

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Objectives	5
1.3	Structure	6
2	Formal Verification of Neural Networks	7
2.1	Neural Networks	7
2.1.1	Training a Neural Network	9
2.1.2	Layer Types	11
2.1.3	Activation Functions	12
2.1.4	Matrix Form	13
2.1.5	Summary	15
2.2	Neural Network Verification Techniques	15
2.3	Complete Techniques	19
2.3.1	Linear Programming	20
2.3.2	SMT Solvers	22
2.3.3	Interval Analysis	23
2.3.4	Symbolic Representation	25
2.3.5	Discussion	27
2.4	Incomplete Techniques	28
2.4.1	Boundary Search	29
2.4.2	Abstract Domain	30
2.4.3	Linear Relaxation	32
2.4.4	Dual Formulation	33
2.4.5	Lipschitz Constant Estimation	34
2.4.6	Discussion	34
2.5	Other Techniques	35
2.6	Tool Comparison	37
2.6.1	Overview of Experiments	38
2.6.2	Methods	39
2.6.3	Evaluation	40
2.6.4	Discussion	44
2.7	Final Considerations	45
3	Neural Networks in RoboChart	46
3.1	RoboChart	46
3.2	ANNs in RoboChart	50

3.2.1	Overview	50
3.2.2	ANN Components Metamodel	61
3.2.3	ANN Component Well-Formedness Conditions	64
3.3	Semantics Overview	68
3.3.1	CSP Overview	68
3.3.2	RoboChart Semantics Overview	69
3.4	CSP Semantics for Neural Networks	74
3.5	JCSP Validation	79
3.5.1	JCSP	79
3.5.2	ANN Component Validation	80
3.5.3	Discussion	86
3.6	Final Considerations	87
4	Verification Strategy	88
4.1	General Pattern of ANN Contracts	88
4.1.1	UTP Reactive Contracts	88
4.1.2	Reactive Contracts for ANN components	89
4.2	ANN Pattern Justification	93
4.3	Cyclic Memoryless RoboChart Controllers	95
4.4	Conformance	97
4.5	System-Level Conformance	99
4.6	Final Considerations	105
5	Conclusions	106
5.1	Summary	106
5.2	Future Work	107
A	Full Meta-model	109
B	UTP Reactive Contract Laws	111
B.1	Supporting Laws	111
B.2	Existing Laws	131
C	ANN Component Conformance Lemmas	133

Chapter 1

Introduction

Creating intelligent machines is a fascinating prospect that has the potential to affect all aspects of society. Makridakis [60] argues that the emergence of artificial intelligence is analogous to the industrial and digital revolution. *Artificial intelligence* (AI) has multiple definitions; but a strong definition, from [52], is machines that can ‘imitate human intelligent behaviour’; behaviour which entails learning, reasoning and self-correction.

Our work contributes to this vision by enabling machines involving AI to be verified. In particular, we focus on developing verified robots whose implementation involve neural networks for control. Next, we detail the motivation for this work. In Section 1.2 we describe our objectives, and Section 1.3 describes the structure of the thesis.

1.1 Motivation

Machine learning (ML) refers to algorithms that automatically detect patterns in data from a training set, and uses these patterns to predict or classify future data provided after training [52]. Machine learning algorithms accomplish this through utilising induction rules to generate models based on the training set [46].

Neural networks are effective, widely used and efficient machine-learning methods. They have been used to generate state-of-the-art results in image recognition [54], and word spotting [92]. Sudholt [92] comments that a type of neural network is able to consistently outperform all other approaches in virtually every field of computer vision. Neural networks have also been proposed for collision detection in: aircraft [47], road vehicles [12] and ships [11].

For computer vision problems, some form of machine learning is all but necessary to obtain state-of-the-art results in tractable time frames. Benalcázar [15] comments that computer vision is a problem that belongs to the field of AI. For low dimensional recognition problems, similar results have been obtained using other methods. An example is the Airborne Collision Avoidance System. In this system, dynamic programming and Markov chains were originally used before a new system was proposed using neural networks, and obtained a slightly higher, though comparable accuracy. On the other hand, the runtime was reduced and the storage space required was reduced by a factor of 1000 [47]. This is because a neural network only requires the parameters of its model to be stored instead of all interactions in a look-up table.

One of the potential applications of ML algorithms with significant impact is the design of robotic systems. Machine learning can enable robots to reason, adapt and perceive their environments with reasoning abilities closer to those of humans. This area is referred to as RAAI, the combination of Robotics, Automation and AI.

In October 2016, the UK's Council for Science and Technology sent a letter to the Prime Minister detailing the potential impact of RAAI. It comments that a recent report estimates that these technologies will have an impact on global markets of up to \$4.5 trillion per annum by 2025 [103]. They describe that RAAI will enable the design of machines that are able to perceive their environments, reason about events, make and revise plans, and control their actions.

ML has a significant role in enabling this type of functionality. As Johnson [46] comments, it would be difficult to envisage how autonomous systems could reason about their environment through collision detection and support sensor fusion without the use of machine learning.

The application areas for RAAI include: healthcare, hazardous domains such as nuclear power, autonomous vehicles and domestic assistants [62, 59]. These application areas involve interactions either with humans or with hazardous environments, so safety of these systems is a major concern. Utilising formal methods can provide evidence for the safety of robotic systems [59].

There are existing platforms for developing robotic software taking advantage of formal methods [101, 50, 37, 58] and there are tools available for verifying neural networks [49, 96, 48]. There is, however, a lack of platforms to verify robotic software that has a component implemented using a neural network.

There are two broad categories of notations used in formal approaches to verification of robotic software: general purpose languages and domain-specific languages. General purpose languages with verification support include: C, C++, Java, UML and its derivatives, SysML, AADL and Focus [101]. Domain-specific languages (DSLs) are more focused, smaller and more usable, their intended application is for a single domain.

Domain-specific modelling languages provide an efficient and flexible way to combine expertise from multiple domains of robotics [70]. In addition, Dhouib [22] comments that robotic experts face significant problems developing their applications in general purpose languages without expert knowledge of programming languages. DSLs provide a solution to this problem.

There are multiple domain-specific languages for robotics. Nordmann [70] mentions 137 DSLs, which, however, are not focused on formal verification. There are also more recent DSLs including: ArmarX, rFSM, CommonLang, RobotML, FlexBE, RoboChart and vTSL [58, 101, 85, 37]. Of these, ArmarX, FlexBE and rFSM do not contain support for formal verification, CommonLang is focused on Java-code generation, and RobotML does not contain support for the verification of non-functional properties [101].

Those with support for formal methods are vTSL and RoboChart. vTSL, however, does not contain support for timed properties [37] and only provides support for model checking. The semantics of vTSL enable translation to Promela [37], the input language of the Spin model checker [41], while RoboChart enables verification through a CSP semantics [101]. Model checking is enabled through FDR [100] and theorem proving is enabled through a predicate

relational semantics [101] that can cater for time, continuous, and probabilistic behaviour, for instance, all relevant to reason about robotics systems.

RoboChart contains support for the modelling of time properties, static and dynamic verification via testing, and interfaces to multiple verification tools, including simulation. RoboChart also has a graphical notation that provides a clear method for designing robotic software, enabling communication between engineers from various disciplines [58].

To encourage adoption by roboticists, we use RoboChart as a basis for our verification technique.

1.2 Objectives

The overall aim of this work is to enable the development of safe RAAI software. Since machine learning can enable more effective robotic autonomy and neural networks are a very effective form of machine learning, our goal is to support verification of robotic software that uses neural networks. We will, therefore, integrate support for neural network verification into RoboChart. This will enable verification of reactive properties of software that is defined in RoboChart and uses neural networks components.

To achieve our goal, we have the following objectives.

1. Extension of RoboChart’s meta-model to accommodate the definition of components that are implemented by neural networks. This will enable the definition of neural network components as part of a complete design. This extension should be compact, usable and consistent with RoboChart’s metamodel.
2. Extension of RoboChart’s semantics to accommodate such components. This enables a formal definition of their behaviour to accommodate verification techniques. These semantics should cover all RoboChart constructs to define systems involving multiple components, including standard state machines.
3. Definition of a technique that can verify the overall model. This enables the verification of system level specifications, where the existing neural network verification tools are concerned with component-level specifications. Moreover, this allows system requirements to be linked to component specifications.
4. Automation of the verification technique. The verification technique should be automated. This is because there are multiple techniques with varying scalability, applicability and precision under various situations. Automation will also take advantage of existing support for RoboChart.

Using our work, it is possible to develop verified robots whose control software is implemented using neural networks for control. This technique allows the definition and proof of rich behavioural properties of robots using RoboChart. In addition, this technique defines verifiable semantics for the entire model to provide guarantees on behaviour. Finally, our work takes advantage of existing support for RoboChart, including code generation and simulation through RoboTool.

1.3 Structure

Chapter 2 discusses background material relevant to the project: ANN verification and ANN tools. That chapter covers the fundamentals of ANNs, which include ANN training, ANN mathematical representation, and ANN types. That chapter also presents ANN verification techniques, considering both complete and incomplete approaches. Finally, that chapter discusses ANN tools; it provides a novel comparison between multiple ANN tools to inform our verification work.

In Chapter 3, we discuss how to integrate ANNs into RoboChart, including: extending RoboChart's meta-model, defining well-formedness conditions, and providing a semantics. That chapter defines how to model ANNs as components in a robotic system, instead of being treated in isolation.

Chapter 4 discusses how to verify properties of RoboChart models with integrated ANN components, as defined in Chapter 3. This involves the use of multiple tools, languages, and frameworks working in conjunction. That chapter defines a novel verification technique appropriate to verify complete robotic systems involving ANNs used to implement controllers.

Chapter 5 concludes this work by summarising our results and discussing future directions.

Chapter 2

Formal Verification of Neural Networks

This chapter discusses how formal verification has been applied to neural networks so far. In Section 2.1, we discuss the fundamentals of neural networks. Section 2.2 gives an overview of the available techniques. In Sections 2.3, 2.4, and 2.5, we discuss the available techniques for verifying neural networks. In Section 2.6, we compare the available tools for verifying neural networks. Finally, we give closing remarks in Section 2.7.

2.1 Neural Networks

Data itself has no inherent meaning to the real world. Machine learning is concerned with providing an interpretation of data. The meaning of data is encoded through labelling data. There are two broad types of learning, supervised and unsupervised learning; where in supervised learning, labels are explicitly provided for training data, and in unsupervised learning, labels are interpreted by the learning algorithm itself.

The general form of training data is given below.

$$\{(x_0, l_0), (x_1, l_1), \dots, (x_i, l_i)\}$$

Here i is the index of data samples and labels in the training set, $i + 1$ represents the number of training samples, x_i denotes an n -dimensional vector, and l_i is the label for this vector, which is an m -dimensional vector. Machine learning generally seeks to approximate a function that maps an n -dimensional input vector to an m -dimensional output vector, according to the given training data.

More formally, machine learning approximates a function F of type:

$$F : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

An *artificial neural network* (ANN) is a well-known, efficient, and powerful machine-learning approach. An ANN is an abstraction of a nervous system of interconnected neurons.

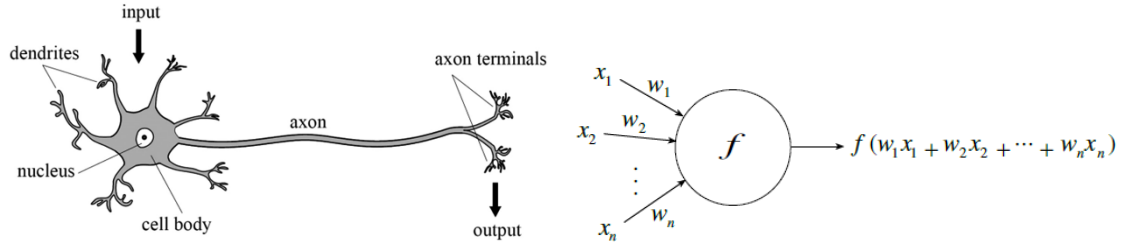


Figure 2.1: A basic biological neuron from [68] In an ANN, these nodes are organised layer-by-layer, where each node on one layer is connected to all nodes in the next layer. The output of one layer is used as the input to the next layer. and a basic node from [82]. w represents the synapses, the input connections represent the dendrites, f represents the cell body, and the output connection represents the axon.

A neuron is a cell with multiple forms and components in biological neural networks. Information is stored at contact points between different neurons, these contact points are known as synapses. The basic function of a neuron is to receive several electrical signals from other neurons through *dendrites* and then to produce an output signal to send to other neurons; through an *axon*. Finally, the neuron's body determines the output signal sent by the axon.

An ANN approximates this process using a computer. The concept of an ANN was originally developed by Wiener, McCulloch, Pitts and Von Neumann as one of five fundamental models of computation in the 1940s [82]. Neural networks were developed alongside: the logic-operational model by Turing; the mathematical model by Kleene and Church; the cellular automata model by Von Neumann; and the physical computer model by Von Neumann.

ANN's approximate biological neurons through nodes (artificial neurons), graphically represented in Figure 2.1:

- *Dendrites* are modelled by input channels from other nodes.
- *Synapses* are modelled by assigning a separate weighting for each node connection.
- The *axon* is modelled by a single output value from the nodes.
- The *cell body* is modelled by a function assigned to each node, referred to as an activation function, which models the output value decision-making.

In a deep neural network, generally referred to as a network with more than one hidden layer, various nodes are arranged in layers, with each connection assigned *trainable* weighting. Each node is also assigned a single value referred to as a *bias*, more details of which are discussed later in this section. These parameters are learnt concerning training data and are used to approximate given training data.

In Figure 2.2, each line represents a connection from the left to the right layer and a trainable parameter. The weights of each layer can be represented as a matrix, and the bias values of each layer can be represented as a vector. More details of this are discussed in Subsection 2.1.4.

This section introduces neural networks to provide context as to how they are made and the

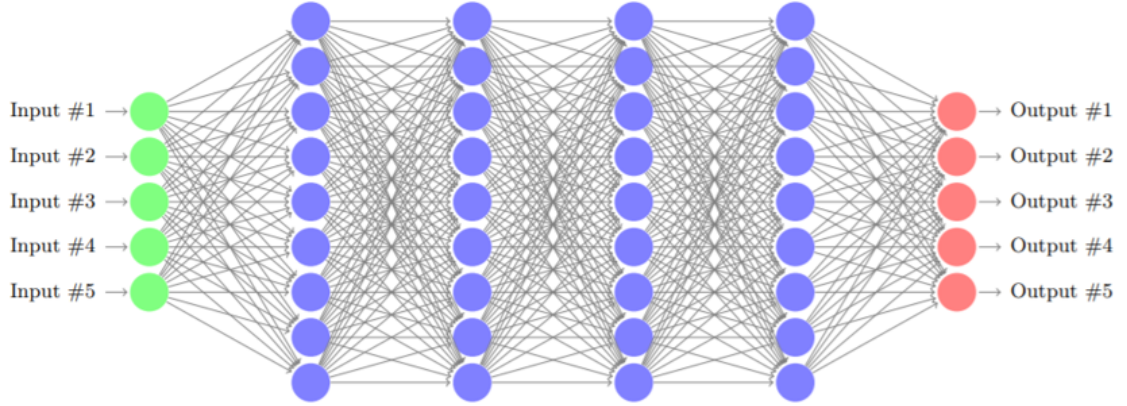


Figure 2.2: An abstract neural network from [48].

basic theory behind them. What follows is a description of the types of layers that a neural network can have.

2.1.1 Training a Neural Network

To make a neural network model useful, it must be able to represent a mapping from data to meaning. This is achieved by minimizing an *objective function*, defined as $F(x)$ where x is the trainable neural network parameters. An objective function is also referred to as, equivalently, an error function or a loss function. An objective function is a function that considers training data, and gives a weighting of 0 for a correct label prediction. So the objective of the training is to have the output of this function be as small as possible, representing that the neural network classifies as many samples correctly as possible.

There are multiple types of objective functions, but one of the most common is the sum of squared errors, which is:

$$F(x) = \sum_{i=1}^n (y_i - \bar{y}_i)^2$$

Here i is the index of the training data sample, y_i is the neural network output for that data sample, and \bar{y}_i is the label for that data sample.

An objective function $F(x)$ is minimized by calculating the derivative of this function and minimizing this value. In this function, x represents all of the trainable values of a neural network, which is the summed weight matrices and bias vectors for every layer (for a fully connected layer, more details of which are given in Subsection 2.1.3). This is a high-dimensional problem, with the number of trainable weights for a neural network can be given by the following.

$$\sum_{i=1}^{i=L} ((m_{i-1} * m_i) + m_i)$$

Here m is the current node number, m_0 represents the size of the input layer, m_{i-1} represents the size of the previous layer, and L is the number of layers.

To compute the derivative of the objective function, the most well-known method is through backpropagation. Here, partial derivatives of each weight are computed iteratively by propagating the error value of each node backwards. These partial derivatives are used in an optimization method to update the weights and fit the network to the objective function. The partial derivative of these trainable parameters shows which weight is more ‘responsible’ for the error, so those with a higher derivative are updated more than those with a lower derivative.

The backpropagation algorithm consists of four phases: feed-forward computation, backpropagation to the output layer, backpropagation to the hidden layers, and updating the weights. The optimization method used in backpropagation is gradient descent, described below.

The four-phase process can be described mathematically as finding $F'(x)$: the derivative of the objective function concerning the network trainable parameters x , Then updating each value according to a learning rate. This learning rate is a value between 0 and 1 and controls the speed of convergence and the network update rate. This parameter’s value depends on the training data set and network structure.

This can be represented as:

$$w_{i_{new}} = w_{i_{prev}} - \delta \frac{\partial F}{\partial w_i}$$

Here δ is the learning rate, and w_i is a trainable parameter. For a fully connected network, the trainable parameters are the weight matrix and bias vector of every layer. This process is repeated until the value of the objective function converges to a sufficient degree, defined by a threshold.

There are several problems, however, that can occur during training, one of which is the *vanishing* gradient problem. This occurs when the partial derivative w_i for a particular parameter becomes too low, and the update for that weight becomes very slow, decreasing exponentially with the number of weights multiplied into it subsequently. This occurs because backpropagation uses the chain rule, multiplying n weights to compute each gradient, so the derivative of a gradient value smaller than 1 decreases exponentially with n .

Another problem faced by training, in general, is overfitting, which is when a machine learning model fits its function too exactly to the training data. This definition incorporates the model learning patterns of noise, being affected too heavily by noise. In general terms, this refers to learning relationships that only exist for that subset of the training data, not the overall data trend.

Finally, when backpropagation terminates, when it reaches an acceptable value of the objective function, it may be a local minimum, so it is not the optimal solution for the objective function. The consequence is that multiple applications of the backpropagation algorithm may produce varying results because of this local optimization problem. This occurs because a neural network represents a non-linear function, which cannot be assumed convex, so non-linear gradient-based optimization must be used. Optimization and convexity are further discussed in Section 2.2.

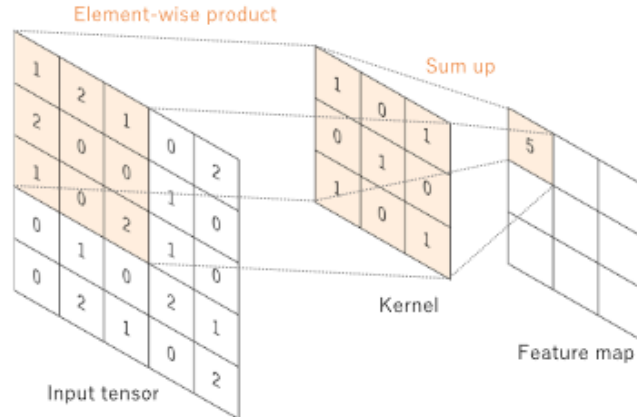


Figure 2.3: An example convolution operation, from [114].

This section briefly introduces how an ANN can be trained, or ‘fit’, to training data. This section describes training fundamentals and some key problems that may arise during training. The next section discusses the types of layers a neural network may contain.

2.1.2 Layer Types

Neural networks are organised in layers, where each layer’s output is input to the next layer. There are a wide variety of layer types used in neural networks. However, the three broad types of layers are fully connected, convolutional and recurrent.

These types of layers can be differentiated based on purpose. Fully connected layers learn a relationship between input nodes. Convolutional layers learn a lower-dimensional representation of the input features; these perform feature extraction. Sudholt and Fink [92] comment that convolutional neural networks (that is, neural networks containing convolutional layers) can consistently outperform all other approaches in virtually every field of computer vision. This is because computer vision applications are usually high-dimensional, and learning a fully connected layer between every input node in a high-dimensional domain becomes ineffective as it contains a large amount of unnecessary information and noise.

Convolutional layers are referred to as such because of the *convolution* linear operation. In this operation, a small matrix, a kernel, is matrix multiplied element-wise across the entire input space. The input space is often represented as an input *tensor*, a term representing a vector by an array of components.

The result of which is summed to produce the output of the layer. This output is sometimes called a ‘feature map’ because the output is designed to represent extracted features of the input tensor. Further operations such as pooling or fully connected layers can then be applied [114]. Figure 2.3 displays a graphical representation of this operation. This shows the first operation of convolution, the kernel is then rotated across the entire input tensor to produce the elements of the feature map.

The trainable parameter in a convolutional layer is the kernel itself. The hyper-parameters, the pre-set parameters of an ANN, that define a convolutional layer are the number of kernels and the size of these kernels [114].

In addition to the convolution layer type, another layer type reduces the dimensionality of the input space. This is the pooling layer. The pooling layer operates with a set kernel, as in the convolutional layer, but instead of performing element-wise multiplication and summation, it performs a single pooling function, commonly the *MAX* function, where only the maximum value is kept of the input section [114].

Recurrent networks are networks containing loops. These recurrent connections allow the network to ‘remember’ its own previous output decisions. This allows the network to learn temporal patterns and patterns in the input vector itself. Recurrent networks experience problems with training, as this model contains loops, and there can be problems such as vanishing and exploding gradients as discussed in [76].

In the next section, we discuss the various types of activation functions that each network node uses.

2.1.3 Activation Functions

The activation function is the non-linear transformation applied to each node’s output of the matrix transformations. This function is often non-linear because it can represent a wider set of functions. This is because if the activation function is linear, no matter the number of layers, the network can be represented by one affine transformation. This limits the types of relationships that the network can learn. Affine transformations are discussed in Section 2.4.

There are many activation functions, however, in this work, we cover well-used, well-documented and fundamental activation functions. So, the activation functions we cover are fundamental and have been modified in various ways but are still popular and widely used.

One of the first activation functions to be proposed was the *sigmoid* activation function, defined in [71] as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid function is a bounded differentiable real function. The sigmoid function gives an output between 0 and 1, so is useful for predicting probability-based output. It has been successfully applied in many domains, including binary classification and modelling logistic regression.

However, the sigmoid activation function has multiple drawbacks, such as slower training on multi-layer neural networks and is non-zero-centred. This means gradient updates can propagate in different directions, making training more inconsistent.

Another activation function is the *hyperbolic tangent function*, or *tanh*, given as [71]:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

This activation function is smoother than the sigmoid function, performs better on multi-layer neural networks, and is a zero-centred function whose range lies between -1 and 1. Both the tanh and sigmoid activation functions, however, suffer from the vanishing gradient problem.

This is because, for large input values, they snap to 1, and for small input values, they ‘snap’ to -1 or 0. The nonlinearity, their sensitivity, is only contained with mid-point values: this can lead to the vanishing or exploding gradient problem.

A more modern function, which was defined to address the problems with the previous activation functions, is the *rectified linear unit (ReLU)*:

$$f(x) = \max(0, x)$$

The advantages of the ReLU function include: it is faster to train, is easier to optimize, and has been reported to perform and generalise better than the sigmoid function [71]. The ReLU function is also piecewise-linear: where piecewise-linearity is a property of a function, which is that the function is composed of multiple linear functions at various intervals. In this case, the function is $f(x) = 0$ if $x < 0$ and $f(x) = x$ if $x \geq 0$. Piecewise linearity has positive implications in implementation, optimization and verification as opposed to fully non-linear functions such as sigmoid or tanh.

A review published in 2015 shows that ReLU is the most popular activation function for deep neural networks [55]. An arXiv pre-print paper submitted in 2018 also comments that ReLU is the most widely used activation function for deep learning applications that achieve state-of-the-art results [71].

Finally, ReLU eliminates the vanishing gradient, as the gradient of the ReLU function is either 1 or 0. This eliminates the vanishing gradient problem, as it can only occur when the gradient of a function is less than 1 and eliminates the exploding gradient problem, as the gradient cannot be greater than 1.

Although ReLU does contain several problems. Such as, certain nodes can ‘die’ when they are forced to 0 gradients and therefore are not updated for the remaining epochs of that batch. Also, in comparison to the sigmoid function, however, ReLU tends to overfit [71]. This is because the change rate is not as smooth, the update is more ‘jagged’, either being forced to 0 or the maximum value.

The dropout technique [71] has been employed to reduce the effect of overfitting. In addition, further developments on the ReLU function, such as the *leaky* ReLU and the *Parametric Rectified Linear Unit* [71], can help reduce the effect of overfitting and dead neurons.

One additional key feature about activation functions is that they contain an activation threshold, for which any value below this threshold is either negligible or zero. This threshold can be modified by input weighting to the activation function. This enables the network to learn boolean relationships between input nodes.

In the next section, we discuss how fully connected layers can be represented using linear algebra.

2.1.4 Matrix Form

Fully connected layers are the essential operation of a neural network: they represent the relational properties learnt on input data. In this section, we detail the operation of fully connected layers.

In a fully connected layer, each layer represents a linear transformation, a linear translation and a non-linear transformation. A linear transformation is represented by matrix multiplication, and a linear translation is represented by vector addition. For the purposes of the following discussion, vectors and matrices are described as $N \times M$, where N is the rows and M is the columns.

The linear transformation can be represented by a matrix product of the input (column) vector $I \times 1$ and an $N \times I$ weight matrix, and the linear translation can be represented as vector addition of bias vector $N \times 1$, where I is the input dimensions and N is the number of neurons in the layer. The full output of the layer also applies a non-linear function f to the output of this affine transformation [96].

$$L_i = f(W_i x_i + b_i)$$

Here L_i represents the output of layer i , and the weight matrix W , input vector x , and bias vector b are indexed by layer i . This operation takes an input vector of size I and outputs a vector of size N .

This is a linear algebraic neural network formulation, so it cannot accurately represent non-linear transformations, such as some types of activation function f . Although, when a network utilises the ReLU function, when $f = ReLU$, linear algebra can be used since, as mentioned, the function is piecewise-linear.

The ReLU function is represented in linear algebra using splits by computing two versions of the function: where one is assuming that the output of the function is 0, and one is assuming that the output of the function is x , where x is the input to that node. This is referred to as the *phase* of the ReLU function.

In a layer-by-layer computation using the formulation above, the number of calculations necessary in each layer equals 2^N , where N is the number of nodes in that layer. This is because every combination of phases must be computed to represent the output of that layer.

This computation can be represented using other formulations, such as through set operations [96, 97] or linear programming [48, 49]. In these formulations, a ‘split’ represents each time a branch is made for a node, a branch of either 0 or x . This number of splits can vary depending on various factors.

As each layer’s I value, which is its input dimension, relies on the previous layer’s output dimension, the number of splits increases exponentially as the number of layers increases. The worst case is when the number of layers equals the number of nodes, with each layer composed of one node. In this case, the number of splits is given by 2^l , where l is the number of layers (and nodes).

The best case number of splits is $2(n)$ where n is the total number of nodes in the network. This would be when the network is single-layered.

A formula to give the number of splits is $n(x^l)$, where n is the number of nodes, l is the number of layers, and x is a variable constant that changes depending on the shape, the exact layer structure, of the network.

This is a basic formulation of a network using linear algebra, assuming no estimation is used on the phase of the ReLU, whether it is 0 or x . In this, the best-case number of splits is 0 if every node phase is inferred. It is worth noting, that the primary goal of many verification techniques is reducing the number of splits, or branches, necessary to verify the property in question.

Even though this linear algebraic formulation cannot be fully applied to every activation and layer type, it illustrates the general operation of feed-forward layers. In that, the number of ‘splits’ and node phases are also relevant factors in all activation functions with an activation threshold. In addition, general non-linearity can be approximated using linear bounding to use this formulation.

The next section discusses how to fit a neural network model to training data.

2.1.5 Summary

Artificial neural networks are a fascinating and multi-faceted model of computation. This section covers how a neural network can learn, interpret and inductively extract patterns from data and the essential types and forms they can take. The essence of a neural network’s operation is inductive pattern extraction from an approximation of the real world.

An artificial neural network represents a complex mathematical function. If a neural network is discretised or binary, it represents a discrete function. If a neural network’s activations are real-valued, the neural network operates on Euclidean real-valued space, which is infinite. The verification of a neural network is the verification of this function, which the neural network represents.

The remainder of this chapter focuses on the problem of, given a pre-trained neural network, which is considered a static real-valued function, what techniques are available to verify the properties of this network.

2.2 Neural Network Verification Techniques

Verifying an ANN is a very complex problem. ANNs operate on the real number domain, so model-checking approaches and boolean satisfiability (SAT) solvers are not directly applicable. Furthermore, as discussed, ANNs utilise non-linear activation functions, so the problem is non-convex. The result is that convex optimization techniques, including linear programming (LP) and semi-definitive programming (SDP) solvers, are also not directly applicable.

So, dedicated specific techniques are required for the verification of ANNs. Verification of even simple properties, those defined by linear constraints, is NP-complete [48]. Also, no accepted approximation or optimal method has been widely accepted. Following this, all verification techniques we discuss here are for verifying properties formed by combinations of linear constraints.

Two broad categories of techniques developed for verifying neural networks are referred to here as complete and incomplete techniques. These are also, equivalently, referred to in the literature as exact and over-approximate verifiers [86, 56]. There are also two primary types of properties, adversarial robustness and general linear constrained properties.

Adversarial examples, first described by Szegedy et al. [93], are defined as data samples imperceptibly different, such as images that differ in just one pixel, that can cause misclassification of that data sample. In response, the concept of adversarial robustness property was developed; it ensures the absence of adversarial examples. Adversarial robustness is defined by an optimization problem, where the objective of this property is to find the minimum adversarial perturbation in a given area around a data sample or verify that none exists. This area is defined by a vector norm from a training data point of interest. The adversarial robustness property can be represented in varying ways, but a concise definition from [86]; is as follows.

$$\min_{x' \in S_{in}(x), i \neq i^*} f_{i^*}(x) - f_i(x') > 0$$

Here a neural network is represented by a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where $f_i(x)$ is the output of logit i , and a logit is an output node of a neural network. Additionally, given an area represented by $S_{in}(x)$, and $i^* = \arg \max_j f_j(x)$, where argmax is the pre-image of the function's maximum, that is to say, it is the set of inputs in the domain of the function that corresponds to the maximum value of its image. Here, the index j represents the logit index, so i^* represents the index of the highest logit, representing the class predicted for the original input data sample. $S_{in}(x)$ represents an area bounded by a vector norm, usually the l_{inf} , l_1 or l_2 norm.

The interpretation of this representation is, given a point x , where the weighting for this class is given by f_{i^*} , there does not exist any sample x' in the area of interest $S_{in}(x)$ that is classified differently. Providing a different classification requires that there must exist a value x' for which $f_i(x') > f_{i^*}(x)$, that is, the logit i of the activation x' that is higher than the highest logit for the original data sample. This can be rewritten as the property of interest is to satisfy $f_{i^*}(x) - f_i(x') > 0$ for all $x' \in S_{in}(x)$.

In contrast to this specification, which is defined as a minimization problem [86, 93], general linear constrained properties are defined as seeking to satisfy a property φ that is a conjunction of linear constraints over the input vector x and the output vector y of an ANN, so, φ takes the form $\varphi = \varphi_1(x) \wedge \varphi_2(y)$, where $\varphi_1(x)$ and $\varphi_2(x)$ are themselves the conjunction of an arbitrary number of linear constraints over x and y .

The first class of techniques we introduce is the complete verification type. These techniques are used to reason over a network's exact operation and, if they terminate, are guaranteed to prove the property or prove it is unsatisfiable. Since the problem of verifying an ANN is NP-complete [48], these techniques all have an exponential worst-case runtime. The primary drawback of these techniques is their scalability and applicability. Where the result's precision is not a factor for complete algorithms themselves.

The next class we introduce is incomplete techniques. These obtain verified upper and lower bounds on the network's output. Given a linear constrained bound, they produce a range of the network's potential outputs. The main problem is the quality of the bounds produced, which may not be tight enough upper and lower bounds to verify properties. This is often referred to as the problem of bound tightness [86].

Incomplete methods are concerned with striking a balance between precision and scalability, as in maintaining useful bounds while still being tractable. Incomplete methods, because

	Type	Name of Technique/Tool*	Authors	First Submitted	Reference
Complete	Linear Programming	MIPVerify	<i>Tjeng et al.</i>	Nov 2017	[95]
		-	<i>Cheng et al.</i>	Apr 2017	[20]
	SMT Solvers	-	<i>Fischetti & Jo</i>	Dec 2017	[27]
		NSVerify	<i>Lomuscio & Maganti</i>	Jun 2017	[57]
		Reluplex	<i>Katz et al.</i>	Feb 2017	[48]
		Marabou	<i>Katz et al.</i>	2019	[49]
		Planet	<i>Ehlers</i>	May 2017	[26]
		NPAQ	<i>Baluta et al.</i>	Jun 2019	[13]
	Interval Analysis	ReluVal	<i>Wang et al.</i>	Apr 2018	[105]
		-	<i>Xiang et al.</i>	Dec 2018	[113]
Symbolic Representation	ExactReach	<i>Xiang et al.</i>	Dec 2017	[111]	
	Exact-Star	<i>Tran et al.</i>	2019	[96]	
Incomplete	Boundary Search	Branch and Bound (BaB)	<i>Bunel et al.</i>	Nov 2017	[18]
		SHERLOCK	<i>Dutta et al.</i>	Sep 2017	[25]
		MaxSens	<i>Xiang et al.</i>	Aug 2017	[112]
	Abstract Domain	Ai2	<i>Gehr et al.</i>	2018	[34]
		DeepZ	<i>Singh et al.</i>	2018	[88]
		RefineZono*	<i>Singh et al.</i>	2019	[90]
		DeepPoly	<i>Singh et al.</i>	Jan 2019	[89]
		RefinePoly*	-	-	[1]
	Linear Relaxation	Approx-Star	<i>Tran et al.</i>	2019	[96]
		Neurify	<i>Wang et al.</i>	Sep 2018	[104]
Fast-Lin		<i>Weng et al.</i>	Apr 2018	[106]	
CROWN		<i>Zhang et al.</i>	Nov 2018	[116]	
Dual Formulation	-	<i>Qin et al.</i>	Feb 2019	[80]	
	LP/LP-FULL	<i>Wong & Kotler</i>	Nov 2017	[107]	
	SDP	<i>Raghunathan et al.</i>	Jan 2018	[81]	
	Duality	<i>Dvijotham et al.</i>	Mar 2018	[53]	
Lipschitz Constant Estimation	Fast-Lip	<i>Weng et al.</i>	Apr 2018	[106]	
	RecurJac	<i>Zhang et al.</i>	Oct 2018	[117]	

Table 2.1: A table summarising the categories of formal verification techniques discussed in this section. *RefineZono and RefinePoly (we were unable to locate the original paper in NeurIPS’19 [1]) are hybrid methods and have a complete mode, but for the purposes of this comparison are listed as incomplete because of their standard operation.

they operate on a representation of a network, not the exact one, can usually be applied to more activation functions and layers easier than complete methods.

Table 2.1 summarises the categories of complete and incomplete formal verification techniques, the details of which are discussed in Sections 2.3 and 2.4. This table displays the class of technique, complete or incomplete, the type of the technique, the date a paper was first published, the name of the technique, the author and a reference. The name of the technique is obtained from the name of the tool implementing that technique (more details of which are in Section 2.3), the name of the technique given by the authors themselves or the name of the technique as referred to in the comparison work by Liu *et al.* [56]. The first submitted date was the first available publication date, either in conference proceedings or on the arXiv open access website (arxiv.org). In this section, we describe the approaches that have been taken to verify neural networks.

In Sections 2.3 and 2.4, we discuss the two main types of formal verification techniques: complete and incomplete. In Section 2.5, we discuss other types of induction and simulation techniques available for verifying and validating neural networks. Before that, however, we discuss a fundamental property that enables tractable verification of ANNs, convexity.

In a paper for the International Congress of Mathematicians 2010, Kjeldsen [51] discusses how the study of convexity is heavily associated with developing tractable mathematical

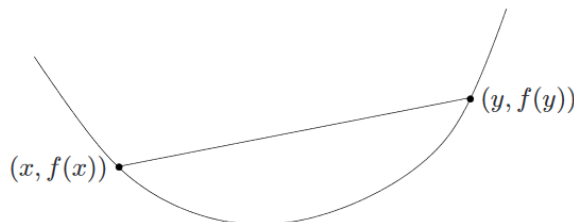


Figure 2.4: A graph of a convex function, displaying the line segment from x to y , $(x, f(x))$ to $(y, f(y))$ from [16].

programming in general. Convexity is a concept crucial to linear programming [51], game theory [51], and most algorithms for verifying neural networks.

Convexity is central in enabling tractable computation for real-number valued analysis. There are multiple reasons for this concerning complexity results. In particular, for optimization problems, which form the basis for most techniques, there are no deterministic algorithms for nonconvex optimization. That is to say, nonconvex optimization is NP-Hard [67]. Conversely, convex optimisation can be solved tractably in polynomial time [16].

A convex function is defined as a function that has an epigraph, the set of points on or above the function, as a convex set. A convex set has no ‘gaps’, that is, for any two points in the set, all points on the line segment between them are also in that set. For any convex set C , for any $x_1, x_2 \in C$, and for any $0 \leq \theta \leq 1$, then $\theta x_1 + (1 - \theta)x_2 \in C$ [16]. In addition, the opposite of convex is concave, and a function is convex when $-f$ is convex.

Another property about convex functions concerns any chord on the function: the line segment between $(x, f(x))$ and $(y, f(y))$. All points on the chord are above the graph of the function. This is shown in Figure 2.4 for a 1-D input.

Convexity and concavity are guaranteed when a function is linear, as a linear function has a constant rate of change. This means that the function cannot contain any inflection points. A neural network, however, is composed of multiple non-linear functions, so convexity cannot be assumed.

To optimise a non-convex function using convex optimization, a convex function must be obtained to represent this function. This may be achieved by multiple convex functions or at least one over-approximate function. This is referred to as convex relaxation. If an over-approximation of the function is generated, any optimum point found will be an upper or lower bound to the function’s true minimum or maximum. In addition, even if a function is convex but non-linear, it may be preferable to relax it to a linear approximation in some situations.

For example, consider the function:

$$f(x) = \max(x, 0)$$

This is the ReLU activation function, and a fully connected layer in an ANN can be composed of a linear combination of such functions. This function is convex but non-linear, and a linear combination of non-linear functions is not necessarily convex. On the other hand, any linear combination of linear functions can still be represented as a linear function, which is

convex. Following this, to form a convex representation of a neural network, these non-linear constraints must be relaxed to linear constraints.

ReLU can be represented as two linear functions, $f(x) = x$ and $f(x) = 0$, however, the consequence is that the number of constraints increases exponentially as the layers increase. This concept is mentioned in Section 2.1.4 on matrix form. This representation can also be described as a node-wise convex relaxation of a ReLU ANN. This linear relaxation is an exact representation and forms the basic concept for the complete verification of ANNs.

An alternative, however, provides an over-approximate convex relaxation of the network, which can be applied layer-wise. A linear combination of ReLU functions forms this layer-wise non-linear function according to the layer size. Convex relaxation is then applied to this non-linear function to form an approximate representation for verification. This is the basic concept of enabling incomplete verification of ANNs.

Further details of how this is performed are discussed in Section 2.3. We present, first, complete techniques in the next section.

2.3 Complete Techniques

Complete analysis of neural networks can be performed in two ways: providing a counter-example disproving the property or generating the complete output range possible for the input bounds and checking that this does not intersect with a given unsafe zone that would violate the property. Here, a counter-example refers to a particular data sample that cannot exist if the property holds.

It is worth noting that, in this work, safe and unsafe refer to whether or not a property is satisfied. This safety does not refer to the safety of the overall system or safety concerning a domain.

For example, we consider a single input x , single output network y , and a property requiring that input bound $x > 5$ ensures output bound $y < 1$. If, when given $x = 6$, the network returns 2, this is a counter-example, as it shows the property is not satisfied. In addition, any property can be checked in this way, instead of verifying that every value greater than 5 produces an output smaller than 1, attempting to find a single example that violates the output bound. In this example, this would be finding an input greater than 5 that produces an output greater than or equal to one. This is the basis of search techniques: SMT and linear programming solvers are examples of search-based solvers.

Considering the same property, $x > 5$ implies $y < 1$, an alternative method to verify it is to estimate the output of this network. This involves finding, given $x > 5$, the possible values of y that can be produced. In this approach, the ANN does not satisfy the property if the reachability set, R , has a non-empty intersection with a given unsafe area U . In other words, if $R \cap U \neq \emptyset$ holds, then the property does not hold. In our example, the unsafe area is characterised by $y \geq 1$. This approach is used by interval analysis and symbolic representation solvers.

Search techniques can be used in two ways, to find a counter-example, as described above, or to find the minimum or maximum possible value produced by the network under given input conditions. This can be useful depending on the property, for example, if it is an

adversarial robustness property, finding a counter-example involves finding the sample a minimum distance from the input.

SMT and LP solvers can be seen as search techniques, searching for the best assignment of decision variables. In contrast, the techniques we present next seek to verify the range of possible values, in other words, define the image of the input domain under the function of the ANN. This image can be represented in multiple ways, but for a complete representation of this image, we define two primary methods: interval analysis and symbolic representation. Both search and reachability techniques can be complete or incomplete. The main differences are in how the ANN is represented, either exactly or as a relaxed form. Here, we discuss how an ANN can be represented exactly. Relaxed form ANNs are covered in Section 2.4.

For effective searching, a network can be represented exactly via two main methods, linear programs and SMT solvers. In this section, we discuss search techniques, then reachability techniques. We discuss linear programming techniques, SMT solvers, interval analysis and, finally, symbolic representation techniques. We first discuss how linear programming techniques.

2.3.1 Linear Programming

A linear program (LP) is a specific formulation of a convex optimization problem where all constraints, including the objective function, are linear functions.

An LP is a problem requiring maximising or minimising a single objective function over a set of decision variables, where the decision variables are defined as real-valued variables of the form x_j , $j = 1, 2, \dots, n$ [102], and the objective function takes the form:

$$\zeta = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

A linear constraint is defined as any constraint of the form $Ax \{\leq, \geq, =\} b$, where x and b are vectors, and A is a weight vector for x , and $\{\leq, \geq, =\}$ denotes the use of one of three comparator operators, \leq , \geq or $=$. Inequalities can be converted easily. For example, the inequality $Ax \leq b$ can be converted to $Ax + w = b$, $w \geq 0$. Here, w is referred to as a slack variable. To convert from an equality to an inequality constraint, for example, $Ax = b$ can be expressed as $Ax \leq b$ and $Ax \geq b$.

There is a type of linear program known as a mixed integer linear program, or simply a mixed integer program (MILP/MIP), where at least one of the decision variables is an integer.

Vanderbei [102] notes that the preferred form is: all constraints are less-than inequalities, and all decision variables are positive. A general example of this is displayed below:

$$\begin{array}{ll} \text{maximise} & c_1x_1 + c_2x_2 + \dots + c_nx_n \\ \text{subject to} & a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m \\ & x_1, x_2, \dots, x_n \geq 0 \end{array}$$

Here, n is the number of decision variables, and m is the number of constraints.

There are multiple specialised tools for solving LPs, including Gurobi¹, GLPK², and LP-solve³. Solving a linear program with arbitrary constraints is a complex problem and may be infeasible. The most used algorithm for solving LPs is the *simplex* method.

The simplex method’s average-case runtime is polynomial but can be exponential. The solution should converge since linear programs are a type of convex optimization, but this depends on the cost function. The LP may even be infeasible based on the set-up of the constraints. While convex optimization, at worst, can be exponential, non-convex optimization, on the other hand, has no clear deterministic method for convergence.

A ReLU ANN can be encoded as a linear program by encoding the objective function as the property in question, or the inverse, to find either a counterexample or a bound to the output to prove the property. The input nodes are encoded as the decision variables so that a program can generate a counter-example or a boundary value for verification.

Lomuscio & Maganti [57] and Cheng et al. [20] propose a method for encoding binary decision variables’ behaviour in formulating a ReLU network called big-M MILP. Cheng et al. [20] utilises a node-wise representation, and Lomuscio & Magnati use a layer-wise representation.

To illustrate the operation of the big-M encoding, we consider the variable $y = \max(0, x)$, where x is the input. This variable cannot be represented directly through linear constraints, as $y = 0$ and $y = x$ are conflicting constraints. The big-M encoding introduces a variable M and a binary variable δ , defining the following constraints.

$$y \leq x + \delta M \text{ and } y \leq M(1 - \delta)$$

Here, δ is a binary variable recording whether the ReLU function is active. If δ is 0, then $y \leq x$ and $y \leq M$, meaning the ReLU function is active, and if δ is 1, then $y \leq 0$ and $y \leq x + M$, meaning the function is inactive. Additional constraints are introduced to force the value to 0 or x , and x is treated as $Wx + b$, with the constraints modified accordingly. The layer-wise representation has fewer constraints and one M value per layer, and the node-wise representation introduces more constraints with a separate M value per node.

Tjeng et al. [95] use indicator variables, and Fischetti & Jo [27] define a MILP layer-based representation using indicator constraints to represent the non-linearity. Indicator variables refer to variables similar to those used by big-M encoding, representing the phase of the ReLU function. In contrast to big-M, though, Tjeng et al. [95] uses separate recorded upper and lower bound variables. Indicator constraints differ from indicator variables because they are logical implication constraints converted internally to valid linear constraints by the LP solver. This approach also generates a distinct upper and lower bound value instead of one M value encompassing the entire constraint. Furthermore, instead of big-M encoding or indicator variables, this approach will always generate a feasible set of constraints.

Tjeng et al. [95] and Fischetti & Jo [27] use asymmetric bounds, separate upper and lower bounds, which the big-M encoders do not use. Tjeng et al. [95] comment that this, in their experiments, creates an increase in the runtime of multiple orders of magnitude, and they

¹//gurobi.com

²//gnu.org/software/glpk/

³//lpsolve.sourceforge.net/5.5/

also mention that by using a restricted input domain, another optimization technique, they can improve upon the approach of Fischetti & Jo by at least one order of magnitude.

To be encoded as a linear program, as noted by Tjeng et al. [95], a property must be expressible as the conjunction or disjunction of linear properties. This is because the output constraints must be encoded in the objective function, and the input constraints in the LP constraints. The layers in the represented network must also be piecewise-linear: contain piecewise-linear or linear transformations. The authors note, however, that this is not a particularly limiting restriction as it allows for convolutional, fully connected and max-pooling layers, as well as ReLU and linear activation functions. Finally, the runtime of these formulations depends on the bounds of the problem but also the tightness of the bounds computed, this is either the upper and lower bounds or the value of M . This is due to the bounds on the linear constraints determining the search space for the linear problem.

2.3.2 SMT Solvers

The next class of search techniques we discuss is the SMT solver technique. A boolean satisfiability solver (SAT) is a specialised solver designed for propositional logic equations, that is, equations using boolean variables, conjunction, disjunction, and implication. A SAT solver operates on a discrete space, so cannot be directly applied to ANN verification.

A satisfiability modulo theorem (SMT) solver extends a SAT solver to include further theories. These theories can include first-order logic, predicate logic, linear real arithmetic, and many others. First-order logic includes quantifiers, variables, and relation symbols. An SMT solver involves a SAT solver at its core. The techniques we discuss here either use SMT or SAT solvers.

Due to its piecewise-linear nature, a ReLU ANN can be encoded directly as a set of linear bounded constraints for solving by an SMT solver. This approach is incredibly inefficient, as demonstrated by Katz et al. [48] in 2017. This is because the search space is intractable, essentially infinite when the constraints are encoded without optimisations. Using this approach, even with state-of-the-art SMT solvers, the authors could not verify even the most basic properties, the form $x \geq c$ for a constant c and an output variable x [48].

One of the first dedicated techniques for ANN verification modified the simplex algorithm for ReLU ANNs. This involved adding further derivations rules to the simplex algorithm to enable efficient solving of ReLU ANNs using SMT solvers. This method extended the simplex algorithm to add the ReLU activation function, hence the name, Reluplex. This method operates by assigning variable pairs for each ReLU connection to represent the inputs and output of each node. An assignment of values to a variable pair is called a ReLU pair. Reluplex attempts to find valid ReLU pairs that characterise a counter-example.

Non-linearity is handled in Reluplex through the ReluSplit rule, which infers the phase of the ReLU activation function and attempts to find a working assignment based on the phase of that particular node. Notably, the method does not need to split on every node, only on those nodes that could be active or inactive, and thus, changing these could lead to a counterexample. The algorithm identifies these nodes using bound tightening by defining upper and lower bounds on the variables.

The key difference in this formulation of Simplex is that Reluplex, based on an SMT solver, not an LP solver, seeks to find a feasible, not the optimal, assignment, and so it searches

for counter-examples. As Reluplex encodes each node as a set of satisfiability rules, this framework applies only to ReLU nodes. Reluplex has been extended, in 2019, into the framework Marabou [49]. It operates on the same principles as Reluplex but with several significant improvements and parallelism enabled through the divide-and-conquer (DnC) mode. Efficient parallel solving is enabled by splitting the input query into multiple sub-problems. Marabou also extends the applicability of Reluplex to general piecewise-linear layers and activation functions.

The Planet approach is similar but uses linear estimation for bounding and a SAT core to compute node phases. This linear estimation allows the solver to prune out large parts of the search domain. Planet applies to piecewise-linear nodes. It uses a SAT core to find a satisfying combination of the phases of the relevant ReLU nodes to find a counterexample instead of searching for an assignment on a real-valued domain over the decision variables, as Reluplex.

It is noted in [18] that the bounding method of Planet is tighter than that of Reluplex. While Reluplex uses a relaxation based on constraint removal, Planet uses a linear estimation. The consequence is that theoretically, more of the search space can be pruned; practically, more of the nodes can be set to either active or inactive state.

Reluplex, Marabou and Planet all operate on piecewise-linear transformations. NPAQ [13], on the other, operates solely on binarised neural networks. NPAQ formulates the binarised ANN as a CNF (a propositional formula of the form $(A \vee B) \wedge (C \vee D) \wedge \dots$). This approach is the only quantitative verification of ANNs we are aware of and can prove rich properties, such as the exact number of discrete values satisfying a property, quantitative analysis of adversarial examples, quantified trojan attacks, and even bias over the class predictions of an ANN [13].

Next, we discuss how an ANN reachable set can be computed.

2.3.3 Interval Analysis

Interval analysis is a classic technique for representing the ranges of functions through intervals. This involves defining a lower and upper bounded input, and propagating this through the function domain, instead of propagating a concrete value. This can represent the possible output range for a function given an input domain or, in other words, the image of the function under a given interval.

Interval analysis, in general, computes over-approximate bounds, but this can be enough for verification purposes. If the over-approximate output range satisfies the required property for a given input range, it is guaranteed that the true output range satisfies the property due to the over-approximation [105].

Interval arithmetic, instead of assigning a concrete value for each variable, defines a range of possible values for each, in the form $X = [\underline{X}, \overline{X}]$, where \underline{X} is the lower bound, and \overline{X} is the upper bound. We consider the function $f(x) = x$ as a simple example for an arbitrary input range $[x]$. The ideal interval extension is a function of intervals that approaches the image of f , such that $[y] = [\underline{y}_1, \overline{y}_1] \times \dots \times [\underline{y}_n, \overline{y}_n]$. This is what the following techniques seek to estimate for an input range $[x]$. It is verified if the $[y]$ does not violate the property.

The work by Xiang et al. [113] proposes an algorithm based on interval analysis and bisection-guided search. This approach is based on splitting the input space into finer and finer

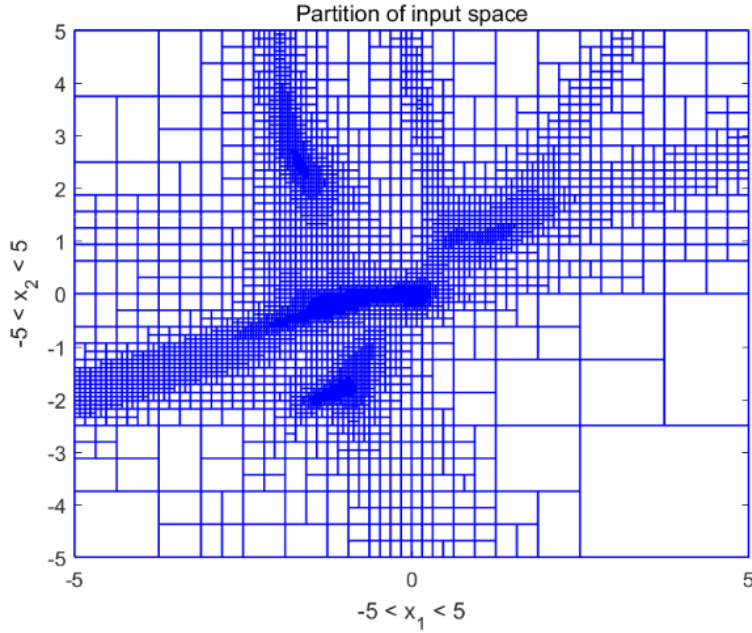


Figure 2.5: An illustration of input space partitioning, from Xiang et al. [113]

partitions approaching the unsafe zone to get to a precision fine enough to prove safety or to reach a pre-defined tolerance limit ϵ . This tolerance is defined as a value greater than 0 and determines the smallest interval size possible for the algorithm. This can provide arbitrary precision on an ANN but with scalability varying with ϵ .

Given an input range and an unsafe output range, the technique in [113] computes the output range through interval arithmetic, and if this intersects with the unsafe zone, bisect this interval into smaller intervals, referred to as partitions, and repeat until tolerance is reached or until no partitions intersect with the unsafe zone. This is shown, for a 2-D input, in Figure 2.5. This figure demonstrates where the input space intersects with the output unsafe zone. Each rectangle represents a partition of the input space, with a corresponding over-approximate range calculated. If the computed over-approximate output does not intersect with the unsafe zone, then the partition is left, it can be pruned out of the search space. Otherwise, the partition may contain a counter-example, so finer partitions must be computed to tighten the over-approximation and prune out unnecessary input sections.

Another approach to applying interval analysis for ANN verification, called ReluVal, is presented by Wang et al. [105]. ReluVal is based on symbolic interval analysis. This is an extension of interval analysis, where instead of maintaining concrete ranges of intervals, it maintains equations based on the previous propagations to preserve information about the relationships between inputs and concretizes (computes exact interval bounds) only when required [105].

ReluVal keeps track of linear equations based on the linear transformations of the weight matrix multiplication and bias vector addition. The nonlinearity of a ReLU presents a problem because when the value of the linear equation representing the interval can be negative, the linear equation does not sufficiently bound the ReLU's behaviour. In this case, the bounds must be concretized with the exact values. The concrete bounds can then be

reduced by iterative interval refinement, which separates the intervals into separate sub-intervals to refine the bounds. The authors note that this can achieve an arbitrary level of precision for any Lipschitz continuous DNN. An arbitrary level of precision is not guaranteed for a non-Lipschitz continuous activation function.

Lipschitz continuity is a strong measure of function continuity. It is important because intervals cannot accurately capture its image if a function is not Lipschitz continuous. A function is Lipschitz continuous if a constant C can be defined for the function, such that, for arbitrary x and y , the following holds:

$$|f(x) - f(y)| \leq C |x - y|$$

One problem with interval analysis for ANNs is the dependency problem: the error caused by intervals producing bounds that depend on conflicting values of input nodes. For example, we consider a single, two-node, hidden layer ANN with nodes n_1 and n_2 . Here, the input to each node is x_1 and x_2 , respectively, and the output of each node is y_1 and y_2 . The output y_1 of node n_1 has a computed upper bound of \bar{y}_1 when the input is x_1 . The output y_2 of node n_2 has an upper bound of \bar{y}_2 , when the input to this node is x_2 . These upper bounds are then used to compute the upper bound of the output layer. However, if $x_1 \neq x_2$, this upper bound for the output layer is impossible, as it depends on the input being two values simultaneously, referred to as the input dependency error or dependency error. Wang et al. [105] prove that in Lipschitz continuous functions, this dependency error decreases as the width of the intervals decreases.

Layers and activations shown to be Lipschitz continuous are fairly extensive: they include the piecewise-linear transformations contained in convolutional layers, ReLU layers, linear layers, max-pooling layers, softmax layers, and contrast-normalization layers [93]. Furthermore, some non-linear activation functions are Lipschitz continuous: such as the sigmoid and tanh function, by [84].

Wang et al. [105], and Xiang et al. [113] can achieve arbitrary precision using interval arithmetic. This is because both approaches use iterative interval refinement, as discussed, and can provide arbitrary precision. ReluVal, however, uses symbolic intervals, which the authors comment reduces dependency errors by maintaining only the relationships between variables. Furthermore, they comment that this reduces computation time, as the bounds are not necessarily concretized at every node of the ANN. Finally, if the ANN is linear, symbolic interval analysis can eliminate the input dependency problem entirely.

2.3.4 Symbolic Representation

The final class of technique we introduce is verification by symbolic representation. Symbolic representation refers to maintaining an interval range, generating a set representing the reachable set, and checking if it intersects with the unsafe zone. This involves defining a set of states via an intermediate representation, which, for computability, is a type of convex polytope with efficient inclusion, translation and transformation operations.

The reachable set is composed of conjunctions of multiple convex polytopes. A hyper-rectangle is the simplest class of convex set. Hyper-rectangles, in this work, have rational endpoints that can be represented by their leftmost and rightmost corners $\underline{x} = (\underline{x}^1, \dots, \underline{x}^n)$

and $\bar{x} = (\bar{x}^1, \dots, \bar{x}^n)$ [61]. A hyper-rectangle is defined as all points $x = (x^1, \dots, x^n)$ satisfying the following as property as given in [61]:

$$\bigwedge_{i=1}^n \underline{x}^i \leq x^i \leq \bar{x}^i$$

For accuracy and flexibility, further representations of convex polytopes are required. A convex polytope is a bounded convex polyhedron [61]: its points form a convex set. A polyhedron is defined here as any n-dimensional shape with flat sides, so a hyper-rectangle is a convex polyhedron but not all convex polyhedrons are hyper-rectangles.

Convex polytopes can be defined through several different representations. Each represents the same class of sets, but the complexity of operations on sets varies depending on the representation [61].

Convex polytopes can be represented canonically in two ways.

1. Vertex Representation: a finite minimal set \tilde{P} such that $P = \text{conv}(\tilde{P})$.
2. Inequality Representation: a conjunction of a minimal set of halfspaces ($H = H^1, \dots, H^k$) such that $P = \bigcap_{i=1}^k H^i$. Syntactically this is represented as $\bigwedge_{i=1}^k a^i x \leq b^i$.

Here \tilde{P} is the vertices of P , and $\text{conv}(\tilde{P})$ is its convex hull. Intuitively, for a set of points S , $\text{conv}(S)$ is the set of all points between the set of all possible convex combinations of the elements of S . [61]. A convex combination of a set $\{x_1, \dots, x_l\}$ is any $x = \lambda_1 x_1 + \dots + \lambda_l x_l$ such that $\sum_{i=1}^l \lambda_i = 1$.

One of the key properties of convex polytopes is that, in contrast to hyper-rectangles, they are closed under general linear operations. That is, if P is a convex polytope, then AP is also a convex polytope where:

$$AP = Ax : x \in P$$

This property is very useful for set-based computation, as they allow the representation of general linear functions. The complexity of set operations varies depending on the representation used, for example, testing membership $x \in P$ is easier using inequalities while checking non-empty intersection $P_1 \cap P_2$ is easier with vertex representation.

Convex polytopes are not closed under non-linear transformations but under affine transformations. These are transformations of objects in affine space: a type of Euclidean, real-valued space where the origin is omitted. A linear transformation of an object is represented by a matrix product, where the origin is preserved. In contrast, an affine transformation is a linear transformation in affine space, where the origin is not fixed, so the object can be translated. So, an affine transformation can be represented by a linear transformation and a translation.

Xiang et al. [111] propose a symbolic representation ANN verification method based on polyhedra defined in inequality form. Their method defines a union of multiple polyhedra, propagated layer-by-layer, to define the reachable set of an ANN.

Tran et al. [96, 98] proposes a symbolic representation method based on star sets, efficient representations of bounded convex polytopes. A star set is a tuple $\Theta = \langle c, V, P \rangle$, where

c is a vector representing the centre of the star, V is a set of basis vectors describing the star bounds, and P is the predicates describing what part of the bounds encompass the star. The predicates are restricted to a conjunction of linear constraints, and so a star represents a bounded convex polytope. A star is closed under affine transformations and halfspace intersections [96].

The input is given as a star set to represent an ANN’s reachable set, which is transformed using an affine transform representing the weight matrix and bias vector computation. The input reachable set is calculated as $\Theta = W * \Theta_{prev} + b = \langle Wc + b, WV, P \rangle$, where the output set from the previous layer is Θ_{prev} , W represents the weight matrix, and b the bias vector of this layer. This is a simplified form of the reachability calculation described in [96].

To handle the ReLU function, each component of the reachable set is analysed, and if the lower bound is less than 0 for that component, then one of the nodes corresponding to that component has the potential to exhibit non-linear behaviour. This is dealt with by defining two further start sets, $\Theta_1 = \Theta \wedge x_i \geq 0$, and $\Theta_2 = \Theta \wedge x_i < 0$, where x_i is the component of the input set whose lower bound is less than 0. Here, Θ_2 represents the inactive state of the ReLU because x_i is now strictly less than 0, and when the ReLU function is applied, that component is set to 0. This process is referred to as stepReLU.

Every new star set created by stepReLU leads to another full reachability construction in subsequent layers, as in another stepReLU for each new star set generated. This leads to a worst-case number of stars for an ANN with n nodes, to be 2^n , which is also the worst-case number of splits, as discussed in Section 2.1.

The symbolic representation algorithms we discuss apply to piecewise-linear layers, such as max pooling, convolutional, ReLU, and linear fully connected activation functions. This is due to the necessity of affine transformations to these representative sets.

2.3.5 Discussion

No complete technique we know applies to pure non-linear activation functions or layers. All are at most applicable to piecewise-linear transformations. This is due to the complexity of capturing an ANN’s output range through computable methods, as this section outlines.

This output set is sometimes known as an adversarial polytope, representing the complete output range. This requires representing a non-convex multi-dimensional polytope through various forms, which is computationally challenging. Piecewise-linear activation functions make this polytope significantly simpler to reason with.

Comparing these techniques’ effectiveness is challenging. This is because they are all new and currently in development, and consistent benchmarks are unavailable for the emerging sub-field of neural network verification. As mentioned, all complete verification techniques’ runtime is, in the worst-case, exponential, but their average and best cases vary and depend significantly on multiple factors.

The scalability of these techniques depends on multiple factors: the width of the network, the depth of the network, the property bounds, the input and output dimensions, and the number of splits required. The width and depth of a network characterise its size: the number of hidden layers and nodes in those hidden layers. The property bounds refer to the shape and complexity of the linear bounds defining the property. The input and output

dimensions refer to the number of input and output nodes corresponding to the size of the input and output vectors.

The most important factor, arguably, is the number of splits required. For example, if the number of splits required is 0, and every node has a value greater than 0 after the weights and bias vectors, the ReLU node is always $f(x) = x$. This system of equations becomes entirely linear. This reduces the number of constraints required dramatically for linear program solvers, makes the network able to be represented by one affine transform of one polytope set, and makes the network able to be represented by a single affine arithmetic computation.

For complete techniques, scalability is the biggest concern, as representing non-linearity and complete verification of ANNs is NP-hard. This leads to problems with larger networks, but also to a problem with utilising exact arithmetic. Using exact or floating point arithmetic is an important consideration for real-valued calculations. Exact arithmetic is usually used in verification methods, such as SMT solvers with the theory of real arithmetic, but is challenging to use with verification techniques, as scalability is already an issue and exact arithmetic is usually at least an order of magnitude slower than floating point arithmetic [48].

Techniques use various types of approximation or error compensation to compensate for this challenge. Some techniques maintain a separate parameter for error tolerance, such as NPAQ [13] or Reluplex. Interval analysis approaches use outward rounding to ensure soundness over the true output range. The intervals, however, may not be precise enough to verify the network. To compensate, they use iterative interval refinement, as discussed. Reluplex, to compensate for the error, tracks the cumulative round-off error, and if it exceeds a certain threshold, essentially backtracks a certain number of steps in its assignment of variables but does not completely guarantee soundness. A recent work outlines how these errors can be exploited for previously verified networks [45].

This area, as mentioned, is currently in active development, and the fusion and collaboration of multiple types of analysis and techniques is a promising direction. This stems from the key difference in the philosophy of search and reachability techniques and the possibility of further collaboration to improve search, branching and bounding. This is suggested by Katz et al. [49] with integrating further network-level reasoning into Marabou, such as reachability techniques, and by Singh et al. [90].

This section has given an overview and compared approaches taken to represent an ANN's behaviour exactly for verification. The next section discusses how to represent an ANN approximately for verification.

2.4 Incomplete Techniques

In this section, we discuss ANN verification techniques which are, algorithmically, incomplete. These techniques generate an approximate output set describing the ANN's behaviour. The techniques we discuss in this section generate strict over-approximations, so they can guarantee behaviour. The over-approximation, however, results in incompleteness. Incomplete ANN verification techniques operate in three main ways: searching for the bound limits using over-approximation, relaxing the ANN's non-linearity, or generating a Lipschitz constant for the network to generate bounds.

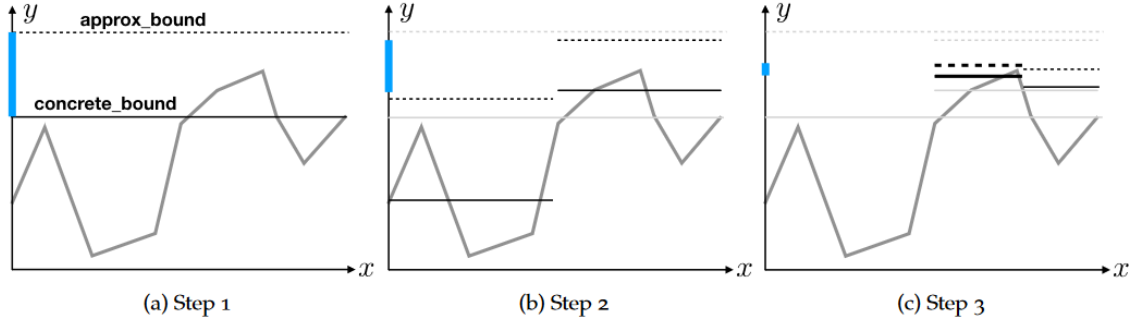


Figure 2.6: The operation of the approach by Bunel et al. [18], from [56]

We first discuss over-approximate search techniques, referred to as boundary search techniques, then relaxation techniques, finally, we discuss Lipschitz constant estimators. We separate the relaxation techniques into 3 types: abstract domain, linear relaxation and dual formulation.

2.4.1 Boundary Search

In a complete technique form, search techniques use higher and higher precision until a valid solution is reached. In incomplete techniques, convergence and arbitrary precision are not guaranteed, traded for the efficiency of evaluation.

The first technique we discuss was proposed by an overview paper of piecewise-linear verification techniques by Bunel et al. [18], referred to as Branch-and-Bound, or BaB.

This technique re-phrases the neural network verification problem to a single optimization problem by adding additional linear layers, with a single node representing the property. This property must be formed from a combination of linear inequalities. Properties based on a single linear inequality can be represented with one additional layer, those based on multiple inequalities require further layers. If the output value falls below 0, the property is violated.

The goal of this technique is to find a single point which is the maximum activation in a given input range. Search techniques achieve this by input space partitioning, removing areas that cannot contain the maximum activation. Note that input space partitioning is used in Xiang et al. [113]’s approach but for intersections with unsafe zones of properties instead of maximum activation searching.

To enable tight bounds, the objective is to maximise the lower bound and minimise the upper bound. This makes the range of possible outputs as tight as possible, enabling effective search space pruning. The lower bound can be generated using stochastic sampling techniques on the input domain, and the upper bounds can be generated using a linear convex relaxation described in [26]. This means the upper bound will always be an over-approximation, and the lower bounds will always be an under-approximation [56].

This reduces the search space of the input domain. This process is illustrated in figure 2.6, where the solid line is the lower bound and the dotted line is the upper bound. Note that, due to the formulation of the property into a single output node, only an upper bound is

required to prove safety, the lower bounds generated here are only for pruning out sections of the input domain, not for generating bounds.

A related search technique is proposed by Dutta et al. [25] and is referred to as SHERLOCK. This approach searches for the upper bound in a related way to Bunel et al. [18]. Dutta et al. [25], however, use a combination of local and global search, instead of linear relaxation and concrete sampling, to estimate the output boundary.

First, a point is sampled from within the valid input constraints. Then, this point is incremented using local search, a gradient-based ascent method. As this method is gradient-based, it is not guaranteed to return the global upper bound if the function is non-convex. To improve the bounds, global search is then used on this upper bound. Global search is an optimization technique that operates on non-convex functions but is not necessarily guaranteed to reach a local optimum. If a global search can find a point with a greater activation, the process repeats, local search is then applied to the new point.

To find the lower bounds, the same method is applied, but in reverse i.e. with gradient descent and global minimization techniques. The tightness of bounds generated by this method is affected by a tolerance parameter δ . This is added to every upper bound computed by local search. The tolerance parameter is a value greater than 0, and the smaller it is the longer the runtime, but the tighter the bounds produced.

The approach by Dutta et al. [25] only applies directly to one-layer neural networks. In contrast, the approach by Bunel et al. [18] is applicable to multiple through the method to re-phrasing the network, whereas Dutta et al. [25] does not give a method of re-phrasing the network in this way.

The next type of technique we introduce is the convex relaxation technique, which includes abstract domain, linear relaxation, and dual formulation techniques. A convex relaxation in itself is relaxing a non-convex problem into an over-approximate convex problem and is used as a bounding step in some complete techniques. In a fully incomplete technique formulation, however, the entire network is relaxed, and the reasoning is entirely over this over-approximate representation.

The following techniques use convex relaxation to simplify the neural network verification problem in LP-form; this LP-form is discussed in the linear programming techniques in Section 2.3, Complete Techniques.

2.4.2 Abstract Domain

Symbolic representation techniques use multiple convex sets to encompass an ANN's exact (non-convex) image. Abstract domain techniques, on the other hand, define the effect an ANN has on a single convex set. That is the smallest convex set that can encompass the entire image of an ANN.

If a network were linear, its image can be represented exactly by a single set: each layer would represent an affine transform. Considering a non-linear network, where the layers represent non-linear transforms, this is not the case. In this case, instead of splitting apart the set to encompass the behaviour, as a symbolic representation, the set is modified with additional variables, or dimensions, to compensate for the non-linearity. The number of splits necessary to represent the network by linear algebra is related to the complexity of the set, not to the number of sets.

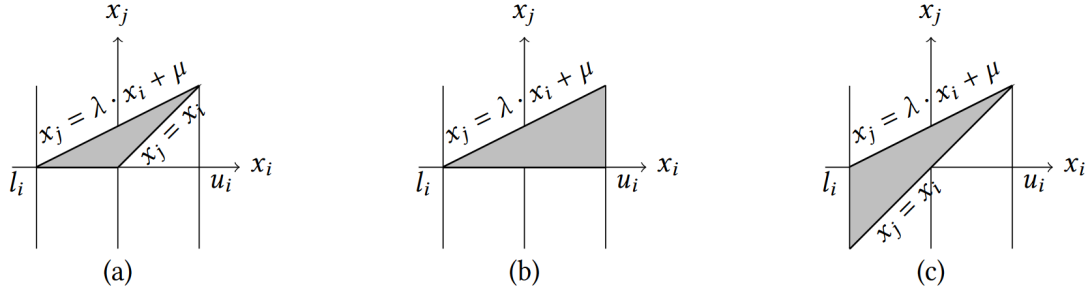


Figure 2.7: Polytope approximations for the ReLU function, from [89]: (a) shows the ideal polytope, (b) and (c) show the two proposed in this work.

DeepPoly, defined by Singh et al. [89] uses polytopes as its domain; it combines floating point polyhedra with interval analysis. In addition, it defines transformers for affine transforms, ReLU, sigmoid, tanh and maxpool layers. Transformer here describes how a polyhedron is transformed to approximately represent these layers.

DeepPoly defines two linear constraints per ReLU, an upper and lower bound constraint. This is to avoid an exponential blowup of the analysis. These linear constraints are defined by minimising a polytope in a 2D input-output plane. Abstract domain methods minimise to reduce the over-approximation in the linear constraints made: the smaller the shape, the tighter the area represented by the linear constraints, so the tighter the over-approximation.

Convex approximations for the ReLU function from Singh et al. [89] are shown in Figure 2.7. Here, λ and μ are the gradient and the intercept, the 1 and 0-dimensional variables defining the upper bound of the polytope.

Gehr et al. [34] and Singh et al. [88] use zonotopes. Zonotopes, as discussed in Complete techniques, are a type of centrally symmetric polytope, and are faster, but more approximate. Singh et al. [88] define zonotopes in affine form, and for every ambiguous value encountered, a new noise symbol is added to the affine form: where an ambiguous value is a value that could either be positive or negative.

Singh et al. [88] use affine arithmetic to define zonotopes, where affine arithmetic is an extension of interval arithmetic. Zonotopes are defined by associating an affine form \hat{x} with each of its dimensions. An affine form is \hat{x} such that $\hat{x} = x_0 + x_1\epsilon_1 + \dots + x_n\epsilon_n$. Where x_0 is the central value and x_1, \dots, x_n are known as the partial deviations, associated with the noise symbols ϵ [21].

Singh et al. [88]’s transformation rules define the input zonotope, as in, the single zonotope transformed with just the linear transformations if the lower bound is greater than 0, defines the affine form $[0, 0]$ if the upper bound is less than or equal to 0, and in the ambiguous case defines the output affine form to be \hat{y} where \hat{y} is:

$$[\lambda_l, \lambda_u] \cdot \hat{x} + [\mu_l, \mu_u] + [\mu_l, \mu_u] \cdot \epsilon_{new}$$

Here, λ is a floating point representation of λ_{opt} and μ , which is the optimal gradient and intercept, respectively, for the lines constructing the zonotope.

This transformation adds another noise symbol ϵ to the affine form, making the runtime linear in the number of noise symbols. Singh et al. [88] also mention that Gehr et al.’s [34] approximation is imprecise and costly due to their approach using the zonotope join operator. The join operator uses one zonotope to cover and approximate all zonotopes generated, whereas Singh et al. [88] uses affine form manipulations, instead.

Singh et al. [88]s approach handles convolutional architectures and is sound with respect to floating point arithmetic.

Approx-Star, by Tran et al. [96] is an abstract domain technique using the star set, an efficient representation of a bounded convex polytope, and is a tighter representation than a zonotope. Approx-Star uses a similar approximation to DeepPoly, but instead of two constraints per node, they use three. This means that this abstract domain is tighter, and more precise than DeepPoly, but may result in a model with more constraints, which can mean less scalable. The star representation, however, contains more set-efficient operations than the polytope representation, so further investigation could be useful for this.

Abstract domain techniques have been defined, because of the over-approximation, of non-linear activation functions, and convolutional layers.

Abstract domain techniques, as they represent the ANNs image with a single polytope, are particularly applicable to image domains, as properties such as rotation, scale, and linear interpolation can be applied as transforms to the generated set, instead of new properties being developed for each type of image property.

2.4.3 Linear Relaxation

Linear relaxation is a convex relaxation technique that defines upper and lower-bound linear functions to approximate the non-linearity in ANNs.

There are three techniques we are aware of which utilise linear relaxation: Weng et al. [106], Fast-Lin; Wang et al. [104], Neurify; and Zhang et al. [116], CROWN. Fast-Lin uses linear relaxation, Neurify utilises symbolic linear relaxation, and CROWN uses linear and polynomial relaxation.

Fast-Lin defines two linear upper and lower bounds to replace the ReLU activation function. These are:

$$\frac{u}{u-l}x \leq \sigma(x) \leq \frac{u}{u-l}(x-l)$$

Here x is the input to the ReLU, $\sigma(x)$ is the ReLU function, and u and l represent the upper and lower bounds of the original x .

Fast-Lin obtains two explicit lower bounded linear functions, f^U and f^L , that bound the value of the original ReLU neural network function, f such that $f^L \leq f \leq f^U$. Obtaining these leads to analytic bounds that can be computed efficiently without using any optimization solvers, to enable fast computation for layer-wise output bounds. The time complexity for computing the output bounds of a ReLU network is polynomial time. Fast-Lin is, however, only applicable to ReLU neural networks.

CROWN is an improved version of Fast-Lin, it bounds ANNs using linear and quadratic relaxation, allowing it to be applicable to general non-linear activation functions. CROWN allows flexible selection of upper and lower bounds for activation functions, enabling up to 26% improvements in certified lower bounds compared to Fast-Lin [116].

Neurify, the approach defined by Wang et al. [104], is a technique utilising symbolic linear relaxation. Symbolic linear relaxation refers to a combination of symbolic interval analysis, as ReluVal [105], and linear relaxation to linear bounds on ANNs. Neurify is only applicable to ReLU networks.

The complete approach by Ehlers [26] also uses a linear relaxation of a network, along with an SMT solver. The approach by Qin et al. [80] also uses this relaxation, however, they define a method to verify non-linear properties through convex relaxation, the only technique we are aware of that is able to do so. Due to their relaxation of the properties, their technique is over-approximating.

2.4.4 Dual Formulation

These techniques formulate a dual optimization problem that obtains valid bounds on the lower bounds, through maximising the dual problem.

Wong & Kotler [107], referred to as LP/LP-full, define a linear program to define the dual. The feasible set of this dual problem, critically, can be expressed as an ANN, and can be solved using backpropagation-like computing, this means verification can be folded into training time, creating a trainable certificate. Another consequence of this approach, however, is the optimization is non-convex, as they express the dual through an ANN. Finally, this approach can train networks with provable robustness against all attacks, instead of generating bounds pointwise, it can generate bounds against all possible attacks for a network.

This approach is sometimes referred to as a certification method, as it can provide guaranteed bounds for an entire network, a 'certificate' of robustness. This approach can be applied only to ReLU networks, however.

Raghunathan et al. [81] present an approach developed in parallel to Kotler & Wong [107], which also presents a certification method using duality. Instead of using linear programming, however, Raghunathan et al. [81] uses semi-definite programming, but is only tractable for one layer, for any additional layers it is NP-hard. It produces a bound on the global Lipschitz constant for a network, this means that the bound is for all samples considered, as opposed to other works which produce a local Lipschitz constant, based on individual samples.

These techniques can be used as a training method, and as a cost function, to obtain verification at training time. A key weakness of these techniques is that they obtain loose bounds on networks not trained with their training methods.

A dual formulation technique developed after, and inspired by, these techniques is Dvijotham et al. [53], referred to by [56] as Duality. This approach operates in four stages: formulate the problem as a linear program, find a Lagrangian relaxation of the constraints, find the dual of this relaxation, and solve using unconstrained minimization.

The linear program developed in the first stage is non-convex, and the Lagrangian relaxation is to develop a convex approximation of this non-convex linear problem. This approach, instead of solving non-convex optimization, as in Kotler & Wong [107], solve the dual problem using unconstrained minimization. A consequence of unconstrained minimization is that their approach is anytime, the process can be halted anytime and still provide valid lower bounds.

Their approach is, in theory, applicable to recurrent neural networks, although the formulation of their approach given in [53] is applicable to layered architectures.

The final type of technique is those that use a Lipschitz constant to generate valid bounds on the output.

2.4.5 Lipschitz Constant Estimation

These techniques are estimators, as to return an exact Lipschitz constant would be complete verification. Both techniques we consider bound the Jacobian matrix to bound a Lipschitz constant: where the Jacobian matrix is the matrix of first-order derivatives of a function, for a one-dimensional function, the gradient.

The approach by Weng et al. [106], referred to as Fast-Lip, bounds the Jacobian matrix by bounding the worst-case construction of the function of the ANN. It achieves this by generating a diagonal matrix of activations for each node to quantify which sections of the function are linear. It then performs analysis on the remaining sections, assuming the worst case for the non-linearity, and uses this to construct the bounds on the Jacobian matrix of the function.

Fast-Lip is applicable solely to ReLU networks, because of how the matrix is constructed, based on the piecewise-linear nature of the ReLU section. Furthermore, Fast-Lip loses efficiency on the number of layers, as the size of the matrix of activations increases exponentially with the number of layers.

Recurjac, by Zhang et al. [117] is a developed version of Fast-Lip which also bounds the Jacobian matrix. Recurjac improves upon fast-Lip in multiple ways: it is applicable to general non-linear activation functions, it does not lose efficiency with multi-layer networks, and it produces a tighter bound on the Lipschitz constant.

Recurjac, instead of processing the network layer-by-layer, presents a recursive refinement algorithm. It finds the maximum gradient in a ball around each activation, obtaining bounds for each layer, known as pre-activations, through CROWN [116]. This algorithm is able to achieve a Lipschitz constant two orders of magnitude smaller than the state-of-the-art (non-recursive) algorithm.

Recurjac has polynomial time complexity, which is also up to M times slower than that of Fast-Lip, where M is the number of layers in the ANN. However, the authors note that this is not a particularly limiting restriction, as it is still tractable for 10-layer 200-node networks, and the time increases linearly with the number of layers, so should not be a particularly limiting restriction.

2.4.6 Discussion

For incomplete techniques, striking a balance between precision and scalability is the key factor. They represent network-level reasoning techniques, as they primarily reason not on the concrete bounds and activations of the original network, but on a relaxed network. This is as opposed to complete verification techniques, which operate primarily on the original network, even if intermediary network-level reasoning techniques are used, these are used to inform concrete reasoning techniques. They can inform this by tightening the search space, and pruning the input domain to rule out areas which cannot contain counter-examples.

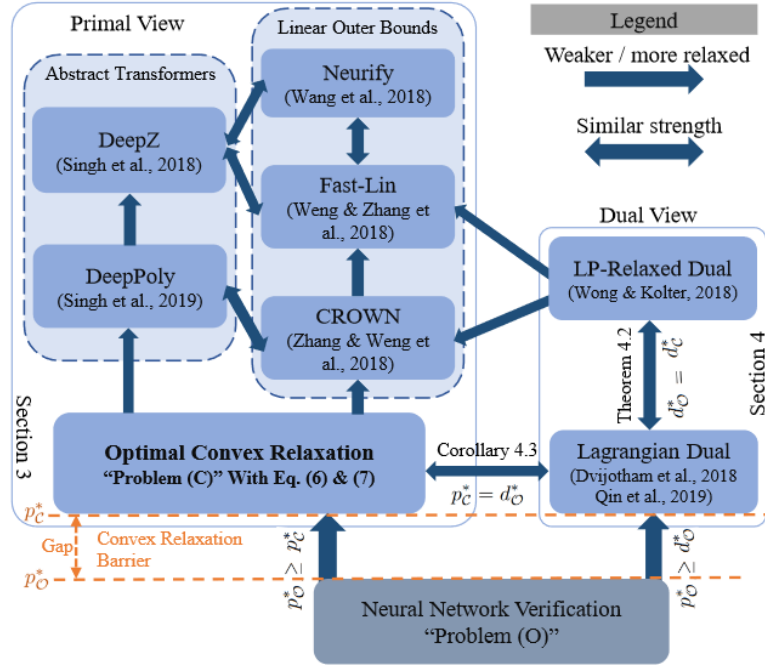


Figure 2.8: The relationship between convex relaxation algorithms, from [86].

Salman et al. [86] describes a relationship between the theoretical tightness of relaxations for convex relaxation techniques, as shown in Figure 2.8. DeepZ is more relaxed than DeepPoly due to the zonotope approximation, and Fast-Lin and Neurify are weaker than CROWN due to the polynomial bounding as well as linear bounding. In addition, they show that the approach by Dvijotham et al. [53] and Wong & Kotler [107] is shown to have similar strength.

Incomplete techniques attempt to balance precision and scalability, and experiments run have not been conclusive about the most precise method of over-estimation. There are multiple factors influencing the effectiveness of these methods, namely: the network shape, the specification type, the theoretical bounding, the network setup and the training data. In particular, in the type of specification, the type of norm used can make a difference in the techniques, such as shown in [106].

In summary, more standardised benchmarks and experiments are necessary to explore the relationships between these techniques and the effect of modifying various parameters. Including exploring the relationship further between Lipschitz constant analysis and convex relaxation of LP methods, and how these two fundamental methods behave under various circumstances.

2.5 Other Techniques

All techniques discussed in Section 2.3 and 2.4 are techniques that provide a direct guarantee on the behaviour of an ANN. In this section, we briefly discuss other types of ANN verification techniques: those that provide guarantees with respect to discretization, adversarial example

crafting, or attack, algorithms, and techniques based on inductive arguments.

Huang et al. [43] and Wu et al. [110] propose techniques based on discretization of the input domain. The consequence of this is these techniques are applicable only to image domains, while this is the main focus of a lot of ANN applications, it is still not verifying the full ANN's behaviour.

Huang et al.'s [43]'s approach is based on forming a derivation tree of possible perturbations from a single point and verifying the non-existence of adversarial examples within these perturbations. These perturbations usually refer to manipulations of images.

Wu et al.'s [110] approach is modelled as a two-player game to create a guaranteed lower bound to adversarial perturbations. This technique has two modes of operation, computing the minimum distance to an adversarial sample, based on the L_0 norm, and quantifying the robustness of individual features to adversarial perturbation features, such as, for images, 'sky', 'grass', etc. This technique requires a Lipschitz constant, which can be generated from Lipschitz constant estimators as discussed in Section 2.4.

Attacks generate the best possible adversarial example; the best possible refers to as a sample as close to the original sample as possible. They do not provide guarantees that this is the best possible sample or even an upper bound on the best possible sample. That is, incomplete techniques generate a guaranteed lower bound, a sample such that no more optimal samples can exist. This provides a guarantee of safety with respect to that point. The samples crafted by attack algorithms do not perform this, but their samples can still be used to provide an upper bound to the true adversarial bounds, whereas incomplete techniques provide a lower bound. If the adversarial example search is incomplete, the generated bounds represent a lower bound on the true adversarial bounds, while an attack represents an over approximate upper bound. Attacks can also be used to augment the training data to increase robustness to attacks, as is demonstrated in [66].

L-BGFS is an algorithm for solving unconstrained non-linear optimization problems. It also refers to an attack algorithm by Szegedy et al. [93], which uses L-BGFS to construct adversarial examples. It does this by minimising the distance between a correctly classified x , and a new adversarial x' , such that x' is as close as possible to x , while still being misclassified.

The fast gradient sign method (FGSM), is an attack proposed by Goodfellow et al. [35]. It is based on linear perturbation of correctly classified samples, and suggests that "ANNs are too linear to resist linear adversarial perturbation". This attack is optimized for the L_∞ norm and is designed for speed instead of accuracy.

JSMA, the Jacobian Saliency Map Attack, proposed by Papernot et al. [74] is an attack based on linear estimation and Jacobian matrix analysis. It modifies each pixel of an input image one at a time and models the impact each pixel has on the resulting classification [75].

DeepFool, proposed by Moosavi-Dezfooli et al. [66], is an attack algorithm designed for L_2 distance metrics. It uses linear estimation and hyperplanes to visualise the classification boundaries of an ANN and uses this to converge on a true solution.

Defensive distillation is a type of training method designed to reduce vulnerabilities to attacks, introduced by Papernot et al. [75]. This training method significantly reduces the effectiveness of the attacks mentioned so far.

Following this work, Carlini & Wagner [19] introduce three attacks that are effective against defensively distilled networks. They propose three attacks for the L_0 , L_2 , and L_∞ norm adversarial distances. These attacks are sometimes referred to as Projected Gradient Descent (PGD). All attacks mentioned so far have been white-box attacks, but black box attacks are possible against ANNs, as outlined by Papernot et al. [73].

All the techniques we have discussed so far in this work are deductive verification techniques. A deductive verification technique refers to the desired property to be verified as known beforehand. Deductive techniques create an argument of safety based on a priori reasoning about the domain. This type of verification is the main focus of our work, as it provides guarantees about the behaviour of ANNs, however, creating an inductive argument, as well as a deductive one, has multiple benefits.

The last type of technique we introduce is referred to as simulation or test data augmentation methods. This type of verification is crucial to satisfying current safety standards. ANNs are often deployed in environments with uncertain and changing contexts of operation, such as autonomous vehicles. In addition, as quoted from the UK Department for Transport in 2015 by Johnson [46], “...no state has fully determined how existing traffic laws should apply to automated vehicles”. Therefore, deductive verification is not sufficient to fully demonstrate safety.

Tian et al. [94] and Pei et al. [78] outline two approaches, referred to as DeepTest and DeepXplore, respectively, for data augmentation. Both of these techniques define neuron coverage and domain-specific transformations as metrics. Neuron coverage refers to the number of neurons activated by a set of test inputs, and domain-specific transformations are transformations that are likely to be encountered in that domain. Both approaches use image domains as examples, and these include transformations such as rotation, translation, scale, and others. Central to both techniques’ operation is that they generate synthetic test data based on these domain-specific transformations that induce a higher neuron coverage, testing more of the decision logic of the ANN.

Fremont et al. [33] define a language for defining probabilistic relations for deep neural network systems. They define syntax and semantics for defining a wide range of scenarios that could be encountered by deep neural network systems. In particular, they use self-driving cars as an example of how their language can define realistic scenarios for simulation.

The approach by Dreossi et al. [23], implemented in the tool AnalyzeNN, is based on test data augmentation using counter-examples. That is, generating domain-specific images using a synthetic image generator, this image generator uses similar methods to those described in [78, 94, 33]. In contrast to these techniques, however, it uses this image generator to find counterexamples and only augments the test data with these counter-examples. This approach also provides an error table to attempt to explain the reason why counter-examples are misclassified. It does this by enabling the analysis of various features of that counter-example.

2.6 Tool Comparison

In this section, we compare, via experiments, available tools for verifying neural networks. Each tool implements one or more of the techniques discussed in Sections 2.3 and 2.4. First, we introduce our comparison criteria and the tools we have selected. In Section 2.6.1, we

provide an introduction to our experiments. In Section 2.6.2, we detail the methods implemented by each tool. In Section 2.6.3, we provide the results of our evaluation experiments. Finally, in Section 2.6.4, we discuss and evaluate our results.

2.6.1 Overview of Experiments

We have selected four criteria to compare the tools: usability, scalability, precision and applicability. Usability is concerned with the facilities for the definition of a property to verify and of the neural network. Scalability explores how the tools deal with larger and more complex networks. Precision refers to whether the results are precise enough to provide proof of the property. Finally, applicability refers to the type of networks that can be verified.

We explore, in our experiments, neural networks for control systems as these are the most relevant in robotics. Examples include a robotic arm network [99] and an airborne collision avoidance system [48]. We are interested in verifying properties such as the robotic arm not reaching an unsafe zone [99] or, if the intruder aircraft is sufficiently far away, the network advises ‘clear of conflict’ [48]. We focus our comparison on linear properties. They can capture a wide range of specifications, including control safety, adversarial robustness, and robotic controller properties. Furthermore, the vast majority of techniques are concerned with linear properties [88, 90, 89, 53, 34, 96, 97, 48].

In addition, our comparison is based on networks with ReLU activation functions. They are widely used, powerful, and easily trained [71], and every verification method and tool, as far as we know, is applicable to ReLU networks. Its piecewise linear nature is the basis for the feasibility of most techniques.

The tools we have selected are those that contain functionality to verify generalised linear specifications: *Matlab Toolbox for Neural Network Verification* (NNV) ⁴, *ETH Robustness Analyzer for Neural Networks* (ERAN) ⁵, *NeuralVerification.jl* ⁶, *Reluplex* ⁷, *Marabou* ⁸, and *SHERLOCK* ⁹. We evaluate them on three benchmark properties of two networks: two properties of a small neural network (TN) based on the implementation of a real robot, and property one of Network 1, 1 of the ACAS Xu neural networks.

TN is a feed-forward and deep neural network with two hidden layers of 32 nodes each, with two input and one output node. This network shares key features with control networks: low input and output dimensionality, and feed-forward ReLU structure. It also has a low number of hidden layers and class-invariant input zones, which can be represented as linear properties.

Network 1, 1 of ACAS Xu refers to one of the networks of the Airborne Collision Avoidance Systems for Unmanned Aircraft (ACAS Xu) first presented by Katz et al. [48] in 2017. The ACAS Xu networks are a set of 45 neural networks critical for ensuring unmanned aircraft avoid aerial collisions. It is desirable to establish properties to guarantee the behaviour of these networks under certain input domains, so ten properties have been identified.

Next, we introduce the methods implemented by each tool.

⁴[//github.com/verivital/nnv](https://github.com/verivital/nnv)

⁵[//github.com/eth-sri/eran](https://github.com/eth-sri/eran)

⁶[//github.com/sisl/NeuralVerification.jl](https://github.com/sisl/NeuralVerification.jl)

⁷[//github.com/guykatzz/ReluplexCav2017](https://github.com/guykatzz/ReluplexCav2017)

⁸[//github.com/NeuralNetworkVerification/Marabou](https://github.com/NeuralNetworkVerification/Marabou)

⁹[//github.com/souradeep-111/sherlock](https://github.com/souradeep-111/sherlock)

2.6.2 Methods

NNV, ERAN, and SHERLOCK are output range analysers [3, 1, 9]. Given an input range, they compute an output range, and if it falls within the safe output zone of the property, the network is determined safe. This computation can either be complete or incomplete, giving the output range or an overestimation of the output range. NNV and ERAN use sets to compute the output range.

NNV has implemented multiple methods for output range analysis: complete output range verification in *exact-star* and *exact-poly*, incomplete output range analysis in *approx-zono*, *approx-star*, *abs-dom* (which uses polytopes) and *approx-hr* (which uses hyper rectangles). We consider them all here.

ERAN has three modes of operation. Two of them, namely, L_{inf} and geometric analysis, are applicable to image domain specifications, so are not considered here. In the third mode, Linear specifications are defined using zonotopes, a centrally symmetric polytope [61]. ERAN defines zonotopes using affine arithmetic, an extension of interval arithmetic. For analysing zonotopes, ERAN has two methods: a hybrid analysis *RefineZono*, and an incomplete output range analysis *DeepZono*. We consider both.

SHERLOCK utilises conjunctions of linear inequalities as input [9]. SHERLOCK combines gradient-based local search with MILP solving: it solves a series of MILP feasibility problems with local search steps. SHERLOCK, however, is only applicable to networks with a single output node [25]. The implementation we evaluate is that in <https://github.com/souradeep-111/sherlock>, as it has proved impossible to build the newer version in https://github.com/souradeep-111/sherlock_2, even with support from the authors.

All the above tools do not require an output set to be defined as they are output range analysers. Defining an unsafe zone after computation can be useful for analysing the generated output range sets, but is not necessary.

Reluplex and Marabou are SMT solvers for neural networks [48, 49, 8, 2]. The implementation of Reluplex we have evaluated is the proof of concept implementation in [8]. They find an activation, a single value within the bounds of the problem given. As they compute a single point, the property is inverted, and it is proved if no activation that satisfies the inverse property is found. The input and output constraints are defined using multiple linear inequalities.

NeuralVerification.jl implements 17 different verification methods, both SMT solvers and output range analysers. Inputs are polytopes and hyper-rectangles, and as output, it uses half-spaces and polytope complements [4]. Polytope complements define outputs for SMT solvers, and halfspaces are used for the output specification for output-range analysers. However, we have been able to run only five of these methods: SHERLOCK (the algorithm and tool are referred to by the same name), BaB (an SMT solver), Duality, ExactReach, and Ai2. Of these, only BaB and SHERLOCK ran on both properties of TN.

In the next section, we discuss these tools and methods in more detail considering our evaluation criteria and experiments.

2.6.3 Evaluation

As mentioned, we evaluate the tools based on four criteria: usability (Section 2.6.3.1), scalability (Section 2.6.3.2), precision (Section 2.6.3.3) and applicability (Section 2.6.3.4).

2.6.3.1 Usability

Specification of Properties In the following, we describe how an I/O property of a neural network can be defined in each tool.

NNV I/O properties in NNV are defined by the input-set object used by their reach methods. Defining an unsafe output set, however, is a simpler way of evaluating the generated output reachable set. Each set is defined as a custom MATLAB object. The simplest way to build these objects is to build a hyper-rectangle based on a lower and upper bound vector, and then convert this using NNV's built in methods. If an output set is defined, NNV is also able to generate an intersection with the generated reachable set with the safe output set.

ERAN The input property is defined as a zonotope in affine form. This allows any convex polytope to be defined through shared error terms, but for linear properties, we define a simple hyper-rectangle input in affine interval form.

NeuralVerification.jl Properties are defined through the creation of input and output set objects, in a similar manner to NNV. NeuralVerification.jl also uses an output set with reachability methods.

There are four implemented types of sets used as I/O definitions; these are hyper-rectangles, halfspaces, hpolytope (halfspace polytopes) and hpolytope complements. Hyper-rectangles, halfspaces and polytopes are the most commonly used; polytope complements are used for the output sets for solvers. This is because the output set needs to be the negation of the safe zone, and a PolytopeComplement of a closed set is necessarily an open set, and non-convex.

Reluplex There is no distinct method to define properties; the code that defines the method has to be modified to deal with the desired input.

Marabou This implementation requires properties to be defined using a set of inequalities, relating to the input and output nodes. The input nodes are defined as x_n , and the output nodes as y_n , where n is the index of the node.

SHERLOCK The input is defined through input constraints as a hyper-rectangle. There is no way to define output zones, and properties are verified by analysing the computed output range.

Neural Network File Formats The file formats used to define neural networks vary, but the key information which all file formats support is: the weight matrix and the bias vector for each layer. We primarily focus on human-readable neural network file formats, since only ERAN is able to natively support non-text based file formats such as ONNX [7],

Tensorflow .pb and .meta files [1]. NNV is also able to support these, but through an extension, NNVMT [6].

NNet Format The nnet file format was created in 2016 to define the ACAS networks in a human-readable format [5]. It contains normalization information and the structure of the network, including input and output dimensions. The primary drawback of this format is that it is only applicable to *feed forward fully connected ReLU networks* [5]. The nnet file format is utilised by Reluplex, Marabou and NeuralVerification.jl.

ERAN Text Data Formats ERAN’s text-based formats are TensorFlows .tf and PyTorch’s .pyt. Unlike nnet files, they can define wider types of activation functions, including *tanh*, *Sigmoid*, and *Affine* functions. They can also define convolutional networks. Neither file format, however, is readable because the input and output dimension of the network has to be inferred from the weight matrices themselves. Also, these file formats do not contain a definition of the structure of the hidden layers, which has to be inferred from the matrices. Only .pyt files contain inherent normalization information.

SHERLOCK Format SHERLOCK utilises a custom file format that defines solely a feed-forward ReLU network. The format defines a network neuron by neuron for small networks. This can provide readable information about the internal structure of the network because it flattens the weight matrix for each layer and delimits the weights via new line characters. For larger networks, however, this is not a readable file format.

NNV NNV builds networks layer by layer, given the activation function, weight matrix and bias vector for each layer. This information can be loaded through any file format supported by MATLAB, for example, mat, csv and txt files. There is, however, no standard neural network file format support without the use of NNVMT.

A property can be defined by inequalities [8, 2, 9], or convex sets [3, 1, 4]. A neural network is defined by associating a weight matrix, a bias vector, and an activation function for each layer, which can be defined node by node, [9], layer by layer [3], or through nnet, pyt or tf files [8, 2, 1, 56].

2.6.3.2 Scalability

We have installed and built the tools on Linux version: ‘Ubuntu 18.04.2 LTS’, apart from NNV which was installed on MATLAB 2019a, Windows version¹⁰. We have used an i5-8265U CPU with 8GB RAM, with the timeout for each experiment set to 2 hours. We repeated each experiment that terminated 10 times.

For the NeuralVerification.jl implementations, we have obtained impossible results using Reluplex. In addition, we have experienced problems executing ExactReach and Ai2 on both benchmark networks.

RefineZono is a hybrid method; results are for RefineZono with its complete part at one second, the default value. We have also ran RefineZono with a 1000-second timeout, however, it was still unable to obtain tight enough bounds to verify property one of ACAS Xu. Finally,

¹⁰www.mathworks.com/products/matlab.html

Table 2.2: Scalability Results (Seconds, 2dp)

Tool:	TN P1:				TN P2:				AX11 ₁ :				
	Method:	Mean	Min	Max	RES	Mean	Min	Max	RES	Mean	Min	Max	RES
NNV	Exact-Star	95.18	77.33	122.57	SAT	73.4	55.72	93.57	SAT	T/O	T/O	T/O	T/O
	Exact-Poly	*	*	*	ERR	*	*	*	ERR	T/O	T/O	T/O	T/O
	Approx-Star	4.83	3.94	6.81	AMB	4.67	4.67	5.78	AMB	42.4	30.83	55.38	AMB
	Zono	0	0	0.01	AMB	0.01	0	0.01	AMB	0.03	0.02	0.04	AMB
	Abs-dom	4.94	3.88	6.81	AMB	4.71	3.7	5.81	AMB	40.86	19.55	49.09	AMB
ERAN	Approx-Hr	0.01	0	0.05	AMB	0.01	0	0.02	AMB	0.01	0.01	0.02	AMB
	DeepZono	0.06	0.06	0.07	AMB	0.06	0.06	0.06	AMB	0.15	0.09	0.16	AMB
	RefineZono	3.42	3.24	3.6	SAT	3.85	3.67	4.04	SAT	38.42	37.21	39.47	AMB*
SHERLOCK	SHERLOCK	*	*	*	ERR	0.9	0.72	1.08	SAT	N/A	N/A	N/A	N/A
NeuralVerification.jl	SHERLOCK	28.64	28.22	29.19	SAT	43.81	41.68	46.12	SAT	N/A	N/A	N/A	N/A
	BaB	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O	N/A	N/A	N/A	N/A
Marabou	Marabou	1.21	0.75	1.52	SAT	0.44	0.43	0.48	SAT	130.84	130.22	134.22	SAT
Reluplex	Reluplex	*	*	*	ERR	*	*	*	ERR	1348.9	1119	2620	SAT

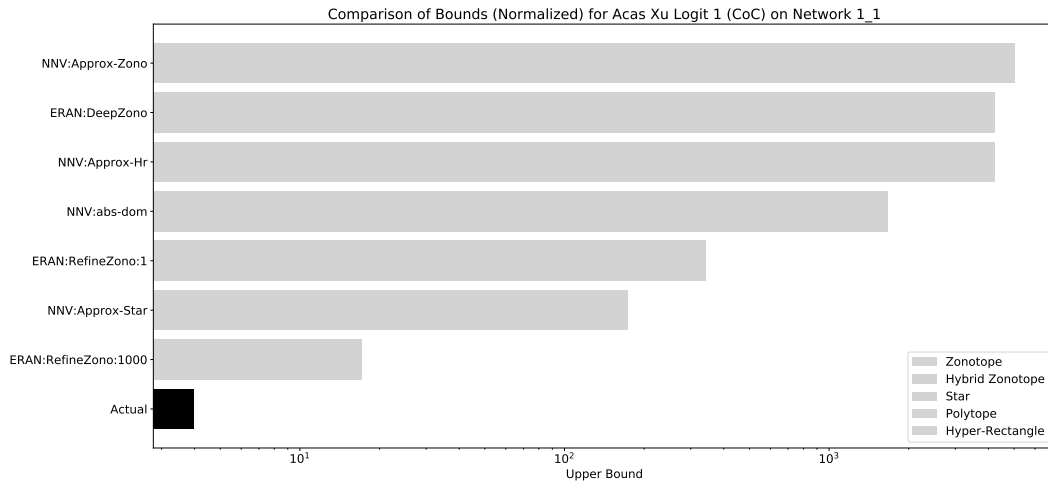


Figure 2.9: ACAS Xu P1 Bounds Comparison

we ran RefineZono using its full complete mode, although this timed out. The result of RefineZono with milp timeout 1000 is discussed in the next section.

Table 1 displays the results of the experiments on the three benchmarks. The ‘RES’ column displays the result of the method: ‘SAT’, the property is satisfied; ‘AMB’, the result of the method was ambiguous, that is, the bounds were not tight enough to prove the property; ‘ERR’, the method did not execute on the property and the time is displayed as *; ‘T/O’ represents a time out result.

Table 1 shows that Marabou is the only tool able to verify all three properties; NNV and ERAN can verify TN but only using complete methods, and SHERLOCK can verify P2 but produces errors on TN P1.

2.6.3.3 Precision

In this section, we evaluate the quality of the output bounds produced by the incomplete methods implemented by the tools. These bounds may not be tight enough to prove the property in question.

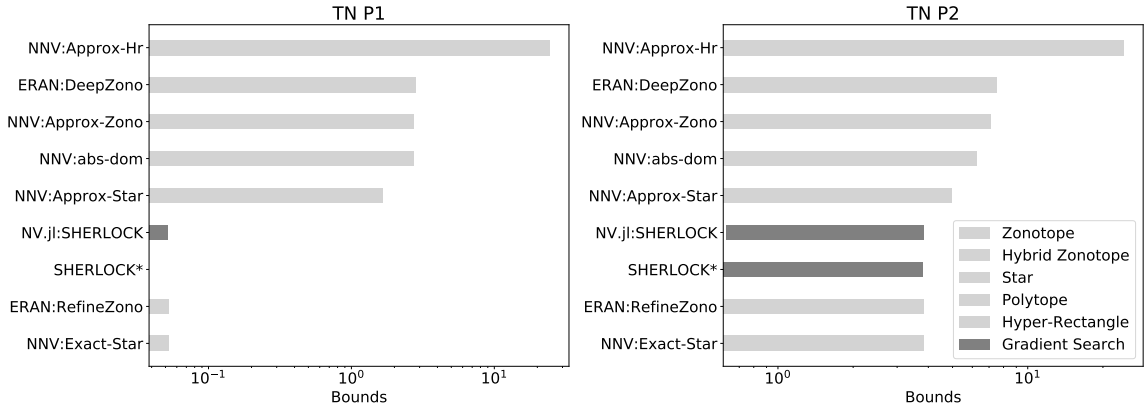


Figure 2.10: TN P1 and P2 Bounds Comparison

Figures 1 and 2 show the positive bounds generated by the methods of the tools using logarithmic scaling. Some techniques generate negative bounds, but we record only the positive bounds, as all networks have ReLU output layers. In addition, the bounds given by complete methods are displayed for reference.

One of our main observations is that the type of convex set used by a method significantly impacts the bounds generated. Ordering them from least to most precise, we have hyper-rectangles, zonotopes, polytopes, and star sets. One exception is that hyper-rectangles are slightly more precise than zonotope methods on ACAS Xu. RefineZono also involves an additional MILP component and a zonotope abstract domain formulation.

2.6.3.4 Applicability

Marabou and Reluplex are applicable solely to feedforward fully connected ReLU networks, as the modified Simplex algorithm they are based on does not allow for other types of nonlinearity. SHERLOCK also exclusively applies to ReLU networks and contains the additional limitation of only one output node.

ERAN implements abstract domain methods and so is the most widely applicable tool. It applies to ReLU, Sigmoid, and Tanh activation functions and to feedforward convolutional and residual layers.

NNV’s methods vary as to their applicability. The exact techniques only apply to ReLU or linear activation functions, while their approximate reachability modes also apply to sigmoid and tanh activation functions. Both types of methods are also applicable to convolutional layer types.

Of the methods we considered of NeuralVerification.jl, those applicable to only ReLU networks are BaB, Reluplex, ExactReach, ConvDual and SHERLOCK. BaB is also only relevant to one output. The implementation of Ai2 applies only to ReLU networks. However, Ai2 applies to further types of networks [34]. Duality applies to any monotone activation function.

2.6.4 Discussion

Marabou is the most successful tool for verifying ReLU networks, overall, by our criteria. It is the most usable, scalable and precise, but is one of the most limited tools in applicability.

The most usable method for defining properties is Marabou's. It is the only tool with a dedicated file format for the specification of a complete property. ERAN has a file for specifying a zonotope in affine form, but this is only the input to the network, not a complete property definition.

Further conversion tools would be helpful for usability, such as a converter from linear inequality to set representation, converters between various models of sets, and a converter between generator-form zonotopes and affine-form zonotopes, as used by ERAN and NNV, respectively. These facilities could facilitate obtaining complementary results from multiple tools.

In terms of defining networks, `nnet` files are the most usable. This is for two main reasons: they display the network structure efficiently and contain normalisation information. In some situations, however, viewing the exact weightings for each node with a node-by-node file structure may be helpful. This can give further details into the weighting assigned to each feature of the data, although this is only feasible with small, simple networks.

Marabou is the most scalable tool for complete verification, obtaining a runtime on TN 2.8x faster than ERAN, the following fastest complete tool. It has a runtime 78x faster than NNV's complete analysis on TN, and NNV timed out in the verification of ACAS Xu. The only means for complete verification that terminated ACAS Xu Property 1 were Reluplex and Marabou.

The runtime of these tools was influenced by the bounds of the property chosen for the network. This is minor in most tools, but Marabou experienced a difference of 2.71x from TN property 1 to property 2. For comparison, the most significant variation other than Marabou was SHERLOCK, with a variation of 1.52x.

The most precise tool for incomplete analysis is ERAN (utilising `RefineZono`); this, however, is also the least scalable incomplete method. Only Marabou, ERAN and SHERLOCK are precise enough to verify TN properties, and only Marabou and Reluplex are precise enough to verify the Acas Xu property.

In terms of verifying ReLU neural networks, all tools apply to all networks. Still, `BaB`, from `NeuralVerification.jl` and SHERLOCK, cannot verify networks with more than one output node. This is a limitation of the algorithms defining their methods. ERAN and NNV are the only tools applicable to types of neural networks beyond feedforward fully connected networks.

NNV, while not quite the most scalable, precise or applicable, has the highest degree of functionality of all tools presented. It is an output range analyser and has methods to generate the unsafe input zones of a problem, a functionality no other method can perform. One advantage of this is that the unsafe input zone can be used as an adversarial input generator for robust training [96]. Furthermore, NNV contains output visualisation and intersection methods, functionality also unique to NNV. This enables a clear representation of the decision logic of a neural network.

Based on the examples we have considered regarding usability, the best tool is Marabou and the worst is SHERLOCK. For scalability, the best is NNV, and the worst is Reluplex. For precision, the best is Marabou, and the worst is NNV. Lastly, for applicability, the best is ERAN, and the worst is SHERLOCK.

2.7 Final Considerations

Formally verifying ANNs is a challenging problem. Currently, there is no accepted optimal method for ANN verification: in general or even for any specific type of ANN. In this work, we want to verify feedforward fully connected networks for use in RAAI systems. The optimal method cannot be determined entirely, theoretically, as the runtime is based on individual model parameters. These determine the sections that can be pruned, affecting the runtime and precision of the method.

Our comparison of tools suggests, first, possible ways of combining tools so that their results complement each other and increase the explainability of the network. Second, we conclude that tool combination and integration is an exciting avenue for future work. Finally, we note that a unified format of representing linear inequalities is helpful in the development of verification tools.

There also are a few accepted benchmark problems for ANN verification. MNIST or CIFAR-10 are often used. However, the ANNs produced for each dataset may vary, making the comparison more challenging. The need for further empirical comparisons and benchmarks is clear.

Our overall goal is to determine a verification system for complete RAAI systems. To do this, we require a model of a complete RAAI system involving ANNs; from there, we can determine a verification technique for the entire system. We discuss this in the following chapter.

Chapter 3

Neural Networks in RoboChart

In this chapter we present an ANN modelling approach integrated with RoboChart. Following from Chapter 2, the ANNs we describe in RoboChart are pre-trained ANNs composed of feed-forward, fully connected layers with ReLU or linear activation functions. In this chapter, we give an overview of RoboChart in Section 3.1, followed by an approach to integrate ANN components into RoboChart in Section 3.2. We then provide an overview of the RoboChart semantics in Section 3.3, define a CSP semantics for our ANN components in Section 3.4, and present its validation via simulation, to complement early simple checks via model checking and assertion reasoning, in Section 3.5. We make final considerations in Section 3.6.

3.1 RoboChart

Throughout our work, we use a simple example of a two-wheeled Segway robot to provide a concrete example of the use of RoboChart (a full description of RoboChart can be found in [101]). The segway consists of: an inertial measurement unit (IMU), and two wheels, each with a Hall effect sensor: a sensor which detects magnetic fields; it is used in the Segway for monitoring the speed of the wheels. IMUs combine magnetometers, accelerometers and gyroscopes to measure the segway's tilt. The software uses this data with three PID controllers to update the velocity of the wheels in a smooth and consistent fashion such that the Segway balances.

RoboChart is a state-machine based modelling language that can be regarded as a profile of UML; it defines an abstraction of an entire robotic control software. The RoboChart language abstracts the functionalities of the hardware using robotic platform components. Controllers and a robotic platform are grouped together in a Module, which represents the complete robotic software [101].

Figure 3.1 describes the segway module. The robotic platform `SegwayRP` communicates with the controller `SegwayController` using asynchronous communication along six connections, each of type `real`, representing the communication of the IMU data and Hall effect sensors. `SegwayController` interacts with its state machine `BalanceSTM` using identical events and types, transferring the information sent from `SegwayRP` to `BalanceSTM`.

RoboChart controllers are in independent (parallel) operation with each other, and each

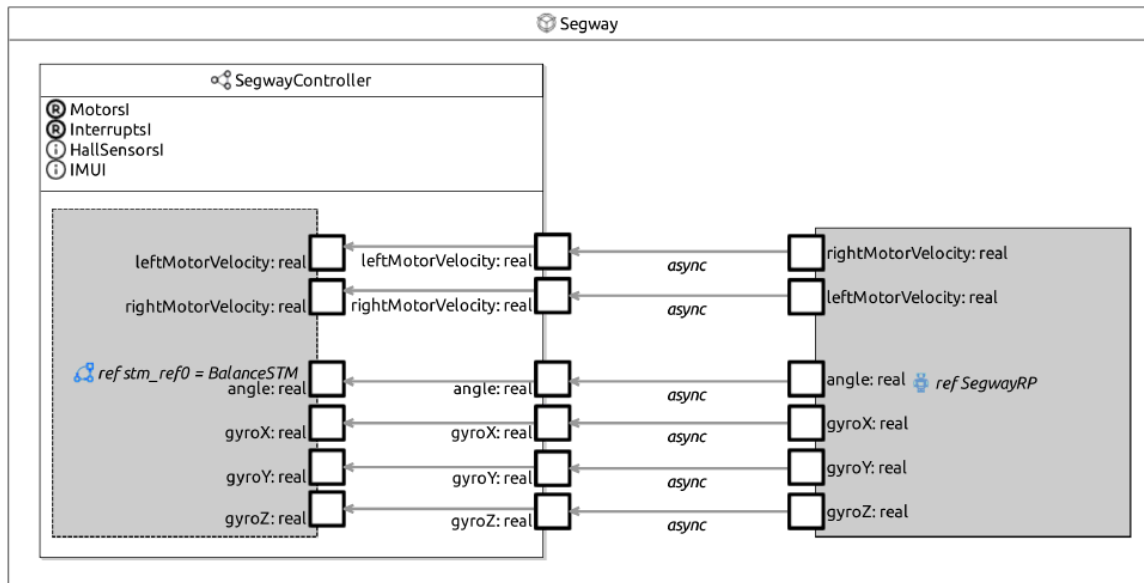


Figure 3.1: Segway module from [14], Legend: \textcircled{S} : module, \textcircled{C} : controller definition, \rightarrow : connection, \textcircled{R} : robotic platform reference, \textcircled{M} : state machine reference, \textcircled{I} : required interface.

individual state machine inside a controller is in parallel operation with other state machines inside a controller. Each controller must have at least one state machine, which represents the behaviour of the controller.

RoboChart components interact with each other via connections. These connections establish a source and a target node (state machine, controller or robotic platform) via their events. Connections can be asynchronous, where the data from the writer node is stored in a temporary buffer, and the writer is not blocked until the reader node is ready to receive. Connections can also be bidirectional, in which case, both nodes can read and write.

In RoboChart, controllers may communicate with other controllers or robotic platforms, and state machines may communicate with other state machines or their controller. Communication with a robotic platform is always asynchronous, capturing the fact that a physical robot is not blocked by the control software. Communication between controllers, however, may be synchronous or asynchronous. State machines communicate synchronously to model parallel threads of behaviour abstractly.

Events are used to represent the inputs and outputs of an individual component. They may be typed, representing data transfer, or untyped, representing a simple interaction such as a signal or interrupt.

State machines in RoboChart are similar to UML state machines, but have some features removed to facilitate the definition of a compositional semantics and enable formal verification. State machines are composed of states and junctions, where states define a stable configuration of the machine. Junctions capture unstable configurations such as decision points. It is possible to define nondeterministic behaviour using state machines, as more than one transition may be enabled from a state or junction.

RoboChart also allows the definition of operations: these represent functionalities provided

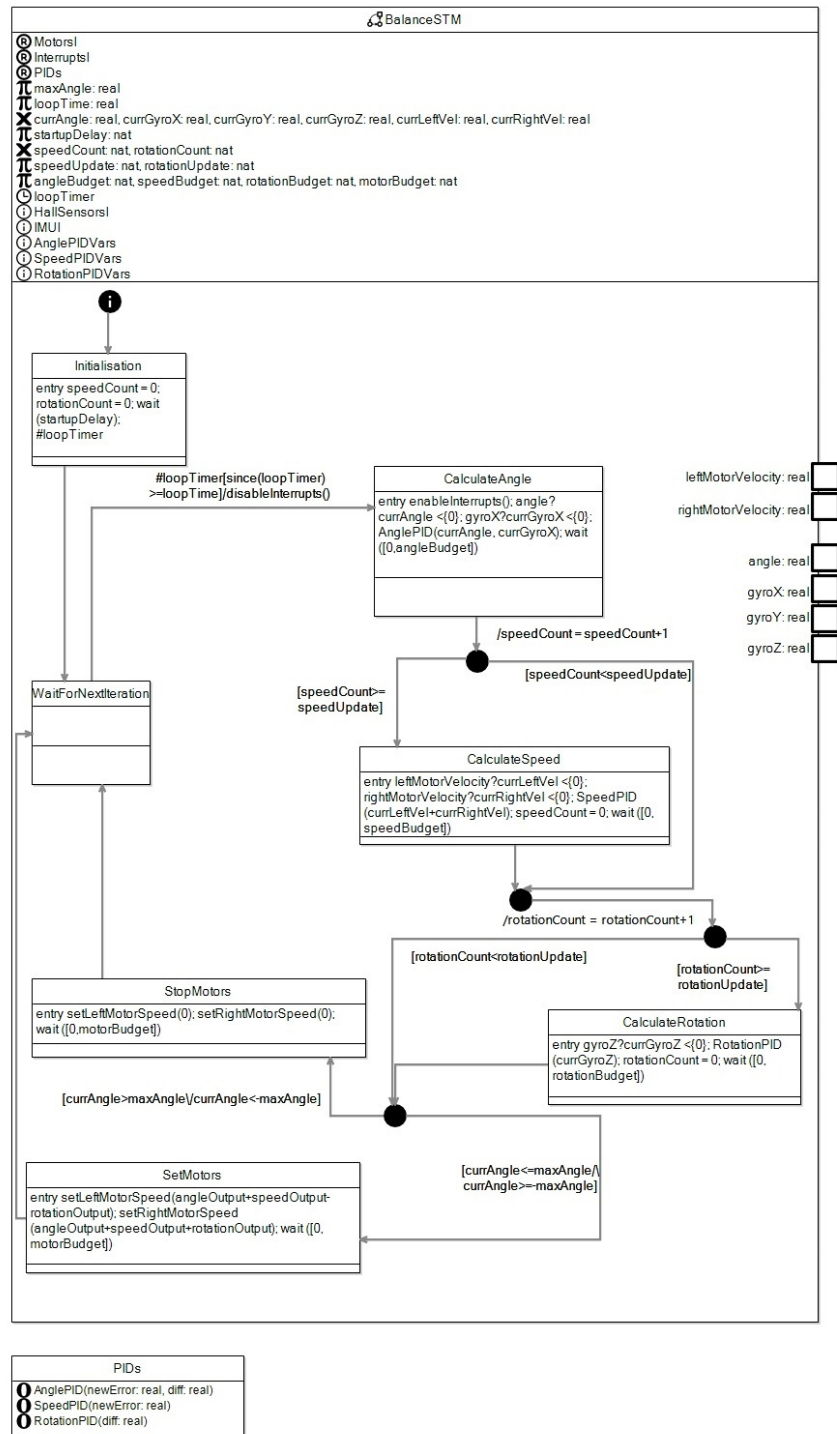


Figure 3.2: Segway model BalanceSTM state machine (final) from [14]. Legend: Ⓜ: state machine definition, ℝ: variable, ●: initial junction, →: transition, ○: operation declaration.

either by the physical robot or implemented as part of the control software. They can be defined either through libraries or through a state machine. Operations used in a state machine need to be declared in a required interface. An operation signature declares the name of the operation, its parameters, and their types. In Figure 3.2, the interface `PIDs` declares three operations (`AnglePID`, `RotationPID` and `SpeedPID`) each taking one or two real parameters.

Figure 3.2 defines `BalanceSTM`: a state machine that describes how the data received from the sensors is used to update the velocities of the segway's motors. `BalanceSTM` has six events that represent input received from the segway's sensors: `leftMotorVelocity`, `rightMotorVelocity`, `angle`, `gyroX`, `gyroY`, and `gyroZ`. The machine uses a clock `loopTimer` to define a time budget for each iteration, given by the `loopTime` constant. The machine defines further time constants `angleBudget`, `speedBudget` and `rotationBudget`; these represent the time for `AnglePID`, `SpeedPID` and `RotationPID` operations to complete, respectively. The machine also defines `startupDelay`, which represents the time for the hardware to initialise.

`AnglePID` is called every iteration, `SpeedPID` and `RotationPID` are called only every `speedUpdate` and `rotationUpdate`, iterations. The variables `speedCount` and `rotationCount` count the iterations for the purposes of calling `SpeedPID` and `RotationPID` at the correct iterations.

The machine starts in the state `Initialisation`, where `speedCount` and `rotationCount` are set to 0. The machine then waits for `startupDelay` time units and resets the clock `loopTimer` through `# loopTimer`. The machine then transitions to state `WaitForNextIteration`, where the machine waits for `since(loopTimer)` to be greater than or equal to `loopTime`. The expression `since(loopTimer)` represents the number of time units since `loopTimer` was reset; so, in `WaitForNextIteration` the machine waits for `loopTime` time units to pass before starting the next iteration. In this transition, `loopTimer` is reset, and there is an addition action `disableInterrupts()`. The clock operates using interrupts, which is also how the hall effect sensors in the robotic platform operate; so, to prevent interference, interrupts are disabled as the clock is reset.

The machine then transitions to the state `CalculateAngle`, where interrupts are enabled immediately, then `currAngle` and `currGyroX` values are read in via the events `angle` and `gyroX`. `AnglePID` is then called with these parameters, and the machine waits for a time between 0 and `angleBudget`: representing the time for `AnglePID` to complete.

There is one transition from `CalculateAngle` (with no condition) that increments `speedCount` by 1. The machine then moves to a junction, where the subsequent state is determined by whether `speedCount` is less than `speedUpdate` or not. If `speedCount` is less than `speedUpdate`, the machine proceeds to another junction; if it greater or equal to `speedCount`, however, then `BalanceSTM` enters the state `CalculateSpeed`.

`CalculateSpeed` first, similarly to `CalculateAngle`, reads in `currLeftVel` via `leftMotorVelocity` and `currRightVel` via `rightMotorVelocity`. `SpeedPID` is then called with the sum of the `currLeftVel` and `currRightVel` as a parameter. Lastly, the variable `speedCount` is reset to 0, and there is a nondeterministic wait to represent the time for `SpeedPID` to complete, as in `AnglePID`.

There is one transition out of `CalculateSpeed`, to the same junction as if `CalculateSpeed` was skipped. This junction has a single transition, which leads to another junction, concerning `RotationPID`, and increments `rotationCount`. This junction determines whether `CalculateRotation` is performed this iteration, depending on whether `rotationCount` is greater than or

equal to `rotationUpdate`. If it is, `BalanceSTM` enters `CalculateRotation`, and if not, the machine skips `CalculateRotation` this iteration and moves to the final junction of this iteration: a decision on how to update the motors this cycle.

The first action of the state `CalculateRotation` is to read the value communicated by the event `gyroZ` into the variable `currGyroZ`. Then, `RotationPID` is called with `currGyroZ`. Finally, similarly to the other states, the variable `rotationCount` is reset to 0 and there is a final nondeterministic wait for `rotationBudget` time units.

The final junction of `BalanceSTM` determines whether to stop or set the motors. In this model, if the segway's angle of tilt is within a range where we can restore balance, then the motor velocities are updated using the results from our PID components. In this case, we transition to the state `SetMotors`. Here, the angle of tilt is determined by the `currAngle` variable, which is read in from the `angle` event, and the recoverable angle range is defined as the range in between the constants `-maxAngle` and `maxAngle`. If the segway's tilt angle is not in such a range, then we stop the motors completely, which we represent by transitioning to the state `StopMotors`.

In `SetMotors`, we update the velocities of the motors via calls to the operations `setLeftMotorSpeed` and `setRightMotorSpeed`, which are accessed through the required interface `MotorsI`. Both operations are updated with the sum of the outputs from the PID controllers, stored in the variables `angleOutput`, `speedOutput`, and `rotationOutput`; for the left motor, however, `rotationOutput` is subtracted instead, compensating for the observed rotation by updating the wheels independently. There is a final nondeterministic wait for a period between 0 and `motorBudget` time units, which represents the time for the update operations to execute. Lastly, `BalanceSTM` transitions to `WaitForNextIteration`, in preparation for the next cycle.

In `StopMotors`, we stop the motors completely through calling the operations `setLeftMotorSpeed` and `setRightMotorSpeed` with the parameter 0. The machine then performs a final nondeterministic wait representing the time to update the motors: between 0 and `motorBudget`. `BalanceSTM` then transitions to `WaitForNextIteration`, as in `SetMotors`, to prepare for the next cycle.

We use the Segway example to illustrate our approach; in particular, we replace the `AnglePID` component with a neural network component. In the next section, we discuss the integration of ANNs into RoboChart, using the Segway example as described here.

3.2 ANNs in RoboChart

In this section, we describe how ANN components can be integrated into RoboChart. First, we provide an overview of these components via an example of a RoboChart model including ANN components in Section 3.2.1. Next, we describe our extension of the RoboChart metamodel to accommodate these components in Section 3.2.2. Finally, we define well-formedness conditions to characterise meaningful models described using the metamodel in Section 3.2.3.

3.2.1 Overview

We describe the structure of our framework to integrate ANN components into RoboChart in Figure 3.3. Here, we describe how we extend the RoboChart semantics to cater for these



Figure 3.3: The structure of our verification framework: the nodes in **Robotic System** represent the broad structure of a RoboChart model with an ANN component; the nodes in **Syntax** represent the artefacts we have developed to support modelling of ANN components; the nodes in **Semantics** represent semantic models of ANN components, and their implementations; finally, the nodes in **Verification** represent verification tools and frameworks.

ANN components. The result is a unified model that enables verification of the complete system. To support tractable verification, we consider only the ANN type discussed in Chapter 2: trained, fully-connected, ReLU activated ANNs.

RoboChart has a formal semantics [101] described in *Circus*, a combination of Z and CSP. Using *Circus* gives an indirect UTP semantics to RoboChart, since the denotational semantics of *Circus* is defined in UTP. A formal semantics in UTP enables proofs of the properties and behaviour of a system. Isabelle/UTP [31] mechanises UTP proofs; it is an implementation of UTP in the Isabelle/HOL proof assistant [69].

We define three encodings of our *Circus* semantics to enable automated verification using a variety of technologies: CSP, JCSP, and a preliminary encoding in Isabelle/UTP.

We illustrate our approach by discussing how ANNs can be integrated into our Segway example. It would be possible to train an ANN to replace every PID controller in our example, that is, we can train an ANN based on all five inputs of the three PID controllers. Our primary goal, however, is to discuss how an ANN can fit in the larger context of a system: how it can interact with other controllers or machines. To this end, we replace just one of the PIDs, namely, the AnglePID, with an ANN component.

We model an ANN in RoboChart by defining a dedicated component. We cater for two approaches: ANN components at the same level of controllers, and ANN components at the same level of operations. They allow for different forms of abstraction. In the former, the ANN executes in parallel with the other controllers, and communicates with them via events. In the operation-based approach, other components can trigger the execution of the ANN and block waiting for the result. Communication, in this case, is via parameter passing and shared variables. In both cases, timing specification is defined explicitly outside the ANN components.

To integrate an ANN controller into the Segway example, we first present a different version of its model that allows all PIDs to operate in parallel. It is feasible to add an ANN controller or an ANN operation to the original model presented in the previous section, but the parallel model presented here allows us to illustrate the idea in a simpler context regarding the semantics of RoboChart.

We first, in Section 3.2.1.1, provide a definition of the parallel segway model. Then, in Section 3.2.1.2, we describe how an ANN component interacting as an operation can be integrated into this model. Finally, in Section 3.2.1.3, we present an ANN component interacting as a controller, and how it can be integrated into the Segway model.

3.2.1.1 Parallel Segway Model

To allow parallel operation, we define each PID operation by a state machine that accepts its inputs through events communicating with BalanceSTM.

We present the parallel version of the module Segway in Figure 3.4; this module replaces all PID controllers with state machine components, named identically to their corresponding operation components: AnglePID, SpeedPID, and RotationPID. We also define a new controller, AnglePID_C, whose behaviour is defined by the state machine AnglePID. We add a controller in this module to demonstrate how we can replace a controller with an ANNController component.

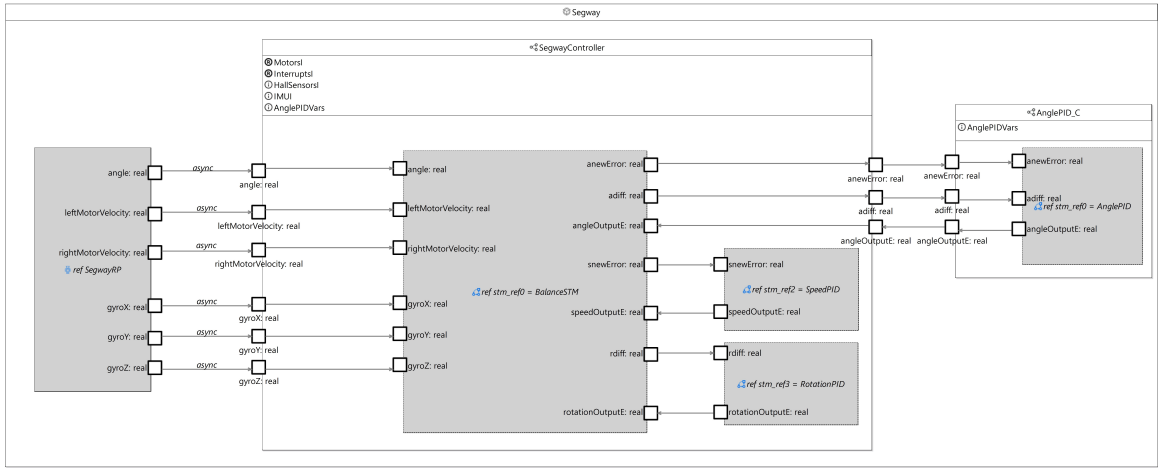


Figure 3.4: A parallel version of the Segway model. Legend: \oplus : Module, $\%$: controller definition, \rightarrow : connection, π : constant, \oplus : robotic platform reference, \oplus : state machine reference.

The input events contain a prefix character indicating which PID they connect to: the input of SpeedPID is `snewError`, the input of RotationPID is `rdiff`, and the inputs of AnglePID are `anewError` and `adiff`.

Figure 3.5 presents our parallel version of the BalanceSTM state machine. It adds the following variables to track whether BalanceSTM has sent input events to the PID state machines: `speedSent`, `angleSent`, `rotationSent`. We also define `angleReceived`, used to prevent multiple readings of the result from AnglePID in a single cycle: required because AnglePID is not timed in the same way as the rotation and speed PIDs.

The machine starts in `Initialisation`, where the entry action, in addition to what it executes in the original model, sets `speedSent`, `angleSent` and `rotationSent` to false. `WaitForNextIteration` and the corresponding transition is unchanged. The state it transitions into, however, is different. It is called `Setup` and its purpose is twofold: to increment the `rotationCount` and `speedCount` variables, and to set `angleReceived` to false.

The machine then moves to `ReceiveInput`, where most of the behaviour of this machine is defined. There are seven transitions from `ReceiveInput`: three to states whose entry actions send data and trigger a PID, three self transitions for receiving data from the PIDs, and one transition to a junction, exiting this state. Sending data is handled by the states `SendRotation`, `SendSpeed` and `SendAngle`. Transitions to these states are guarded by conditions on `speedSent`, `rotationSent`, and `angleSent`. This is to prevent consecutive redundant input events being sent to the PIDs.

The transition to `SendAngle` requires both `angleSent` and `angleReceived` to be false, to prevent multiple sends of AnglePID per iteration. These variables are both false at the start of the iteration. We set `angleSent` to true after the `SendAngle` state terminates, and is set to false once the result of AnglePID is received. The variable `angleReceived` records whether the output of the PID has been received in the current cycle. If it has been, if `angleReceived` is true, then AnglePID cannot be started again this cycle.

`SendAngle` reads in `currAngle` and `currGyroX` from the events `angle` and `gyroX`, then sends

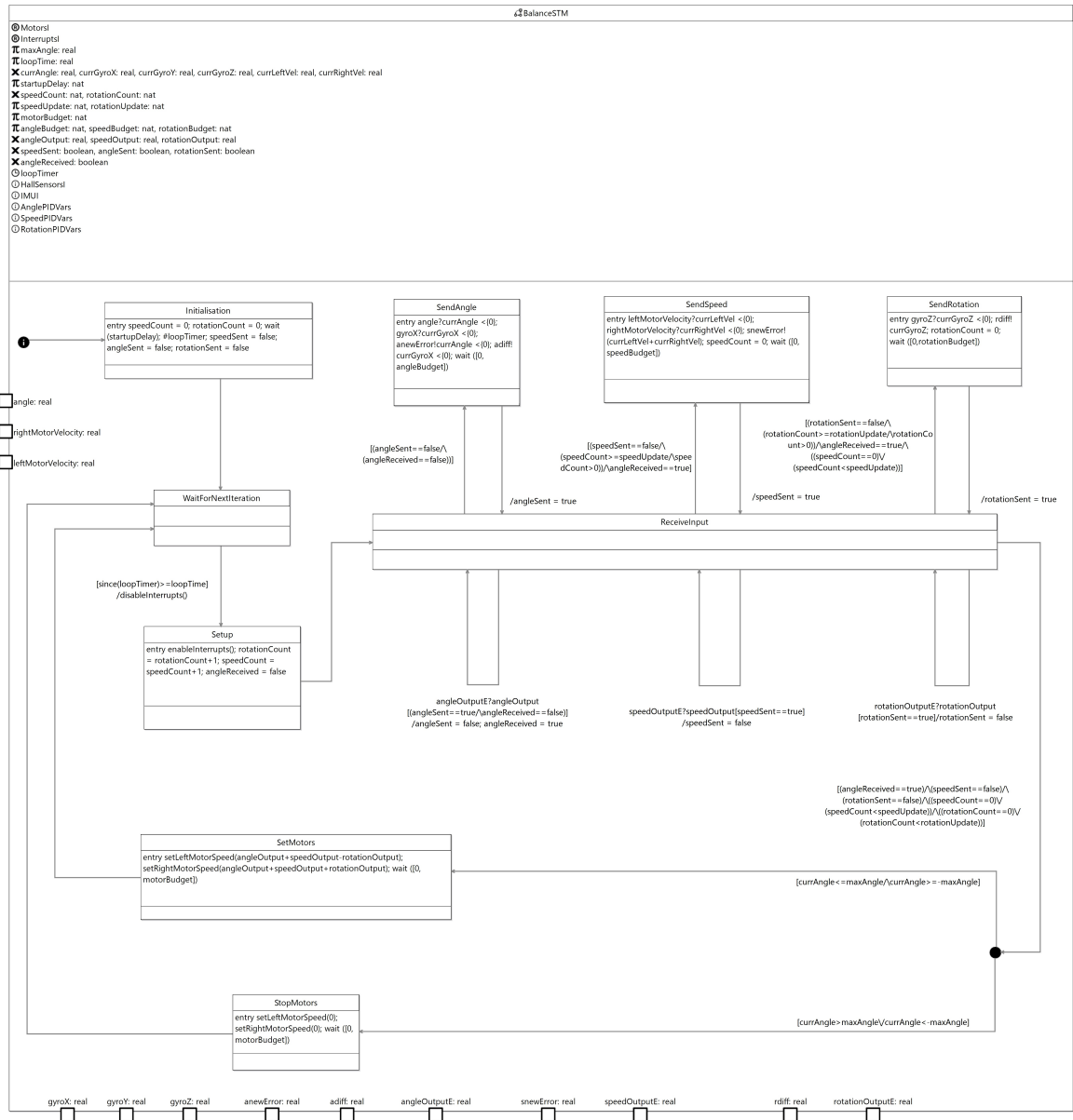


Figure 3.5: A parallel version of the BalanceSTM state machine. Legend: Σ : state machine definition, \times : variable, π : constant, \bullet : initial junction, \rightarrow : transition.

them via the `anewError` and `adiff` events to the `AnglePID` state machine. As an action of the transition from this state, `angleSent` is set to true.

The transition to `SendSpeed` is also guarded by the condition that `speedCount` must be greater than or equal to `speedUpdate`, and greater than 0. This means that, as `speedCount` is only incremented once every iteration, the speed PID is only executed every `speedUpdate` iterations. We also add the condition that `angleReceived` must be true, ensuring each PID is called in the correct order. In `SendSpeed`, the entry action reads in the data needed for `SpeedPID` (`currLeftVel` and `currRightVel`) then communicates their sum (like in `BalanceSTM`) to the `SpeedPID`, through an output via the event `snewError`. Next, `speedCount` is set to 0, to keep the speed PID operating on the correct iterations. Finally, `speedSent` is set to true as an action of the transition back to `ReceiveInput`.

`SendRotation` is similar to `SendSpeed`. Instead of `speedCount` and `speedUpdate`, `rotationCount` and `rotationUpdate` are used, and instead of `speedSent`, `rotationSent` is set to true. We also ensure correct ordering by requiring that `angleReceived` is true, and that `SpeedSent` either has terminated, or is not called this cycle through a condition on `speedCount`. `RotationPID` only requires one input to be read from the robotic platform, `gyroZ`, and this value is sent to `RotationPID` using the `rdiff` event.

Receiving data is handled using transitions from `ReceiveInput` to itself. The triggers are the output events of the PIDs: `speedOutputE`, `angleOutputE` and `rotationOutputE`. These are recorded in the variables `speedOutput`, `angleOutput` and `rotationOutput`. The guards require that each `sent` variable is true, as the machine should not receive output from a PID until it has called that PID. The second condition for `angleOutput` is that `angleReceived` must be false; this prevents performing multiple reads from `AnglePID` in a single cycle. This cannot occur for the `SpeedPID` and `RotationPID` state machines as they cannot send input data twice in the same iteration due to the speed and rotation counts.

The action of the receiving transitions resets the `sent` variables: `rotationSent`, `speedSent` and `angleSent` are set to false to indicate that the output has been received and new data can be sent. In the `angleOutput` transition, `angleReceived` is also set to true, indicating that `angleOutput` cannot be updated again this cycle.

The machine is timed to reflect the same design as the original `BalanceSTM`, presented in Figure 3.2. In each sending state (`SendAngle`, `SendRotation`, and `SendSpeed`) there is a nondeterministic wait operation to represent the time spent on the calculations.

The machine transitions to a junction when all required PID results are read. This is denoted by three conditions. The first is that output must have been received from all PIDs that have been called this cycle. `AnglePID` is called every cycle, so `angleReceived` must be true. `SpeedPID` and `RotationPID` may or may not have been called this cycle, but if they have been, and an output has not been received this cycle, then `speedSent` or `rotationSent` will be true. So, we require that `speedSent` and `rotationSent` are false. The second condition to terminate is, either `speedCount` is 0, or `speedCount` must be less than `speedUpdate`; `speedCount` is 0 on the cycles where `SendSpeed` has been called, and this should not occur when `speedCount` is less than `speedUpdate`. The next condition is identical but for `RotationPID`. These conditions ensure that the machine can only transition from `ReceiveInput` when all required PID results are read.

The junction, and the proceeding states, `StopMotors` and `SetMotors`, are identical to those

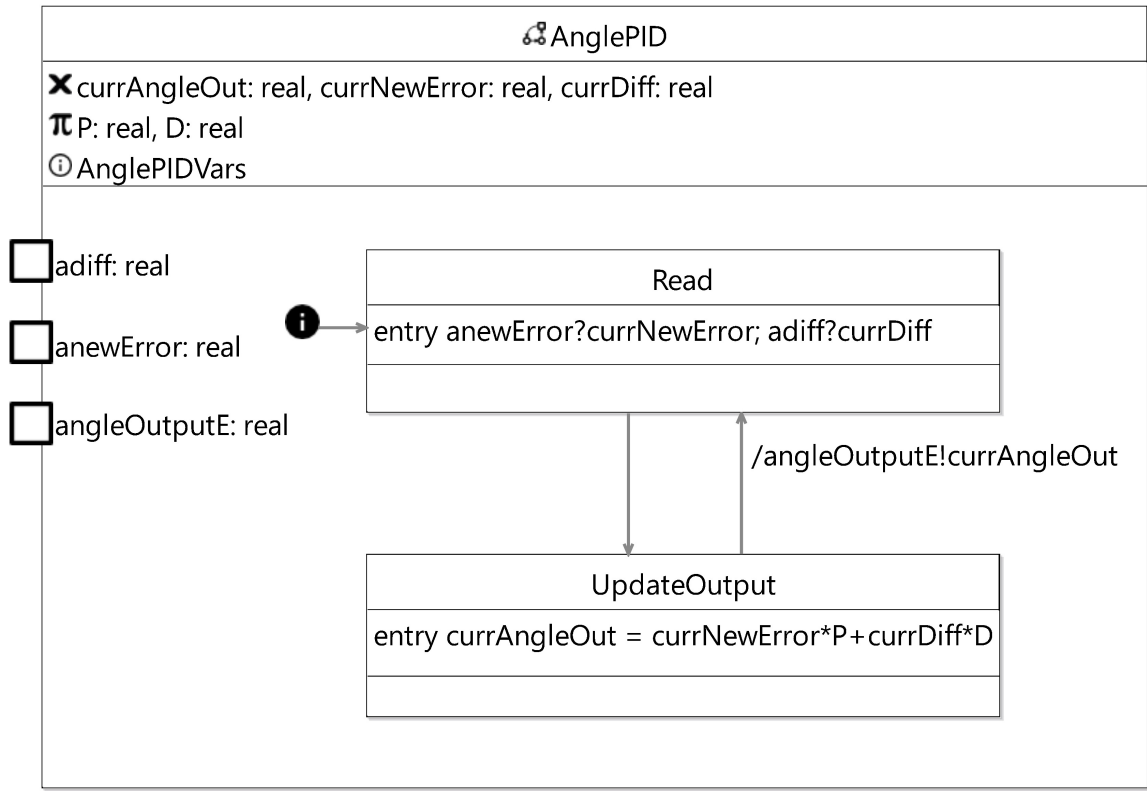


Figure 3.6: AnglePID defined as a state machine. Legend: \odot : state machine definition, \times : variable, \bullet : initial junction, \rightarrow : transition.

in the original model. The machine then proceeds to WaitForNextIteration and repeats the iteration.

The AnglePID state machine, in Figure 3.6, starts in the Read state, where inputs anewError?currNewError and adiff?currDiff are accepted. When both inputs are received, the machine transitions to UpdateOutput, where currAngleOut is calculated. This is the sum of the product of currNewError and P, representing the proportional constant, and the product of currDiff and D, the derivative constant. The machine then transitions back to Read; in the transition action, we communicate currAngleOut via the event angleOutputE.

The parallel Segway model is equivalent to the sequential Segway model presented in Section 3.1. That is to say, refinement holds both directions between the module Segway presented in Figures 3.1 and 3.4.

Next, we present how ANNs can be integrated into this model through the use of the ANN-Operation component.

3.2.1.2 ANN Operation component

Here, we define an ANN component that interacts with the system as an operation; Figure 3.7 displays an ANNOperation component: AnglePIDANN. This component interacts with the rest of the system through parameters. This allows the component to be invoked from within a state machine or another operation. In addition, pre-conditions and post-conditions

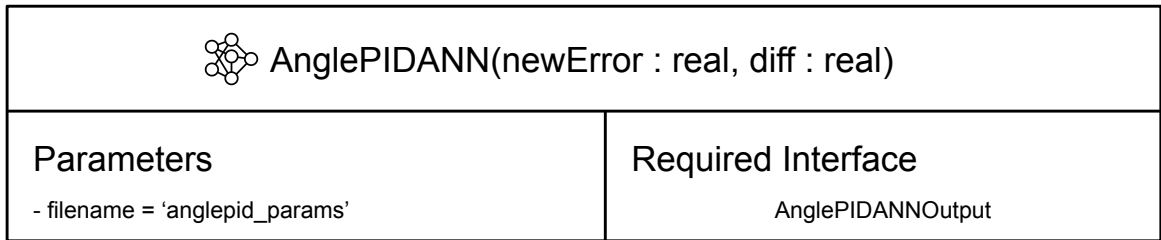


Figure 3.7: AnglePIDANN defined as an ANNOperation. Legend: : ANN component.

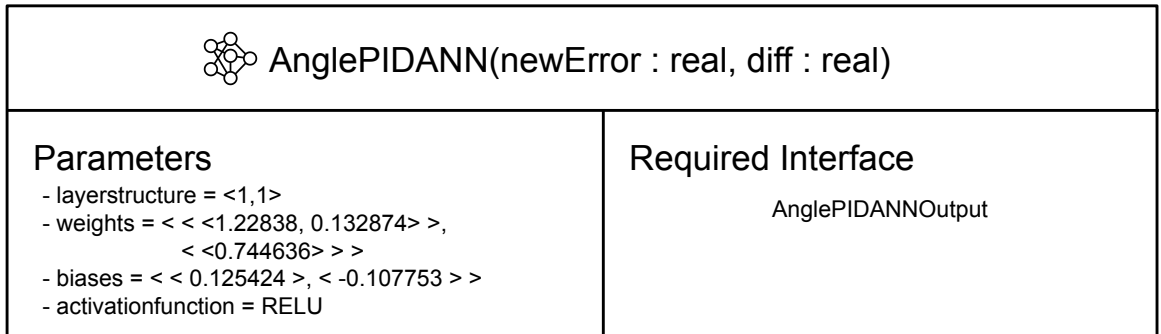



Figure 3.8: AnglePIDANN defined as an ANNOperation, with explicit parameters. Legend: : ANN component.

can be defined for the ANN component.

An ANNOperation is defined through populating two partitions of an ANNOperation block. For AnglePIDANN, the values of the parameters are shown in Figure 3.7. The first is its parameters, in the box labelled Parameters, we define this by setting filename to the string anglepid_params. The second partition defines the required interface, in the box labelled Required Interface, that ANNOperation's use to communicate the output of the ANN to the rest of the system.

The parameters of an ANNOperation, similarly to an operation definition in RoboChart, is defined in brackets after the name of of the component. In our example, the parameters of ANNOperation are newError and diff, both of type real.

The Parameters box of an ANN component can be filled in two ways. The first is through defining a filename parameter, from which the weights, biases, layerstructure, and activation-function parameters can be inferred. The second is defining the weights, biases, layerstructure,

```

1 ReLU,
2 1.22838e+00,1.32874e-01,
3 1.25424e-01,
4 7.44636e-01,
5 -1.07753e-01,

```

Figure 3.9: The anglepid_params csv file, used to define an ANN for AnglePID.

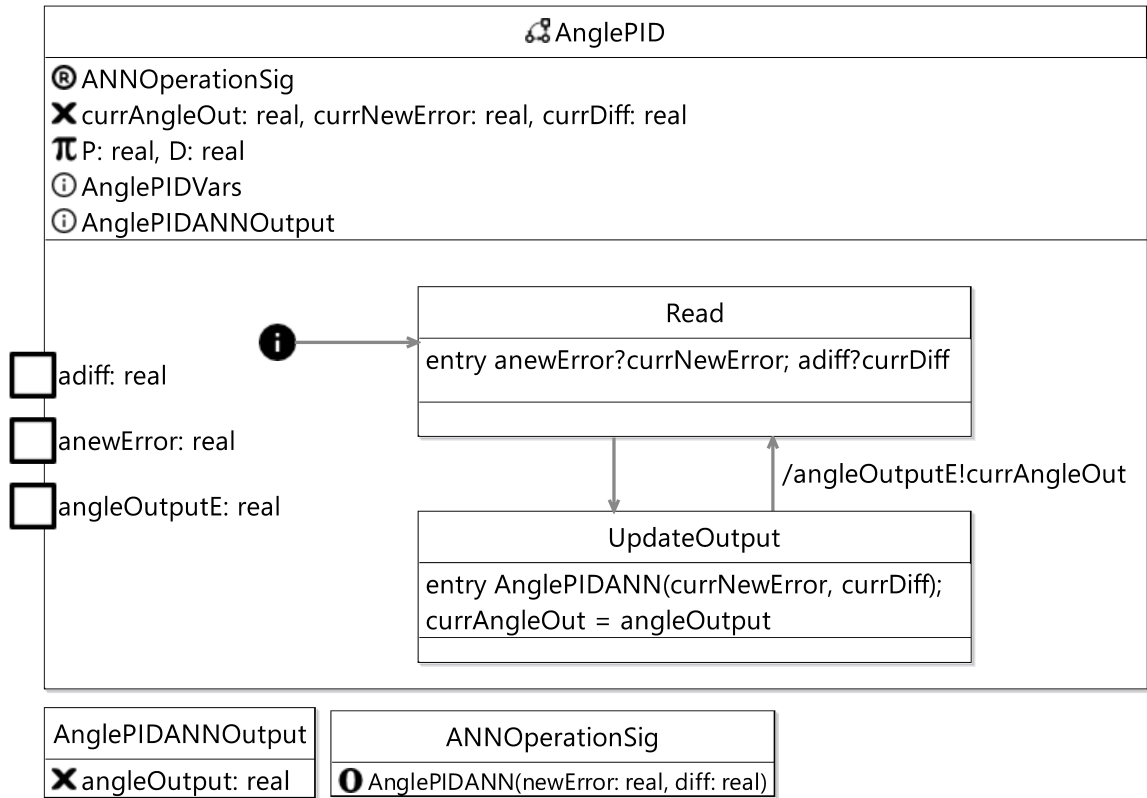


Figure 3.10: AnglePID defined using an ANN Operation component. Legend: ⚙ : state machine definition, \times : variable, \bullet : initial junction, \rightarrow : transition.

and activationfunction parameters explicitly inside the Parameters box. We demonstrate the first method in Figure 3.7, and the second in Figure 3.8.

We display the contents of the file 'anglepid_params' in Figure 3.9. A parameter file is a csv file containing the information required to define an ANN component. The weights and biases must be extracted from a trained ANN. We can, for example, use the Keras API ¹.

Keras is a widely-used and powerful deep learning API built on the TensorFlow machine-learning platform. Given a Keras model instance, weights and biases can be extracted using the method 'get_weights' ². This Keras model instance can be generated from the Keras 'H5' model format, or the 'SavedModel' format ³.

The values of the weights and biases should be real numbers listed in E notation with a maximum of 7 decimal places, and an exponent value between -38 and 38 . This is to support single-precision 32-bit floating point machine architecture. The number of supported precise decimal digits in single precision architecture is 7, and the maximum decimal exponent value is 38 [10]. In addition, the standard practice for training ANNs uses single-precision architecture [36]. There is, however, nothing in our approach that is dependent on these restrictions.

¹<https://keras.io/>

²https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer#get_weights

³https://www.tensorflow.org/guide/keras/save_and_serialize

The activation function is given as an enumerated type, one of `RELU` or `LINEAR`. The activation functions of an ANN are usually defined when the ANN is built, or, given a Keras model instance, can be displayed using the ‘summary’ method ⁴.

The parameter file starts with a single string defining the activation function, line 1 of Figure 3.9. The remainder of the file is split into groups of lines: one for each layer of the ANN, excluding the input layer.

In Figure 3.9, we have two groups: lines 2 to 3 for the hidden layer, and lines 4 to 5 for the output layer. Every group of lines has two sections. The first section defines the weights, where each line contains the weights for a single node, delimited by commas. This section has a number of lines equal to the number of nodes. The second section defines the bias values for the layer; this section also has a number of lines equal to the number of nodes, with each line defining a single bias value.

In our example, line 1 defines the activation function as ‘ReLU’. Line 2 defines the weights over just one line as there is one node, and providing two values because the network contains two inputs. Line 3 defines the bias value: a single value as there is a single node.

The second, and final, layer is defined in lines 4-5 in a similar way. This layer has a single input, as the output size of the previous layer is 1, so the weight value in line 5 contains a single value.

Figure 3.10 displays the modified version of `AnglePID`, described as a state machine, which uses the ANN component `AnglePIDANN`. This machine requires the interface `ANNOperationSig`, which defines the signature of `AnglePIDANN`, and uses the interface `AnglePIDANNOutput` to receive the output from `AnglePIDANN`.

Compared to the definition of `AnglePID` in Figure 3.6, the difference in the behaviour of the machine is that `currAngleOut` is calculated by a call to `AnglePIDANN`, instead of being calculated with an equation. In the state `UpdateOutput`, we first call `AnglePIDANN` with the parameters `currNewError` and `currDiff`, then we set `currNewError` to `angleOutput`, the output of `AnglePIDANN`.

Here, interaction with the ANN component operates through interfaces containing variables. Next, we discuss an alternative modelling approach, where communication with the ANN component operates via events.

3.2.1.3 ANN Controller component

The ANN can also be integrated at the controller level. Given the parallel version of the Segway model, communicating through events, we can introduce this type of interaction to this model without much change. This approach allows an ANN to interact in parallel with other controllers, which is useful for larger networks where computation time is a more significant factor, and for ANN components that interact with other ANN components.

We describe an example to this approach in Figure 3.11, where we replace the controller `AnglePID_C` with the ANN component `AnglePIDANN`. It interacts with `BalanceSTM` using the same events as the controller `AnglePID_C`. In this way, the only aspect of the model that needs to be changed is the definition of the controller, just to connect to another (ANN)

⁴<https://keras.io/api/models/model/>

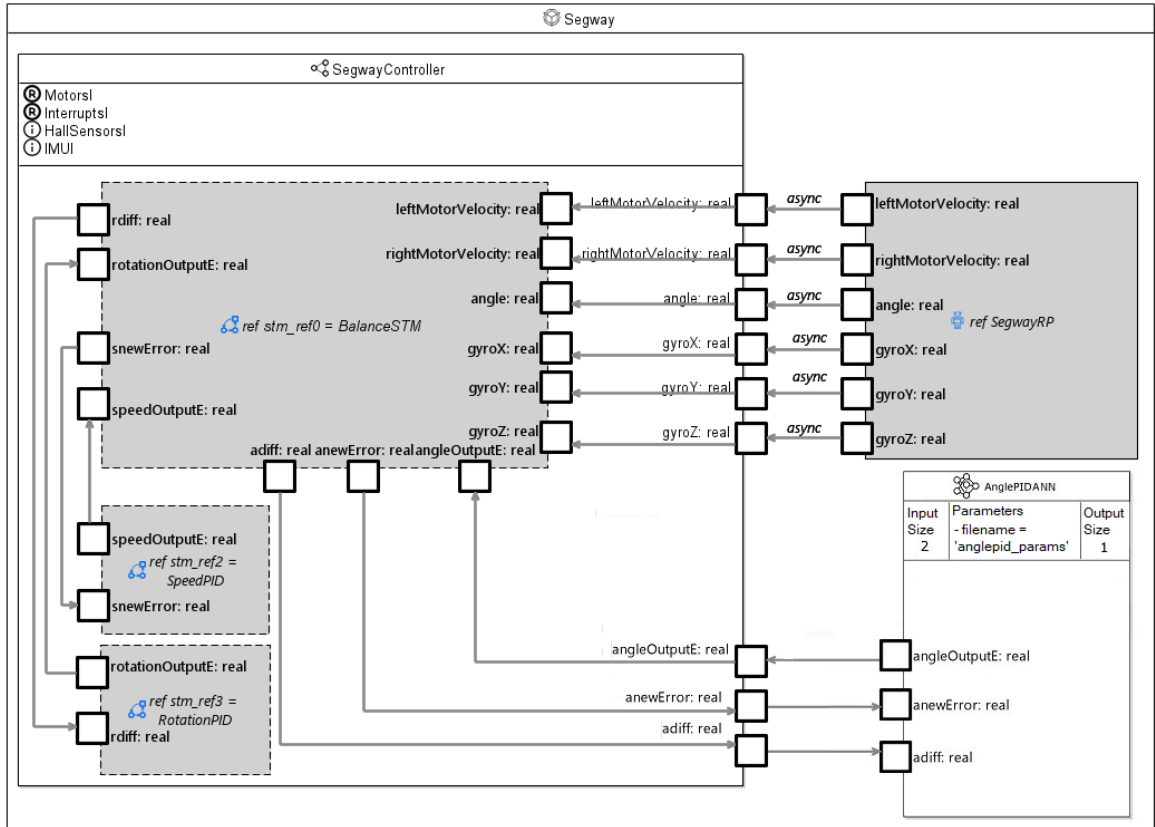


Figure 3.11: A parallel version of the Segway model with AnglePIDANN modelled as an ANNController. Legend: \odot : Module, \odot : controller definition, \rightarrow : connection, π : constant, \oplus : robotic platform reference, \oplus : state machine reference, \oplus : ANN component.

controller, rather than to the state machine AnglePID. All state machines are unchanged in this approach.

Similarly to an ANNOperation, an ANNController requires a Parameters box, filled in the same way as demonstrated in Figures 3.7 and 3.8. An ANNController also requires two more boxes, both containing integers: one containing the input size, labelled Input Size; and one containing the output size, labelled Output Size. We note that these parameters can be inferred from the parameters and required interface of an ANNOperation, but need to be given explicitly in an ANNController. The ANNController component does not contain a Required Interface partition, but this component can contain events on its borders, similarly to a controller. We note that we give the parameters of AnglePIDANN via a file in Figure 3.11, but we could also have defined them explicitly, as in Figure 3.8.

ANNs are trained to operate on vectors of real numbers. These can be represented either by a single sequence of real numbers, or by multiple separate real numbers. The size of an ANN’s input and output vectors can be, and often are, different. In RoboChart, a controller’s input or output can be represented by either a single event, communicating a sequence representing the whole vector, or multiple events: one for each element of the vector. For example, in our model, we define multiple events to represent the input and output vectors, as our ANN is low-dimensional. We declare two input events, as the Input Size of AnglePIDANN is 2, and

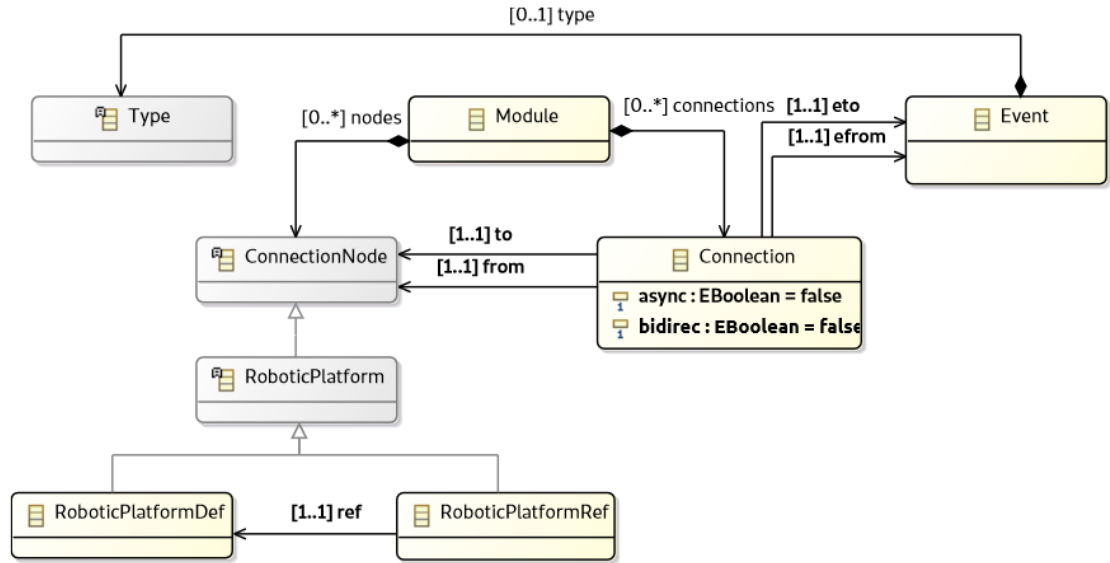


Figure 3.12: Metamodel of a RoboChart module [65]. Key: grey background box - abstract class; yellow background box - class; bold elements - required elements; non-bold elements - optional elements.

one output event, as the Output Size of AnglePIDANN is 1. The input events are `anewError` and `adiff`, and the output event is `angleOutputE`.

In the next section we extensions to the RoboChart metamodel to accommodate these components.

3.2.2 ANN Components Metamodel

We define two types of ANN component: a controller and an operation. We begin by describing how, in the existing metamodel, controllers and operations are represented: see Figure 3.12. Following this, we discuss how ANN components are integrated into the metamodel.

A module describes a robotic system; it is composed of multiple `ConnectionNode` objects and multiple `Connection` objects that define the relationships between `ConnectionNodes` [65]. A `ConnectionNode` represents elements that can be connected, namely: robotic platforms, controllers, and state machines. A `RoboticPlatform` component can be included by either a definition, `RoboticPlatformDef`, or a reference to a definition, `RoboticPlatformRef`. Every `Connection` establishes a relationship between two `ConnectionNode` objects. A `Connection` can be either synchronous or asynchronous, and unidirectional or bidirectional, and it defines two events of the nodes that they connect. Via these two events a connection can define control and data flow between these nodes.

Figure 3.13 describes how an operation is defined and used in other RoboChart components. An operation can be defined through an `OperationDef`, which defines the operation’s behaviour as a state machine. Alternatively, an operation can be specified via an `OperationRef` reference to an `OperationDef`.

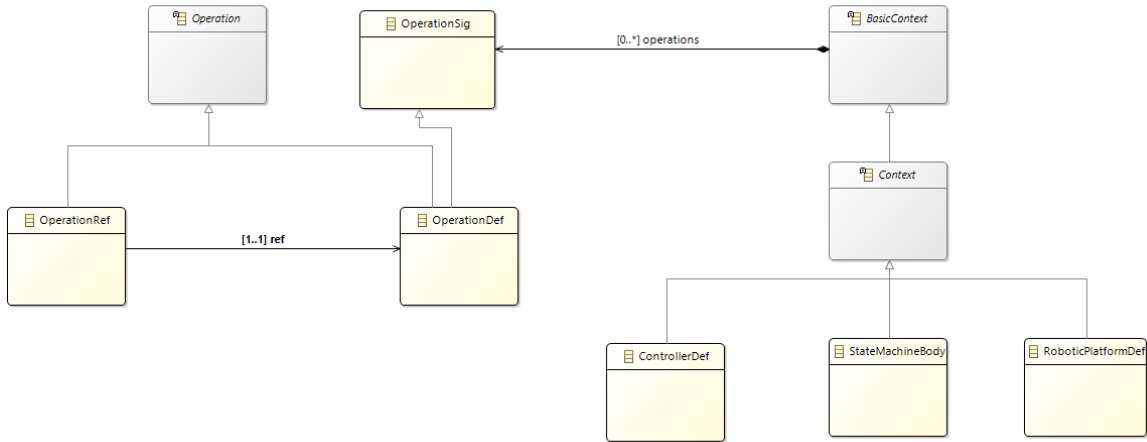


Figure 3.13: Metamodel of RoboChart operations. Key: grey background box: abstract class, yellow background box: class, bold elements: required elements, non-bold elements: optional elements.

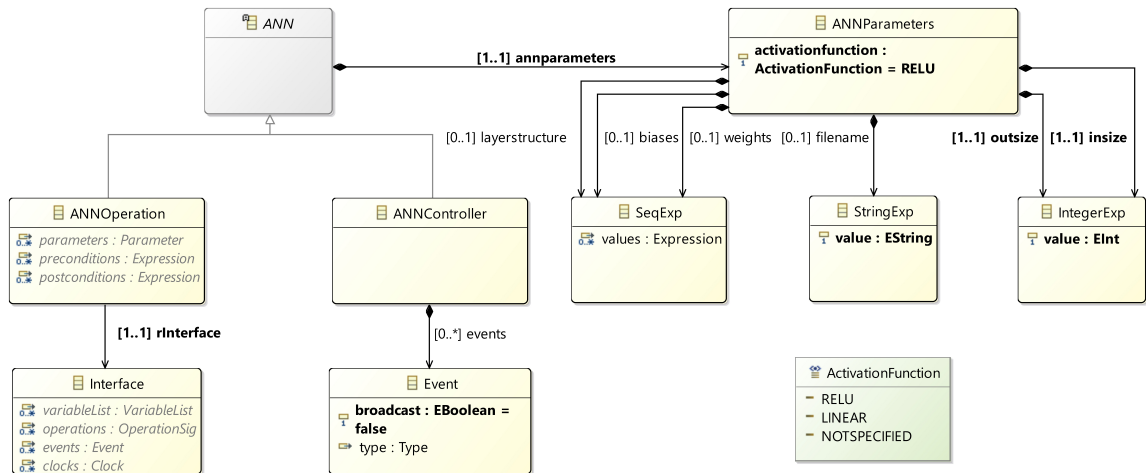


Figure 3.14: A diagram of our extensions to the RoboChart metamodel. Key: grey background box: abstract class, yellow background box: class, bold elements: required elements, non-bold elements: optional elements.

The **OperationSig** class represents details about an operation: its parameters, and its pre and postconditions. **OperationSig** is used in the definition of contexts that require use of the operation. The **BasicContext** abstract class defines the variables, constants, operations, and events of an element; the **Context** class is a **BasicContext** that also has interfaces. A **BasicContext** has a property **operations**, which contains 0 or more **OperationSig** objects. The subclasses for **Context** include **RoboticPlatformDef**, **StateMachineBody**, and **ControllerDef**.

We present a diagram of our extensions to the RoboChart metamodel in Figure 3.14. We present the full metamodel in Appendix A. Principally, we define two new classes: **ANNOperation** and **ANNController** to represent operations and controllers defined by an ANN.

The abstract class **ANN** contains a single reference to an **ANNParameters** instance. **ANNOper-**

ation and `ANNController` are subclasses of `ANN`. `ANNOperation` contains a reference `rInterface` to an `Interface`, allowing it to communicate its output through the variables in this interface. `ANNController` contains a single reference to a list of events, these are the events that the component can engage in.

The class `ANNParameters` defines the parameters of an ANN. This class contains six references, two of which are of type `IntegerExp`: a class which represents an expression of type integer. These are `insize`, the input size of the ANN, and `outsize`, the output size of the ANN. `ANNParameters` contains three sequence properties, represented in RoboChart through references to the `SeqExp` class. The property `layerstructure` defines the size of each layer, and `weights` and `biases` define the weights and biases of the trained ANN, respectively. The `weights`, `biases`, and `layerstructure` values are optional, as they may be defined implicitly through `filename`.

We note that `weights` and `biases` are both sequences of real numbers. RoboChart's type system is based on \mathbb{Z} [101]. Hence, we can use the \mathbb{Z} type system to represent real numbers, specifically, by using the approach in [79] to define the relevant subset of \mathbb{A} , the \mathbb{Z} type for numbers.

Although it is possible for different layers to use different functions, here we assume that all layers use a single function. In addition, we consider just `RELU` and `LINEAR` functions, as indicated in the definition of the enumeration type `ActivationFunction`. These are convenient for the verification technology we discuss in the next chapter. Extension of this metamodel to consider the possibility to use additional functions, and different functions in different layers is simple. It requires just extension of the `ActivationFunction`, and the definition of `activationfunction` as a sequence.

Our ANN components are not defined using state machines, so the existing structures that define controllers and operations in RoboChart, namely `ControllerDef` and `OperationDef` (presented in [65]) are not applicable to our components. To address this, we introduce two abstract classes, `GeneralController` and `GeneralOperation`, that allow controllers and operations to be defined using multiple methods. We present these new classes, and the discussed changes in Figure 3.15.

The abstract classes `Controller`, and `Operation` are unchanged in our work, but we add to the way we allow these components to be defined. The classes that we use to represent references to these components, `ControllerRef` and `OperationRef`, are also unchanged.

`ControllerRef` now contains a reference to an instance of a `GeneralController`, not to a `ControllerDef` instance. Both `ControllerDef` and `ANNController` are concrete subtypes of `GeneralController`, so both can be used to define the behaviour of a controller.

Similarly, `OperationRef` now refers to a `GeneralOperation` instance, where `ANNOperation` and `OperationDef` have roles similar to those of the corresponding controller definition classes. In this case, though, `OperationSig` is a super-class of both `ANNOperation` and `OperationDef`, allowing both to contain a signature: enabling the definition of parameters, termination, and pre and postconditions. These modifications allow other RoboChart components to refer to an `ANNOperation` whenever they could refer to an `OperationDef`.

We also rephrase some of the well-formedness conditions of RoboChart to accommodate the above changes. These changes are minor and concerning with differentiating components defined with state machines and those defined using neural networks.

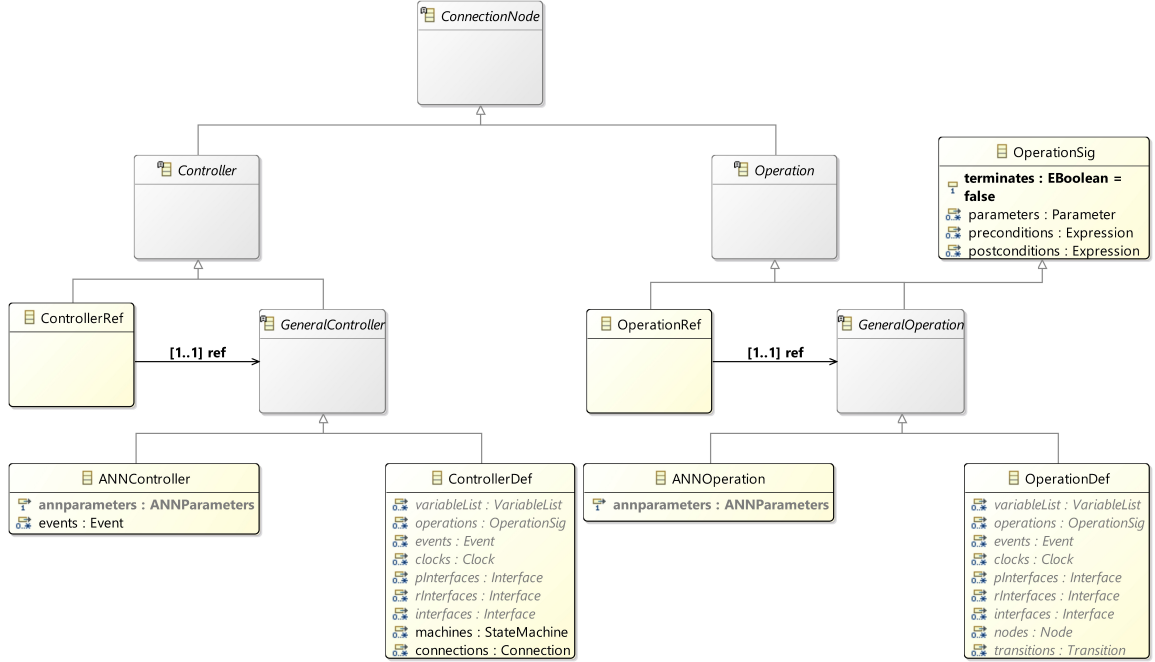


Figure 3.15: A diagram of our modifications to the RoboChart metamodel. Key: grey background box: abstract class, yellow background box: class, bold elements: required elements, non-bold elements: optional elements.

In the next section, we discuss well-formedness conditions of the new classes, required to enable meaningful semantics models to be generated.

3.2.3 ANN Component Well-Formedness Conditions

The metamodel in Fig. 3.14 enables the definition of models that are not meaningful. In Tab. 3.1, we present well-formedness conditions that need to be satisfied by a RoboChart model that includes ANN components. These are in addition to the existing well-formedness conditions of RoboChart [64].

WF1 reflects properties of trained ANNs. This condition restricts the values that the parameters of the ANNParameters class can take. We formalise **WF1** in Z below.

WF1 : \mathbb{P} ANNParameters

$$\forall \text{annparams} : \text{ANNParameters} \bullet$$

$$\text{annparams.insize} > 0 \wedge$$

$$\text{annparams.outsize} > 0 \wedge$$

$$(\text{annparams.weights} \neq \text{null_seq} \wedge \text{annparams.biases} \neq \text{null_seq} \wedge$$

$$\text{annparams.layerstructure} \neq \text{null_seq} \Rightarrow$$

$$(\text{tensor} \sim) \text{annparams.weights} \neq \emptyset \wedge$$

$$(\text{matrix} \sim) \text{annparams.biases} \neq \emptyset \wedge$$

$$(\text{vector} \sim) \text{annparams.layerstructure} \neq \emptyset \wedge$$

$$\#(\text{tensor} \sim) \text{annparams.weights} = \#(\text{vector} \sim) \text{annparams.layerstructure} \wedge$$

$$\#(\text{matrix} \sim) \text{annparams.biases} = \#(\text{vector} \sim) \text{annparams.layerstructure}$$

WF1	<i>insize</i> and <i>outside</i> are greater than 0, and <i>layerstructure</i> , <i>weights</i> , and <i>biases</i> are non-empty and of the same size, if not null.
WF2	If <i>filename</i> is null, then <i>weights</i> , <i>biases</i> , and <i>layerstructure</i> are not.
WF3	Either <i>layerstructure</i> , <i>weights</i> , and <i>biases</i> are all null, in which case <i>filename</i> is not, or they are all different from null.
WF4	<i>activationfunction</i> is NOTSPECIFIED if, and only if, <i>filename</i> is not null.
WF5	For every <i>i</i> , the size of <i>weights i</i> and <i>biases i</i> is <i>layerstructure i</i> .
WF6	For every <i>i</i> , and for all <i>j</i> , the size of <i>weights i j</i> is <i>layerstructure (i - 1)</i> when <i>i</i> is greater than 1, or <i>insize</i> otherwise.
WF7	An ANNController must have exactly the sum of <i>insize</i> and <i>outside</i> events, or exactly two events.
WF8	The connections to and from an ANNController match the nature of the events (inputs and outputs) in their directions and types.
WF9	An ANNOperation must have exactly <i>insize</i> parameters or just one, and must require exactly one variable, or <i>outside</i> variables.

Table 3.1: RoboChart ANN well-formedness conditions

Here, we define a set *WF1* containing those ANNParameter objects that satisfy the **WF1** well-formedness condition. In our specification, we define objects as a free type with the following constructor functions: *tensor*, whose domain is the triple nested sequences; *matrix*, defined on the double nested sequences; and *vector*, defined on the sequence type. We extract the values contained in the free types using the inverse of these functions: *tensor*[~], *matrix*[~], and *vector*[~]. We denote that a SeqExp object (*weights*, *biases*, *layerstructure*) is null using the constant *null_seq*, and we denote that the sequence contained by a SeqExp object is non-empty by stating that it is not equal to the empty set (\emptyset). More information on free types and the Z specification language is available in [91, 109].

WF2 ensures that if we are not defining the ANN's parameters via a file, if the *filename* reference is null, then the parameters must be defined through the *weights*, *biases*, and *layerstructure* references; so, they must not be null. We provide a formalisation of **WF2** below.

$$\begin{aligned}
 &WF2 : \mathbb{P} \text{ANNParameters} \\
 &\forall \text{annparams} : \text{ANNParameters} \bullet \\
 &\quad \text{annparams.filename} = \text{null_string} \Rightarrow \\
 &\quad \quad \text{annparams.weights} \neq \text{null_seq} \wedge \\
 &\quad \quad \text{annparams.biases} \neq \text{null_seq} \wedge \\
 &\quad \quad \text{annparams.layerstructure} \neq \text{null_seq}
 \end{aligned}$$

Here, *filename* is of type StringExp, so we denote that it is null using the constant *null_string*.

WF3 guarantees that the references *layerstructure*, *weights*, and *biases* can only be null if *filename* is non-null, in every other case they must contain values. We formalise this condition below.

WF3 : \mathbb{P} ANNParameters

$$\begin{aligned} &\forall \text{annparams} : \text{ANNParameters} \bullet \\ &\quad \text{annparams.weights} = \text{null_seq} \wedge \\ &\quad \text{annparams.biases} = \text{null_seq} \wedge \\ &\quad \text{annparams.layerstructure} = \text{null_seq} \wedge \\ &\quad \text{annparams.filename} \neq \text{null_string} \\ &\vee \\ &\quad \text{annparams.weights} \neq \text{null_seq} \wedge \\ &\quad \text{annparams.biases} \neq \text{null_seq} \wedge \\ &\quad \text{annparams.layerstructure} \neq \text{null_seq} \end{aligned}$$

If the ANN's parameters are declared via a file, then the activation function should not be declared in the attribute `activationfunction`, it should take the value `NOTSPECIFIED`. We capture this through **WF4**, and we formalise this condition below.

WF4 : \mathbb{P} ANNParameters

$$\begin{aligned} &\forall \text{annparams} : \text{ANNParameters} \bullet \\ &\quad (\text{annparams.activationfunction} = \text{NOTSPECIFIED}) \Leftrightarrow \\ &\quad \text{annparams.filename} \neq \text{null_string} \end{aligned}$$

WF5 and **WF6** concern the specific structure of the `weights` and `biases` sequences, as they must be constructed with strict size considerations. First, the size of the both sequences must be equal to the size of `layerstructure` (the number of layers). Next, every element indexed by i in `weights` and `biases`, which is also a sequence, must be of size equal to `layerstructure(i)`. This is because there is a separate weight vector and bias scalar for each node in each layer. Finally, the size of each sequence `weights(i)(j)` must be equal to the size of the previous layer, or the input size in the case of the first layer. We present the formalisations of both conditions below.

WF5 : \mathbb{P} ANNParameters

$$\begin{aligned} &\forall \text{annparams} : \text{ANNParameters} \bullet \\ &\quad (\text{annparams.filename} = \text{null_string}) \Rightarrow \\ &\quad \forall i : \text{dom}((\text{vector}^{\sim}) \text{annparams.layerstructure}) \bullet \\ &\quad \quad \#(\text{tensor}^{\sim}) \text{annparams.weights } i = \\ &\quad \quad \quad (\text{vector}^{\sim}) \text{annparams.layerstructure } i \\ &\quad \wedge \\ &\quad \quad \#(\text{matrix}^{\sim}) \text{annparams.biases } i = \\ &\quad \quad \quad (\text{vector}^{\sim}) \text{annparams.layerstructure } i \end{aligned}$$

WF6 : \mathbb{P} ANNParameters

$$\begin{aligned} & \forall \text{annparams} : \text{ANNParameters} \bullet \\ & \quad (\text{annparams.filename} = \text{null_string}) \Rightarrow \\ & \quad \forall i : \text{dom}((\text{vector} \sim) \text{annparams.layerstructure}) \bullet \\ & \quad \quad \forall j : \text{dom}((\text{tensor} \sim) \text{annparams.weights } i) \bullet \\ & \quad \quad \quad (i > 1) \Rightarrow \#(\text{tensor} \sim) \text{annparams.weights } i j = \\ & \quad \quad \quad (\text{vector} \sim) \text{annparams.layerstructure } i \\ & \quad \quad \quad \wedge \\ & \quad \quad \quad (i = 1) \Rightarrow \#(\text{tensor} \sim) \text{annparams.weights } i j = \\ & \quad \quad \quad \text{annparams.insize} \end{aligned}$$

Both **WF5** and **WF6** are only active when we are not defining the parameters through a file, so we add the antecedent that the **filename** must be **null** to these conditions.

WF7 ensures conformance between the ANN's declared number of inputs and outputs and the number of events of the controller. We give its formalisation below.

WF7 : \mathbb{P} ANNController

$$\begin{aligned} & \forall \text{anncontroller} : \text{ANNController} \bullet \\ & \quad \# \text{anncontroller.events} = 2 \vee \\ & \quad \# \text{anncontroller.events} = \\ & \quad \quad \text{anncontroller.annparameters.insize} + \\ & \quad \quad \text{anncontroller.annparameters.outsize} \end{aligned}$$

WF8 states that if a module contains an ANN component, then events intended to represent inputs are used in incoming connections, and those intended to represent outputs are used in outgoing connections. Its formalisation is below.

WF8 : \mathbb{P} RCModule

$$\begin{aligned} & \forall \text{module} : \text{RCModule} \bullet \\ & \quad \forall \text{anncontroller} : \text{ANNController} \mid \text{anncontroller} \in \text{module.nodes} \bullet \\ & \quad \quad \# \{ \text{conns} : \text{module.connections} \mid \text{conns.to} = \text{anncontroller} \} = \\ & \quad \quad \quad \text{anncontroller.annparameters.insize} \wedge \\ & \quad \quad \# \{ \text{conns} : \text{module.connections} \mid \text{conns.from} = \text{anncontroller} \} = \\ & \quad \quad \quad \text{anncontroller.annparameters.outsize} \wedge \\ & \quad \quad \forall \text{from_conn}, \text{to_conn} : \text{module.connections} \mid \\ & \quad \quad \quad \text{from_conn.from} = \text{anncontroller} \wedge \text{to_conn.to} = \text{anncontroller} \bullet \\ & \quad \quad \quad \exists \text{ann_input}, \text{ann_output} : \text{anncontroller.events} \bullet \\ & \quad \quad \quad \text{from_conn.e.from.type} = \text{ann_output.type} \wedge \\ & \quad \quad \quad \text{to_conn.e.to.type} = \text{ann_input.type} \end{aligned}$$

Lastly, we provide a single well-formedness condition concerning the **ANNOperation** class: an **ANNOperation** must have exactly *insize* parameters or just one. Similarly, an **ANNOperation** must require a single variable, or *outsize* variables, representing scalar or vector output. As with the other conditions, we present a formalisation below.

WF9 : \mathbb{P} ANNOperation

$$\begin{aligned} &\forall \textit{annoperation} : \textit{ANNOperation} \bullet \\ &\quad (\# \textit{annoperation.parameters} = 1 \vee \\ &\quad \# \textit{annoperation.parameters} = \textit{annoperation.annparameters.insize}) \\ &\quad \wedge \\ &\quad (\# \bigcup \{ \textit{varList} : \textit{VariableList} \mid \\ &\quad \quad \textit{varList} \in \textit{annoperation.rInterface.variableList} \wedge \\ &\quad \quad \textit{varList.modifier} = \textit{VAR} \bullet \textit{varList.vars} \} = 1 \\ &\quad \vee \\ &\quad \# \bigcup \{ \textit{varList} : \textit{VariableList} \mid \\ &\quad \quad \textit{varList} \in \textit{annoperation.rInterface.variableList} \wedge \\ &\quad \quad \textit{varList.modifier} = \textit{VAR} \bullet \textit{varList.vars} \} = \\ &\quad \quad \textit{annoperation.annparameters.outsize}) \end{aligned}$$

Here, we capture the required variables of by defining all variables contained in the interface *rInterface*, using a set comprehension term. We consider the variable *varList* as those variable lists in *rInterface*, as such this variable must fulfil two conditions. The first is that *varList* is contained in the *variableList* of the interface *rInterface*: denoted by $\textit{varList} \in \textit{annoperation.rInterface.variableList}$. The second is that the modifier of *varList* is *VAR*, not *CONST*, denoting a constant. Then, the set is formed of the expression *varList.vars*: the set of variables contained in a single variable list. Finally, as we now have a set of sets of the required variables, we take the distributed union of this set to define every variable contained the interface *rInterface*.

The metamodel and well-formedness conditions presented in this section define how to specify ANN components in RoboChart. In the next section, we define semantics for these components to enable integration, validation, and verification.

3.3 Semantics Overview

In this section, we present an overview of the existing RoboChart semantics as the basis of our integrated ANN component semantics. We describe CSP in Section 3.3.1 and the RoboChart semantics in Section 3.3.2.

3.3.1 CSP Overview

CSP stands for Communicating Sequential Processes; it was first introduced by Tony Hoare in 1978 [38], and has been described in numerous works since [39, 83, 87]. A CSP model describes a set of abstract processes that continuously interact with their environment via events. Communication channels enable processes to interact with each other and their environment. Processes can transfer information through writing and reading via channels, a communication via a channel is an event. CSP incorporates function definitions, set theory, and propositional calculus.

A process can be seen as an abstraction of a thread of behaviour in a system. An event in CSP is an abstraction of a process's instantaneous interaction with its environment. Events in CSP may require multiple processes to participate simultaneously. The alphabet of all

processes, that is, all events in which a process can engage, is often referred to as Σ [83]. The complete set of events that the channel can communicate is referred to as the communications of a channel.

Table 3.2: CSP Operators

Symbol	Name
$Skip$	Skip
$P \parallel [a] Q$	Parallel Composition
$P \parallel\parallel Q$	Interleaving
$\{ c \}$	Channel Set
$c \rightarrow P$	Prefix
$c?x \rightarrow P$	Input
$c!e \rightarrow P$	Output
$P; Q$	Sequential Composition
$P \Theta_{cs} Q$	Exception
$P \setminus cs$	Hiding
$P[[c \leftarrow d]]$	Renaming

Table 3.2 presents the CSP operators used in our work. We describe these operators as we use them in the following sections. Next, we give an overview of the RoboChart semantics encoded in CSP.

3.3.2 RoboChart Semantics Overview

In this section, we give a brief overview of the RoboChart semantics. It defines a CSP process for each module, controller, and state machine component and their memories, holding values for their local variables. (The local variables of a module are those of its robotic platform.) These processes are composed together using either parallel or sequential composition. We display the structure of this composition in Figure 3.16.

Each box in the diagram represents a CSP process. Memory processes are represented by double bordered boxes; these processes handle updating variables and propagating them to each component that requires them. The Robotic Platform Memory propagates changes to the Controller Memory, which propagates changes to the State Machine Memory.

Each module process is composed of a Robotic Platform memory process in parallel with multiple Controller processes, as shown in Figure 3.16. Likewise, each Controller is composed of a Controller Memory process interacting in parallel with multiple State Machine processes.

Events are used to represent changes in the values of the variables of the platform, occurrences of platform events, and calls to platform operations. These are the visible events in a module. Events representing internal interactions of controllers, which are those that do not involve the robotic platform, are hidden. The definition of hiding in CSP is as shown below.

$$P \setminus cs$$

Here, the operator \setminus describes that events in the set cs are hidden from the process P . A process may not engage with a hidden event; its sub-processes, however, can still engage with this event.

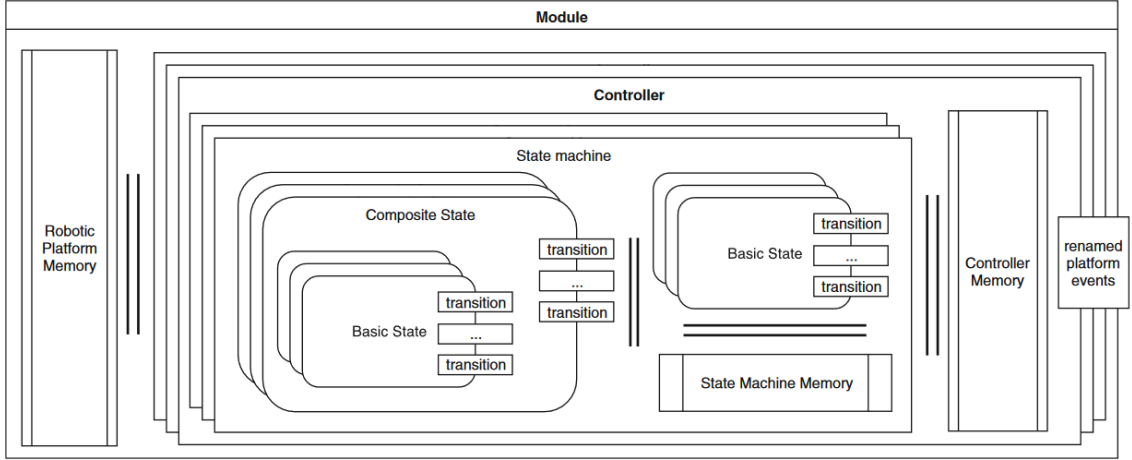


Figure 3.16: The structure of the RoboChart semantics; Figure 11 in [101]. Parallel composition is indicated by stacked components and parallel lines. Memory processes are indicated by double bordered boxes, and bordered boxes represent interaction points between components.

To illustrate the semantics, we describe the CSP semantics of the segway model, in particular the semantics of the module `Segway` in terms of the controller `SegwayController` as presented in Section 3.2. We use the naming scheme from [101], where we prefix each channel name with its component identifier, such that for a channel c of component C , the name will be C_c .

A module process is defined by the parallel composition of controller processes, one for every controller in the module, as displayed in Figure 3.16. In our example, there is a single controller in the module, so the parallel composition of each controller is just the process representing the single controller, which we will refer to as *SegwayController*. This leads to an initial definition of the process representing the entire module, which we will refer to as *Segway*, as the following process:

$$Segway = ((SegwayController[\dots])\Theta_{\{end\}}Skip) \setminus \{end\}$$

A module process terminates when all of its controller processes engage with the event end , which is a special RoboChart event that represents termination of a component. We use CSP's exception operator to denote this termination, the definition of which is as shown below.

$$P\Theta_{cs}Q$$

Here, the exception operator Θ represents the process that behaves like P until an event in the set cs occurs, in which case, it behaves like Q . In our example, the process *Segway* behaves like *SegwayController* until the event end occurs, when it behaves like *Skip*, which is a special process in CSP representing successful termination. That is to say, our module process *Segway* successfully terminates if it engages in the event end . In addition, the event end is also hidden in *Segway*, as it is an event a module process itself cannot engage in, only its controller processes.

We recall that the SegwayRP robotic platform, presented in Section 3.2.1, defines six events: `angle`, `gyroX`, `gyroY`, `gyroZ`, `leftMotorVelocity`, and `rightMotorVelocity`. In the semantics, a separate channel is defined for every event in the RoboChart model, each communicating values of the type *InOut.DataType*. Here, the *InOut* type corresponds to the two values *in* or *out*, used to denote input or output across the channel. The *DataType* type corresponds to the type of data communicated through the channel (if any), which is omitted here. Each channel is named according to its RoboChart event prefixed by *SRP*. The SegwayRP platform also defines four operations: `setRightMotorSpeed`, `setLeftMotorSpeed`, `enableInterruptsCall`, and `disableInterruptsCall`. A channel is defined for every operation, where the name of that channel corresponds to the name of the operation suffixed with *Call*. Additionally, we note that channels representing events that are involved in connections between controllers are hidden in a module process. There are no such connections, however, in the module `Segway`, so no event hiding is required in the process *Segway*.

This leads to the definition of the visible events of the *Segway* process, defined from the `Segway` module, as those events associated with the channels listed in the set below:

$$\{SRP_angle, SRP_gyroX, SRP_gyroY, SRP_gyroZ, \\ SRP_leftMotorVelocity, SRP_rightMotorVelocity, \\ setLeftMotorSpeedCall, setRightMotorSpeedCall, \\ enableInterruptsCall, disableInterruptsCall\}$$

The process *SegwayController* representing the controller `SegwayController`, in the context of the *Segway* process representing the module `Segway`, must have those channels that represent RoboChart events renamed to capture connections between components of the module. In our parallel Segway model, all events of the robotic platform `SegwayRP` connect to identically named events of the controller `SegwayController`. In the semantics, we represent connections involving robotic platforms by renaming all channels involved according to the qualified name of the robotic platform. For example, the event `angle` of `SegwayController` is involved in a connection with `SegwayRP`, so the channel *SC_angle* (*SC* as an abbreviation of `SegwayController`) is renamed to *SRP_angle* (*SRP* as an abbreviation of `SegwayRP`). The definition of renaming in CSP is as shown below.

$$P[[c \leftarrow d]]$$

This represents process *P*, where the event *c* is now referred to by the name *d*. Notably, if there were multiple events renamed to *d*, a single call to *d* synchronises on all events that were renamed to *d*, as they are now referred to by the same name.

We present the renaming of the process *SegwayController*, in the context of the process representing a module, *Segway*. Specifically, we only rename channels representing RoboChart events, not channels that represent operation calls.

$$SegwayController[[\\ SC_angle \leftarrow SRP_angle \\ SC_gyroX \leftarrow SRP_gyroX \\ SC_gyroY \leftarrow SRP_gyroY \\ SC_gyroZ \leftarrow SRP_gyroZ \\ SC_leftMotorVelocity \leftarrow SRP_leftMotorVelocity \\ SC_rightMotorVelocity \leftarrow SRP_rightMotorVelocity]]$$

The process *SegwayController* represents the semantics of the RoboChart controller *SegwayController* (considering the parallel *SegwayController*, as presented in Figure 3.4). This is defined as the parallel composition of processes for its state machines, as shown in Figure 3.16. *SegwayController* is composed of three state machines: *SpeedPID*, *RotationPID*, and *BalanceSTM*. Following this, we define *SegwayController* as the parallel composition of three processes, where each represents a single state machine component of identical name: *SpeedPID*, *RotationPID*, and *BalanceSTM*. Finally, *SegwayController* behaves like *Skip* if the event *end* occurs, defined using the exception operator.

$$\begin{aligned} \textit{SegwayController} = & (\\ & \textit{SpeedPID}[\dots][[\textit{SpeedPID_evts}]] (\\ & \quad \textit{RotationPID}[\dots][[\textit{RotationPID_evts}]] (\\ & \quad \quad \textit{BalanceSTM}[\dots])) \Theta_{\textit{end}} \textit{Skip} \\ & \setminus \textit{SC_hidden_evts} \end{aligned}$$

To define parallel composition, we use the following CSP operator:

$$P \parallel [a] Q$$

Here, the operator \parallel is used to denote that a process P is composed in parallel with the process Q , along the synchronisation set a . This means that each process must cooperate on events in the set a , such that neither can perform these events independently. In our example, all four state machines processes are in parallel with each other, synchronising on three separate sets, each containing the events used for communication between the state machines.

The visible events of the process *SegwayController* should correspond to communications along channels representing RoboChart events, communications along channels representing calls to RoboChart operations, and the event *end* (which is visible in processes for controllers). The RoboChart controller *SegwayController* defines six events: *angle*, *gyroX*, *gyroY*, *gyroZ*, *leftMotorVelocity*, and *rightMotorVelocity*; each is represented by a channel prefixed with *SC*. *SegwayController* also requires four operations: *setLeftMotorSpeed*, *setRightMotorSpeed*, *enableInterrupts*, and *disableInterrupts*. In the semantics, a separate channel is defined for each, suffixed by *Call*. This leads to the complete visible events of *SegwayController* as communications along the following channels:

$$\begin{aligned} & \{ \textit{SC_angle}, \textit{SC_gyroX}, \textit{SC_gyroY}, \textit{SC_gyroZ}, \\ & \quad \textit{SC_leftMotorVelocity}, \textit{SC_rightMotorVelocity}, \\ & \quad \textit{setLeftMotorSpeedCall}, \textit{setRightMotorSpeedCall}, \\ & \quad \textit{enableInterruptsCall}, \textit{disableInterruptsCall}, \\ & \quad \textit{end} \} \end{aligned}$$

Each process for a state machine in the context of the controller process *SegwayController* (*SpeedPID*, *RotationPID*, and *BalanceSTM*) is renamed to capture the connections involved in the RoboChart controller *SegwayController*. In a controller process, each channel (representing a RoboChart event) involved in a connection to the controller itself is renamed according to the qualified name of the controller. This is similar to renaming a controller's channels (which connect to a robotic platform) to the qualified name of the robotic platform in a module process. For example, RoboChart event *angle* of *BalanceSTM* is involved in a

connection with the controller `SegwayController`, so the channel `BSTM_angle` (`BSTM` as an abbreviation of `BalanceSTM`) is renamed to `SC_angle`. `SegwayController`, however, contains connections between state machines, referred to as internal connections of the controller, which are defined in a different way.

An internal controller connection (which is both unidirectional and synchronous) defines a relationship between two events of separate state machines: one outgoing event communicating with one incoming event. In the semantics, a channel is defined for each event, and a process for each state machine. The outgoing process (for the outgoing state machine) communicates via output on its channel, which is received by the incoming process (for the incoming state machine) via input on its channel. In the semantics, this connection is realised by renaming all *in* events (from the *InOut* type) of the incoming channel to *out* events of the outgoing channel. For example, the event `snewError` of `BalanceSTM` connects to `snewError` of `SpeedPID`, in the semantics the outgoing channel is `BSTM_snewError.out` (`BSTM` is an abbreviated version of `BalanceSTM`) and the incoming channel is `SPID_snewError.in` (`SPID` is an abbreviated version of `SpeedPID`). Therefore, `SPID_snewError.in` is renamed to `BSTM_snewError.out`, which means that any communication on `BSTM_snewError.out` is synchronised with `SPID_snewError.in`, connecting the two channels. All *out* events of the incoming channel are also renamed to the *in* events of the outgoing channel. A connection may also be bidirectional or asynchronous; all connections in our example, however, are unidirectional and synchronous.

We present the renaming of `SpeedPID` below, which captures its connections to `BalanceSTM`. To clarify, the channel `SPID_speedOutputE` is not renamed, as it represents the event `speedOutputE`, which is involved in an outgoing connection event from `SpeedPID` to `BalanceSTM`.

```
SpeedPID = SpeedPID_STM[[
  SPID_snewError.in ← BSTM_snewError.out
  SPID_snewError.out ← BSTM_snewError.in]]
```

Given this renaming, we can now define the synchronisation set of `SpeedPID`, which we refer to as `SpeedPID_evt`s. This set contains all channel communication events that `SpeedPID` and `BalanceSTM` synchronise on, as the state machine `SpeedPID` is connected to only `BalanceSTM` (if it were connected to more, it would contain all shared communication events). These processes communicate on two channels: `BSTM_snewError` and `SPID_speedOutputE`. These channels are the outgoing channels representing the three connections that they share in the RoboChart model. Following this, the synchronisation set needs to contain all channel communication events of these channels. This can be done using the channel set operator, which defines all events associated with a given channel. We present this operator below.

```
{| c |}
```

Here, the brackets `{| c |}` denote the channel set of `c`, this is the set of all possible communication events of `c`. That is to say, this is all possible events of the type of the channel `c`. This operator is used here to form the set of the complete events in these channels to synchronise on.

We present the synchronisation set $SpeedPID_evts$ below. This set enables connection between these two processes, because communications over these channels synchronise with events in both $SpeedPID$ and $BalanceSTM$ (due to renaming). In addition, the set contains the event end , as all state machine processes need to terminate simultaneously.

$$SpeedPID_evts = \{| \textit{BSTM_snewError}, \textit{SPID_angleOutputE} |\} \cup \{end\}$$

Each PID state machine process interacts in parallel based on a separate synchronisation set, following a similar process as described for $SpeedPID$. In a controller, all internal connections, that is, connections between state machines, are hidden. The set SC_hidden_evts , therefore, represents all events associated with these internal connections, and is hidden in the definition of $SegwayController$.

Given these three renamed state machines, the only events that remain visible are those in terms of the $SegwayController$ itself, which is why all events used in the parallel composition is hidden. This leads to the visible events of $SegwayController$ as defined earlier in this section.

Our ANNController components interact on the same level as a Controller. Our semantics defines a single process that represents an ANNController. This process is composed in parallel with the processes for the other controllers (whether defined as additional ANNControllers or not) to form the definition of a module process. In the next section, we define the CSP semantics of these components.

3.4 CSP Semantics for Neural Networks

Our semantics defines constants to capture the metamodel. They are $insize : \mathbb{N}$, $outside : \mathbb{N}$, and $layerstructure : seq\mathbb{N}$. In addition, $layerNo : \mathbb{N}$ and $maxSize : \mathbb{N}$ record properties of $layerstructure$: its length, and its largest element. Finally, we have $weights : seq(seq(seq(Value)))$ and $biases : seq(seq(Value))$.

$Value$ is a type that represents the data communicated by our ANN. This is defined based on the types used in the ANN component in RoboChart. Some examples of the types that can be used are floating-point, integer, or binary values. If there are various ANN components, there is a definition for a type $Value$ for each of them. Equally, constants such as $layerstructure$, $maxSize$, and the others mentioned here are defined for each component.

In our semantics, we use the $ReLU$ activation function as piecewise linearity is useful for the scalability of verification approaches. Our semantics, however, can be easily adapted to other activation functions.

We use two channels. The first $layerRes : \{0 .. layerNo\}.\{1 .. maxSize\}.Value$ is used to communicate with other processes in the RoboChart semantics and for inter-layer communications. An event $layerRes.l.n.v$ represents the communication of a value v to or from the process for the n th in the process for the l th layer.

The second channel $nodeOut : \{1 .. layerNo\}.\{1 .. maxSize\}.\{1 .. maxSize\}.Value$ is for intra-node communication; $nodeOut.l.n.i.v$ refers to the layer, node and value for $layerRes$. The use of an index i is a technicality explained below.

In our semantics, presented in Fig. 3.18, we treat inputs to the ANN process as events on the channel $layerRes$, with 0 as the first argument's value. In this way, events $layerRes.0$

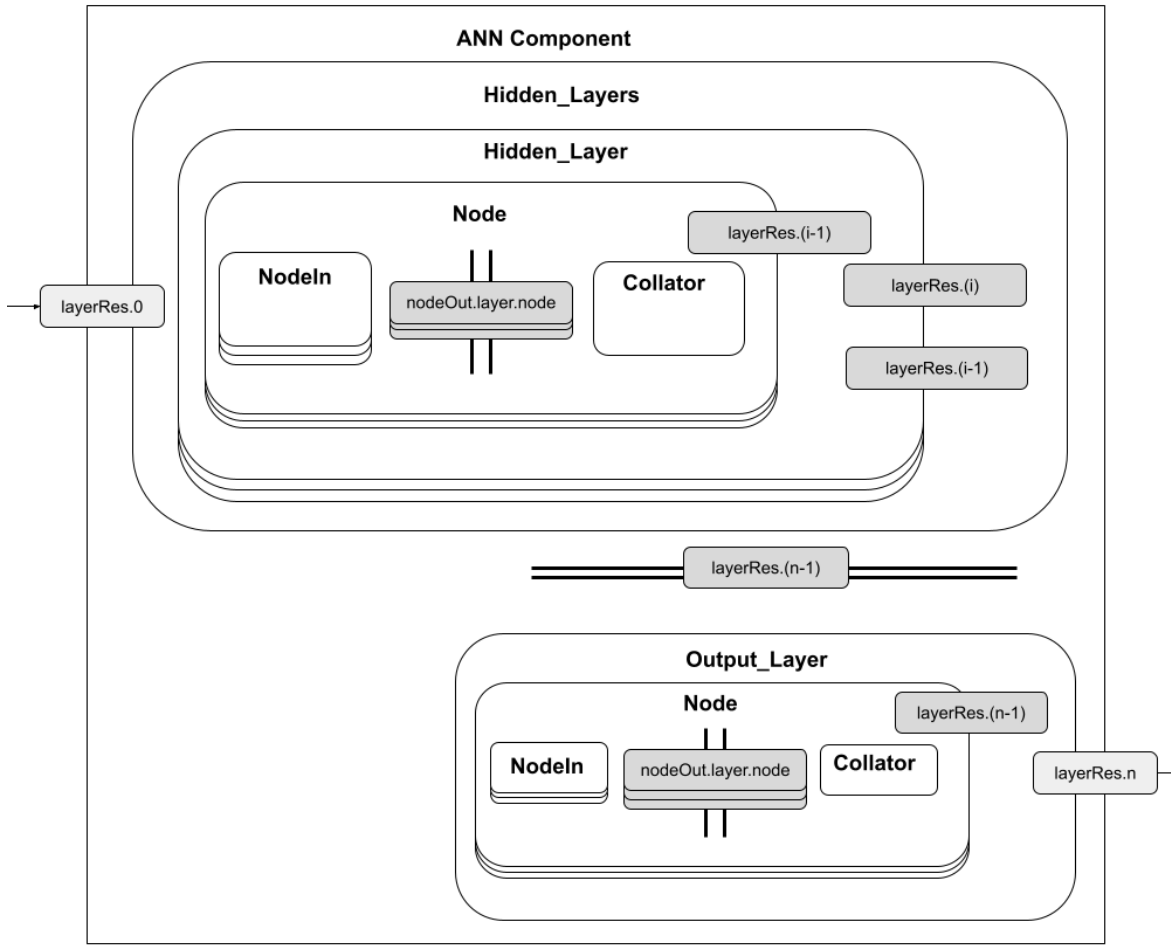


Figure 3.17: Processes defining the semantics of an ANN component. Light grey boxes: visible channels, Grey boxes: hidden channels, Parallel lines: parallel composition (synchronisation set intersects the lines), Stacked processes: parallel composition (synchronisation set is on the border), Stacked channels: multiple channels representation.

represent inputs to the ANN process from other components in the RoboChart model. All other communications on *layerRes* represent results from layer processes. Events *layerRes.2* represent the outputs of the ANN.

Fig. 3.18 presents the specification of the process *ANN*, defining our semantics for ANNControllers. We further illustrate our semantics through the diagram in Figure 3.17. It terminates (*Skip*) on the occurrence of a special event *end*, as defined using the exception operator Θ_{end} . This is an event raised by other controllers when all state machines terminate.

The operator $P \parallel [cs] \parallel Q$ describes the process whose behaviour is defined by those of *P* and *Q*, synchronising on all events in the set *cs*. Also, $P \setminus cs$ defines a process that behaves as *P*, but its events from the set *cs* are hidden. *ANN* composes in parallel the processes *HiddenLayers* and *OutputLayer*, then repeats via a recursive call. Since the *OutputLayer* communicates only with the last hidden layer, these processes synchronise on the events

$$\begin{aligned}
ANN &= \\
& \quad ((HiddenLayers \parallel [\{ layerRes.(layerNo - 1) \}] \parallel OutputLayer) \setminus ANNHiddenEvs \\
& \quad \ominus_{end} Skip); ANN \\
ANNHiddenEvs &= \Sigma \setminus \{ layerRes.0, layerRes.layerNo, end \} \\
HiddenLayers &= \parallel i : 1 .. layerNo - 1 \bullet [\{ layerRes.(i - 1), layerRes.i \}] \\
& \quad HiddenLayer(i, layerSize(i), layerSize(i - 1)) \\
HiddenLayer(l, s, inpSize) &= \parallel i : 0 .. s - 1 \bullet [\{ layerRes.(l - 1) \}] Node(l, i, inpSize) \\
Node(l, n, inpSize) &= \\
& \quad ((\parallel i : 0 .. inpSize - 1 \bullet NodeIn(l, n, i) \\
& \quad \parallel [\{ nodeOut.l.n \}] \\
& \quad Collator(l, n, inpSize)) \setminus \{ nodeOut \} \\
NodeIn(l, n, i) &= layerRes.(l - 1).n?x \rightarrow nodeOut.l.n.i!(x * weight) \rightarrow Skip \\
Collator(l, n, inpSize) &= \mathbf{let} \\
& \quad C(l, n, 0, sum) = layerRes.l.n!(ReLU(sum + bias)) \rightarrow Skip \\
& \quad C(l, n, i, sum) = nodeOut.l.n.i?x \rightarrow C(l, n, (i - 1), (sum + x)) \\
& \quad \mathbf{within} \\
& \quad \quad C(l, n, inpSize, 0) \\
OutputLayer &= \parallel i : 0 .. layer(layerNo) - 1 \bullet [\{ layerRes.(layerNo - 1) \}] \\
& \quad Node(layerNo, i, layerSize(layerNo - 1))
\end{aligned}$$

Figure 3.18: CSP ANN Semantics - General.

$layerRes.(layerNo - 1)$. The set $\{ c \}$ includes all events representing communications via c .

All events in $ANNHiddenEvs$ are hidden. This includes all events (Σ), except those of $layerRes.0$, representing inputs, $layerRes.layerNo$, representing outputs, and end . These define the visible behaviour of an $ANNController$.

We define the process $HiddenLayers$ via an iterated alphabetised parallel composition (\parallel) over an index i for hidden layers ranging from 1 to $layerNo - 1$. For each i , the layer-process $HiddenLayer(i, layerSize(i), layerSize(i - 1))$ for the i th layer is associated with the alphabet containing the set of events on $layerRes.(i - 1)$ and $layerRes.i$. In an iterated alphabetised parallelism, the parallel processes synchronise on the intersection of their alphabets. So, a layer-process $HiddenLayers$ synchronises with the process for the previous layer on $layerRes.(i - 1)$ and the process for the following layer on $layerRes.i$. So, the output events of each layer are used as the input events for the next layer.

The second argument $layerSize(i)$ passed to a layer-process is the value of the i -th element of $layerstructure$, that is, the number of nodes in the i -th layer if i is greater than 0, and $insize$ when i is 0. Similarly, the third argument $layerSize(i - 1)$ concerns the layer $i - 1$. In our example, $layerNo - 1$ is 1, and there is a single $HiddenLayer$ process, instantiated with arguments 1, 1, and 2. These are the values of $insize$ and $layerstructure(1)$ for the example.

$HiddenLayer(l, s, inpSize)$ is also defined by an iterated alphabetised parallelism: over an index i ranging from 1 to s , to compose s node processes $Node(l, i, inpSize)$ interacting via events in $\{| layerRes.(layer - 1) |\}$. This set contains all events the previous layer's node processes use for output because a node process requires the outputs from all nodes in the previous layer.

The $Node(l, n, inpSize)$ process represents the n th node in the layer l , which has $inpSize$ inputs. We define $Node(l, n, inpSize)$ as the parallel composition of interleaved $NodeIn(l, n, i)$ processes, with i ranging over 1 to $inpSize$, and a $Collator(l, n, inpSize)$ process. Interleaved processes ($|||$) do not synchronise.

$NodeIn(l, n, i)$ captures a weight application to an input. A $NodeIn$ process receives inputs via $layerRes.(l-1).n$ and communicates output through $nodeOut.l.n.i$. $NodeIn(l, n, i)$'s output is its input weighted by the constant $weight$, which is given by the expression $weights\ l\ n\ i$. After engaging in this output event, $NodeIn$ terminates, denoted by $Skip$.

We denote input via channels using $c?x \rightarrow P$, which defines the process that engages in an event $c.x$, then behaves like P . This process accepts input x over the channel c 's type. The output prefix $c!e \rightarrow P$ is a process that outputs (synchronises) on the specific event $c.e$ and then behaves like P . $Collator(l, n, inpSize)$ sums all values output by the $NodeIn$ processes and applies the $bias$ value, given by $biases\ l\ n$. The output of $Collator(l, n, inpSize)$ on $layerRes$ is the output of the node process. The definition of $Collator(l, n, inpSize)$ uses a local recursive process $C(l, n, i, sum)$; its extra argument is the accumulated sum of the outputs. In the base case $C(layer, node, 0, sum)$, we have an output sum , with the $bias$ term applied, subject to the activation function $ReLU$. In the recursive case $C(layer, node, i, sum)$, we get an input x via $nodeOut.l.n.i$, and a recursive call with a descending index $i - 1$, and the sum of x and sum .

Finally, the definition of $OutputLayer$ is similar to that of $HiddenLayer$.

The visible events of an ANN process are used to define the connection to other components of the RoboChart semantics and for defining termination. In our example, these are the events $layerRes.0$, $layerRes.2$, and end .

We rename the visible events of our ANN semantics to match the CSP events used to represent the events defined in the RoboChart model. For our example, these events, from Section 3.2.1.3, are $anewError$, $adiff$, and $angleOutputE$. We give the renaming of our example, $AnglePIDANN$, below.

$$\begin{aligned} AnglePIDANN \llbracket & layerRes.0.1 \leftarrow anewError.in \\ & layerRes.0.2 \leftarrow adiff.in \\ & layerRes.2.1 \leftarrow angleOutputE.out \rrbracket \end{aligned}$$

To illustrate the behaviour of our semantics, we describe a trace of a single iteration of our $AnglePIDANN$ component. In this example, we discretize all constants of $AnglePIDANN$ to enable model checking. Here, all weight values are 1 and all bias values are 0. In our example, the values communicated through the channels $anewError$ and $adiff$ are 1 and 0, respectively.

$$\begin{aligned} anewError.in.1 & \rightarrow \\ adiff.in.0 & \rightarrow \\ angleOutputE.out.1 & \end{aligned}$$

We now describe a trace of events on the component level: a trace of the AnglePIDANN semantics without hiding or renaming. Through the process described earlier, the events *anewError.in.0* and *adiff.in.0* correspond to the following *layerRes* events.

layerRes.0.1.1 →
layerRes.0.2.0 →

The *NodeIn* processes of *Node* synchronise with events on *layerRes.0*. So, each *NodeIn* receives the input and communicates the result of the weight application on the *nodeOut* channel. We give these events below.

nodeOut.1.1.1.1 →
nodeOut.1.1.2.0 →

Next, *Collator* receives these values, sums them, then applies the bias term to them. Following this, the *Collator* applies the *ReLU* function to the result (here, however, this does not change the result). The *Collator* process then communicates its result on the *layerRes* channel. We display the corresponding event below.

layerRes.1.1.1 →

Communication events on *layerRes.1.1* are synchronised with the *NodeIn* processes of the *OutputLayer*. Its *NodeIn* processes apply the weight to the value communicated (in this case 1), and communicates this value through the channel *nodeOut*. We show this event below.

nodeOut.2.1.1.1 →

This value is the product of the weight of the output layer’s node and the output of the previous layer’s single node.

Communications on *nodeOut.2.1.1* are synchronised with the *Collator* process of the output layer; this process then produces the final output of the layer (and the model). It communicates this value on the channel *layerRes* as shown below.

layerRes.2.1.1

Lastly, we rename *layerRes.2* to *angleOutputE.out*, as discussed earlier, obtaining a full trace of a single iteration of AnglePIDANN.

For a primary validation of our semantics, we have used a CSP model checker to compare the semantics of the AnglePIDANN to that of the controller AnglePID_C of the parallel version of the Segway model. For the latter, we have used the semantics automatically generated by the RoboChart tool⁵. We have used a discretised neural network to make model checking feasible. In this setting, we have been able to show refinement (in both directions) automatically. Further validation has been provided by implementing our semantics in Java using the JCSP package [77]. This has enabled simulation and assertion-based reasoning via JML in a setting where values are floating-point numbers.

In the next section, Section 3.5, we discuss this Java implementation and its implications for assertion-based reasoning.

⁵robostar.cs.york.ac.uk/robotool/

3.5 JCSP Validation

In this section, we present an implementation of our the parallel Segway model, including AnglePIDANN defined as an ANNController, as defined in Section 3.4, in Java⁶. We utilise the package JCSP, which provides an implementation of CSP in Java [77]. Our work here enables simulation and analysis by assertion-based reasoning of our RoboChart ANN components as presented in Section 3.2. We are able to perform this simulation because JCSP accommodates floating-point arithmetic.

In this section, and in what follows, we work only with nonterminating RoboChart models. For those, the event *end* and the exception operator used in the RoboChart semantics are unnecessary. Encoding the exception operator in JCSP and in Isabelle is left as future work.

In Section 3.5.1, we give a brief description of JCSP. Our encoding of the semantics of a RoboChart ANN controller in JCSP, and the Segway model, is the subject of Section 3.5.2.

3.5.1 JCSP

A key part of the JCSP implementation of CSP is the `CSPProcess` interface. It specifies the behaviour of a process; classes that implement this interface behave similarly to a process. `CSPProcess` contains a single abstract method: `run()`, which defines the behaviour of the process. We can define parallel composition and synchronisation of instances of these classes. We refer to instances of a class that implements the `CSPProcess` interface as *process objects*, to distinguish them from CSP process.

In order to communicate values between these processes, CSP channels are implemented in JCSP through the class `Channel`. A `Channel` can communicate either an object or an integer. Additionally, the `Channel` class provides static methods for constructing various types of writers and readers. In a `Channel`, channel synchronisation is defined as synchronisation between one reader object, and one writer object. These writer and reader objects are defined using two dedicated interfaces: `ChannelInput` for the reader and `ChannelOutput` for the writer.

`ChannelInput` and `ChannelOutput` are superinterfaces for other specialised forms of reader and writer. Similarly, specific types of channel are specified using interfaces. One example of a `Channel` interface is `One2OneChannel`, which specifies a channel that relates a single reader to a single writer object. Another sub-interface of `Channel` is `Any2OneChannel`, which defines a channel that attaches multiple writers to a single reader. This, however, does not allow multiple synchronous writers to one reader. This is because each writer object can only synchronise with a single reader, internally.

To illustrate channels in JCSP, we present below an example of creating all required objects for a `One2OneChannel`. We note that `Channel.one2oneChannel()` is a static method of `Channel` designed to create a `One2OneChannel`.

```

1   One2OneChannel c = Channel.one2oneChannel();
2   ChannelInput in = c.in();
3   ChannelOutput out = c.out();

```

⁶<https://docs.oracle.com/javase/7/docs/technotes/guides/language/>

In this example, we create a single channel `c` of the type `One2OneChannel`, and we use this to define a reading end `in` and a writing end `out`. These two objects, `in` and `out`, are linked together, internally, such that writing calls on `out` are synchronised with reading calls on `in`. These can be passed to process objects to be used as required. Notably, if either is not available in the context of a running process, the processes deadlock.

To link CSP concepts to JCSP, we present in Table 3.3 how the CSP operators we use are implemented in JCSP. We describe further details on these operations as we use them.

Table 3.3: CSP Operators

Symbol	Name	JCSP Implementation
$Skip$	Skip	<code>new Skip();</code>
$P \parallel [a] Q$	Parallel Composition	<code>new Parallel(new CProcess[] {P, Q});</code>
$c?x \rightarrow P$	Input	<code>(ChannelInput)c.read();</code>
$c!e \rightarrow P$	Output	<code>(ChannelInput)c.read();</code>

We can define parallel composition for process objects by using an instance of the class `Parallel`. We can construct an instance of `Parallel` by providing an array containing the objects for processes to be executed in parallel. The `Parallel` class implements `CProcess`, so instances of `Parallel` are themselves process objects. The behaviour of `Parallel` instances is defined as the parallel composition of its constituent `CProcess` arguments. This composition respects the synchronisation present in channels and other constructs. In Table 3.3, we create a new `CProcess` array containing `P` and `Q`. Here, `P` and `Q` are process objects.

In the following section, we discuss how we can use JCSP to validate our ANN components, based on our CSP model from Section 3.4.

3.5.2 ANN Component Validation

In the CSP model, as already explained, each layer interacts with its proceeding layer using the channel `layerRes`, given in Equation (2) of Section 3.4. We implement this in JCSP as:

```
1 One2AnyChannel[][] layerRes = new One2AnyChannel[layerNo+1][];
```

The channel `layerRes` is ultimately used to define events `layerRes.layer.index.value` that represent a communication of a `value` from a node identified by `index` in the given `layer` to another node in the next layer. In JCSP, we represent each such communication means between processes, here, processes for nodes, by a channel. We, therefore, define `layerRes` as a 2D array of type `One2AnyChannel`. Each channel identified by `layerRes[layer][index]` is used to represent events `layerRes.layer.index`. In CSP, for use in FDR, we need to identify a limit `maxSize` for the values of `index`. In Java, this can be defined on a per layer basis instead. We take advantage of this facility to avoid the creation of unnecessary channels when the layers have different numbers of nodes. Finally, we use `One2AnyChannel` because each node process in a layer needs to write to every node in its proceeding layer.

The events `layerRes.0.1` represent the inputs to the ANN model. We represent these events in JCSP using `One2AnyChannel layerRes[0][1]`. The definition of `Value` is determined by the RoboChart model.

In our JCSP implementation, we use double-precision floating-point arithmetic, `double` in Java, for this implementation. For the sake of illustration, we could represent the event `layerRes.0.1.0` in JCSP through two function calls as follows. First, we would write 0 on the writing end, using `layerRes[0][1].out().write(0)`. To read this value from the appropriate reading end, we would use `layerRes[0][1].in().read()`. These two function calls represent an occurrence of the event `layerRes.0.1.0`.

In our example, the class `AnglePIDANN` represents the controller `AnglePIDANN`, as presented in Figure 3.11. We construct a `AnglePIDANN` in line 11 of Figure 3.19. The constructor of this class is empty, as its behaviour is defined by the constants of the model.

We implement the constants using the public interface `Constants`, that stores the constants as `final` and `static` variables. The use of the `final` keyword makes their values immutable. All classes implement `Constants`, representing the CSP constants we discuss in Sections 3.4 and 3.2.

We represent CSP sequences as arrays because ordering is preserved, duplicates are preserved, and there is native support for them in Java. Here, we represent the type `Value` through a Java `double`, as mentioned. So, all constants which are of type of `Value` in CSP, are of type `double` in JCSP. As an example, the types of both `weights` and `biases` are arrays of `double` values in JCSP, where they are sequences of type `Value` in CSP.

We show an example of the use of a the complete `Segway` module defined using JCSP in Figure 3.19. The class `Segway` presented in this example implements a simplified, but equivalent, version of the model of the `Segway` semantics.

The process definition, of the semantics of the module `Segway`, is given in the implementation of the `run` and `runANN` methods of the `Segway` class. The `run` method, starting on line 14, represents the standard configuration of the parallel `Segway` model, as presented in Figure 3.4. This method defines that the following process objects synchronise in parallel: `segwayRP`, representing the semantics of `SegwayRP`; `segwayController`, representing the semantics of `SegwayController`; and `anglePIDC`, representing the semantics of the controller `AnglePID_C`. The method `runANN`, on the other hand, defines a process where an ANN controller, `AnglePIDANN` (Figure 3.11), replaces the controller `AnglePID_C`. In this version, the `Segway` behaves as the parallel composition of `segwayRP`, `segwayController`, and `anglePIDANN` process objects.

This implementation is for a single configuration of the values of the `Segway` module's visible events. We construct `SegwayRP` objects using a constructor that receives five parameters, each representing the value of a corresponding visible event of the `SegwayRP` robotic platform: `currAngle`, the value of the event `angle`; `currGyroX`, the value of the event `gyroX`; `currGyroZ`, corresponding to the event `gyroZ`; `currLeftVel`, corresponding to the event `leftMotorVelocity`; and `currRightVel`, the value communicated by `rightMotorVelocity`. We note that the event `gyroY` is not engaged in by the sequential nor the parallel version of `BalanceSTM`, so it not implemented here. We construct the process object `segwayRP` using this method in line 8 of Figure 3.19.

The class `ANN` is the core of our encoding of the CSP semantics of RoboChart ANN controllers. Its implementation matches the CSP definition of the process `ANN` presented in Section 3.4. The class `AnglePIDANN` calls this process internally, as it represents an ANN controller.

In the definition of the CSP process `ANN`, Figure 3.18, we hide all events except for those

```
1 public class Segway implements CSProcess {
2     SegwayRP segwayRP;
3     SegwayController segwayController;
4     AnglePID anglePIDC;
5     AnglePIDANN anglePIDANN;
6
7     public Segway(double currAngle, double currGyroX, double currGyroZ, double
8         currLeftVel, double currRightVel) {
9         this.segwayRP = new SegwayRP(currAngle, currGyroX, currGyroZ, currLeftVel,
10            currRightVel);
11        this.segwayController = new SegwayController();
12        this.anglePIDC = new AnglePID();
13        this.anglePIDANN = new AnglePIDANN();
14    }
15
16    public void run() {
17        new Parallel(new CSProcess[] {
18            segwayRP,
19            segwayController,
20            anglePIDC
21        }).run();
22    }
23
24    public void runAnn() {
25        new Parallel(new CSProcess[] {
26            segwayRP,
27            segwayController,
28            anglePIDANN
29        }).run();
30    }
31 }
```

Figure 3.19: The semantics of the parallel Segway model implemented in Java.

indexed by *layerRes.0*, *layerRes.2*, and *end*. More generally, we hide all events, except those representing the inputs and outputs of the ANN. As there is no explicit hiding operator in JCSP, we use the Java visibility keywords to represent event hiding. To capture this, we define the channel `layerRes` using the Java `private` keyword, and we represent the visible events using the following attributes of ANN.

```

1 public ChannelOutput[] visibleInputs;
2 public ChannelInput[] visibleOutputs;

```

Here, the array `visibleInputs` represents the events indexed by *layerRes.0*. This attribute is defined through the following: `visibleInputs = Channel.getOutputArray(layerRes[0])`, meaning `visibleInputs` represents input on the channels indexed by *layerRes.0*. Here, the method `Channel.getOutputArray(c)` returns the `ChannelOutput` object array corresponding to the array of channels `c`. The array `visibleOutputs` represents the events of the last layer, defined through `visibleOutputs = Channel.getInputArray(layerRes[layerNo])`; the method `getInputArray` operates similarly to `getOutputArray`, but returns the reading ends, instead of the writing ends, of the channel.

The implementation of ANN uses two process objects operating in parallel: `hiddenlayers` and `outputlayer`. We give the Java code below.

```

1 new Parallel(new CSPProcess[] {
2     (HiddenLayers) hiddenlayers,
3     (OutputLayer) outputlayer
4 }).run();

```

Here, line 1 creates a new parallel process, as described above. Line 2 references our hidden layers process, `hiddenlayers`; this represents the process *HiddenLayers*, presented in Figure 3.18. Line 3 references a process `outputlayer`, which implements the CSP process *OutputLayer*. This reflects the behaviour of the CSP process *ANN*.

Each `HiddenLayers` process object is composed of multiple `HiddenLayer` process objects. Our `HiddenLayer` class represents a hidden layer in our ANN. We present the constructor of this class below:

```

1 public HiddenLayers(One2AnyChannel[][] layerRes)

```

The CSP process *HiddenLayers* does not contain any parameters; in its JCSP implementation, however, we define a single parameter. In JCSP, we provide each process with only the channel and channel end objects required by that process. Despite, we take the channel `layerRes` as a parameter, to hold the full channel object. `HiddenLayers` does not directly engage in events on those channels but it passes the reading and writing ends onto the processes that require them.

The CSP process `HiddenLayers` behaves as the alphabetised parallel composition of all *HiddenLayer* processes, from Figure 3.18 of Section 3.4. We implement that in JCSP through the use of the `Parallel` object, as defined in Table 3.3. In our example, this is composed of a single `HiddenLayer` process. We give the structure of this process below.

```

1 new Parallel((HiddenLayer[]) layers).run();

```

Here, we define parallel composition of all processes in an array of `HiddenLayer` process objects referred to as `layers`. This implements the parallel composition of all layer processes in `HiddenLayers`.

Next, we define the class `HiddenLayer`, which represents the process `HiddenLayer`. We present its constructor below.

```

1 public HiddenLayer(int layer, int size, int inpSize,
2     ChannelInput[] layerInput, ChannelOutput[] layerOutput)

```

Our constructor of `HiddenLayer` requires five parameters: `layer`, `size`, `inpSize`, `layerInput`, and `layerOutput`. The first three parameters correspond one-to-one to a parameter in CSP: `layer` represents to the CSP parameter l , `size` corresponds to s , and `inpSize` represents $inpSize$. The final two parameters store the `ChannelInput` and `ChannelOutput` objects required to connect this layer to the rest of the model.

In our CSP model, the process `HiddenLayer` is composed of the parallel composition of processes for its nodes. Following this, our `HiddenLayer` process behaves as the parallel composition of process objects of the type `Node`, a class that represents a `Node` process. We display this in JCSP, below, where `nodes` is an array of `Node` objects.

```

1 new Parallel((Node[]) nodes).run();

```

We note that we define output layers in a very similar way to the class `HiddenLayer`, using the class `OutputLayer`.

Below, we present the class `Node`, which represents the `Node` process, in JCSP.

```

1 public Node(int layer, int node, int inpSize,
2     ChannelInput[] nodeInputs, ChannelOutput nodeOutput)
3

```

We recall that we require three parameters in our CSP process representing a node (l , n , and $inpSize$). Similarly to `HiddenLayer`, the first three parameters constructor of our `Node` class represent the CSP parameters: `layer` represents l , `node` represents n , and `inpSize` represents $inpSize$. The last two parameters store the reading and writing ends that this process uses: `nodeInputs` for input from the previous layer's nodes, and `nodeOutput` for the single output event of a node.

We define the class `NodeIn` to represent `NodeIn` processes, and the class `Collator` to represent `Collator` processes. According to the CSP process `Node`, we use the following code snippet in Java to define the behaviour of a `Node`.

```

1 new Parallel(
2     new CSProcess[] {
3         new Parallel((NodeIn[]) nodeIns),
4         (Collator) collator
5     }
6 ).run();

```

Here, `nodeIns` is an array of type `NodeIn`, and `collator` is of type `Collator`.

Each `NodeIn` communicates values to its respective `Collator` process along a single JCSP channel object. These processes communicate on channels from the channel `nodeOut`. We define this as a `Any2OneChannel`, as this channel connects multiple `NodeIn` writers to one `Collator` reader. We define the constructor of `NodeIn` below.

```
1 public NodeIn(int layer, int node, int index, ChannelInput input, ChannelOutput
   output)
```

Here, again, `layer` corresponds to the CSP parameter l , `node` corresponds to n , and `index` corresponds to i . In this process, we receive input from a `Node` via `input`, and output its output to a `Collator` through `output`.

We define the `nodeOut` channel with the following.

```
1 Any2OneChannel nodeOut = Channel.any2one();
```

This represents the `nodeOut` channel from Section 3.4. There must be a single `Any2OneChannel` for each node in our ANN. Each `Node` process defines a `nodeOut` channel, and it passes the reading and writing ends to the `NodeIn` and `Collator` processes. Each `NodeIn` receives values from the previous layer, from the appropriate channels on `layerRes`. Then, it writes its result to the `nodeOut` channel. Next, its `Collator` receives the values from the `NodeIn` processes through `nodeOut`, and finally writes the result of the node to the next layer via `layerRes`.

Lastly, we introduce the constructor of `Collator` below.

```
1 public Collator(int layer, int node, int inpSize,
2               AltingChannelInput input, ChannelOutput output)
```

Here, the constructor of `Collator` takes five parameters: `layer`, `node`, `inpSize`, `input`, and `output`. As with the other classes, `layer` represents the parameter l , `node` represents the parameter n , and `inpSize` represents the `inpSize` parameter of the `Collator` process. We require a single `AltingChannelInput` `input` to read in the results from the `NodeIn` processes. The interface `AltingChannelInput` enables a choice between channel inputs, allowing the output from the `NodeIn` processes to be accepted in any order. Finally, we require the parameter `output`, to communicate its result to the next layer.

So far, we have presented the pattern of interaction between the classes that form our JCSP implementation. The actual behaviour of each class is defined in its `run` method. We present below the `run` method for the `NodeIn` class, as this forms the basis of our JCSP implementation.

```
1 public void run() {
2     double in;
3     in = (Double) input.read();
4
5     double transfer = in * weights[layer][node][index];
6     nodeOut.write(transfer);
7 }
```

In this context, `input` is an instance of `ChannelInput`, which represents the input to this node from the previous layer. The variable `output` is an instance of `ChannelOutput`, which writes to its `Collator`. The weight of this `NodeIn` process is defined using the appropriate index of the array `weights`, which represents the *weights* CSP constant (see Section 3.4). In line 1, we initialise the method. In line 3, we set the local variable `in` to be the result of the method `read()` of `ChannelInput`, cast as a `double` because of the precision of this implementation, and `read()` returns a value of type `Object`. In line 5, we calculate the output of the `NodeIn`, which is the product of `in` and `weight`, just like in the CSP *ANN* process, presented in Figure 3.18. This value is assigned to `transfer`, and, finally, in line 6, we write the value `transfer` to the `ChannelOutput` `nodeOut`, which is received by the `Collator` process object.

The *Collator* process synchronises with the next layer’s *NodeIn* processes. Each *Node* receives the events from all *Collators* in the previous layer, so multi-synchronisation is present in this system. Implementing multi-synchronisation in JCSP requires a dedicated protocol, as each event in JCSP may only synchronise between two processes. Freitas and Cavalcanti [32] present a protocol to implement multi-synchronisation in JCSP. Additionally, a protocol to implement multi-synchronisation in Handel-C is presented in [72]. These protocols handle general multi-synchronisation.

In this work, we use a special case of multi-synchronisation. This is because each *Collator* writes to a single channel, and it is always the unique writer to this channel. There are any number of *Node* processes reading from a single channel, so there are any number of readers. We can represent this case of multi-synchronisation using a simpler protocol.

We have proved this in CSP. We have first defined a specification that contains multi-synchronisation, then defined an implementation that does not, and proved that this implementation refines the specification. We then implement the resulting process in JCSP.

In the specification, we represent the *Collator* process as a *Writer* process, and each *Node* as a *Reader* process. To demonstrate this protocol, we consider a system with three *Writer* processes, each writing on a separate channel. These writers are interleaved, such that there is no ordering between events. Finally, the communications between each *Writer* and *Reader* use multi-synchronisation. In particular, each communication is synchronised between one *Writer* and three *Reader* processes.

In the implementation, each *Writer* engages with as many events as there are readers, instead of one, where each of these events synchronises with a single *Reader*. We also define a new channel to accommodate this, and show its relationship to the specification using renaming and hiding. We have proven refinement in both directions using FDR, such that we have proven equality between the specification and the implementation.

We implement this protocol in JCSP through the `Collator` engaging in as many output events as there are nodes in the following layer. This implements the protocol because the number of readers assigned to each `Collator` process objects is equal to the number of nodes in the following layer.

In the next and final section, we discuss the usage and implementation of our encoding.

3.5.3 Discussion

We have used our JCSP implementation for simulating ANNs defined in our CSP framework, as we can use floating-point arithmetic. For a previous version, we have also enhanced

this implementation by adding JML annotations, which enables theorem proving via formal verification tools.

To validate our JCSP implementation, and, the semantics that it encodes, we have defined two other implementations of the `AnglePIDANN` component: a C++ implementation, and a python implementation. To validate these, we have ran them against a test data set, of size 30,000, formed of values with consistent intervals across the normalised range. We have observed a maximum error of $1e - 6$ across all three implementations. This error value is expected, as the C++ implementation uses single-precision values and each compiler's implementation varies.

3.6 Final Considerations

This concludes our description of how ANNs can be integrated into RoboChart. We have introduced ANN components in RoboChart in Section 3.2, given an overview of the RoboChart semantics in Section 3.3, and presented CSP semantics for our ANN components in Section 3.4, and provided an implementation of these in Section 3.5. These implementations can be used for validation and simulation of our ANN components. Our CSP semantics can be used to provide early validation of our ANN components, and verification via model-checking of discretised neural network components. In the next chapter, we present a strategy to verify properties of these models, which capture the software of RAAI systems.

Chapter 4

Verification Strategy

In this chapter, we describe a technique to verify robotic systems involving ANN components. Our technique proves that replacing an existing standard RoboChart controller with an ANN controller is sound with respect to a conformance notion we introduce here to cater for numerical imprecision. Although we focus here on verification with respect to RoboChart controllers whose behaviour is specified by state machines, our approach also applies to verification based on RoboChart operations and our new ANN operations.

The use of model checking, even in the absence of a neural network using real numbers, is challenging. We pursue instead a theorem-proving approach. For that, we take advantage of the predicative UTP semantics of Circus (and CSP). Namely, we consider the encoding of our semantics in a UTP theory. Specifically, we use the theory of reactive contracts, which captures the failures-divergences semantics of CSP [30].

In this chapter, we describe a pattern of reactive contracts for the semantics of our ANN components in Section 4.1, which we then justify in Section 4.2, and in Section 4.3 we describe a contract pattern for those RoboChart controllers that we can replace with ANNs. Next, in Section 4.4, we define a relation that formally establishes conformance between reactive contracts. In Section 4.5, we present our system-level verification technique, which establishes *system-level* conformance. We automate this technique using a combination of Isabelle and Marabou. Finally, we provide concluding remarks in Section 4.6.

4.1 General Pattern of ANN Contracts

In this section, we first introduce the theory of reactive contracts, in Section 4.1.1. We then describe the general pattern of contracts for our ANN components, in Section 4.1.2.

4.1.1 UTP Reactive Contracts

UTP is a semantic framework to describe concepts to give denotational semantics to a wide range of computational paradigms. UTP is based on a predicative alphabetised relational calculus expressed pointwise. Every UTP predicate has an alphabet of variables to which the predicate can refer. UTP uses binary relations to denote the observations that can be made about programs, separating initial observations and final observations.

In the UTP, a theory describes a semantic domain, characterising relations by predicates with a given alphabet and satisfying given healthiness conditions. Theories can be combined to define the semantics of richer languages. So, there is support to extend our work to consider our results in the context of languages other than RoboChart, that define reactive behaviours but perhaps also include notions of continuous time [28] and probability [115].

The alphabet of a theory is often composed of *observational variables*, which encode concepts essential to the paradigm defined by the theory, and *state variables*, concerned with the individual mechanisms denoted. In a UTP theory, the alphabet is usually open to extension, enabling state variables to be declared and added.

In our work, we use the UTP theory of *reactive contracts* [108, 29], which can represent CSP processes and has a rich set of calculational laws. Its observational variables are st , st' , ok , ok' , $wait$, $wait'$, ref , ref' , and tt . Here, st refers to the state of the process: the set of variables it uses. The observational variable ref represents the set of events the process refuses. Finally, tt describes the trace contribution: the events from tr' that the process has engaged in. The boolean observational variable ok represents the termination of the process, and the boolean $wait$ represents that the process is waiting for interaction with its environment. Neither ok nor $wait$ are referred to explicitly by any predicate in this theory but are implicit in the structure of a reactive contract.

The dashed versions of the alphabet variables represent later observations of their values. A predicate describes a relation by restricting the possible initial and later observations. There is no tt' because tt itself is an expression on the variables capturing the trace: $tr' - tr$.

These contracts take the form $[R_1[tt, st] \vdash R_2[tt, st, ref'] \mid R_3[tt, st, st']]$. The square brackets define the observational variables to which each predicate can refer. The precondition, R_1 , describes conditions on the pre-state st and the trace tt . Next, the pericondition R_2 describes a condition on the pre-state, the value of the trace tt , and which events are refused by referring to ref' . Lastly, the postcondition R_3 describes a condition on the state, the state update on st' , the program's effect on the post-state, and the final assignment of tt .

Here, we use operators $\mathcal{E}[t, E]$ and $\Phi[t]$, simplified versions of those in Def. 4.6 from [30], where we consider that a CSP process has no state. With \mathcal{E} , we can construct a pericondition stating that t has been observed, that $tt = t$, and the events in X are not refused, no event in X is in ref' . On the other hand, Φ constructs a postcondition, stating that the final trace observed is characterised by t . For example, $\mathcal{E}[\langle \rangle, \{| c |\}]$ stands for $\mathcal{E}[true, \langle \rangle, \{| c |\}]$, the predicate obtained by replacing tt and E with $\langle \rangle$ and $\{| c |\}$, and $true$ for s , denoting no restriction on st , in \mathcal{E} . Similarly, $\Phi[\langle \rangle]$ stands for $\Phi[true, id, \langle \rangle]$, where we replace tt with $\langle \rangle$, s with $true$, and st' with the identity id , denoting no change in state.

In the next section, we use these contracts to describe a pattern of reactive contracts which captures the semantics for our ANN components.

4.1.2 Reactive Contracts for ANN components

Def. 4.1 provides a pattern for contracts corresponding to an optimised version of the CSP process ANN in Fig. 3.18. The pattern is for the process defining one iteration of the ANN : the parallelism between *HiddenLayers* and *OutputLayer*. So, we consider one application of the ANN. With that, the compositionality of refinement allows us to make direct deductions about the overall ANN process.

To optimise reasoning, we eliminate the interleavings that allow inputs and outputs to be received and offered in any order and internal computations among and inside the layers to occur in any order. Our highly parallel semantics captures the common use of parallelisation to optimise the performance of implementations of ANNs. We have proved, however, that the different interleavings produce equivalent outputs once the internal events are hidden.

First, the model of the ANN is deterministic, so hiding the events representing the communications between the nodes (and the layers) introduces no nondeterminism. This means that the internal order of computation (as signalled by the events) in the layers and their nodes is irrelevant. Second, if we add a wrapper process that keeps that responsiveness for the inputs but feeds them to *ANN* in a fixed order, the values and responsiveness of outputs are maintained. With this, we have rigorous evidence that parallelisation is a valid implementation strategy and that we can use a simpler model for reasoning.

For brevity, in Def. 4.1, the contract is defined using a sequence *input* containing only the events representing inputs extracted from the trace *tt*. Formally, $input = tt \upharpoonright I$, here $I = \{ | layerRes.0 \}$. (We use \upharpoonright to denote sequence filtering.)

Definition 4.1 (ANN Component General Contract).

$$\begin{aligned}
 & \text{GeneralANNContract} \hat{=} \\
 & \left[\begin{array}{l}
 true_r \\
 \vdash \#input < insize \wedge \mathcal{E}[input, \{ | layerRes.0.(\#input + 1) \}] \\
 \vee \\
 \#input = insize \wedge \\
 \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
 \mathcal{E}[front \circ layeroutput(l, n), \{ last \circ layeroutput(l, n) \}] \\
 | \#input = insize \wedge \Phi[layeroutput(layerNo, layerSize(layerNo))] \\
 \vdash
 \end{array} \right]
 \end{aligned}$$

The pattern in Def. 4.1 is for contracts that require that the process does not diverge: the precondition is $true_r$. This is appropriate as no ANN diverges.

To define the pericondition and the postcondition, we specify the valid observations using the predicate operators \mathcal{E} and Φ . The pericondition characterises the stable intermediate states of the ANN where some or all inputs have been received. We identify these states by considering the size of *inputs*. When some of the inputs are available ($\#input < insize$), the trace is *input*, and the next input event $layerRes.0.(\#input + 1)$ is not refused.

When all inputs are available ($\#input = size$), we specify the trace of *layerRes* interactions up to where $layerRes.l.n$ has occurred using a function $layeroutput(l, n)$, where l and n are layer and node indices. In the pericondition, we consider all layer indices l and all node indices n in l , from 1 to $layerSize(l)$. The function $layeroutput(l, n)$ encodes the specification of the ANN, in terms of its structure, into a trace-based specification. For example, for an ANN with input size 2, with two nodes in its first layer, and all weights and biases defined as 2, if *tt* defines the *input* sequence as $\langle layerRes.0.1.1, layerRes.0.2.1 \rangle$, then $layeroutput(1, 2)$ is $\langle layerRes.0.1.1, layerRes.0.2.1, layerRes.1.1.6, layerRes.1.2.6 \rangle$. This reflects that the inputs in *input* are taken first, and the output of each node is the weighted sum of these inputs. Since our weights and biases are all 2, each node outputs 6, resulting in the trace above.

With $layeroutput(l, n)$, we define the entire trace up to and including the result of the calculation of the node n on the layer l , which is the last element of $layeroutput(l, n)$.

Therefore, the trace in the case $\#input = size$, where all inputs have been received, includes all elements in $layeroutput(l, n)$ but the last, denoted using the *front* function. We define the set of accepted events as the singleton containing the event $last \circ layeroutput(l, n)$.

To specify the postcondition, we first state that all inputs have been received, $\#input = insize$, and then we state that the trace contribution of this process, variable tt , must be equal to the expression $layeroutput(layerNo, layerSize(layerNo))$. Here, this expression represents the trace after the last node (that of index $layerSize(layerNo)$) of the last layer (that of index $layerNo$) has occurred, in other words, after all nodes in the ANN have terminated.

Next, we present the formal definitions of the functions we use to construct our general pattern in Definition 4.1, starting with the definition of *layeroutput*.

Definition 4.2 (Layer Output Function).

$$\begin{array}{l}
 \underline{layeroutput : \mathbb{Z} \times \mathbb{Z} \rightarrow \text{seq } Event} \\
 \forall l : 0 \dots layerNo \bullet \\
 \quad \forall n : 1 \dots layerSize(l) \mid \#input = insize \bullet \\
 \quad \quad layeroutput(0, n) = (1 \dots n) \upharpoonright input \\
 \quad \quad \wedge \\
 \quad \quad layeroutput(l, n) = \\
 \quad \quad \quad layeroutput(l-1, layerSize(l-1)) \hat{\ } \\
 \quad \quad \quad layercalc(lastn(layeroutput(l-1, layerSize(l-1)), \\
 \quad \quad \quad \quad \quad \quad \quad layerSize(l-1)), \\
 \quad \quad \quad \quad \quad \quad \quad l, \\
 \quad \quad \quad \quad \quad \quad \quad n)
 \end{array}$$

We require that all inputs are available ($\#input = insize$) for $layeroutput(l, n)$ to be well defined. This precondition is made explicit above.

We define *layeroutput* by two equations, one for a base case and one for a recursive case. In the base case, when l is 0, the result is the first n elements of *input*. In the recursive case, $layeroutput(l, n)$ is the concatenation of two sequences. The first is the complete trace of the previous layer, $layeroutput(l-1, layerSize(l-1))$, that is, the layer indexed by $(l-1)$. The second trace records the output of the current layer, index l , up to the node n . We describe the trace of the current layer using the function *layercalc*.

The function *layercalc*, Def. 4.3, defines the trace of a single layer in an ANN using three arguments. The first argument is the trace of the previous layer, used to calculate the current layer's output values. Here, we define that trace as the last $layerSize(l-1)$ elements of the trace of all previous layers defined using *layeroutput* as explained above. We extract the last $layerSize(l-1)$ elements using a function *lastn*, specified in Def. B.5. The second and third parameters specify the layer and node index of the node to calculate the trace up to.

We illustrate the application of *layeroutput* in Ex. 4.1. We consider an ANN with input size 2, with two nodes in its first layer, and all weights and biases are 2. We also define the *input* sequence as $\langle layerRes.0.0.1, layerRes.0.1.1 \rangle$.

Example 4.1 (Layer Output Function).

$$layeroutput(1, 2)$$

$$\begin{aligned}
&= \text{layeroutput}(0, \text{layerSize}(0)) \hat{\ } && [\text{Def. 4.2}] \\
&\quad \text{layercalc}(\\
&\quad\quad \text{lastn}(\\
&\quad\quad\quad \text{layeroutput}(0, \text{layerSize}(0)), \\
&\quad\quad\quad \text{layerSize}(0), \\
&\quad\quad\quad 1, \\
&\quad\quad\quad 2) \\
&= \langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle \hat{\ } \\
&\quad \text{layercalc}(\langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle, 1, 2) \\
&= \langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle \hat{\ } && [\text{Def. 4.3}] \\
&\quad \langle \text{layerRes.1.1.6}, \text{layerRes.1.2.6} \rangle \\
&= \langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1}, \text{layerRes.1.1.6}, \text{layerRes.1.2.6} \rangle
\end{aligned}$$

As said, our second function, *layercalc*, defines the trace of a single layer up to a given node n . Its parameters are the sequence pl of results of the previous layer, a layer index l , and a node index n . We specify this function in Def. 4.3.

Definition 4.3 (Layer Calculation Function).

$$\begin{array}{l}
\text{layercalc} : (\text{seq Event} \times \mathbb{Z} \times \mathbb{Z}) \rightarrow \text{seq Event} \\
\forall pl : \text{seq Value}; l : 1 \dots \text{layerNo} \bullet \\
\quad \forall n : 1 \dots \text{layerSize}(l) \mid \#pl = \text{layerSize}(l - 1) \bullet \\
\quad\quad \text{layercalc}(pl, l, 0) = \langle \rangle \\
\quad\quad \wedge \\
\quad\quad \text{layercalc}(pl, l, n) = \\
\quad\quad\quad \text{layercalc}(pl, l, (n - 1)) \hat{\ } \\
\quad\quad\quad \langle \text{layerRes.l.n.} \\
\quad\quad\quad\quad \text{relu}(\text{dotprod}(\text{dropseq}(pl), \text{weights } l \ n) \\
\quad\quad\quad\quad + \\
\quad\quad\quad\quad \text{biases } l \ n) \rangle
\end{array}$$

For *layercalc* to be well-defined, we require that sequence pl 's size is equal to the size of the previous layer, which is $\text{layerSize}(l - 1)$, because to define the result of any node, we require the complete trace of the layer preceding this node.

We define *layercalc* using two recursive equations. The base case is when the node index is 0; this evaluates to the empty trace because nodes are indexed from 1 in our ANN model. In the recursive case, this function evaluates to the concatenation of the sequence given by $\text{layercalc}(pl, l, (n - 1))$ (representing the trace up to the previous node), with a singleton sequence representing node n 's output event.

We define the trace for a node as a singleton sequence containing an event of the form $\langle \text{layerRes.l.n.v} \rangle$, where l and n are the layer and node indices, and v is the output of this node. In our ANN model, the output of a single node is given by an activation function applied to the weighted sum of the previous layer's output, with a given bias value added.

The previous layer's output is given by $\text{dropseq}(pl)$, specified in Def. B.6. It identifies a sequence of values v corresponding to a sequence of events of the form layerRes.l.n.v . We

represent the weights of this node using the sequence *weights l n*. Using these two sequences, we can calculate the weighted sum of this node using the *dotprod* (Def. B.4) function. Next, we add the *biases l n* value to this weighted sum, representing the bias application. Finally, we apply the *relu* function, presented in Def. B.3, capturing our activation function.

Next, we present an example, Ex. 4.2, where we apply *layercalc* with identical parameters as from Ex. 4.1: which are $(\langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle, 1, 2)$.

Example 4.2 (Layer Calculation Function).

$$\begin{aligned}
& \text{layercalc}(\langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle, 1, 2) \\
&= \text{layercalc}(\langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle, 1, 1) \hat{\wedge} \quad [Def. 4.3] \\
&\quad \langle \text{layerRes.1.2.relu}(\text{dotprod}(\langle 1, 1 \rangle, \langle 2, 2 \rangle) + 2) \rangle \\
&= \text{layercalc}(\langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle, 1, 1) \hat{\wedge} \quad [Def. B.4] \\
&\quad \langle \text{layerRes.1.2.relu}((4) + 2) \rangle \\
&= \text{layercalc}(\langle \text{layerRes.0.1.1}, \text{layerRes.0.2.1} \rangle, 1, 1) \hat{\wedge} \quad [Def. B.3] \\
&\quad \langle \text{layerRes.1.2.6} \rangle \\
&= (\langle \rangle \hat{\wedge} \langle \text{layerRes.1.1.6} \rangle) \hat{\wedge} \langle \text{layerRes.1.2.6} \rangle \\
&= \langle \text{layerRes.1.1.6}, \text{layerRes.1.2.6} \rangle
\end{aligned}$$

In the next section, we use the laws of reactive contracts and the definition of the CSP operators as reactive contracts [29] to prove that the pattern in Def. 4.1 captures our ANN component's CSP semantics, given fully in Figure 3.18.

4.2 ANN Pattern Justification

Using laws of reactive contracts and the definition of the CSP operators [29], we prove that the pattern in Def. 4.1 captures the RoboChart ANN semantics.

Theorem 4.1. *The semantics of*

$$\text{HiddenLayers} \llbracket \{ \{ \text{layerRes.}(\text{layerNo} - 1) \} \} \rrbracket \text{OutputLayer}$$

can be expressed using the pattern in Def. 4.1.

Proof. In this proof, we make a case for the precondition, pericondition, and postcondition separately. Using FDR, we have, as already mentioned, established that the *ANN* process in Fig. 3.18 is deterministic and that the order in which the inputs are accepted has no impact on the output. So, our contract is for a fully sequential version of that process. In particular, the parallelism between *HiddenLayers* and *OutputLayer* is given by the process below.

$$\begin{aligned}
\text{SANN} &= (\text{SANN_Layers} \llbracket \{ \{ \text{add_input}, \text{get_input} \} \} \rrbracket \text{Input}) \\
&\quad \setminus \{ \{ \text{add_input}, \text{get_input} \} \} \\
\text{SANN_Layers} &= \text{SANN_InputLayer}; (; l : 1 .. \text{layerNo} \bullet \text{SANN_Layer}(l)) \\
\text{SANN_InputLayer} &= ; i : 1 .. \text{insize} \bullet \text{SANN_InputNode}(i) \\
\text{SANN_InputNode}(i) &= \text{layerRes.0.i?}x \rightarrow \text{add_input}.x \rightarrow \text{Skip} \\
\text{SANN_Layer}(l) &= ; n : 1 .. \text{layerSize}(l) \bullet \text{SANN_Node}(l, n)
\end{aligned}$$

$$SANN_Node(l, n) = get_input?in \rightarrow layerRes.l.n.annoutput(l, n, in) \rightarrow Skip$$

We define the process $SANN$ as the parallel composition of $SANN_Layers$, which gives a sequential account of the calculations in the layers, and an $Input$ process, which records the sequence of values of the input nodes. $SANN_Layers$ and $Input$ synchronise on events add_input and get_input . The event $get_input?in$ is used to retrieve the sequence of inputs in . We record all inputs, the values of the events of the form $layerRes.0$, through $add_input.v$.

$SANN_Layers$ represents all hidden layers and the output layer of our ANN process. We define $SANN_Layers$ as $SANN_InputLayer$ sequentially composed with the sequential composition of $layerNo$ processes $SANN_Layer(l)$, one for each layer l . With $SANN_InputLayer$, we ensure that all inputs are received (in order) before the hidden-layer calculations start.

$SANN_InputLayer$ is defined by the sequential composition of *insize* $SANN_InputNode(n)$ processes, representing the n th input nodes. This process accepts any event $layerRes.0.n?x$ and records x in the $Input$ process (omitted here) via the event $add_input.x$.

$SANN_Layer$ is the sequential composition of processes $SANN_Node(l, n)$ for the n th node of a layer l . $SANN_Node(l, n)$ first retrieves the sequence in of inputs, then raises the event $layerRes.l.n.annoutput(l, n, in)$. For the purposes of this proof, we provide an implementation of our $annoutput$ function, Def. B.2, in CSPM, which calculates the output value of a given node with respect to an input sequence in .

We have used FDR to prove that the process $SANN$ is equivalent to the parallel composition of $HiddenLayers$ and $OutputLayer$, in parallel with a third process that fixes the order of the $layerRes$ events. Namely, we have enforced an ascending order of layer and node indices inside each layer. As we have explained, this fixed order does not affect the values of the ANN output events. We have proved this result for a variety of input and layer sizes.

The parallelism with the $Input$ process is an encoding in CSP for a data-rich process. UTP does not require such encoding because its CSP theory caters to an imperative data-rich view of processes. So, our justification is for $SANN_Layers$, with a view that it can access data directly rather than via communication with a memory process as in our encoding above.

Precondition Our precondition is *true* because the semantics is divergence free. We have established this via model checking and observed that we use one recursive process in $SANN$, which is in $SANN_Node$. This process cannot diverge because it is productive: it engages in an event before recursing. So, the recursion is guarded and has a unique fixed point [30].

Periccondition Next, we show that the periccondition of the contract for $SANN$ is that of $GeneralANNContract$ in Def. 4.1. We use the notation P_2 to refer to the periccondition of the contract capturing process P , and P_3 to refer to the postcondition of P . We prove this in Lemma B.2.

Postcondition Our proof of the postcondition is similar to that of the periccondition, we present this in Lemma B.3.

□

In the next section, we describe a pattern of reactive contracts for those RoboChart controllers that can be replaced by an ANN components, in order to compare them to our

pattern of reactive contracts for ANN components, presented here.

4.3 Cyclic Memoryless RoboChart Controllers

An ANN cannot implement reactive behaviour where input and output events are arbitrarily interspersed according to environmental interactions. So, we consider specifications that define a controller whose control flow alternates between taking inputs and producing outputs, that does not terminate, and does not contain memory across cycles. (This is the flow of simulations, for example.) For instance, the inputs of the `AnglePID`, as presented in Figure 3.6, are `anewError` and `adiff`, and the output is `angleOutputE` as indicated by the connections to the `SegwayController` (see Fig. 3.4). For such controllers, the RoboChart semantics can be captured by a reactive design of a particular format. For example, the `AnglePID` state machine can be captured by the following contract:

$$\begin{aligned}
 & \text{AnglePID_UTP} \hat{=} \\
 & [\text{true}_r \\
 & \quad \vdash \\
 & \quad \exists \text{currNewError}, \text{currDiff}, \text{currAngleOut} : \text{Value} \mid \\
 & \quad \quad \text{currAngleOut} = P * \text{currNewError} + D * \text{currDiff} \bullet \\
 & \quad \quad \text{wait}' \wedge (\text{tt} = \langle \rangle \wedge \text{anewError}.\text{currNewError} \notin \text{ref}' \vee \\
 & \quad \quad \quad \text{tt} = \langle \text{anewError}.\text{currNewError} \rangle \wedge \text{adiff}.\text{currDiff} \notin \text{ref}' \vee \\
 & \quad \quad \quad \text{tt} = \langle \text{anewError}.\text{currNewError}, \text{adiff}.\text{in}.\text{currDiff} \rangle \wedge \\
 & \quad \quad \quad \text{angleOutputE}.\text{currAngleOut} \notin \text{ref}') \\
 & \quad \vee \\
 & \quad \neg \text{wait}' \wedge \\
 & \quad \quad \text{tt} = \langle \text{anewError}.\text{currNewError}, \text{adiff}.\text{currDiff}, \text{angleOutputE}.\text{currAngleOut} \rangle \\
 &]
 \end{aligned}$$

This contract captures the behaviour of one iteration of the `AnglePID` state machine: it receives inputs via channels `anewError` and `adiff` and produces an output via `angleOutputE`. The local variables of `AnglePID` are quantified and defined according to the RoboChart model in terms of constants P and D . The precondition is just true_r , as the process cannot diverge.

The postcondition comprises two parts: either the process is waiting on interaction (wait'), or not ($\neg \text{wait}'$). In the first case, three cases are distinguished by the trace contribution tt : no input events have happened, just `anewError` has occurred, or both `anewError` and `adiff` have been provided. In each case, we state that ref' does not contain an event, denoting that we accept that event in this state. When wait' is false, tt contains all inputs and the output. The local variables record the values communicated, according to the definition of `AnglePID`.

The contract for the `AnglePID` follows a pattern defined below for a cyclic controller, where we consider inp and out to be lists of input and output events. For every event, we have a quantified variable that records the value communicated via that event: x_1 to $x_{\#\text{inp}}$ for inputs, and y_1 to $y_{\#\text{out}}$ for outputs. We also consider a predicate p to capture the permissible values these variables can take, according to the RoboChart model.

pattern for a cyclic controller, the inputs and outputs are related by the predicate p . We can, for example, define even nondeterministic behaviour with this predicate. Finally, the alphabet of events in the patterns is different: one is based on the *layerRes* events and the other on RoboChart application-specific events to represent inputs and outputs.

To summarise, in this section, we have presented a pattern of RoboChart components that ANN components can replace, in Definition 4.4. Those components have the following properties: they are memoryless, they do not terminate, and they receive all input events before engaging in output events.

In the next section, we discuss a strategy to formally verify properties which relate patterns of ANN components and patterns of cyclic RoboChart controllers.

4.4 Conformance

In our approach to verification, we want to replace a RoboChart controller with an ANN component. Therefore, we want to prove that the ANN component implements all behaviours of the RoboChart controller, or in other words, we want to show that an ANN component is a refinement of the RoboChart controller. ANN components, however, contain numerical imprecision in their output, so we allow an error tolerance on the values communicated by the output events of an ANN component. Formally, we define a conformance relation $Q \text{ conf}(\epsilon) P$ that holds if, and only if, Q is a refinement of P , where the value of P 's output events can vary by at most by a non-negative error value ϵ as formalised below.

Definition 4.5 (Conformance Relation).

$$\begin{array}{|l}
 \hline
 \text{conf} : \mathbb{R} \rightarrow Proc \leftrightarrow Proc \\
 \hline
 \forall \epsilon : \mathbb{R}; P, Q : Proc \mid \epsilon \geq 0 \bullet \\
 \quad Q \text{ conf}(\epsilon) P \\
 \quad \Leftrightarrow \\
 \quad \exists s : seq\ Event; a : \mathbb{P}\ Event \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
 \quad P[s, (\alpha P \setminus a)/tt, ref'] \sqsubseteq Q
 \end{array}$$

We say that $Q \text{ conf}(\epsilon) P$ if, and only if, Q is a refinement of $P[s, (\alpha P \setminus a)/tt, ref']$, that is, we accept P as a specification that restricts the trace s and the refusals $\alpha P \setminus a$, instead of tt and ref' , where s and a are approximations of tt and the set a of acceptances as captured by relations $\text{seqapprox}(\epsilon)$ and $\text{setapprox}(\epsilon)$. Here, $s_1 \text{ seqapprox}(\epsilon) s_2$ relates sequences s_1 and s_2 if, and only if, s_1 differs from s_2 just in that its output values are within ϵ of those of s_2 . $A_1 \text{ setapprox}(\epsilon) A_2$ if, and only if, their input events are the same, and the output events are the same, but the communicated values differ by a maximum tolerance of ϵ .

Here, we use the type *Proc* to refer to any reactive contract that implements a CSP process, and, therefore, this conformance definition relates any reactive contracts.

We refer to the $\text{seqapprox}(\epsilon)$ relation, as the relation defined by seqapprox under ϵ , as we use seqapprox primarily as a relation.

Definition 4.6 (Sequence approximation relation).

$$\text{seqapprox} : \mathbb{R} \rightarrow (\text{seq Event} \leftrightarrow \text{seq Event})$$

$$\forall \epsilon : \mathbb{R} \mid \epsilon \geq 0 \bullet$$

$$\forall s_1, s_2, s_3 : \text{seq Event} \mid s_2 = s_1 \upharpoonright (I \cup O) \wedge \#s_3 = \#s_2 \bullet$$

$$s_1 \text{ seqapprox}(\epsilon) s_3 \Leftrightarrow$$

$$\forall i : \text{dom } s_2 \bullet$$

$$(s_2(i) \in I \Rightarrow s_3(i) = s_2(i)) \wedge$$

$$(s_2(i) \in O \Rightarrow \text{ev}(s_3(i)) = \text{ev}(s_2(i)) \wedge$$

$$\exists r : (-\epsilon, \epsilon) \bullet \text{value}(s_3(i)) = \text{value}(s_2(i)) + r)$$

We use the sets I and O to refer to the sets containing just the input and output events from the alphabet of an arbitrary process. These have to be instantiated for each process, based on its definition. In the definition above for $s_1 \text{ seqapprox}(\epsilon) s_3$, we define the sequence s_2 as s_1 filtered to contain only input or outputs events, as these are the only events relevant for our notion of conformance.

The above states that $s_1 \text{ seqapprox}(\epsilon) s_3$ if, and only if, for every index i of s_2 , every event $s_2(i)$ is either an input event, and identical to $s_3(i)$, or an output event, then the channel of both events is identical, and the *value* of $s_2(i)$ is within ϵ of the value of $s_3(i)$. We use the function ev to extract the channel identifier of an event, and the function value to refer to the value communicated by the event. The specific definition of these functions varies depending on the type of channel used in the process.

The relation $\text{seqapprox}(\epsilon)$ is reflexive as long as we are comparing sequences that contain only input or output events. We prove this in Lemma B.4.

Lemma B.5 proves that if we are comparing two sequences of output events, with identical lengths and event names, we can simplify the $\text{seqapprox}(\epsilon)$ relation to an inequality predicate. In other words, if we have these conditions, the sequence s_3 is an approximation of s if and only the difference in the values communicated by each event of s and s_3 is less than ϵ .

The relation $\text{setapprox}(\epsilon)$ defines what it means for a set to be an approximation of another. Its definition below is similar to $\text{seqapprox}(\epsilon)$.

Definition 4.7 (Set approximation relation).

$$\text{setapprox} : \mathbb{R} \rightarrow (\mathbb{P} \text{Event} \leftrightarrow \mathbb{P} \text{Event})$$

$$\forall \epsilon : \mathbb{R} \mid \epsilon \geq 0 \bullet$$

$$\forall r_1, r_2, r : \mathbb{P} \text{Event} \mid r_2 = r_1 \cap (I \cup O) \wedge \#r = \#r_2 \bullet$$

$$r_1 \text{ setapprox}(\epsilon) r \Leftrightarrow$$

$$\forall e : r_2 \bullet$$

$$(e \in I \Leftrightarrow e \in r) \wedge$$

$$(e \in O \Leftrightarrow \exists e_1 : r \bullet \text{ev}(e) = \text{ev}(e_1) \wedge$$

$$\exists c : (-\epsilon, \epsilon) \bullet \text{value}(e) = \text{value}(e_1) + c)$$

Similarly to seqapprox , we say that $r_1 \text{ setapprox}(\epsilon) r$ if, and only if, for any event e in r_2 , whose value is the set obtained by restricting r_1 to inputs and outputs, e is either an input event, and therefore e is in the set r , or an output event, and then there exists an event e_1 such that $\text{ev}(e) = \text{ev}(e_1)$ and the *value* of e_1 is within ϵ of the *value* of e . This relation is used to compare acceptance sets, where for conformance to hold, if an event is accepted in r_1 , then there must exist a corresponding event in r .

Similar to $seqapprox(\epsilon)$, we can show that $setapprox(\epsilon)$ is reflexive if all events are either input or events. This assists in reasoning concerning $setapprox(\epsilon)$. We present this, formally, in Lemma B.6.

Lemma B.7, similar to Lemma B.5, formalises the conditions under which comparing the values communicated by the events in the sets is sufficient to prove set approximation. In this case, these conditions are that: both sets contain solely output events, both sets are of the same size, and that we have corresponding event identifiers in both sets.

In the next section, we use the formal material presented here and in previous sections to define our system-level verification technique involving ANN components.

4.5 System-Level Conformance

As presented in Section 4.4, our conformance definition enables component-level conformance. In this work, however, we are concerned with *system-level* conformance to verify systems involving ANN components, not just the components themselves. So, in this section, we use the definitions presented in Sections 4.1.2, 4.3, and 4.4 to define a system-level verification approach concerning ANN components and cyclic RoboChart controllers.

Our verification approach starts with an abstract RoboChart model. That model can be refined using the structural rules of RoboChart justified by its CSP semantics and refinement relation. These rules are out of scope here, but we refer to [63] for examples of the kinds of laws of interest. Refinement in our approach may need to be used to specify a cyclic controller. In our example, we had to extract the AnglePID state machines out of the SegwayController where it was originally to obtain the Segway module in Fig. 3.4.

With a refined model, we can identify a cyclic controller CDes to be implemented by an ANN and prove conformance of the ANN controller to CDes according to $conf(\epsilon)$. The following result justifies the joint use of refinement and $conf(\epsilon)$.

Theorem 4.2. $P \sqsubseteq Q \wedge R \text{ conf}(\epsilon) Q \Rightarrow R \text{ conf}(\epsilon) P$

This ensures that the ANN conforms to the original specification.

In the context of our work regarding ANN controllers in RoboChart, the proof of conformance is in the following form (see Figure 3.18).

$$(Q \setminus ANNHIDDENEVENTS)[inp/layerRes.0, out/layerRes.layerNo] \text{ conf}(\epsilon) P \quad (4.1)$$

Here, Q is a reactive contract that instantiates the pattern in Def. 4.1, and P captures the semantics of a cyclic controller described using the pattern in Def. 4.4. As said, our general contract for ANN components does not capture the hiding in the CSP semantics (Figure 3.18), so we add it to Q above. Moreover, the pattern is concerned with $layerRes$ events and the specification with RoboChart events. So, we substitute $layerRes.0$ and $layerRes.layerNo$ with the input and output events captured by the sequences inp and out .

For our example, we use the reactive contract, which captures the component AnglePIDANN. Besides hiding the $layerRes.1$ events, we rename $layerRes.0.1$ and $layerRes.0.2$ to $currNewError$ and $layerRes.0.2$ to $currRadiff$, and $layerRes.2.1$ to $currAngleOutput$.

In the UTP theory of reactive contracts general hiding is (as yet) undefined. So, as a compromise, we define relations capturing event hiding for specifically those contracts defined using the pattern in Definition 4.1.

For an instance Q of the pattern in Definition 4.1, $Q \setminus S = [true_r \vdash Q_2 \setminus_{peri} S \mid Q_3 \setminus_{post} S]$, where the hide operators \setminus_{peri} and \setminus_{post} are defined in Appendix B.

Below, we define theorems that identify verification conditions sufficient to prove conformance for instances of our patterns, taking advantage of structural similarities in the patterns.

Theorem 4.3. $Q \text{ conf}(\epsilon) P$ provided

$$[Q_2 \Rightarrow \exists s : \text{seq Event}; a : \mathbb{P} \text{ Event} \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox} a \bullet P_2[s, (\alpha P \setminus a)/tt, ref']]$$

and

$$[Q_3 \Rightarrow \exists s : \text{seq Event} \mid tt \text{ seqapprox}(\epsilon) s \bullet P_3[s/tt]]$$

where Q and P are instances of the patterns in Definitions 4.1 and 4.4.

Proof.

$$\begin{aligned} & Q \text{ conf}(\epsilon) P \\ &= \exists s : \text{seq Event}; a : \mathbb{P} \text{ Event} \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet P[s, (\alpha P \setminus a)/tt, ref'] \sqsubseteq Q && \text{[Def. 4.5]} \\ &= \exists s : \text{seq Event}; a : \mathbb{P} \text{ Event} \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet [P_1 \vdash P_2 \mid P_3][s, (\alpha P \setminus a)/tt, ref'] \sqsubseteq [Q_1 \vdash Q_2 \mid Q_3] \\ & && \text{[reactive Contract Definition, Def. B.13]} \\ &= \exists s : \text{seq Event}; a : \mathbb{P} \text{ Event} \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet [P_1[s, (\alpha P \setminus a)/tt, ref'] \vdash P_2[s, (\alpha P \setminus a)/tt, ref'] \mid P_3[s, (\alpha P \setminus a)/tt, ref']] \sqsubseteq [Q_1 \vdash Q_2 \mid Q_3] \\ & && \text{[substitution distributes through reactive contracts]} \\ &= \exists s : \text{seq Event}; a : \mathbb{P} \text{ Event} \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet [\vdash P_2[s, (\alpha P \setminus a)/tt, ref'] \mid P_3[s, (\alpha P \setminus a)/tt, ref']] \sqsubseteq [\vdash Q_2 \mid Q_3] \\ & && \text{[precondition of both is } true_r \text{]} \\ &= \exists s : \text{seq Event}; a : \mathbb{P} \text{ Event} \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet [\vdash P_2[s, (\alpha P \setminus a)/tt, ref'] \mid P_3[s/tt]] \sqsubseteq [\vdash Q_2 \mid Q_3] \\ & && \text{[ref' not free in } P_3, \text{ CRF healthiness condition [30]]} \\ &= [\vdash \exists s : \text{seq Event}; a : \mathbb{P} \text{ Event} \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet P_2[s, (\alpha P \setminus a)/tt, ref'] \\ & \quad \mid \exists s : \text{seq Event} \mid tt \text{ seqapprox}(\epsilon) s \bullet P_3[s/tt]] \sqsubseteq [\vdash Q_2 \mid Q_3] \end{aligned}$$

$$\begin{aligned}
& \text{[existential quantification distributes through disjunction]} \\
= & (\exists s : \text{seq Event}; a : \mathbb{P} \text{Event} \mid tt \text{seqapprox}(\epsilon) s \wedge \alpha P \setminus \text{ref}' \text{setapprox} a \bullet \\
& P_2[s, (\alpha P \setminus a)/tt, \text{ref}'] \sqsubseteq Q_2) \\
& \wedge \\
& (\exists s : \text{seq Event} \mid tt \text{seqapprox}(\epsilon) s \bullet P_3[s/tt]) \sqsubseteq Q_3) \\
& \text{[Thm. B.4 (no precondition)]} \\
= & [Q_2 \Rightarrow \exists s : \text{seq Event}; a : \mathbb{P} \text{Event} \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox} a \bullet \\
& P_2[s, (\alpha P \setminus a)/tt, \text{ref}']] \\
& \wedge \\
& [Q_3 \Rightarrow \exists s : \text{seq Event} \mid tt \text{seqapprox}(\epsilon) s \bullet P_3[s/tt]] \\
& \text{[expand UTP refinement operator [40] (square brackets here show universal closure)]}
\end{aligned}$$

□

In short, Theorem 4.3 gives two verification conditions that distribute the conformance definition over the peri and postconditions of Q . The first condition requires the periconditions P_2 and Q_2 to be related by $\text{conf}(\epsilon)$. The second condition makes the same requirement of the postconditions P_3 and Q_3 and is simpler because postconditions do not restrict ref' .

We can discharge these verification conditions using Isabelle and the laws of UTP and establish Equation 4.1 to prove the properties of the segway. For instance, we have proved that “when P is non-zero, other PID constants are 0, and values greater than or equal to maxAngle and less than or equal to maxAngle are communicated by the event angle , the values set by $\text{setLeftMotorSpeed}()$ and $\text{setRightMotorSpeed}()$ are equal to the value communicated by angle multiplied by P ”, using the original model of the segway with the AnglePID state machine. With our proof of (4.1), we can obtain the same result for the version that uses AnglePIDANN , but we need to accept an error tolerance ϵ for the values set.

Moreover, for the particular case where the conformance that is being proved is of the form (4.1) above, the following theorem maps both conditions to set reachability conditions that can be discharged by ANN verification tools and, in particular, by Marabou. The compromise is that while we can carry out proofs for any input values in Isabelle, Marabou does not have facilities for dealing with universal quantification over real-valued sets. So, we approximate the input range with intervals and form properties based on these intervals.

Theorem 4.4.

$$\begin{aligned}
& \neg \exists x_1, \dots, x_{\text{insize}} : \text{Value} \bullet \exists y_1, \dots, y_{\text{outsize}} : \text{Value} \mid p \bullet \exists i : 1 \dots \text{outsize} \bullet \\
& \quad \{ \text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \cap \{ x : \mathbb{R} \mid |x - y_i| < \epsilon \} = \emptyset \\
\Rightarrow & [(Q_2 \setminus_{\text{peri}} \text{ANNHiddenEvs})[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.\text{layerNo}] \Rightarrow \\
& \quad \exists s : \text{seq Event}; a : \mathbb{P} \text{Event} \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
& \quad P_2[s, (\alpha P \setminus a)/tt, \text{ref}']] \\
& \wedge \\
& [(Q_3 \setminus_{\text{post}} \text{ANNHiddenEvs})[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.\text{layerNo}] \Rightarrow \\
& \quad \exists s \mid tt \text{seqapprox}(\epsilon) s \bullet P_3[s/tt]]
\end{aligned}$$

provided Q_2 is an ANN’s pericondition, Q_3 is its postcondition, P_2 is a cyclic Robochart controller’s pericondition, P_3 is its postcondition, and inp and out are sequences of events with $\#\text{inp} = \text{insize}$ and $\#\text{out} = \text{outsize}$.

Proof. Our proof of Theorem 4.4 consists of proving two separate conditions: first, that the antecedent is stronger than our pericondition; second, that the antecedent is stronger than our postcondition. Together, these provide a proof of the complete theorem.

With lemma C.1, we prove that the antecedent is stronger than our pericondition, and we establish that the antecedent is stronger than our postcondition through Lemma C.2. Due to the propositional logic laws of implication, our antecedent is stronger than the conjunction of our pericondition and postcondition. This completes our proof. \square

Theorem 4.4 states that if an output variable y_i with an error greater than ϵ does not exist when considering all possible input combinations, our verification conditions are discharged. The error, here, refers to the difference between the ANN's output value, denoted using the expression $annoutput(l, n, \langle x_1, \dots, x_{insize} \rangle)$ (Def. B.2), and our cyclic RoboChart controller's output value, captured by the variables y_i and the predicate p . We guarantee that output variables y_i have an error less than ϵ by requiring that the intersection between the set $\{annoutput(layerNo, i, \langle x_1, \dots, x_{insize} \rangle)\}$ and $\{x : \mathbb{R} \mid |x - y_i| < \epsilon\}$ is empty.

We provide an example below of the reachability conditions we obtain using Theorem 4.4, based on *AnglePID_UTP* (Sect. 4.3), capturing the semantics of *AnglePID* (Fig.3.6).

Example 4.3. *The antecedent of Theorem 4.4 for our example is the following verification condition. (Here, i takes just the value 1).*

$$\begin{aligned} \neg \exists currNewError, currDiff : Value \bullet \\ \exists currAngleOut \mid currAngleOut = P * currNewError + D * currDiff \bullet \\ \{annoutput(layerNo, 1, \langle currNewError, currDiff \rangle)\} \cap \\ \{x : \mathbb{R} \mid |x - currAngleOut| < \epsilon\} = \emptyset \end{aligned}$$

The verification condition presented above can be encoded as a set of reachability conditions if we define *Value* as a collection of sets that, together, encompass the complete range of *Value*: $\bigcup \{n : 0 \dots c \bullet [min + n \times c, min + (n + 1) \times c]\}$. Here, *min* is a minimum value, and *c* is a natural number. Given these constants, we can obtain finite conditions to prove in Marabou, as illustrated by the lemma below.

Lemma 4.1. *The antecedent of Theorem 4.4 for AnglePIDANN is as follows.*

$$\begin{aligned} \neg \exists n_1, n_2 : 0 \dots c \bullet \exists currNewError, currDiff : \mathbb{R} \bullet \\ \exists y : \mathbb{R} \mid y = annoutput(layerNo, 1, \langle currNewError, currDiff \rangle) \bullet \\ min + n_1 \times c \leq currNewError \leq min + (n_1 + 1) \times c \wedge \\ min + n_2 \times c \leq currDiff \leq min + (n_2 + 1) \times c \wedge \\ y \leq (P * (min + n_1 \times c) + D * min + n_2 \times c) - \epsilon \\ \vee \\ min + n_1 \times c \leq currNewError \leq min + (n_1 + 1) \times c \wedge \\ min + n_2 \times c \leq currDiff \leq min + (n_2 + 1) \times c \wedge \\ y \geq (P * (min + (n_1 + 1) \times c) + D * min + (n_2 + 1) \times c) + \epsilon \end{aligned}$$

Proof.

$$\neg \exists currNewError, currDiff : Value \bullet$$

$$\begin{aligned}
& \exists \text{currAngleOut} \mid \text{currAngleOut} = P * \text{currNewError} + D * \text{currDiff} \bullet \\
& \quad \{ \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \} \cap \\
& \quad \{ x : \mathbb{R} \mid |x - \text{currAngleOut}| < \epsilon \} = \emptyset \\
= & \neg \exists \text{currNewError}, \text{currDiff} : \text{Value} \bullet \quad \text{[one-point rule on currAngleOut]} \\
& \quad \{ \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \} \cap \\
& \quad \{ x : \mathbb{R} \mid |x - (P * \text{currNewError} + D * \text{currDiff})| < \epsilon \} = \emptyset \\
= & \neg \exists \text{currNewError}, \text{currDiff} : \text{Value} \bullet \quad \text{[set theory]} \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \leq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) - \epsilon \\
& \quad \vee \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \geq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) + \epsilon \\
= & \neg \exists \text{currNewError}, \text{currDiff} : \bigcup \{ n : 0 .. c \bullet [\text{min} + n \times c, \text{min} + (n + 1) \times c] \} \bullet \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \leq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) - \epsilon \\
& \quad \vee \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \geq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) + \epsilon \\
& \quad \text{[introduce set collection version of Value]} \\
= & \forall S, S1 : \{ n : 0 .. c \bullet [\text{min} + n \times c, \text{min} + (n + 1) \times c] \} \bullet \\
& \quad \neg \exists \text{currNewError} : S, \text{currDiff} : S1 \bullet \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \leq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) - \epsilon \\
& \quad \vee \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \geq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) + \epsilon \\
& \quad \text{[equivalent condition]} \\
= & \forall S, S1 : \{ n : 0 .. c \bullet [\text{min} + n \times c, \text{min} + (n + 1) \times c] \} \bullet \quad \text{[logic]} \\
& \quad \neg \exists \text{currNewError}, \text{currDiff} : \mathbb{R} \bullet \\
& \quad \text{currNewError} \in S \wedge \\
& \quad \text{currDiff} \in S1 \wedge \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \leq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) - \epsilon \\
& \quad \vee \\
& \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \geq \\
& \quad (P * \text{currNewError} + D * \text{currDiff}) + \epsilon
\end{aligned}$$

$$\begin{aligned}
&= \forall n_1, n_2 : 0 \dots c \bullet \quad \text{[set theory]} \\
&\quad \neg \exists \text{currNewError}, \text{currDiff} : \mathbb{R} \bullet \\
&\quad \quad \text{currNewError} \leq \text{min} + (n_1 + 1) \times c \wedge \\
&\quad \quad \text{currNewError} \geq \text{min} + n_1 \times c \wedge \\
&\quad \quad \text{currDiff} \leq \text{min} + (n_2 + 1) \times c \wedge \\
&\quad \quad \text{currDiff} \geq \text{min} + n_2 \times c \wedge \\
&\quad \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \leq \\
&\quad \quad (P * \text{currNewError} + D * \text{currDiff}) - \epsilon \\
&\quad \quad \vee \\
&\quad \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \geq \\
&\quad \quad (P * \text{currNewError} + D * \text{currDiff}) + \epsilon \\
&= \forall n_1, n_2 : 0 \dots c \bullet \neg \exists \text{currNewError}, \text{currDiff} : \mathbb{R} \bullet \quad \text{[propositional logic]} \\
&\quad \quad \text{min} + n_1 \times c \leq \text{currNewError} \leq \text{min} + (n_1 + 1) \times c \wedge \\
&\quad \quad \text{min} + n_2 \times c \leq \text{currDiff} \leq \text{min} + (n_2 + 1) \times c \wedge \\
&\quad \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \leq \\
&\quad \quad (P * (\text{min} + n_1 \times c) + D * \text{min} + n_2 \times c) - \epsilon \\
&\quad \quad \vee \\
&\quad \quad \text{min} + n_1 \times c \leq \text{currNewError} \leq \text{min} + (n_1 + 1) \times c \wedge \\
&\quad \quad \text{min} + n_2 \times c \leq \text{currDiff} \leq \text{min} + (n_2 + 1) \times c \wedge \\
&\quad \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \geq \\
&\quad \quad (P * (\text{min} + (n_1 + 1) \times c) + D * \text{min} + (n_2 + 1) \times c) + \epsilon \\
&= \neg \exists n_1, n_2 : 0 \dots c \bullet \exists \text{currNewError}, \text{currDiff} : \mathbb{R} \bullet \quad \text{[predicate logic]} \\
&\quad \quad \text{min} + n_1 \times c \leq \text{currNewError} \leq \text{min} + (n_1 + 1) \times c \wedge \\
&\quad \quad \text{min} + n_2 \times c \leq \text{currDiff} \leq \text{min} + (n_2 + 1) \times c \wedge \\
&\quad \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \leq \\
&\quad \quad (P * (\text{min} + n_1 \times c) + D * \text{min} + n_2 \times c) - \epsilon \\
&\quad \quad \vee \\
&\quad \quad \text{min} + n_1 \times c \leq \text{currNewError} \leq \text{min} + (n_1 + 1) \times c \wedge \\
&\quad \quad \text{min} + n_2 \times c \leq \text{currDiff} \leq \text{min} + (n_2 + 1) \times c \wedge \\
&\quad \quad \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \geq \\
&\quad \quad (P * (\text{min} + (n_1 + 1) \times c) + D * \text{min} + (n_2 + 1) \times c) + \epsilon \\
&= \neg \exists n_1, n_2 : 0 \dots c \bullet \exists \text{currNewError}, \text{currDiff} : \mathbb{R} \bullet \quad \text{[existential introduction]} \\
&\quad \quad \exists y : \mathbb{R} \mid y = \text{annoutput}(\text{layerNo}, 1, \langle \text{currNewError}, \text{currDiff} \rangle) \bullet \\
&\quad \quad \quad \text{min} + n_1 \times c \leq \text{currNewError} \leq \text{min} + (n_1 + 1) \times c \wedge \\
&\quad \quad \quad \text{min} + n_2 \times c \leq \text{currDiff} \leq \text{min} + (n_2 + 1) \times c \wedge \\
&\quad \quad \quad y \leq (P * (\text{min} + n_1 \times c) + D * \text{min} + n_2 \times c) - \epsilon \\
&\quad \quad \quad \vee \\
&\quad \quad \quad \text{min} + n_1 \times c \leq \text{currNewError} \leq \text{min} + (n_1 + 1) \times c \wedge \\
&\quad \quad \quad \text{min} + n_2 \times c \leq \text{currDiff} \leq \text{min} + (n_2 + 1) \times c \wedge \\
&\quad \quad \quad y \geq (P * (\text{min} + (n_1 + 1) \times c) + D * \text{min} + (n_2 + 1) \times c) + \epsilon
\end{aligned}$$

□

This verification condition amounts to proving $(c+1) \times (c+1)$ conditions: one for each value of n_1 and n_2 . If any of these conditions fail, Marabou produces a counterexample, where we identify the assignment of input variables x_i that causes the error. This tells us exactly where the failure is, and the ANN can be retrained using this counterexample [23].

Lemma 4.1's constraints form a hyper-rectangle in the domain of an ANN, and form a convex polytope, in inequality form, in the range (see Section 2.3.4). We can use these sets to specify properties in NNV and ERAN, as well as in Marabou. In particular, NNV and ERAN can handle non-linear activation functions such as *tanh* and *sigmoid*, as well as *ReLU*. In our approach, we choose the value of ϵ based on the needs of the system. So, even though Marabou cannot find a least upper bound for ϵ , in our work, this is not necessary.

This concludes the description and automation of our approach to proving the conformance of robotic systems involving ANN components, defined through the context of RoboChart.

4.6 Final Considerations

This chapter has shown how we can generate, and establish verification conditions concerning systems involving ANN components (defined in Chapter 3). Our approach is to replace RoboChart controllers with ANN components, so we generate a specification for the system based on these RoboChart controllers. We enable this formally by, first, in Sections 4.1 to 4.3, defining reactive contract patterns for ANN components, and those RoboChart controllers that we can replace with ANN components. Afterwards, in Sections 4.4 and 4.5, we discuss how we can generate a specification for the system using these components, and, finally, how to prove that the system conforms to this specification.

In the following, final, chapter, we discuss our conclusions with respect to the entire project.

Chapter 5

Conclusions

In this thesis, we have defined the first framework that enables the modelling, validation, and formal verification of robotic software involving neural network components. We have combined and coordinated multiple notations, tools, and techniques to enable this framework. In this chapter, we provide a summary of our contributions in Section 5.1, and then, lastly, we discuss future directions in Section 5.2.

5.1 Summary

First, in Chapter 2, Sections 2.1 to 2.5, we have described and analysed the tools and techniques that can be used to verify artificial neural networks in isolation. In particular, we place a focus on approaches that verify ANNs for control, not for recognition, as generating specifications concerning such ANNs is an open problem, and existing specifications are of debatable reliability. To conclude this chapter, in Section 2.6, we perform a novel comparison of verification tools and techniques. Through this investigation, we determine that Marabou is the most appropriate tool to integrate into our framework, as it has a dedicated property specification file, and is both precise and scalable.

We begin the description of our novel framework in Chapter 3, where we discuss how we can integrate components implemented using ANNs into the context of a robotic system. We use the RoboChart domain-specific language to reify our ideas, where we define components whose behaviour is determined by an artificial neural network. We make the following significant modelling decisions in this chapter: we model trained feed-forward, fully connected, ReLU activated neural networks. We model this specific type of ANN because it is widely used and effective for control purposes, and supports tractable verification because of its piecewise-linearity. Moreover, modelling trained ANNs allows for any of the wide, and rapidly evolving, training techniques for ANNs to be used.

We complete our framework in Chapter 4 by defining a verification technique specific to robotic systems involving ANN components as presented in Chapter 3. This technique involves replacing robotic controllers with ANN components. Through theories presented in this chapter, we define a method to show that a robotic system involving ANN components conforms to the behaviour defined by a traditional control algorithm. The limits of this technique are that the system must contain a cyclic controller, that is, a stateful, memoryless controller that receives all input events before engaging in output events, and does not

terminate. Further, our ANN components are limited in form, as discussed, but not in size, shape, or training method.

We address our research objectives, as defined in Section 1.2, throughout Chapters 3 and 4. We address our first objective in Section 3.2 through extending RoboChart’s metamodel and defining well-formedness conditions to support this model. Sections 3.3 and 3.4 fulfil our second objective by extending the semantics of RoboChart to support these components. Section 3.5, on the other hand, implements these semantics in JCSP, enabling validation activities to help establish these objectives.

Our third objective states that we should define a technique that can verify the overall system model, which enables the verification of system-level specifications. We have addressed this objective in two stages. First, in Section 4.4, we define and formalise conformance, a property that we can establish about systems involving ANNs. Then in Section 4.5, we define a technique to establish conformance for system models involving ANN components.

Building upon our third objective, our fourth objective states that the verification technique should be automated, taking advantage of existing support for RoboChart. We have defined two automation approaches in Section 4.5. The first takes advantage of existing support for RoboChart, namely, the use of the Isabelle proof assistant, and the reactive contract UTP theory, implemented in the Isabelle/UTP tool. With Theorem 4.3, we define conditions that we can discharge to Isabelle/UTP, constituting the first automation of our verification technique. Through Theorem 4.4, we present verification conditions that can be discharged by SMT solver techniques, enabling automation via these techniques. Finally, through Lemma 4.1, we translate Theorem 4.4 to a format that can be directly interpreted by Marabou, constituting our second automation method, and fulfilling our fourth objective.

This concludes the summary of the contributions we have made in this work, next, we discuss the future directions.

5.2 Future Work

The future work suggested by this framework can be broadly summarised into three potential directions. First, directly extending this framework to accommodate more types of neural network, and linking more tools to address the properties we create in this work. Second, to generate more techniques to address more types of system-level properties, particularly involving ANNs for perception. Third, to explore dynamic verification of AI systems, how we can verify their learning as they explore and adapt to new environments, which could involve exploring different types of refinement specific to learning components.

An immediate goal is to generalise the ANN components. Our metamodel and semantics can easily accommodate several activation functions and can be extended to cater for convolutional neural networks with minor changes. Various tools and techniques remain applicable because the layer function is piecewise linear. Recurrent neural networks require more changes; fewer techniques and tools are available, although some are emerging [44].

We can extend the tools we use in our framework in numerous ways. For example, we can use ERAN [88] to enable verification via abstract interpretation, which can provide support for a wider variety of ANN types. As another alternative, we can use NNV [96] to carry out analysis via reachability analysis, which we can use to propagate output spaces back to input spaces to refine the property. Moreover, this suggests the possibility of defining a toolchain

of ANN-specific tools instead of discharging our proof to a single tool. It would require techniques to reduce the search space and discharge properties using complete techniques. It would allow us to verify more extensive and complex ANNs.

A further possibility is to integrate the work by Brucker and Stell in [17], where they also use Isabelle/HOL, but utilising a different approach to that taken in our work. Their work describes an approach to use Isabelle/HOL to verify the properties of feed-forward ANNs, in isolation. Using this approach instead of Marabou to automate the proofs generated using our framework is feasible, and avoid input and output value restrictions.

Our second direction is generating a verification approach for perception-based ANNs. Developing meaningful specifications for such components is challenging, but there is a growing body of work to address this [42]. The goal of this approach would be to guarantee against certain behaviours and situations instead of a guaranteed error bound.

Finally, we consider approaches which can support the dynamic verification of ANNs, particularly, new forms of refinement concerning learning components. The work by Dupont et al. [24] describes an approach based on refinement to deal with robotic systems involving continuous behaviours. Their approach enables two methods: upwards approximation, where an approximated system is refined to an exact system; and downwards approximation, where an exact system is refined to an approximated version of that system. Our work is a type of upwards approximation: we refine an approximate system into an exact system. Exploring downwards approximation, and approximate refinement with respect to AI components, however, is an interesting potential future direction.

This work represents progress towards the colossal task of unifying the languages, tools, and techniques required to enable safe, reliable, and transparent RAAI systems. Our work also suggests a new perspective of how to use AI, from an angle not concerned with the details of its creation, but with its form and utility, and contributes to the discussion of how we can use intelligent machines in the future.

Appendix A

Full Meta-model

We describe the following classes as Ecore classes formatted by the tool OCLinEcore. A summary of ECore is available ¹, and a summary of OCLinEcore is available ².

Our additions:

```
1  abstract class ANN
2  {
3      property annparameters : ANNParameters[1] { composes };
4  }
5  class ANNParameters
6  {
7      property insize : IntegerExp[1] { composes };
8      property outsize : IntegerExp[1] { composes };
9      property layerstructure : SeqExp[1] { composes };
10     property weights : SeqExp[?] { composes };
11     property biases : SeqExp[?] { composes };
12     property filename : StringExp[?] { composes };
13     attribute activationfunction : ActivationFunction[1];
14 }
15 enum ActivationFunction { serializable }
16 {
17     literal RELU;
18     literal LINEAR;
19     literal NOTSPECIFIED;
20 }
21 abstract class GeneralController extends Controller;
22
23 class ANNController extends ANN,GeneralController
24 {
25     property events : Event[*] { composes };
26 }
27
28 abstract class GeneralOperation extends Operation, OperationSig;
29
```

¹<https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html>

²<https://help.eclipse.org/2021-06/index.jsp?topic=\\%2Forg.eclipse.oclin.doc\\%2Fhelp\\%2FOCLinEcore.html>

```

30     class ANNOperation extends ANN,GeneralOperation
31     {
32         property rInterface : Interface[1];
33
34     }

```

Our modifications to the existing metamodel:

```

1     class RCPackage extends BasicPackage
2     {
3         property interfaces : Interface[*] { composes };
4         property robots : RoboticPlatformDef[*] { composes };
5         property types : TypeDecl[*] { composes };
6         property machines : StateMachineDef[*] { composes };
7         property controllers : GeneralController[*] { composes };
8         property modules : RModule[*] { composes };
9         property operations : GeneralOperation[*] { composes };
10        property functions : Function[*] { composes };
11    }
12    class OperationDef extends StateMachineBody,GeneralOperation;
13    class OperationRef extends Operation,Reference
14    {
15        property ref : GeneralOperation[1];
16    }
17    class ControllerDef extends Context,GeneralController
18    {
19        property machines : StateMachine[*] { composes };
20        property lOperations : Operation[*] { composes };
21        property connections : Connection[*] { composes };
22    }
23    class ControllerRef extends Controller
24    {
25        property ref : GeneralController[1];
26    }

```

Appendix B

UTP Reactive Contract Laws

Here, we present formal material that augments the material presented in Chapter 4. In Section B.1, we present laws and definitions we have defined to support our work, and in Section B.2, we present existing laws that we rely on in our work.

B.1 Supporting Laws

In the function *layeroutput*, the trace contribution *tt* appears free (through the reference to *input*). We would like to reason about the ANN contracts without referring to the complete trace of the process. To this end, we provide a function *nodeoutput* which defines the output event engaged of a given node, without referring to *tt*. Its parameters are *l*, the layer index, *n*, the node's index, and *in*, a numeric sequence of type \mathbb{A} , that captures the input of the ANN component.

Definition B.1 (Node Output Function).

$$\frac{\text{nodeoutput} : \mathbb{Z} \times \mathbb{Z} \times \text{seq } \mathbb{A} \rightarrow \text{Event}}{\forall l : 1 \dots \text{layerNo}; \text{in} : \text{seq } \mathbb{A} \bullet \forall n : 1 \dots \text{layerSize}(l) \bullet \text{nodeoutput}(l, n, \text{in}) = \text{layerRes}.l.n.\text{annoutput}(l, n, \text{in})}$$

We define *nodeoutput* as the single event *layerRes.l.n.annoutput(l, n, in)*. Here, the expression *annoutput(l, n, in)* denotes the value communicated by a node indexed by *n* of layer *l*. For example, if we evaluate *layeroutput(1, 2, ⟨1, 1⟩)*, we get *layerRes.1.2.annoutput(1, 2, ⟨1, 1⟩)*.

We note that *layercalc* calculates the complete event sequence of a layer, while *annoutput* calculates just the value communicated by the output event engaged in by a single node. To illustrate, *annoutput(l, n, in)* is equivalent to *value(last(layercalc(l, n, pl)))*.

We define the function *annoutput* formally below. It takes identical input parameters as *nodeoutput*, but *annoutput* evaluates to a single numeric value instead of an *Event*.

Definition B.2 (ANN Output Function).

$$\begin{array}{l}
\text{annoutput} : \mathbb{Z} \times \mathbb{Z} \times \text{seq } \mathbb{A} \rightarrow \mathbb{A} \\
\forall l : 0 \dots \text{layerNo}; \text{in} : \text{seq } \mathbb{A} \bullet \forall n : 1 \dots \text{layerSize}(l) \bullet \\
\text{annoutput}(0, n, \text{in}) = \text{in } n \wedge \\
\text{annoutput}(l, n, \text{in}) = \text{relu}(\text{dotprod}(\{pn : 1 \dots \text{layerSize}(l-1) \bullet \\
\qquad (pn, \text{annoutput}(l-1, pn, \text{in}))\}, \\
\qquad \text{weights } l \ n) \\
\qquad + \\
\qquad \text{biases } l \ n)
\end{array}$$

We define *annoutput* using two equations. The first is for the input layer, that returns the index of the input sequence *in* according to the node index *n*. The second captures the result of any other node, given by the function *relu* applied to the sum of: the weighted output of the previous layer, and the *biases l n* scalar. Here, the previous layer's results are captured by the sequence $\{pn : 1 \dots \text{layerSize}(l-1) \bullet (pn, \text{annoutput}(l-1, pn, \text{in}))\}$, that contains the results, given by *annoutput*, of all nodes indexed by *pn* of the previous layer $l-1$. We then capture the weighted sum of this sequence via the function *dotprod*: given the previous layer's results, as above, and the sequence *weights l n*.

As an example, consider a node in an ANN with the following parameters: layer index 1, node index 1, weight vector $\langle 2, -1 \rangle$, bias scalar 1, and with two input nodes. In this context, *annoutput*(1, 1, $\langle 1, 1 \rangle$) would evaluate to simply 2.

Next, we define a lemma that formalises the relationship between *layeroutput* and *nodeoutput*, and the assumption we make about the variable *input*, that was discussed earlier. That is, the sequence *input* contains all input events occurring in order, and the value communicated by each event is from the set *Value*. We capture this, below, by introducing separate bound variables $x_1, \dots, x_{\text{insize}}$ for each value communicated by an input event.

Lemma B.1. *For all layer indices $l \in 1 \dots \text{layerNo}$ and node indices $n \in 1 \dots \text{layerSize}(l)$, and for a given input sequence $\text{input} \in \text{seq } \text{Event}$ the following holds:*

$$\begin{array}{l}
\exists x_1, \dots, x_{\text{insize}} : \text{Value} \bullet \\
\text{layeroutput}(l, n) = \\
\quad \wedge / \text{in} : 1 \dots \text{insize} \bullet \langle \text{layerRes}.0.\text{in}.x_{\text{in}} \rangle \\
\quad \wedge / p_l : 1 \dots l-1 \bullet \\
\quad \quad \wedge / p_n : 1 \dots \text{layerSize}(p_l) \bullet \\
\quad \quad \quad \langle \text{nodeoutput}(p_l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
\quad \wedge / p_n : 1 \dots n \bullet \langle \text{nodeoutput}(l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle
\end{array}$$

This lemma states that the function *layeroutput*(*l*, *n*) is equal to the concatenation of three distributed concatenations (denoted by $\wedge /$). The first represents the *input* sequence, and states that all input events occur in order, and are of the form *layerRes*.0.*in*.*x_{in}*. The second captures the result of all previous layers, layers before the layer *l*, where we represent the previous layers index by *p_l*, and the previous node's index by *p_n*. Here, we define the events with the expression *nodeoutput*(*p_l*, *p_n*, $\langle x_1, \dots, x_{\text{insize}} \rangle$), where $\langle x_1, \dots, x_{\text{insize}} \rangle$ is the sequence containing the values communicated by all input events. Lastly, our third

distributed concatenation captures all nodes up to and including the node n , on the layer l .

Next, we discuss functions that enable our definitions above.

First, we define our *relu* function below. If $x < 0$, then the function evaluates to 0, otherwise x . We define ReLU as a function on \mathbb{A} , which our ANN uses.

Definition B.3 (ReLU).

$$\begin{array}{|l} \hline \text{relu} : \mathbb{A} \rightarrow \mathbb{A} \\ \hline \forall x : \text{Value} \bullet \\ \quad (x < 0 \Rightarrow (x, 0) \in \text{relu}) \wedge \\ \quad (x \geq 0 \Rightarrow (x, x) \in \text{relu}) \end{array}$$

We define our sequence dot product function below. It takes two sequences as arguments, both containing numeric values and of identical size.

Definition B.4 (Sequence Dot Product Function).

$$\begin{array}{|l} \hline \text{dotprod} : (\text{seq } \mathbb{A} \times \text{seq } \mathbb{A}) \rightarrow \mathbb{A} \\ \hline \forall s1, s2 : \text{seq } \mathbb{A} \mid \#s1 = \#s2 \bullet \\ \quad \text{dotprod}(\langle \rangle, \langle \rangle) = 0 \\ \quad \wedge \\ \quad \text{dotprod}(s1, s2) = \\ \quad \quad (\text{head}(s1) * \text{head}(s2)) + \\ \quad \quad \text{dotprod}(\text{tail}(s1), \text{tail}(s2)) \end{array}$$

For example, $\text{dotprod}(\langle 1, 2, 3 \rangle, \langle 2, 3, 4 \rangle)$ is the scalar value 20 ($2 + 6 + 12$). The output value of a single ANN node can be expressed as the dot product of the two vectors representing the weights and the previous layer's output. This dot product is the sum of all elements multiplied element-wise, so, *dotprod* captures the vector dot product operator.

Finally, we define the *lastn* function, which returns a sequence's last n elements. This function takes a sequence of a generic type X and an integer quantity of elements to return, and delivers a sequence of type X as output.

Definition B.5 (Last N Elements Function).

$$\begin{array}{|l} \hline \text{lastn} : (\text{seq } X \times \mathbb{Z}) \rightarrow \text{seq } X \\ \hline \forall s : \text{seq } X; i : \mathbb{N} \mid i \leq \#s \bullet \text{lastn}(s, i) = (\#s - i \dots \#s) \upharpoonright s \end{array}$$

For $\text{lastn}(s, i)$ to be well-defined, we require that i is less than or equal to the size of the sequence s . This is because we cannot extract more elements from s than exist in s .

We define *lastn* as taking all elements from s indexed from $\#s - i$ to $\#s$, the last section of the sequence. For example, $\text{lastn}(\langle 1, 2, 3, 4 \rangle, 2)$ evaluates to the sequence $\langle 3, 4 \rangle$.

The function *dropseq* takes a sequence of *layerRes* events and returns a sequence of arithmetic values corresponding to each *layerRes* event's value, denoted by v in *layerRes.l.n.v*.

Definition B.6 (Event Drop Function).

$$\begin{array}{|l} \hline \text{dropseq} : \text{seq } \mathit{Event} \rightarrow \text{seq } \mathbb{A} \\ \hline \forall s : \text{seq } \mathit{Event} \bullet \\ \quad \text{dropseq}(\langle \rangle) = \langle \rangle \\ \quad \wedge \\ \quad \text{dropseq}(s) = \text{dropseq}(\text{tail}(s)) \hat{\wedge} \langle \text{head}(s).4 \rangle \end{array}$$

In the base case, given the empty sequence, we return the empty sequence. In the recursive case, we take the head of the sequence and return the 4th component of the pair given by the *Event*. Here, we consider *Event* to capture *layerRes* events.

For example, if we evaluate *dropseq* given the sequence below:

$$\langle \text{layerRes}.0.1.1, \text{layerRes}.0.2.1, \text{layerRes}.1.1.6, \text{layerRes}.1.2.6 \rangle$$

We obtain the sequence $\langle 1, 1, 6, 6 \rangle$.

Lemma B.2. *The pericondition of the reactive contract capturing $\mathit{SANN_Layers}$ is the pericondition of $\mathit{GeneralANNContract}$: $\mathit{SANN_Layers}_2 = \mathit{GeneralANNContract}_2$.*

Proof.

$$\begin{aligned} & (\mathit{SANN_Layers})_2 \\ = & (\mathit{SANN_InputLayer})_2 \vee && [\text{Theorem B.1, law (1).}] \\ & (\mathit{SANN_InputLayer})_3; \\ & (; l : 1 .. \text{layerNo} \bullet \mathit{SANN_Layer}(l))_2 \\ = & (; i : 1 .. \text{insize} \bullet (\mathit{SANN_InputNode}))_2 \vee && [\text{unfolding } \mathit{SANN_InputLayer}] \\ & (\mathit{SANN_InputLayer})_3; \\ & (; l : 1 .. \text{layerNo} \bullet \mathit{SANN_Layer}(l))_2 \\ = & (; i : 1 .. \text{insize} \bullet \square v : \mathit{Value} \bullet (\mathit{Do}(\text{layerRes}.0.i.v); \mathit{Skip}))_2 \vee \\ & (\mathit{SANN_InputLayer})_3; \\ & (; l : 1 .. \text{layerNo} \bullet \mathit{SANN_Layer}(l))_2 \\ &&& [\text{unfolding } \mathit{SANN_InputNode} \text{ and converting to UTP notation}] \\ = & (; i : 1 .. \text{insize} \bullet \square v : \mathit{Value} \bullet \mathit{Do}(\text{layerRes}.0.i.v))_2 \vee \\ & (\mathit{SANN_InputLayer})_3; \\ & (; l : 1 .. \text{layerNo} \bullet \mathit{SANN_Layer}(l))_2 \\ &&& [\mathit{Skip} \text{ is a right unit of } ; \text{ in the CSP theory}] \\ = & ([\text{true}_r \vdash &&& [\text{Lemma B.8}] \\ & \exists s : \text{seq } \mathit{Event} \mid \#s = \text{insize} \wedge (\forall i : 1 .. \text{insize} \bullet \exists v : \mathit{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\ & \quad \text{tt prefix } s \wedge \#tt < \#s \wedge \text{wait}' \wedge \text{ref}' \subseteq \{s(\#tt + 1)\} \vee \text{tt} = s \wedge \neg \text{wait}'])_2 \\ & \vee \\ & (\mathit{SANN_InputLayer})_3; \\ & (; l : 1 .. \text{layerNo} \bullet \mathit{SANN_Layer}(l))_2 \end{aligned}$$

$$\begin{aligned}
&= ([\text{true}_r \vdash \quad [tt = \text{input when } \#tt \leq \text{insize}] \\
&\quad \exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \text{input prefix } s \wedge \#input < \#s \wedge \text{wait}' \wedge \text{ref}' \subseteq \{s(\#input + 1)\} \wedge tt = \text{input} \vee \\
&\quad \quad tt = s \wedge tt = \text{input} \wedge \neg \text{wait}']_2 \\
&\quad \vee \\
&\quad (\text{SANN_InputLayer})_3; \\
&\quad (; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&= ([\text{true}_r \vdash \quad [\{s(\#input + 1)\} \subseteq \{ | \text{layerRes}.0.(\#input + 1) | \}] \\
&\quad \exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \text{input prefix } s \wedge \#input < \#s \wedge \text{wait}' \wedge \text{ref}' \subseteq \{ | \text{layerRes}.0.(\#input + 1) | \} \\
&\quad \quad \wedge tt = \text{input} \\
&\quad \quad \vee \\
&\quad \quad tt = s \wedge tt = \text{input} \wedge \neg \text{wait}']_2 \\
&\quad \vee \\
&\quad (\text{SANN_InputLayer})_3; \\
&\quad (; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&= ([\text{true}_r \vdash \quad [\#s = \text{insize}] \\
&\quad \exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \text{input prefix } s \wedge \#input < \text{insize} \wedge \text{wait}' \wedge \text{ref}' \subseteq \{ | \text{layerRes}.0.(\#input + 1) | \} \\
&\quad \quad \wedge tt = \text{input} \\
&\quad \quad \vee \\
&\quad \quad tt = s \wedge tt = \text{input} \wedge \neg \text{wait}']_2 \\
&\quad \vee \\
&\quad (\text{SANN_InputLayer})_3; \\
&\quad (; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&= ([\text{true}_r \vdash \quad [\exists \text{ distributes through } \vee] \\
&\quad \exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \text{input prefix } s \wedge \#input < \text{insize} \wedge \text{wait}' \wedge \text{ref}' \subseteq \{ | \text{layerRes}.0.(\#input + 1) | \} \\
&\quad \quad \wedge tt = \text{input} \\
&\quad \quad \vee \\
&\quad \quad \exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \quad tt = s \wedge tt = \text{input} \wedge \neg \text{wait}']_2 \\
&\quad \vee \\
&\quad (\text{SANN_InputLayer})_3; \\
&\quad (; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2
\end{aligned}$$

$$\begin{aligned}
&= ([\text{true}_r \vdash \hspace{15em} \text{[predicate calculus]} \\
&\quad (\exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \text{input prefix } s) \\
&\quad \wedge \\
&\quad \#input < \text{insize} \wedge \text{wait}' \wedge \text{ref}' \subseteq \{ \mid \text{layerRes}.0.(\#input + 1) \mid \} \wedge \text{tt} = \text{input} \\
&\quad \vee \\
&\quad \exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \text{tt} = s \wedge \text{tt} = \text{input} \wedge \neg \text{wait}'])_2 \\
&\vee \\
&(\text{SANN_InputLayer})_3 ; \\
&(\ ; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&= ([\text{true}_r \vdash \hspace{15em} \text{[definition of input and propositional calculus]} \\
&\quad \#input < \text{insize} \wedge \text{wait}' \wedge \text{ref}' \subseteq \{ \mid \text{layerRes}.0.(\#input + 1) \mid \} \wedge \text{tt} = \text{input} \\
&\quad \vee \\
&\quad \exists s : \text{seq Event} \mid \#s = \text{insize} \wedge (\forall i : 1 \dots \text{insize} \bullet \exists v : \text{Value} \bullet s\ i = \text{layerRes}.0.i.v) \bullet \\
&\quad \quad \text{tt} = s \wedge \text{tt} = \text{input} \wedge \neg \text{wait}'])_2 \\
&\vee \\
&(\text{SANN_InputLayer})_3 ; \\
&(\ ; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&= ([\text{true}_r \vdash \hspace{15em} \text{[one-point rule and definition of input]} \\
&\quad \#input < \text{insize} \wedge \text{wait}' \wedge \text{ref}' \subseteq \{ \mid \text{layerRes}.0.(\#input + 1) \mid \} \wedge \text{tt} = \text{input} \\
&\quad \vee \\
&\quad \text{tt} = \text{input} \wedge \neg \text{wait}'])_2 \\
&\vee \\
&(\text{SANN_InputLayer})_3 ; \\
&(\ ; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&= ([\text{true}_r \vdash \#input < \text{insize} \wedge \mathcal{E}[\text{input}, \{ \mid \text{layerRes}.0.(\#input + 1) \mid \}] \\
&\quad \mid \#input = \text{insize} \wedge \Phi[\text{input}]])_2 \\
&\vee \\
&(\text{SANN_InputLayer})_3 ; \\
&(\ ; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&\hspace{15em} \text{[Def. B.14, Def.B.13, and input definition]} \\
&= (\#input < \text{insize} \wedge \mathcal{E}[\text{input}, \{ \mid \text{layerRes}.0.(\#input + 1) \mid \}]) \\
&\vee \\
&(\text{SANN_InputLayer})_3 ; \\
&(\ ; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&\hspace{15em} \text{[pericondition extraction (Theorem B.3)}] \\
&= (\#input < \text{insize} \wedge \mathcal{E}[\text{input}, \{ \mid \text{layerRes}.0.(\#input + 1) \mid \}]) \\
&\vee \\
&([\text{true}_r \vdash \#input < \text{insize} \wedge \mathcal{E}[\text{input}, \{ \mid \text{layerRes}.0.(\#input + 1) \mid \}] \\
&\quad \mid \#input = \text{insize} \wedge \Phi[\text{input}]])_3 ; \\
&(\ ; l : 1 \dots \text{layerNo} \bullet \text{SANN_Layer}(l))_2 \\
&\hspace{15em} \text{[argument similar to the above]}
\end{aligned}$$

$$\begin{aligned}
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad (\#input = insize \wedge \Phi[input]); \\
&\quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_2 \\
&\hspace{15em} [\text{postcondition extraction (Theorem B.3)}] \\
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad (\#input = insize \wedge \Phi[input]); \\
&\quad (; l : 1 .. layerNo \bullet ; n : 1 .. layerSize(l) \bullet \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad Do(layerRes.l.n.annoutput(l, n, in)))_2 \\
&\hspace{10em} [\text{unfolding } SANN_Layer(l) \text{ and } SANN_Node, \text{ and } Skip \text{ is a right unit of ;}] \\
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad (\#input = insize \wedge \Phi[input]); \\
&\quad (; l : 1 .. layerNo \bullet ; n : 1 .. layerSize(l) \bullet \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad Do(nodeoutput(l, n, in)))_2 \\
&\hspace{15em} [\text{equivalent to } nodeoutput(l, n, in), \text{ Def. B.1}] \\
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad (\#input = insize \wedge \Phi[input]); \\
&\quad ([true_r \\
&\quad \quad \vdash \\
&\quad \quad \exists l : 1 .. layerNo; n : 1 .. layerSize(l) \bullet \\
&\quad \quad \quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \quad \mathcal{E}[\bigwedge / pl : 1 .. (l - 1); pn : 1 .. layerSize(pl) \bullet \langle nodeoutput(pl, pn, in) \rangle \\
&\quad \quad \quad \quad \bigwedge / cn : 1 .. (n - 1) \bullet \langle nodeoutput(l, cn, in) \rangle, \{ nodeoutput(l, n, in) \}] \\
&\quad \quad \quad | \\
&\quad \quad \quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \quad \Phi[\bigwedge / l : 1 .. layerNo; n : 1 .. layerSize(l) \bullet \langle nodeoutput(l, n, in) \rangle])_2 \\
&\hspace{15em} [\text{Lemma B.9, here } E \text{ is } layerSize] \\
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad (\#input = insize \wedge \Phi[input]); \\
&\quad \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
&\quad \quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \mathcal{E}[\bigwedge / pl : 1 .. (l - 1); pn : 1 .. layerSize(pl) \bullet \langle nodeoutput(pl, pn, in) \rangle \\
&\quad \quad \quad \bigwedge / cn : 1 .. (n - 1) \bullet \langle nodeoutput(l, cn, in) \rangle, \{ nodeoutput(l, n, in) \}] \\
&\hspace{15em} [\text{pericondition extraction, Theorem B.3}]
\end{aligned}$$

$$\begin{aligned}
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad \#input = insize \wedge \\
&\quad \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
&\quad \quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \quad \mathcal{E}[\wedge / in \wedge \\
&\quad \quad \quad \quad \wedge / pl : 1 .. (l - 1); pn : 1 .. layerSize(pl) \bullet \langle nodeoutput(pl, pn, in) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / cn : 1 .. (n - 1) \bullet \langle nodeoutput(l, cn, in) \rangle, \{nodeoutput(l, n, in)\}] \\
&\hspace{15em} [\text{sequential composition, Theorem B.1}] \\
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \hspace{10em} [input \text{ assumption}] \\
&\quad \vee \\
&\quad \#input = insize \wedge \\
&\quad \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
&\quad \quad \forall in : seq Value; x_1, \dots, x_{insize} : Value \mid in = \#insize \bullet \\
&\quad \quad \quad \mathcal{E}[\wedge / n : 1 .. insize \bullet \langle layerRes.0.n.x_n \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / pl : 1 .. (l - 1); pn : 1 .. layerSize(pl) \bullet \langle nodeoutput(pl, pn, in) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / cn : 1 .. (n - 1) \bullet \langle nodeoutput(l, cn, in) \rangle, \\
&\quad \quad \quad \quad \{nodeoutput(l, n, in)\}] \\
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad \#input = insize \wedge \\
&\quad \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
&\quad \quad \forall x_1, \dots, x_{insize} : Value \bullet \\
&\quad \quad \quad \mathcal{E}[\wedge / n : 1 .. insize \bullet \langle layerRes.0.n.x_n \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / pl : 1 .. (l - 1); pn : 1 .. layerSize(pl) \bullet \\
&\quad \quad \quad \quad \quad \langle nodeoutput(pl, pn, \wedge / in : 1 .. insize \bullet \langle x_{in} \rangle) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / cn : 1 .. (n - 1) \bullet \langle nodeoutput(l, cn, \wedge / in : 1 .. insize \bullet \langle x_{in} \rangle) \rangle, \\
&\quad \quad \quad \quad \{nodeoutput(l, n, \wedge / in : 1 .. insize \bullet \langle x_{in} \rangle)\}] \\
&\hspace{15em} [in \text{ must contain all input variables } x] \\
&= (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
&\quad \vee \\
&\quad \#input = insize \wedge \\
&\quad \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
&\quad \quad \forall x_1, \dots, x_{insize} : Value \bullet \\
&\quad \quad \quad \mathcal{E}[\wedge / n : 1 .. insize \bullet \langle layerRes.0.n.x_n \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / pl : 1 .. (l - 1); pn : 1 .. layerSize(pl) \bullet \\
&\quad \quad \quad \quad \quad \langle nodeoutput(pl, pn, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / cn : 1 .. (n - 1) \bullet \langle nodeoutput(l, cn, \langle x_1, \dots, x_{insize} \rangle) \rangle, \\
&\quad \quad \quad \quad \{nodeoutput(l, n, \langle x_1, \dots, x_{insize} \rangle)\}]
\end{aligned}$$

$$\begin{aligned}
& \text{[introduce shorthand]} \\
& = (\#input \leq insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) | \}]) \\
& \quad \vee \\
& \quad \#input = insize \wedge \\
& \quad \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
& \quad \quad \mathcal{E}[front \circ layeroutput(l, n), \{last \circ layeroutput(l, n)\}] \\
& \quad \quad \text{[Lemma B.1, and } nodeoutput = last \circ layeroutput \text{ in this context]} \\
& = GeneralANNContract_2 \quad \text{[Def. 4.1]}
\end{aligned}$$

□

Lemma B.3. *The postcondition of the reactive contract capturing SANN_Layers is the postcondition of GeneralANNContract: $SANN_Layers_3 = GeneralANNContract_3$.*

Proof.

$$\begin{aligned}
& (SANN_Layers)_3 \\
& = (SANN_InputLayer)_3; \\
& \quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_3 \\
& \quad \quad \text{[unfolding } SANN_Layers, \text{ law (1)-Theorem B.1, definition of } (P)_3] \\
& = (; i : 1 .. insize \bullet (SANN_InputNode))_3; \quad \text{[unfolding } SANN_InputLayer] \\
& \quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_3 \\
& = (; i : 1 .. insize \bullet \square v : Value \bullet (Do(layerRes.0.i.v); Skip))_3; \\
& \quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_3 \\
& \quad \quad \text{[unfolding } SANN_InputNode \text{ and converting to UTP notation]} \\
& = (; i : 1 .. insize \bullet \square v : Value \bullet Do(layerRes.0.(Skip)))_3; \quad \text{[Skip is a right unit of ; in the CSP theory]} \\
& \quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_3 \\
& = ([true_r \vdash \quad \text{[Lemma B.8]} \\
& \quad \exists s : seq Event \mid \#s = insize \wedge (\forall i : 1 .. insize \bullet \exists v : Value \bullet s i = layerRes.0.i.v) \bullet \\
& \quad \quad tt \text{ prefix } s \wedge \#tt < \#s \wedge wait' \wedge ref' \subseteq \{s(\#tt + 1)\} \vee tt = s \wedge \neg wait'])_3; \\
& \quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_3 \\
& = ([true_r \vdash \quad \text{[} tt = input \text{ when } \#tt \leq insize] \\
& \quad \exists s : seq Event \mid \#s = insize \wedge (\forall i : 1 .. insize \bullet \exists v : Value \bullet s i = layerRes.0.i.v) \bullet \\
& \quad \quad input \text{ prefix } s \wedge \#input < \#s \wedge wait' \wedge ref' \subseteq \{s(\#input + 1)\} \wedge tt = input \vee \\
& \quad \quad tt = s \wedge tt = input \wedge \neg wait'])_3; \\
& \quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_3 \\
& = ([true_r \vdash \quad \text{[} \{s(\#input + 1)\} \subseteq \{| layerRes.0.(\#input + 1) | \} \\
& \quad \exists s : seq Event \mid \#s = insize \wedge (\forall i : 1 .. insize \bullet \exists v : Value \bullet s i = layerRes.0.i.v) \bullet \\
& \quad \quad input \text{ prefix } s \wedge \#input < \#s \wedge wait' \wedge ref' \subseteq \{| layerRes.0.(\#input + 1) | \} \\
& \quad \quad \wedge tt = input \\
& \quad \quad \vee \\
& \quad \quad tt = s \wedge tt = input \wedge \neg wait'])_3; \\
& \quad (; l : 1 .. layerNo \bullet SANN_Layer(l))_3
\end{aligned}$$

$$\begin{aligned}
&= (\#input = insize \wedge \Phi[input]); \\
&\quad (; l : 1 .. layerNo \bullet ; n : 1 .. layerSize(l) \bullet \forall in : seq Event \mid \#in = insize \bullet \\
&\quad \quad Do(nodeoutput(l, n, in)))_3 \\
&\hspace{20em} [\text{equivalent to } nodeoutput(l, n, in) \text{ by Def.B.1}] \\
&= (\#input = insize \wedge \Phi[input]); \\
&\quad ([true_r \\
&\quad \quad \vdash \\
&\quad \quad \exists l : 1 .. layerNo \bullet \exists n : 1 .. layerSize(l) \bullet \\
&\quad \quad \quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \quad \mathcal{E}[\wedge / pl : 1 .. (l-1); pn : 1 .. layerSize(pl) \bullet \langle nodeoutput(pl, pn, in) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \wedge / cn : 1 .. (n-1) \bullet \langle nodeoutput(l, cn, in) \rangle, \{nodeoutput(l, n, in)\}] \\
&\quad \quad \quad \mid \\
&\quad \quad \quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \quad \Phi[\wedge / l : 1 .. layerNo; n : 1 .. layerSize(l) \bullet \langle nodeoutput(l, n, in) \rangle])_3 \\
&\hspace{20em} [\text{Lemma B.9, here, } E \text{ is } layerSize] \\
&= (\#input = insize \wedge \Phi[input]); \hspace{10em} [\text{postcondition extraction, Theorem B.3}] \\
&\quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \Phi[\wedge / l : 1 .. layerNo; n : 1 .. layerSize(l) \bullet \langle nodeoutput(l, n, in) \rangle] \\
&= \#input = insize \wedge \hspace{10em} [\text{sequential composition, Theorem B.1}] \\
&\quad \forall in : seq Value \mid \#in = insize \bullet \\
&\quad \quad \Phi[\overset{\wedge}{input} \\
&\quad \quad \quad \wedge / l : 1 .. layerNo \bullet \wedge / n : 1 .. layerSize(l) \bullet \langle nodeoutput(l, n, in) \rangle] \\
&= \#input = insize \wedge \hspace{10em} [input \text{ assumption}] \\
&\quad \forall in : seq Value; x_1, \dots, x_{insize} : Value \mid \#in = insize \bullet \\
&\quad \quad \Phi[\overset{\wedge}{\wedge / n : 1 .. insize \bullet \langle layerRes.0.n.x_n \rangle} \\
&\quad \quad \quad \wedge / l : 1 .. layerNo \bullet \wedge / n : 1 .. layerSize(l) \bullet \langle nodeoutput(l, n, in) \rangle] \\
&= \#input = insize \wedge \\
&\quad \forall x_1, \dots, x_{insize} : Value \bullet \\
&\quad \quad \Phi[\overset{\wedge}{\wedge / n : 1 .. insize \bullet \langle layerRes.0.n.x_n \rangle} \\
&\quad \quad \quad \wedge / l : 1 .. layerNo \bullet \wedge / n : 1 .. layerSize(l) \bullet \\
&\quad \quad \quad \quad \langle nodeoutput(l, n, \wedge / n : 1 .. insize \bullet \langle x_n \rangle) \rangle] \\
&\hspace{20em} [in \text{ must contain all input variables } x] \\
&= \#input = insize \wedge \Phi[layeroutput(layerNo, layerSize(l))] \\
&\hspace{20em} [\text{equivalent to } layeroutput(layerNo, layerSize(l)) \text{ by Lemma B.1}] \\
&= GeneralANNContract_3 \hspace{15em} [\text{Def. 4.1}]
\end{aligned}$$

□

Lemma B.4 (*seqapprox*(ϵ) reflexivity).

$$\forall s : seq Event \mid \text{ran } s \subseteq (I \cup O) \bullet s \text{ seqapprox}(\epsilon) s$$

$$\begin{aligned}
&= \forall s : \text{seq } \mathit{Event} \bullet && \text{[equality]} \\
&\quad (s, s) \in \text{seqapprox}(\epsilon) \Leftrightarrow \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad (s(i) \in O \Rightarrow \exists r : (-\epsilon, \epsilon) \bullet \text{value}(s(i)) = \text{value}(s(i)) + r) \\
&= \forall s : \text{seq } \mathit{Event} \bullet && \text{[select a witness, for } r: 0, \text{ which is in the open range } (-\epsilon, \epsilon)\text{]} \\
&\quad (s, s) \in \text{seqapprox}(\epsilon) \Leftrightarrow \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad (s(i) \in O \Rightarrow \text{value}(s(i)) = \text{value}(s(i)) + 0) \\
&= \forall s : \text{seq } \mathit{Event} \bullet && \text{[arithmetic]} \\
&\quad (s, s) \in \text{seqapprox}(\epsilon) \Leftrightarrow \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad (s(i) \in O \Rightarrow \text{true}) \\
&= \forall s : \text{seq } \mathit{Event} \bullet && \text{[implication]} \\
&\quad (s, s) \in \text{seqapprox}(\epsilon) \Leftrightarrow \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad \text{true}
\end{aligned}$$

□

Lemma B.5 (*seqapprox*(ϵ) output events).

$$\begin{aligned}
&\forall s, s_3 : \text{seq } \mathit{Event} \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad s \text{ seqapprox}(\epsilon) s_3 \Leftrightarrow \forall i : \text{dom } s \bullet \left| \text{value}(t(i)) - \text{value}(s(i)) \right| < \epsilon
\end{aligned}$$

given ϵ is a non-negative real number, and that the event names given by event are identical for each index.

Proof.

$$\begin{aligned}
&\forall s, s_3 : \text{seq } \mathit{Event} \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet s \text{ seqapprox}(\epsilon) s_3 \\
&= \forall s, s_3 : \text{seq } \mathit{Event} \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#t \bullet \\
&\quad \forall s_2 : \text{seq } \mathit{Event} \mid s_2 = s_1 \upharpoonright (I \cup O) \wedge \#t = \#s_2 \bullet \\
&\quad \forall i : \text{dom } s_2 \bullet (s_2(i) \in I \Rightarrow t(i) = s_2(i)) \wedge \\
&\quad (s_2(i) \in O \Rightarrow \text{ev}(s_3(i)) = \text{ev}(s_2(i)) \wedge \\
&\quad \quad \exists r : (-\epsilon, \epsilon) \bullet \text{value}(s_3(i)) = \text{value}(s_2(i)) + r) \\
&\hspace{15em} \text{[unfolding } \text{seqapprox}, \text{ Definition 4.6]} \\
&= \forall s, s_3 : \text{seq } \mathit{Event} \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad (s(i) \in I \Rightarrow s_3(i) = s(i)) \wedge \\
&\quad (s(i) \in O \Rightarrow \text{ev}(s_3(i)) = \text{ev}(s(i)) \wedge \\
&\quad \quad \exists r : (-\epsilon, \epsilon) \bullet \text{value}(s_3(i)) = \text{value}(s(i)) + r) \\
&\hspace{15em} \text{[} s_2 = s, \text{ because } \text{ran } s \subseteq O\text{]} \\
&= \forall s, s_3 : \text{seq } \mathit{Event} \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad (s(i) \in O \Rightarrow \text{ev}(s_3(i)) = \text{ev}(s(i)) \wedge \\
&\quad \quad \exists r : (-\epsilon, \epsilon) \bullet \text{value}(s_3(i)) = \text{value}(s(i)) + r) \\
&\hspace{15em} \text{[} \text{ran}(s) \subseteq O, \text{ so } s(i) \in I \text{ is false]}
\end{aligned}$$

$$\begin{aligned}
&= \forall s, s_3 : \text{seq } Event \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad (s(i) \in O \Rightarrow \exists r : (-\epsilon, \epsilon) \bullet \text{value}(s_3(i)) = \text{value}(s(i)) + r) \\
&\hspace{20em} [\text{given the event names are the same}] \\
&= \forall s, s_3 : \text{seq } Event \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad \exists r : (-\epsilon, \epsilon) \bullet \text{value}(s_3(i)) = \text{value}(s(i)) + r \\
&\hspace{20em} [s(i) \in O \text{ always true}] \\
&= \forall s, s_3 : \text{seq } Event \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad \exists r : \mathbb{R} \mid -\epsilon < r < \epsilon \bullet \text{value}(s_3(i)) = \text{value}(s(i)) + r \\
&\hspace{20em} [\text{definition of open interval}] \\
&= \forall s, s_3 : \text{seq } Event \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad \exists r : \mathbb{R} \bullet \text{value}(s_3(i)) = \text{value}(s(i)) + r \wedge \\
&\quad \quad \quad -\epsilon < r < \epsilon \\
&\hspace{20em} [\text{existential quantification laws}] \\
&= \forall s, s_3 : \text{seq } Event \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad \exists r : \mathbb{R} \bullet r = \text{value}(s_3(i)) - \text{value}(s(i)) \wedge \\
&\quad \quad \quad -\epsilon < r < \epsilon \\
&\hspace{20em} [\text{arithmetic}] \\
&= \forall s, s_3 : \text{seq } Event \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad -\epsilon < \text{value}(s_3(i)) - \text{value}(s(i)) < \epsilon \\
&\hspace{20em} [\text{one-point rule on } r] \\
&= \forall s, s_3 : \text{seq } Event \mid (\text{ran } s \cup \text{ran } s_3) \subseteq O \wedge \#s = \#s_3 \bullet \\
&\quad \forall i : \text{dom } s \bullet \\
&\quad \quad |\text{value}(s_3(i)) - \text{value}(s(i))| < \epsilon \\
&\hspace{20em} [\text{arithmetic}]
\end{aligned}$$

□

Lemma B.6 (*setapprox*(ϵ) reflexivity).

$$\forall r : \mathbb{P} \text{ Event} \mid r \subseteq (I \cup O) \bullet r \text{ setapprox}(\epsilon) r$$

given ϵ is a non-negative real number.

Proof.

$$r \text{ setapprox}(\epsilon) r$$

$$\begin{aligned}
&= \forall r_1, r_2, r : \mathbb{P} \text{Event} \mid r_2 = r_1 \cap (I \cup O) \wedge \#r = \#r_2 \bullet \\
&\quad (r_1, r) \in \text{setapprox}(\epsilon) \Leftrightarrow \\
&\quad \quad \forall e : r_2 \bullet \\
&\quad \quad \quad (e \in I \Leftrightarrow e \in r) \wedge \\
&\quad \quad \quad (e \in O \Leftrightarrow \exists e_1 : r \bullet \text{ev}(e) = \text{ev}(e_1) \wedge \\
&\quad \quad \quad \quad \exists c : (-\epsilon, \epsilon) \bullet \text{value}(e) = \text{value}(e_1) + c) \\
&\quad \quad \quad \hline \\
&\quad \quad \quad r_1 = r \\
&\quad \quad \quad \text{[setapprox}(\epsilon) \text{ definition 4.7]} \\
&= \forall r, r_2 : \mathbb{P} \text{Event} \mid r_2 = r \cap (I \cup O) \wedge \#r = \#r_2 \bullet \\
&\quad (r, r) \in \text{setapprox}(\epsilon) \Leftrightarrow \\
&\quad \quad \forall e : r_2 \bullet \\
&\quad \quad \quad (e \in I \Leftrightarrow e \in r) \wedge \\
&\quad \quad \quad (e \in O \Leftrightarrow \exists e_1 : r \bullet \text{ev}(e) = \text{ev}(e_1) \wedge \\
&\quad \quad \quad \quad \exists c : (-\epsilon, \epsilon) \bullet \text{value}(e) = \text{value}(e_1) + c) \\
&\quad \quad \quad \text{[equality substitution]} \\
&= \forall r : \mathbb{P} \text{Event} \bullet \\
&\quad (r, r) \in \text{setapprox}(\epsilon) \Leftrightarrow \\
&\quad \quad \forall e : r \bullet \\
&\quad \quad \quad (e \in I \Leftrightarrow e \in r) \wedge \\
&\quad \quad \quad (e \in O \Leftrightarrow \exists e_1 : r \bullet \text{ev}(e) = \text{ev}(e_1) \wedge \\
&\quad \quad \quad \quad \exists c : (-\epsilon, \epsilon) \bullet \text{value}(e) = \text{value}(e_1) + c) \\
&\quad \quad \quad \text{[}r_2 = r, \text{ as } r_1 \subseteq I \cup O\text{]} \\
&= \forall r : \mathbb{P} \text{Event} \bullet \\
&\quad (r, r) \in \text{setapprox}(\epsilon) \Leftrightarrow \\
&\quad \quad \forall e : r \bullet \\
&\quad \quad \quad (e \in I \Leftrightarrow e \in r) \wedge \\
&\quad \quad \quad (e \in O \Leftrightarrow \text{ev}(e) = \text{ev}(e) \wedge \\
&\quad \quad \quad \quad \exists c : (-\epsilon, \epsilon) \bullet \text{value}(e) = \text{value}(e) + c) \\
&\quad \quad \quad \text{[select witness for } e_1 : e\text{]} \\
&= \forall r : \mathbb{P} \text{Event} \bullet \\
&\quad (r, r) \in \text{setapprox}(\epsilon) \Leftrightarrow \\
&\quad \quad \forall e : r \bullet \\
&\quad \quad \quad (e \in O \Leftrightarrow \exists c : (-\epsilon, \epsilon) \bullet \text{value}(e) = \text{value}(e) + c) \\
&\quad \quad \quad \text{[discharge input constraint and } \text{ev}(e) = \text{ev}(e)\text{]} \\
&= \forall r : \mathbb{P} \text{Event} \bullet \\
&\quad (r, r) \in \text{setapprox}(\epsilon) \Leftrightarrow \\
&\quad \quad \forall e : r \bullet \\
&\quad \quad \quad (e \in O \Leftrightarrow \text{value}(e) = \text{value}(e) + 0) \\
&\quad \quad \quad \text{[witness for } c : 0\text{]} \\
&= \forall r : \mathbb{P} \text{Event} \bullet \\
&\quad (r, r) \in \text{setapprox}(\epsilon) \Leftrightarrow \\
&\quad \quad \forall e : r \bullet \\
&\quad \quad \quad \text{true} \\
&\quad \quad \quad \text{[arithmetic]}
\end{aligned}$$

$$\begin{aligned}
&= \forall r_1, r : \text{seq } \mathit{Event} \mid (r_1 \cup r) \subseteq O \wedge \#r_1 = \#r \bullet \\
&\quad \forall e : r_1 \bullet \\
&\quad \quad \exists e_1 : r \bullet \exists c : \mathbb{R} \bullet c = \text{value}(e) - \text{value}(e_1) \wedge \\
&\quad \quad \quad -\epsilon < c < \epsilon \\
&\hspace{20em} \text{[arithmetic]} \\
&= \forall r_1, r : \text{seq } \mathit{Event} \mid (r_1 \cup r) \subseteq O \wedge \#r_1 = \#r \bullet \\
&\quad \forall e : r_1 \bullet \\
&\quad \quad \exists e_1 : r \bullet -\epsilon < \text{value}(e) - \text{value}(e_1) < \epsilon \\
&\hspace{20em} \text{[one-point rule on } c \text{]} \\
&= \forall r_1, r : \text{seq } \mathit{Event} \mid (r_1 \cup r) \subseteq O \wedge \#r_1 = \#r \bullet \\
&\quad \forall e : r_1 \bullet \exists e_1 : r \bullet \left| \text{value}(e) - \text{value}(e_1) \right| < \epsilon \\
&\hspace{20em} \text{[arithmetic]} \\
&\hspace{20em} \square
\end{aligned}$$

Lemma B.8. *Given an event $c.i.v$, a communication on the channel c , where v is of a type V , the following holds for all $i \in 1..n$, where $n \geq 1$:*

$$\begin{aligned}
& ; i : 1..n \bullet \square v \in V \bullet \text{Do}(c.i.v) = \\
& \left[\text{true}_r \right. \\
& \quad \vdash \\
& \quad \exists s : \text{seq } \mathit{Event} \mid \#s = n \wedge (\forall i : 1..n \bullet \exists v : V \bullet s i = c.i.v) \bullet \\
& \quad \quad \text{tt prefix } s \wedge \#tt < \#s \wedge \text{wait}' \wedge \text{ref}' \subseteq \{s(\#tt + 1)\} \vee \\
& \quad \quad \text{tt} = s \wedge \neg \text{wait}' \\
& \left. \right]
\end{aligned}$$

Lemma B.9. *Given an expression $e(i, j)$, of type Event , where $i \in 1..n$, and $j \in 1..E(i)$, where $E(i)$ is an expression returning a natural number, based on i , where, for all $n \geq 1$ and $m \geq 1$, the following holds:*

$$\begin{aligned}
& ; i : 1..n \bullet (; j : 1..E(i) \bullet \text{Do}(e(i, j))) = \\
& \left[\text{true}_r \right. \\
& \quad \vdash \\
& \quad \exists i : 1..n \bullet \exists j : 1..E(i) \bullet \\
& \quad \quad \mathcal{E} \left[(\wedge / pi : 1..(i-1); pj : 1..E(pi) \bullet \langle e(pi, pj) \rangle) \wedge \wedge / cj : 1..(j-1) \bullet \langle e(i, cj) \rangle, \right. \\
& \quad \quad \left. \{e(i, j)\} \right] \\
& \quad \mid \\
& \quad \Phi[\wedge / i : 1..n; j : 1..E(i) \bullet \langle e(i, j) \rangle] \\
& \left. \right]
\end{aligned}$$

First, though, we define a simple function which hides events from sequences of events.

Definition B.7 (Sequence Hiding).

$$\begin{array}{l}
\boxed{\text{[} X \text{]}} \\
\text{shide} : (\text{seq } X \times \mathbb{P} X) \rightarrow \text{seq } X \\
\forall s : \text{seq } X; E : \mathbb{P} X \bullet s \text{shide } E = s \upharpoonright (\text{ran } s \setminus E)
\end{array}$$

To illustrate, consider the expression $\langle a, b, c \rangle \text{ shide } \{b\}$, here: $\text{ran } s$ is $\{a, b, c\}$, E is $\{b\}$, and therefore $\text{ran } s \setminus E$ is $\{a, c\}$, so in the definition above, we get $\langle a, b, c \rangle \upharpoonright \{a, c\}$, which is $\langle a, c \rangle$.

Given this definition, we now define notation to capture hiding for our ANN contracts. As our hiding is specific to a type of reactive contract, we require a different operator for pre, peri, and postcondition hiding. Still, since we have a *true* precondition, we only need operators for the peri and postconditions. We denote our hiding operators for the peri and postconditions using \setminus_{peri} and \setminus_{post} , and we present their characteristic predicates below.

Definition B.8 (ANN Contract Event Hiding). *Given a hidden event set E , and that no input event is hidden.*

$$\begin{aligned} & \text{GeneralANNContract}_2 \setminus_{\text{peri}} E \Leftrightarrow \\ & \#input < insize \wedge \mathcal{E}[input, \{ | \text{layerRes}.0.(\#input + 1) | \}] \\ & \vee \\ & \#input = insize \wedge \\ & \exists l : 1..layerNo \bullet \exists n : 1..layerSize(l) \bullet \\ & \quad \text{CRR}(tt = (\text{front} \circ \text{layeroutput}(l, n)) \text{ shide } E \wedge \text{last} \circ \text{layeroutput}(l, n) \notin E \cup \text{ref}') \end{aligned}$$

$$\begin{aligned} & \text{GeneralANNContract}_3 \setminus_{\text{post}} E \Leftrightarrow \\ & \#input = insize \wedge \Phi[\text{layeroutput}(\text{layerNo}, \text{layerSize}(\text{layerNo})) \text{ shide } E] \end{aligned}$$

Our \setminus_{post} definition is fairly straightforward. We apply the *shide* function to the complete trace of our ANN contracts, given by *layeroutput* (Def. 4.2), as discussed. For example, suppose we have an ANN with input $\langle 1, 1 \rangle$, *layerNo* = 2 and one output node.

Example B.1 (Postcondition Hiding Example).

$$\begin{aligned} & \#input = insize \wedge \Phi[\text{layeroutput}(2, 1)] \setminus_{\text{post}} \{ | \text{layerRes}.1 | \} \\ & = \#input = insize \wedge \Phi[\text{layeroutput}(2, 1) \text{ shide } \{ | \text{layerRes}.1 | \}] \quad [\text{Def. B.8}] \\ & = \#input = insize \wedge \Phi[\langle \text{layerRes}.0.1.1, \text{layerRes}.0.2.1, \text{layerRes}.1.1.1, \text{layerRes}.2.1.1 \rangle \\ & \quad \text{shide } \{ | \text{layerRes}.1 | \}] \quad [\text{Def. 4.2 and input is } \langle 1, 1 \rangle] \\ & = \#input = insize \wedge \Phi[\langle \text{layerRes}.0.1.1, \text{layerRes}.0.2.1, \text{layerRes}.1.1.1, \text{layerRes}.2.1.1 \rangle \\ & \quad \upharpoonright \{ | \text{layerRes} | \} \setminus \{ | \text{layerRes}.1 | \}] \quad [\text{Def. B.7 and ran } s \text{ here is } \{ | \text{layerRes} | \}] \\ & = \#input = insize \wedge \Phi[\langle \text{layerRes}.0.1.1, \text{layerRes}.0.2.1, \text{layerRes}.2.1.1 \rangle] \quad [\text{evaluate } \upharpoonright] \end{aligned}$$

However, our pericondition hiding operator, denoted by \setminus_{peri} , is more involved. First, we do not permit the hiding of input events, as, in an ANN, this introduces nondeterminism. Moreover, in this context, hiding is used to hide the events of the hidden layers.

In our general pattern, presented in Definition 4.1, we capture each node's output event by a single predicate formed by the reactive relational operator $\mathcal{E}[t, A]$. This operator, as discussed, specifies that $tt = t$ and that no event in the set A is in the refusal set, meaning

those events are accepted. Further, this operator applies the healthiness-condition CRR (Def.B.14) to denote that this relation is a stateful-failure reactive relation; more details are in [30].

When considering hiding, we can remove those hidden events from the trace using $shide$, as in the postcondition hiding operator. In the refusals, however, if we remove those events in set E from the acceptance set A , in a state where we only accept hidden events, we obtain a condition of the form $\mathcal{E}[t, \{\}]$. This is a problem as we now have a state in which no event is accepted, but there is a restriction on tt ; in other words, we introduce deadlock. Further, we present a nondeterministic chance of deadlock whenever events are hidden because we hide events from the trace t using $shide$.

To address this, we define a new condition to capture each node's output instead of using \mathcal{E} , which is also CRR -healthy. This condition sets tt to the trace up to this node, given by $front \circ layeroutput(\dots)$, with the hidden events removed using $shide$, as in the postcondition. In this condition, we also specify that the output event of the node, denoted by $last \circ layeroutput(l, n)$, is not contained within the union of the hidden event set E and the refusals set ref' . So, when considering a state where the output event of a node is hidden, this condition evaluates to $false$ instead of evaluating to $tt = t \wedge \{\} \notin ref'$, which is deadlock. In other words, this state is no longer considered a stable state of the reactive contract.

We describe, below, an example illustrating the application of \backslash_{peri} using the same ANN as used in Example B.1 and the same hidden events.

Example B.2 (Pericondition Hiding Example).

$$\begin{aligned}
& (\#input < insize \wedge \mathcal{E}[input, \{ | layerRes.0.(\#input + 1) | \}] \vee \\
& \#input = insize \wedge \\
& \exists l : 1..2 \bullet \exists n : 1..layerSize(l) \bullet \\
& \quad \mathcal{E}[front \circ layeroutput(l, n), \{ last \circ layeroutput(l, n) \}]) \\
& \backslash_{post} \{ | layerRes.1 | \} \\
& = (\#input < insize \wedge \mathcal{E}[input, \{ | layerRes.0.(\#input + 1) | \}] \vee \\
& \quad \#input = insize \wedge \\
& \quad (\mathcal{E}[front \circ layeroutput(1, 1), \{ last \circ layeroutput(1, 1) \}] \vee \\
& \quad \mathcal{E}[front \circ layeroutput(2, 1), \{ last \circ layeroutput(2, 1) \}])) \\
& \quad \backslash_{post} \{ | layerRes.1 | \} \\
& \hspace{20em} [unfolding existential quantifiers] \\
& = (\#input < insize \wedge \mathcal{E}[input, \{ | layerRes.0.(\#input + 1) | \}] \vee \hspace{10em} [Def.4.2] \\
& \quad \#input = insize \wedge \\
& \quad (\mathcal{E}[\langle layerRes.0.1.1, layerRes.0.2.1 \rangle, \{ layerRes.1.1.1 \}] \vee \\
& \quad \mathcal{E}[\langle layerRes.0.1.1, layerRes.0.2.1, layerRes.1.1.1 \rangle, \{ layerRes.2.1.1 \}])) \\
& \quad \backslash_{post} \{ | layerRes.1 | \}
\end{aligned}$$

$$\begin{aligned}
&= \#input < insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) |\}] \vee && [Def.B.8] \\
&\quad \#input = insize \wedge \\
&\quad (CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1 \rangle \text{shide } \{| layerRes.1 |\} \wedge \\
&\quad \quad layerRes.1.1.1 \notin \{| layerRes.1 |\} \cup ref')) \\
&\quad \vee \\
&\quad CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1, layerRes.1.1.1 \rangle \\
&\quad \quad \text{shide } \{| layerRes.1 |\} \\
&\quad \quad \wedge \\
&\quad \quad layerRes.2.1.1 \notin \{| layerRes.1 |\} \cup ref')) \\
&= \#input < insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) |\}] \vee \\
&\quad \#input = insize \wedge \\
&\quad (CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1 \rangle \text{shide } \{| layerRes.1 |\} \wedge \\
&\quad \quad false)) \\
&\quad \vee \\
&\quad CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1, layerRes.1.1.1 \rangle \\
&\quad \quad \text{shide } \{| layerRes.1 |\} \\
&\quad \quad \wedge \\
&\quad \quad layerRes.2.1.1 \notin \{| layerRes.1 |\} \cup ref')) \\
&&& [layerRes.1.1.1 \in \{| layerRes.1 |\}] \\
&= \#input < insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) |\}] \vee && [logic] \\
&\quad \#input = insize \wedge \\
&\quad (CRR(false) \vee \\
&\quad CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1, layerRes.1.1.1 \rangle \\
&\quad \quad \text{shide } \{| layerRes.1 |\} \\
&\quad \quad \wedge \\
&\quad \quad layerRes.2.1.1 \notin \{| layerRes.1 |\} \cup ref')) \\
&= \#input < insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) |\}] \vee \\
&\quad \#input = insize \wedge \\
&\quad (CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1, layerRes.1.1.1 \rangle \\
&\quad \quad \text{shide } \{| layerRes.1 |\} \\
&\quad \quad \wedge \\
&\quad \quad layerRes.2.1.1 \notin \{| layerRes.1 |\} \cup ref')) \\
&&& [false is CRR-healthy] \\
&= \#input < insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) |\}] \vee \\
&\quad \#input = insize \wedge \\
&\quad (CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1 \rangle \\
&\quad \quad \wedge \\
&\quad \quad layerRes.2.1.1 \notin \{| layerRes.1 |\} \cup ref')) \\
&&& [evaluate shide] \\
&= \#input < insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) |\}] \vee \\
&\quad \#input = insize \wedge \\
&\quad (CRR(tt = \langle layerRes.0.1.1, layerRes.0.2.1 \rangle \\
&\quad \quad \wedge \\
&\quad \quad layerRes.2.1.1 \notin ref')) \\
&&& [layerRes.2.1.1 \notin \{| layerRes.1 |\}]
\end{aligned}$$

$$\begin{aligned}
&= \#input < insize \wedge \mathcal{E}[input, \{| layerRes.0.(\#input + 1) |\}] \vee && [Def.B.14] \\
&\quad \#input = insize \wedge \\
&\quad \mathcal{E}\langle \{layerRes.0.1.1, layerRes.0.2.1\}, \{layerRes.2.1.1\} \rangle
\end{aligned}$$

The next section presents the laws of the UTP reactive contract calculus that we rely on in this work.

B.2 Existing Laws

Here, we present definitions and theorems from work by Foster et al. in [30] and [29]. All reactive contracts we use in this work are *NCSP*-healthy, as we use them to capture CSP processes.

First, we present the basic reactive operators used to construct the contracts capturing our CSP processes. These definitions are extracted from Def. 4.7 and Def. 2.10 from [30].

Definition B.9 (Reactive Operators).

$$\begin{aligned}
Do(a) &\hat{=} [true_r \vdash \mathcal{E}[true, \langle \rangle, \{a\}] \mid \Phi[true, id, \langle a \rangle]] \\
Skip &\hat{=} [true_r \vdash false \mid tt = \langle \rangle \wedge st' = st]
\end{aligned}$$

Next, we present selected laws for the sequential composition of reactive contracts from Theorem 2.17 and Theorem 4.8 in [30].

Theorem B.1 (Reactive Contract Sequential Composition).

$$\begin{aligned}
[\vdash P_2 \mid P_3]; [\vdash Q_2 \mid Q_3] &= [\vdash P_2 \vee (P_3; Q_2) \mid P_3; Q_3] && [1] \\
\Phi[s_1, \sigma_1, t_1]; \Phi[s_2, \sigma_2, t_2] &= \Phi[s_1 \wedge \sigma_1 \dagger s_2, \sigma_2 \circ \sigma_1, t_1 \frown \sigma_1 \dagger t_2] && [2] \\
\Phi[s_1, \sigma_1, t_1]; \mathcal{E}[s_2, t_2, E] &= \mathcal{E}[s_1 \wedge \sigma_1 \dagger s_2, t_1 \frown \sigma_1 \dagger t_2, \sigma_1 \dagger E] && [3] \\
(\bigwedge i \in I \bullet \mathcal{E}[s(i), t, E(i)]) &= \mathcal{E}[\bigwedge i \in I \bullet s(i), t, \bigcup i \in I \bullet E(i)] && [4] \\
(\bigvee i \in I \bullet \mathcal{E}[s(i), t, E(i)]) &= \mathcal{E}[\bigvee i \in I \bullet s(i), t, \bigcap i \in I \bullet E(i)] && [5]
\end{aligned}$$

Definition 5.1 from [30].

Definition B.10. *Indexed External Choice*

$$\begin{aligned}
\Box i \in I \bullet [P_1(i) \vdash P_2(i) \mid P_3(i)] &\hat{=} \\
[\bigwedge i \in I \bullet P_1(i) \vdash (\bigwedge i \in I \bullet R5(P_2(i))) \vee (\bigvee i \in I \bullet R4(P_2(i))) \mid \bigvee i \in I \bullet P_3(i)]
\end{aligned}$$

Theorem 5.2 from [30].

Theorem B.2. *Trace Filtering*

$$\begin{aligned}
R4(\Phi[s, \sigma, \langle \rangle]) &= false \\
R4(\Phi[s, \sigma, \langle a, \dots \rangle]) &= \Phi[s, \sigma, \langle a, \dots \rangle] \\
R5(\mathcal{E}[s, \langle \rangle, E]) &= \mathcal{E}[s, \langle \rangle, E] \\
R5(\mathcal{E}[s, \langle a, \dots \rangle, E]) &= false
\end{aligned}$$

Definition 6.1 from [29].

Definition B.11. *Pre-, Peri- and postcondition extraction functions*

$$\begin{aligned} pre_R(P) &\hat{=} \neg_r P[true, false, false / ok, ok', wait] \\ peri_R(P) &\hat{=} P[true, true, false, true / ok, ok', wait, wait'] \\ post_R(P) &\hat{=} P[true, true, false, false / ok, ok', wait, wait'] \end{aligned}$$

Theorem 6.2 from [30].

Theorem B.3. *Condition extraction on reactive contracts*

$$\begin{aligned} pre_R([P_1 \vdash P_2 \mid P_3]) &= P_1 \\ peri_R([P_1 \vdash P_2 \mid P_3]) &= P_1 \Rightarrow_r P_2 \\ post_R([P_1 \vdash P_2 \mid P_3]) &= P_1 \Rightarrow_r P_3 \end{aligned}$$

Given P_1 , P_2 and P_3 are RR healthy.

Definition B.12. *Extraction function abbreviations*

$$\begin{aligned} (P)_1 &\hat{=} pre_R(P) \\ (P)_2 &\hat{=} peri_R(P) \\ (P)_3 &\hat{=} post_R(P) \end{aligned}$$

Definition 2.12 from [30].

Definition B.13.

$$[P_1 \vdash P_2 \mid P_3] \hat{=} R_1 \circ R_2 \circ R3_h(ok \wedge P_1 \Rightarrow ok' \wedge (P_2 \triangleleft wait' \triangleright P_3))$$

Definition 4.6 from [30].

Definition B.14.

$$\begin{aligned} \mathcal{I}[s(st), t(st)] &\hat{=} CRC(s(st) \Rightarrow_r \neg_r (t(st) \leq tt)) \\ \mathcal{E}[s(st), t(st), E(st)] &\hat{=} CRR(s(st) \wedge tt = t(st) \wedge (\forall e \in E(st) \bullet e \notin ref')) \\ \Phi[s(st), \sigma, t(st)] &\hat{=} CRF(s(st) \wedge st' = \sigma(st) \wedge tt = t(st)) \end{aligned}$$

Definition 2.1 from [29].

Definition B.15.

$$\begin{aligned} P \sqsubseteq Q &\hat{=} [Q \Rightarrow P] \text{ where } \alpha(P) = \alpha(Q) \\ \text{if } \alpha(P) = \{x_1 \dots x_n\} &\text{ then } [P] \hat{=} (\forall x_1 \dots x_n \bullet P) \end{aligned}$$

Theorem 2.16 from [30].

Theorem B.4 (Reactive Design Refinement).

$$[P_1 \vdash P_2 \mid P_3] \sqsubseteq [Q_1 \vdash Q_2 \mid Q_3] \text{ if, and only if, } Q_1 \sqsubseteq P_1, P_2 \sqsubseteq (Q_2 \wedge P_1), \text{ and } P_3 \sqsubseteq (Q_3 \wedge P_1)$$

Appendix C

ANN Component Conformance Lemmas

Lemma C.1.

$$\begin{aligned}
& \neg \exists x_1, \dots, x_{insize} : \text{Value} \bullet \exists y_1, \dots, y_{outside} : \text{Value} \mid p \bullet \exists i : 1 \dots outside \bullet \\
& \quad \{ \text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{insize} \rangle) \} \cap \{ x : \mathbb{R} \mid |x - y_i| < \epsilon \} = \emptyset \\
\Rightarrow & [(Q_2 \setminus_{\text{peri}} \text{ANNHiddenEvs})[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
& \quad \exists s : \text{seq Event}; a : \mathbb{P} \text{Event} \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
& \quad P_2[s, \alpha P \setminus a/\text{tt}, \text{ref}']]
\end{aligned}$$

Proof.

$$\begin{aligned}
& [(Q_2 \setminus_{\text{peri}} \text{ANNHiddenEvs})[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
& \quad \exists s : \text{seq Event}, a : \mathbb{P} \text{Event} \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
& \quad P_2[s, \alpha P \setminus a/\text{tt}, \text{ref}']] \\
= & \quad \forall st, st', \text{wait}, \text{wait}', \text{ref}, \text{ref}', \text{ok}, \text{ok}', \text{tr}, \text{tr}', \text{tt} \bullet \quad \text{[unfolding universal closure]} \\
& \quad Q_2 \setminus_{\text{peri}} \text{ANNHiddenEvs}[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
& \quad \exists s; a \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet P_2[s, \alpha P \setminus a/\text{tt}, \text{ref}']] \\
= & \quad \forall st, st', \text{wait}, \text{wait}', \text{ref}, \text{ref}', \text{ok}, \text{ok}', \text{tr}, \text{tr}', \text{tt} \bullet \\
& \quad (\text{ok} \wedge \text{ok}' \wedge \neg \text{wait} \wedge \text{wait}' \wedge \text{st}' = \text{st} \wedge \\
& \quad \#input < insize \wedge \mathcal{E}[\text{input}, \{ | \text{layerRes}.0.(\#input + 1) | \}]) \\
& \quad \vee \\
& \quad \#input = insize \wedge \\
& \quad \exists l : 1..layerNo \bullet \exists n : 1..layerSize(l) \bullet \\
& \quad \text{CRR}(\text{tt} = (\text{front} \circ \text{layeroutput}(l, n)) \text{shide} \text{ANNHiddenEvs} \wedge \\
& \quad \text{last} \circ \text{layeroutput}(l, n) \notin \text{ANNHiddenEvs} \cup \text{ref}') \\
& \quad)[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
& \quad \exists s; a \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet P_2[s, \alpha P \setminus a/\text{tt}, \text{ref}']] \\
& \quad \text{[unfolding } Q_2: \text{ pericondition extraction } (-_2) \text{ and hiding, Definition B.8]}
\end{aligned}$$

$$\begin{aligned}
& = \forall st, st', wait, wait', ref, ref', ok, ok', tr, tr', tt \bullet \\
& \quad (ok \wedge ok' \wedge \neg wait \wedge wait' \wedge st' = st \wedge \\
& \quad \#input < insize \wedge \mathcal{E}[input, \{ | layerRes.0.(\#input + 1) | \}]) \\
& \quad \vee \\
& \quad \#input = insize \wedge \\
& \quad \exists l : 1..layerNo \bullet \exists n : 1..layerSize(l) \bullet \\
& \quad \quad CRR(tt = (front \circ layeroutput(l, n)) shide ANNHIDDENEVTS \wedge \\
& \quad \quad \quad last \circ layeroutput(l, n) \notin ANNHIDDENEVTS \cup ref') \\
& \quad) [inp/layerRes.0, out/layerRes.layerNo] \Rightarrow \\
& \quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
& \quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet \\
& \quad \quad (ok \wedge ok' \wedge \neg wait \wedge wait' \wedge st' = st \wedge \\
& \quad \quad \quad ((\exists i : \text{dom } inp \bullet tt = \bigwedge / j : 1 .. (i-1) \bullet \langle inp(j).x_j \rangle \wedge \\
& \quad \quad \quad \quad \quad \quad \quad inp(i).x_i \notin ref') \\
& \quad \quad \quad \vee \\
& \quad \quad \quad (\exists i : \text{dom } out \bullet tt = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \wedge \\
& \quad \quad \quad \quad \quad \quad \quad \bigwedge / j : 1 .. (i-1) \bullet \langle out(j).y_j \rangle \wedge \\
& \quad \quad \quad \quad \quad \quad \quad out(i).y_i \notin ref')) \\
& \quad \quad) [s, \alpha P \setminus a/tt, ref'] \\
& \hspace{20em} [\text{unfolding } P_2: \text{ pericondition extraction } (-_2) \text{ and Definition 4.4}] \\
& = \forall ref, ref', tr, tr', tt \bullet \\
& \quad (\#input < insize \wedge \mathcal{E}[input, \{ | layerRes.0.(\#input + 1) | \}]) \\
& \quad \vee \\
& \quad \#input = insize \wedge \\
& \quad \exists l : 1..layerNo \bullet \exists n : 1..layerSize(l) \bullet \\
& \quad \quad CRR(tt = (front \circ layeroutput(l, n)) shide ANNHIDDENEVTS \wedge \\
& \quad \quad \quad last \circ layeroutput(l, n) \notin ANNHIDDENEVTS \cup ref') \\
& \quad) [inp/layerRes.0, out/layerRes.layerNo] \Rightarrow \\
& \quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
& \quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet \\
& \quad \quad (\exists i : \text{dom } inp \bullet tt = \bigwedge / j : 1 .. (i-1) \bullet \langle inp(j).x_j \rangle \wedge \\
& \quad \quad \quad \quad \quad \quad \quad inp(i).x_i \notin ref' \\
& \quad \quad \quad \vee \\
& \quad \quad \quad (\exists i : \text{dom } out \bullet tt = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \wedge \\
& \quad \quad \quad \quad \quad \quad \quad \bigwedge / j : 1 .. (i-1) \bullet \langle out(j).y_j \rangle \wedge \\
& \quad \quad \quad \quad \quad \quad \quad out(i).y_i \notin ref') \\
& \quad \quad) [s, \alpha P \setminus a/tt, ref'] \\
& \hspace{20em} [\text{one-point rule on } ok, ok', wait, wait', st]
\end{aligned}$$

$$\begin{aligned}
= & \forall ref, ref', tr, tr', tt \bullet \\
& (\#input < insize \wedge tt = input \wedge \{ | layerRes.0.(\#input + 1) | \} \cap ref' = \emptyset \\
& \vee \\
& \#input = insize \wedge \\
& \exists l : 1..layerNo \bullet \exists n : 1..layerSize(l) \bullet \\
& \quad tt = (front \circ layeroutput(l, n)) \text{ shide } ANNHIDDENEVTS \wedge \\
& \quad \text{last} \circ layeroutput(l, n) \notin ANNHIDDENEVTS \cup ref' \\
&) [inp / layerRes.0, out / layerRes.layerNo] \Rightarrow \\
& \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
& \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet \\
& \quad (\exists i : \text{dom } inp \bullet tt = \bigwedge / j : 1..(i-1) \bullet \langle inp(j).x_j \rangle \wedge \\
& \quad \quad \quad inp(i).x_i \notin ref' \\
& \quad \vee \\
& \quad \exists i : \text{dom } out \bullet tt = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \wedge \\
& \quad \quad \quad \bigwedge / j : 1..(i-1) \bullet \langle out(j).y_j \rangle \wedge \\
& \quad \quad \quad out(i).y_i \notin ref' \\
&) [s, \alpha P \setminus a / tt, ref']
\end{aligned}$$

[CRR healthiness and unfold \mathcal{E} , Definition B.14]

$$\begin{aligned}
= & \forall ref, ref', tr, tr', tt \bullet \\
& (\exists x_1, \dots, x_{insize} \bullet \\
& \quad \exists i : 1..insize \bullet \\
& \quad \quad tt = \bigwedge / j : 1..(i-1) \bullet \langle layerRes.0.j.x_j \rangle \wedge \\
& \quad \quad \quad layerRes.0.i.x_i \notin ref' \\
& \quad \vee \\
& \quad \exists l : 1..layerNo \bullet \exists n : 1..layerSize(l) \bullet \\
& \quad \quad tt = front \circ layeroutput(l, n) \text{ shide } ANNHIDDENEVTS \wedge \\
& \quad \quad \text{last} \circ layeroutput(l, n) \notin ANNHIDDENEVTS \cup ref' \\
&) [inp / layerRes.0, out / layerRes.layerNo] \Rightarrow \\
& \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
& \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet \\
& \quad (\exists i : \text{dom } inp \bullet tt = \bigwedge / j : 1..(i-1) \bullet \langle inp(j).x_j \rangle \wedge \\
& \quad \quad \quad inp(i).x_i \notin ref' \\
& \quad \vee \\
& \quad \exists i : \text{dom } out \bullet tt = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \wedge \\
& \quad \quad \quad \bigwedge / j : 1..(i-1) \bullet \langle out(j).y_j \rangle \wedge \\
& \quad \quad \quad out(i).y_i \notin ref' \\
&) [s, \alpha P \setminus a / tt, ref']
\end{aligned}$$

[assumption about *input*]

$$\begin{aligned}
&= \forall \text{ref}', \text{tr}, \text{tr}', \text{tt} \bullet \\
&\quad (\exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{layerRes}.0.j.x_j \rangle \wedge \\
&\quad \quad \quad \text{layerRes}.0.i.x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \exists l : 1 \dots \text{layerNo} \bullet \exists n : 1 \dots \text{layerSize}(l) \bullet \\
&\quad \quad \text{tt} = \text{front} \circ \text{layeroutput}(l, n) \text{ shide } \text{ANNHiddenEvs} \wedge \\
&\quad \quad \text{last} \circ \text{layeroutput}(l, n) \notin \text{ANNHiddenEvs} \cup \text{ref}' \\
&)[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.\text{layerNo}] \Rightarrow \\
&\quad \exists s; a \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists x_1, \dots, x_{\# \text{inp}}; y_1, \dots, y_{\# \text{out}} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet \text{tt} = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&)[s, \alpha P \setminus a/\text{tt}, \text{ref}'] \\
&\quad \quad \quad [\text{ref} \text{ not referred to in periconditions, CRR healthiness condition [29]]] \\
&= \forall \text{ref}', \text{tr}, \text{tr}', \text{tt} \bullet \\
&\quad (\exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{layerRes}.0.j.x_j \rangle \wedge \\
&\quad \quad \quad \text{layerRes}.0.i.x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \exists l : 1 \dots \text{layerNo} \bullet \exists n : 1 \dots \text{layerSize}(l) \bullet \\
&\quad \quad \text{tt} = \text{front} \circ \text{layeroutput}(l, n) \text{ shide } \text{ANNHiddenEvs} \wedge \\
&\quad \quad \text{nodeoutput}(l, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ANNHiddenEvs} \cup \text{ref}' \\
&)[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.\text{layerNo}] \Rightarrow \\
&\quad \exists s; a \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists x_1, \dots, x_{\# \text{inp}}; y_1, \dots, y_{\# \text{out}} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet \text{tt} = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&)[s, \alpha P \setminus a/\text{tt}, \text{ref}'] \\
&\quad \quad \quad [\text{last} \circ \text{layeroutput}(l, n) = \text{nodeoutput}(l, n, \langle x_1, \dots, x_{\text{insize}} \rangle)]
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', \text{tr}, \text{tr}', \text{tt} \bullet && \text{[Lemma B.1 (modified for front)]} \\
&\quad (\exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{layerRes}.0.j.x_j \rangle \wedge \\
&\quad \quad \quad \text{layerRes}.0.i.x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \exists l : 1 \dots \text{layerNo} \bullet \exists n : 1 \dots \text{layerSize}(l) \bullet \\
&\quad \quad \text{tt} = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{layerRes}.0.in.x_{in} \rangle \\
&\quad \quad \quad \bigwedge \\
&\quad \quad \quad \bigwedge / p_l : 1 \dots l-1 \bullet \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots \text{layerSize}(p_l) \bullet \\
&\quad \quad \quad \quad \quad \langle \text{nodeoutput}(p_l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \bigwedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots n-1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{nodeoutput}(l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \text{shide ANNHIDDENEVTS} \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \text{nodeoutput}(l, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ANNHIDDENEVTS} \cup \text{ref}' \\
&\quad \quad \quad \text{])[inp/layerRes}.0, \text{out/layerRes}.layerNo] \Rightarrow \\
&\quad \exists s; a \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom inp} \bullet \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \vee \\
&\quad \quad \quad \exists i : \text{dom out} \bullet \text{tt} = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad \quad \text{])[s, \alpha P \setminus a/\text{tt}, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt \bullet && \text{[unfold nodeoutput, Definition B.1]} \\
&\quad (\exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{layerRes}.0.j.x_j \rangle \wedge \\
&\quad \quad \quad \text{layerRes}.0.i.x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \exists l : 1 \dots \text{layerNo} \bullet \exists n : 1 \dots \text{layerSize}(l) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{layerRes}.0.in.x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_l : 1 \dots l-1 \bullet \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots \text{layerSize}(p_l) \bullet \\
&\quad \quad \quad \quad \quad \langle \text{layerRes}.p_l.p_n.\text{annoutput}(p_l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots n-1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{layerRes}.l.p_n.\text{annoutput}(l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \text{shide ANNHIDDENEVTS} \\
&\quad \quad \wedge \\
&\quad \quad \quad \text{layerRes}.l.n.\text{annoutput}(l, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ANNHIDDENEVTS} \cup \text{ref}' \\
&)] \text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.\text{layerNo}] \Rightarrow \\
&\exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&)] s, \alpha P \setminus a/tt, \text{ref}'
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt \bullet \quad \text{[apply [inp/layerRes.0, out/layerRes.layerNo]]} \\
&\quad (\exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \exists l : 1 \dots \text{layerNo} - 1 \bullet \exists n : 1 \dots \text{layerSize}(l) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_l : 1 \dots l - 1 \bullet \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots \text{layerSize}(p_l) \bullet \\
&\quad \quad \quad \quad \quad \langle \text{layerRes.p_l.p_n.annoutput}(p_l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{layerRes.l.p_n.annoutput}(l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \text{shide ANNHIDDENEVTS} \\
&\quad \quad \wedge \\
&\quad \quad \quad \text{layerRes.l.n.annoutput}(l, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ANNHIDDENEVTS} \cup \text{ref}' \\
&\quad \vee \\
&\quad \exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_l : 1 \dots \text{layerNo} - 1 \bullet \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots \text{layerSize}(p_l) \bullet \\
&\quad \quad \quad \quad \quad \langle \text{layerRes.p_l.p_n.annoutput}(p_l, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(p_n).annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \text{shide ANNHIDDENEVTS} \\
&\quad \quad \wedge \\
&\quad \quad \quad \text{out}(n).annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ANNHIDDENEVTS} \cup \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt \bullet && \text{[evaluate } \textit{shide} \textit{ANNHiddenEvs}] \\
&\quad (\exists x_1, \dots, x_{\textit{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \textit{insize} \bullet \\
&\quad \quad \quad tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \textit{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \textit{inp}(i).x_i \notin \textit{ref}' \\
&\quad \vee \\
&\quad \exists l : 1 \dots \textit{layerNo} - 1 \bullet \exists n : 1 \dots \textit{layerSize}(l) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \textit{insize} \bullet \\
&\quad \quad \quad \langle \textit{inp}(in).x_{in} \rangle \\
&\quad \quad \wedge \\
&\quad \quad \textit{layerRes}.l.n.\textit{annoutput}(l, n, \langle x_1, \dots, x_{\textit{insize}} \rangle) \notin \textit{ANNHiddenEvs} \cup \textit{ref}' \\
&\quad \vee \\
&\quad \exists n : 1 \dots \textit{layerSize}(\textit{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \textit{insize} \bullet \\
&\quad \quad \quad \langle \textit{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \langle \textit{out}(p_n).\textit{annoutput}(\textit{layerNo}, p_n, \langle x_1, \dots, x_{\textit{insize}} \rangle) \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \textit{out}(n).\textit{annoutput}(\textit{layerNo}, n, \langle x_1, \dots, x_{\textit{insize}} \rangle) \notin \textit{ANNHiddenEvs} \cup \textit{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \textit{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \textit{ref}') \textit{setapprox}(\epsilon) a \bullet \\
&\quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid P \bullet \\
&\quad \quad (\exists i : \textit{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \textit{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \textit{inp}(i).x_i \notin \textit{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \textit{dom out} \bullet tt = \bigwedge / n : \textit{dom inp} \bullet \langle \textit{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \textit{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \textit{out}(i).y_i \notin \textit{ref}' \\
&\quad) [s, \alpha P \setminus a / tt, \textit{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt \bullet && [\text{evaluate} \notin \text{ANNHiddenEvs} \cup \text{ref}'] \\
&\quad (\exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \exists l : 1 \dots \text{layerNo} - 1 \bullet \exists n : 1 \dots \text{layerSize}(l) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \wedge \\
&\quad \quad \text{false} \\
&\quad \vee \\
&\quad \exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid P \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', \text{tr}, \text{tr}', \text{tt} \bullet && \text{[logic of false]} \\
&\quad (\exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad \exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \quad \exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad \quad \text{tt} = \bigwedge / \text{in} : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \quad \langle \text{inp}(\text{in}).x_{\text{in}} \rangle \\
&\quad \quad \quad \quad \bigwedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots n-1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \wedge \\
&\quad \quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\quad \exists s; a \mid \text{tt seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists x_1, \dots, x_{\# \text{inp}}; y_1, \dots, y_{\# \text{out}} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom inp} \bullet \text{tt} = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \vee \\
&\quad \quad \quad \quad \exists i : \text{dom out} \bullet \text{tt} = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a/\text{tt}, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&\Leftarrow \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists i : 1 .. \text{insize} \bullet \\
&\quad \quad tt = \bigwedge / j : 1 .. (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \vee \\
&\quad \exists n : 1 .. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 .. \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge \\
&\quad \quad \quad \bigwedge / p_n : 1 .. n-1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 .. (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : 1 .. (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

[strengthen condition, universally quantify input variables]

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet && \text{[antecedant has a disjunction]} \\
&\quad (\exists i : 1 .. \text{insize} \bullet \\
&\quad \quad tt = \bigwedge / j : 1 .. (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 .. (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 .. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}'] \\
&\quad \wedge \\
&\quad (\exists n : 1 .. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 .. \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1 .. n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 .. (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 .. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&\Leftarrow \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet && \text{[in the first conjunct, strengthen the disjunction]} \\
&\quad (\exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a/tt, \text{ref}'] \\
&\wedge \\
&\quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n-1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad) \\
&\quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet && \text{[substitute } s \text{ and } \alpha P \setminus a \text{ in the first conjunct]} \\
&\quad (\exists i : 1 \dots \text{insize} \bullet \\
&\quad \quad tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet s = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \alpha P \setminus a \\
&\quad \quad) \\
&\quad \wedge \\
&\quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n-1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \vee \\
&\quad \quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall ref', tr, tr', tt, x_1, \dots, x_{insize} \bullet \quad [\text{extend scope of variables } s, a \text{ and } y] \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad (\exists i : 1 \dots insize \bullet \\
&\quad \quad tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle inp(j).x_j \rangle \wedge \\
&\quad \quad inp(i).x_i \notin ref' \\
&\quad) \\
&\quad \Rightarrow \\
&\quad (\exists i : \text{dom } inp \bullet s = \bigwedge / j : 1 \dots (i-1) \bullet \langle inp(j).x_j \rangle \wedge \\
&\quad \quad inp(i).x_i \notin \alpha P \setminus a \\
&\quad) \\
&\quad \wedge \\
&\quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots insize \bullet \\
&\quad \quad \quad \langle inp(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n-1 \bullet \\
&\quad \quad \quad \quad \langle out(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&\quad \quad \wedge \\
&\quad \quad \quad out(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{insize} \rangle) \notin ref' \\
&\quad) \\
&\quad \Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad (\exists i : \text{dom } inp \bullet tt = \bigwedge / j : 1 \dots (i-1) \bullet \langle inp(j).x_j \rangle \wedge \\
&\quad \quad inp(i).x_i \notin ref' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom } out \bullet tt = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i-1) \bullet \langle out(j).y_j \rangle \wedge \\
&\quad \quad \quad out(i).y_i \notin ref' \\
&\quad) [s, \alpha P \setminus a / tt, ref']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet && \text{[extend scope of } i \text{]} \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out}, i : 1.. \text{insize} \mid p \bullet \\
&\quad \quad \quad tt = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \Rightarrow \\
&\quad \quad \quad s = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \alpha P \setminus a \\
&\quad \wedge \\
&\quad (\exists n : 1.. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1.. \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1.. n-1 \bullet \\
&\quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\quad \Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \vee \\
&\quad \quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \quad \bigwedge / j : 1..(i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{insize} \bullet && \text{[propagate predicate of } s \text{ and } a\text{]} \\
&\quad \exists s; a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out}, i : 1..insize \mid p \bullet \\
&\quad \quad \quad tt \text{ seqapprox}(\epsilon) s \wedge \\
&\quad \quad \quad (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \wedge \\
&\quad \quad \quad (tt = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}') \\
&\quad \quad \quad \Rightarrow \\
&\quad \quad \quad s = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \alpha P \setminus a \\
&\quad \wedge \\
&\quad (\exists n : 1..layerSize(layerNo) \bullet \\
&\quad \quad tt = \bigwedge / in : 1..insize \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1..n-1 \bullet \\
&\quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle) \\
&\quad \wedge \\
&\quad \text{out}(n).\text{annoutput}(layerNo, n, \langle x_1, \dots, x_{insize} \rangle) \notin \text{ref}' \\
&\quad) \\
&\quad \Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}') \\
&\quad \quad \quad \vee \\
&\quad \quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \quad \bigwedge / j : 1..(i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}') \\
&\quad \quad \quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet && \text{[one-point rule on } tt \text{ and } s \text{ in first conjunct]} \\
&\quad \exists a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out}, i : 1..insize \mid p \bullet \\
&\quad \quad \quad \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \text{seqapprox}(\epsilon) \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \wedge \\
&\quad \quad \quad (\text{inp}(i).x_i \notin \text{ref}') \\
&\quad \quad \quad \Rightarrow \\
&\quad \quad \quad \text{inp}(i).x_i \notin \alpha P \setminus a) \\
&\quad \wedge \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad (\exists n : 1..layerSize(layerNo) \bullet \\
&\quad \quad \quad tt = \bigwedge / in : 1..insize \bullet \\
&\quad \quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1..n-1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(layerNo, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(layerNo, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}') \\
&\quad) \\
&\quad \Rightarrow \\
&\quad \exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}') \\
&\quad \quad \quad \vee \\
&\quad \quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge / j : 1..(i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}') \\
&\quad \quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \exists a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out}, i : 1 \dots \text{insize} \mid p \bullet \\
&\quad \quad \quad (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \wedge \\
&\quad \quad \quad (\text{inp}(i).x_i \notin \text{ref}') \\
&\quad \quad \Rightarrow \\
&\quad \quad \quad \text{inp}(i).x_i \notin \alpha P \setminus a) \\
&\wedge \\
&\forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \wedge \\
&\quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}') \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}') \\
&\quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

[all elements in $I \cup O$, so $\text{seqapprox}(\epsilon)$ is reflexive, Lemma B.4]

$$\begin{aligned}
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[rearrange notation, add predicate to universal quantifier]} \\
&\quad \exists a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out}, i : 1..insize \mid p \wedge \text{inp}(i).x_i \notin \text{ref}' \bullet \\
&\quad \quad \quad (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \alpha P \setminus a \\
&\quad \wedge \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \quad (\exists n : 1..layerSize(layerNo) \bullet \\
&\quad \quad \quad tt = \bigwedge / in : 1..insize \bullet \\
&\quad \quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1..n-1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(layerNo, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \text{out}(n).\text{annoutput}(layerNo, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad \quad \quad) \\
&\quad \Rightarrow \\
&\quad \exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1..(i-1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \quad \vee \\
&\quad \quad \quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \quad \quad \bigwedge / j : 1..(i-1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad \quad \quad \quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[select a witness for } a: \{ \text{inp}(i).x_i \} \text{]} \\
&\quad \exists y_1, \dots, y_{\#out}, i : 1 \dots \text{insize} \mid p \wedge \text{inp}(i).x_i \notin \text{ref}' \bullet \\
&\quad (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) \{ \text{inp}(i).x_i \} \wedge \\
&\quad \text{inp}(i).x_i \notin \alpha P \setminus \{ \text{inp}(i).x_i \} \\
&\quad \wedge \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1 \dots \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\quad \Rightarrow \\
&\quad \exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet && \text{[set theory]} \\
&\quad \exists y_1, \dots, y_{\#out}, i : 1.. \text{insize} \mid p \wedge \text{inp}(i).x_i \notin \text{ref}' \bullet \\
&\quad (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) \{ \text{inp}(i).x_i \} \\
&\quad \wedge \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists n : 1.. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1.. \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1.. n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \wedge \\
&\quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \exists y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1.. (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1.. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}'] \\
&\Leftarrow \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet && \text{[strengthen quantifier on } y \text{ variables and } i] \\
&\quad \forall y_1, \dots, y_{\#out}, i : 1.. \text{insize} \mid p \wedge \text{inp}(i).x_i \notin \text{ref}' \bullet \\
&\quad (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) \{ \text{inp}(i).x_i \} \\
&\quad \wedge \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists n : 1.. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1.. \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1.. n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \wedge \\
&\quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1.. (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1.. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a / tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \neg \exists \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[double negation on first conjunct]} \\
&\quad \exists y_1, \dots, y_{\#out}, i : 1.. \text{insize} \mid p \wedge \text{inp}(i).x_i \notin \text{ref}' \bullet \\
&\quad \neg (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) \{ \text{inp}(i).x_i \} \\
&\quad \wedge \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists n : 1.. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1.. \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1.. n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \wedge \\
&\quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1.. (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1.. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad) [s, \alpha P \setminus a/tt, \text{ref}'] \\
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet \quad \text{[select a witness for ref': } \alpha P \setminus \{ \text{inp}(i).x_i \}] \\
&\quad \exists y_1, \dots, y_{\#out}, i : 1.. \text{insize} \mid p \wedge \text{inp}(i).x_i \notin \alpha P \setminus \{ \text{inp}(i).x_i \} \bullet \\
&\quad \neg (\alpha P \setminus \alpha P \setminus \{ \text{inp}(i).x_i \}) \text{setapprox}(\epsilon) \{ \text{inp}(i).x_i \} \\
&\quad \wedge \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists n : 1.. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1.. \text{insize} \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1.. n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \wedge \\
&\quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\exists s; a \mid tt \text{seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1.. (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \vee \\
&\quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \bigwedge / j : 1.. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \text{true} \wedge \quad \text{[logic]} \\
&\quad \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \\
&\quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1..insize \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom } \text{inp} \bullet tt = \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \quad \vee \\
&\quad \quad \quad \quad \exists i : \text{dom } \text{out} \bullet tt = \bigwedge / n : \text{dom } \text{inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad \quad) [s, \alpha P \setminus a/tt, \text{ref}'] \\
&= \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \quad \text{[logic]} \\
&\quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad tt = \bigwedge / in : 1..insize \bullet \\
&\quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \wedge \\
&\quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad) \\
&\Rightarrow \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (\exists i : \text{dom } \text{inp} \bullet tt = \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \quad \vee \\
&\quad \quad \quad \quad \exists i : \text{dom } \text{out} \bullet tt = \bigwedge / n : \text{dom } \text{inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad \quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet && \text{[extend scope of } s, a \text{ and } y\text{]} \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad \quad tt = \bigwedge / in : 1..insize \bullet \\
&\quad \quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad \quad \quad \quad) \\
&\quad \quad \Rightarrow \\
&\quad \quad (\exists i : \text{dom inp} \bullet tt = \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{inp}(j).x_j \rangle \wedge \\
&\quad \quad \quad \quad \text{inp}(i).x_i \notin \text{ref}' \\
&\quad \quad \quad \quad \vee \\
&\quad \quad \quad \quad \exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad \quad \quad) [s, \alpha P \setminus a/tt, \text{ref}'] \\
&\Leftrightarrow \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet && \text{[stronger condition, disjunction laws]} \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists n : 1 \dots \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad \quad tt = \bigwedge / in : 1..insize \bullet \\
&\quad \quad \quad \quad \langle \text{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \bigwedge / p_n : 1 \dots n - 1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(p_n).\text{annoutput}(\text{layerNo}, p_n, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \quad \wedge \\
&\quad \quad \quad \quad \text{out}(n).\text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad \quad \quad \quad) \\
&\quad \quad \Rightarrow \\
&\quad \quad (\exists i : \text{dom out} \bullet tt = \bigwedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad \quad \quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \quad [\text{rename } n \text{ to } i, p_n \text{ to } j, \text{ and } in \text{ to } n] \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\exists i : 1 .. \text{layerSize}(\text{layerNo}) \bullet \\
&\quad \quad \quad tt = \widehat{\wedge} / n : 1 .. \text{insize} \bullet \\
&\quad \quad \quad \quad \langle \text{inp}(n).x_n \rangle \\
&\quad \quad \quad \quad \widehat{\wedge} \\
&\quad \quad \quad \quad \widehat{\wedge} / j : 1 .. i - 1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(j).\text{annoutput}(\text{layerNo}, j, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad \quad) \\
&\quad \Rightarrow \\
&\quad \quad (\exists i : \text{dom out} \bullet tt = \widehat{\wedge} / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \widehat{\wedge} \\
&\quad \quad \quad \widehat{\wedge} / j : 1 .. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad) [s, \alpha P \setminus a/tt, \text{ref}'] \\
&\Leftarrow \forall \text{ref}', tr, tr', tt, x_1, \dots, x_{\text{insize}} \bullet \quad [\text{extend scope of } i, \text{layerSize}(\text{layerNo}) = \text{outside}] \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \forall i : 1 .. \text{outside} \bullet \\
&\quad \quad \quad (tt = \widehat{\wedge} / n : 1 .. \text{insize} \bullet \\
&\quad \quad \quad \quad \langle \text{inp}(n).x_n \rangle \\
&\quad \quad \quad \quad \widehat{\wedge} \\
&\quad \quad \quad \quad \widehat{\wedge} / j : 1 .. i - 1 \bullet \\
&\quad \quad \quad \quad \quad \langle \text{out}(j).\text{annoutput}(\text{layerNo}, j, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \\
&\quad \quad \quad \Rightarrow \\
&\quad \quad \quad (tt = \widehat{\wedge} / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \widehat{\wedge} \\
&\quad \quad \quad \quad \widehat{\wedge} / j : 1 .. (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \text{out}(i).y_i \notin \text{ref}' \\
&\quad \quad \quad) [s, \alpha P \setminus a/tt, \text{ref}']
\end{aligned}$$

$$\begin{aligned}
&= \forall ref', tr, tr', tt, x_1, \dots, x_{insize} \bullet && \text{[Apply } s \text{ and } \alpha P \text{ substitutions]} \\
&\quad \exists s; a \mid tt \text{ seqapprox}(\epsilon) s \wedge (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \forall i : 1 \dots outside \bullet \\
&\quad \quad \quad (tt = \bigwedge / n : 1..insize \bullet \\
&\quad \quad \quad \quad \langle inp(n).x_n \rangle \\
&\quad \quad \quad \quad \bigwedge \\
&\quad \quad \quad \quad \bigwedge / j : 1 \dots i - 1 \bullet \\
&\quad \quad \quad \quad \langle out(j).annoutput(layerNo, j, \langle x_1, \dots, x_{insize} \rangle) \rangle) \\
&\quad \quad \quad \wedge \\
&\quad \quad \quad out(i).annoutput(layerNo, i, \langle x_1, \dots, x_{insize} \rangle) \notin ref' \\
&\quad \quad \quad \Rightarrow \\
&\quad \quad \quad (s = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle out(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad out(i).y_i \notin \alpha P \setminus a \\
&\quad \quad \quad) \\
&= \forall ref', x_1, \dots, x_{insize} \bullet && \text{[one-point } tt] \\
&\quad \exists s; a \mid (\bigwedge / n : 1..insize \bullet \langle inp(n).x_n \rangle \bigwedge \\
&\quad \quad \bigwedge / j : 1 \dots i - 1 \bullet \langle out(j).annoutput(layerNo, j, \langle x_1, \dots, x_{insize} \rangle) \rangle) \\
&\quad \quad \text{seqapprox}(\epsilon) s \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad \forall i : 1 \dots outside \bullet \\
&\quad \quad \quad \quad (out(i).annoutput(layerNo, i, \langle x_1, \dots, x_{insize} \rangle) \notin ref' \\
&\quad \quad \quad \quad \Rightarrow \\
&\quad \quad \quad \quad (s = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle out(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad \quad out(i).y_i \notin \alpha P \setminus a \\
&\quad \quad \quad \quad) \\
&\quad \quad \quad) \\
&= \forall ref', x_1, \dots, x_{insize} \bullet && \text{[rearrange quantifiers and add predicate]} \\
&\quad \exists s; a \mid (\bigwedge / n : 1..insize \bullet \langle inp(n).x_n \rangle \bigwedge \\
&\quad \quad \bigwedge / j : 1 \dots i - 1 \bullet \langle out(j).annoutput(layerNo, j, \langle x_1, \dots, x_{insize} \rangle) \rangle) \\
&\quad \quad \text{seqapprox}(\epsilon) s \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus ref') \text{ setapprox}(\epsilon) a \bullet \\
&\quad \quad \forall y_1, \dots, y_{\#out}; i : 1 \dots outside \mid \\
&\quad \quad \quad p \wedge out(i).annoutput(layerNo, i, \langle x_1, \dots, x_{insize} \rangle) \notin ref' \bullet \\
&\quad \quad \quad s = \bigwedge / n : \text{dom } inp \bullet \langle inp(n).x_n \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge / j : 1 \dots (i - 1) \bullet \langle out(j).y_j \rangle \wedge \\
&\quad \quad \quad \quad out(i).y_i \notin \alpha P \setminus a
\end{aligned}$$

$$\begin{aligned}
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[propagate conditions of } s \text{ and } a\text{]} \\
&\quad \exists s; a \bullet \\
&\quad \forall y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid \\
&\quad \quad p \wedge \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \bullet \\
&\quad \quad (\wedge / n : 1.. \text{insize} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \wedge / j : 1 \dots i - 1 \bullet \langle \text{out}(j).\text{annoutput}(\text{layerNo}, j, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \quad \text{seqapprox}(\epsilon) s \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \wedge \\
&\quad \quad s = \wedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \quad \wedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \text{out}(i).y_i \notin \alpha P \setminus a \\
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[one-point } s\text{]} \\
&\quad \exists a \bullet \\
&\quad \forall y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid \\
&\quad \quad p \wedge \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \bullet \\
&\quad \quad (\wedge / n : 1.. \text{insize} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \wedge / j : 1 \dots i - 1 \bullet \langle \text{out}(j).\text{annoutput}(\text{layerNo}, j, \langle x_1, \dots, x_{\text{insize}} \rangle) \rangle) \\
&\quad \quad \text{seqapprox}(\epsilon) \\
&\quad \quad \wedge / n : \text{dom inp} \bullet \langle \text{inp}(n).x_n \rangle \wedge \\
&\quad \quad \wedge / j : 1 \dots (i - 1) \bullet \langle \text{out}(j).y_j \rangle \wedge \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \wedge \\
&\quad \quad \text{out}(i).y_i \notin \alpha P \setminus a \\
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[using reflexivity of } \text{seqapprox}, \text{ inputs identical, and output lemma B.5]} \\
&\quad \exists a \bullet \\
&\quad \forall y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid \\
&\quad \quad p \wedge \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \bullet \\
&\quad \quad \forall n : 1 \dots i - 1 \bullet \\
&\quad \quad \quad \left| \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n \right| < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) a \wedge \\
&\quad \quad \text{out}(i).y_i \notin \alpha P \setminus a \\
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[select witness for } a: \{ \text{out}(i).y_i \}\text{]} \\
&\quad \forall y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \wedge \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \bullet \\
&\quad \quad \forall n : 1 \dots i - 1 \bullet \\
&\quad \quad \quad \left| \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n \right| < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) \{ \text{out}(i).y_i \} \wedge \\
&\quad \quad \text{out}(i).y_i \notin \alpha P \setminus \{ \text{out}(i).y_i \} \\
&= \forall \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[set theory]} \\
&\quad \forall y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \wedge \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \bullet \\
&\quad \quad \forall n : 1 \dots i - 1 \bullet \\
&\quad \quad \quad \left| \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n \right| < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) \{ \text{out}(i).y_i \}
\end{aligned}$$

$$\begin{aligned}
&= \neg \exists \text{ref}', x_1, \dots, x_{\text{insize}} \bullet \quad \text{[double negation]} \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \wedge \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \text{ref}' \bullet \\
&\quad \neg (\forall n : 1 \dots i - 1 \bullet \\
&\quad \quad | \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n | < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus \text{ref}') \text{ setapprox}(\epsilon) \{ \text{out}(i).y_i \} \\
&\quad \quad) \\
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \wedge \\
&\quad \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \notin \\
&\quad \alpha P \setminus \{ \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \bullet \\
&\quad \neg (\forall n : 1 \dots i - 1 \bullet \\
&\quad \quad | \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n | < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad (\alpha P \setminus \\
&\quad \quad \quad \alpha P \setminus \{ \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \\
&\quad \quad \quad \text{setapprox}(\epsilon) \\
&\quad \quad \quad \{ \text{out}(i).y_i \} \\
&\quad \quad) \\
&\quad \quad \text{[select a witness for } \text{ref}' : \alpha P \setminus \{ \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \} \\
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet \quad \text{[set theory]} \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \bullet \\
&\quad \neg (\forall n : 1 \dots i - 1 \bullet \\
&\quad \quad | \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n | < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad \{ \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \text{ setapprox}(\epsilon) \{ \text{out}(i).y_i \} \\
&\quad \quad) \\
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet \quad \text{[output event comparison, using Lemma B.7]} \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \bullet \\
&\quad \neg (\forall n : 1 \dots i - 1 \bullet \\
&\quad \quad | \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n | < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad \forall e : \{ \text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \bullet \\
&\quad \quad \quad \exists e1 : \{ \text{out}(i).y_i \} \bullet | \text{value}(e) - \text{value}(e1) | < \epsilon \\
&\quad \quad) \\
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet \quad \text{[singleton sets]} \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \bullet \\
&\quad \neg (\forall n : 1 \dots i - 1 \bullet \\
&\quad \quad | \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n | < \epsilon \\
&\quad \quad \wedge \\
&\quad \quad | \text{value}(\text{out}(i).\text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle)) - \text{value}(\text{out}(i).y_i) | < \epsilon \\
&\quad \quad)
\end{aligned}$$

$$\begin{aligned}
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet && \text{[evaluate value]} \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \bullet \\
&\quad \neg \left(\forall n : 1 \dots i - 1 \bullet \right. \\
&\quad \quad \left. \left| \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n \right| < \epsilon \right. \\
&\quad \quad \wedge \\
&\quad \quad \left. \left| \text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_i \right| < \epsilon \right. \\
&\quad \quad \left. \right) \\
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet && \text{[combining indicies]} \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \bullet \\
&\quad \neg \left(\forall n : 1 \dots i \bullet \right. \\
&\quad \quad \left. \left| \text{annoutput}(\text{layerNo}, n, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_n \right| < \epsilon \right. \\
&\quad \quad \left. \right) \\
&= \neg \exists x_1, \dots, x_{\text{insize}} \bullet && \text{[equivalent condition, } n \text{ not needed]} \\
&\quad \exists y_1, \dots, y_{\#out}; i : 1 \dots \text{outside} \mid p \bullet \\
&\quad \neg \left| \text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) - y_i \right| < \epsilon \\
&= \neg \exists x_1, \dots, x_{\text{insize}} : \text{Value} \bullet \exists y_1, \dots, y_{\text{outside}} : \text{Value} \mid p \bullet \exists i : 1 \dots \text{outside} \bullet \\
&\quad \{ \text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \cap \{ x : \mathbb{R} \bullet |x - y_i| < \epsilon \} = \emptyset \\
&\quad \text{[equivalent condition, in set reachability form (and } \#out = \text{outside)}]
\end{aligned}$$

□

Lemma C.2.

$$\begin{aligned}
&\neg \exists x_1, \dots, x_{\text{insize}} : \text{Value} \bullet \exists y_1, \dots, y_{\text{outside}} : \text{Value} \mid p \bullet \exists i : 1 \dots \text{outside} \bullet \\
&\quad \{ \text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{\text{insize}} \rangle) \} \cap \{ x : \mathbb{R} \mid |x - y_i| < \epsilon \} = \emptyset \\
&\Rightarrow [(Q_3 \setminus_{\text{post}} \text{ANNHiddenEvs})[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet P_3[s/tt]]
\end{aligned}$$

Proof.

$$\begin{aligned}
&[Q_3 \setminus_{\text{post}} \text{ANNHiddenEvs}[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet P_3[s/tt]] \\
&= \forall st, st', wait, wait', ref, ref', ok, ok', tr, tr', tt \bullet && \text{[unfolding universal closure]} \\
&\quad Q_3 \setminus_{\text{post}} \text{ANNHiddenEvs}[\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet P_3[s/tt] \\
&= \forall st, st', wait, wait', ref, ref', ok, ok', tr, tr', tt \bullet \\
&\quad (ok \wedge ok' \wedge \neg wait \wedge \neg wait' \wedge st' = st \wedge \\
&\quad \#input = \text{insize} \wedge \\
&\quad \Phi[\text{layeroutput}(\text{layerNo}, \text{layerSize}(\text{layerNo})) \text{ shide } \text{ANNHiddenEvs} \\
&\quad][\text{inp}/\text{layerRes}.0, \text{out}/\text{layerRes}.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet P_3[s/tt] \\
&\quad \text{[unfolding } Q_3: \text{ postcondition extraction } (_3), \text{ Def. B.11 and Definition B.8]}
\end{aligned}$$

$$\begin{aligned}
&= \forall st, st', wait, wait', ref, ref', ok, ok', tr, tr', tt \bullet && \text{[unfolding } \Phi, \text{ Definition B.14]} \\
&\quad (ok \wedge ok' \wedge \neg wait \wedge \neg wait' \wedge st' = st \wedge \\
&\quad \#input = insize \wedge \\
&\quad tt = layeroutput(layerNo, layerSize(layerNo)) \text{ shide } ANNHIDDENEVTS \\
&\quad) [inp/layerRes.0, out/layerRes.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet P_3[s/tt] \\
&= \forall st, st', wait, wait', ref, ref', ok, ok', tr, tr', tt \bullet && \text{[unfolding } P_3: \text{ Definitions 4.4 and B.11]} \\
&\quad (ok \wedge ok' \wedge \neg wait \wedge \neg wait' \wedge st' = st \wedge \\
&\quad \#input = insize \wedge \\
&\quad tt = layeroutput(layerNo, layerSize(layerNo)) \text{ shide } ANNHIDDENEVTS \\
&\quad) [inp/layerRes.0, out/layerRes.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet \\
&\quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet (ok \wedge ok' \wedge \neg wait \wedge \neg wait' \wedge st' = st \wedge \\
&\quad \quad \quad tt = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle) [s/tt] \\
&= \forall st', ref, ref', tr, tr', tt \bullet && \text{[one-point rule on } ok, ok', wait, wait' \text{ and } st]} \\
&\quad (\#input = insize \wedge \\
&\quad tt = layeroutput(layerNo, layerSize(layerNo)) \text{ shide } ANNHIDDENEVTS \\
&\quad) [inp/layerRes.0, out/layerRes.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet \\
&\quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet st' = st' \wedge \\
&\quad \quad \quad (tt = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle) [s/tt] \\
&= \forall ref, ref', tr, tr', tt \bullet && \text{[reflexivity of equality]} \\
&\quad (\#input = insize \wedge \\
&\quad tt = layeroutput(layerNo, layerSize(layerNo)) \text{ shide } ANNHIDDENEVTS \\
&\quad) [inp/layerRes.0, out/layerRes.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet \\
&\quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet (tt = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle) [s/tt] \\
&= \forall tr, tr', tt \bullet && \text{[no references to } ref \text{ or } ref' \text{ in postcondition (CRF healthiness condition [30])]} \\
&\quad (\#input = insize \wedge \\
&\quad tt = layeroutput(layerNo, layerSize(layerNo)) \text{ shide } ANNHIDDENEVTS \\
&\quad) [inp/layerRes.0, out/layerRes.layerNo] \Rightarrow \\
&\quad \exists s \mid tt \text{ seqapprox}(\epsilon) s \bullet \\
&\quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet (tt = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle) [s/tt]
\end{aligned}$$

$$\begin{aligned}
&= \forall tr, tr', tt \bullet \quad \text{[evaluate } \textit{shide} \textit{ ANNHIDDENEVTS}] \\
&\quad (\exists x_1, \dots, x_{\textit{insize}} : \textit{Value} \bullet tt = \bigwedge / in : 1 .. \textit{insize} \bullet \langle \textit{layerRes}.0.in.x_{in} \rangle \\
&\quad \quad \quad \bigwedge / p_{-n} : 1 .. \textit{layerSize}(\textit{layerNo}) \bullet \\
&\quad \quad \quad \langle \textit{layerRes}.\textit{layerNo}.p_{-n}.\textit{annoutput}(\textit{layerNo}, p_{-n}, \\
&\quad \quad \quad \langle x_1, \dots, x_{\textit{insize}} \rangle \rangle \\
&\quad \quad \quad) [\textit{inp}/\textit{layerRes}.0, \textit{out}/\textit{layerRes}.\textit{layerNo}] \Rightarrow \\
&\quad \quad \quad \exists s \mid tt \textit{seqapprox}(\epsilon) s \bullet \\
&\quad \quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet (tt = \bigwedge / i : \textit{dom} \textit{inp} \bullet \langle \textit{inp}(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \textit{dom} \textit{out} \bullet \langle \textit{out}(j).y_j \rangle) [s/tt] \\
&= \forall tr, tr', tt \bullet \quad \text{[applying } [\textit{inp}/\textit{layerRes}.0, \textit{out}/\textit{layerRes}.\textit{layerNo}]] \\
&\quad (\exists x_1, \dots, x_{\textit{insize}} : \textit{Value} \bullet tt = \bigwedge / in : 1 .. \textit{insize} \bullet \langle \textit{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge / p_{-n} : 1 .. \textit{layerSize}(\textit{layerNo}) \bullet \\
&\quad \quad \quad \langle \textit{out}(p_{-n}).\textit{annoutput}(\textit{layerNo}, p_{-n}, \langle x_1, \dots, x_{\textit{insize}} \rangle) \rangle \\
&\quad \quad \quad) \Rightarrow \\
&\quad \quad \quad \exists s \mid tt \textit{seqapprox}(\epsilon) s \bullet \\
&\quad \quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet (tt = \bigwedge / i : \textit{dom} \textit{inp} \bullet \langle \textit{inp}(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \textit{dom} \textit{out} \bullet \langle \textit{out}(j).y_j \rangle) [s/tt] \\
&= \forall tr, tr', tt \bullet \quad \text{[applying } [s/tt]] \\
&\quad (\exists x_1, \dots, x_{\textit{insize}} : \textit{Value} \bullet tt = \bigwedge / in : 1 .. \textit{insize} \bullet \langle \textit{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge / p_{-n} : 1 .. \textit{layerSize}(\textit{layerNo}) \bullet \\
&\quad \quad \quad \langle \textit{out}(p_{-n}).\textit{annoutput}(\textit{layerNo}, p_{-n}, \langle x_1, \dots, x_{\textit{insize}} \rangle) \rangle \\
&\quad \quad \quad) \Rightarrow \\
&\quad \quad \quad \exists s \mid tt \textit{seqapprox}(\epsilon) s \bullet \\
&\quad \quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet (s = \bigwedge / i : \textit{dom} \textit{inp} \bullet \langle \textit{inp}(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \textit{dom} \textit{out} \bullet \langle \textit{out}(j).y_j \rangle) \\
&= \forall tr, tr', tt \bullet \quad \text{[propagate condition of } s] \\
&\quad (\exists x_1, \dots, x_{\textit{insize}} : \textit{Value} \bullet tt = \bigwedge / in : 1 .. \textit{insize} \bullet \langle \textit{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge / p_{-n} : 1 .. \textit{layerSize}(\textit{layerNo}) \bullet \\
&\quad \quad \quad \langle \textit{out}(p_{-n}).\textit{annoutput}(\textit{layerNo}, p_{-n}, \langle x_1, \dots, x_{\textit{insize}} \rangle) \rangle \\
&\quad \quad \quad) \Rightarrow \\
&\quad \quad \quad \exists s \bullet \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet tt \textit{seqapprox}(\epsilon) s \wedge \\
&\quad \quad \quad (s = \bigwedge / i : \textit{dom} \textit{inp} \bullet \langle \textit{inp}(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \textit{dom} \textit{out} \bullet \langle \textit{out}(j).y_j \rangle) \\
&= \forall tr, tr', tt \bullet \quad \text{[one-point rule on } s] \\
&\quad (\exists x_1, \dots, x_{\textit{insize}} : \textit{Value} \bullet tt = \bigwedge / in : 1 .. \textit{insize} \bullet \langle \textit{inp}(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge / p_{-n} : 1 .. \textit{layerSize}(\textit{layerNo}) \bullet \\
&\quad \quad \quad \langle \textit{out}(p_{-n}).\textit{annoutput}(\textit{layerNo}, p_{-n}, \langle x_1, \dots, x_{\textit{insize}} \rangle) \rangle \\
&\quad \quad \quad) \Rightarrow \\
&\quad \quad \quad \exists x_1, \dots, x_{\#inp}; y_1, \dots, y_{\#out} \mid p \bullet tt \textit{seqapprox}(\epsilon) \bigwedge / i : \textit{dom} \textit{inp} \bullet \langle \textit{inp}(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \textit{dom} \textit{out} \bullet \langle \textit{out}(j).y_j \rangle
\end{aligned}$$

$$\begin{aligned}
&\Leftarrow \forall tr, tr', tt \bullet && \text{[strengthen condition]} \\
&\quad \forall x_1, \dots, x_{insize} : \text{Value} \bullet \\
&\quad \quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad \quad (tt = \bigwedge_{in : 1 \dots insize} \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \quad \quad \bigwedge_{p_n : 1 \dots layerSize(layerNo)} \bullet \\
&\quad \quad \quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle \rangle) \rangle \\
&\quad \quad \quad \quad \Rightarrow \\
&\quad \quad \quad \quad tt \text{ seqapprox}(\epsilon) \bigwedge_{i : \text{dom } inp} \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \quad \quad \bigwedge_{j : \text{dom } out} \bullet \langle out(j).y_j \rangle \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[one-point rule on } tt] \\
&\quad \forall y_1, \dots, y_{\#out} \mid p \bullet \\
&\quad \quad (\bigwedge_{in : 1 \dots insize} \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge_{p_n : 1 \dots layerSize(layerNo)} \bullet \\
&\quad \quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle \rangle) \rangle \\
&\quad \quad \text{seqapprox}(\epsilon) \\
&\quad \quad \quad (\bigwedge_{i : \text{dom } inp} \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge_{j : \text{dom } out} \bullet \langle out(j).y_j \rangle) \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[outsized} = \#out] \\
&\quad \forall y_1, \dots, y_{outsized} \mid p \bullet \\
&\quad \quad (\bigwedge_{in : 1 \dots insize} \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge_{p_n : 1 \dots layerSize(layerNo)} \bullet \\
&\quad \quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle \rangle) \rangle \\
&\quad \quad \text{seqapprox}(\epsilon) \\
&\quad \quad \quad (\bigwedge_{i : \text{dom } inp} \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \quad \bigwedge_{j : \text{dom } out} \bullet \langle out(j).y_j \rangle) \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[introduce bound variables for sequence expressions for clarity]} \\
&\quad \forall y_1, \dots, y_{outsized} \mid p \bullet s_1 \text{ seqapprox}(\epsilon) \ t \\
&\quad \hline
&\quad s_1 = \bigwedge_{in : 1 \dots insize} \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \bigwedge_{p_n : 1 \dots layerSize(layerNo)} \bullet \\
&\quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle \rangle) \rangle \\
&\quad t = \bigwedge_{i : \text{dom } inp} \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \bigwedge_{j : \text{dom } out} \bullet \langle out(j).y_j \rangle
\end{aligned}$$

$$\begin{aligned}
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[unfold definition of } seqapprox(\epsilon)\text{]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \forall i : \text{dom } s_2 \bullet \\
&\quad \quad (s_2(i) \in I \Rightarrow t(i) = s_2(i)) \wedge \\
&\quad \quad (s_2(i) \in O \Rightarrow ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet value(t(i)) = value(s_2(i)) + r) \\
&\quad \hline
&\quad s_1 = \bigwedge / in : 1 .. insize \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \bigwedge / p_n : 1 .. layerSize(layerNo) \bullet \\
&\quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&\quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad s_2 = s_1 \upharpoonright (I \cup O) \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[unfold } s_2 \text{ (} s_1 \text{ no longer used)}\text{]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \forall i : \text{dom } s_2 \bullet \\
&\quad \quad (s_2(i) \in I \Rightarrow t(i) = s_2(i)) \wedge \\
&\quad \quad (s_2(i) \in O \Rightarrow ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet value(t(i)) = value(s_2(i)) + r) \\
&\quad \hline
&\quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad s_2 = \bigwedge / in : 1 .. insize \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \bigwedge / p_n : 1 .. layerSize(layerNo) \bullet \\
&\quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[distribute universal quantifier of } i\text{]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : \text{dom } s_2 \bullet (s_2(i) \in I \Rightarrow t(i) = s_2(i)) \wedge \\
&\quad \quad \forall i : \text{dom } s_2 \bullet (s_2(i) \in O \Rightarrow ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet value(t(i)) = value(s_2(i)) + r) \\
&\quad \hline
&\quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad s_2 = \bigwedge / in : 1 .. insize \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \bigwedge / p_n : 1 .. layerSize(layerNo) \bullet \\
&\quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle
\end{aligned}$$

$$\begin{aligned}
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet \quad [\text{implication in a universal quantifier}] \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : \text{dom } s_2 \mid s_2(i) \in I \bullet t(i) = s_2(i) \wedge \\
&\quad \quad \forall i : \text{dom } s_2 \mid s_2(i) \in O \bullet ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet value(t(i)) = value(s_2(i)) + r \\
&\quad \quad \quad \text{---} \\
&\quad \quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad \quad s_2 = \bigwedge / in : 1 .. insize \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge / p_n : 1 .. layerSize(layerNo) \bullet \\
&\quad \quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet \quad [\text{separate into the indices of dom } s_2 \text{ such that their constraint holds}] \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : 1 .. insize \bullet t(i) = s_2(i) \wedge \\
&\quad \quad \forall i : insize + 1 .. outside \bullet ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet value(t(i)) = value(s_2(i)) + r \\
&\quad \quad \quad \text{---} \\
&\quad \quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad \quad s_2 = \bigwedge / in : 1 .. insize \bullet \langle inp(in).x_{in} \rangle \\
&\quad \quad \quad \bigwedge / p_n : 1 .. layerSize(layerNo) \bullet \\
&\quad \quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet \quad [layerSize(layerNo) = outside, \text{ and rename } in \text{ to } i] \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : 1 .. insize \bullet t(i) = s_2(i) \wedge \\
&\quad \quad \forall i : insize + 1 .. outside \bullet ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet value(t(i)) = value(s_2(i)) + r \\
&\quad \quad \quad \text{---} \\
&\quad \quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad \quad s_2 = \bigwedge / i : 1 .. insize \bullet \langle inp(i).x_i \rangle \\
&\quad \quad \quad \bigwedge / p_n : 1 .. outside \bullet \\
&\quad \quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet \quad [\text{replace } t(i) \text{ and } s_2(i) \text{ with their values in the first conjunct}] \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : 1 .. insize \bullet inp(i).x_i = inp(i).x_i \wedge \\
&\quad \quad \forall i : insize + 1 .. outside \bullet ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet value(t(i)) = value(s_2(i)) + r \\
&\quad \quad \quad \text{---} \\
&\quad \quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad \quad s_2 = \bigwedge / i : 1 .. insize \bullet \langle inp(i).x_i \rangle \\
&\quad \quad \quad \bigwedge / p_n : 1 .. outside \bullet \\
&\quad \quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle
\end{aligned}$$

$$\begin{aligned}
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[equality]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : insize + 1 .. outside \bullet \quad ev(t(i)) = ev(s_2(i)) \wedge \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet \quad value(t(i)) = value(s_2(i)) + r \\
&\quad \overline{\quad} \\
&\quad t = \bigwedge / i : \text{dom } inp \bullet \langle inp(i).x_i \rangle \bigwedge \\
&\quad \quad \bigwedge / j : \text{dom } out \bullet \langle out(j).y_j \rangle \\
&\quad s_2 = \bigwedge / i : 1 .. insize \bullet \langle inp(i).x_i \rangle \\
&\quad \quad \bigwedge / p_n : 1 .. outside \bullet \\
&\quad \quad \quad \langle out(p_n).annoutput(layerNo, p_n, \langle x_1, \dots, x_{insize} \rangle) \rangle \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[replace } t(i) \text{ and } s_2(i) \text{ in the output constraint]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : insize + 1 .. outside \bullet \\
&\quad \quad \quad ev(out(i - insize).y_{i-insize}) = \\
&\quad \quad \quad \quad ev(out(i - insize).annoutput(layerNo, (i - insize), \langle x_1, \dots, x_{insize} \rangle)) \wedge \\
&\quad \quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet \quad value(out(i - insize).y_{i-insize}) = \\
&\quad \quad \quad \quad \quad value(out(i - insize).annoutput(layerNo, (i - insize), \langle x_1, \dots, x_{insize} \rangle)) + r \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[evaluate event and value functions]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : insize + 1 .. outside \bullet \\
&\quad \quad \quad out(i - insize) = out(i - insize) \wedge \\
&\quad \quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet \quad y_{i-insize} = annoutput(layerNo, (i - insize), \langle x_1, \dots, x_{insize} \rangle) + r \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[equality]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : insize + 1 .. outside \bullet \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet \quad y_{i-insize} = annoutput(layerNo, (i - insize), \langle x_1, \dots, x_{insize} \rangle) + r \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[index rearranging]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : 1 .. outside \bullet \\
&\quad \quad \quad \exists r : (-\epsilon, \epsilon) \bullet \quad y_i = annoutput(layerNo, (i), \langle x_1, \dots, x_{insize} \rangle) + r \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[open range definition]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : 1 .. outside \bullet \\
&\quad \quad \quad \exists r : \mathbb{R} \mid -\epsilon < r < \epsilon \bullet \quad y_i = annoutput(layerNo, (i), \langle x_1, \dots, x_{insize} \rangle) + r \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[existential quantification laws]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : 1 .. outside \bullet \\
&\quad \quad \quad \exists r : \mathbb{R} \bullet \quad y_i = annoutput(layerNo, (i), \langle x_1, \dots, x_{insize} \rangle) + r \wedge \\
&\quad \quad \quad \quad -\epsilon < r < \epsilon \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[arithmetic]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \quad \forall i : 1 .. outside \bullet \\
&\quad \quad \quad \exists r : \mathbb{R} \bullet \quad r = y_i - annoutput(layerNo, (i), \langle x_1, \dots, x_{insize} \rangle) \wedge \\
&\quad \quad \quad \quad -\epsilon < r < \epsilon
\end{aligned}$$

$$\begin{aligned}
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[one-point rule on } r\text{]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \forall i : 1 \dots outside \bullet \\
&\quad \quad -\epsilon < y_i - \text{annoutput}(\text{layerNo}, (i), \langle x_1, \dots, x_{insize} \rangle) < \epsilon \\
&= \forall x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[arithmetic]} \\
&\quad \forall y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \forall i : 1 \dots outside \bullet \\
&\quad \quad |y_i - \text{annoutput}(\text{layerNo}, (i), \langle x_1, \dots, x_{insize} \rangle)| < \epsilon \\
&= \neg \exists x_1, \dots, x_{insize} : \text{Value} \bullet && \text{[double negate condition]} \\
&\quad \exists y_1, \dots, y_{outside} \mid p \bullet \\
&\quad \exists i : 1 \dots outside \bullet \\
&\quad \quad \neg |y_i - \text{annoutput}(\text{layerNo}, (i), \langle x_1, \dots, x_{insize} \rangle)| < \epsilon \\
&= \neg \exists x_1, \dots, x_{insize} : \text{Value} \bullet \exists y_1, \dots, y_{outside} \mid p \bullet \exists i : 1 \dots outside \bullet && \text{[set theory]} \\
&\quad \{ \text{annoutput}(\text{layerNo}, i, \langle x_1, \dots, x_{insize} \rangle) \} \cap \{ x : \mathbb{R} \mid |x - y_i| < \epsilon \} = \emptyset
\end{aligned}$$

□

List of References

- [1] ETH Robustness Analyzer for Neural Networks (ERAN). github.com/eth-sri/eran, accessed 14/04/20.
- [2] Marabou. github.com/NeuralNetworkVerification/Marabou, accessed 15/04/20.
- [3] Matlab Toolbox for Neural Network Verification (NNV). github.com/verivital/nnv, accessed 13/04/20.
- [4] Neuralverification.jl. github.com/sisl/NeuralVerification.jl, accessed 14/04/20.
- [5] Nnet repository. github.com/sisl/NNet, accessed 14/04/20.
- [6] NNVMT: A Translation Tool for Feedforward Neural Network Models. github.com/verivital/nnvmt, accessed 20/04/20.
- [7] Onnx. github.com/onnx/onnx, accessed 14/04/20.
- [8] Reluplex cav 2017. github.com/guykatzz/ReluplexCav2017, accessed 14/04/20.
- [9] Sherlock. github.com/souradeep-111/sherlock, accessed 14/04/20.
- [10] IEEE Standard for Floating-Point Arithmetic, 2019. IEEE 754-2019.
- [11] Jin-Hyeong Ahn, Key Rhee, and Youngjun You. A study on the collision avoidance of a ship using neural networks and fuzzy logic. *Applied Ocean Research*, 37:162–173, 08 2012.
- [12] P.E. An, C.J. Harris, R. Tribe, and N. Clarke. Aspects of neural networks in intelligent collision avoidance systems for prometheus. In *Joint Framework for Information Technology (01/01/93)*, pages 129–135, 1993.
- [13] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In *CCS 2019: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1249–1264, 2019. arXiv:1906.10395.
- [14] James Baxter, Ana Cavalcanti, Madiel Conserva, Alvaro Miyazawa, and Augusto Sampaio. RoboStar demonstrator: a segway. Technical report, University of York, 2021.
- [15] Marco E. Benalcázar. Machine learning for computer vision: a review of theory and algorithms. *Revista Ibérica de Sistemas e Tecnologias de Informação*, 19:608–618, 04 2019.

- [16] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.
- [17] Achim D. Brucker and Amy Stell. Verifying feedforward neural networks for classification in isabelle/hol. In *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*, page 427–444, Berlin, Heidelberg, 2023. Springer-Verlag.
- [18] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A Unified View of Piecewise Linear Neural Network Verification. In *NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 4795–4804, December 2018.
- [19] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks, 2016. arXiv:1608.04644.
- [20] Chih-Hong Cheng, Geog Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *Automated Technology for Verification and Analysis. ATVA 2017*, volume 10482 of *Lecture Notes in Computer Science*. Springer, Cham, 2017.
- [21] Luiz Henrique de Figueiredo and Jorge Stolf. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37:147–158, 2003. doi.org/10.1023/B:NUMA.0000049462.70970.b6.
- [22] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In Itsuki Noda, Noriaki Ando, Davide Brugali, and James J. Kuffner, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [23] Tommaso Dreossi et al. Counterexample-Guided Data Augmentation, 2018. arXiv:1805.06962.
- [24] Guillaume Dupont, Yamine Aït-Ameur, Marc Pantel, and Neeraj K. Singh. Event-b refinement for continuous behaviours approximation. In *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings*, page 320–336, Berlin, Heidelberg, 2021. Springer-Verlag.
- [25] Souradeep Dutta, Susmit Jha, Sriram Sanakaranarayanan, and Ashish Tiwari. Output range analysis for deep neural networks, 2017.
- [26] Rüdiger Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Automated Technology for Verification and Analysis. ATVA 2017*, volume 10482 of *LNCS*. Springer, Cham, 2017.
- [27] Matteo Fischetti and Jason Jo. Deep neural networks as 0-1 mixed integer linear programs: A feasibility study, 2017. arXiv:1712.06174.
- [28] S. Foster, J. Baxter, A. L. C. Cavalcanti, J. C. P. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197, 2020.

- [29] Simon Foster et al. Unifying Theories of Reactive Design Contracts. *CoRR*, abs/1712.10233, 2017.
- [30] Simon Foster et al. Automated verification of reactive and concurrent programs by calculation. *CoRR*, abs/2007.13529, 2020.
- [31] Simon Foster et al. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197:102510, Oct 2020.
- [32] Angela Freitas and Ana Cavalcanti. Automatic translation from circus to java. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*. Springer Berlin Heidelberg, 2006.
- [33] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Scenic: a language for scenario specification and scene generation. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*, 2019.
- [34] T. Gehr, M. Mirman, D. Drachler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2018.
- [35] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples, 2014. arXiv:1412.6572.
- [36] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision, 2015. arXiv:1502.02551.
- [37] C. Heinzemann and R. Lange. vTSL - A Formally Verifiable DSL for Specifying Robot Tasks. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8308–8314, 2018.
- [38] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.
- [39] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall International, January 1985.
- [40] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International, July 1998.
- [41] Gerard Holzmann, Elie Najm, and Ahmed Serhrouchni. SPIN model checking: An introduction. *STTT*, 2:321–327, 03 2000.
- [42] Boyue Caroline Hu et al. If a human can see it, so should your system. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022.
- [43] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *Computer Aided Verification. CAV 2017*.
- [44] Yuval Jacoby, Clark W. Barrett, and Guy Katz. Verifying recurrent neural networks using invariant inference. *CoRR*, abs/2004.02462, 2020.
- [45] Kai Jia and Martin Rinard. Exploiting Verified Neural Networks via Floating Point Numerical Error, 2020. arXiv:2003.03021.

- [46] Chris W. Johnson. The Increasing Risks of Risk Assessment: On the Rise of Artificial Intelligence and Non-Determinism in Safety-Critical Systems. In *Symposium: Safety-Critical Systems Symposium 2018 (SSS'18)*. Safety-Critical Systems Club, 2018.
- [47] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, Mar 2019.
- [48] Guy Katz et al. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. *LNCS*, page 97–117, 2017.
- [49] Guy Katz et al. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Computer Aided Verification, CAV 2019*, volume 11561 of *LNCS*. Springer, Cham, 07 2019.
- [50] S. Khan, F. Khalid, O. Hasan, and J. M. P. Cardoso. Formal verification of a domain specific language for run-time adaptation. In *2018 Annual IEEE International Systems Conference (SysCon)*, pages 1–8, 2018.
- [51] Tinne Hoff Kjeldsen. History of convexity and mathematical programming: Connections and relationships in two episodes of research in pure and applied mathematics of the 20th century. In *Proceedings of the International Congress of Mathematicians 2010 (ICM 2010)*.
- [52] Joost N. Kok, Walter A. Kosters Egbert J. W. Boers, Peter van der Putten, and Mannes Poel. *Artificial Intelligence - Volume 1*, chapter Artificial Intelligence: Definition, Trends, Techniques, and Cases. UNESCO/UOLSS, 2002.
- [53] Krishnamurthy, Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. A Dual Approach to Scalable Verification of Deep Networks, 2018. arXiv:1803.06578.
- [54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [55] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015. doi.org/10.1038/nature14539.
- [56] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2019. arXiv:1903.06758.
- [57] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks, 2017. arXiv:1706.07351.
- [58] Matt Luckcuck, Marie Farrell, Louise Dennis, Clare Dixon, and Michael Fisher. Formal Specification and Verification of Autonomous Robotic Systems: A Survey, 2018. arXiv:1807.00048.
- [59] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Comput. Surv.*, 52(5), September 2019.

- [60] Spyros Makridakis. The Forthcoming Artificial Intelligence (AI) Revolution: Its Impact on Society and Firms. *Futures*, 04 2017.
- [61] Oded Maler. Computing reachable sets: An introduction, 2008.
- [62] Theresa May. PM Response - CST Robotics letter, November 2016. Retrieved from assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/592420/Robotics_automation_and_artificial_intelligence_-_pm_response.pdf on the 22nd of May 2020.
- [63] A. Miyazawa and A. L. C. Cavalcanti. Formal Refinement in SysML. In E. Albert and E. Sekerinski, editors, *11th International Conference on Integrated Formal Methods*, Lecture Notes in Computer Science, pages 155–170. Springer, 2014.
- [64] A. Miyazawa, P. Ribeiro, K. Ye, A. L. C. Cavalcanti, W. Li, J. Timmis, and J. C. P. Woodcock. RoboChart: Modelling, Verification and Simulation for Robotics. Technical report, University of York, Department of Computer Science, York, UK, 2020. Available at www.cs.york.ac.uk/robostar/notations/.
- [65] Alvaro Miyazawa, Ana Cavalcanti, Pedro Ribeiro, Wei Li, Jim Woodcock, and Jon Timmis. RoboChart Reference Manual. Technical report, 2019. barom.org.uk/roboschart/documents/roboschart-reference.pdf.
- [66] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: a simple and accurate method to fool deep neural networks, 2015. arXiv:1511.04599.
- [67] Katta G. Murty and Santosh N. Kabadi. Some NP-complete problems in quadratic and nonlinear programming. *Mathematical Programming*, 39:117–129, 1987. doi:/10.1007/BF02592948.
- [68] Ana Nevers, Ignacio Gonzalez, John Leander, and Raid Karoumi. A New Approach to Damage Detection in Bridges Using Machine Learning. pages 73–84, 01 2018.
- [69] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [70] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A Survey on Domain-Specific Languages in Robotics. In *LNAI, volume 8810*, 10 2014.
- [71] Chigozie Nwankpa et al. Activation Functions: Comparison of trends in Practice and Research for Deep Learning, 2018. arXiv:1811.03378.
- [72] Marcel Vinicius Medeiros Oliveira, Ivan Soares De Medeiros Júnior, and Jim Woodcock. A verified protocol to implement multi-way synchronisation and interleaving in csp. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods*. Springer Berlin Heidelberg, 2013.
- [73] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks against Machine Learning, 2017. arXiv:1602.02697.
- [74] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings, 2015. arXiv:1511.07528.

- [75] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks, 2016. arXiv:1511.04508.
- [76] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks, 2012. arXiv:1211.5063.
- [77] P.D.Austin and P.H.Welch. CSP for Java™ (JCSP) 1.1-rc4 API Specification, 2008. www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/.
- [78] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. *Proceedings of the 26th Symposium on Operating Systems Principles*, Oct 2017.
- [79] Proofpower z reference manual, 2006.
- [80] Chongli Qin, Krishnamurthy, Dvijotham, Brendan O’Donoghue, Rudy Bunel, Robert Stanforth, Sven Gowal, Jonathan Uesato, Grzegorz Swirszcz, and Pushmeet Kohli. Verification of Non-Linear Specifications for Neural Networks, 2019. arXiv:1902.09592.
- [81] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified Defenses against Adversarial Examples, 2018. arXiv:1801.09344.
- [82] Raúl Rojas. *Neural Networks — A Systematic Introduction*, chapter 7. Springer-Verlag, 1996.
- [83] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
- [84] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, 2018.
- [85] Adrian Rutle, Jonas Backer, Kolbein Foldøy, and Robin Bye. CommonLang: A dsl for defining robot tasks. 10 2018.
- [86] Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks. In *NeurIPS 2019*, 2019.
- [87] Bryan Scattergood and Philip Armstrong. CSPM: A Reference Manual, January 2011. www.cs.ox.ac.uk/ucs/cspm.pdf.
- [88] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10802–10813. Curran Associates, Inc., 2018.
- [89] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3:1–30, 01 2019.

- [90] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations (ICLR)*. 2019.
- [91] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliff, 1992.
- [92] S. Sudholt and G. A. Fink. PHOCNet: A Deep Convolutional Neural Network for Word Spotting in Handwritten Documents. In *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 277–282, 2016.
- [93] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2013. arXiv:1312.6199.
- [94] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest. *Proceedings of the 40th International Conference on Software Engineering*, May 2018.
- [95] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR, 2019*.
- [96] Hoang-Dung Tran et al. Star-Based Reachability Analysis of Deep Neural Networks. In *Formal Methods – The Next 30 Years*, pages 670–686. Springer, 2019.
- [97] Hoang-Dung Tran, Patrick Musau, Diego Manzanas Lopez, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Parallelizable Reachability Analysis Algorithms for Feed-Forward Neural Networks. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE '19*, page 31–40. IEEE Press, 2019.
- [98] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 3–17, Cham, 2020. Springer International Publishing.
- [99] Weiming Xiang and Dung Tran and Taylor Johnson. Output reachable set estimation and verification for multi-layer neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, PP, 08 2017.
- [100] University of Oxford. *FDR Manual*, May 2020. Release 4.2.7. Retrieved from d1.cocotec.io/fdr/fdr-manual.pdf, on the 31st of May 2020.
- [101] University of York. *RoboChart Reference Manual*. www.cs.york.ac.uk/circus/RoboCalc/robotool/.
- [102] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*, volume 196 of *International Series in Operations Research & Management Science*. Springer, fourth edition, 2014.
- [103] Sir Mark Walport and Professor Dame Nancy Rothwell. Robotics, automation and artificial intelligence (RAAI), October 2016. Retrieved on the 22nd of May 2020.

- [104] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems 31 (NIPS'18)*, 2018. arXiv:1809.08098.
- [105] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *27th USENIX Security Symposium*, pages 1599–1614, Baltimore, MD, August 2018. USENIX Association.
- [106] Tsui-Wei Weng et al. Towards fast computation of certified robustness for relu networks. In *ICML*, pages 5273–5282, July 2018.
- [107] Eric Wong and J. Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope, 2017. arXiv:1711.00851.
- [108] Jim Woodcock. The miracle of reactive programming. In Andrew Butterfield, editor, *Unifying Theories of Programming*, pages 202–217, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [109] Jim Woodcock and Jim Davies. *Using Z*. Prentice Hall, 1996.
- [110] Min Wu, Matthew Wicker, Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. A game-based approximate verification of deep neural networks with provable guarantees. *Theoretical Computer Science*, 807:298 – 329, 2020. In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part II.
- [111] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Reachable set computation and safety verification for neural networks with relu activations, 2017. arXiv:1712.08163.
- [112] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5777–5783, 2018.
- [113] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Specification-guided safety verification for feedforward neural networks, 2018. arXiv:1812.06161.
- [114] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9:611–629, 2018. doi.org/10.1007/s13244-018-0639-9.
- [115] K. Ye, S. Foster, and J. C. P. Woodcock. Automated reasoning for probabilistic sequential programs with theorem proving. In U. Fahrenberg, M. Gehrke, L. Santocanale, and M. Winter, editors, *Relational and Algebraic Methods in Computer Science*, volume 13027 of *LNCS*, pages 465–482. Springer, 2021.
- [116] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient Neural Network Robustness Certification with General Activation Functions. In *Advances in Neural Information Processing Systems 31*, 2018.
- [117] Huan Zhang, Pengchuan Zhang, and Cho-Jui Hsieh. RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications, 2018. arXiv:1810.11783.