Modelling and Verifying Robotic Software that uses Neural Networks

Ziggy Attala, Ana Cavalcanti, and Jim Woodcock

University of York, UK

Abstract. Verifying learning robotic systems is challenging. Existing techniques and tools for verification of an artificial neural network (ANN) are concerned with component-level properties. Here, we deal with robotic systems whose control software uses ANN components, and with properties of that software that may depend on all components. Our focus is on trained fully connected ReLU neural networks for control. We present an approach to (1) modelling ANN components as part of behavioural models for control software and (2) verification using traditional and ANN-specific verification tools. We describe our results in the context of RoboChart, a domain-specific modelling language for robotics with support for formal verification. We describe our modelling notation and a strategy for automated proof using Isabelle and Marabou, for example.

Keywords: verification, CSP, theorem proving, Isabelle, Marabou

1 Introduction

Artificial neural networks (ANN) are effective, powerful, and widely used [20]. They have been proposed for use in control software for robotic systems, performing various tasks such as collision detection [2, 1], and path finding [16]. When ANN components are used instead of non-ML components, they can increase time and space efficiency [20]. In addition, ANN-based systems are highly adaptable to new data and environments [7]. On the other hand, the behaviour of an ANN is highly dependent on the training data used in its development. There is, therefore, great interest in several forms of verification to ensure that an ANN-based system satisfies key properties of concern.

Existing formal-verification work focuses on ANN components. The other components are either considered informally to generate a specification [6, 19], or not at all. Here, we define a framework to model and verify the entire control software; we refer to such properties as module-level properties.

Several domain-specific languages support model-based software engineering in robotics [27]. Most, however, are not focused on verification. RoboChart [25] is distinctive in its support for verification by model checking and theorem proving. Our framework uses a denotational process-algebraic semantics for ANN components that integrates with the RoboChart semantics. We use it to enable verification mechanised using Isabelle/UTP [12] and Marabou [22], for instance.

The semantics of RoboChart is described in the CSP process algebra [32]. CSP enables verification via model checking [13], but is also a front-end to a predicative alphabetised relational theory described using the Unifying Theories of Programming [14] (UTP) for theorem proving.

We introduce novel ANN components in RoboChart, giving them a CSP semantics. To support tractable verification, we model fully connected ReLU pretrained ANNs [28]. Our semantics, however, supports any activation function, allowing additional tool integration. For proof, we use the UTP encoding of CSP.

In summary, our contributions are as follows. First, we describe a (RoboChart) ANN component with formal semantics. Second, we present an approach to verification that can be mechanised using Isabelle/UTP and Marabou in combination. In a recent survey [4], we have found that Marabou proved a collection of properties we identified twice as fast when compared with 13 other tools. To cater for numerical uncertainty, we use a new notion of conformance with a precision parameter defined in terms of refinement.

Next, in Sect. 2, we provide the background to our work. Sect. 3 presents our ANN components in RoboChart. Sect. 4 describes a semantics, and Sect. 5 discusses verification. Sect. 6 concludes and considers related and future work.

2 Background

We introduce in this section two concepts essential to our work. Sect. 2.1 briefly introduces ANNs, and Sect. 2.2 introduces CSP and UTP.

2.1 ANNs

An ANN is an abstraction of a nervous system of interconnected neurons: cells with multiple forms and components in biological neural networks. Information is stored at synapses: contact points between different neurons. 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*. The neuron's body determines the output signal sent by the axon.

ANN's approximate biological neurons through nodes (artificial neurons), graphically represented in Figure 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, nodes are arranged in layers. Each node is also assigned a value referred to as a bias. The weights and the biases are parameters of the ANN learnt from training data. Figure 2 shows the overall structure of an ANN, with each line representing a connection from the left to the right layer. The weights of each layer can be represented as a matrix, with one value for the connection of a node in the layer to a node in the previous layer. The bias of a layer can be represented as a vector, with a value for each node.

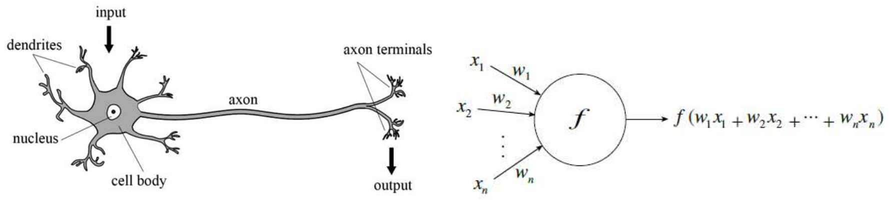

Fig. 1. A basic biological neuron from [26] and a basic node from [31]; w represents the synapses, the input connections represent the dendrites, f represents the cell body, and the output connection represents the axon.

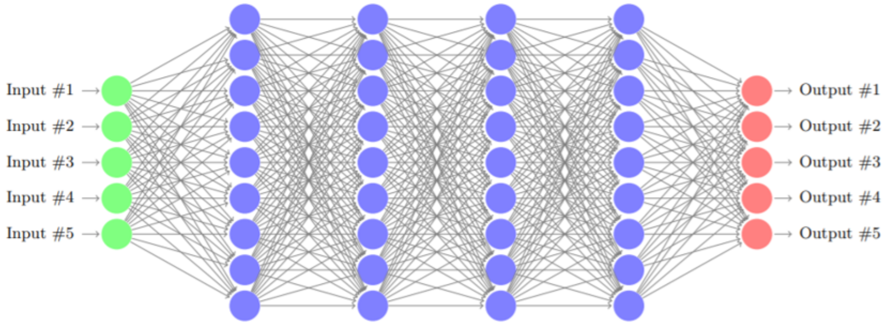

Fig. 2. An abstract neural network from [21].

We can consider an ANN as the definition of a function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ based on training data. (Any training process can be used.) Every node takes the weighted sum of the outputs of nodes of the previous layer and applies a bias and a non-linear activation function to this result. This work considers only the ReLU activation function: $f(x) = max(0, x)$.

The ReLU activation function is faster to train, easier to optimise, and performs and generalises well [28]. ReLU is also piecewise linear, which can be viewed as the composition of multiple linear functions. Piecewise linearity has positive implications in implementation, optimisation and verification as opposed to fully non-linear functions such as sigmoid or tanh. Furthermore, ReLU can eliminate the vanishing gradient problem (preventing the weight from changing value), is widely used, and can achieve state-of-the-art results [23].

2.2 CSP and UTP

Communicating Sequential Processes (CSP) [15] is a notation for modelling, validating, and verifying reactive programs via refinement.

A CSP model describes a set of processes defining interaction patterns of a mechanism. Processes interact with each other and their environment via atomic

Table 1. CSP Operators. Here, we use P and Q as metavariables to stand for processes, cs to stand for a channel set, defining a set of events potentially based on channels, e to stand for an event, i for an index, and T for a finite type. In addition, for the replicated (iterated) operators, $a(i)$ stands a set of events identified by an index, and similarly, $P(i)$ is a process identified by the index i.

Symbol	Name	Symbol	Name
$Skip$	Skip	$e \rightarrow P$	Prefix
$P \,|[\, cs \,]|\, Q$	Parallel Composition	$\parallel i : T \bullet [a(i)]P(i)$	Replicated Parallel
$P \mid\mid\mid Q$	Interleaving	$\mid\mid\mid\, i : T \bullet P(i)$	Replicated Interleaving
$P \Theta_{cs} Q$	Exception	$P \setminus cs$	Hiding

and instantaneous events representing channel communications. The set of all events a process can engage in is named Σ [33]. We present the CSP operators we use in Tab. 1; they are further described as we use them.

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. In 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 languages other than RoboChart that define reactive behaviours, but perhaps also include notions of continuous time [9] and probability [39], for instance.

All UTP theories describe relations between observations of variables. The change in an observation x is captured by the relation between the before value of the observation (named x) and its after value (named x').

We use the UTP theory of reactive contracts [10], which gives semantics to state-rich CSP processes and has a large set of algebraic laws for verification. The observational variables of this theory are st, st', ok, ok', $wait$, $wait'$, tt, ref, and ref'. The variables st and st' record the programming state of the process: its variables. The Boolean variables ok and ok' record the process's stability. The Boolean variables $wait$ and $wait'$ identify when the process is waiting for interaction with the environment. So, ok' and $\neg\, wait'$ indicate termination.

The sequence tt describes the trace of events in the life of the process up to the moment of observation: it is the difference between the trace of all events at the moment of observation and the trace as it was at the initiation of the current process. There is no tt' because tt is defined as $tr' - tr$, where tr and tr' record the traces of the process. The set ref' records the events that the process cannot perform when it next makes visible progress. This set is known as the process's refusals. A healthiness condition makes the value of ref irrelevant, as a process cannot depend on what was being refused before it started.

Reactive 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. The postcondition R_3 describes a condition on the state st, the state update st', and the final value of tt. In addition, we have a third predicate $R2$, which is called a pericondition. It captures the observations that can be made of a process when in a quiescent but not final state, that is, when it awaits its environment's interaction. The pericondition defines a condition on the pre-state st, the value of the trace tt, and which events are refused by referring to ref'.

Here, in defining reactive contracts, we use operators $\mathcal{E}[t, E]$ and $\Phi[t]$, which are simplified versions of those in Def. 4.6 from [11], where we consider that a CSP process does not have state variables. With $\mathcal{E}[t, E]$, we can construct a pericondition stating that t has been observed and the events in E are not refused. On the other hand, $\Phi[t]$ constructs a postcondition, stating that the final trace observed is characterised by t. Finally, we use $\{\mid c \mid\}$ to denote the set of all events for the channel c, communicating values according to type of c.

Next, we introduce our novel ANN components in RoboChart.

3 Modelling ANN Components in RoboChart

RoboChart is a diagrammatic modelling language that can be used to define a simple component model and behaviour of control software for robotics. In RoboChart, the overall software of is represented by a module block, which identifies one or more controllers interacting with a robotic platform. The block for a robotic platform specifies an abstraction of the hardware and its embedded software via events and operations representing services provided by the robot and used by the software. A controller block defines a thread of control, engaging in events, calling platform operations, and communicating with other controllers. One or more (parallel) state machines define the behaviour of a controller.

In our work, we extend RoboChart with a new form of controller defined by an ANN block. In Fig. 3, we present a RoboChart module for a Segway robot that includes an ANN component AnglePIDANN. This module, called Segway, contains a robotic platform SegwayRP, a standard controller SegwayController, and an ANN controller AnglePIDANN. SegwayRP has events representing data provided by the segway sensors and operations to control movement via the segway motors. SegwayController describes the behaviour of this system, defined through three state machines: BalanceSTM, RotationPID and SpeedPID.

As shown in Fig. 3, the block SegwayController has three blocks that represent references to its state machines, which are omitted here, but are available in [3]. SegwayController has a cyclic behaviour defined by BalanceSTM, which updates the motor speeds using the outputs of the PID machines and of the AnglePIDANN controller to keep the segway upright. In the original version of this example, there is a third state machine AnglePID. In our version here, we have an ANN instead, with the same interface. Just like AnglePID, the ANN component AnglePIDANN accepts as input the events anewError and adiff and communicates its output through the event angleOutputE.

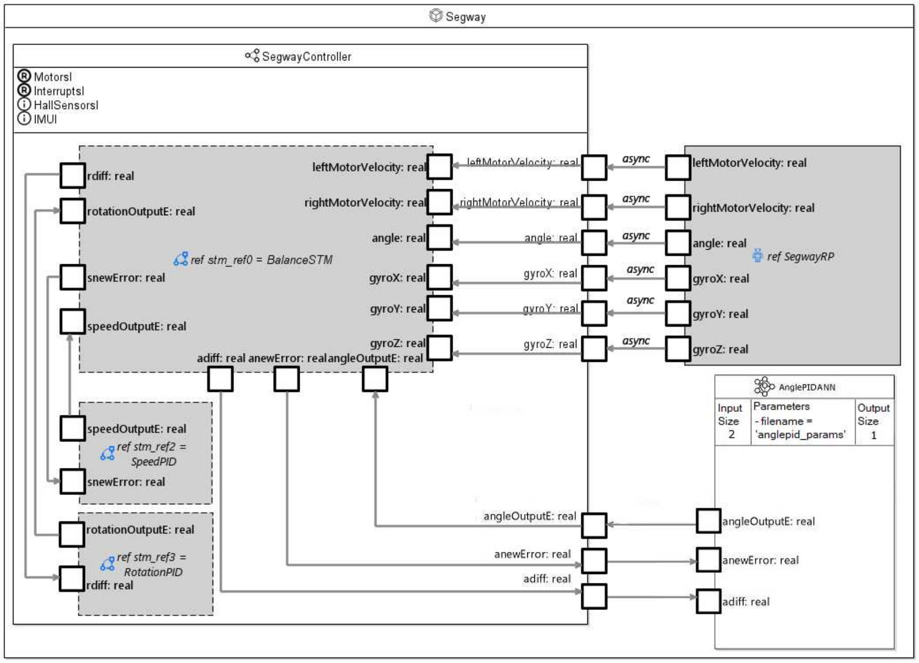

Fig. 3. A parallel version of the Segway model with an ANN component.

Legend: ⬡ : Module, ⋖ : controller definition, → : connection, π : constant, 🤖 : robotic platform reference, ⇆ : state machine reference, ⚛: ANN component.

The block for an ANN component (marked using the symbol ⚛) has its behaviour defined by the following parameters. First, we have the ANN's input and output sizes, representing the sizes of the vector the ANN is trained on for input and output. Next, we specify a parameter file that defines the layer structure, giving, for each layer, the weights and bias, and the activation function.

An ANN controller operates as a slave component. It can communicate with other controllers via events. The types of the events are restricted: they can either contain one event for every input and output, providing a scalar representation of the ANN, or precisely two events, capturing a vector representation for the inputs and outputs. In our example, we define multiple events (that communicate scalar values) to represent the inputs and output, as our ANN is low-dimensional. We declare two input events anewError and adiff, as the Input Size of AnglePIDANN is 2, and one output event angleOutputE, as the Output Size is 1.

The metamodel for our RoboChart extension is very simple; details are given in [3]. Principally, we have a class ANNController to represent a controller defined using an ANN. It defines the parameters of an ANN so that we have specifications for the values of six properties: insize, the input size of the ANN; outsize, the output size of the ANN; layerstructure, defining the size of each layer, and weights and biases, defining the weights and biases of the ANN. RoboChart's type system is based on Z [38, 35]. Hence, we can represent real numbers using the approach in [30]. Although different layers of an ANN can use different functions, we assume all layers use just one function. Extending our work to consider additional functions and different functions in different layers is straightforward.

The following section discusses the semantics of ANNControllers.

4 CSP Semantics

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

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 of a type *Value* for each of them. Equally, constants such as *layerstructure*, *maxSize*, and the others mentioned here are defined for each component.

The semantics of an ANN component is a process presented in Fig. 4. It is defined by parallel composition of processes representing nodes and layers.

We use two channels. The first $layerRes : \{0 \mathinner{..} layerNo\}.\{1 \mathinner{..} maxSize\}.Value$ 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 nth node in the process for the lth layer. The channel $nodeOut : \{1 \mathinner{..} layerNo\}.\{1 \mathinner{..} maxSize\}.\{1 \mathinner{..} maxSize\}.Value$ is for intra-node communication; $nodeOut.l.n.i.v$ refers to the layer, node and value as for *layerRes*. The additional index value i identifies the node in the following layer (of index $l + 1$) that receives this communication.

In our semantics, 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 $layerRes.0$ represent inputs to the ANN process from other components in the RoboChart model. All other communications on *layerRes* represent results from layer processes. Events $layerRes.layerNo$ represent the outputs of the ANN.

Fig. 4 presents the specification of a process *ANN*, defining the semantics for an ANNController. It terminates (*Skip*) on the occurrence of a special event *end*, as determined using the exception operator Θ_{end}. This is an event raised by other controllers when all state machines terminate. An ANN does not terminate in any other way, so termination is determined by the other controllers.

The operator $P \,[\![\, cs \,]\!]\, Q$ 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

$$
\begin{array}{l}
\mathit{ANN} = \\
\quad ((\mathit{HiddenLayers} \;|[\, \{\!|\; \mathit{layerRes}.(\mathit{layerNo} - 1) \;|\!\}\,]|\; \mathit{OutputLayer}) \setminus \mathit{ANNHiddenEvts} \\
\quad \Theta_{\mathit{end}}\ \mathit{Skip}); \\
\quad \mathit{ANN}
\end{array}
$$

$$
\mathit{ANNHiddenEvts} = \Sigma \setminus \{\!|\; \mathit{layerRes}.0, \mathit{layerRes}.\mathit{layerNo}, \mathit{end} \;|\!\}
$$

$$
\begin{array}{l}
\mathit{HiddenLayers} = \big\| \; i : 1 \mathrel{..} \mathit{layerNo} - 1 \bullet [\, \{\!|\; \mathit{layerRes}.(i-1), \mathit{layerRes}.i \;|\!\}\,] \\
\qquad\qquad \mathit{HiddenLayer}(i, \mathit{layerSize}(i), \mathit{layerSize}(i-1))
\end{array}
$$

$$
\mathit{HiddenLayer}(l, s, \mathit{inpSize}) = \big\| \; i : 1 \mathrel{..} s \bullet [\{\!|\; \mathit{layerRes}.(l-1) \;|\!\}] \mathit{Node}(l, i, \mathit{inpSize})
$$

$$
\begin{array}{l}
\mathit{Node}(l, n, \mathit{inpSize}) = \\
\quad ((\,|||\; i : 1 \mathrel{..} \mathit{inpSize} \bullet \mathit{NodeIn}(l, n, i)) \\
\qquad |[\; \{\!|\; \mathit{nodeOut}.l.n \;|\!\} \;]| \\
\quad\; \mathit{Collator}(l, n, \mathit{inpSize})\,) \setminus \{\!|\; \mathit{nodeOut} \;|\!\}
\end{array}
$$

$$
\mathit{NodeIn}(l, n, i) = \mathit{layerRes}.(l-1).n?x \rightarrow \mathit{nodeOut}.l.n.i!(x * \mathit{weight}) \rightarrow \mathit{Skip}
$$

$$
\begin{array}{l}
\mathit{Collator}(l, n, \mathit{inpSize}) = \mathbf{let} \\
\qquad\qquad C(l, n, 0, \mathit{sum}) = \mathit{layerRes}.l.n!(\mathit{ReLU}(\mathit{sum} + \mathit{bias})) \rightarrow \mathit{Skip} \\
\qquad\qquad C(l, n, i, \mathit{sum}) = \mathit{nodeOut}.l.n.i?x \rightarrow C(l, n, (i-1), (\mathit{sum} + x)) \\
\qquad \mathbf{within} \\
\qquad\qquad C(l, n, \mathit{inpSize}, 0)
\end{array}
$$

$$
\begin{array}{l}
\mathit{OutputLayer} = \big\| \; i : 1 \mathrel{..} \mathit{layerSize}(\mathit{layerNo}) \bullet [\, \{\!|\; \mathit{layerRes}.(\mathit{layerNo} - 1) \;|\!\}\,] \\
\qquad\qquad \mathit{Node}(\mathit{layerNo}, i, \mathit{layerSize}(\mathit{layerNo} - 1))
\end{array}
$$

Fig. 4. CSP ANN Semantics - General.

via a recursive call. Since the *OutputLayer* communicates only with the last hidden layer, these processes synchronise on the events $\mathit{layerRes}.(\mathit{layerNo} - 1)$.

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

We define the process *HiddenLayers* via an iterated alphabetised parallel composition ($\big\|$) over an index i for hidden layers ranging from 1 to $\mathit{layerNo} - 1$. For each i, the layer-process $\mathit{HiddenLayer}(i, \mathit{layerSize}(i), \mathit{layerSize}(i-1))$ for the ith layer is associated with the alphabet containing the set of events on $\mathit{layerRes}.(i-1)$ and $\mathit{layerRes}.i$. 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 $\mathit{layerRes}.(i-1)$ and the process for the following layer on $\mathit{layerRes}.i$. So, the output events of each layer are used as the input events for the next layer.

The second argument $\mathit{layerSize}(i)$ 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 nth 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 its output through $nodeOut.l.n.i$. The output of $NodeIn(l, n, i)$ 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 (*Skip*).

An input of a value x via a channel c can be specified in CSP using the prefixing construct $c?x \to P$, which defines the process that engages in an event $c.x$ and behaves like P. This process accepts inputs x over the channel c's type. The output prefix $c!v \to P$ is a process that outputs (synchronises) on the specific event $c.v$ 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 whose arguments are 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 its 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, as mentioned in Sect. 3, these events are anewError, adiff, and angleOutputE.

The RoboChart semantics defines a CSP process for the module by composing processes for each controller, each state machine, and memory, holding values for local variables. The semantics of an ANN component fits in the semantics of a RoboChart module as that of a controller process. With this semantics, we can prove the properties of the RoboChart module in terms of the events and operations of the robotic platform rather than just the inputs of the ANN.

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 machine AnglePID of the original version of the segway model. For the latter, we have used the semantics automatically generated by the RoboChart tool[1]. 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 [29]. This has enabled simulation and assertion-based reasoning via JML in a setting where values are floating-point numbers.[2]

In general, however, we require a proof approach that caters for use of real numbers. Next, we describe our proof approach based on UTP.

5 Automated verification using UTP

In this section, we define UTP reactive contracts that capture the semantics of our ANN components presented in Sect. 4 and an approach to verification. In Sect. 5.1, we describe a general pattern of UTP reactive contracts for ANN components. In Sect. 5.2, we present a pattern for the semantics of standard RoboChart controllers that we use as specification. Finally, in Sect. 5.3, we present our notion of conformance for ANN components, a verification conditions to prove properties combining Isabelle/UTP and Marabou, for example.

5.1 General Pattern

Def. 1 below provides a pattern for contracts corresponding to an optimised version of the CSP process *ANN* in Fig. 4. 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, compositional reasoning 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, and 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.

[1] Available at robostar.cs.york.ac.uk/robotool/

[2] All the artefacts related to this validation work are available at github.com/UoY-RoboStar/robochart-ann-components/.

For brevity, in Def. 1, the contract is defined using a sequence *input* containing only the events representing inputs extracted from the trace *tt*. Formally, $\mathit{input} = \mathit{tt} \upharpoonright \{\mid \mathit{layerRes}.0 \mid\}$. (We use $\upharpoonright$ for sequence filtering.)

Definition 1 (ANN Component General Contract).

$$
\begin{array}{l}
\mathit{GeneralANNContract} \mathrel{\widehat{=}} \\
\qquad [\ \ \mathit{true}_r \\
\qquad \vdash \#\mathit{input} < \mathit{insize} \land \mathcal{E}[\mathit{input}, \{\mid \mathit{layerRes}.0.(\#\mathit{input} + 1) \mid\}] \\
\qquad\quad\ \lor \\
\qquad\quad\ \#\mathit{input} = \mathit{insize} \land \\
\qquad\quad\ \exists\, l : 1 \mathinner{..} \mathit{layerNo} \bullet \exists\, n : 1 \mathinner{..} \mathit{layerSize}(l) \bullet \\
\qquad\qquad \mathcal{E}[\mathit{front} \circ \mathit{layeroutput}(l, n), \{\ \mathit{last} \circ \mathit{layeroutput}(l, n)\ \}] \\
\qquad \mid\ \ \#\mathit{input} = \mathit{insize} \land \Phi[\mathit{layeroutput}(\mathit{layerNo}, \mathit{layerSize}(\mathit{layerNo}))] \\
\qquad]
\end{array}
$$

The pattern in Def. 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 ($\#\mathit{input} < \mathit{insize}$), the trace is *input*, and the next input event $\mathit{layerRes}.0.(\#\mathit{input} + 1)$ is not refused.

When all inputs are available ($\#\mathit{input} = \mathit{size}$), we specify the trace of *layerRes* interactions up to where $\mathit{layerRes}.l.n$ has occurred using a function $\mathit{layeroutput}(l, n)$, 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 $\mathit{layerSize}(l)$. The function $\mathit{layeroutput}(l, n)$ encodes the specification of the ANN, in terms of its structure, into a trace-based specification. For instance, for an ANN with input size 2, with two nodes in its first layer, like in our example, if *tt* defines the *input* sequence as $\langle \mathit{layerRes}.0.1.1, \mathit{layerRes}.0.2.1\rangle$, then $\mathit{layeroutput}(1, 2)$ is $\langle \mathit{layerRes}.0.1.1, \mathit{layerRes}.0.2.1, \mathit{layerRes}.1.1.(1.75), \mathit{layerRes}.1.2.(1.80)\rangle$. This reflects the fact that the inputs are taken first, and then the output of each node is the weighted sum of these inputs. Here, we consider all weights to be 0.5, the bias value of the first node to be 0.75, and of the second node to be 0.8. The output of the first node is captured by the event $\mathit{layerRes}.1.1.(1.75)$, where the value 1.75 communicated is the result of the calculation $((1 * 0.5) + (1 * 0.5)) + 0.75$; the output 1.8 for the second node results from considering the bias 0.8.

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

To specify the postcondition, we use $\mathit{layeroutput}(\mathit{layerNo}, \mathit{layerSize}(\mathit{layerNo}))$ for when the trace for the last node (that of index $\mathit{layerSize}(\mathit{layerNo})$) of the last layer (that of index $\mathit{layerNo}$) has occurred.

For conciseness, we omit here the definition of $\mathit{layeroutput}$. It can be found in [3], along with all other definitions and proofs omitted here.

Using laws of reactive contracts and the definition of the CSP operators [10], we can prove that the pattern in Def. 1 captures the RoboChart ANN semantics.

5.2 Cyclic Memoryless RoboChart controllers

An ANN cannot implement reactive behaviour, where events are interspersed according to environmental interactions. So, we consider specifications that define a cyclic controller, whose events can be classified as inputs or outputs, and whose control flow alternates between taking inputs and producing outputs, never terminating and without memory across cycles. (This is the flow of simulations, for example.) For such controllers, the RoboChart semantics of one iteration can be captured by a reactive design of a particular format. A reactive design defines a relation via just a precondition and a postcondition, which, however, specifies both intermediate final observations. In other words, a reactive design combines the pericondition and the postcondition in a single predicate.

In the case of the segway, as already mentioned, the inputs of the AnglePID are anewError and adiff, and the output is angleOutputE as indicated by the connections to the SegwayController (see Fig. 3). The reactive design for AnglePID captures the behaviour of one iteration of the state machine: it receives inputs *anewError* and *adiff* and produces an output via *angleOutputE*. It has precondition true_r and the following postcondition, where the local variables of the RoboChart model are quantified and defined according to that model in terms of constants P and D. (In spite of its name, AnglePID is a PD controller.)

$$
\begin{array}{l}
\exists\, \mathit{currNewError}, \mathit{currDiff}, \mathit{currAngleOut} : \mathit{Value} \mid \\
\quad \mathit{currAngleOut} = P * \mathit{currNewError} + D * \mathit{currDiff} \bullet \\
\qquad \mathit{wait}' \wedge (\, (\mathit{tt} = \langle\rangle \wedge \mathit{anewError}.\mathit{currNewError} \notin \mathit{ref}') \vee \\
\qquad\qquad\qquad (\mathit{tt} = \langle \mathit{anewError}.\mathit{currNewError} \rangle \wedge \mathit{adiff}.\mathit{currDiff} \notin \mathit{ref}') \vee \\
\qquad\qquad\qquad (\mathit{tt} = \langle \mathit{anewError}.\mathit{currNewError}, \mathit{adiff}.\mathit{in}.\mathit{currDiff} \rangle \wedge \\
\qquad\qquad\qquad\qquad \mathit{angleOutputE}.\mathit{currAngleOut} \notin \mathit{ref}') \,) \\
\qquad \vee \\
\qquad \neg\, \mathit{wait}' \wedge \mathit{tt} = \langle \mathit{anewError}.\mathit{currNewError}, \mathit{adiff}.\mathit{currDiff}, \\
\qquad\qquad\qquad \mathit{angleOutputE}.\mathit{currAngleOut} \rangle
\end{array}
$$

The postcondition comprises two parts: either the process is waiting on interaction (wait'), or not ($\neg\, \mathit{wait}'$). When wait' holds, there are three cases distinguished by the trace contribution tt: no input events have happened, just *anewError* has been provided, or both *anewError* and *adiff* have been provided. In each case, the next event is not refused, that is, it does belong to ref'. When wait' is false, tt contains all inputs and the output. In this case, the value of ref' is irrelevant and not specified, since the process has terminated.

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

Definition 2 (Cyclic RoboChart Controller Pattern).

$$
\begin{array}{l}
Cyclic\_RC\_Controller \mathrel{\widehat=} \\
\quad [\ true_r \\
\quad \vdash \\
\quad \ \exists\, x_1, \ldots, x_{\#inp};\ y_1, \ldots, y_{\#out} : Value \mid p \bullet \\
\qquad wait' \land (\exists\, i : \mathrm{dom}\, inp \bullet\ tt = {\frown}/\ j : 1..(i-1) \bullet \langle inp(j).x_j \rangle \land \\
\qquad\qquad\qquad\qquad\qquad\quad inp(i).x_i \notin ref') \\
\qquad\qquad\quad \lor \\
\qquad\qquad\quad (\exists\, i : \mathrm{dom}\, out \bullet\ tt = {\frown}/\ n : \mathrm{dom}\, inp \bullet \langle inp(n).x_n \rangle {}^\frown \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad {\frown}/\ j : 1 \mathinner{..} (i-1) \bullet \langle out(j).y_j \rangle \land \\
\qquad\qquad\qquad\qquad\qquad\quad out(i).y_i \notin ref') \\
\qquad \lor \\
\qquad \neg\, wait' \land \\
\qquad tt = {\frown}/\ i : \mathrm{dom}\, inp \bullet \langle inp(i).x_i \rangle {}^\frown {\frown}/\ j : \mathrm{dom}\, out \bullet \langle out(j).y_j \rangle \\
\quad]
\end{array}
$$

The reactive design for AnglePID is an instance of $Cyclic\_RC\_Controller$ above, where we have two inputs: x_1 is $currNewError$ and x_2 is $currDiff$. The output y_1 is $currAngleOut$. So, $\mathrm{dom}\, inp$ is $\{1, 2\}$, and $\mathrm{dom}\, out$ is $\{1\}$. The predicate p characterises values for the outputs in terms of local variables x_i and y_i.

In Def. 2, in the $wait'$ case, we have a disjunction of two existential quantifications. In the first, the quantification on i ranges over $\mathrm{dom}\, inp$, and we define a value for tt in terms of a distributed concatenation (${\frown}/$), that is, the concatenation of a sequence of sequences. The concatenation comprises singleton sequences $\langle inp(j).x_j \rangle$, with j ranging over $1 \mathinner{..} (i-1)$. These represent all input events before the i-th input given by the event $inp(i).x_i$. So we get $tt = \langle\rangle$ for $i = 1$, or $tt = \langle anewError.currNewError \rangle$ for $i = 2$ and $j = 1$. For the definition of ref', we specify that the input event $inp(i).x_i$, which is either $anewError.currNewError$ or $adiff.currDiff$, is not refused. This corresponds to the first two disjuncts in the $wait'$ case of the postcondition for AnglePID.

The second quantification is on i from $\mathrm{dom}\, out$, with tt formed of two distributed concatenations. The first is of sequences $\langle inp(n).x_n \rangle$, like in the first quantification, but now n ranges over the whole $\mathrm{dom}\, inp$, so we get all input events. The second is of sequences $\langle out(j).y_j \rangle$, representing proper prefixes of the sequences of all output events. Our example has one output, so this sequence resolves to the empty trace. The refusal does not include the following output. For our example, we accept $out(i).y_i$, which is $angleOutputE.currAngleOut$.

In the terminating case, that is, $\neg\, wait'$, we define tt as the concatenation of all input events followed by all output events. In our example, we get the trace $\langle anewError.currNewError, adiff.currDiff, angleOutputE.currAngleOut \rangle$.

A reactive design that instantiates the pattern in Def. 2 defines one iteration of a cyclic RoboChart controller. In the full model of the controller, that design is the body of a loop with the weakest fixed-point semantics. Since the precondition is $true_r$, the weakest fixed-point operator transfers to the postcondition [10].

Besides structural differences in the patterns in Def. 1 and 2, we have a substantial difference in how outputs are defined regarding the inputs. In an ANN contract, the results are determined by a deterministic function based on the parameters of an ANN. In the 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.

We next consider how to verify an ANN component against a cyclic controller.

5.3 Conformance

In our approach to verification, we take a RoboChart standard controller as the specification for an ANN component. So, our goal is to prove that the ANN is correct with respect to the RoboChart controller. ANN components, however, contain numerical imprecision, so we allow an error tolerance on the values communicated by the output events of an ANN component. Formally, we define a conformance relation $Q\ conf(\epsilon)\ P$ that holds if, and only if, Q is a refinement of P, where the value of P's outputs can vary by at most ϵ as formalised below.

Definition 3 (Conformance Relation).

$$\begin{array}{l} Q\ \mathit{conf}(\epsilon)\ P \Leftrightarrow \\ \quad \exists\, s : \mathit{seq}\ \mathit{Event};\ a : \mathbb{P}\,\mathit{Event} \mid \mathit{tt}\ \mathit{seqapprox}(\epsilon)\ s \wedge (\alpha P \setminus \mathit{ref}')\ \mathit{setapprox}(\epsilon)\ a \bullet \\ \qquad P[s, (\alpha P \setminus a)/\mathit{tt}, \mathit{ref}'] \sqsubseteq Q \end{array}$$

We say that $Q\ conf(\epsilon)\ P$ 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 $seqapprox(\epsilon)$ and $setapprox(\epsilon)$. Here, αP is the set of events used in P, and $\setminus$ is the set difference operator, so that $\alpha P \setminus ref'$ is the set of events that P is not refusing, that is, its acceptances. Moreover, $s_1\ seqapprox(\epsilon)\ s_2$ 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. Similarly, $A_1\ setapprox(\epsilon)\ A_2$ if, and only if, their input events are the same, but although the output events are the same the communicated values differ by at most ϵ.

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 [24] for examples of the kinds of laws of interest.) In particular, refinement may need to be used to derive the specification of a cyclic controller for implementation using an ANN. In our example, we have used refinement to justify transformations to extract the AnglePID state machine out of the SegwayController where it was originally and obtain the Segway module in Fig. 3.

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

Theorem 1. $P \sqsubseteq Q \land R\ conf(\epsilon)\ Q \Rightarrow R\ conf(\epsilon)\ P$

This ensures that the ANN conforms to the original specification. So, the ANN may have removed nondeterminism present in the original controller, and exhibit some numeric imprecision bounded by ϵ, but that is all.

The following theorem identifies verification conditions that are sufficient to prove conformance for instances of our patterns. In Theorem 3, we further instantiate the verification conditions to consider conformance proofs using the semantics of standard and ANN controllers in RoboChart.

Theorem 2. *$Q\ conf(\epsilon)\ P$ provided*

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

and

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

where Q and P are instances of the patterns in Definitions 1 and 2.

In short, Theorem 2 gives two verification conditions that distribute conformance over the pericondition and postcondition of Q. For any reactive contract RC, we use RC_2 and RC_3 to refer to its pericondition and to its postcondition. The first verification condition requires the periconditions P_2 and Q_2 to be related by $conf(\epsilon)$. The second condition makes the same requirement of the postconditions P_3 and Q_3 and is simpler because postconditions do not restrict ref'.

In the context of our work, the proof of conformance is in the following form.

$$(Q \setminus ANNHiddenEvts)[inp/layerRes.0, out/layerRes.layerNo]\ conf(\epsilon)\ P \quad (1)$$

Here, Q is a reactive contract that instantiates the pattern in Def. 1, and P captures the semantics of a cyclic controller described using the pattern in Def. 2. As said, our general contract for ANN components does not capture the hiding in the CSP semantics (Figure 4), 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 inputs and outputs.

For our example, the conformance requirement is based on $AnglePIDANN$, the reactive contract for AnglePIDANN. Besides hiding the $layerRes.1$ events, we rename $layerRes.0.1$ and $layerRes.0.2$ to $currNewError$ and $layerRes.0.2$ to $curradiff$, and $layerRes.2.1$ to $currangleOutput$. With this, we can discharge the verification conditions identified in Theorem 2 using Isabelle and the laws of UTP 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 setLeftMotorSpeed() and 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 (1), we can obtain the same result for the version of the segway software that uses AnglePIDANN, although we need to accept a tolerance for the values set.

For the particular case where the conformance that is being proved is of the form (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 3.

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

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 $\#inp = insize$ and $\#out = outsize$.

Theorem 3 states that if we show that there is no combination of input and output values for which there is an output y_i whose error, as defined by comparison to $\mathit{annoutput}(\mathit{layerNo}, i, \langle x_1, .., x_{insize} \rangle)$, is greater than ϵ, then our verification conditions are discharged. By requiring that the intersection between the singleton set $\{\mathit{annoutput}(\mathit{layerNo}, i, \langle x_1, .., x_{insize} \rangle)\}$ and $\{x : \mathbb{R} \ \mid \ |x - y_i| \leqslant \epsilon\}$ is empty, we require the output y_i to be in range. The error refers to the difference between the ANN's output $\mathit{annoutput}(l, n, \langle x_1, .., x_{insize} \rangle)$ and the cyclic RoboChart controller's output, captured by the variables y_i and the predicate p. The ANN's output value is characterised using $\mathit{annoutput}(l, n, \mathit{in})$, which determines the value communicated by the output event of the n-th node of layer l, given a sequence in of inputs to the ANN.

We provide an example below of the reachability conditions we obtain using Theorem 3, based on our AnglePID example.

Example 1. The antecedent of Theorem 3 for our example is the following verification condition. (Here, i takes just the value 1).

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

The verification condition can be encoded as a set of reachability conditions if we define *Value* to be a set *MValue* defined in terms of a minimal value *min* and a natural number c as: $\bigcup \{\, n : 0 \,..\, c \bullet [\mathit{min} + n \times c, \mathit{min} + (n+1) \times c] \,\}$. Given these constants, we can obtain finite conditions to prove in Marabou if we accept this limitation, as illustrated by the lemma below.

Lemma 1. *The antecedent of Theorem 3 for* AnglePIDANN *is as follows.*

$$
\begin{array}{l}
\neg\ \exists\, n_1, n_2 : 0 \,..\, c \bullet \exists\, \mathit{currNewError}, \mathit{currDiff} : \mathbb{R} \bullet \\
\quad \mathit{min} + n_1 \times c \leqslant \mathit{currNewError} \leqslant \mathit{min} + (n_1 + 1) \times c \wedge \\
\quad \mathit{min} + n_2 \times c \leqslant \mathit{currDiff} \leqslant \mathit{min} + (n_2 + 1) \times c \wedge \\
\quad \mathit{annoutput}(\mathit{layerNo}, 1, \langle \mathit{currNewError}, \mathit{currDiff} \rangle) \leqslant \\
\quad\quad (P * (\mathit{min} + n_1 \times c) + D * \mathit{min} + n_2 \times c) - \epsilon \\
\quad \vee \\
\quad \mathit{min} + n_1 \times c \leqslant \mathit{currNewError} \leqslant \mathit{min} + (n_1 + 1) \times c \wedge \\
\quad \mathit{min} + n_2 \times c \leqslant \mathit{currDiff} \leqslant \mathit{min} + (n_2 + 1) \times c \wedge \\
\quad \mathit{annoutput}(\mathit{layerNo}, 1, \langle \mathit{currNewError}, \mathit{currDiff} \rangle) \geqslant \\
\quad\quad (P * (\mathit{min} + (n_1 + 1) \times c) + D * \mathit{min} + (n_2 + 1) \times c) + \epsilon
\end{array}
$$

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, using a similar approach to that in [7]. 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.

Lemma 1's constraints form a hyper-rectangle in the domain of an ANN, and form a convex polytope, in inequality form, in the range. We can use these sets to specify properties in other tools [34, 36] as well as in Marabou as we have done. In particular, we can use tools that are able to handle non-linear activation functions, such as tanh and sigmoid, as well as ReLU.

6 Conclusions

As far as we know, we have proposed the first verification technique for robotic software in which an ANN is viewed as a white-box component whose reliability can and should be established. We guarantee properties, specified by state machines, of software that is implemented using instead trained, feed-forward, fully connected ReLU ANNs of any size or shape.

We have defined an ANN as a controller-like component in RoboChart and have validated the semantics via model checking using FDR [37], for discretised versions of the ANN, and via simulation using JCSP [29]. We have also presented a refinement-based method to prove ANN properties. We cater for an ANN component's numerical instability and provide a notion of conformance that can be used to justify replacing an existing RoboChart controller with an ANN if the error bound is accepted. We have identified sufficient verification conditions to establish conformance and shown how to combine ANN-specific (Marabou) and general theorem-proving tools (Isabelle) to discharge them. For illustration, we have applied our technique to a PID controller.

The work by Brucker and Stell in [5] is closely related to ours: they use Isabelle/HOL to verify the properties of feed-forward ANNs. They use their framework to demonstrate the properties of image-classification networks not considered here. Their goal, however, is component-level verification. It is feasible to use their results instead of Marabou to automate our proofs using only Isabelle and avoid input and output value restrictions.

Dupont et al. [8] define approximate notions of refinement for continuous systems. Their work considers two different views of conformance: upwards approximation, where an approximated system is refined to an exact system, and downwards approximation, the inverse. Our approach uses upward approximation: we refine an approximate system into an exact system. We, however, are concerned with ANN outputs, not trajectories of a continuous system.

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 [18].

Our second future goal is to define a toolchain of ANN-specific tools, so that, instead of relying on discharging our proof obligations using just a single tool, we have a collection of tools available. This requires techniques to reduce the search space and prove properties using complete techniques. This toolchain would allow us to verify more extensive and complex ANNs.

Finally, another future goal is to consider use of an ANN for perception, where the availability of a specification is not immediate. Developing meaningful specifications for such components is challenging, but there is a growing body of relevant work to address this [17] that we plan to consider.

References

1. 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.
2. 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*, pages 129–135, 1993.

3. Z. Attala. Verification of RoboChart Models with ANN Components. Technical report, University of York, 2023. Available at robostar.cs.york.ac.uk/publications/reports/Ziggy_Attala_Draft_Thesis.pdf.
4. Z. Attala, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Comparison of Neural Network Tools for the Verification of Linear Specifications of ReLU Networks. In A. Albarghouthi, G. Katz, and N. Narodytska, editors, *3rd Workshop on Formal Methods for ML-Enabled Autonomous System*, pages 22–33, 2020.
5. 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.
6. Arthur Clavière, Eric Asselin, Christophe Garion, and Claire Pagetti. Safety verification of neural network controlled systems. *CoRR*, abs/2011.05174, 2020.
7. Tommaso Dreossi et al. Counterexample-Guided Data Augmentation, 2018. arXiv:1805.06962.
8. 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.
9. 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.
10. Simon Foster et al. Unifying Theories of Reactive Design Contracts. *CoRR*, abs/1712.10233, 2017.
11. Simon Foster et al. Automated verification of reactive and concurrent programs by calculation. *CoRR*, abs/2007.13529, 2020.
12. Simon Foster et al. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197:102510, Oct 2020.
13. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.
14. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.
15. C.A.R Hoare. *Communicating Sequential Processes.* Prentice Hall International, January 1985.
16. Victoria J. Hodge, Richard Hawkins, and Rob Alexander. Deep Reinforcement Learning for Drone Navigation using Sensor Data. *Springer Nature - Neural Computing and Applications*, 2020. DOI: 10.1007/s00521-020-05097-x.
17. 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.
18. Yuval Jacoby, Clark W. Barrett, and Guy Katz. Verifying recurrent neural networks using invariant inference. *CoRR*, abs/2004.02462, 2020.
19. Kyle D. Julian and Mykel J. Kochenderfer. Guaranteeing safety for neural network-based aircraft collision avoidance systems. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC).* IEEE, sep 2019.
20. 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.
21. Guy Katz et al. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. *LNCS*, page 97–117, 2017.

22. 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.
23. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015. doi.org/10.1038/nature14539.
24. 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.
25. Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18:3097–3149, 2019.
26. 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.
27. Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A Survey on Domain-Specific Languages in Robotics. In *LNAI,volume 8810*, 10 2014.
28. Chigozie Nwankpa et al. Activation Functions: Comparison of trends in Practice and Research for Deep Learning, 2018. arXiv:1811.03378.
29. P.D.Austin and P.H.Welch. CSP for JavaTM (JCSP) 1.1-rc4 API Specification, 2008. www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/.
30. ProofPower-Z reference manual, 2006.
31. Raúl Rojas. *Neural Networks — A Systematic Introduction*, chapter 7. Springer-Verlag, 1996.
32. A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
33. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
34. 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.
35. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliff, 1992.
36. 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.
37. University of Oxford. *FDR Manual*, May 2020. Release 4.2.7. Retrieved from dl.cocotec.io/fdr/fdr-manual.pdf, on the 31st of May 2020.
38. Jim Woodcock and Jim Davies. *Using Z*. Prentice Hall, 1996.
39. 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.