNN2C generated MNNs

Figure 5.5 shows that the verifier stills work efficiently with Keras trained MNNs compiled to C code. With the original images, the verifier caught more than 70% of all misclassifications caught more than 60% of all misclassifications when the inputs were perturbed. As the MNNs are more trained, the number of caught misclassifications increases. This shows that the system works better with more extensively trained MNNs. Unlike the Darknet MNNs, the verifier was not able to catch 100% of all the misclassifications for untrained MNNs. The MNN2C SNNs used an identical system to the Darknet SNNs, however they were not trained under identical circumstances, nor were they trained for the same number of epochs. Thus, by adjusting the system's tolerances to suit the MNN2C SNNs, the system using could likely produce identical results to that of the Darknet SNNs. ANNs trained in Keras are trained slightly differently and, as a result, the confidence of the classifier was different to that of the Darknet classifier. However, the majority of the misclassifications were still caught, regardless of the number of epochs the classifier was trained for.

## 5.5 Discussion

Deep Artificial Neural Networks (ANNs) [59], ANNs with large, complex inputs (such as images) and a large number of layers are difficult to statically verify due to their complex na-

ture [21]. Lots of work has been put into the verification of ANNs, but the results yielded from this work have many limitations and are generally time consuming.

Previously, this thesis introduced the concept of using runtime enforcement to dynamically enforce the safety properties of an ANN as it runs. While this works very well for simple, feed forward ANNs, this does not extend to more complicated CNNs. The output of a CNN cannot be enforced as the input images are not recognisable by a simple algorithm.

This chapter presents techniques to increase the safety of AV systems. We create an Autonomous Vehicle (AV) case study that focuses on the object detection of the AV. To verify the safety of the system, we use MNN ensembles, consisting of multiple CNNs, which work together to increase the prediction accuracy and runtime verification to verify the integrity of the MNN's classifications, attempting to detect misclassifications dynamically and put the vehicle in safe state if one is detected. These two techniques work in tandem to greatly increase the safety of the AV system where object misclassification is concerned.

Creating a MNN with three synchronous CNNs, forming a MNN ensemble, did have a large effect on the classification accuracy of the system. The classification accuracy was higher than that of the average, ordinary CNN. Using MNN ensembles is more effective for poorly or incompletely trained CNNs, but does not offer an increase in classification accuracy when extensively trained CNNs are used. The ensembles do, however address the issue of worst case ANNs, by effectively classifying at the same accuracy as a best cast ANN would. Implementing three CNNs is more resource intensive; each has to be trained separately to a suitable standard and on different data, each has to be implemented separately and each has to run separately. This takes more time, memory and processing power than implementing a single CNN to classify the input images. The MNN ensembles in this case study used three times the resources that a single CNN would use, but it can be argued that this trade-off is worth the increase in classification accuracy where human lives are concerned, as the ensemble tend to the worst case scenario. The ensemble function used for these tests was a very basic function that went little beyond simple averaging. With a better ensemble function for these MNN ensembles, a more significant increase in accuracy may be seen.

The runtime verifier increased the prediction accuracy by 43% for an extensively trained MNN when input perturbations are involved, but only improved by 6% when there were no input perturbations. Regardless of the amount of training the MNN ensembles had, the runtime verifier was still able to detect a large number of misclassifications. With no training, the classification accuracy of the system matched the accuracy of the LiDAR sensor. As the system trained, and the MNN became more confident with its decisions, the accuracy of the system began to match the accuracy of the MNN.

This system showed that the functional verification of ANNs for CPSs was possible using the reactive RV. As opposed to the static verification methods such as [21] [26], this technique does not need to view the internals of the ANNs, but rather regards the ANNs as a black box.

This allows easier, scalable verification of ANN systems. Additionally, ensembles were demonstrated to be a method to cater for the worst case trained ANN using synchronous compositions of SNN. With more work on an averaging function, these ensembles could increase prediction accuracy even for the best case trained ANN.

# 6
# Conclusions

In this thesis, we proposed the use of synchronous programming towards the safe use of Artificial Neural Networks (ANNs) in Cyber-Physical Systemss (CPSs), which are safety-critical systems. Using the synchronous programming language Esterel, Synchronous Neural Networks (SNNs), were developed to create ANNs that maintained synchronous semantics. We were able to create fully synchronous, predictable ANNs that were able to be used as the controllers in all our benchmarks. The synchronous approach to ANNs, which includes formal definitions of the ANNs, simulations and implementations of CPS, was demonstrated throughout this thesis. The results and concerns of this approach are discussed.

Two different structure types were defined: SNNs and MNNs. SNNs are ANNs that run in a synchronous manner and maintain synchronous semantics, while MNNs are compositions of multiple SNNs, also retaining synchronous semantics. Using the time predictable T-CREST platform, a set of benchmarks were developed using Esterel and C to prove the efficacy of SNNs and MNNs. These benchmarks ranged from basic, Multi-layer Perceptrons (MLPs) to larger, more complex Meta Neural Networks (MNNs). Static timing analysis was done on these benchmarks, proving that the implemented SNNs and MNNs were not only time predictable, but also that the system were able to meet real-time deadlines. Additionally, a Python tool chain, termed Meta Neural Network to C (MNN2C) was introduced. This compiler was able to produce time-predictable, MNNs from existing Keras trained ANNs. This tool chain produced more accurate, lower Worst Case Reaction Time (WCRT) measurements compared to the original MNNs.

The next body of work introduced the concept of combining Run-time Enforcement (RE) with MNNs. To this end, the definition of a MNN was expanded to include arbitrary synchronous components, such as run-time enforcer, and not SNNs. These new MNNs were termed Synchronous Neural Networks (SNNs). An AV case study was used to demonstrate the efficacy of run-time enforced SNNs by enforcing a set of safety policies to ensure the safe behaviour of an AV when presented with dangerous situations. This approach was shown to vastly increase the safety of an AV, allowing the AV to run autonomously for long periods of time. However, a complication arose with the realisation that the I/O of an ANN with complex inputs, such as a Convolutional Neural Network (CNN), cannot be enforced.

The issue of input perturbation, where the inputs to the system are altered beyond the control of the system, was addressed next. We proposed the use of Run-time Verification (RV), sensor fusion and SNNs to decrease the risk posed by input perturbation. To show the effect of input perturbation, and test our approach to addressing it, a different AV system was created. This system was a simulation of the object classifier in an AV, i.e. the part of the controller responsible for identifying the AV's surrounding environment. Perturbed inputs were shown to decrease the classification accuracy of the system by up to 50%, however our solution showed that RV was able to "catch" more than 70% of all misclassification, and when the inputs perturbation occurred the RV was able to "catch" more than 80% of the misclassifications. Using the MNN2C tool, this approach was tested on Keras generated SNNs. These SNNs produced the

same results, showing that more than 70% of all misclassifications were detected by the RV.

## 6.1 Future Work

While synchronous, time predictable ANNs proved to be of great value, they were only implemented and tested using C on single core processors. The multi-core implementation and timing was not covered in thesis, but was researched with the construction of MNN2C. More research is required to the efficient multi-core implementations of SNNs for embedded systems. Additionally, the hardware implementation of SNNs was not researched for this thesis, but is also a key bit of research. SNNs have the potential to be implemented on a whole range of hardware, from Field-Programmable Gate Arrays (FPGAs) to graphics cards. This needs to be carefully researched.

This thesis suggested RE as an approach to creating safe ANNs, however only a single run-time enforcer was used in each benchmark. While this was shown to be largely effective, this is not the limit to which RE can be used in these SNNs. In this thesis, we did not fully examine the enforcement of multiple networks in a SNN, nor did we fully investigate all the possible compositions and combinations of RE and MNNs. Examining more compositions of SNNs with their run-time enforcers could lead to more capable systems that are still safe, or safer. Approaches such as using RE during the training of SNNs has yet to be explored, and RE between layers of individual SNNs has also yet to be explored.

The ensembles used in Chapters 4 and 5 only used basic averaging functions to increase the accuracy of worst case trained ANNs. With a more advanced averaging function, these ensembles could increase the classification accuracy of even the best trained ANNs.

Only the basics of RV was discussed in this thesis. RV has a lot more value than just being used for CNNs in AV systems. This approach can be taken on any system where the inputs to the system are complex, such a intrusion detection system on networks. More benchmarks should be developed that expand on a lot more CPS that could benefit from RV.

Lastly, the composition of SNNs was not fully researched in this thesis. The training phase of SNNs was skipped, instead each ANN was trained individually. SNNs are a unique composition of SNNs and other synchronous components, and the training of such systems needs further investigation. Additionally, the only structure of SNN used in this thesis was a SNN ensemble, where each SNN in the ensemble works with the others in the ensemble to provide more accurate output. There exist endless ways that SNNs can be combined, and more of these ways should be implemented and tested.

# References


[1] 127071. (2014) industry-525119. [Online]. Available: https://pixabay.com/en/industry-industrial-plant-525119/

[2] R. Alur, *Principles of cyber-physical systems*. MIT Press, 2015.

[3] A. Aniculaesei, D. Arnsberger, F. Howar, and A. Rausch, "Towards the verification of safety-critical autonomous systems in dynamic environments," in *Proceedings of the The First Workshop on Verification and Validation of Cyber-Physical Systems, V2CPS@IFM 2016, Reykjavík, Iceland, June 4-5, 2016.*, ser. EPTCS, M. Kargahi and A. Trivedi, Eds., vol. 232, 2016, pp. 79–90. [Online]. Available: https://doi.org/10.4204/EPTCS.232.10

[4] D. F. Bedford, G. Morgan, and J. Austin, "Requirements for a standard certifying the use of artificial neural networks in safety critical applications," in *Proceedings of the international conference on artificial neural networks*, 1996.

[5] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, Sep. 1991.

[6] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[7] G. Berry, "The foundations of esterel." in *Proof, language, and interaction*, 2000, pp. 425–454.

[8] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, Nov. 1992.

[9] C. M. Bishop, "Novelty detection and neural network validation," *IEE Proceedings - Vision, Image and Signal Processing*, vol. 141, no. 4, pp. 217–222, Aug 1994.

[10] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end

learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[11] D. Bullock, J. C. Fiala, and S. Grossberg, "A neural model of timed response learning in the cerebellum," *Neural Networks*, vol. 7, no. 6-7, pp. 1101–1114, 1994.

[12] K. Chaudhari, A. Ukil, K. N. Kumar, U. Manandhar, and S. K. Kollimalla, "Hybrid optimization for economic deployment of ESS in PV-integrated EV charging stations," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 106–116, Jan 2018.

[13] C. H. Cheng, F. Diehl, G. Hinz, Y. Hamza, G. Nuehrenberg, M. Rickert, H. Ruess, and M. Truong-Le, "Neural networks for safety-critical applications 2014; challenges, experiments and perspectives," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1005–1006.

[14] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[15] M. Clark, X. Koutsoukos, J. Porter, R. Kumar, G. Pappas, O. Sokolsky, I. Lee, and L. Pike, "A study on run time assurance for complex cyber physical systems," Air Force Research Laboratory, Vanderbilt University, Iowa State University, University of Pennsylvania, Galois Inc., Tech. Rep., 2013.

[16] D. Coldewey, "Uber in fatal crash detected pedestrian but had emergency braking disabled," *TechCrunch*, May 2018. [Online]. Available: https://techcrunch.com/2018/05/24/uber-in-fatal-crash-detected-pedestrian-but-had-emergency-braking-disabled/

[17] D. de Niz, B. Andersson, and G. Moreno, "Safety enforcement for the verification of autonomous systems," in *Proc. (SPIE) Vol. 10643: Autonomous Systems: Sensors, Vehicles, Security, and the Internet of Everything*, 05 2018, p. 2.

[18] DrSJS. (2014) girl-320262. [Online]. Available: https://pixabay.com/en/girl-woman-face-eyes-close-up-320262/

[19] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results," http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[20] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Embedded Software: First International Workshop, EMSOFT 2001 Tahoe City, CA, USA, October 8–10, 2001 Proceedings*, 2001, pp. 469–485.

[21] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. T. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 3–18, 2018.

[22] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner, "The platin tool kit - the T-CREST approach for compiler and WCET integration," in *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[23] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 3–29.

[24] G. Hurlburt, "How much to trust artificial intelligence?" *IT Professional*, vol. 19, no. 4, pp. 7–11, 2017.

[25] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, Nov 2003.

[26] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 97–117.

[27] T. Kohonen, "An introduction to neural computing," *Neural networks*, vol. 1, no. 1, pp. 3–16, 1988.

[28] Z. Kurd, "Artificial neural networks in safety-critical applications," PhD dissertation, University of York, 2002.

[29] Z. Kurd and T. Kelly, "Establishing safety criteria for artificial neural networks," in *Knowledge-Based Intelligent Information and Engineering Systems*, V. Palade, R. J. Howlett, and L. Jain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 163–169.

[30] Z. Kurd, T. Kelly, and J. Austin, "Developing artificial neural networks for safety critical systems," *Neural Computing and Applications*, vol. 16, no. 1, pp. 11–19, Jan 2007. [Online]. Available: https://doi.org/10.1007/s00521-006-0039-9

[31] Z. Kurd and T. P. Kelly, "Safety lifecycle for developing safety critical artificial neural networks," in *Computer Safety, Reliability, and Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 77–91.

[32] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, May 2008, pp. 363–369.

[33] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, May 2008, pp. 363–369.

[34] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, no. 1, pp. 2–16, Feb 2005. [Online]. Available: https://doi.org/10.1007/s10207-004-0046-8

[35] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results," in *Computer Security – ESORICS 2010*, D. Gritzalis, B. Preneel, and M. Theoharidou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 87–100.

[36] F. Liu and M. Yang, "Verification and validation of ai simulation systems," in *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.04EX826)*, vol. 5, Aug 2004, pp. 3100–3105 vol.5.

[37] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, "A survey of WCET analysis of real-time operating systems," in *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, May 2009, pp. 65–72.

[38] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659 – 1671, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608097000117

[39] I. Maqsood, M. R. Khan, and A. Abraham, "An ensemble of neural networks for weather forecasting," *Neural Computing & Applications*, vol. 13, no. 2, pp. 112–122, Jun 2004. [Online]. Available: https://doi.org/10.1007/s00521-004-0413-4

[40] L. Medsker and L. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, 2001.

[41] T. Menzies and C. Pecheur, "Verification and validation and artificial intelligence," *Advances in computers*, vol. 65, pp. 153–201, 2005.

[42] MichaelGaida. (2016) architecture-1639990. [Online]. Available: https://pixabay.com/en/architecture-industry-1639990/

[43] K. T. Monadjem, H. Pearce, P. S. Roop, and S. Pinisetty, "Dealing with adversarial perturbation in neural networks using synchronous neural networks combined with run-time

verification," *IEEE Trans. Neural Netw. and Learning Sys.*, 2019, unpublished; in preparation.

[44] J. Moore, J. Desmond, and N. Berthier, "Adaptively timed conditioned responses and the cerebellum: a neural network approach," *Biological cybernetics*, vol. 62, no. 1, pp. 17–28, 1989.

[45] T. Mudau and D. Coulter, "Ann-mind: A comparative study on the training of neural networks with incomplete datasets," in *2018 IST-Africa Week Conference (IST-Africa)*, May 2018, pp. Page 1 of 8–Page 8 of 8.

[46] K. Okano and T. Sekizawa, "Safety verification of multiple autonomous systems by formal approach," in *Computer Safety, Reliability, and Security*, A. Bondavalli, A. Ceccarelli, and F. Ortmeier, Eds. Cham: Springer International Publishing, 2014, pp. 11–18.

[47] O. A. Olanrewaju and C. Mbohwa, "Comparison of artificial intelligence techniques for energy consumption estimation," in *2016 IEEE Electrical Power and Energy Conference (EPEC)*, Oct 2016, pp. 1–5.

[48] D. E. O'Leary, "Artificial intelligence and big data," *IEEE Intelligent Systems*, vol. 28, no. 2, pp. 96–99, March 2013.

[49] N. G. Paterakis, E. Mocanu, M. Gibescu, B. Stappers, and W. van Alst, "Deep learning versus traditional machine learning methods for aggregated energy demand prediction," in *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, Sept 2017, pp. 1–6.

[50] S. Pinisetty, P. Roop, V. Sawant, and G. Schneider, "Security of pacemakers using runtime verification," 10 2018, pp. 1–11.

[51] S. Pinisetty, P. S. Roop, S. Smyth, N. Allen, S. Tripakis, and R. V. Hanxleden, "Runtime enforcement of cyber-physical systems," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 178:1–178:25, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3126500

[52] W. Qiang and Z. Zhongli, "Reinforcement learning model, algorithms and its application," in *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, Aug 2011, pp. 1143–1146.

[53] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[54] J. Redmon and A. Angelova, "Real-time grasp detection using convolutional neural networks," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1316–1322.

[55] D. M. Rodvold, "A software development process model for artificial neural networks in critical applications," in *Neural Networks, 1999. IJCNN '99. International Joint Conference on*, vol. 5, 1999, pp. 3317–3322 vol.5.

[56] J. D. Rolston, D. A. Wagenaar, and S. M. Potter, "Precisely timed spatiotemporal patterns of neural activity in dissociated cortical cultures," *Neuroscience*, vol. 148, no. 1, pp. 294–303, 2007.

[57] P. S. Roop, S. Andalam, R. Von Hanxleden, S. Yuan, and C. Traulsen, "Tight WCRT analysis of synchronous c programs," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2009, pp. 205–214.

[58] S. Russell, D. Dewey, and M. Tegmark, "Research priorities for robust and beneficial artificial intelligence," *Ai Magazine*, vol. 36, no. 4, pp. 105–114, 2015.

[59] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[60] F. B. Schneider, "Enforceable security policies," in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, Dec 2003, pp. 117–137.

[61] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449 – 471, 2015.

[62] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, Grenoble, France, March 2011, pp. 11–20.

[63] S. A. Seshia, D. Sadigh, and S. S. Sastry, "Towards verified artificial intelligence," *arXiv preprint arXiv:1606.08514*, 2016.

[64] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, no. 0,

pp. –, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608012000457

[65] J. Stewart, "Tesla's autopilot was involved in another deadly car crash," *Wired*, Mar 2018. [Online]. Available: https://www.wired.com/story/tesla-autopilot-self-driving-crash-california/

[66] C. Szepesvari, "Algorithms for reinforcement learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 4, no. 1, pp. 1–103, 2010. [Online]. Available: https://doi.org/10.2200/S00268ED1V01Y201005AIM009

[67] T-CREST. (2019) Patmos. [Online]. Available: https://github.com/t-crest/patmos

[68] Tama66. (2018) port-3109757. [Online]. Available: https://pixabay.com/en/port-crane-harbour-crane-3109757/

[69] S. Tripakis, "Data-driven and model-based design," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, May 2018, pp. 103–108.

[70] J. Wang, M. Mendler, P. Roop, and B. Bodin, "Timing analysis of synchronous programs using WCRT algebra: Scalability through abstraction," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 177, 2017.

[71] P. Wei, L. Cagle, T. Reza, J. Ball, and J. Gafford, "LiDAR and Camera Detection Fusion in a Real Time Industrial Multi-Sensor Collision Avoidance System," *ArXiv e-prints*, Jul. 2018.

[72] WikiImages. (2013) mars-67522. [Online]. Available: https://pixabay.com/en/mars-mars-rover-space-travel-robot-67522/

[73] B. M. Wilamowski, "Neural network architectures and learning algorithms," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 56–63, Dec 2009.

[74] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem: overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[75] T. Wu, C. Tsai, and J. Guo, "Lidar/camera sensor fusion technology for pedestrian detection," in *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, Dec 2017, pp. 1675–1678.

[76] B. Yegnanarayana, "Artificial neural networks for pattern recognition," *Sadhana*, vol. 19, no. 2, pp. 189–238, 1994.

[77] S. Yuan, L. H. Yoong, and P. S. Roop, "Compiling Esterel for multi-core execution," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 2011, pp. 727–735.

[78] X. Zeng, D. S. Yeung, and X. Sun, "Sensitivity analysis of multilayer perceptron to input perturbation," in *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0*, vol. 4, Oct 2000, pp. 2509–2514 vol.4.