Process-Algebraic Semantics for Verifying Intelligent Robotic Control Software

Ziggy Attala(✉), Fang Yan, Simon Foster, Ana Cavalcanti, and Jim Woodcock

University of York, York, UK
{ziggy.attala,fang.yan,simon.foster,ana.cavalcanti,jim.woodcock}@york.ac.uk

Abstract. Verification of robotic systems that use neural networks is a challenge. In this paper, we present a formal technique supported by tools to model and verify control software involving neural networks. Our technique enables reasoning about the reactive, communication-based, properties of a system through a process-algebraic lens. We support our framework with a link to state-of-the-art ANN verification tools, using them to prove contextual properties of a neural network. Our approach is flexible, platform-independent, and focuses on the logic of neural network models, instead of on a training method or specific use case.

Keywords: verification · *Circus* · theorem proving · Isabelle · Marabou

1 Introduction

The question of how we can, and should, use machine learning in robotics is of significant interest to industry, to the public, and to governing bodies. Reliability of robotics and autonomous systems (RAS) is a key concern in any context where the system interacts with humans or hazardous materials. In general, the fact that a robot has a physical presence and has the potential to affect its environment directly typically raises safety concerns.

Here, we present an approach to modelling and verifying control software for robotic systems that may include components realised by an artificial neural network (ANN). Our approach is based on diagrammatic, behavioural models, from which a process-algebraic formal semantics can be automatically generated. Verification uses a refinement-based conformance notion, and is automated using the theorem prover Isabelle [26] and the ANN verification tool Marabou [18]. We consider the fully-connected feed-forward ANN model, but the nature of process algebra means we can extend our semantics to provide a similar model for further network types. In addition, our notion of conformance is general.

For modelling, we consider RoboChart [31], a diagrammatic modelling language with semantics based on CSP [14]. It can capture reactive behaviour, parallelism, data flow, and event order and availability. RoboChart enables model-based engineering, including notions of refinement and composition.

In [1], we have described an extension of RoboChart that allows the definition of behaviour using the hyperparameters of an ANN and state machines. Here, we present a metamodel for that extension, well-formedness conditions, and their implementation. We have extended RoboTool, a set of Eclipse plug-ins that support design and verification using RoboChart, to deal with ANN models.

The work in [1] describes informally a semantics for our RoboChart extension. Here, we extend that semantics to cover normalisation, formalise it, and describe its implementation to generate *Circus* automatically [27] models for use in Isabelle. *Circus* is a combination of CSP and Z [34], and with its encoding in Isabelle, we can deal with the rich data model of an ANN and of RoboChart.

Finally, here we generalise and formalise the notion of conformance described in [1]. We mechanise it in Isabelle, prove key properties characterising it as a simulation relation [15], and present two theorems that identify verification conditions for (1) use in proofs based on Isabelle; and (2) joint use of Isabelle and Marabou. Our notion of conformance can be used to compare arbitrary RoboChart components. We pursue, in particular, an approach where a traditional component, specified using state machines, is compared to a component defined by an ANN. As such, this part of our work is focused around ANN models used for control, not recognition.

To summarise, our contributions here are as follows. First, we developed a tool for modelling and analyzing software involving ANN components. Second, we defined a denotational semantics for these components targeting *Circus*. Third, we have developed a tool for the automated analysis of the *Circus* semantics. Finally, we have proved theorems that allow for the sound use of ANN-specific solvers to prove the properties of the system. Although our focus here is on semantics and conformance for ANN components, the nature of our semantic model and our notation for conformance is key to enabling the modelling and analysis of software involving ANN components.

We discuss related work in Sect. 2. We provide an overview of our approach in Sect. 3. We describe our extensions to RoboChart in Sect. 4. Section 5 presents our mechanisation of the semantics, and associated lemmas and theorems. We conclude and provide future directions for work in Sect. 6.

2 Related Work

Many component-level verification approaches and tools exist for ANNs, such as [13,20,35]. We can leverage them for software system-level reasoning.

The use of hybrid automata is dominant [17,23,30,36] for verifying AI-enabled systems. In this line of work, the physical properties of a system, including the scenarios, the robotic platform, and the software, are captured in a single hybrid automaton, with each ANN in a single state. In contrast, we use a diagrammatic domain-specific language (DSL), namely RoboChart, to capture software models from which we can calculate mathematical models automatically. Moreover, as a platform-independent account of the software, a RoboChart model can generate code with a related notation available to capture physical

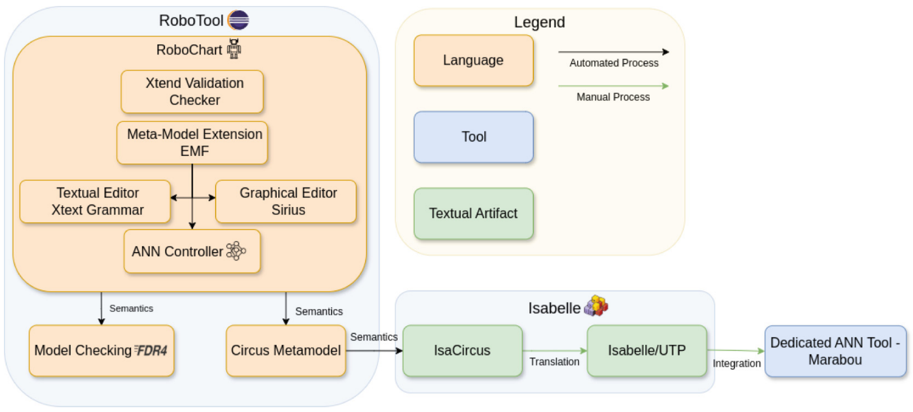

Fig. 1. Our toolchain for modelling and verifying neural control software for robotics. Black edges are steps automated by tools, and green edges are currently completed manually. (Color figure online)

behavior [25]. On the other hand, some hybrid automata work can deal with verifying quantitative properties, which is not our focus here. However, there is potential to prove quantitative properties on RoboChart models involving ANN components using the technique presented in [33].

The work in [28] considers ANN components in the context of Simulink diagrams [24]. Similar to what we do, the goal is to use ANN components to replace Simulink controllers. System-level properties are proved using the Simulink Design Verifier [29]. The use of Simulink enables automation and links to other (industry) tools, including, for example, a C code generator, and the use of the CMBC model checker [7] to verify its properties. That work, however, does not provide a formal semantics for ANNs or a general notion of conformance.

VEHICLE [8] is a DSL with support for the specification of properties of ANN components using HOL, the base logic of Isabelle. It is based on lambda calculus supporting arithmetic, vectors, and logic. The tool can translate VEHICLE specifications to Marabou input and a loss function for training, an activity we do not cover. System-level verification, however, is not considered in [8], but the authors indicate that this can be enabled via a separate prover.

Isabelle also verifies ANNs in [4], where the authors formalise concepts common to ANNs. The models are platform-independent and can be used to verify properties using Isabelle's theories for real arithmetic. There are two encodings of an ANN: the 'textbook' style, as graphs, and layer style, capturing Tensorflow layers. This work has close links to practical file formats for the specification of hyperparameters; it can support the automated interpretation of a network trained in TensorFlow in Isabelle and can be used to establish the correctness of transformations from one file format to another. That work is well aligned with the proof of properties of classification networks. Unlike us, however, they are concerned with ANNs in isolation.

3 Overview

Figure 1 gives an overview of our work and tools. Our modelling approach is mechanised as part of RoboTool [31], a set of Eclipse plug-ins that implements RoboChart and related notations. In the mechanisation of our extension of RoboChart, we leverage the Eclipse Modelling Framework (EMF) [5] to implement its metamodel, and Sirius [32] to enable graphical modelling. We enable textual editing of models via an Xtext grammar [2]. The well-formedness conditions are mechanised through a validation checker written in Xtend [2]. A pre-release version of this tool is available online[1].

We have mechanised our semantics of RoboChart ANNs using a model-to-text translation from RoboChart to CSP, enabling FDR [12] for model checking. This requires a severe abstraction for scalability, and because FDR cannot deal with real numbers, it is not suitable for our work here. We outlined these semantics in [1], but we generated our semantics manually in that work.

For verification with Isabelle, we translate RoboChart to *Circus* using a model-to-model transformation implemented in Epsilon [19]. We then generate a textual artefact that defines the semantics in a format that is accepted by our encoding of *Circus* in Isabelle, called IsaCircus[2].

We require a reachability condition expressed as a predicate on a vector space to use ANN-specific tools such as Marabou. To obtain this condition formally, we use Hoare and He's Unifying Theories of Programming [15] (UTP) and its mechanisation in Isabelle, Isabelle/UTP [9]. Using a UTP theory for *Circus*, we can give a predicate semantics for RoboChart, including ANN components, and establish verification conditions for conformance in terms of reachability conditions. Here, we show how to use Marabou [18] to prove such conditions.

All engineering activities associated with RoboTool, from validation to semantics generation, are automated. The automated translation from IsaCircus to Isabelle/UTP is in progress, using the semantics in [27].

The following section presents how we define ANN components in RoboChart.

4 RoboChart with ANN Components

This section presents our RoboChart extension: metamodel, well-formedness conditions (Sect. 4.1), and semantics (Sect. 4.2). A full account is in [6].

4.1 Metamodel and Well-Formedness

Our extension of RoboChart adds a few classes to its metamodel, shown in Fig. 2. They introduce the concept of an ANN as an abstract class. Instances of ANN can be a RoboChart controller—our focus here—or an operation. Figure 2 shows

[1] https://github.com/UoY-RoboStar/robotool-ann/releases/tag/NFM-19122024.
[2] https://github.com/UoY-RoboStar/NFM2025/tree/main/RoboChart2IsaCircus.

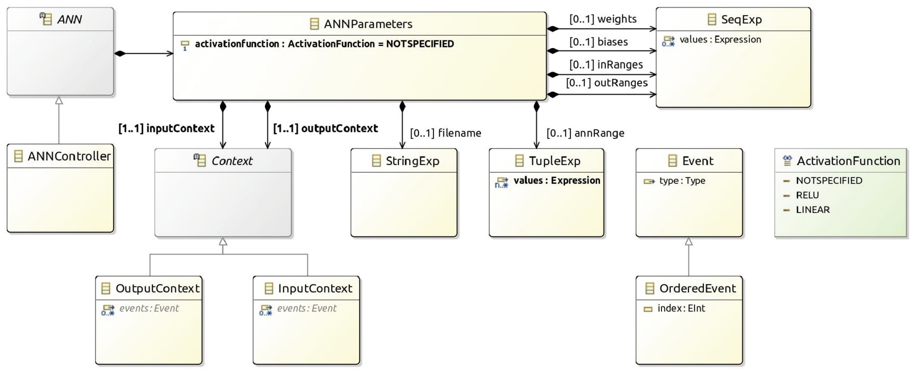

Fig. 2. Metamodel for ANN components in RoboChart: classes in grey are abstract, attributes in grey are inherited, and attributes in bold are not optional. (Color figure online)

ANNController, which inherits from another class, GeneralController, omitted in Fig. 2. A controller in RoboChart is typically used to represent functionality allocated to a computational unit or a self-contained architectural component. Our extension allows a controller to be defined by an ANN.

An ANN includes ANNParameters to define the hyperparameters and the trained parameters of an ANN component, as shown in Fig. 2. The activation function is given using an enumerated type. Here, we give semantics only to an ANN that uses ReLU, which is suitable for using Marabou in reasoning. The use of our semantics in conjunction with other functions is straightforward. Moreover, given the nature of process algebra, explicitly devised to model networks (of processes), our overall approach is suitable for any ANN structure.

We capture the input layer, including its size, through an inputContext and the output layer by an outputContext. Such a context can be used to define, possibly via interfaces, input and output events to connect the ANN to other RoboChart components. Context is an existing RoboChart concept used to define the interface of every component. Each Event in an inputContext or outputContext must be of a new class, OrderedEvent, which adds an integer to an Event definition to specify an order for the inputs and outputs of an ANN component.

We capture the trained parameters through references weights, a three-dimensional tensor (represented as a triple-nested sequence), and biases, a matrix (a double-nested sequence). The class SeqExp captures sequence expressions.

The range to which an ANN is normalised is captured through a pair annRange. (Typically, this range is between -0.5 and 0.5, or 0 and 1.) We capture the range that each input value can take through inRanges, and the range for outputs with outRanges: both are sequences of pairs. Normalisation is a common consideration when defining an ANN; it involves scaling all input ranges in the training data to a new range with a mean close to 0 [22].

The reference filename supports using ANN parameter files (using formats such as ONNX, for instance) instead of explicitly defining parameter values.

Our well-formedness conditions for ANN assert that the parameters are defined either in the model itself or in a linked file. They also ensure that the trained parameters' size and shape correspond to the hyperparameters and that the size of the normalisation sequences matches the number of inputs and outputs to the component, and, for each element in these sequences representing a range, the maximum is strictly greater than the minimum. Finally, at the RoboChart model level, we ensure that connections to and from the ANN component are in accordance with its definition of input and output events.

Next, we present the semantics of these well-formed ANN components.

4.2 Formal Semantics

This section presents our denotational semantics. We introduce *Circus*and then provide an overview of the semantics and present key definitions.

Circus. *Circus* [27] is a process-algebraic language for specification of concurrent systems; it defines data models and control behaviour of independent components: processes. State declarations and actions define their behaviour; every process has one main action that defines its behaviour. Actions are similar to CSP processes [14] but define stateful behaviour. The definition of a basic *Circus* process has the form: **process** $Proc \mathrel{\widehat{=}}$ **begin** $\ldots \bullet A$ **end**. Here, the process is named $Proc$ and has main action A. The body of the definition, between **begin** and **end**, contains declarations of state variables and actions that are local to $Proc$. We describe selected *Circus* action operators used in our semantics in Table 1; further details are provided as needed. A *Circus* process can also be defined in terms of other processes, using process operators similar to CSP's. We do not use a composition of processes here, but it is via these operators that we can combine the semantics of an ANN controller, which is a *Circus* process, with those for other components of a RoboChart model.

The semantics of *Circus* [27] is defined using Unifying Theories of Programming [15] (UTP). It defines processes and actions as reactive contracts [10]: alphabetised predicates that capture their reactive behaviour. The UTP-based reactive contracts semantics is ideal for supporting theorem proving.

Overview. As mentioned earlier, the semantics of an ANN controller is a *Circus* process. Figure 3 depicts the structure of the main action of such a process. We capture every node of the ANN as a *Circus* action and then define each layer as the parallel composition of these actions and the whole ANN as the parallel composition of the layer actions. We treat the input nodes as input for the first layer. The parallel composition of node actions, represented by the parallel lines between them in Fig. 3, and the parallel composition of layer actions define the expected data flow.

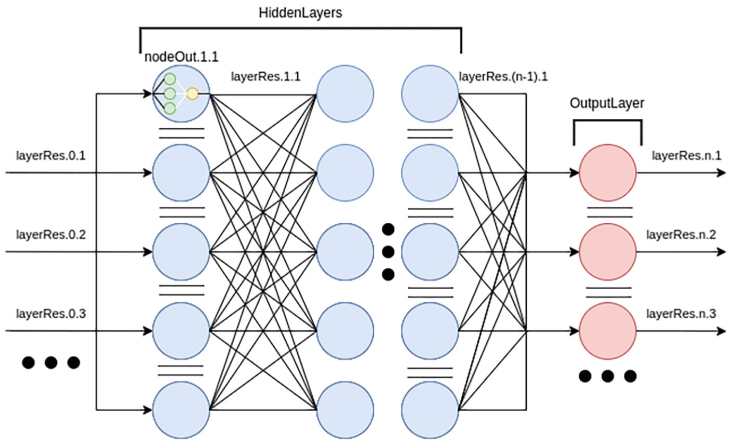

Fig. 3. A diagram showing the structure of our semantics of ANN controllers. Circles represent actions representing nodes, and labelled edges represent communications. The parallel lines denote parallel composition between actions. We have parallel composition at the level of nodes and the level of layers. The ellipses indicate that we allow for an arbitrary hidden layer structure and support any input and output layer size. On the top left-hand corner, we also show the internal structure of a node action.

Table 1. Selected *Circus* action operators. Here, we use A and B as metavariables to stand for actions, cs for a set of channels, e for an event, i for an index, and T for a finite type. For the replicated (iterated) operators, $A(i)$ is an action identified i.

Symbol	Name	Symbol	Name
Skip	Skip	$e \longrightarrow A$	Prefix
$A \; [\![\mid cs \mid]\!] \; B$	Parallel Composition	$[\![cs]\!] \, i : T \bullet A(i)$	Replicated Parallel
$A;\ Q$	Sequential Composition	$;\ i : T \bullet A(i)$	Replicated SequentialComposition
$c?x \longrightarrow A$	Input	$c!e \longrightarrow A$	Output
$(c)\&A$	Guarded Action	$A \setminus cs$	Hiding
$A[e := e1]$	Renaming	$A \triangle B$	Interrupt
$A \mid\mid\mid B$	Interleaving		

The Exchange of information between nodes, represented by the lines connecting node actions in Fig. 3, is captured by CSP events on a channel *layerRes*. The event *layerRes*.l.n is for communication from the nth node of the l-th layer. A *layerRes*.0.n event represents the n-th input of the ANN, and *layerRes*.*layerNo*.n, the n-th output; here, *layerNo* is the number of layers.

Each node action is defined by the parallel composition of actions that receive communications from the previous layer, passing this information, after applying the node's weight, to another action that calculates the node's overall output. The intra-node action communications are via a channel *nodeOut*. A *nodeOut*.l.n.i event is for the i-th input of the node n in layer l.

Rule 1. Semantics of ANN Components $\underline{[\![\mathsf{c} : \mathsf{ANNController}]\!]_{\mathcal{ANN}} : \textbf{Program} =}$

$\underline{\mathsf{ANNChannelDecl(c)}}$
$\underline{\mathsf{ANNConstants(c)}}$
$\underline{\mathsf{ANNProc(c)}}$

Semantic Rules. We formalise our semantics via a set of rules that together define a function $[\![\mathsf{C}]\!]_{\mathcal{ANN}}$ from a RoboChart ANN controller C to a *Circus* program including some channel and constant declarations, and a process. This semantics fits into the definition of a process formalising a RoboChart model that includes the ANN component as specified in [1]. Here, we provide an overview of these rules; the complete set can be found in [6, Chapter 3].

A rule definition provides a number and a brief description, followed by the declaration of the function defined by the rule and an expression in a metalanguage to specify the function. In that expression, elements of the metalanguage are underlined. Our top-level rule (Rule 1), of the top-level *Circus* syntactic category Program, is defined by three functions: *ANNChannelDecl*, which specifies channel declarations (as described above); *ANNConstants*, which specifies constants; and *ANNProc*, which gives the behaviour of the process. We define *ANNProc* in Rule 2; the complete definitions of *ANNChannelDecl* and *ANNConstants* are omitted here but can be found in [6].

The rule *ANNConstants* first records constants corresponding to attributes of C: *weights*, *biases*, *annRange*, *inRanges*, *outRanges*, and *layerstructure*. Further, the rule defines constants for the function *relu* (the activation function), the input to each layer *layerInput* (derived from the controller C), and the normalisation functions *norm*, *normIn*, and *denormOut*. Our semantics defines a normalised ANN, so we normalise every input using the function *normIn*, then denormalise every output using *denormOut*. These functions are defined as *norm*, which scales a value from one range to another.

The constants *layerstructure* and *layerInput* give the shape of our *Circus* semantics; for our example, these take the values as shown below. We obtain these constants from the ANN component in RoboChart, AnglePIDANN: we present a graphical representation of this component in Fig. 5. We use these constants to illustrate our semantic rules throughout this section.

$$layerstructure = \langle 1, 1 \rangle$$
$$layerInput = \langle 2, 1, 1 \rangle$$

The process, an element of the *Circus* syntactic category ProcDecl, is defined by the function ANNProc(C) defined in the Rule 2 presented here. That process is named according to the name attribute of C. Its main action is $\underline{\mathsf{CircANN(C)}}$, shown after the •. This action uses the local actions *ANN* and *Interpreter*, which

Rule 2. Function ANNProc $\mathsf{ANNProc(C)} : \mathsf{ProcDecl} =$

```
$\mathbf{process}\ \underline{\mathsf{C.name}} \mathrel{\widehat=} \mathbf{begin}$
 $Collator \mathrel{\widehat=} l, n, i : \mathbb{N};\ sum : Value \bullet$
   $(i = 0) \& layerRes.l.n\,!(relu(sum + (biases(l)(n)))) \longrightarrow \mathbf{Skip}$
   $\Box$
   $(i > 0) \& nodeOut.l.n.(layerInput(l) - i + 1)\,?x \longrightarrow$
             $Collator(l, n, (i - 1), (sum + x))$
 $Edge \mathrel{\widehat=} l, n, i : \mathbb{N} \bullet$
   $layerRes.(l - 1).i\,?x \longrightarrow nodeOut.l.n.i\,!(x * (weights(l)(n)(i))) \longrightarrow \mathbf{Skip}$
 $Node \mathrel{\widehat=} l, n, inpSize : \mathbb{N} \bullet$
   $((;\ i : 1 \mathinner{..} inpSize \bullet Edge(l, n, i))$
     $[\![ \mid \{\!|\ nodeOut.l.n |\!\} \mid ]\!]$
    $Collator(l, n, inpSize, 0)) \setminus \{\!|\ nodeOut.l.n |\!\}$
 $HiddenLayer \mathrel{\widehat=} l, n, inpSize : \mathbb{N} \bullet$
   $([\![ \{\!|\ layerRes.(l - 1) |\!\} ]\!]\, i : 1 \mathinner{..} s \bullet Node(l, i, inpSize))$
 $HiddenLayers \mathrel{\widehat=} \underline{\mathsf{HiddenLayers(C)}}$
 $OutputLayer \mathrel{\widehat=} \underline{\mathsf{OutputLayers(C)}}$
 $ANN \mathrel{\widehat=} (HiddenLayers [\![ \mid \{\!|\ layerRes.(\underline{\mathsf{layerNo(C)} - 1}) |\!\} \mid ]\!] OutputLayer);\ ANN$
 $Interpreter \mathrel{\widehat=} \underline{\mathsf{Interpreter(C)}}$
 $\bullet \underline{\mathsf{CircANN(C)}}$
 $\mathbf{end}$
```

capture the data flow of the ANN, to define its behaviour within the RoboChart context: using the input and output events of the RoboChart model, dealing with normalisation, and handling termination.

For example, we consider the RoboChart module SegwayANN partially shown in Fig. 5. It has two controller blocks: SegwayController defined by a state machine, and AnglePIDANN defined by an ANN. We give the complete semantics of AnglePIDANN in Fig. 4, which was generated using Rule 2. In the semantics of AnglePIDANN, the action *ANN* captures the behaviour of the ANN in terms of its hyperparameters and its trained parameters. The action *Interpreter* captures normalising all input communications to the ANN, then denormalising all output communications from the ANN. Using these actions, we define the main action of the process for *AnglePIDANN* in Fig. 4 using parallel composition ($A\ [\![\mid cs \mid]\!]\ B$), hiding ($\setminus$), and interrupt ($\triangle$).

The parallel composition of *Interpreter* and *ANN* captures the behaviour of our ANN in terms of the input and output events of the controller. The definition of the simple *Interpreter* action is determined by the semantic function $\underline{\mathsf{Interpreter(C)}}$, which is omitted here. *Interpreter* takes inputs for the controller in any order and outputs their normalised values, in any order, via $layerRes.0$ events to *ANN*. It also takes outputs from *ANN* via $layerRes.\underline{\mathsf{layerNo}}$ (where $\underline{\mathsf{layerNo}}$ is the index of the last layer) events and outputs their denormalised

$$
\begin{array}{l}
\textbf{process } \mathit{AnglePIDANN} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \ldots \\
\quad \mathit{HiddenLayers} \mathrel{\widehat{=}} \mathit{HiddenLayer}(1,1,2) \\
\quad \mathit{OutputLayer} \mathrel{\widehat{=}} [\![\{\!|\, \mathit{layerRes}.1 \,|\!\}]\!]\ i : 1 \mathinner{..} 1 \bullet\ \mathit{Node}(l,i,1)) \\
\quad \mathit{ANN} \mathrel{\widehat{=}} (\mathit{HiddenLayers} [\![\, | \ \{\!|\, \mathit{layerRes}.1 \,|\!\} \ | \,]\!] \mathit{OutputLayer})\ ;\ \mathit{ANN} \\
\quad \mathit{Interpreter} \mathrel{\widehat{=}} \\
\quad\quad (\mathit{anewError\_in}?x \longrightarrow \mathit{layerRes}.0.1.(\mathit{normIn}(1,x)) \longrightarrow \textbf{Skip}\ |\!|\!| \\
\quad\quad \mathit{adiff\_in}?x \longrightarrow \mathit{layerRes}.0.2.(\mathit{normIn}(2,x)) \longrightarrow \textbf{Skip}); \\
\quad \mathit{layerRes}.2.1?y \longrightarrow \mathit{angleOutputE\_out}.(\mathit{denormOut}(1,y)) \longrightarrow \textbf{Skip}; \\
\quad \mathit{Interpreter} \\
\quad \bullet\ ((\mathit{Interpreter}\ [\![\, | \ \{\!|\, \mathit{layerRes}.0, \mathit{layerRes}.2 \,|\!\} \ | \,]\!]\ \mathit{ANN}) \setminus \{\!|\, \mathit{layerRes} \,|\!\}) \\
\quad\quad\quad\quad \triangle\ \mathit{terminate} \longrightarrow \textbf{Skip} \\
\quad \textbf{end}
\end{array}
$$

Fig. 4. *Circus* semantics for the ANN controller *AnglePIDANN*, generated via an application of Rule 2. The ellipsis (...) denotes that the definition of the *Collator*, *Edge*, *Node*, and *HiddenLayer* actions are constant in Rule 2, so are omitted here.

values. Figure 4 presents the *Interpreter* action for our example; here, we have two inputs (*anewError_in* and *adiff_in*), and one output (*angleOutputE_out*). *Interpreter* first behaves as the interleaved composition of two sub-actions, each of which accepts an input event and then communicates the normalised (with *normIn*) value of this event to a *layerRes*.0 event. Next, *Interpreter* waits on the single output event of the *ANN* action (*layerRes*.2.1); when received, it outputs the denormalised (with *denormOut*) value on the *angleOutputE_out* channel. Finally, *Interpreter* repeats and waits for fresh input events.

In *Circus*, $A\ [\![\, |\ cs\ |\,]\!]\ B$ defines the parallelism of actions A and B, which can perform any events outside of cs set independently but must engage on any event in cs. Here, *Interpreter* and *ANN* synchronise on the set containing all *layerRes*.0 and $\mathit{layerRes}.\underline{\mathsf{layerNo}}$. We hide all communications on *layerRes*, so *Proc* defines interactions over the inputs and outputs of the controller, as expected: *anewError_in*, *adiff_in*, *angleOutput_out*.

Finally, the main action of an ANN controller process can be interrupted (operator $\triangle$) by an event *terminate*, raised if all the controllers of the RoboChart terminate, and so the whole software, including the ANN, also terminates (**Skip**).

ANN is defined by the parallel composition of actions *HiddenLayers* and *OutputLayer*, synchronising on $\mathit{layerRes}.\underline{(\mathsf{layerNo}(\mathsf{C}) - 1)}$ events, and sequentially composed with *ANN* in a tail recursion. *HiddenLayers* and *OutputLayer* are defined by semantic functions (omitted here) that compose in parallel actions for the hidden layers and the output layer's nodes. *HiddenLayer* has parameters l, n, and *inpSize* and captures the semantics of the lth layer with n nodes and *inpSize* inputs coming from the previous layer or to the ANN as a whole. In our example, shown in Fig. 4, the ANN component has only a single hidden layer, so *HiddenLayers* is simply *HiddenLayer*(1, 1, 2), denoting the 1st layer with 1 node and 2 inputs. We can derive the number and size of the hidden layers using

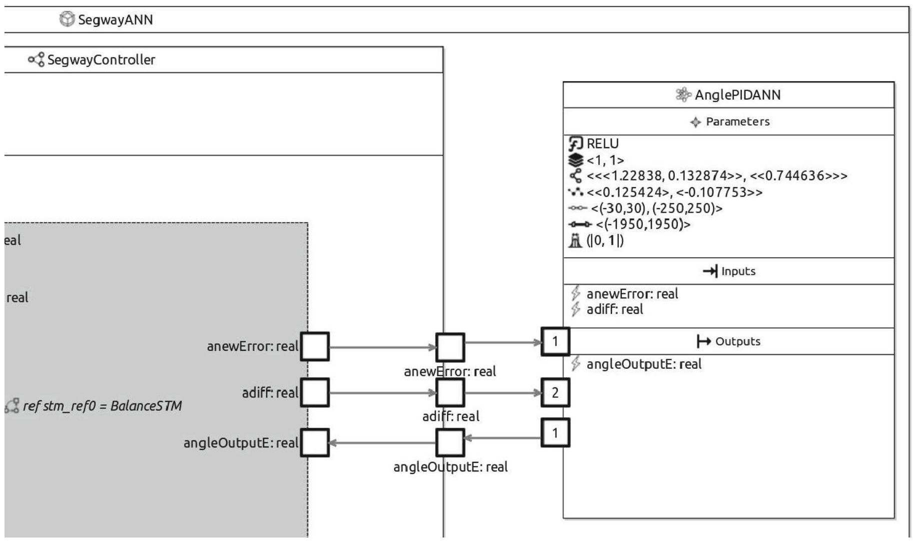

Fig. 5. Part of RoboChart module with an ANN for segway control defined in the block AnglePIDANN. The first parameter is the activation function, ReLU. The second parameter is the layer structure: $\langle 1, 1\rangle$. The third and fourth parameters are the trained parameters: the weights and biases. The remaining three parameters are associated with normalisation: the *inRanges*, the *outRanges*, and the *annRange*.

the *layerstructure* sequence, and we know the input size of each layer through either that sequence or the number of elements in the input context—the input size. We define *OutputLayer* as a layer with no parameters; in our example, see Fig. 4, *OutputLayer* is the distributed parallel composition of just 1 *Node* action, because our example has one node in its output layer (the last element of *layerstructure* denotes the size of the output layer). The definition of this action is similar to that of *HiddenLayer* in Rule 2.

The definition of *HiddenLayer* uses a replicated alphabetised parallelism $[\![cs]\!] i : T \bullet A(i)$ composing actions $A(i)$ in parallel, with i ranging over T synchronising on all events in the set cs. For *HiddenLayer*, the index i ranges from 1 to n, composing n processes $Node(l, i, inpSize)$ which capture the semantics of the ith node of the layer l. Synchronisation is on the $layerRes.(l-1)$ events representing the inputs to the layer (see Fig. 3).

$Node(l, n, inpSize)$ is defined by the replicated sequential composition (;) of *inpSize* actions $Edge(l, n, i)$, in parallel with a *Collator* action. *Edge* actions collect inputs from the $layerRes.(l-1)$ channels providing those values to *Collator* after applying the *weights* via the *nodeOut* channel. *Collator* sums its inputs to define the node output, communicated via $layerRes.l$. This output includes the bias and reflects the output of the activation function *relu*. This action uses a one-based index for *nodeOut* events to match the sequential composition of *Edge* actions; we use the sequence *layerInput* to define this index, where $layerInput(l)$

is the size of layer $l-1$, or the input size for the first layer. The definitions of *HiddenLayer*, *Node*, *Edge*, and *Collator* are identical for every ANN.

Our semantics is fully formalised by ten rules in [6]. The automatic generation of the semantics is also implemented, as described next.

5 Mechanisation in Isabelle

In this section, we describe the mechanisation of our semantics (Sect. 5.1) and its use for automated verification using Isabelle and Marabou (Sect. 5.2).

5.1 IsaCircus

To verify RoboChart models in Isabelle, we have created IsaCircus, a mechanisation of *Circus* in Isabelle. This section introduces the IsaCircus syntax and outlines its foundational types and operators. A key feature of our approach is the abstraction of actions and processes, providing a modular and flexible modelling framework while ensuring concise and manageable definitions.

IsaCircus has an abstract type for *Circus* actions, `('e,'s)action`, parametrised by two types: `'e` for events and `'s` for state. The event type `'e` identifies the set of all events in scope for the action. This type declaration is pivotal, as actions are the fundamental building blocks of *Circus* processes. To distinguish processes from actions, IsaCircus has a type `'e process`, defined as a special action whose state type is `unit`, since processes have no externally visible state. Finally, IsaCircus has a type `('a, 'e)chan` for a channel communicating values of type `'a` to define an event of type `'e`. Listing 1 sketches the IsaCircus encoding of the process for the AnglePIDANN controller defined by Rule 2.

In IsaCircus, channels are declared with the **`chantype`** keyword, which establishes the complete set of events for the model. In our example, we declare an event type `ANNChan` (line 1 in Listing 1). It includes all channels used in the semantics. Each channel is explicitly typed; for instance, `adiff_in` is of type `real`, reflecting its role in transmitting real-valued data.

Isabelle `definitions` can be used directly for channel sets and global definition (called axiomatic definitions in Z and *Circus*). In Listing 1, we show the definition of the activation function `relu` (line 7).

A process is encoded using a **`locale`**, which provides a separate namespace to introduce the process's definitions, actions, and main action. In Listing 1, we show part of the **`locale`** for `AnglePIDANN` (lines 9–23).

Actions are specified using the **`actions`** keyword. In Listing 1, we show three of the nine actions of `AnglePIDANN`, among which the main action `MainAction` is identified as `CircANN` (line 22) using the `cProcess` operator.

IsaCircus defines the *Circus* action operators as Isabelle semantic constants. There are constants for each operator in Table 1. For example, the constant shown below defines the generalised parallelism operator `cparallel`.

```
cparallel :: "('e,'s) action ⇒ 'e set ⇒ ('e,'s) action ⇒ ('e,'s) action"
```

Listing 1. AnglePIDANN example in IsaCircus

```
chantype ANNchan = terminate        :: "unit"
                   layerRes         :: "nat × nat × real"
                   nodeOut          :: "nat × nat × nat × real"
                   adiff_in         :: "real"
                   ...

definition relu :: "real ⇒ real" where "relu x = max 0 x"

locale <AnglePIDANN>
begin
actions is "(ANNchan, unit) action" where

"Collator(l, n, i :: nat, sum :: real) =
    (i = 0)&layerRes.l.n!relu(sum + biases(l)(n)) → Skip □
    (i > 0)&nodeOut.l.n.(layerInput(l) - i + 1)?x →
         Collator(l,n,(i-1),(sum+x))"  |
...
"ANN = (HiddenLayers ⟦| ⦃layerRes.1.1⦄ |⟧ OutputLayer);; ANN" |

"CircANN = ((Interpreter ⟦| ⦃layerRes.0, layerRes.2⦄ |⟧ ANN) \ ⦃layerRes⦄)
               △ (terminate → Skip)"

definition "MainAction = cProcess CircANN"
end
```

The function `cparallel` takes two actions and a set of events (`'e set`) on which the actions synchronise, and returns an action as their parallel composition.

Syntactic constants are defined for each of the semantic constants. For example, instead of using `cparallel`, we can use the infix brackets of the *Circus* notation when defining a parallelism. As illustrated in Listing 1, the notation of the IsaCircus definition of `AnglePIDANN` is very similar to that indicated in Rule 2. The complete syntax definition is available online[3].

5.2 Verification

Verification is facilitated through reactive contracts [10,11], which describe how a *Circus* process evolves regarding state updates, event traces, and refused events. A reactive contract is a triple consisting of a precondition (P_1), a pericondition (P_2), and a postcondition (P_3), as shown below:

$$[\, P_1(tt, s) \vdash P_2(tt, s, ref') \mid P_3(tt, s, s') \,]$$

The precondition is a predicate over the trace variable (tt) and the initial state variable (s), which characterises the non-divergent behaviours. The pericondition ranges over tt, s and the refusal set (ref'), characterising behaviours where the process awaits interaction. Finally, the postcondition ranges over tt, s, and the final state (s'), characterising terminating behaviours.

Reasoning with the UTP and reactive contracts is based on refinement. Reactive contracts can be used for both specifications of reactive behaviour and also

[3] https://github.com/UoY-RoboStar/NFM2025/tree/main/IsaCircus_syntax.

to provide a denotational semantics for *Circus* actions and processes. Then, refining one contract by another requires we weaken the precondition and strengthen the peri- and postconditions [10]. However, refinement in and of itself is inadequate to reason about an ANN since outputs can have an error. We define a notion of conformance based on refinement that can be used to compare an ANN controller to other controllers, taking into account a precision parameter ϵ. We identify verification conditions sufficient to establish conformance that can be discharged using Isabelle or SMT-based tools, such as Marabou, in conjunction with Isabelle.

As specifications for ANNs, we consider cyclic memoryless RoboChart controllers, which perform all inputs before any outputs and then repeat, with no memory between cycles. We denote these controllers with *StandardController*. The semantics of such a controller as a reactive contract follows a pattern that captures the relationship between the input and output communications using a predicate p that ranges over the trace. Equally, our semantics for an ANN defined by Rule 2 can be captured as a reactive contract of a particular pattern. We describe the patterns and how we obtain contracts instantiating them in [1].

Both the specification and ANN patterns follow tail recursions. So, the compositionality of conformance allows us to focus on the verification that a single iteration of the ANN conforms to a single interaction of the cyclic controller. Definition 1 captures the pattern for a single iteration of an ANN.

Definition 1 (ANN Reactive Contract Pattern).

$$\begin{array}{ll}
[& \mathit{true} \\
\vdash & \#\mathit{in} < \mathit{insize} \land \mathit{tt} = \mathit{in} \land \{\!|\ \mathit{layerRes}.0.(\#\mathit{in}+1) |\!\} \not\subseteq \mathit{ref}' \\
 & \lor \\
 & \#\mathit{in} = \mathit{insize} \land \\
 & \exists\, l : 1\,..\,\mathit{layerNo} \bullet \exists\, n : 1\,..\,\mathit{layerSize}(l) \bullet \\
 & \quad \mathit{tt} = \mathit{front}(\mathit{layeroutput}(l, n, \mathit{in})) \land \\
 & \quad \mathit{last}(\mathit{layeroutput}(l, n, \mathit{in})) \notin \mathit{ref}' \\
\mid & \#\mathit{in} = \mathit{insize} \land \mathit{tt} = \mathit{layeroutput}(\mathit{layerNo}, \mathit{layerSize}(\mathit{layerNo}), \mathit{in})\ ]
\end{array}$$

An instance of this pattern above for the parallelism between *HiddenLayers* and *OutputLayer* (see Rule 2) is a reactive contract L that has a *true* precondition, indicating that the ANN cannot diverge. The pericondition of L is a predicate specifying that, in an intermediate state, the trace of events observed so far includes only $\mathit{layerRes}.0.n$ events (denoted by *in* here), without repeated values for n, and that $\mathit{layerRes}.0.m$ events that have not been observed are not refused. Alternatively, the trace can include $\mathit{layerRes}.0.n$ events, for all values of n, followed by $\mathit{layerRes}.m.n$ events, for unique values of m and n, but covering all but one $\mathit{layerRes}.\underline{\mathsf{layerNo}}$ event. The postcondition states that the last output is added to the trace, and the action terminates. We use an instance of this pattern to define *ANNController*: the semantics of our ANN components.

We specify the traces that include events other than inputs (*layerRes*.0 events) using a function *layeroutput*. With $layeroutput(l, n, in)$, we get the trace up to the point where the n-th node of the layer l has produced its output, with input *in*. Here, *in* is a trace of input events, and l and n are natural numbers.

The definition of $layeroutput(l, n, in)$ uses a function $annout(l, n, inv)$, which specifies the value communicated by the n-th node of the layer; here, *inv* is the sequence of values of the inputs defined in the trace *in*. We can automatically extract the definition of this function from our process-algebraic semantics.

Next, we describe our notion of conformance and our verification approach, which we apply to the patterns for the semantics of controllers.

The relation $Q\ conf(\epsilon, tc)\ P$ defined below allows for a tolerance of ϵ on the values communicated by the process Q via channels in the set tc when comparing it to a process P. We accommodate this tolerance to capture the numerical instability present in the predictions of a machine learning model: all other communications of the system should remain precise.

Definition 2. $P\ conf(\epsilon, tc)\ Q \Leftrightarrow (Approx\ \epsilon\ tc\ Q) \sqsubseteq P$

Ultimately, conformance requires refinement ($\sqsubseteq$), but the specification Q is modified using the function $Approx\,\epsilon\,tc$. For every possible communication that Q can engage in through tc, we have that $Approx\,\epsilon\,tc\,Q$ can engage in any communication where the value carried varies by up to ϵ.

Next, we present a theorem that establishes verification conditions that can be discharged to prove conformance and avoid proof from first principles.

Theorem 1 (Verification condition for conformance)

$$\begin{array}{l}(\forall\, x_1, .., x_{insize} : Value \bullet \forall\, y_1, .., y_{outsize} : Value \mid p \bullet \forall\, i : 1 \,..\, outsize \bullet \\ \qquad \mid denormO(i, annout(layerNo, i, inpv)) - y_i \mid \leq \epsilon) \\ \Rightarrow ANNController\ conf(\epsilon, \{\!|\ out |\!\})\ StandardController\end{array}$$

where ANNController and StandardController are instances of the patterns for reactive contracts for ANN controllers and for cyclic memoryless controllers; out denotes the output channels of these patterns; the predicate p is the part of the pericondition and postcondition of the instance StandardController that relates its inputs to its outputs; and the sequence of input values inpv is given by $\langle normI(1, x_1), .., normI(insize, x_{insize})\rangle$.

The verification condition identified by Theorem 1 requires that for all sequences of inputs and all sequences of outputs, whose values x_j and y_i are related by a predicate p arising from the *StandardController* specification, the outputs of the ANN must be acceptable. For example, p for the RoboChart Controller AnglePID is a function $y_1 = P(x_1) * D(x_2)$. Precisely, acceptability requires that for each i indexing an output y_i, the output of the ANN does not differ from y_i by more than ϵ. The i-th output of the ANN is defined by $denormO(i, annout(layerNo, i, inpv))$, in terms of *annout*. Here, *inpv* is the sequence of input values obtained by the normalisation of each input x_i, that is, $\langle normI(1, x_1), .., normI(insize, x_{insize})\rangle$.

For improved automation of verification of RoboChart models, we provide a further result that can justify the combined use of IsaCircus and Marabou. This is possible when the domain of each input is already bounded, and the predicate p from Theorem 1 defines a monotonic function F on sequences of input and output communications. In this situation, we can use the verification condition shown in Theorem 2.

The input to Marabou needs to define a split of the domain containing all possible input values. A split is characterised by a constant *noInt*, representing the number of closed intervals into which the range of every input value x_i is divided. The smaller the value of ϵ, the larger the value of *noInt* that is required. For each of the *noInt* intervals int_i in a split, we need to provide to Marabou its lower bound $int_i.min$ and its upper bound $int_i.max$. A valid split for an ANN needs to satisfy the restriction that, for every value v of every input x_j, there is an interval int_i such that $int_i.min \leq v \leq int_i.max$.

For Marabou, given the value of ϵ, we need to give a precision value δ. We need to provide this to account for the fact that Marabou does not support an open interval as a specification for its properties due to its SMT solver backend. We use a closed interval in conformance, so encoding the negated condition would require an open interval. Given that the ANN is already normalised to the range of our RoboChart controller *StandardController*, we can prove conformance in Marabou with the verification condition as shown below.

Theorem 2 (Verification conditions for Marabou)

$$
\begin{array}{l}
\neg\ \exists\, x_1, ..x_{insize} : Value \bullet \exists\, y_1, ..y_{outsize} : Value \bullet \\
\quad \langle int_1.min, ..., int_{(noInt)}.min \rangle\ F\ \langle y_1, ..y_{outsize} \rangle \land \\
\quad \exists\, y'_1, ..y'_{outsize} : Value \bullet \\
\qquad \langle int_1.max, ..., int_{(noInt)}.max \rangle\ F\ \langle y'_1, ..y'_{outsize} \rangle \land \\
\qquad \exists\, i : 1 \,..\, outsize \bullet \\
\qquad\quad annout(layerNo, i, \langle x_1, ..x_{insize} \rangle) \leq y'_i - \epsilon - \delta \lor \\
\qquad\quad annout(layerNo, i, \langle x_1, ..x_{insize} \rangle) \geq y_i + \epsilon + \delta \\
\Rightarrow ANNController\ conf(\epsilon, \{\!|\ out |\!\})\ StandardController
\end{array}
$$

Theorem 2 states that, given that all intervals are valid, there does not exist an input valuation (x_i) such that either of the following conditions holds. First, *annout* is less than or equal to F evaluated at the maximum point of the interval (y'_i) under ϵ. Second, *annout* is greater than F evaluated at the minimum point of the interval (y_i) with ϵ applied. If no such input valuation exists, then *annoutput* must be within F under all possible values of x.

One query in Marabou encodes this verification condition for every interval int_i. If Marabou returns UNSAT for every condition, we can use the value of ϵ in Isabelle soundly for conformance via a certificate that can automatically be obtained through Marabou [16]. Such certificates can be reconstructed using CVC4 and CVC5 in Isabelle [21]; this would constitute progress towards integrating Marabou with Isabelle via Sledgehammer [3].

By applying Theorem 2 to our example with AnglePID and AnglePIDANN, we obtain a condition of the form below.

$$
\begin{array}{l}
\neg\ \exists\, x_1, x_2 : \mathit{Value} \bullet \\
\quad \forall\, i : 1\,..\,2 \bullet \\
\qquad \mathit{int}_i.\mathit{min} \leq x_i \leq \mathit{int}_i.\mathit{max} \land \\
\qquad y_1 \geq (P(\mathit{int}_i.\mathit{min}) + D(\mathit{int}_i.\mathit{min}) + \epsilon + \delta) \lor \\
\qquad y_1 \leq (P(\mathit{int}_i.\mathit{max}) + D(\mathit{int}_i.\mathit{max}) - \epsilon - \delta)
\end{array}
$$

When we instantiate the constants in the conditions above, we obtain an ϵ error value of 0.085. We use the range $\{0..1\}$ as *Value*, given our normalisation assumption. We have also instantiated F as a function capturing the behaviour of the RoboChart controller AnglePID in terms of the constants P and D. Finally, *noInt* is set to 100, and we instantiate δ as $1e - 6$. Thus, we have used Marabou and Isabelle to verify the ANN component AnglePIDANN.

6 Final Considerations

This paper presents an approach that enables process-algebraic verification of intelligent software. As far as we know, this is the first line of work enabling this style of verification for software involving ANNs.

Our approach focuses on the fundamental concepts of ANN models: the underlying logic of ANN models, irrespective of either the context of use, or the size and shape of the specific ANN. Due to its abstraction level, this perspective of modelling and verification accommodates any current and future approaches for training, fine-tuning, or fixing ANN models. Our perspective enables reasoning about the logic and the system context of an ANN controller, and enables us to soundly use component-level reasoning techniques to prove properties of this surrounding context. Our approach also allows us to accommodate future advances in component-level reasoning techniques for ANNs.

Our work suggests several possible future research directions. First, our semantic style naturally allows for a clean definition of multiple types of activation functions. Further, extending our ANN components to support various types of ANN will support work on image and sound recognition, for instance. Integration with other ANN tools is a promising line of work to explore the interplay between theorem proving and ANN solvers. Finally, using process algebraic reasoning, we can extract properties about the propagation of imprecision.

Acknowledgements. The work is partially funded by the UK EPSRC Grants EP/R025479/1 and EP/V026801/2, by the Royal Academy of Engineering Grants No CiET1719/45 and IF2122\183. This work is partially supported by the RoboSAPIENS project funded by the European Commission's Horizon Europe programme under grant agreement number 101133807. Jim Woodcock acknowledges the support of Southwest University and Aarhus University.

References

1. Attala, Z., Cavalcanti, A.L.C., Woodcock, J.C.P.: Modelling and verifying robotic software that uses neural networks. In: Ábrahám, E., Dubslaff, C., Tarifa, S.L.T. (eds.) Theoretical Aspects of Computing, pp. 15–35. Springer, Heidelberg (2023)
2. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing (2016)
3. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reason. **9**(1), 101–148 (2016)
4. Brucker, A.D., Stell, A.: Verifying feedforward neural networks for classification in Isabelle/HOL. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods, pp. 427–444. Springer, Cham (2023)
5. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education, Boston (2003)
6. Cavalcanti, A., et al.: RoboSapiens WP1 D1.1 Report (Draft). https://robostar.cs.york.ac.uk/publications/reports/Draft_RoboSapiensD1_1.pdf'
7. CBMC - Bounded Model Checker for C and C++ programs (2024). http://www.cprover.org/cprover-manual/
8. Daggitt, M.L., Kokke, W., Atkey, R., Slusarz, N., Arnaboldi, L., Komendantskaya, E.: Vehicle: bridging the embedding gap in the verification of neuro-symbolic programs (2024)
9. Foster, S., Baxter, J., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying semantic foundations for automated verification tools in Isabelle/UTP. Sci. Comput. Program. **197**, 102510 (2020)
10. Foster, S., Cavalcanti, A., Canham, S., Woodcock, J., Zeyda, F.: Unifying theories of reactive design contracts. Theor. Comput. Sci. **802**, 105–140 (2020)
11. Foster, S., Ye, K., Cavalcanti, A., Woodcock, J.: Automated verification of reactive and concurrent programs by calculation. J. Logical Algebraic Methods Program. **121**, 100681 (2021)
12. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 - a modern refinement checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 187–201 (2014)
13. Henriksen, P., Lomuscio, A.: Efficient neural network verification via adaptive refinement and adversarial search. In: European Conference on Artificial Intelligence (2020)
14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
15. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle river (1998)
16. Isac, O., Barrett, C.W., Zhang, M., Katz, G.: Neural network verification with proof production. In: 2022 Formal Methods in Computer-Aided Design (FMCAD), pp. 38–48 (2022)
17. Ivanov, R., Jothimurugan, K., Hsu, S., Vaidya, S., Alur, R., Bastani, O.: Compositional learning and verification of neural network controllers. ACM Trans. Embedded Comput. Syst. (TECS) **20**(5s), 1–26 (2021)
18. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
19. Kolovos, D.S., Paige, R.F., Polack, F.: The epsilon object language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_11

20. Kotha, S., Brix, C., Kolter, Z., Dvijotham, K., Zhang, H.: Provably bounding neural network preimages. In: Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23. Curran Associates Inc., Red Hook (2024)
21. Lachnitt, H., et al.: IsaRare: automatic verification of SMT rewrites in isabelle/HOL. In: Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I, pp. 311–330. Springer-Verlag, Heidelberg (2024)
22. LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient BackProp. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) Neural Networks: Tricks of the Trade. LNCS, vol. 7700, pp. 9–48. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35289-8_3
23. Lopez, D.M., Choi, S.W., Tran, H.D., Johnson, T.T.: Nnv 2.0: the neural network verification tool. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 397–412. Springer, Cham (2023)
24. The MathWorks, Inc. Simulink. www.mathworks.com/products/simulink
25. Miyazawa, A., et al.: Diagrammatic physical robot models. Accepted for publication in Softw. Syst. Model (2024)
26. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283. Springer, Heidelberg (2002)
27. Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs Using Circus. PhD thesis, University of York (2006)
28. Raman, R., Gupta, N., Jeppu, Y.: Framework for formal verification of machine learning based complex system-of-systems. Insight **26**(1), 91–102 (2023)
29. Simulink Design Verifier (2024). https://www.mathworks.com/products/simulink-design-verifier.html
30. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC '19, pp. 147–156. Association for Computing Machinery, New York (2019)
31. University of York. RoboChart Reference Manual. www.cs.york.ac.uk/circus/RoboCalc/robotool/
32. Viyović, V., Maksimović, M., Perisić, B.: Sirius: a rapid development of DSM graphical editor. In: IEEE 18th International Conference on Intelligent Engineering Systems INES 2014, pp. 233–238 (2014)
33. Woodcock, J., Cavalcanti, A., Foster, S., Mota, A., Ye, K.: Probabilistic semantics for RoboChart. In: Ribeiro, P., Sampaio, A. (eds.) UTP 2019. LNCS, vol. 11885, pp. 80–105. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31038-7_5
34. Woodcock, J.C.P., Davies, J.: Using Z - Specification, Refinement, and Proof. Prentice-Hall (1996)

35. Zhang, Z., Wu, Y., Liu, S., Liu, J., Zhang, M.: Provably tightest linear approximation for robustness verification of sigmoid-like neural networks. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22. Association for Computing Machinery, New York (2023)
36. Zhi, D., Wang, P., Liu, S., Luke Ong, C.-H., Zhang, M.: Unifying qualitative and quantitative safety verification of DNN-controlled systems. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification, pp. 401–426. Springer, Cham (2024)