Safety Verification of Neural Network Controlled Cyber-Physical Systems under Precision Errors

Harikishan T S

Safety Verification of Neural Network Controlled Cyber-Physical Systems under Precision Errors

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Technology
in
Computer Science

by

Harikishan T S
[Roll No: CS2212]

under the guidance of

Dr. Sumana Ghosh
Assistant Professor
Electronics and Communication Sciences Unit

Indian Statistical Institute
Kolkata-700108, India

June 2024

DECLARATION

I, **Harikishan T S**, hereby declare that the dissertation titled **"Safety Verification of Neural Network Controlled Cyber-Physical Systems under Precision Errors"** submitted to Indian Statistical Institute, Kolkata, in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Computer Science**, is my original work. This dissertation has not been submitted to any other university or institution for the award of any degree or diploma. All sources of information and assistance received during the preparation of this dissertation have been duly acknowledged.

Harikishan T S (CS2212)

Master of Technology in Computer Science
Indian Statistical Institute, Kolkata

CERTIFICATE

This is to certify that the dissertation titled **"Safety Verification of Neural Network Controlled Cyber-Physical Systems under Precision Errors"** submitted by **Harikishan T S** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the rules and regulations of this institute and, according to me, has reached the standard needed for submission.

Sumana Ghosh

Sumana Ghosh

Assistant Professor
Electronics and Communication Sciences Unit (ECSU)
Indian Statistical Institute, Kolkata

Acknowledgement

I would like to thank my supervisor, *Dr. Sumana Ghosh*, Electronics and Communication Sciences Unit, Indian Statistical Institute, Kolkata and my collaborator *Dr. Debasmita Lohar*, Karlsruhe Institute of Technology, Germany for their continuous guidance and unwavering support. For the entire duration of the thesis, I have had many opportunities to learn and improve myself and my work. Their guidance has given me a much better appreciation of the research sphere and the value of good-quality work.

My deepest thanks to the faculties of the Indian Statistical Institute for their support and assistance throughout the duration of the thesis.

I would also like to thank my parents, friends and peers for their help and support. I thank all those, whom I have missed out on the above list.

Harikishan T S
Roll No. CS2212
Indian Statistical Institute
Kolkata - 700108, India.

Abstract

Cyber-physical systems (CPS) are increasingly utilizing neural networks (NN) as controllers due to their ability to model complex, non-linear dynamics of the system. Thus, ensuring the safety of these systems becomes crucial, specifically in the context of safety-critical applications. State-of-the-art safety verification techniques for CPS primarily analyze the reachability of safe states, considering bounded errors stemming from noisy dynamic environments or inaccurate implementations. However, it assumes real-valued arithmetic and does not account for the round-off error due to fixed-point quantization while deploying NN controller on resource-constrained embedded systems. Some recent efforts have focused on generating sound quantized NN implementations in fixed-point, ensuring specific target error bounds, but they assume the safety of the NN controller is already proven.

To bridge this gap, this thesis introduces *Nexus*, a novel two-phase framework combining reachability analysis of CPS with real-valued NNs and sound NN quantization. Nexus provides an end-to-end solution that ensures CPS safety within bounded errors while generating mixed-precision fixed-point implementations for NN controllers. In the first phase, Nexus performs reachability analysis on a CPS using a real-valued NN controller to compute over-approximated reachable sets that prove system safety up to a given time bound. In the second phase, Nexus generates an optimized fixed-point mixed-precision implementation of the NN controller that preserves this safety guarantee. Additionally, Nexus's extended code generation exploits the inherent parallelism in neural network computations to generate implementations for automated parallelization on FPGAs using directives in commercial HLS compilers. Nexus has been evaluated on 12 neural network-controlled CPS benchmarks and demonstrated Nexus's ability to perform safety verification and quantization of the NN controller while satisfying the given error bound to generate safe hardware implementation of the NN controller. Nexus's extended code generation significantly reduces the latency of the implementation for all the benchmarks. We also evaluate Nexus's extended code generation on larger benchmarks to show the scalability of Nexus while generating efficient implementation.

Keywords: Cyber-physical systems, safety verification, reachability analysis, neural network controller, mixed-precision fixed-point quantization.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A Cyber-Physical System (CPS) is an autonomous system consisting of physical plants, software controllers, sensors, actuators and a communicating network. The plant and controller of CPS operate in a feedback loop to carry out its intended tasks. An example of CPS is automatic pilot avionics. The task is to navigate through the air space without colliding with any obstacles. Here, the flight is the plant and the software which controls the flight is the controller. Depending on the current plant state, the controllers take the appropriate action. Generally, the controller inspects the plant state at a regular interval and this interaction between the plant and the controller continues to preserve the desired functionality of the entire plant-control system.

Modern CPSs are rapidly evolving to meet the demands of increasingly complex functionalities within highly dynamic, nonlinear, and uncertain environments. Consequently, traditional CPS design methods face challenges in effectively navigating through these intricate requirements. In response, there is a growing trend toward adopting neural networks (NNs) as controllers for a variety of closed-loop systems due to their inherent flexibility and adaptability. NNs are employed to generalize the controller function. So, it can generate control action for all possible states. The advent of massively parallel computation based on GPUs has enabled the efficient training and deployment of NNs. Moreover, NNs can model complex control functions occupying less memory. For example, in the case of ACAS Xu (Airborne Collision Avoidance System for drones), in general, the controller lookup table is of size 2GB whereas the NN occupies only 3MB [14]. However, deploying these NN-controlled systems in safety-critical applications, such as adaptive cruise control of cars or collision avoidance systems of airplanes [20], necessitates ensuring that these systems operate without violating their safety while optimizing their implementations for efficiency.

Recent efforts have focused on automatically verifying the safety and correctness of NN-controlled systems [9–11, 36]. These methods typically consider NN controllers trained in high-precision floating-point arithmetic on server-like machines equipped with graphics processing units (GPUs), which emulate exact real-valued arithmetic. These NN controllers sense system states at discrete intervals, compute control values, and adjust system dynamics given as ordinary differential equations. The goal of the verification here is to estimate safe, reachable states of the whole system within a finite time, accounting for bounded errors either due to noisy environments or implementation inaccuracies. How-

ever, implementing these high-precision floating-point NN controllers directly in the embedded architectures is often impractical for resource-constrained embedded systems due to high computation costs or the unavailability of dedicated floating-point co-processors or software-based emulation capabilities. Consequently, in practice, these trained NNs are quantized using low-precision fixed-point arithmetic [15, 16] to achieve efficiency in terms of area or latency. Therefore, it becomes crucial to verify that after quantization, the implementations of NN controllers still satisfy the error bounds derived (or utilized) during safety verification. Unfortunately, state-of-the-art safety verification does not account for the final implementations of the controllers and thus does not guarantee that implementations meet the derived error bounds.

Various approaches have been proposed for automated quantization of neural networks. Some methods adopt a uniform precision for all layers of models [15, 16, 21], which may not always be optimal and not all layers may have similar effects on the overall accuracy of the model. This is because, in uniform precision, if one precision is barely insufficient, upgrading all operations to higher precision becomes necessary. Moreover, not all layers may have similar effects on the overall accuracy of the model. Consequently, other approaches [24,29,31,32] focusing on mixed-precision quantization have gained popularity in recent times. This is because it offers different precision values for different operations or layers, resulting in greater resource savings compared to uniform precision. However, all of these techniques, do not provide any guarantee for all possible input scenarios. While most quantization techniques focus on dynamically comparing the classification accuracy of NN classifiers, providing no guarantee for all possible inputs, only the tool Aster [24] concentrates on sound NN controller computing control values or performing regression tasks. It generates implementations that guarantee predefined error bounds. However, it solely concentrates on quantization and assumes that the neural network controller models have already been proven safe.

In this thesis, to bridge this gap, we propose a two-phase framework called *Nexus*, which combines safety verification with sound mixed-precision fixed-point quantization of NN controllers. We employ a scalable reachability analysis technique to first prove the safety of the closed-loop NN-controlled system in finite time, considering an error that bounds the inaccuracies of the implementation. If the system is proven safe, we then utilize the state-of-the-art sound NN quantizer Aster to generate a mixed-precision fixed-point implementation guaranteeing the predefined error bound, which can be directly compiled using commercial HLS compilers. Furthermore, Nexus extends Aster's code generation and produces code with loops that can be automatically parallelized by the HLS compiler to improve the latency of the final implementations. In summary, Nexus ensured safety for 7 benchmarks at $1e-3$ and 8 benchmarks at $1e-5$ error bounds and generated their implementations. Its extended code generation significantly reduced latency compared to other methods. For larger benchmarks, Nexus minimized latency, highlighting its capability for handling large neural network functions efficiently. This demonstrates Nexus' efficiency in generating optimized, parallelized NN controller implementations.

1.1 Motivation of this Dissertation

Neural network controllers must be verified before deploying them in safety-critical real-time applications. Reachability analysis methods developed so far are limited to safety

verification of NN controllers which are trained on high-precision floating-point arithmetic (software-level verification). However, in practice, the NN controllers have to be implemented in resource-constrained embedded platforms with the support of finite fixed precision only. There might be an unsafe trace of system states which violates the safety properties because of the round-off errors coming due to fixed precision-based implementation in the hardware. To ensure the robustness of CPS, errors due to quantization have to be taken care of in the safety verification of the NN controllers as well as while generating hardware implementation after quantization to be used for the underlying embedded platform.

1.2 Contributions of this Dissertation

In this thesis, we introduce *Nexus*, a two-phase framework that integrates safety verification with mixed-precision fixed-point quantization of NN controllers. We employ a scalable reachability analysis to first prove the safety of the closed-loop NN-controlled system in finite time, considering an error for implementation inaccuracies. If the system is proven safe, we then utilize the state-of-the-art sound NN quantizer Aster to generate a mixed-precision fixed-point implementation guaranteeing the error bound. This implementation can then be compiled using commercial HLS compilers. Furthermore, *Nexus* extends Aster's code generation to generate code featuring loops, automatically parallelizable by HLS compilers, thereby reducing the latency of final implementations.

Although there is a large body of work on integrating different analyses, our goal of combining reachability analysis with sound NN quantization is novel as it presents the first comprehensive solution for precision-aware, safe, NN-controlled CPSs that are more useful in more practical applications. In summary, the thesis makes the following contributions.

- *Development of Nexus*: A two-phase framework Nexus that combines closed-loop safety verification of CPS and sound quantization of NN controllers, thus providing an end-to-end solution to the problem.

- *Extended Code Generation*: An efficient, scalable and parallelizable code generation for NN controllers to produce code formats compatible with a commercial HLS compiler for automated parallelization with lesser latency and compilation time.

- *Experimental Evaluation*: We use standard benchmarks of CPS to evaluate our tool and demonstrate that *Nexus* can generate a safe and efficient implementation for these benchmarks. These benchmarks are collected from both academic and industrial settings.

Details of the benchmarks, the proposed approach Nexus along with its toolchain, and the details of all of our experiments are available publicly in `https://github.com/harikishants/Nexus`.

1.3 Organization of the Dissertation

This dissertation is organized into 7 chapters. A summary of the contents of the chapters is as follows:

Chapter 1: This chapter contains an introduction and a summary of the major contributions of this work.

Chapter 2: This chapter describes the background required for this thesis and the literature survey to position the novelty of this thesis work.

Chapter 3: This chapter describes the two tools used for the safety verification of CPS and the quantization of neural network controllers. These two tools are used within our tool framework.

Chapter 4: This chapter describes the proposed two-phase approach of safety verification and quantization and the corresponding tool framework, *Nexus*.

Chapter 5: This chapter describes all the benchmarks used and the results of experiments done using Nexus.

Chapter 6: This chapter summarizes with conclusions on the contributions of this dissertation.

Chapter 2

Background and Literature Survey

This chapter provides the essential background required for understanding the concepts discussed in this thesis and presents a comprehensive literature survey on the existing methods for safety verification and quantization of neural network controllers.

2.1 Cyber-Physical System

A cyber-physical system (CPS) is an autonomous system comprising physical plants, software controllers, sensors, actuators and a communicating network. The two major components of a CPS are the plant and the controller. The plant is characterized by a set of plant variables $\bar{\mathbf{x}} = [x_1, x_2, \ldots, x_n]$, collectively known as the *plant state*. Generally, a set of control inputs $\bar{\mathbf{u}} = [u_1, u_2, \ldots, u_k]$ are applied to the plant at regular intervals, known as the *sampling period* or *control period*. The controller monitors the plant state $\bar{\mathbf{x}}$ regularly following the control period h and then it applies the appropriate control input $\bar{\mathbf{u}}$. The closed-loop interaction between the plant and the controller where the plant operates under the guidance of the controller is illustrated in Figure 2.1. The continuous-time behaviour of the plant is discretized based on the value of the control period. The plant and controller models are usually represented by linear or non-linear ordinary differential equations (ODEs).

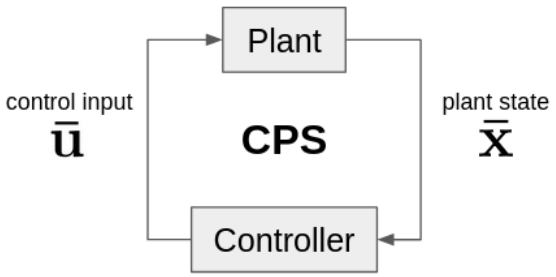

Figure 2.1: Plant-Controller Loop of CPS

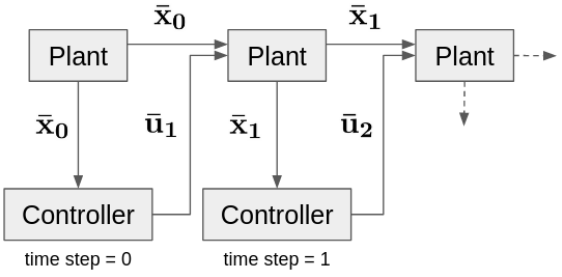

Figure 2.2: System Evolution over Time

The system starts with an initial plant state $\bar{\mathbf{x}}_\mathbf{0}$ and the controller provides the corresponding initial control input, $\bar{\mathbf{u}}_\mathbf{1}$, to the plant. At any discrete time step k, the plant

state $\bar{\mathbf{x}}$ is updated based on the plant dynamics, the previous plant state $\bar{\mathbf{x}}_{\mathbf{k-1}}$, and the current control input $\bar{\mathbf{u}}_{\mathbf{k}}$, as shown in Figure 2.2. The control design problem involves finding a sequence of control inputs $\bar{\mathbf{u}}_{\mathbf{1}}, \bar{\mathbf{u}}_{\mathbf{2}}, \ldots$, over time, such that the sequence of states $\bar{\mathbf{x}}_{\mathbf{0}}, \bar{\mathbf{x}}_{\mathbf{1}}, \ldots$, starting from an initial state $\bar{\mathbf{x}}_{\mathbf{0}}$, meet certain safety properties or reach certain desired state.

2.2 Neural Network Controller

Modern CPSs are rapidly evolving to handle complex functionalities in highly dynamic, nonlinear, and uncertain environments. With the growing complexity of physical systems and this ever-increasing demand for novel functionalities, classical control design methodologies are falling short. Traditional feedback controllers like PID and LQR [3] are being increasingly replaced by Neural Network (NN) controllers. In certain application domains like airplane collision avoidance systems, control inputs/actions for all possible plant states are stored in a lookup table [14]. Given a plant state, the appropriate control action is selected accordingly. However, in the case of an infinite number of plant states, storing control actions for all of them is not practically feasible. Nowadays, in most autonomous systems, various sensors like cameras, LiDARs are used to calculate system information, e.g., distance, velocity, etc, from images through computer vision algorithms comprising neural networks. These networks incur delays in sensing and lead to highly uncertain and dynamic system behaviour. Traditional controllers are not capable of taking care of all such dynamic situations.

These are the primary reasons for the replacement of traditional controllers by learning-based components such as artificial neural networks for their flexibility and adaptability. NNs are employed to generalize the controller function. The advent of massively parallel computation based on GPUs has enabled the efficient training and deployment of NNs. NNs can model complex control functions occupying less memory. For example, in ACAS Xu (Airborne Collision Avoidance System for drones), the control action lookup table size is 2GB whereas the NN occupies only 3MB NN controllers are trained using techniques like Reinforcement Learning etc.

2.3 Safety Verification of Neural Network Controlled CPS

Since most of the NN-controlled CPSs have their application in safety-critical domains like automotive, avionics, etc., safety verification plays a crucial role. In particular, safety verification is an essential step before we deploy NN controllers in real-time applications. NN-controlled systems require ensuring safety and being optimized for efficiency under all operating conditions. In general, CPSs satisfy certain safety properties based on the application. For example, the maximum velocity of an autonomous car should be within a certain speed limit. The goal of a safety verification task is to ensure that no system behaviour violates the specified safety properties. Most approaches involve deriving a set that captures all possible system behaviours and then verifying that this set satisfies the safety property. Thus, it ensures that the system will provide correct functionality all the time.

The safety verification problem is solved by computing the reachable states of the system and ensuring that they do not intersect with the unsafe region. We can formally define the safety verification problem as follows.

Definition 2.1 (Safety Verification Problem) *Given a plant and a controller, a set of initial states X_0, and a set of unsafe states X_{unsafe}, does the closed-loop system starting from some state in X_0 reach any state in X_{unsafe}?*

There are two distinct types of safety properties that we have focused on in this work.

- **Global Safety Properties**: These must be satisfied at all time steps under consideration.

- **Targeted Safety Properties**: These need to be satisfied only at specific time intervals, often referred to as target properties.

The reachable state computation of the closed-loop system either through a direct state-space exploration or through an abstraction-based analysis, requires solving reachability problems on the neural network controller and the dynamical system [13, 36]. Here, we focus specifically on the reachability analysis of neural networks, since it is sparsely explored in the literature.

2.3.1 Reachability Analysis

In formal verification, the system is defined by a mathematical model, and we aim to prove that no behaviour of the model violates the given property. This typically involves computing all reachable states of the model; if no unsafe state is included, the system is deemed safe. This approach is known as reachability analysis.

Definition 2.2 (Reachability Problem) *Given an n-dimensional system $S : \dot{\mathbf{x}} = f(\mathbf{x}, u)$ and an initial set $X_0 \subseteq \mathbb{R}^n$, the reachability problem for S is to verify whether a given state $\mathbf{Y} \in \mathbb{R}^n$ is reachable within a specified time interval T. The problem is referred to as bounded if T is finite.*

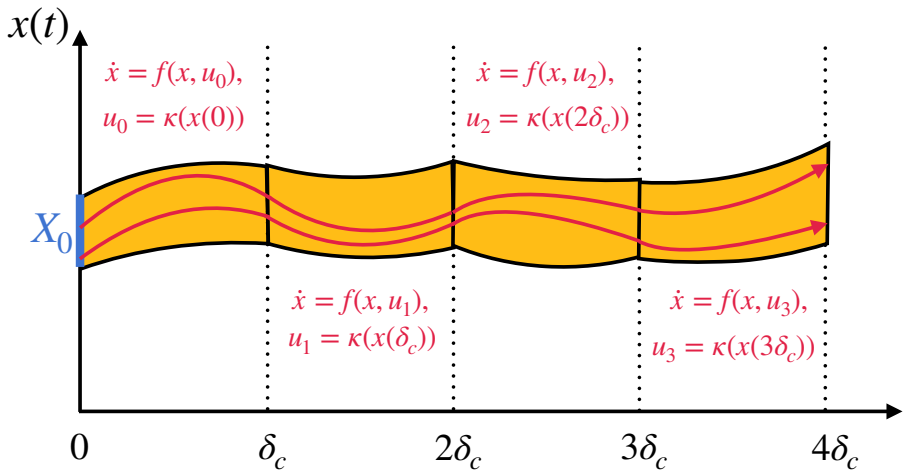

Figure 2.3: Flowpipe Computation

Soundness of Reachability Analysis: Determining the reachability of a state of a system is challenging due to the state-space explosion problem. Nonetheless, we can compute an over-approximation of the reachable state set. If this over-approximation does not contain any unsafe state, the exact reachable set is also safe. If the over-approximation does contain unsafe states, the safety remains uncertain, and we may attempt to refine the over-approximation.

Interval Arithmetic: To ensure conservativeness during reachability analysis every real number is treated as an interval. This approach captures the uncertainties and variations in system behaviours by enclosing the actual values within lower and upper bounds. For instance, instead of representing a variable x as a precise real number, it is represented as an interval $[x_{\min}, x_{\max}]$, where $x_{\min}$ and $x_{\max}$ are the minimum and maximum possible values of x, respectively. By using interval arithmetic, the reachability analysis constructs an over-approximation of the reachable set given an initial condition. Given a bounded time horizon $[0, T]$, the reachable set is iteratively computed as a sequence of sets $F_1, \ldots, F_N$, where each F_i (for $1 \leq i \leq N$) is an over-approximation of the reachable set over a time segment at the i-th step. This technique, known as flowpipe construction, helps in ensuring that all possible system behaviours are considered.

Consider Figure 2.3 where the system starts from the initial plant state X_0 with dynamics f and a controller κ. The control period or control step size is denoted as δ_c. The flowpipe is computed at each δ_c. Additionally, the flowpipe can be refined within $[0, \delta_c]$ into smaller time intervals, referred to as the *flowpipe step size*. The series of states calculated over time is called the flowpipe, which serves as an over-approximation of the reachable set, encompassing all possible trajectories from X_0.

However, interval arithmetic has its limitations. One significant drawback is the *wrapping effect*, where the intervals tend to overestimate the true bounds due to repeated operations, leading to less precise results. This overestimation can result in conservative approximations that may not be practical for verifying the safety of systems with tight constraints even though they are straightforward and computationally less intensive.

2.3.2 Taylor Model Arithmetic

Function over-approximation methods, such as Taylor models, provide a more accurate alternative to interval approximation. Taylor models utilize Taylor series expansions combined with interval arithmetic to bound the remainder terms, resulting in reduced wrapping effects and tighter bounds compared to pure interval arithmetic. By considering higher-order derivatives of system functions, Taylor models can capture the system's dynamics more precisely, leading to less conservative and more informative over-approximations. Originally proposed to compute higher-order over-approximations for the ranges of continuous functions [5], Taylor models can be viewed as a higher-order extension of intervals [28], which represent sets of real numbers between lower and upper bounds. For example, the interval $[a, b]$, where $a \leq b$, represents the set $\{x \mid a \leq x \leq b\}$.

Definition 2.3 (Taylor Model) *A Taylor model (TM) consists of a polynomial p of degree k over a finite set of variables $x_1, \ldots, x_n$ defined on an interval domain $D \subset \mathbb{R}^n$, along with a remainder interval I.*

The range of a Taylor model is obtained by taking the Minkowski sum of the range of its polynomial and the remainder interval. Thus, a range of TM (p, I) is represented as $p + I$. Taylor models are closed under operations such as addition, multiplication, and integration [5]. Given functions f and g over-approximated by Taylor models (p_f, I_f) and (p_g, I_g) respectively, a Taylor model for $f + g$ can be computed as $(p_f + p_g, I_f + I_g)$ [36], and an order k Taylor model for $f \cdot g$ can be computed as $(p_f \cdot p_g - r_k, I_f \cdot B(p_g) + B(p_f) \cdot I_g + I_f \cdot I_g + B(r_k))$, where $B(p)$ denotes an interval enclosure of the range of p, and the truncated part r_k consists of the terms in $p_f \cdot p_g$ of degrees greater than k.

Definition 2.4 (Taylor Polynomial) *The order k Taylor polynomial of $f(x)$ at $x = c$ for some $c \in (a, b)$ is given by:*

$$p_k(x) = f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 + \cdots + \frac{f^{(k)}(c)}{k!}(x - c)^k.$$

If $f(x)$ is also $(k + 1)$ times differentiable, the approximation error of $p_k(x)$ for any $x \in (a, b)$ can be expressed as:

$$r(x) = f(x) - p_k(x).$$

This error can be explicitly expressed by the Lagrange remainder term $r_k(x)$:

$$r_k(x) = \frac{f^{(k+1)}(\xi)}{(k + 1)!}(x - c)^{k+1}$$

for some ξ between x and c.

The interval remainders capture the error due to over-approximation.

Example 2.1 (Approximating a Function) *Approximating $f(x) = \exp(x)$ over the interval $x \in [-1, 1]$ using a second-order Taylor model.*

Evaluating at the midpoint $x = 0$, we obtain the second-order polynomial approximation $p_2(x) = 1 + x + \frac{1}{2!}x^2$. The remainder interval I is calculated as $\frac{1}{3!} \cdot f'''(\xi) \cdot \xi^3$, where $\xi \in [-1, 1]$, resulting in $I = [-0.453, 0.453]$. Thus, $\exp(x)$ is approximated by the following Taylor model.

$$(p, I) = TM\left(1 + x + \frac{1}{2!}x^2, [-0.453, 0.453]\right)$$

Example 2.2 (Approximating ODEs of Plant-Controller Model) *Consider a plant with three variables x_1, x_2, and a controller u, governed by the following ordinary differential equations (ODEs).*

$$\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= u \\
\dot{u} &= 0 \quad \textit{(since } u \textit{ is constant as it is obtained from the controller)}
\end{aligned}$$

The initial conditions at $t = 0$ are $x_1(0) = [5, 6]$, $x_2(0) = [1, 2]$, and $u(0) = 10$ (from the controller). We aim to find the values of x_1, x_2, and u at $t = 1$. Let the control period be $h = \Delta t = 1$. We approximate this using a second-order Taylor polynomial as follows.

$$x(t + h) = x(t) + \dot{x}(t) \cdot h + \frac{\ddot{x}(t) \cdot h^2}{2!}$$

Simplifying, we have:

$$\begin{aligned}
x_1(1) &= x_1(0) + x_1'(0) + \frac{x_1''(0)}{2} = x_1(0) + x_2(0) + \frac{u(0)}{2} \\
&= [5, 6] + [1, 2] + 0 = [6, 8]
\end{aligned}$$

$$\begin{aligned}
x_2(1) &= x_2(0) + x_2'(0) + \frac{x_2''(0)}{2} = x_2(0) + u(0) \\
&= [1, 2] + 10 = [11, 12]
\end{aligned}$$

$$u(1) = u(0) + u'(0) + \frac{u''(0)}{2} = 10$$

Thus, at $t = 1$, we have $x_1(1) = [6, 8]$, $x_2(1) = [11, 12]$, and $u(1) = 10$. Here, we have used range approximation. However, function approximation works similarly where variables x_1, x_2, and u are represented as Taylor models (p, I), and Taylor arithmetic is carried out in that case.

2.3.3 Neural Network Over-Approximation

Consider an NN-controlled CPS with plant dynamics f and the NN controller κ. The reachable state at time t can be represented as $\bar{\mathbf{x}}_\mathbf{t} = f(\bar{\mathbf{x}}_{\mathbf{t-1}}, \kappa(\bar{\mathbf{x}}_{\mathbf{t-1}}))$. Functional over-approximation can track state dependencies across the closed-loop system and over multiple time steps in reachability analysis. The approach of layer-by-layer propagation computes a function over-approximation of the neural network by exploiting its structure and using Taylor model arithmetic to efficiently obtain a function over-approximation of κ. This is achieved by propagating the Taylor models (TMs) layer by layer through the network. However, due to the limitations of basic TM arithmetic, these approaches

cannot handle non-differentiable activation functions and suffer from rapid growth of the remainder during propagation. For instance, the explosion of the interval remainder would degrade TM propagation to mere interval analysis.

We describe how a Taylor model (TM) output is computed from a given TM input for a single layer, as illustrated in Figure 2.4. The circles in the right column represent the neurons in the current layer, which is the i-th layer, while those in the left column represent the neurons in the previous layer. The weights on the incoming edges to the current layer are organized as a matrix W_i, and B_i denotes the vector of biases in the current layer. Given that the output range of the neurons in the previous layer is represented as a TM

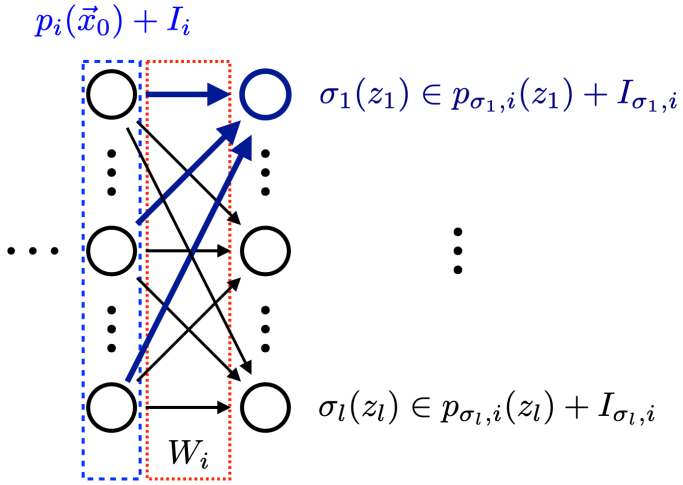

Figure 2.4: TM Propagation in Single Layer

vector $(p_i(\mathbf{x}_0), I_i)$, where $\mathbf{x}_0$ are the variables ranging in the initial set, the output TM $(p_{i+1}(\mathbf{x}_0), I_{i+1})$ of the current layer can be obtained as follows:

- Compute the polynomial approximations $p_{\sigma_1,i}, \ldots, p_{\sigma_l,i}$ for the activation functions $\sigma_1, \ldots, \sigma_l$ of the neurons in the current layer.
- Evaluate the interval remainders $I_{\sigma_1,i}, \ldots, I_{\sigma_l,i}$ for these polynomials to ensure that for each $j = 1, \ldots, l$, $(p_{\sigma_j,i}, I_{\sigma_j,i})$ is a TM of the activation function σ_j with respect to z_j ranging in the j-th dimension of the set $W_i(p_i(\mathbf{x}_0) + I_i)$.
- Compute $(p_{i+1}(\mathbf{x}_0), I_{i+1})$ as the TM composition $p_{\sigma,i}(W_i(p_i(\mathbf{x}_0) + I_i)) + I_{\sigma,i}$, where $p_{\sigma,i}(\mathbf{z}) = (p_{\sigma_1,i}(z_1), \ldots, p_{\sigma_l,i}(z_l))^T$ and $I_{\sigma,i} = (I_{\sigma_1,i}, \ldots, I_{\sigma_l,i})^T$.

For networks with multiple layers, the output TM of one layer is used as the input TM for the next layer. The final output TM is computed by composing TMs layer-by-layer, starting from the first layer.

2.3.4 Approximation of ReLU Activation

The approximation can also be achieved using polynomials other than the Taylor polynomial. One such polynomial is the Bernstein polynomial, which requires only the continuity

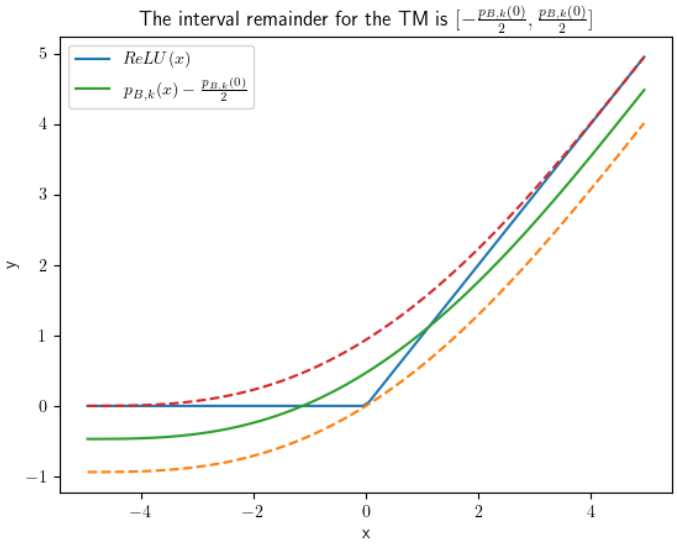

Figure 2.5: Bernstein Approximation of ReLU

of the activation function in (p_t, I_t), making it applicable in more general situations and yields better polynomial approximations than Taylor expansions [26].

Bernstein Approximation The over-approximation of ReLU using the Bernstein polynomial is shown in Figure 2.5. The Taylor model over-approximation $p(x)+I$ of ReLU(x) is given by $p(x) = p_{B,k}(x) - \frac{p_{B,k}(0)}{2}$ and $I = \left[-\frac{p_{B,k}(0)}{2}, \frac{p_{B,k}(0)}{2}\right]$, where $p_{B,k}(0)$ is the Bernstein polynomial $p_{B,k}(x)$ evaluated at $x = 0$. For $x \in [a, b]$ with $a < 0 < b$, the bounds of the interval remainder I are tight for any order-k Bernstein polynomial approximation with $k \geq 1$.

The soundness of this error-bound estimation has been proven in [17] for multivariate Bernstein polynomials. Since univariate Bernstein polynomials are a special case of multivariate Bernstein polynomials, it is also sound.

2.3.5 Selection of Polynomial Approximations

For activation functions, both Taylor and Bernstein approximations are univariate, with sizes linear in the order k. Given two order k TMs $(p_1(), I_1)$ and $(p_2(), I_2)$ approximating the same function $f()$ over a bounded domain $D \subset \mathbb{R}^n$, where $(p_1(), I_1) \prec_k (p_2(), I_2)$ denotes the former as more accurate. One needs to check whether this accuracy persists after composing with another function $g()$ approximated by a TM $(q(), J)$. *Now* $p_1(q() + J) + I_1 \prec_k p_2(q() + J) + I_2$ *does not hold using order* k *TM arithmetic* as proved in [36]. Therefore, both Taylor and Bernstein's approximations are computed for activation functions and the one with the smaller remainder interval I_r is selected.

2.4 Floating-Point and Fixed-Point Precision

In digital computing, the representation of numerical values can significantly impact the performance and accuracy of computations. Two common methods for representing numbers are floating-point precision and fixed-point precision. This section explains these two types of precision, discusses the errors involved, and compares them with examples.

2.4.1 Floating-Point Precision

Floating-point precision is a method used to represent real numbers that can handle a wide range of values by using a floating decimal point. The IEEE 754 standard is widely used for floating-point arithmetic. A floating-point number is typically represented as:

$$(-1)^s \times m \times 2^e$$

where, s is the sign bit (0 for positive, 1 for negative), m is the mantissa and e is the exponent stored following *excess* notation.

2.4.2 Fixed-Point Precision

Fixed-point precision represents numbers with a fixed number of digits after the decimal point. This method is simpler and faster than floating-point arithmetic but offers a smaller range and less flexibility. Fixed-point numbers are particularly useful in embedded systems where resources are limited. A fixed-point number is represented in the form:

$$\textit{integer part} \cdot \textit{fractional part}$$

where the integer part and fractional part determine the number of integer bits and fractional bits respectively.

Example 2.3 *Consider the real number 5.75.*

Binary Representation:
– *Integer part: 5 represented in binary as 101.*
– *Fractional part: 0.75 represented in binary as 0.11.*
– *Combining both 5.75 in binary is 101.11.*

Floating-point Representation *(IEEE 754 single-precision, 32-bit):*
– *Normalize the binary number: 101.11 becomes* 1.0111×2^2.
– *Sign bit: 0 (since 5.75 is positive).*
– *Exponent: 2 + 127 (bias) = 129. In binary: 10000001.*
– *Mantissa: 01110000000000000000000 (fractional part, padded to 23 bits).*
Thus, the floating-point Representation is: 0 10000001 01110000000000000000000

Fixed-point representation *(both the integer part and the fractional part need 4 bits):*
– *Integer part: 5 represented as 0101.*
– *Fractional part: 0.75 represented as 1100.*
Thus, the fixed-point representation is: 0101.1100

2.4.3 Error due to Round-off and Quantization

Floating-point arithmetic can introduce rounding errors because not all real numbers can be exactly represented. For instance, repeating decimals or very large/small numbers may be approximated, leading to precision loss. Fixed-point representation errors arise from both quantization and rounding. Since the number of fractional bits is fixed, some values cannot be exactly represented, leading to a quantization error. Table 2.1 illustrates

	Bits	Range	Error
`Floating-point (Single Precision)`	32	$\approx \pm 3.4 \times 10^{38}$	$\approx 10^{-7}$
`Floating-point (Double Precision)`	64	$\approx \pm 1.8 \times 10^{308}$	$\approx 10^{-16}$
`Fixed-point (8 Integer + 8 Fractional)`	16	[-128, 127.996]	$\approx 10^{-3}$
`Fixed-point (16 Integer + 16 Fractional)`	32	$\approx \pm 32768$	$\approx 10^{-5}$

Table 2.1: Comparison of Floating-Point and Fixed-Point Precision

that the fixed-point representation generally incurs larger errors compared to the floating-point representation for real numbers. Given that NN controllers operate on hardware with limited resources, the use of floating-point arithmetic can be impractical due to its demand for specialized processors or slower software emulations. Therefore, an alternative approach involves quantizing the neural network controller using fixed-point arithmetic. Hence, we focus only on fixed-point representation and arithmetic in this thesis.

2.5 Quantization of Neural Network Controller

In the past decades, the reachability analysis of NN controllers has been well studied. So far, the tools developed for safety verification consider only the real-valued network for computing control input. However, directly implementing high-precision floating-point NN controllers on embedded architectures with constrained resources is often impractical due to high computation costs or the lack of dedicated floating-point co-processors or software-based emulation capabilities. While implemented in hardware, the neural network parameters (weights and bias) have to be quantized and have to be implemented in finite-precision arithmetic, which introduces roundoff errors. Manually selecting a suitable precision that ensures safety without wasting resources is a complex task, especially for fixed-point arithmetic, where overflow freedom must also be guaranteed.

2.5.1 Fixed-Point Quantization

In a fixed-point implementation, all program variables and constants are represented using integers with an implicit format denoted as $< Q, \pi >$. Here, $Q \in \mathbb{N}$ represents the total word length (the number of bits including the sign bit), and $\pi \in \mathbb{N}$ indicates the position of the binary point, counted from the least significant bit.

This format effectively divides the total bits into an integer part $I = Q - \pi - 1$, and a fractional part π. The integer part must have enough bits to prevent overflow, ensuring the representable range is $[-2^I, 2^I]$. The fractional part determines the precision of a variable or operation. The greater the number of fractional bits, the higher the precision. Assuming truncation is used as the rounding mode, the maximum round-off error with π fractional bits is $2^{-\pi}$.

To ensure overflow freedom and sufficient accuracy of the final result, each operation must be assigned enough integer and fractional bits. To ensure that the overall round-off error does not exceed a given bound ϵ, we need to propagate and accumulate the errors of individual operations. At the same time, it is crucial to use only the necessary number of bits to avoid wasting resources.

Mixed-Precision Tuning: Using uniform fixed-point precision, where the same word length is used for all operations, can be sub-optimal. If a single point in the program requires higher precision, all operations must be upgraded to the next higher precision. However, not all operations contribute equally to the overall accuracy. Therefore, it can be more resource-efficient to assign different precision (word lengths) to different operations to meet a target error bound, implementing the model in mixed precision. That is why mixed-precision is more resource-efficient.

2.5.2 MILP-based Mixed-Precision Tuning

In this thesis, we focus on feed-forward neural networks with 'relu' and 'linear' activation functions designed for regression tasks, producing continuous outputs. The objective is to minimize the resource usage of these networks while maintaining safety by ensuring that the control outputs remain within a specified error bound. Given a real-valued neural network architecture, input ranges, and an output roundoff error bound, in case of mixed-precision tuning, the goal is to generate a fixed-point mixed-precision neural network implementation that minimizes the precision of the variables and constants while ensuring that the roundoff errors at the output remain within the specified bound [24].

The fixed-point precision tuning is formulated as a mixed-integer linear programming (MILP) problem. A cost function [7] represents the total number of bits required, considering uniform bit lengths for operations within each individual layer, but allowing bit lengths to vary between layers. The constraints optimize the number of fractional bits to ensure overflow-freedom while keeping the overall error within the specified error bound ϵ. The technique consists of three main steps as described in [24]:

1. Computing the integer bits using interval arithmetic: Interval arithmetic is used to propagate intervals through the network, computing an interval for all variables, constants, and intermediate results that soundly over-approximates the real-valued ranges. Although the errors are typically small and do not affect the integer bits in practice, this step ensures accurate range calculations.

2. Optimizing the fractional bits via MILP: Each layer involves three operations: computing the dot product, adding the bias vector, and applying the activation function. The problem of computing the fractional bits for dot product results, bias additions, and

activation functions is reduced to a mixed-integer linear programming problem. The costs of these operations are computed individually and summed to obtain the total cost of each layer. The total cost over all layers is then computed by adding up the costs of all layers.

Error constraint: The overall roundoff error of the network ϵ must be bounded by the user-specified error ϵ_{target}. The propagation error in layer $i > 1$ depends on the errors from previous layers and the absolute magnitudes of the weights. The maximum weight values for each layer are pre-computed and used as constants in the optimization problem. This ensures a sound over-approximation of the total propagated error by assuming the maximum magnification of errors for all neurons in the previous layer. The new roundoff error at layer i is defined as the sum of the errors from the activation function, dot product computation, and bias addition.

Range constraint: The integer bits required for the real-valued result are pre-computed and used to determine the maximum representable range after each operation. It is essential to ensure that the finite ranges after each operation in each layer do not overflow.

3. Computing the precision of constants and intermediate variables: After solving the MILP, the fractional bits required for the dot product and bias addition are obtained by ensuring that the integer bits computed in the first phase are sufficient to prevent overflow even in the presence of errors. To generate a complete executable fixed-point implementation, the precision of intermediate operations (sum of products in the dot product) and the constants are also computed. The fractional bits of the weight constants and the bias constants are also determined to ensure the error bound holds.

2.6 Literature Survey

In this section, we review existing methods for safety verification and quantization of neural network controllers. Our literature survey covers a range of tools and techniques currently being developed and used in this direction of research.

2.6.1 Safety Verification

The safety verification of NN-controlled CPSs has attracted significant attention in the past few years [9–11, 18, 33, 36]. One scalable attempt towards this is [10] where the idea of local Taylor model over-approximation of the NN controller is leveraged and then a flowpipe is constructed [8] to compute a tight over-approximated reachable set of neural-controlled systems. In contrast, [18] converts the NN controller to an equivalent nonlinear hybrid system which is then combined with the plant dynamics to verify the given safety properties. Along a similar line, NN verification is done by combining it with standard hybrid automata verification [8] by approximating it with a polynomial with error bounds [9] or using other set representations such as star sets [33] with a tight over-approximation error bound. Each of these methods develops a tool framework describing its unique approach and capabilities. Next, we discuss some of them.

CORA The Continuous Reachability Analyzer (CORA) [1,13] focuses on scalable solutions for verifying hybrid systems with nonlinear continuous dynamics and/or nonlinear differential-algebraic equations. It supports the consideration of uncertain parameters and system inputs. CORA is implemented as an object-oriented MATLAB code, designed modularly to allow resource-efficient representations of multi-dimensional sets and operations on them, making it adaptable for various verification tasks.

JuliaReach JuliaReach [6] is a software suite for reachability computations of dynamical systems, written in Julia, a high-level language for scientific computing. The core library, ReachabilityAnalysis.jl, along with LazySets.jl for set computations, enables JuliaReach to analyze systems in both continuous-time and discrete-time semantics. JuliaReach employs several reachability algorithms which can be combined with different approximation models for efficient reachability analysis.

Verse Verse [23] is an open-source Python library for verifying multi-agent scenarios by converting scenarios into hybrid systems and using reachability analysis. It handles systems with both linear and nonlinear dynamics, uncertainty in discrete transitions, and initial states. Verse employs simulation-driven reachability analysis, using a finite number of simulations and a discrepancy function to over-approximate reachable states.

NNV The Neural Network Verification Tool (NNV) [34,35] is a MATLAB toolbox that implements reachability analysis methods for neural network verification, focusing on closed-loop neural network control systems in autonomous cyber-physical systems. NNV uses a star-set state-space representation and reachability algorithm for layer-by-layer computation of reachable sets for neural networks. It integrates with tools like HyST [4] and CORA [1, 13] for comprehensive verification of closed-loop systems.

Verisig Verisig [18] verifies the safety properties of closed-loop hybrid systems with deep neural network components. It supports sigmoid and tanh-based networks by transforming the neural network into an equivalent hybrid system, enabling the use of state-of-the-art reachability tools for verification. Verisig enhances scalability and accuracy through Taylor model preconditioning, shrink wrapping, and a parallelizable implementation in C++.

Sherlock Sherlock [9,10] is a tool for range analysis of deep feedforward neural networks, using a combination of gradient-based local search and mixed integer linear programming for global optimization. Sherlock can verify the properties of closed-loop systems with deep neural network controllers by generating polynomial rules to characterize the local behaviour of the network, facilitating the computation of reach sets for systems modelled as ODEs with neural network controllers.

POLAR-Express POLAR-Express [36] is a C++ open-source tool designed for efficient time-bounded reachability analysis of neural-network controlled systems. Built using GNU open-source libraries and the C++ library of Flow* [8], POLAR-Express

focuses on computing functional overapproximations for the flowmap that governs all system executions, rather than merely range over-approximations for NNCS reachable sets. This tool excels in the precise layer-by-layer propagation of Taylor Models (TMs) for general feed-forward neural networks. Basic TM arithmetic encounters difficulties with ReLU activation functions, which are non-differentiable and cannot be represented by polynomials, and also suffers from low approximation precision due to large remainders. POLAR-Express overcomes these challenges by employing univariate Bernstein polynomial approximation and symbolic remainders. Univariate Bernstein polynomial approximation enables the handling of non-differentiable activation functions and allows for local refinement of Taylor models. Symbolic remainders help control the growth of interval remainders, effectively mitigating the wrapping effect in linear mappings.

Although many tools use various approximation methods for verification, all these techniques assume real-valued arithmetic which often needs the support of high-end floating-point co-processors for the implementation of NN controllers. Moreover, unlike the approach presented in this thesis, they do not verify if these implementations, which are generally quantized in fixed-point arithmetic for resource-constrained embedded platforms, remain within the assumed error bounds.

2.6.2 Quantization

On the other hand, there is also rich literature on the quantization of neural networks [15, 16, 21, 24, 29, 31, 32]. We now review some of these techniques.

Uniform-Precision and Mixed-Precision Tuning Research efforts proposed in [15, 16, 21] consider a uniform precision for all layers of NN models. The issue with uniform precision is that if one precision is barely insufficient, upgrading all operations to higher precision becomes necessary, and thus, these works suffer from sub-optimality. Moreover, not all layers may have similar effects on the overall accuracy of the NN model. On the other side, approaches like [19, 22, 24, 32] focus on mixed-precision quantization. Shiftry [22] automatically chooses a mixed fixed-point precision by iteratively reducing the precision of variables in recurrent neural networks from 16 to 8 bits to run on memory-restricted hardware. Daisy [19] includes a sound mixed-precision tuning procedure for both floating-point and fixed-point arithmetic, using an iterative search approach. It addresses difficulties in phrasing and solving sound global optimization problems, treating errors as uncorrelated and additive, thus providing a feasible optimization approach for neural networks of limited size. Aster [24] formulates the bit allocation for fixed-point precision to an optimization problem and generates an implementation. The allocation bits are done such that the implementation satisfies a target error bound. However, the primary issue is it assumes that the neural network controllers are already safe, and therefore, does not investigate further as we do in this thesis work.

Post-Quantization Verification A recent work [27] introduces a MILP formulation for verifying the robustness of quantized neural networks, focusing on post-quantization verification without considering the impact of roundoff errors. Another study [37] focuses on computing output ranges of feed-forward neural networks with nonlinear activations

in safety-critical systems, encoding nonlinear activations with linear constraints. Quantization techniques are commonly used to reduce the memory footprint of neural networks. Most methods have been applied to neural network classifiers outside of safety-critical applications without providing accuracy guarantees. These methods typically select a uniform (custom) floating-point or fixed-point precision and demonstrate empirical performance on specific datasets.

Apart from this, the only work similar in flavour to ours is presented in [30], where the focus is also on verifying a floating-point implementation w.r.t a high-level linear time-invariant model of the controller. However, they do not consider NN controllers or fixed mixed-precision implementations. Also, they extract a high-level model from the implementation and verify it, whereas, in this thesis, we automatically generate the implementation that satisfies the properties of the high-level model.

Chapter 3

Underlying Tools for Safety Verification and Quantization

There is a wide range of tools developed by researchers over the past few years like Verisig [18], Sherlock [9], NNV [35], Juliareach [6], POLAR-Express [36] etc. Among them, POLAR-Express (polynomial arithmetic framework) developed in 2023 shows that it could perform better than the other state-of-the-art tools in most of the verification benchmarks. POLAR-Express requires less execution time and support for many activation functions and efficient over-approximation methods. This motivates us to use the POLAR-Express tool for the safety verification of NN-controlled CPS in our work.

In the literature, there exist various automated quantization approaches for NNs. Some of them use uniform precision for all layers [15,16,21], while others focus on mixed-precision quantization [24,29,31,32]. In recent times, mixed-precision-based quantization has gained more popularity for using different precisions for different operations or layers, leading to greater resource savings. While most quantization methods dynamically compare classification accuracy without guaranteeing for all possible inputs, only the tool Aster [24] provides a mixed-precision fixed-point arithmetic quantization approach that optimizes the number of bits needed to implement a network while guaranteeing a provided error bound and generates a sound NN controller implementation. This is the primary reason, we use the tool Aster for NN quantization in our work.

Therefore, this chapter presents the overview of these two tools; POLAR-Express and Aster; that we have used in this thesis work for developing the precision-aware safe implementation of NN controllers.

3.1 POLAR-Express: The Tool for Safety Verification

POLAR-Express [36] enables precise layer-by-layer propagation of TMs for general feed-forward neural networks. Basic Taylor model arithmetic cannot handle ReLU, which is non-differentiable and suffers from a large remainder. POLAR-Express overcomes these key challenges by employing a novel use of univariate Bernstein approximation and symbolic remainder. The univariate Bernstein polynomial facilitates the handling

of non-differentiable activation functions and allows local refinement of Taylor models. Symbolic remainder helps to taper the growth of interval remainders by avoiding the wrapping effect (ref. Chapter 2). POLAR-Express inherently uses *Flowstar* [8] for Taylor model arithmetic along with Taylor and Bernstein approximations for neural network over-approximation. An overview of POLAR-Express is depicted in Figure 3.1. It takes

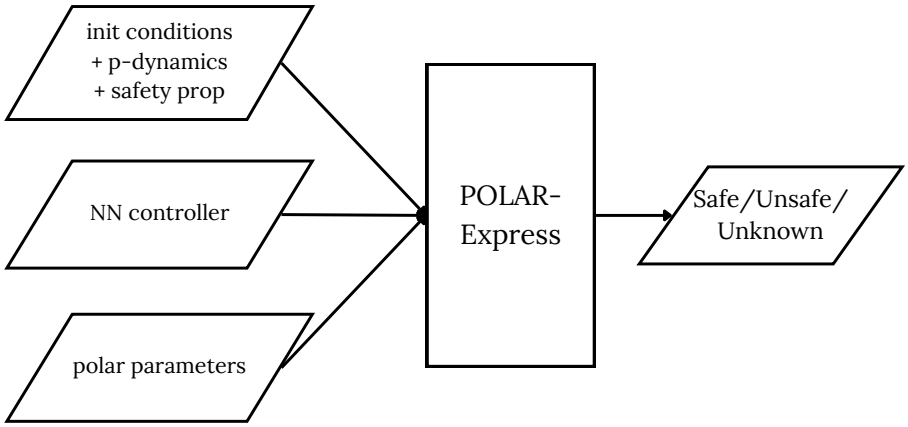

Figure 3.1: An Overview of POLAR-Express

plant and controller dynamics, initial plant state information, and the safety property to be verified as inputs. As output it declares as 'safe' if the given safety property is satisfied, 'unsafe' if not, and 'unknown' if it fails to decide safety (mainly due to excessive over-approximation error). The initial state is represented as $\bar{\mathbf{x}}_\mathbf{0}$, and the reachable state at time t is given by $\bar{\mathbf{x}}_\mathbf{t} = f(\bar{\mathbf{x}}_\mathbf{t-1}, \kappa(\bar{\mathbf{x}}_\mathbf{t-1}))$, where f and κ represent the plant dynamics and the NN controller, respectively. POLAR-Express iteratively simulates the plant-controller loop for each control step up to a specified maximum time, calculating the reachable states $\bar{\mathbf{x}}_\mathbf{0}, \bar{\mathbf{x}}_\mathbf{1}, \ldots, \bar{\mathbf{x}}_\mathbf{T}$, where T is the final time step. The polar parameters include information which is tool specific like Taylor order, Bernstein order, flowpipe step-size, etc.

For each time step, the safety property is evaluated, and the system's safety status is

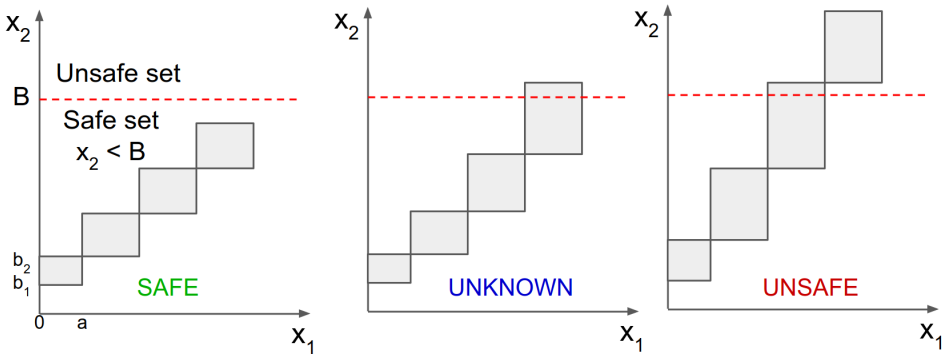

Figure 3.2: Checking Safety Property during Flowpipe Computation

reported. If any safety property is violated at any step, the system is deemed unsafe. The system is considered safe only if all safety properties are satisfied in all steps. If the over-approximation is unable to verify a property or the process terminates due to excessive

over-approximation error, the system's safety status is declared as unknown. Consider two variables, x_1 and x_2, starting from initial conditions $[0, a]$ and $[b_1, b_2]$, respectively, as shown in Figure 3.2. Let the safety property be defined on x_2, such that $x_2 < B$, and simulate for three time steps. The system is safe if all the over-approximated flowpipes lie within the boundary $x_2 = B$. The system is unsafe if at least one flowpipe is entirely outside the boundary $x_2 = B$. The safety status is unknown if the flowpipes intersect the boundary $x_2 = B$, making it unclear whether the system is safe or unsafe.

3.2 Aster: The Tool for Neural Network Quantization

Aster [24] is a fully automated mixed-precision quantization tool designed specifically for deep feed-forward neural networks with ReLU and linear activation functions. Aster takes a neural network, an error bound ϵ, and the input ranges of the neural network as input, as shown in Figure 3.3. These input ranges represent the operational range of the NN controller, assuming that only values within these ranges will be provided as input to the controller. Aster parameters include the maximum and minimum fractional bit lengths (π range) of the input vector and the initial fractional bit length (π_0) (ref. Chapter 2).

Given these inputs, Aster quantizes the neural network by optimizing the number of bits allocated for each real value used in the NN controller definition. Thus it enhances the performance and reduces memory usage while satisfying the user-defined error bound on the NN output. The tool integrates with the SCIP optimization suite [12], utilizing the underlying SoPlex solver to solve the underlying mixed-integer linear problem that is required to synthesize the optimal number of bits for all real values in the NN code. Aster generates fully quantized code with accurate precisions for the weights, written in C++ using the `ap_fixed` library.

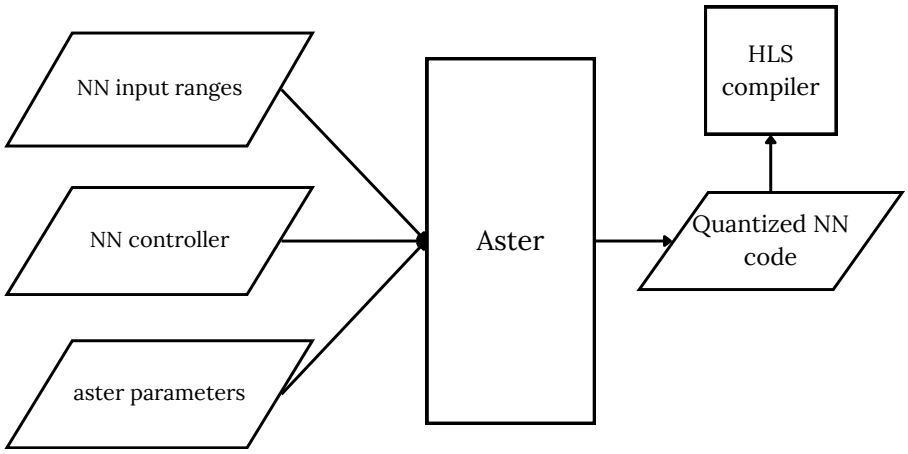

Figure 3.3: Aster Input and Output

Example: `ap_fixed<18, 6>` represents an 18-bit variable with 6 bits for the integer part (including the sign bit) and 12 bits for the fractional part.

This code can be directly compiled for an FPGA using any state-of-the-art HLS compiler like Xilinx. Aster's generated code expresses matrix operations as for-loops or, alternatively, can fully unroll these loops.

Chapter 4

Nexus: A Novel Approach for Precision-Aware Safe Implementation of NN Controllers

The quantization of neural network controllers introduces round-off errors in computation, potentially leading to imprecise control actions in CPS. These errors in control actions accumulate over time as the plant-controller loop iterates, ultimately impacting the system's ability to satisfy the safety properties. Even though the system was declared safe when computation was done in high precision with floating-point arithmetic, the round-off error due to quantization in the embedded system may now cause the system to be unsafe which is highly critical. Therefore, it is crucial to verify that after quantization, the implementations still satisfy the error bounds derived during safety verification.

In this chapter, we introduce the Nexus framework, a two-phase approach that combines safety verification with sound neural network (NN) controller quantization for implementing on resource-constrained embedded systems. Nexus begins with the safety verification phase, using reachability analysis to prove system safety under real arithmetic while considering quantization errors. If the system is verified as safe, the framework proceeds to the quantization phase, where it generates mixed-precision fixed-point implementation while maintaining the error bound. The chapter details each phase, from verifying safety, extracting safe ranges, and optimizing and generating efficient hardware implementation that leverages parallelism for improved performance. This chapter also covers the architecture and workflow of the Nexus tool.

4.1 The Two-Phase Framework of Nexus

First, we present the overview of the two-phase framework of Nexus and then provide a running example to elaborate on its working in detail, followed by the tool's implementation description.

Nexus takes the plant and NN controller information along with an error bound as input and generates a final safe implementation of the NN controller as the output that ensures

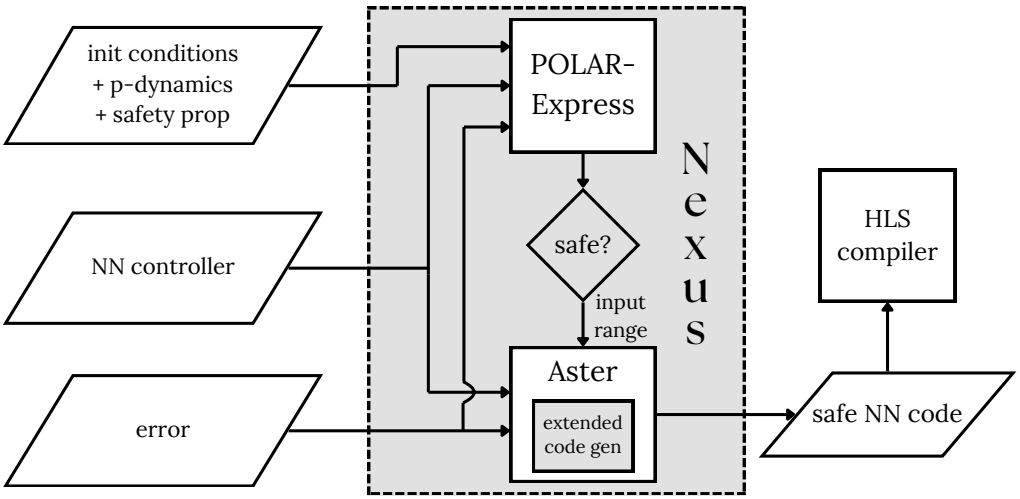

Figure 4.1: An Overview of Nexus

the predefined error bound. In particular, from the plant side, it takes the plant dynamics, the initial conditions for the plant state, and the safety property to be ensured concerning those initial conditions. The NN controller is generally trained in high-precision floating-point arithmetic. The overview of Nexus is depicted in Figure 4.1. In the two-phase framework of Nexus, the first phase involves the verification of the safety property for the given plant-control model. If the property is found safe, then in the second phase, it generates the low-precision fixed-point implementation of the NN controller guaranteeing the predefined error bound. This is how Nexus preserves the safety property of a given plant-control model even after NN quantization. Moreover, to reduce the latency during the compilation of the NN implementation, Nexus incorporates an *extended code generation* segment which produces formats compatible with a commercial High-Level Synthesis (HLS) compiler for automated parallelization.

Consider the example of an inverted pendulum (`InvPend`) taken from [20] to illustrate the working principle of Nexus. The plant state information is represented by a 4-parameter vector $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4]$ with initial conditions of the individual parameters as $x_1 \in [0.1, 0.11]$, $x_2 = x_3 = x_4 = 0$. The control input u is set to 0 initially. The plant dynamics are given by:

$$\dot{x}_1 = x_2,\ \dot{x}_2 = 0.0043x_4 - 2.75x_3 + 1.94u - 10.95x_2,$$
$$\dot{x}_3 = x_4,\ \dot{x}_4 = 28.58x_3 - 0.044x_4 + 4.44u + 24.92x_2$$

The safety property requires $x_1, x_2, x_3, x_4 \in [-0.2, 0.2]$ for 1 s with a control period of 0.05 s [20]. Since the control step size is 0.05 s, the simulation of 1 s to verify the safety property boils down to simulating the plant-control loop for 20 iterative steps. The NN controller takes $\bar{\mathbf{x}}$ as input and outputs u. An error bound of $e-5$ is considered to account for implementation errors. The goal is to prove that the system is safe and, if so, generate a controller implementation that satisfies the error bound.

As mentioned earlier, Nexus begins with the plant dynamics, initial conditions, safety property, and error bound and verifies system safety using reachability analysis with POLAR-Express [36] under real arithmetic. If the system is proven safe, Nexus *extracts the maximum and minimum values* to define the *sound ranges* for the states of `InvPend`. Next, these ranges and the error bound $e-5$ are used to quantize the NN controller into a fixed-point mixed-precision C++ implementation, guaranteeing the error bound with

Aster [24]. This is achieved through the extended code generation developed within Nexus for automated parallelization in the NN code. Therefore, this final implementation/code of the NN controller guarantees that the quantization error is not more than the predefined error bound of $1e-5$. This approach ensures that Nexus not only proves system safety but also generates implementations that maintain system safety and the error bound. The following sections will explain each of these phases in detail.

4.1.1 First Phase: The Safety Verification

In this phase, we prove the safety by simulating exact or over-approximated reachable sets containing all possible trajectories of the plant and checking if any unsafe state is reachable from a given initial state. POLAR-Express, the underlying tool in Nexus, takes the input as shown in Figure 4.1. It computes efficient functional over-approximations of reachable states using layer-by-layer propagation of Taylor model $TM(p, I)$, where p represents the polynomial used for approximation and I denotes the interval remainder, capturing the error due to over-approximation.

The safety verification process begins by initializing variables with the provided ranges. In the first iterative step, these initial ranges are used to compute the control input to the plant. Let the initial state be $\bar{\mathbf{x}}_\mathbf{0}$ and the neural network be κ. The over-approximated range of the control input $\bar{\mathbf{u}}_\mathbf{1}$ at the first iterative step is calculated as $\bar{\mathbf{u}}_\mathbf{1} = \kappa(\bar{\mathbf{x}}_\mathbf{0})$. The reachable state in the first step $\bar{\mathbf{x}}_\mathbf{1}$ is then computed by over-approximating the plant's ODEs denoted by f. The reachable state in the first step is represented as $\bar{\mathbf{x}}_\mathbf{1} = f(\bar{\mathbf{x}}_\mathbf{0}, \kappa(\bar{\mathbf{x}}_\mathbf{0}))$. The sequence of over-approximated reachable states $\bar{\mathbf{x}}_\mathbf{t}$, where t varies from 1 to T (the final step), is calculated as $\bar{\mathbf{x}}_\mathbf{t} = f(\bar{\mathbf{x}}_\mathbf{t-1}, \kappa(\bar{\mathbf{x}}_\mathbf{t-1}))$. At each step of computing the reachable state, the safety property is checked. ISince this property defines a safe state, if at any step, the reachable state extends beyond the safe regionstate, the system is deemed unsafe. If the reachable state remains within the safe region, the system is considered safe.

For the inverted pendulum, the safety property requires that x_1, x_2, x_3, and x_4 remain within the interval $[-0.2, 0.2]$. Thus, at each step, the plant variables $\bar{\mathbf{x}}$ are checked to ensure that the over-approximated state at each step falls within this interval.

Inclusion of Error due to Quantization in Safety Verification:

While POLAR-Express ensures tight over-approximations by precisely adjusting the error term I, it does not account for implementation errors that may arise when quantizing high-precision floating-point NN controllers on resource-constrained embedded platforms using low-precision fixed-point arithmetic. To address this, in Nexus, we also add a predefined quantization error (ϵ) during safety verification. In each step of the reachability analysis, the error ϵ is added to the interval remainder I of the NN output (control input to the plant). Let $u(p, I)$ denote the Taylor model of the NN output, where $I = [a, b]$ represents the error due to over-approximation. By adding $[-\epsilon, \epsilon]$ to I, it becomes $[a-\epsilon, b+\epsilon]$, thus capturing errors from both over-approximations and quantization. Thus, safety analysis is performed considering errors from both over-approximations of the reachability analysis and code quantization.

If Nexus proves that the system is safe, then we can generate a safe hardware implementation whose error is bounded ϵ by using the obtained over-approximated sound ranges of plant variables from POLAR-Express. Nexus extracts ranges ($\mathbf{R}_i$) for all plant state variables from i^{th} simulation step by adding I to the over-approximated range ensuring soundness. From the set of all $\mathbf{R}_i$ for all plant state variables, Nexus computes the max and min values to generate sound ranges for these variables.

For the inverted pendulum, the NN controller takes $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4]$ as the input and has one hidden layer with 10 ReLU-activated neurons, and one output neuron that produces the value of u, hence, the size of the NN is $4 \times 10 \times 1$. Nexus runs POLAR-Express in the background for the safety analysis of this plant-control system for 20 simulation steps considering the initial conditions and safety property with an error of $e-5$. At the end of the simulation, it declares the above property as safe and generates the following ranges for the plant state variables: $x_1 \in [0.07, 0.11]$, $x_2 \in [-0.05, 0.05]$, $x_3 \in [-0.01, 0.00]$, and $x_4 \in [-0.12, 0.02]$.

4.1.2 Second Phase: The Sound NN Quantization

In the next phase of Nexus, the goal is to quantize the safe real-valued NN controllers to generate a mixed-precision fixed-point (represented by the total number of bits and the binary point position deciding the number of fractional bits) code that can be run efficiently in embedded systems.

For this purpose, Nexus utilizes the ranges of the NN input variables generated in the first phase and the error bound to generate an implementation by solving an optimization problem. This optimization problem aims to minimize the number of bits required for all variables and constants in the implementation while remaining within the error bound. It reports infeasibility if the optimization problem cannot be solved, i.e., generating an implementation with a max of 32 bits is impossible. Nexus internally uses Aster to create the problem instance, solve it, and generate a complete C++ implementation that can be directly compiled using commercial HLS compilers like Xilinx Vitis HLS [2]. Consequently, an FPGA design can be synthesized, and the simulated running time (or latency) can be obtained in terms of machine cycles.

For the inverted pendulum, we have used the ranges generated in the first phase and configured Nexus to use a range $[10, 32]$ of fractional bits for all variables and an initial error 2^{-32} and were successful in generating an implementation.

Extended Code Generation:

As mentioned before, Nexus internally uses Aster for the second phase. Aster generates implementations with both fully *unrolled* code (where matrices and vectors are converted into scalar variables) and a *looped* version that keeps the data structures intact. However, it does not leverage the inherent parallelism available in custom hardware like FPGAs. Nexus extends the vanilla code generation by adding directives and nested looped constructs to enable automated parallelism in standard HLS compilers, synthesizing a parallelized hardware design that reduces latency.

Algorithm 1 Nexus's Code Generation

```
def code_gen(in x̄, weight W, bias b, act α, lyr L):
  in ← x̄
  for l ← 2 to L: tmp, sum, out ← ∅
    for i ← neurons in layer l: // parallelized
      for j ← neurons in layer l − 1: // parallelized
        tmp_j ← w^l_ij · in_j
      sum_i ← b^l_i
      for j ← neurons in layer l − 1:
          sum_i ← sum_i + tmp_j
      out_i ← α_l(sum_i)
    in ← out
```

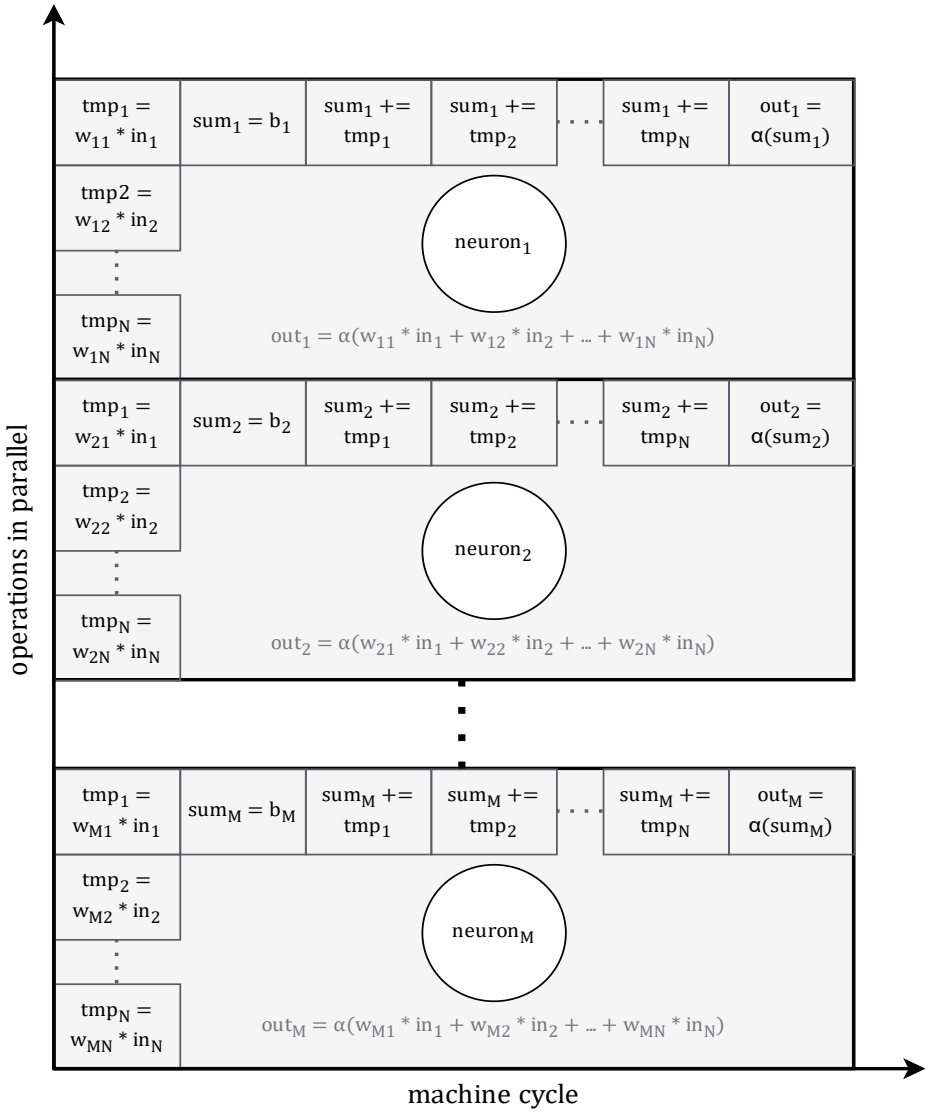

Figure 4.2: Parallelization of NN computations at a hidden layer with M neurons, getting input from the previous layer with N neurons

We present the extended code generation in Algorithm 1. The algorithm takes the input vector $\bar{\mathbf{x}}$ and the information about NN such as weights, biases, number of layers, and the activation functions of each layer. It generates the NN function in C++ using a nested-loop format with directives for parallelization. The constants (weights and biases) and variables are allocated an optimum number of bits obtained from Aster. In a neural network, each layer depends on the previous layer, requiring serial computation across layers i.e., a layer has to wait till the computation of the previous layer is completed. However, within each layer, the computation of each neuron's output is independent and can be parallelized (line 4). Additionally, the multiplications of weights and inputs within each neuron are independent and can be parallelized (line 5). Thus, Nexus returns a C++ code with nested loops optimized for parallel execution.

For instance, consider an NN controller with an input layer of size N, a hidden layer of size M, and an output layer. Each neuron in the hidden layer receives inputs from the previous layer (i.e., N inputs) and computes its output out_i using the ReLU activation function: $out_i = \text{ReLU}(w_{i_1}x_1 + w_{i_2}x_2 + \cdots + w_{i_N}x_N + b_i)$, where $1 \leq i \leq M$. The computation of each neuron's output, $out_1, out_2, \ldots, out_M$, is independent and can be parallelized. Additionally, the multiplications within each neuron, $w_{i_1}x_1, w_{i_2}x_2, \cdots, w_{i_N}x_N$, are also independent of each other and can be parallelized to further reduce latency.

The operations involved in computing the output of each neuron in the hidden layer are illustrated in Figure 4.2. For example, to calculate out_1, we perform the following operation: $out_1 = \alpha(w_{11} * in_1 + w_{12} * in_2 + \cdots + w_{1N} * in_N)$. The multiplications of weights and inputs $tmp_1 = w_{11} * in_1$, $tmp_2 = w_{12} * in_2$, $\ldots$, $tmp_N = w_{1N} * in_N$ can be scheduled within a single machine cycle, as they are independent of each other. Similarly, the multiplications for other neurons can also be scheduled in the same machine cycle because each neuron's output calculation is independent of the others. After completing the multiplications, the results can be summed. The summation operations, while dependent, can be scheduled serially over consecutive machine cycles. However, the summation and activation functions for each neuron can be processed in parallel since the neurons are independent of one another. This scheduling of operations parallelizes both the neuron calculations and the multiplications within each neuron. A High-Level Synthesis (HLS) compiler can identify these parallelisms and generate an efficient parallel implementation, thereby, significantly reducing latency.

To achieve this, we employ *three nested for-loops*: the first for-loop (line 4) iterates over all neurons in a layer to compute out_i, the second for-loop (line 5) iterates over each multiplication term $w_{i_j}.in_j$, and the third for-loop (line 8) handles the summation of these terms $w_{i_j}.in_j$ within each neuron, including bias addition.

For our running example of `InvPend`, Nexus's nested looped code achieves a latency of 14 cycles, outperforming vanilla Aster's unrolled code with 16 cycles and looped code with 18 cycles.

4.2 Nexus Tool Architecture

The complete pipeline of Nexus is shown in Figure 4.3. The input format, output format and code workflow are explained in the following sub-sections. We take our running example of `InvPend` to explain the input and output formats.

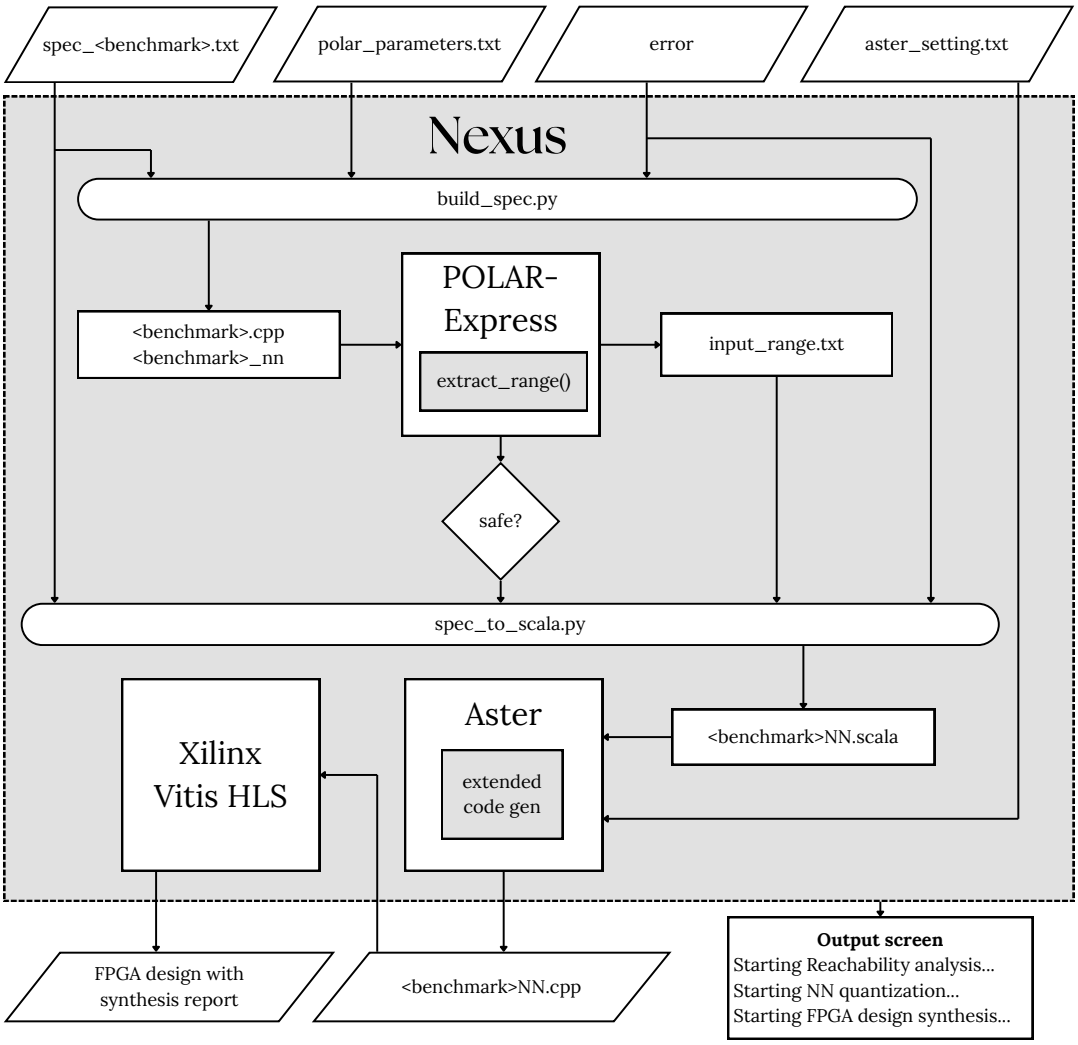

Figure 4.3: Full Pipeline Details of Nexus

4.2.1 Input Format

In Nexus, we have provided a user interface for giving all the inputs required to generate the safe hardware implementation of an NN controller. There are four inputs to Nexus as shown in 4.3. The details of these four inputs are given next.

a) `spec_<benchmark>.txt` This file contains all the information about the plant and controller and has the following format for giving the information.

```
/*plant*/ // contains all plant information
#vars // plant variables along with initial conditions
```

```
6 // number of plant variables
x1=[0.1,0.11]
x2=0.0
x3=0.0
x4=0.0

#dynamics // plant dynamics
x2
0.0043*x4 - 2.75 * x3 + 1.94 * u - 10.95 * x2
x4
28.58 * x3 - 0.044 * x4 - 4.44 * u + 24.92 * x2
0

/*model parameters*/
#steps // number of iterative steps
20
#control steps
0.05 // control period in seconds

/*safety-property*/
8 // number of safety properties
x1 - 0.2<=0 // take all the terms to left hand side
x2 - 0.2<=0
x3 - 0.2<=0
x3 - 0.2<=0
-x1 - 0.2<=0
-x2 - 0.2<=0
-x3 - 0.2<=0
-x4 - 0.2<=0

/*plant-NN*/ // neural network controller
#in-vars
4 // number of input variables of the controller
x1
x2
x3
x4

#out-vars
1 // number of output variables of the controller
u
u=0 // initial control input

#hid-layers
1 // number of hidden layers
10 // number of neurons in the first hidden layer

#act-funcs
relu // activation function of first hidden layer
linear // activation function of the output layer

#weights // weights of neural network controller starting from the first layer
Matrix(((-0.87796504,0.49901101,0.71454037,0.60627043),
(-0.12559077,-0.71710082,0.25209344,0.06111571),
(-0.34278285,-0.06739243,-0.6965151,-1.0038189),
(0.12920291,-0.69232796,0.14556374,0.41597126),
(1.53150904,2.29478818,3.39869284,0.28470497),
(-0.7424501,0.61288627,0.806291,0.31396899),
(-0.08479985,-1.35184861,-6.23189326,-1.85929063),
(-1.04077445,0.36186145,-0.11679802,-0.3906688),
(0.48112712,-0.46653835,-1.86325403,-0.79252175),
```

```
(0.63784228,-0.53025377,-1.09627429,-0.55679737)))

Matrix
    (((0.78713543,-0.6175272,-0.771196974,0.876145832,4.29984875,-0.778517245,
     -6.50494973,3.30555729e-12,-1.83581283,-0.812065464)))

#bias // bias of neural network controller starting from the first layer
Vector((-0.69577008,-0.59025015,0.85458828,-0.96125374,2.09416249,-0.9418259,
    1.22624701,-0.25044766,0.5365559,0.40367892))

Vector((0.94397898))

/*print*/ // printing of variables (optional)
0 // number of variables to print
x1
x2
x3
x4

/*print-remainder*/
OFF // printing of interval remainder of NN input and output variables (
    optional)
```

b) `polar_parameters.txt` This file contains all the specific parameters used by POLAR-Express to control various aspects of its functionality. We also include options to adjust parameters for the Flowstar tool utilized for Taylor arithmetic at the backend of POLAR-Express.

```
/*polar parameters*/
#taylor order
3 // order of Taylor polynomial used for approximation
#bernstein order
3 // order of Bernstein polynomial used for approximation
#partition num
10 // number of partitions on the interval for Bernstein approximation
#neuron_approx_type
Mix // uses both Taylor and Bernstein for ReLU approximation. 'Taylor' and '
    Berns' to use only one of them
#remainder_type
Symbolic // uses symbolic remainders. 'Concrete' does not use symbolic
    remainders
#num_threads
-1 // enable serial computation only. Set to a value greater than 1 to enable
    parallel computation in POLAR-Express

/*Flowstar parameters*/
#cutoff_threshold
1e-9 // polynomial terms below this threshold are removed and added to the
    interval remainder
#flowpipe_step_size
0.01 // time interval to calculate the flowpipe, should be less than or equal
    to the control period (i.e., control step size)
#symbolic_queue_size
100 // maximum number of terms to keep symbolically while using symbolic
    remainders
```

c) aster_setting.txt This file specifies the constraints on the number of bits allocated for the fixed-point implementation generated by Aster.

```
minLen=10 // minimum number of bits allowed
maxLen=32 // maximum number of bits allowed
initLen=32 // initial number of bits to start with
```

d) error This is the error under which the safety verification and quantization are bounded. The error primarily accounts for the over-approximation error due to safety verification and the error due to the quantization of the NN controller during implementation.

4.2.2 Output Format

Nexus takes all the inputs and generates <benchmark>NN.cpp which is the safe implementation NN controller. The output screen provides updates on each running step in the user terminal. The output folder of Nexus contains <benchmark>NN.cpp along with the FPGA design synthesis report.

<benchmark>NN.cpp This file contains the quantized neural network controller function for InvPend, generated by Nexus' extended code generation in C++ which can be compiled directly by the HLS compiler. The pragmas on lines 5 and 6 of the following code snippet provide directives to the HLS compiler to exploit the inherent parallelism within the code. The allocation of bits varies for constants (weights and biases), variables, and intermediate (temporary) variables. The inverted pendulum controller includes two weight matrices (lines 7 and 9) and two bias vectors (lines 11 and 13), each with different bit allocations. Similarly, the first layer (lines 15-27) and the second layer, which is the output layer (lines 29-38), have different bit allocations. The nested loops generated by Nexus' extended code generation can be seen in lines 31-38 for the second layer and in lines 17-27 for the first layer, where _dot_tmp1 is written sequentially because variables x_0, x_1, x_2, and x_3 have different data types. Since the operations (lines 18-21) are independent, they are also parallelized.

```
1 #include <hls_math.h>
2 #include < ap_fixed .h>
3
4 void nn1( ap_fixed <33,1> x_0,  ap_fixed <33,1> x_1,  ap_fixed <33,1> x_2,  ap_fixed
      <33,1> x_3,  ap_fixed <32,9> _result[1]) {
5 #pragma HLS PIPELINE
6 #pragma HLS ARRAY_PARTITION variable=_result complete
7    ap_fixed <27,4> weights1[10][4] =
      {{-0.87796504,0.49901101,0.71454037,0.60627043},
      {-0.12559077,-0.71710082,0.25209344,0.06111571},
      {-0.34278285,-0.06739243,-0.6965151,-1.0038189},
      {0.12920291,-0.69232796,0.14556374,0.41597126},
      {1.53150904,2.29478818,3.39869284,0.28470497},
      {-0.7424501,0.61288627,0.806291,0.31396899},
      {-0.08479985,-1.35184861,-6.23189326,-1.85929063},
      {-1.04077445,0.36186145,-0.11679802,-0.3906688},
      {0.48112712,-0.46653835,-1.86325403,-0.79252175},
      {0.63784228,-0.53025377,-1.09627429,-0.55679737}};
```

```
8
9    ap_fixed <32,4> weights2[1][10] =
      {{0.78713543,-0.6175272,-0.771196974,0.876145832,
      4.29984875,-0.778517245,-6.50494973,3.30555729e
      -12,-1.83581283,-0.812065464}};
10
11   ap_fixed <28,3> bias1[10] = {-0.69577008,-0.59025015,0.85458828,-0.96125374,
      2.09416249,-0.9418259,1.22624701,-0.25044766,0.5365559,0.40367892};
12
13   ap_fixed <32,1> bias2[1] = {0.94397898};
14
15   ap_fixed <28,3> layer1[10];
16   ap_fixed <28,3> _dot_tmp1[4];
17   for (int i = 0; i < 10; i++) {
18     _dot_tmp1[0] = weights1[i][0] * x_0;
19     _dot_tmp1[1] = weights1[i][1] * x_1;
20     _dot_tmp1[2] = weights1[i][2] * x_2;
21     _dot_tmp1[3] = weights1[i][3] * x_3;
22      ap_fixed <28,3> _bias_tmp = bias1[i];
23     for (int k = 0; k < 4; k++) {
24       _bias_tmp = _bias_tmp + _dot_tmp1[k];
25     }
26     layer1[i] = fmax(0, _bias_tmp);
27   }
28
29   ap_fixed <32,9> layer2[1];
30   ap_fixed <32,9> _dot_tmp2[10];
31   for (int i = 0; i < 1; i++) { // parallelized
32     for (int j = 0; j < 10; j++) { // parallelized
33       _dot_tmp2[j] = weights2[i][j] * layer1[j];
34     }
35      ap_fixed <32,9> _bias_tmp = bias2[i];
36     for (int k = 0; k < 10; k++) {
37       _bias_tmp = _bias_tmp + _dot_tmp2[k];
38     }
39     layer2[i] = _bias_tmp;
40   }
41
42   _result[0] = layer2[0]; // output of controller
43 }
```

4.2.3 Code Workflow

a) `build_spec.py` The specification file input format is designed for easy interfacing but cannot be directly given to POLAR-Express. This Python script loads all information into instances of the plant and controller classes. It then creates `<benchmark>.cpp` and `<benchmark>_nn` files in the POLAR-Express format. Using these files, POLAR-Express performs reachability analysis.

- `<benchmark>.cpp`: A C++ file containing plant information, safety properties, and an error. This file calls the Taylor arithmetic modules and functions.
- `<benchmark>_nn`: A file containing controller information such as inputs, outputs, hidden layer details, weights, and biases.

b) `extract_range()` This function is implemented within the Taylor arithmetic modules of the POLAR Express. It extracts the NN input ranges for all iterative steps and generates a sound NN input range, `input_range.txt`, for the controller which will be used in the second phase during quantization.

c) `Safety Check` Before proceeding to the quantization phase, a safety check is performed. If the system is found to be unsafe, the workflow stops. If the system is safe, the process continues to quantization to generate a safe hardware implementation.

d) `spec_to_scala.py` This Python script loads the specification file into an instance of the controller class and creates a `<benchmark>NN.scala` which is a suitable format for Aster. The `<benchmark>NN.scala` file contains neural network information, input ranges, and error. Using this, Aster can generate a safe NN code, `<benchmark>NN.cpp`. In the next step, Xilinx Vitis HLS takes `<benchmark>NN.cpp` and synthesizes an FPGA design along with a synthesis report which contains performance metrics such as latency in cycles. The detail of this report is explained in the next chapter.

Chapter 5

Experimental Evaluation

This chapter provides a comprehensive evaluation of Nexus' in generating safe precision-aware hardware implementation. The experiments were conducted using standard 12 neural network (NN) controller benchmarks and their corresponding plant dynamics. The objective of this evaluation is to show the safety verification and safe code generation capabilities of Nexus, particularly for larger benchmarks with substantial neural network sizes. We also show Nexus's extended code generation is better than Aster's code generation in terms of latency and design synthesis time.

5.1 Benchmarks

This section presents a collection of standard cyber-physical system (CPS) benchmarks used for evaluating and comparing with state-of-the-art techniques. These benchmarks are sourced from the competition at the ARCH (Applied Verification for Continuous and Hybrid Systems) workshop from the years 2019 [25] and 2020 [20]. These benchmarks represent real-world control problems with varying degrees of complexity, non-linearity, and dimensionality. Each benchmark provides the plant dynamics, initial conditions, neural network controller, control step size (i.e., control period), simulation time, and safety/target properties to be verified. The controllers are feed-forward neural networks with ReLU activations trained for the specific control tasks.

5.1.1 Inverted Pendulum

An inverted pendulum is a widely recognized system frequently utilized as a benchmark in control theory. This system involves a pendulum mounted on a cart, which stands upright and is inherently unstable without a controller. The objective of verifying this benchmark is to demonstrate that the pendulum eventually stabilizes in the upright position and remains there. The plant state information is represented by a 4-parameter vector $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4]$ with initial conditions as $x_1 \in [0.1, 0.11]$ and $x_2 = x_3 = x_4 = 0$. The control

input u and time t is set to 0 initially. The equations governing the system are:

$$\begin{aligned}\dot{x}_1 &= x_2\\ \dot{x}_2 &= 0.0043x_4 - 2.75x_3 + 1.94u - 10.95x_2\\ \dot{x}_3 &= x_4\\ \dot{x}_4 &= 28.58x_3 - 0.044x_4 + 4.44u + 24.92x_2\end{aligned}$$

The safety property requires that $x_1, x_2, x_3, x_4 \in [-0.2, 0.2]$ for the following 1 s. The NN controller takes $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4]$ as the input and has one hidden layer with 10 ReLU-activated neurons, and one output neuron that produces the value of u. The architecture of the NN controller is $4 \times 10 \times 1$. The control step size is 0.05 s. We have to simulate the plant-control loop for 1 s, which is 20 iterative steps.

5.1.2 Mountain Car

In this benchmark, a vehicle encounters the task of ascending a steep incline with inadequate power. To overcome this challenge, the vehicle employs a strategic manoeuvre: ascending a slope behind it to gather momentum before climbing up the primary incline. The objective of this benchmark is to develop a controller capable of analyzing the vehicle's position and velocity to generate an appropriate acceleration command. The plant state information is represented by a 2-parameter vector $\bar{\mathbf{x}} = [x_0, x_1]$ with initial conditions as $x_0 \in [-0.53, -0.5]$ and $x_1 = 0$. The control input u and time t are set to 0 initially. The equations governing the system are as follows.

$$\begin{aligned}\dot{x}_0 &= x_1\\ \dot{x}_1 &= 0.0015 * u - 0.0025 * cos(3 * x_0)\end{aligned}$$

The safety property requires that $x_0 \in [-1.2, 0.6]$ and $x_1 \in [-0.07, 0.07]$ at all times. The target property requires that, within 95 steps, $x_0 >= 0.2$ and $x_1 >= 0$, which means that the vehicle should cross $x_0 = 0.2$ and maintain a positive velocity along the x-direction. The NN controller takes $\bar{\mathbf{x}} = [x_0, x_1]$ as the input and has two hidden layers with 16 ReLU-activated neurons each, and one output neuron that produces the value of u. The architecture of the NN controller is $2 \times 16 \times 16 \times 1$. The control step size is 1 s. We have to run the simulation of the plant-control loop for 95 iterative steps.

5.1.3 Single Pendulum

This benchmark presents a classic pendulum setup, where a ball of mass m is affixed to a massless beam of length L. The beam is subjected to an applied torque T, while accounting for the presence of viscous friction characterized by coefficient c. The governing equation of motion is expressed as:

$$\ddot{\theta} = \frac{g}{L}\sin\theta + \frac{1}{mL^2}\left(T - c\dot{\theta}\right)$$

Here, θ denotes the angle formed by the link with the vertical axis, and $\dot{\theta}$ denotes the angular velocity. The state vector comprises the variables θ and $\dot{\theta}$. The model involves constants, as follows:

$$m = 0.5, \quad L = 0.5, \quad c = 0, \quad g = 1.0$$

For simplicity and uniformity, we represent θ as x_0, $\dot{\theta}$ as x_1 and T as u. The plant state information is represented by a 2-parameter vector $\bar{\mathbf{x}} = [x_0, x_1]$ with initial conditions as $x_0 = 1.2$ and $x_1 = [0.0, 0.2]$. The control input u and time t are set to 0 initially. The simplified dynamics of the system are as follows.

$$\dot{x}_0 = x_1$$
$$\dot{x}_1 = 2 * sin(x_0) + 8 * u$$

The safety property requires that $x_0 \in [0, 1]$ when $t \in [0.5, 1]$. The NN controller takes $\bar{\mathbf{x}} = [x_0, x_1]$ as the input and has two hidden layers with 25 ReLU-activated neurons each, and one output neuron that produces the value of u. The architecture of the NN controller is $2 \times 25 \times 25 \times 1$. The control step size is 0.05 s. We have to simulate the plant-control loop for 1 s, which is 20 iterative steps.

5.1.4 Double Pendulum

The double pendulum comprises two connected links with equal point masses m at their ends, suspended from a pivot. The links are controlled by torques T_1 and T_2, while accounting for viscous friction with coefficient c. The governing equations of motion are as follows:

$$2\ddot{\theta}_1 + \ddot{\theta}_2 \cos(\theta_2 - \theta_1) - \dot{\theta}_1^2 \sin(\theta_2 - \theta_1) - \frac{2g}{L} \sin\theta_1 + \frac{c}{mL^2}\dot{\theta}_1 = \frac{T_1}{mL^2}$$

$$\ddot{\theta}_1 \cos(\theta_2 - \theta_1) + \ddot{\theta}_2 + \dot{\theta}_2^2 \sin(\theta_2 - \theta_1) - \frac{g}{L} \sin\theta_2 + \frac{c}{mL^2}\dot{\theta}_2 = \frac{T_2}{mL^2}$$

Here, θ_1 and θ_2 represent the angles of the links relative to the vertical axis, and the state is denoted as $[\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$. The angular velocity and acceleration of the links are represented by $\dot{\theta}_1, \dot{\theta}_2, \ddot{\theta}_1$, and $\ddot{\theta}_2$, respectively. The gravitational acceleration is denoted by g. The model involves constants, as follows:

$$m = 0.5, \quad L = 0.5, \quad c = 0, \quad g = 1.0$$

For simplicity and uniformity, we represent θ_1 as x_0, θ_2 as x_1, $\dot{\theta}_1$ as x_2, $\dot{\theta}_2$ as x_3, T_1 as u_0 and T_2 as u_1. The plant state information is represented by a 4-parameter vector $\bar{\mathbf{x}} = [x_0, x_1, x_2, x_3]$. The linearized dynamics of the system are given as follows.

$$\dot{x}_0 = x_2$$
$$\dot{x}_1 = x_3$$
$$\dot{x}_2 = \cos(x_0 - x_1) \left(\frac{x_2^2 \sin(x_0 - x_1) - \cos(x_0 - x_1) \left(2\sin(x_0) - \frac{x_3^2 \sin(x_0 - x_1)}{2} + 4u_0 \right)}{\cos^2(x_0 - x_1) - 2} \right)$$
$$- \cos(x_0 - x_1) \left(\frac{2\sin(x_1) + 8u_1}{\cos^2(x_0 - x_1) - 2} \right) - \frac{x_3^2 \sin(x_0 - x_1)}{2} + 2\sin(x_0) + 4u_0$$
$$\dot{x}_3 = - \left(\frac{x_2^2 \sin(x_0 - x_1) - \cos(x_0 - x_1) \left(2\sin(x_0) - \frac{x_3^2 \sin(x_0 - x_1)}{2} + 4u_0 \right) + 2\sin(x_1) + 8u_1}{\cos^2(x_0 - x_1) \cdot 0.5 - 1} \right)$$

Double Pendulum Non Robust V1 The initial conditions are $x_0 = x_1 = x_2 = x_3 = 1.3$. The control inputs u_0, u_1 and time t are set to 0 initially. The safety property requires that $x_0, x_1, x_2, x_3 \in [-1.5, 2]$ at all times.

Double Pendulum Robust V2 The initial conditions are $x_0 = x_1 = x_2 = x_3 = 1$. The control inputs u_0, u_1 and time t are set to 0 initially. The safety property requires that $x_0, x_1, x_2, x_3 \in [-1.5, 1.5]$ at all times.

The NN controller architecture and control step size are the same for both versions V1 and V2, except for the parameters of the neural network. The NN controller takes $\bar{\mathbf{x}} = [x_0, x_1, x_2, x_3]$ as the input, two hidden layers with 25 ReLU-activated neurons each and one output neuron that produces the value of u. The architecture of the NN controller is $4 \times 25 \times 25 \times 1$. The control step size is 0.05 s. We have to simulate the plant-control loop for 1 s, which is 20 iterative steps.

5.1.5 Adaptive Cruise Control

The Adaptive Cruise Control (ACC) benchmark consists of two vehicles, an ego car and a lead car moving in the same lane. The ego car is equipped with an adaptive cruise control system and it is moving behind the lead car. This system utilizes a radar sensor to measure the relative distance (D_{rel}) and relative velocity (V_{rel}) between the ego vehicle and the lead vehicle. The system operates in two distinct modes: speed control and spacing control. In speed control mode, the ego vehicle maintains a preset speed of $V_{set} = 30$. Conversely, in spacing control mode, the ego vehicle aims to maintain a safe distance (D_{safe}) from the lead vehicle. The system switches between these modes based on the following conditions:

$$\text{Mode} = \begin{cases} \text{Speed Control} & \text{if } D_{rel} \geq D_{safe}. \\ \text{Spacing Control} & \text{otherwise.} \end{cases}$$

Neural network adaptive cruise controllers of varying sizes are trained to replace the existing Model Predictive Control (MPC) controller. The control period is set to 0.1 seconds. The dynamics of the benchmark is as follows.

$$\begin{aligned} \dot{x}_{\text{lead}}(t) &= v_{\text{lead}}(t), \\ \dot{v}_{\text{lead}}(t) &= \gamma_{\text{lead}}, \\ \dot{\gamma}_{\text{lead}}(t) &= -2\gamma_{\text{lead}}(t) + 2a_{\text{lead}} - \mu v_{\text{lead}}^2(t), \\ \dot{x}_{\text{ego}}(t) &= v_{\text{ego}}(t), \\ \dot{v}_{\text{ego}}(t) &= \gamma_{\text{ego}}, \\ \dot{\gamma}_{\text{ego}}(t) &= -2\gamma_{\text{ego}}(t) + 2a_{\text{ego}} - \mu v_{\text{ego}}^2(t) \end{aligned}$$

Mathematically, the safety property is represented as $D_{\text{rel}} \geq D_{\text{safe}}$, where $D_{\text{rel}} = x_{\text{lead}} - x_{\text{ego}}$, $D_{\text{safe}} = D_{\text{Default}} + T_{\text{gap}} * v_{\text{ego}}$ and $D_{\text{Default}} = 10$. Simplifying, we get:

$$-x_{\text{lead}} + x_{\text{ego}} + 10 + T_{\text{gap}} \cdot v_{\text{ego}} \leq 0. \tag{5.1}$$

For simplicity and uniformity, we represent V_{set} as x_0, T_{gap} as x_1, x_{lead} as x_2, x_{ego} as x_3, v_{lead} as x_4, v_{ego} as x_5, a_{lead} as x_6, and a_{ego} as x_7. The plant state information is represented

by an 8-parameter vector $\bar{\mathbf{x}} = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]$ with initial conditions as $x_0 = 30$, $x_1 = 1.4$, $x_2 \in [90, 110]$, $x_3 \in [10, 11]$, $x_4 \in [32, 32.2]$, $x_5 \in [30, 30.2]$, and $x_6 = x_7 = 0$. The control input u and time t are initially set to 0. The simplified dynamics of the system are:

$$\begin{aligned}
\dot{x}_0 &= 0, \\
\dot{x}_1 &= 0, \\
\dot{x}_2 &= x_4, \\
\dot{x}_3 &= x_5, \\
\dot{x}_4 &= x_6, \\
\dot{x}_5 &= x_7, \\
\dot{x}_6 &= -4 - 2x_6 - 0.0001x_4^2, \\
\dot{x}_7 &= 2u - 2x_7 - 0.0001x_5^2.
\end{aligned}$$

From 5.1, the safety property requires that $-x_2 + x_3 + 10 + x_1 x_5 \leq 0$ for a time period of 5 s. The NN controller takes $\bar{\mathbf{x}} = [x_0, x_1, x_5, x_2 - x_3, x_4 - x_5]$ as input and produces u as output. NN controllers with different numbers of hidden layers are trained: ACC3, ACC5, ACC7, and ACC10 with 3, 5, 7, and 10 hidden layers, respectively, each with 20 ReLU-activated neurons. The control step size is 0.1 s for all ACC versions. We have to simulate the plant-control loop for 5 s, which is 50 iterative steps.

5.1.6 Unicycle

This benchmark represents the Unicycle model of a car in a 2-dimensional plane. The plant state information is represented by a 4-parameter vector $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4]$ with initial conditions as $x_1 \in [9.5, 9.55]$, $x_2 \in [-4.5, -4.45]$, $x_3 \in [2.1, 2.11]$ and $x_4 \in [1.5, 1.51]$. The control inputs u_1, u_2 and time t are initially set to 0. The equations governing the system are:

$$\begin{aligned}
\dot{x}_1 &= x_4 \cos(x_3) \\
\dot{x}_2 &= x_4 \sin(x_3) \\
\dot{x}_3 &= u_2 \\
\dot{x}_4 &= u_1 + w
\end{aligned}$$

where $w \in [-0.0001, 0.0001]$ introduces noise in the control input. The target property requires that, after 10 seconds, $x_1 \in [-0.6, 0.6]$, $x_2 \in [-0.2, 0.2]$, $x_3 \in [-0.06, 0.06]$, and $x_4 \in [-0.3, 0.3]$. The NN controller takes $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4]$ as input, one hidden layer with 500 ReLU-activated neurons and two output neurons that produce the values of u_1 and u_2. The architecture of the NN controller is $4 \times 500 \times 2$. The control step size is 0.2 s. We have to simulate the plant-control loop for 10 s, which is 50 iterative steps.

5.1.7 Airplane

This benchmark represents a simplified model of an airplane. The plant state information is represented by a 12-parameter vector $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}]$ with initial conditions $x_1 = x_2 = x_3 = x_{10} = x_{11} = x_{12} = 0$ and $x_4 = x_5 = x_6 = x_7 =$

$x_8 = x_9 = 0$. The control inputs $\bar{\mathbf{u}} = [u_1, u_2, u_3, u_4, u_5, u_6]$ and time t are initially set to 0. The equations governing the system are as follows.

$$
\begin{aligned}
\dot{x}_1 &= x_6(\sin(x_7)\sin(x_9) + \cos(x_7)\cos(x_9)\sin(x_8)) \\
&\quad - x_5(\cos(x_7)\sin(x_9) - \cos(x_9)\sin(x_7)\sin(x_8)) + x_4\cos(x_8)\cos(x_9) \\
\dot{x}_2 &= x_5(\cos(x_7)\cos(x_9) + \sin(x_7)\sin(x_8)\sin(x_9)) \\
&\quad - x_6(\cos(x_9)\sin(x_7) - \cos(x_7)\sin(x_8)\sin(x_9)) + x_4\cos(x_8)\sin(x_9) \\
\dot{x}_3 &= x_6\cos(x_7)\cos(x_8) - x_4\sin(x_8) + x_5\cos(x_8)\sin(x_7) \\
\dot{x}_4 &= u_1 - \sin(x_8) + x_5x_{10} - x_6x_{12} \\
\dot{x}_5 &= u_2 + \cos(x_8)\sin(x_7) - x_4x_{10} + x_6x_{11} \\
\dot{x}_6 &= u_3 + \cos(x_7)\cos(x_8) + x_4x_{12} - x_5x_{11} \\
\dot{x}_7 &= x_{11} + \frac{x_{10}\cos(x_7)\sin(x_8)}{\cos(x_8)} + \frac{x_{12}\sin(x_7)\sin(x_8)}{\cos(x_8)} \\
\dot{x}_8 &= x_{12}\cos(x_7) - x_{10}\sin(x_7) \\
\dot{x}_9 &= \frac{x_{10}\cos(x_7)}{\cos(x_8)} + \frac{x_{12}\sin(x_7)}{\cos(x_8)} \\
\dot{x}_{10} &= u_6 \\
\dot{x}_{11} &= u_4 \\
\dot{x}_{12} &= u_5
\end{aligned}
$$

The safety property requires that $x_2 \in [-0.5, 0.5]$ and $x_7, x_8, x_9 \in [-1, 1]$ at all times. The NN controller takes $\bar{\mathbf{x}} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}]$ as input. It consists of three hidden layers with 20, 100, and 100 ReLU-activated neurons respectively, and six output neurons that produce the values of $\bar{\mathbf{u}} = [u_1, u_2, u_3, u_4, u_5, u_6]$. The architecture of the NN controller is $12 \times 100 \times 100 \times 20 \times 6$. The control step size is 0.1 s. We have to simulate the plant-control loop for 2 s, which is 20 iterative steps.

5.1.8 TORA

This benchmark represents the TORA (Translational Oscillations by a Rotational Actuator) model. The system consists of a cart attached to a wall with a spring, free to move on a frictionless surface. The cart has an internal weight attached to a rotating arm, which serves as the control input to stabilize the cart. The plant state information is represented by a 4-parameter vector $\bar{\mathbf{x}} = [x_0, x_1, x_2, x_3]$ with initial conditions of the individual parameters as $x_0 \in [0.6, 0.7]$, $x_1 \in [-0.7, -0.6]$, $x_2 \in [-0.4, -0.3]$, and $x_3 \in [0.5, 0.6]$. The control input u and time t are initially set to 0. The equations governing the system are as follows.

$$
\begin{aligned}
\dot{x}_0 &= x_1 \\
\dot{x}_1 &= -x_0 + 0.1\sin(x_2) \\
\dot{x}_2 &= x_3 \\
\dot{x}_3 &= u
\end{aligned}
$$

The safety property requires that $x_0, x_1, x_2, x_3 \in [-2, 2]$ at all times. The NN controller takes $\bar{\mathbf{x}} = [x_0, x_1, x_2, x_3]$ as input, three hidden layers with 500 ReLU-activated neurons each and one output neuron that produces the value of u. The architecture of NN can

benchmarks	#plant-vars	neural controller specification		
		ctrl-step	#hid-lyr	#params
InvPend	6	0.05	1	60
MountCar	3	1.00	2	336
SglPend	4	0.05	2	775
DblPendV1	7	0.05	2	825
DblPendV2	7	0.02	2	825
ACC3	10	0.10	3	980
ACC5	10	0.10	5	1,820
ACC7	10	0.10	7	2,660
Unicycle	7	0.20	1	3,500
ACC10	10	0.10	10	3,920
Airplane	19	0.10	3	13,540
TORA	5	1.00	3	20,800

Table 5.1: Benchmark details (plant controller specifications)

represented as $4 \times 100 \times 100 \times 100 \times 1$. The control step size is 1 s. We have to simulate the plant-control loop for 20 iterative steps

These above-mentioned benchmarks controllers are abbreviated as: `InvPend` (inverted pendulum), `SglPend` (single pendulum), `DblPend` (double pendulum) nonrobust `V1` and robust `V2`, `Unicycle`, `MountCar` (mountain car), `TORA`, `Airplane`. Finally, the adaptive cruise controllers with different hidden layers as `ACC`s. All these benchmarks are sorted based on their controller size which is the total number of parameters (i.e., weights and biases) and the summary is presented in Table 5.1.

5.2 Experimental Setup

All experiments were done on an Ubuntu 20.04 running on an Intel Core i5 system with 3.3 GHz clock speed and 32 GB RAM. We used POLAR-Express (commit 13d42b0) and Aster (commit 2c991fb), downloaded from GitHub on Aug. 18, 2023, and Mar. 20, 2024, respectively. For FPGA design synthesis, we employed Xilinx's Vitis HLS [2] (version 2023.2), downloaded on Feb. 22, 2024.

At the backend of Nexus, we configured Aster with two settings and two error bounds: A for error $e-3$ and B for $e-5$. Setting A initialized the number of fractional bits to 20 (error: 2^{-20}), and setting B to 32 (error: 2^{-32}). Both settings allowed a maximum of 32 bits, with ranges of fractional bits as $[5, 32]$ for setting A and as $[10, 32]$ for setting B. A run of Nexus was allocated a 5-hour time budget.

benchmarks	error: e-3, setting A			error: e-5, setting B		
	safe	latency	time(s)	safe	latency	time(s)
InvPend	✓	12	3.15	✓	14	3.16
MountCar	✗	-	-	✓	25	66.35
SglPend	✓	23	5.16	✓	27	5.15
DblPendV1	✓	26	7.10	✓	27	6.80
DblPendV2	✓	26	5.80	✓	28	5.84
ACC3	✓	40	10.49	✓	39	10.47
ACC5	✓	inf	-	✓	63	19.78
ACC7	✓	inf	-	✓	inf	-
Unicycle	✕	-	-	✕	-	-
ACC10	✕	-	-	✕	-	-
Airplane	✗	-	-	✗	-	-
TORA	✗	-	-	✗	-	-

Table 5.2: Safety analysis (✓: safe, ✗: unsafe, ✕: analysis fails), latencies of safe controller implementations (inf: tool returns infeasible) and running times of Nexus

5.3 Safety Verification and Sound Code Generation

Table 5.2 provides an overview of all experiments done in Nexus. As expected, more benchmarks (8 out of 12) are safe with the smaller error bound of $e-5$ compared to the error bound of $e-3$ (7 out of 12). Because of the addition of the error to the interval remainder of the control input at each iterative step, the interval of variables widens, which causes the system to go beyond the safe range and become unsafe or reachability analysis terminates due to large over-approximation errors. This is observed in `MountCar` where the system is safe with the error $e-5$ but not with the error of $e-3$. However, most of the benchmarks were reported to be safe with both errors. For larger benchmarks like `Unicycle` and `ACC10`, large over-approximations error causes the reachability analysis to fail which prevented us from proving safety. The largest benchmarks, `Airplane` and `TORA`, were found unsafe given our initial conditions, safety properties, and error bounds.

For benchmarks which are proven safe in the first phase, we attempted to generate implementations with Nexus's extended code generation. Setting A, starting with a larger initial error (2^{-20}), is expected to report more infeasibility i.e., generating a sound implementation that satisfies both the error bound and the maximum bit length is impossible. This was observed for `ACC5` where we could generate an implementation with setting B but not with setting A. For `ACC7`, Nexus reported infeasible with both the settings due to the increasing over-approximation errors with the increasing number of layers.

For benchmarks where we could generate implementations, we compiled them for a standard FPGA architecture using Xilinx Vitis HLS and presented the latencies in machine cycles that the compiler reported for the final hardware implementations. As expected, the latencies of the smaller benchmarks are lower and vice versa. This is intuitive, as larger

benchmarks	latencies of implementations (cycles)					
	unrolled		looped		nested-looped	
	serial	parallel	serial	parallel	serial	parallel
InvPend	16	15	36	18	57	14
MountCar	30	30	92	38	106	25
SglPend	31	30	122	47	141	27
DblPendV1	32	31	135	50	89	27
DblPendV2	35	34	134	51	90	28
ACC3	58	58	162	65	158	39
ACC5	99	99	273	107	229	63

Table 5.3: Comparing latencies of Aster's unrolled serial and looped serial implementations with Nexus's optimized implementations (nested-looped serial and all parallels) with error $e-5$ and setting B

benchmarks require more computation time, resulting in higher latency. The implementations with the error bound of $e-5$ have slightly higher latencies due to the increased number of bits required to satisfy the smaller error bound compared to implementations with the error bound of $e-3$.

The running times of Nexus are shown in Table 5.2 (Columns 4, 7). The running time depends on various factors like the width of the initial range, the number of plant variables, controller size, etc. As expected, the running time increases with the size of the controller. The wide initial ranges of `MountCar` take more time for reachability analysis and thus report more time than other benchmarks. However, Nexus took a maximum of 20 s for the largest safe benchmark, `ACC5`.

5.4 Looped Code Generation

This section compares Nexus's extended code generation with Aster's code generation, as shown in Table 5.3. The benchmarks presented are those for which implementations were successfully generated. We refer to the code without parallelization directives as *serial* and the code with directives as *parallel*. We generated parallel versions of Aster's unrolled and looped code as well as both serial and parallel versions of Nexus's nested-loop code.

Our results show that using parallelization directives in unrolled code does not improve latencies compared to serial version due to instruction inter-dependencies, except for `InvPend` and `DblPend`. However, the compiler effectively identified parallelism in the looped and nested-looped versions, significantly reducing latency in the parallel versions compared to the serial versions, i.e., for all benchmarks, looped-parallel reported lesser latency than looped-serial and nested-looped-parallel reported lesser latency than nested-looped-serial. Moreover, unrolled serial implementations reported lower latencies, as the compiler identified instructions which are independent. The last column of Table 5.3 shows the nested-looped parallel implementations generated by Nexus's extended code

benchmarks	design synthesis time (s)					
	unrolled		looped		nested-looped	
	serial	parallel	serial	parallel	serial	parallel
InvPend	27.75	26.84	24.85	25.93	23.41	24.37
MountCar	43.37	43.41	28.33	34.81	24.43	31.32
SglPend	72.37	72.89	35.30	51.08	24.72	46.16
DblPendV1	77.90	77.34	35.30	51.22	25.46	44.34
DblPendV2	78.97	76.10	36.65	52.63	25.49	43.21
ACC3	94.06	94.64	38.54	59.29	26.26	49.91
ACC5	199.12	191.53	50.34	99.26	33.62	98.23

Table 5.4: Comparing design synthesis time of Aster's unrolled serial and looped serial implementations with Nexus's optimized implementations (nested-looped serial and all parallels) with error $e-5$ and setting B

benchmarks	Aster (serial)		Nexus (parallel)	
	unrolled	looped	looped	nested-looped
Unicycle	29	864	265	18
Airplane	75	510	152	43
TORA	✗	604	186	41

Table 5.5: Comparing latencies of Aster's and Nexus's implementations for larger benchmarks (✗: Xilinx fails)

generation, outperform all other versions due to efficient parallelization, especially for larger benchmarks.

We also compared the design synthesis times. The nested-looped version reported less time than the looped and unrolled version due to the more compact representation of the neural network function. Serial implementations have shorter synthesis times than parallel versions due to the procedure involved in identifying parallelism by the compiler.

To demonstrate the utility of Nexus's extended code generation for larger benchmarks with larger neural network sizes, we used Nexus's extended code generation to generate implementations for the three largest unsafe benchmarks (i.e., `Unicyle, Airplane, TORA`) with the error bound of $e-3$ and Setting B. Note that Setting A was infeasible due to the large initial error. The respective result is shown in Table 5.5. Generating looped implementations is crucial for these benchmarks as serial implementations can become too large to compile (e.g., 62K lines of code for `TORA`). Nexus's parallel nested-looped code significantly outperforms Aster for all three benchmarks. The reported latencies are the lowest with Nexus's nested-looped parallel version, which shows the scalability of Nexus's extended code generation while generating efficient implementation.

Chapter 6

Conclusion

This thesis presents *Nexus*, a novel and scalable integration of reachability analysis with sound quantization for NN-controlled cyber-physical systems. We propose a method to incorporate the error due to the quantization of the NN controller during the safety verification analysis and generate sound input ranges for the NN controller. Additionally, we implement extended code generation, leveraging the inherent parallelism in neural network architecture and hardware architecture to produce parallelizable implementations. Nexus provides a simple interface for conducting safety verification and subsequent quantization, resulting in safe and efficient implementations. Our results demonstrate that Nexus can generate safe implementations when given inputs in the specified format. Nexus also produces optimized implementations with the lowest latency, especially for larger benchmarks, showcasing significant optimization potential for custom hardware like FPGAs. Currently, we focus on feed-forward NN controllers with ReLU and linear activations due to the constraints of the underlying tools. In future, we aim to enhance Nexus by supporting non-linear activations, hybrid controllers, and other networks like convolutional neural networks.

Dissemination out of this Work

- **Harikishan T S,** Sumana Ghosh and Debasmita Lohar, ***"Towards Precision-Aware Safe Neural-Controlled Cyber-Physical Systems"***, in ACM SIGBED International Conference on Embedded Software 2024 (Under review).

Bibliography

[1] Althoff, M. An introduction to cora 2015. In *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems* (2015), G. Frehse and M. Althoff, Eds., vol. 34 of *EPiC Series in Computing*, EasyChair, pp. 120–151.

[2] AMD. Vitis HLS, 2023.

[3] Åström, K. J., and Wittenmark, B. *Computer-controlled systems*. Prentice-Hall, Inc., 1997.

[4] Bak, S., Bogomolov, S., and Johnson, T. Hyst: A source transformation and translation tool for hybrid automaton models. *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015* (04 2015), 128–133.

[5] Berz, M., and Makino, K. Verified integration of odes and flows using differential algebraic methods on high-order taylor models. *Reliable Computing* (1998).

[6] Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., and Schilling, C. Juliareach: a toolbox for set-based reachability. *CoRR abs/1901.10736* (2019).

[7] Boldo, S., Gallois-Wong, D., and Hilaire, T. A correctly-rounded fixed-point-arithmetic dot-product algorithm. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)* (2020).

[8] Chen, X., Ábrahám, E., and Sankaranarayanan, S. Flow*: An Analyzer for Non-Linear Hybrid Systems. In *CAV* (2013).

[9] Dutta, S., Chen, X., and Sankaranarayanan, S. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *HSCC* (2019).

[10] Dutta, S., Jha, S., Sankaranarayanan, S., and Tiwari, A. Learning and Verification of Feedback Control Systems using Feedforward Neural Networks. *IFAC-PapersOnLine*, 16 (2018).

[11] Everett, M., Habibi, G., and How, J. P. Efficient Reachability Analysis of Closed-Loop Systems with Neural Network Controllers. In *ICRA* (2021).

[12] Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.-K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Le Bodic, P., Maher, S., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M., Schlösser, F.,

Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., and Witzig, J. *The SCIP Optimization Suite 7.0.* 2020.

[13] Ga{\ss}mann, V., and Althoff, M. Implementation of ellipsoidal operations in cora 2022. In *Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22)* (2022), G. Frehse, M. Althoff, E. Schoitsch, and J. Guiochet, Eds., vol. 90 of *EPiC Series in Computing*, EasyChair, pp. 1–17.

[14] Goldberger, B., Katz, G., Adi, Y., and Keshet, J. Minimal modifications of deep neural networks using verification. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (2020), vol. 73, pp. 260–278.

[15] Gopinath, S., Ghanathe, N., Seshadri, V., and Sharma, R. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *PLDI* (2019).

[16] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep Learning with Limited Numerical Precision. In *ICML* (2015).

[17] Huang, C., Fan, J., Li, W., Chen, X., and Zhu, Q. ReachNN: Reachability analysis of neural-network controlled systems. *ACM Transactions on Embedded Computing Systems* (2019).

[18] Ivanov, R., Weimer, J., Alur, R., Pappas, G. J., and Lee, I. Verisig: Verifying Safety Properties of Hybrid Systems with Neural Network Controllers. In *HSCC* (2019).

[19] Izycheva, A., Darulova, E., and Seidl, H. *Synthesizing Efficient Low-Precision Kernels.* 10 2019, pp. 294–313.

[20] Johnson, T. T., Lopez, D. M., Musau, P., Tran, H., Botoeva, E., Leofante, F., Maleki, A., Sidrane, C., Fan, J., and Huang, C. ARCH-COMP20 Category Report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for Continuous and Hybrid Systems Plants. In *ARCH* (2020), EPiC Series in Computing.

[21] Kumar, A., Seshadri, V., and Sharma, R. Shiftry: RNN Inference in 2KB of RAM. In *OOPSLA* (2020).

[22] Kumar, A., Seshadri, V., and Sharma, R. Shiftry: Rnn inference in 2kb of ram. *Proceedings of the ACM on Programming Languages 4* (11 2020), 1–30.

[23] Li, Y., Zhu, H., Braught, K., Shen, K., and Mitra, S. Verse: A python library for reasoning about multi-agent hybrid system scenarios, 2023.

[24] Lohar, D., Jeangoudoux, C., Volkova, A., and Darulova, E. Sound Mixed Fixed-Point Quantization of Neural Networks. *ACM-TECS* (2023).

[25] Lopez, D. M., Musau, P., Tran, H., and Johnson, T. T. Verification of Closed-loop Systems with Neural Network Controllers. *EPiC Series in Computing* (2019).

[26] Lorentz, G. *Bernstein Polynomials.* American Mathematical Society, 2013.

[27] Mistry, S., Saha, I., and Biswas, S. An milp encoding for efficient verification of quantized deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41*, 11 (2022), 4445–4456.

[28] Moore, R. E., Kearfott, R. B., and Cloud, M. J. *Introduction to Interval Analysis*. SIAM, 2009.

[29] Park, E., Kim, D., and Yoo, S. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *ICSA* (2018).

[30] Park, J., Pajic, M., Sokolsky, O., and Lee, I. Automatic Verification of Finite Precision Implementations of Linear Controllers. In *TACAS* (2017).

[31] Sharma, H., Park, J., Suda, N., Lai, L., Chau, B., Chandra, V., and Esmaeilzadeh, H. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *ISCA* (2018).

[32] Song, Z., Fu, B., Wu, F., Jiang, Z., Jiang, L., Jing, N., and Liang, X. DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration. In *ICSA* (2020).

[33] Tran, H., Cai, F., Diego, M. L., Musau, P., Johnson, T. T., and Koutsoukos, X. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM-TECS* (2019).

[34] Tran, H., Manzanas Lopez, D., Musau, P., Yang, X., Nguyen, L., Xiang, W., and Johnson, T. Star-based reachability analysis of deep neural networks. In *Formal Methods – The Next 30 Years - 3rd World Congress, FM 2019, Proceedings* (2019), M. ter Beek, A. McIver, and J. Oliveira, Eds., Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer, pp. 670–686. Publisher Copyright: © Springer Nature Switzerland AG 2019.; 23rd Symposium on Formal Methods, FM 2019, in the form of the 3rd World Congress on Formal Methods, 2019 ; Conference date: 07-10-2019 Through 11-10-2019.

[35] Tran, H.-D., Yang, X., Lopez, D. M., Musau, P., Nguyen, L. V., Xiang, W., Bak, S., and Johnson, T. T. Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems, 2020.

[36] Wang, Y., Zhou, W., Fan, J., Wang, Z., Li, J., Chen, X., Huang, C., Li, W., and Zhu, Q. POLAR-Express: Efficient and Precise Formal Reachability Analysis of Neural-Network Controlled Systems. *IEEE-TCAD* (2023).

[37] Xu, Z., Liu, Y., Qin, S., and Ming, Z. Output range analysis for feed-forward deep neural networks via linear programming. *IEEE Transactions on Reliability 72*, 3 (2023), 1191–1205.