ANN Circus Rules

Ziggy Attala

March 2024

1 Z Specification of ANNControllers

section *spec* **parents** *standard_toolkit*

We just assume we have a context, then we can run the allEvents metalanguage function on this context.

$[Context]$
$ActivationFunction ::= RELU \mid LINEAR \mid NOTSPECIFIED$
$Value == \mathbb{A}$
$SeqExp ::= null\_seq \mid list\langle\langle \mathrm{seq}\, \mathbb{N} \rangle\rangle \mid matrix\langle\langle \mathrm{seq}\, \mathrm{seq}\ Value \rangle\rangle \mid tensor\langle\langle \mathrm{seq}\, \mathrm{seq}\, \mathrm{seq}\ Value \rangle\rangle$

ANNParameters

$layerstructure : SeqExp$
$weights : SeqExp$
$biases : SeqExp$
$activationfunction : ActivationFunction$
$inputContext : Context$
$outputContext : Context$

ANN

$annparameters : ANNParameters$

ANNController

ANN

2 Semantic Rules

NOTES:

- We use *Value* as a meta-language type, we define it in our Z Specification, but we also use it in our Circus program, they are different, in Circus, it is specialised to the real number type. We keep this, and not just $\mathbb{R}$, to support BNNs, or other types of ANNs, or other precisions of ANNs.
- $\{\!|$ and $|\!\}$ brackets, used in rule 16, for example, in the metalanguage, for us, it is in the target language, but how do we express, the set in between, it is all the elements, in the set in between, technically, one at a time. Syntax inspired by connevts from rule 14, in target language is brackets, and inside is the set, that returns a set of events.
- Previous rules used: Rule 4, defined on page 36, of the robochart reference manual, $allEvents(c : Context) : Set(Event)$.
- in TRule environment, latex, cannot make new lines, in second argument of environment, goes off edge in Rule 6, chansplit, metafunction declaration. Will fix, just noting.
- Just using c.name for name of process, what should it be, fully qualified name, to link to other semantic components, or will the process be renamed in circus?
- The only semantic rule we will have to implement, that is not here, in epsilon, is *allEvents*, Rule 4.
- As the RC semantics, we assume the existence of $eventId(e : Event)$, unique identifiers for the name of the event.
- for all, needs to be in order, it is a set of events, needs to be ordered. the implementation is ordered, we need to have an order, defined by the user, replicated according to an ordering function. That can be implemented, assume existence of *order* function, implemented by the implemetnation as a list.
- going to assume the existence of an $order(s : \mathbb{P}\, Event) : \text{seq}\, Event$ function, that returns a sequence of events, of the same size of the set of events, can be implemented by lists, in Eclipse, by ordered, containement lists.

3 Semantics (with separate channels)

This is the semantics with a separate channel for each communication, as fits with Circus without partial channel instnatiation.

Also, in this semantics, we extract the values of the weights and biases individually, not using a function.

$[Context]$
$ActivationFunction ::= RELU \mid LINEAR \mid NOTSPECIFIED$
$Value == \mathbb{A}$
$SeqExp ::= null\_seq \mid list\langle\langle \text{seq}\,\mathbb{N}\rangle\rangle \mid matrix\langle\langle \text{seq seq}\ Value\rangle\rangle \mid tensor\langle\langle \text{seq seq seq}\ Value\rangle\rangle$

ANNParameters

$layerstructure : SeqExp$
$weights : SeqExp$
$biases : SeqExp$
$activationfunction : ActivationFunction$
$inputContext : Context$
$outputContext : Context$

ANN

$annparameters : ANNParameters$

ANNController

ANN

Figure 1: RoboChart types for ANN translations, specified in Z.

We specify our translation rules as functions from RoboChart types to *Circus* types. We use the *Circus* types as presented in Appendix A of Oliveira's thesis [**?**] in BNF form. We require a minimal subset of types of RoboChart to specify our translation rules, and we specify these in Z, displayed in Figure 2. These are to establish a clear basis for the translation rules in our meta-language and to define some helpful shorthand to simplify the description of these rules.

First, declare the type *Context* to represent the RoboChart Context class; we do not require any information from this type apart from its declaration for our translation rules. We also declare this type to link to the *allEvents* functions defined in the RoboChart reference manual [**?**]. Next, we define ActivationFunction using a simple free type in Z with three constants. We define *Value* as a type synonym for the type of values communicated by our abstract ANNs, known as the arithmos type. We consider the real numbers functionally, but we define a general form of ANN, which can operate on any arithmetic type. We

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

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

Rule 2. Function ANNChannelDecl
$\mathsf{ANNChannelDecl(c : ANNController) : Program} =$

$\mathsf{ANNChannels(c, layerNo(c), lastLayerS)}$
channel *terminate*
channelset *ANNHiddenEvts* == {| hiddenEvts |}
where
$\mathsf{hiddenEvts} = \{\mathsf{l}, \mathsf{n} : \mathbb{N} \mid (0 < \mathsf{l} < \mathsf{layerNo}) \land \mathsf{n} \in 1 \,..\, \mathsf{layerSize(l)} \bullet \mathsf{layerRes(l, n)}\}$
$\mathsf{lastLayerS} = \mathsf{last}\,((\mathsf{list}^{\sim})\mathsf{c.annparameters.layerstructure})$

define *SeqExp*, a type in RoboChart representing a sequence expression, using three constructors: *list*, *matrix*, and *tensor*. For convenience, we define *list* as a sequence of natural numbers, as we only require natural number sequences for our ANN components. We use *matrix* and *tensor* as shorthand for double or triple nested sequences, as we capture matrices and (3D) tensors using nested sequences. Defining *SeqExp* this way enables a more convenient representation of our translation rules.

We capture the types of our ANN classes using simple Z schemas, which we use to enable syntax matching with our EMF models defining the RoboChart metamodel. First, *ANNParameters* is a schema with a declared variable for each element of ANNParameters. Finally, we define *ANN* and *ANNController* as schemas to support syntax close to the RoboChart metamodel.

We begin with Rule 22, our top-level rule. This rule uses the semantic brackets, takes any *ANNController*, and returns semantics that may be any number of *Circus* paragraphs, such as channel declarations, channel set declarations, and process declarations. We represent this using the *Program* type, the top-level element of the *Circus* AST. This rule calls two further meta-language rules: *ANNChannelDecl* and *ANNProc*.

Rule 23 is the top-level rule for defining the channels required by our semantics. We first call the meta-language rule *ANNChannels*, which represents the variable channels: the channels that are specific to each ANN component. We declare one channel, *terminate*, that can be seen as a constant channel in our meta-language function and represents termination. An ANN component can never choose to engage in this event itself, but it needs to be able to engage in it to synchronise with other RoboChart components. We also declare a channel set *ANNHiddenEvts*, representing all the channels that should be hidden

Rule 3. Function ANNChannels
ANNChannels(c : ANNController, l, n : $\mathbb{N}$) : Program =

```
if(l == 0) ∧ (n == 1)
then
        channel layerRes(l, n)
else
        channel layerRes(1, n)
        if(l ≠ 0)
                NodeOutChannels(l, n, LStructure(c, (l − 1)))
        if(n > 1)
                ANNChannels(l, (n − 1))
        else
                ANNChannels((l − 1), LStructure(c, (l − 1))
```

in the complete behaviour of our semantics. It is defined by a variable in our meta-language: *hiddenEvts*.

We define the variable *hiddenEvts* using a set comprehension expression. This variable defines all channels associated with the hidden layers that should be hidden in the overall behaviour of the process. We define this variable using a set comprehension expression, using local variables l representing the layer and n representing the node index. We limit the values l and n, which can be taken using the predicate part of this set comprehension expression, and the expression we form defines our channel using the *layerRes* meta-language rule, which returns a channel declaration. Using this, we obtain all the channels that should be hidden in our semantics.

We define the variable *lastLayerS*, the size of the last layer, using the inverse of the *list* constructor of our free type for sequence expressions in Figure 2. We use this to obtain a sequence of natural numbers, and we then get the last element using the *last* function, itself defined in the Z mathematical toolkit.

Rule 3 defines the function that represents all the variable channels in our semantics as a recursive meta-language function. This function has three parameters: c, an *ANNController*, l, representing the layer of the ANN component we are considering, and n, the node index of the ANN component. This function computes all channels by proceeding backwards from the output layer and the last node back to the first node in the input layer.

The base case is when l is 0, and n is 1 because we use 0 to represent the input layer, which has defined channels but no behaviour. We capture the input layer in a similar way to an interface. We use one-based indexed for the nodes, as we do not have the concept of an input node. In the base case, we return just the channel *layerRes*.0.1, as this is the channel associated with the first node of the input layer.

In the recursive case, we first define the channel representing the layer output of the current layer and node, $layerRes(l, n)$. Then, if we do not consider

Rule 4. Function NodeOutChannels
NodeOutChannels(l, n, i : $\mathbb{N}$) : Program =

```
if(i == 1)
then
    channel nodeOut(l, n, i)
else
    channel nodeOut(1, n, i)
    NodeOutChannels(l, n, (i − 1))
```

Rule 5. Function ANNProc
ANNProc(c : ANNController) : ProcDecl =

```
process c.name ≙
    begin
    Collator ≙ l, n, i : ℕ; sum : Value • Collator(c)
    NodeIn ≙ l, n, i : ℕ • NodeIn(c)
    Node ≙ l, n, inpSize : ℕ • Node(c)
    HiddenLayer ≙ l, s, inpSize : ℕ • HiddenLayer(c)
    HiddenLayers ≙ HiddenLayers(c, layerNo(c) − 1)
    OutputLayer ≙ OutputLayer(c)
    ANN ≙ ((HiddenLayers⟦ | IndexedLayerRes(layerNo(c) − 1) | ⟧OutputLayer)
          \ ANNHiddenEvts) ; ANN
    ANNRenamed ≙ ANNRenamed(c)
    • ANNRenamed
    end
```

the channels of an input layer, that is, if $l \neq 0$, then we need to consider the channels of the node itself, which we define through the meta-language function *NodeOutChannels*, which operates in a similar way to this function. The recursive call changes if we call the previous node in this layer, $n > 1$, or if we call the previous layer when $n = 0$.

Rule 4 functions in a very similar way to Rule 3. We need these channels to enable intra-node communication between the processes representing the input to this node and the collator of this node. Here, i represents the input index, the size of the previous layer's output, used to transmit the results from the last layer to the next layer.

We now consider the rules defining our ANN model's process paragraph. We start with the top-level process rule *ANNProc* in Rule 24. In a paragraph, we define multiple actions, similar to CSP processes, then a main action after the syntax •, which represents the behaviour of this process. We name our

Rule 6. Function Collator
Collator(c : ANNController) : CSPAction =

$$\Box \,\mathsf{l} : 1 \,..\, \mathsf{layerNo(c)};\ \mathsf{n} : 1 \,..\, \mathsf{LStructure(c, l)};\ \mathsf{i} : 0 \,..\, \mathsf{LStructure(c, (l-1))}$$
$$\bullet\big(l = \mathsf{l} \wedge n = \underline{\mathsf{n}} \wedge i = \underline{\mathsf{i}}\big)\,\&$$
$$\underline{\textbf{if}\,(\mathbf{i == 0})}$$
$$\underline{\textbf{then}}$$
$$\underline{\mathsf{layerRes(l, n)}}!relu(sum + (\underline{\mathsf{bias(c, l, n)}})) \longrightarrow \textbf{Skip}$$
$$\underline{\textbf{else}}$$
$$\underline{\mathsf{nodeOut(l, n, i)}}?x \longrightarrow Collator(l, n, (i-1), (sum + x))$$

process using the *name* attribute of our *ANNController*. We have access to this attribute as an ANNController is a NamedElement in RoboChart. In our first four actions, *Collator*, *NodeIn*, *Node*, and *HiddenLayer*, we have fixed local variables represented by non-underlined syntax in our meta-language, followed by a meta-language function representing the ANN-specific definition of this action.

For the last four definitions, we do not use local variable definitions as the behaviour of these actions represents the overall structure of an ANN pattern; their behaviour is less influenced by the hyperparameters of an ANN model. *HiddenLayers* and *OutputLayer* are each defined by a meta-language function capturing the composition of the hidden layers and the behaviour of just the output layer in isolation. The action *ANN* defines an ANN component's main behaviour: the repeating parallel composition of *HiddenLayers* and *OutputLayer*, with *ANNHiddenEvts* hidden. We capture the synchronisation set of this parallel composition using the $IndexedLayerRes(layerNo(c)-1)$ meta-function, which returns all channels in the penultimate layer, where $layerNo(c)$ is the number of layers in the ANN model. We use the function *ANNRenamed* to capture the contextual renaming of our ANN component to events of the RoboChart model. Finally, our main action is defined as *ANNRenamed*, which captures the behaviour of our ANN semantics.

Rule 6 defines the rule for the behaviour of the *Collator* action, represented using the *CSPAction* AST element. This rule defines, at its core, a recursive action that sums all values communicated by the channel *nodeOut* and then communicates these results through the $layerRes(l, n)$ channel as its base case, with the *bias* from the *bias* meta-function. In the target language, this recursive action is also called *Collator* and is shown here as a constant element in the recursive case in this rule. The constructs surrounding this behaviour are a distributed external choice over l, the layer size, n, the node size, and i, the input size, the size of the previous layer. Each is guarded by checking that the target language variable is equal to the meta-language variable, meaning only one behaviour is possible at every parameter instantiation. This structure is

Rule 7. Function NodeIn
NodeIn(c : ANNController) : CSPAction =

$$\Box \mathsf{l} : 1 \,..\, \mathsf{layerNo(c)};\ \mathsf{n} : 1 \,..\, \mathsf{LStructure(c, l)};\ \mathsf{i} : 1 \,..\, \mathsf{LStructure(c, (l-1))}$$
$$\bullet\big(l = \mathsf{l} \wedge n = \mathsf{n} \wedge i = \mathsf{i}\big) \&$$
$$\mathsf{layerRes(l-1, i)}?x \longrightarrow \mathsf{nodeOut(l, n, i)}!(x * \mathsf{weight(c, l, n, i)}) \longrightarrow \mathbf{Skip}$$

Rule 8. Function Node
Node(c : ANNController) : CSPAction =

$$\Box \mathsf{l} : 1 \,..\, \mathsf{layerNo(c)};\ \mathsf{n} : 1 \,..\, \mathsf{LStructure(c, l)}$$
$$\bullet\big(l = \mathsf{l} \wedge n = \mathsf{n}\big) \ \&\ \ ((\Big|\Big|\Big| i : 1 \,..\, inpSize \bullet NodeIn(l, n, i))$$
$$[\![\mid \{\!| \mathsf{IndexedNodeOut(l, n)} |\!\} \mid]\!]$$
$$Collator(l, n, inpSize, 0) \setminus \{\!| \mathsf{IndexedNodeOut(l, n)} |\!\})$$

only needed to support the creation of distinct channels for every layer, node, and input size, as is required in *Circus*.

Rule 7 defines the behaviour of the *NodeIn* action: to receive the value communicated through the channel $layerRes(l-1, i)$, then to transmit a modified version of this using the channel $nodeOut(l, n, i)$. It is a buffer that applies the appropriate weight to the value communicated by the previous layer's event. It is a modifying buffer. In our *Circus* model of an ANN's behaviour, each *NodeIn* action is used to transmit the result from the previous layer onto the current layer. We use the *nodeOut* channel as an intermediate communication channel, on which every *Collator* process synchronises to obtain the weighted output of the previous layer. We use a surrounding structure identical to that of Rule 6 to support distinct channel naming.

Similar to Rule 7 and Rule 6, Rule 8 uses the external choice and guarding considering the l and n indices to support the distinct channel naming. The parallel composition of two actions defines this rule. First, a distributed interleaving of all *NodeIn* processes indexed by the layer, node, and considering every input from *inpSize*. Here, *inpSize* is one of the parameters for *Node* in the target language; see Rule 24. This entire action is defined in the target language, as its definition does not change based on the parameters of *Node*. The second action is the *Collator* action to receive all communications from the *NodeIn* interleavings and transmit the result on the *layerRes* channel. We use $IndexedNodeOut(l, n)$ to define the channels we synchronise on. This meta-function returns all *nodeOut* channels associated with this node: $nodeOut.l.n.i$

Rule 9. Function HiddenLayer
HiddenLayer(c : ANNController) : CSPAction =

$$\Box \, \mathsf{l} : 1 \,..\, \mathsf{layerNo(c)}$$
$$\bullet \big(l = \mathsf{l}\big) \;\&\; ([\![\{\!| \, \mathsf{IndexedLayerRes(l-1)} \, |\!\} \,]\!] \; i : 1 \,..\, s \bullet \mathit{Node}(l, i, \mathit{inpSize}))$$

Rule 10. Function HiddenLayers
HiddenLayers(c : ANNController, l : $\mathbb{N}$) : CSPAction =

if(l == 1)
then
$(\mathit{HiddenLayer}(\mathsf{l}, \mathsf{LStructure(l)}, \mathsf{LStructure(l-1)}))$
else
$(\mathsf{HiddenLayers(c, (l-1))}$
$[\![\mid \{\!| \, \mathsf{IndexedLayerRes(c, l-1)} \, |\!\} \mid]\!]$
$\mathit{HiddenLayer}(\mathsf{l}, \mathsf{LStructure(l)}, \mathsf{LStructure(l-1)})$

where i is the size of the previous layer. We use the channel set operator, denoted using $\{\!|\ |\!\}$, of these channels to define all events that can be communicated using this set of channels. Finally, we hide this synchronisation set from the resultant process, so the internal communication channel *nodeOut* is not visible in the *Node* action.

Rule 9 defines the behaviour of our *HiddenLayer* action. We define this using a replicated parallel, synchronised on the channel set of all *layerRes* channels associated with this layer, captured by the meta-function $\mathit{IndexedLayerRes}(l-1)$. A layer in an ANN model synchronises with the previous layer's result $(l-1)$ because these events are shared across every node in this layer. Each node process engages with a single event indexed by $\mathit{layerRes}.l$ to communicate its output; these communications do not need to be synchronised as each node has a unique output channel. Using this rule, Rule 25 defines the behaviour of an arbitrary number of hidden numbers as a recursive composition of parallel actions.

Rule 26 defines the output layer for our ANN model. This layer can be defined as a layer whose parameters are constant, using the meta-function $\mathit{layerNo}(c)$ as a constant to define the shape of our output layer. We separate the hidden and output layers of an ANN model to capture the standard pattern used when designing an ANN model. There is a fixed shape to the hidden layers, for example, 6 layers with 50 nodes in each [**?**], and the shape of both the input and output layers are distinct from this shape, for example, 5 and 5. Moreover, the shape of the input and output layers are defined by the context

Rule 11. Function OutputLayer
OutputLayer(c : ANNController) : CSPAction =

$$[\![\{\!| \text{IndexedLayerRes}(\text{layerNo}(c) - 1) |\!\}]\!] \; i : 1 \,..\, \text{LStructure}(\text{layerNo}(c)) \bullet Node(l, i, \text{LStructure}(\text{layerNo}(c) - 1)))$$

Rule 12. Function ANNRenamed
ANNRenamed(c : ANNController) : CSPAction) =

$$(ANN) \;\; [\text{orderedLayerRes} := \text{eventList}] \triangle terminate \longrightarrow \textbf{Skip}$$

where

$$\text{orderedLayerRes} = \text{order}(\{l : \{0, \text{layerNo}(c)\};\; n : 1 \,..\, \text{LStructure}(c, l) \bullet \text{layerRes}(l, n)\})$$

$$\text{eventList} = \text{order}(\text{allEvents}(c.\text{annparameters.inputContext}))^\frown \text{order}(\text{allEvents}(c.\text{annparameters.outputContext}))$$

where an ANN model is used, but the shape of the hidden layers is defined by the complexity of the desired relation, which is independent of context and is a training consideration. Finally, output layers often have probabilistic interpretations, and their results are subject to additional external analysis, which we can support easier with a separate defined output layer action in *Circus*.

We present the last of our functional rules that define the process presented in Rule 24, in Rule 27. This rule defines the target language action ANN, see Rule 24, where a renaming is applied, is interrupted ($\triangle$) by the action $terminate \longrightarrow \textbf{Skip}$. This is to support if the other components in the system terminate, represented by the event *terminate*, then the ANN component should terminate and behave as **Skip**. The renaming operation is to rename the *layerRes* channels to the contextual channels defined by the user in the *inputContext* and *outputContext* in RoboChart. We define that all events in *orderedLayerRes* are renamed to the events from *eventList*, the contextual events. Here, we define *orderedLayerRes* by a set comprehension expression $layerRes(l, n)$, where l is either 0 or *layerNo* (input or output layer), and n is all nodes in these layers. We define *eventList* as all events, using the *allEvents* meta-function [**?**], of the input context concatenated with the events of the output context. The *order* function is then applied to both of these sets, here we assume this function takes a set and creates a sequence ordering this set, we can implement this function in EMF as the references of an object is defined using lists. In other words, our implementation is based on the order in which the events appear in the textual language for RoboChart, so in our current implementation, the ordering of events is an important consideration for ANN components.

The explanation of the rules that define the functional behaviour of the

Rule 13. Function LStructure
LStructure(c : ANNController, i : $\mathbb{N}$) : $\mathbb{N}$ =

if(i == **0**)
then
    # allEvents(c.annparameters.inputContext)
else
    ((list$^\sim$)c.annparameters.layerstructure) i

Rule 14. layerRes Channel
layerRes(l : $\mathbb{N}$, n : $\mathbb{N}$) : CSExp =

*layerRes*ln : *Value*

semantics for our ANN components is now complete. Rules 29 through to our final rule 36 present helper functions for values, channels, and channel sets used to define the behaviour of our rules. These rules also enable a cleaner syntax for presenting the functional rules.

Rule 28 and Rule 32 define constants about our ANN model. We define *LStructure* as the size of each layer, using the inverse of the *list* free type constructor indexed by i and the number of events in the input context if we refer to the input layer ($l = 0$). The function *layerNo* defines the number of layers in our model, simply the size of the *layerstructure* sequence in our *ANNController*.

Next, we discuss our helper rules that refer to channels, a *CSExp* in *Circus*. Our basic channels are defined in Rule 29 and Rule 31, defining *layerRes* with l and n, and *nodeOut* indexed with l, n, and i. We define the indexed versions of these channels in Rules 30 and 36. The indexed versions take all but the last parameter and return the channel set containing all channels constructed using the l parameter for *layerRes* and the l and n parameters for *nodeOut*. Finally, for the *nodeOut* channel only, we define *AllNodeOut* in Rule 35. This function is a shorthand for the channel set of *nodeOut* channels used by the *ANNController* c. To describe this, we define the number of layers, for l, by *layerNo*, the size of each layer, for n, by $LStructure(l)$, and the size of the previous layer, for the index i, set by $LStructure(l-1)$.

Finally, we define our trained parameters, the weights and biases, in Rule 33 and Rule 34. The definition of these uses the inverse of the *tensor* and *matrix*

Rule 15. Function IndexedLayerRes
IndexedLayerRes(l : $\mathbb{N}$) : CSExp =

{| {n : 1 .. LStructure(l) • layerRes(l, n)} |}

Rule 16. nodeOut Channel
nodeOut(l : $\mathbb{N}$, n : $\mathbb{N}$, i : $\mathbb{N}$) : CSExp =

$nodeOut$lni : $Value$

Rule 17. Function layerNo (number of layers)
layerNo(c : ANNController) : $\mathbb{N}$ =

#((list ~) c.annparameters.layerstructure)

Rule 18. Function weight
weight(c : ANNController; l, n, i : $\mathbb{N}$) : Value =

((tensor ~) c.annparameters.weights) l n i

Rule 19. Function bias
bias(c : ANNController; l, n : $\mathbb{N}$) : Value =

((matrix ~) c.annparameters.biases) l n

Rule 20. Function AllNodeOut
AllNodeOut(c : ANNController) : CSExp =

{| {l : 1 .. layerNo(c); n : 1 .. LStructure(c, l); i : 1 .. LStructure(c, (l − 1)) • nodeOut(l, n, i)} |}

Rule 21. Function IndexedNodeOut
IndexedNodeOut(c : ANNController, l, n : $\mathbb{N}$) : CSExp =

{| {i : 1 .. LStructure(c, l − 1) • nodeOut(l, n, i)} |}

constructors, as discussed in our Z assumptions, and each is indexed by l, n, and i for the weight values.

This concludes our description of our semantic rules, that defines any ANN component in RoboChart in *Circus*, next, we conclude this section with final considerations.

4 Semantic Rules (with parameterised channels)

This is the semantics with parameterised channels and partial channel instnatiatons. Also, we define weights and biases as a function in the target language, that has a trivial definition based on the sequences from RoboChart.

note: The text is not updated for these semantic rules.

This section presents our denotational semantic rules using a meta-language for ANN components in RoboChart. We use semantic brackets to denote that each ANNController is given a precise meaning in *Circus*, meaning these semantics are compositional. This matches with the RoboStar style of verification and proof.

We define our ANN components as networks of processes. This means that we do not need state in *Circus*, but our semantics in *Circus* and not CSP broadens what we can prove about ANN components for several reasons. First, we want to create an integrated model for proofs of the complete software system in RoboChart, and we are defining the semantics of RoboChart in *Circus*. Secondly, in our approach, we wish to soundly replace a controller with an ANN component, specifying the controller in *Circus* allows the controller to have stateful behaviour, which we can use to generate verification conditions on our ANN component in our approach to proof. Section **??** discusses more details of our verification approach. Finally, due to the encoding of *Circus* and CSP. We encode our *Circus* models in Isabelle; we do not have an automated encoding scheme or strategy for CSP processes in EMF. The UTP reactive contract theory can capture the semantics of CSP, but we do not have an automated encoding tool; it is only automated encoding in CSPM, which only permits discrete models.

We enable two forms of automated reasoning with these semantics: first, an automated translation to Isabelle, which enables us to prove theorems about the complete behaviour of an ANN component using a predicate model; second, reasoning in CSPM enabled in RoboTool using cyclic controller semantics to verify structural properties of RoboChart models. These simplified CSP semantics are a refinement of this cyclic process pattern, so we can soundly use these semantics to establish general but limited properties about the behaviour of ANN components.

We specify our translation rules as a set of functions from RoboChart types to *Circus* types. We use the *Circus* types as presented in Appendix A of Oliveira's thesis [**?**] in BNF form. We require a very limited subset of types of RoboChart to specify our translation rules, and we specify these in Z, displayed in Figure 2. These are to establish a clear basis for the translation rules in our meta-language, and to define some useful shorthand to simplify the description of

$[Context]$
$ActivationFunction ::= RELU \mid LINEAR \mid NOTSPECIFIED$
$Value == \mathbb{A}$
$SeqExp ::= null\_seq \mid list\langle\langle \mathrm{seq}\, \mathbb{N} \rangle\rangle \mid matrix\langle\langle \mathrm{seq}\, \mathrm{seq}\, Value \rangle\rangle \mid tensor\langle\langle \mathrm{seq}\, \mathrm{seq}\, \mathrm{seq}\, Value \rangle\rangle$

ANNParameters

$layerstructure : SeqExp$
$weights : SeqExp$
$biases : SeqExp$
$activationfunction : ActivationFunction$
$inputContext : Context$
$outputContext : Context$

ANN

$annparameters : ANNParameters$

ANNController

ANN

Figure 2: RoboChart types for ANN translations, specified in Z.

these rules.

First, declare the type *Context* to represent the RoboChart Context class, we do not require any information from this type apart from its declaration for our translation rules. We also declare this type to link to the *allEvents* functions, as defined in the RoboChart reference manual [**?**]. Next, we define ActivationFunction using a simple free type in Z, with three constants. We define *Value* as a type synonym for the type of values communicated by our abstract ANNs, known as the arithmos type. We consider the real numbers functionally, but we define a general form of ANN, which can operate on any arithmetic type. We define *SeqExp*, a type in RoboChart representing a sequence expression, using three constructors: *list*, *matrix*, and *tensor*. We define *list* as a sequence of natural numbers for convience, as we only require natural number sequences for our ANN components. We use *matrix* and *tensor* as shorthand for double or triple nested sequences, as we capture matrices and (3D) tensors using nested

Rule 22. Semantics of ANNs
$[\![\mathsf{c} : \mathsf{ANNController}]\!]_{\mathcal{ANN}} : \mathsf{Program} =$

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

Rule 23. Function ANNChannelDecl
$\mathsf{ANNChannelDecl(c : ANNController) : Program} =$

channel $layerRes : \mathbb{N} \times \mathbb{N} \times Value$
channel $nodeOut : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times Value$
channel $terminate$
channelset $ANNHiddenEvts == \{\!|\, \mathsf{hiddenEvts} \,|\!\}$
where
$\mathsf{hiddenEvts} = \{\mathsf{l}, \mathsf{n} : \mathbb{N} \mid (0 < \mathsf{l} < \mathsf{layerNo}) \land \mathsf{n} \in 1 \,..\, \mathsf{layerSize(l)} \bullet \mathsf{layerRes(l, n)}\}$
$\mathsf{lastLayerS} = \mathsf{last}\,((\mathsf{list}^{\sim})\mathsf{c.annparameters.layerstructure})$

sequences. Defining *SeqExp* this way enables a more convenient representation of our translation rules.

We capture the types of our ANN classes using simple Z schemas, we use this to enable syntax matching with our EMF models defining the RoboChart metamodel. First, *ANNParameters* is a schema with a declared variable for each element of ANNParameters. Finally, we define *ANN* and *ANNController* as schemas to support syntax close to the RoboChart metamodel.

We begin with Rule 22, our top-level rule. This rule uses the semantic brackets and takes any *ANNController*, and returns semantics that may be any number of *Circus* paragraphs, such as channel declarations, channel set declarations, and process declarations. We represent this using the *Program* type, which is the top-level element of the *Circus* AST. Here, this rule calls two further meta-language rules: *ANNChannelDecl* and *ANNProc*.

Rule 23 is the top-level rule for defining the channels required by our semantics. We first call the meta-language rule *ANNChannels*, which represents the variable channels, that is, the channels that are specific to each ANN component. We declare one channel, *terminate*, that can be seen as a constant channel in our meta-language function, and represents termination. An ANN component can never choose to engage on this event itself, but it needs to be able to engage on it to synchronise with other RoboChart components. We also declare a channel set *ANNHiddenEvts*, representing all the channels that should be hidden in the complete behaviour of our semantics. It is defined by a variable in our meta-language: *hiddenEvts*.

We define the variable *hiddenEvts* using a set comprehension expression. This variable defines all channels associated with the hidden layers, that should be hidden in the overall behaviour of the process. We define this variable using a

set comprehension expression, using local variables l representing the layer and n representing the node index. We limit the values l and n can take using the predicate part of this set comprehension expression, and the expression we form is defining our channel using the *layerRes* meta-language rule, which returns a channel declaration. Using this, we obtain all the channels that should be hidden in our semantics.

We define the variable *lastLayerS*, the size of the last layer, using the inverse of the *list* constructor of our free type for sequence expressions, in Figure 2. We use this to obtain a sequence of natural numbers, that we then obtain the last element of using the *last* function, itself defined in the Z mathematical toolkit.

Rule 3 defines the function that represents all the variable channels in our semantics as a recursive meta-language function. This function has three parameters: c, an *ANNController*, l, representing the layer of ANN component we are considering, and n, the node index of the ANN component. This function computes all channels by proceeding backwards from the output layer and the last node, back to the first node in the input layer.

The base case is when l is 0 and n is 1, this is because we use 0 to represent the input layer, which has defined channels but no behaviour. We capture the input layer in a similar way to an interface. We use one-based indexed for the nodes, as we do not have the concept of an input node. In the base case, we return just the channel *layerRes*.0.1, as this is the channel associated with the first node of the input layer.

In the recursive case, we first define the channel representing the layer output of the current layer and node, $layerRes(l, n)$. Then, if we not considering the channels of an input layer, that is if $l \neq 0$, then we need to consider the channels of the node itself, which we define through the meta-language function *NodeOutChannels*, which operates in a similar way to this function. The recursive call changes if we are calling the previous node in this layer, $n > 1$, or if we call the previous layer, when $n = 0$.

Rule 4 functions in a very similar way to Rule 3. We need these channels to enable intra-node communication between the processes representing the input to this node, and the collator of this node. Here, i represents the input index, that is, the size of the previous layers output, used to transmit the results from the previous layer to the next layer.

We now consider the rules which define the process paragraph for our ANN model. We start with the top-level process rule *ANNProc* in Rule 24. In a paragraph, we define multiple actions, similar to CSP processes, then a main action after the syntax $\bullet$ which represents the behaviour of this process. We name our process using the *name* attribute of our *ANNController*. As an ANNController is a NamedElement in RoboChart, we have access to this attribute. In our first four actions, *Collator*, *NodeIn*, *Node*, and *HiddenLayer*, we have fixed local variables, represented by non-underlined syntax in our meta-language, followed by a meta-language function representing the ANN-specific definition of this action.

For the last four definitions, we do not use local variable definitions as the behaviour of these actions represents the overall structure of an ANN pattern,

Rule 24. Function ANNProc
ANNProc(c : ANNController) : ProcDecl =

$$
\begin{array}{l}
\textbf{process}\ \underline{\mathsf{c.name}} \mathrel{\widehat{=}} \\
\quad \textbf{begin} \\
\quad Collator \mathrel{\widehat{=}} l, n, i : \mathbb{N};\ sum : Value \bullet \\
\qquad \big(i == 0\big)\ \&\ layerRes.l.n!relu(sum + bias(l,n)) \longrightarrow \textbf{Skip}\ \Box \\
\qquad \big(i > 0\big)\ \&\ nodeOut.l.n.i?x \longrightarrow Collator(l, n, (i-1), (sum + x)) \\
\quad NodeIn \mathrel{\widehat{=}} l, n, i : \mathbb{N} \bullet \\
\qquad layerRes.(l-1).i?x \longrightarrow nodeOut.l.n.i!(x * weight(l,n,i)) \longrightarrow \textbf{Skip} \\
\quad Node \mathrel{\widehat{=}} l, n, inpSize : \mathbb{N} \bullet \\
\qquad ((\Big|\Big|\Big|\ i : 1\,..\,inpSize \bullet NodeIn(l,n,i)) \\
\qquad [\![\ |\ \{\!|\ nodeOut.l.n\ |\!\}\ |\ ]\!] \\
\qquad Collator(l, n, inpSize, 0) \setminus \{\!|\ nodeOut.l.n\ |\!\}) \\
\quad HiddenLayer \mathrel{\widehat{=}} l, s, inpSize : \mathbb{N} \bullet \\
\qquad ([\![\{\!|\ layerRes.(l-1)\ |\!\}\ ]\!]\ i : 1\,..\,s \bullet Node(l, i, inpSize)) \\
\quad HiddenLayers \mathrel{\widehat{=}} \underline{\mathsf{HiddenLayers(c, layerNo(c) - 1)}} \\
\quad OutputLayer \mathrel{\widehat{=}} \\
\qquad [\![\{\!|\ \underline{\mathsf{IndexedLayerRes(layerNo(c) - 1)}}\ |\!\}\ ]\!]\ i : 1\,..\,\underline{\mathsf{LStructure(layerNo(c))}} \bullet \\
\qquad\quad Node(l, i, \mathsf{LStructure(layerNo(c) - 1)})) \\
\quad ANN \mathrel{\widehat{=}}\ ((HiddenLayers[\![\ |\ \underline{\mathsf{IndexedLayerRes(layerNo(c) - 1)}}\ |\ ]\!] OutputLayer) \\
\qquad\quad \setminus ANNHiddenEvts)\ ;\ ANN \\
\quad ANNRenamed \mathrel{\widehat{=}} \underline{\mathsf{ANNRenamed(c)}} \\
\quad \bullet\ ANNRenamed \\
\quad \textbf{end}
\end{array}
$$

their behaviour is less influenced by the hyperparameters of an ANN model. *HiddenLayers* and *OutputLayer* are each defined by a meta-language function capturing the composition of the hidden layers and the behaviour of just the output layer in isolation. The action *ANN* defines the main behaviour of an ANN component: it is the repeating parallel composition of *HiddenLayers* and *OutputLayer*, with *ANNHiddenEvts* hidden. We capture the synchronisation set of this parallel composition using the $\mathit{IndexedLayerRes}(\mathit{layerNo}(c) - 1)$ meta-function, that returns all channels in the penultimate layer, where $\mathit{layerNo}(c)$ is the number of layers in the ANN model. We use the function *ANNRenamed* to capture the contextual renaming of our ANN component to events of the RoboChart model. Finally, our main action is defined as *ANNRenamed*, which captures the behaviour of our ANN semantics.

Rule 6 defines the rule for the behaviour of the *Collator* action, represented using the *CSPAction* AST element. This rule defines, at its core, a recursive action that sums all values communicated by the channel *nodeOut*, and then communicates these results through the $\mathit{layerRes}(l, n)$ channel as its base case, with the *bias* from the *bias* meta-function. This recursive action, in the target language, is also called *Collator*, and is shown here as a constant element in the recursive case in this rule. The constructs surrounding this behaviour is a distributed external choice over l, the layer size, n, the node size, and i, the input size, the size of the previous layer. Each is guarded by checking that the target language variable is equal to the meta-language variable, meaning only one behaviour is possible at every parameter instantiation. All of this structure is only needed to support the creation of distinct channels for every layer, node, and input size, as is required in *Circus*.

Rule 7 defines the behaviour of the *NodeIn* action: to receive the value communicated through the channel $\mathit{layerRes}(l - 1, i)$, then to transmit a modified version of this using the channel $\mathit{nodeOut}(l, n, i)$. It is a buffer which applies the appropriate weight to the value communicated by the previous layer's event, it is a modifying buffer. In our *Circus* model of an ANN's behaviour, each *NodeIn* action is used to transmit the result from the previous layer onto the current layer. We use the *nodeOut* channel as an intermediate communication channel, that every *Collator* process synchronises on to obtain the weighted output of the previous layer. We use a surrounding structure identical to that of Rule 6, to support distinct channel naming.

Similar to Rule 7 and Rule 6, Rule 8 uses the external choice and guarding considering the l and n indices to support the distinct channel naming. This rule is defined by the parallel composition of two actions. First, a distributed interleaving of all *NodeIn* processes indexed by the layer, node, and considering every input, from *inpSize*. Here, *inpSize* is one of the parameters for *Node* in the target language, see Rule 24. This entire action is defined in the target language, as its definition does not change based on the parameters of *Node*. The second action is the *Collator* action to receive all communications from the *NodeIn* interleavings and transmit the result on the *layerRes* channel. To define the channels we synchronise on, we use $\mathit{IndexedNodeOut}(l, n)$, a meta-function that returns all *nodeOut* channels associated with this node: $\mathit{nodeOut}.l.n.i$ where i

Rule 25. Function HiddenLayers
HiddenLayers(c : ANNController, l : $\mathbb{N}$) : CSPAction =

if(l == **1**)
then
($HiddenLayer$(l, LStructure(l), LStructure(l − 1)))
else
(HiddenLayers(c, (l − 1))
⟦ | {| IndexedLayerRes(c, l − 1) |} | ⟧
$HiddenLayer$(l, LStructure(l), LStructure(l − 1))

Rule 26. Function OutputLayer
OutputLayer(c : ANNController) : CSPAction =

⟦{| IndexedLayerRes(layerNo(c) − 1) |}⟧ i : 1 . . LStructure(layerNo(c)) ●
$Node(l, i,$ LStructure(layerNo(c) − 1)))

is the previous layers size. We use the channel set operator, denoted using {| |}, of these channels to define all events that can be communicated using this set of channels. Finally, we hide this synchronisation set from the resultant process, so the internal communication channel *nodeOut* is not visible in the *Node* action.

Rule 9 defines the behaviour of our *HiddenLayer* action. We define this using a replicated parallel, synchronised on the channel set of all *layerRes* channels associated with this layer, captured by the meta-function $IndexedLayerRes(l - 1)$. A layer in an ANN model synchronises on the previous layer's result $(l-1)$, because these events are shared across every node in this layer. Each node process engages with a single event indexed by $layerRes.l$ to communicate its output, these communications do not need to be synchronised as each node has a unique output channel. Using this rule, Rule 25 defines the behaviour of an arbitrary number of hidden numbers as a recursive composition of parallel actions.

Rule 26 defines the output layer for our ANN model. This layer can be seen as the definition of a layer whose parameters are constant, using the meta-function $layerNo(c)$ as a constant to define the shape of our output layer. We separate the hidden layers and the output layer of an ANN model to capture the common pattern used when designing an ANN model: there is a fixed shape to the hidden layers, for example 6 layers with 50 nodes in each [**?**], and the shape of both the input and output layers are distinct from this shape, for example 5 and 5. Moreover, the shape of the input and output layers are defined by the context where an ANN model is used, but the shape of the hidden layers is defined by the complexity of the desired relation, which is independent of context

Rule 27. Function ANNRenamed
ANNRenamed(c : ANNController) : CSPAction) =

$$
\begin{array}{l}
(ANN)\ \ [\mathsf{orderedLayerRes} := \mathsf{eventList}] \triangle\ terminate \longrightarrow \mathbf{Skip} \\
\mathbf{where} \\
\mathsf{orderedLayerRes} = \\
\quad \mathsf{order}(\{\mathsf{l} : \{0, \mathsf{layerNo(c)}\};\ \mathsf{n} : 1 \,..\, \mathsf{LStructure(c, l)} \bullet \mathsf{layerRes(l, n)}\}) \\
\mathsf{eventList} = \mathsf{order(allEvents(c.annparameters.inputContext))}^\frown \\
\quad \mathsf{order(allEvents(c.annparameters.outputContext))}
\end{array}
$$

Rule 28. Function LStructure
LStructure(c : ANNController, i : $\mathbb{N}$) : $\mathbb{N}$ =

$$
\begin{array}{l}
\mathbf{if}(\mathbf{i} == \mathbf{0}) \\
\mathbf{then} \\
\quad \#\,\mathsf{allEvents(c.annparameters.inputContext)} \\
\mathbf{else} \\
\quad (((\mathsf{list}^\sim)\mathsf{c.annparameters.layerstructure})\,\mathsf{i}
\end{array}
$$

and is a training consideration. Finally, output layers often have probabilistic interpretations, and their results are subject to additional external analysis, that we can support easier with a separate defined output layer action in *Circus*.

We present the last of our functional rules, that defines the process presented in Rule 24, in Rule 27. This rule defines the target language action *ANN*, see Rule 24, where a renaming is applied, is interrupted ($\triangle$) by the action $terminate \longrightarrow \mathbf{Skip}$. This is to support if the other components in the system terminate, represented by the event *terminate*, then the ANN component should terminate, behave as **Skip**, as well. The renaming operation is to rename the *layerRes* channels to the contextual channels as defined by the user in the *inputContext* and *outputContext* in RoboChart. We define that all events in *orderedLayerRes* are renamed to the events from *eventList*, the contextual events. Here, we define *orderedLayerRes* by a set comprehension expression $layerRes(l, n)$, where l is either 0 or *layerNo* (input or output layer), and n is all nodes in these layers. We define *eventList* to be all events, using the *allEvents* meta-function [**?**], of the input context concatenated with the events of the output context. The *order* function is then applied to both of these sets, here we assume this function takes a set and creates a sequence ordering this set, we can implement this function in EMF as the references of an object is defined using lists. In other words, in our implementation is based on the order that the events appear in the textual language for RoboChart, so, in our current implementation, the ordering of events is an important consideration for ANN components.

The explanation of the rules that define the functional behaviour of the semantics for our ANN components now complete. Rules 29 through to our

Rule 29. layerRes Channel
layerRes(l : ℕ, n : ℕ) : CSExp =

$$layerRes.\mathsf{l}.\mathsf{n}$$

Rule 30. Function IndexedLayerRes
IndexedLayerRes(l : ℕ) : CSExp =

$$\{\!|\, layerRes.\mathsf{l} \,|\!\}$$

final rule 36 present helper functions for values, channels, and channel sets used to define the behaviour of our rules. These rules also enable a cleaner syntax for the presentation of the functional rules.

Rule 28 and Rule 32 define constants about our ANN model. We define *LStructure* to be the size of each layer, using the inverse of the *list* free type constructor indexed by i, and the number of events in the input context if we are referring to the input layer ($l = 0$). The function *layerNo* defines the number of layers in our model, which is simply the size of the *layerstructure* sequence in our *ANNController*.

Next, we discuss our helper rules that refer to channels, a *CSExp* in *Circus*. Our basic channels are defined in Rule 29 and Rule 31, defining *layerRes* with l and n, and *nodeOut* indexed with l, n, and i. We define the indexed versions of these channels in Rules 30 and 36. The indexed versions take all but the last parameter, and return the channel set containing of all channels constructed using the l parameter for *layerRes*, and the l and n parameter for *nodeOut*. Finally, for the *nodeOut* channel only, we define *AllNodeOut* in Rule 35. This function is a shorthand for the channel set of *nodeOut* channels used by the *ANNController* c. To describe this, we define the number of layers, for l, by *layerNo*, the size of each layer, for n, by $LStructure(l)$, and the size of the previous layer, for the index i, set by $LStructure(l-1)$.

Finally, we define our trained parameters, the weights and biases, in Rule 33 and Rule 34. The definition of these using the inverse of the *tensor* and *matrix* constructors, as discussed in our Z assumptions, and each are indexed by l, n, and i for the weight values.

This concludes our description of our semantic rules, that defines any ANN

Rule 31. nodeOut Channel
nodeOut(l : ℕ, n : ℕ, i : ℕ) : CSExp =

$$nodeOut.\mathsf{l}.\mathsf{n}.\mathsf{i}$$

Rule 32. Function layerNo (number of layers)
layerNo(c : ANNController) : $\mathbb{N}$ =

#((list ~) c.annparameters.layerstructure)

Rule 33. Function weight
weight(c : ANNController; l, n, i : $\mathbb{N}$) : Value =

((tensor ~) c.annparameters.weights) l n i

Rule 34. Function bias
bias(c : ANNController; l, n : $\mathbb{N}$) : Value =

((matrix ~) c.annparameters.biases) l n

Rule 35. Function AllNodeOut
AllNodeOut(c : ANNController) : CSExp =

{| $nodeOut$ |}

Rule 36. Function IndexedNodeOut
IndexedNodeOut(c : ANNController, l, n : $\mathbb{N}$) : CSExp =

{| $nodeOut$.l.n |}

component in RoboChart in *Circus*, next, we conclude this section with final considerations.

5 AnglePIDANN Circus Program

5.1 Preliminary Material

5.2 Process Definition

6 Notes

Differences to the CSP semantics, where the Circus actions representing the CSP processes differ. We have proved, in FDR, for the *AnglePIDANN* and *AnglePIDANN*2 examples, and for a binarised version of *AnglePIDANN*, that our Circus semantics are equivalent, in the traces model, to the CSP semantics.

- channels are renamed, no longer use indexed channels, the indexed channel abstraction. There are multiple cases, on each process, with a guard and each process is chained together by external choice, such that only one process should not evaluate to $STOP$, the rest are $STOP \Box P$, which evaluates to P.
- ANNHiddenEvts defined constructively, not all those events without the inputs and outputs.
- We are not hiding all of node out, like we do in the original, because unnnecessary, and in the meta-language, we have to define it anyway, $\{\!|\, nodeOut.1.1 \,|\!\}$ we have to define anyway, so its cleaner, to define we synchronise on that, then hide just that.
- parallel synchronisation, without the variable sets, needs | characters, from the circus guide, it should not, but it does for us: $[\![\mid \{\!|\, layerRes11 \,|\!\} \mid]\!]$. $[\![layerRes11]\!]$ does not compile, when it says this is valid syntax.
- No longer using replicated alphabetsied parallel in *HiddenLayers*, we are now using multiple generalised parallel in *HiddenLayers*.

Issues or Questions:

- Stateless, so we omit the state reserved word, allowed in CZT, but Marcel's BNF seem to imply it is always required.
- In this document, I am using the Circus latex style, not the csp or CZT, so we are using circinterrupt instead of interrupt for CSP interrupt, but both produce the same symbol.
- We use binarised parameters, and the *sign* activation function instead of *relu*, for validation of our Circus programs.

- We use the roboworld 2d toolkit, we only really need the real type, and the decimal point definition, in CZT, but we do need this declaration, otherwise the process would be very ugly, but this is required as well as the standard circus toolkit, to write the circus programs in CZT.

7 ANN Circus semantics in the Circus metamodel

7.1 Important Classes in metamodel

- Term (abstract class, represents a term).
- Para -¿ Term (abstract class, represents a paragraph).
- Types of Para: AxPara, ActionPara, ChannelPara, ChannelSetPara, ConjPara, FreePara, GivenPara, NameSetPara, ProcessPara, etc.
- Sect, is a Term, abstract class.
- Concrete subclasses of Sect, ZSect, that is it, just ZSect.
- ZSect -¿ Sect (Z section, has name: EString, paraList: ParaList, parents: Parent).
- ParaList, abstract class, list of paragraphs, ZParaList, paras, ZParaList is a concrete type of ParaList.
- ZParaList, concrete ParaList, list of, paras: Para.
- Para,
- ConstDecl, has name: ZName, and expr: Expr
- Expresssions, Expr, types of expression:
- Expr, is a term, an abstract class.
- Concrete instantiations: BasicChannelSetExpr, BindExpr, CondExpr, NumExpr, RefExpr, SigmaExpr, SchExpr, RefExpr,
- RefExpr
- SchTExt, ZSchText, schema text.
- ZName, is word, id, operatorName, strokesLsit, strokesList, list of Z strokes used.
- word is the name of ZName, an EString.
- id, 8571, just a number in CZT. real is

Notes from CZT API, https://czt.sourceforge.net/corejava/corejava-z/apidocs/index.html

- Stroke, is a Term, an abstract Stroke,
- Stroke, ?,

This is the AST,

7.2 Channel declarations, and the ReLU declaration

The CZT Circus AST, representation of our example, AST for our example:

- Horizontal Definition Paragraph (
- Schema text "Value"
- List of declarations "Value"
- Constant declaration "Value" (has a name and a reference expression) [ConstDecl in EMF]
- name, then a reference expression, has name,
- reference expression "real", [RefExpr in EMF]
- reference expression, has a list of expressions, [expression list].
-

Name of the constant declaration, is "Value", name of the reference expression, "real", list of expressions, no Z strokes,

7.3 Mechanisation Notes (EOL)

Mechanisation, in CZT Circus API, of the various BNF rules that we refer to, is:

- Program: multiple circus paragraphs, in CZT AST, this is: List of Paragraphs, in Tool, it is ZParaList, in Circus AST, you can put Circus and Z paragraphs in this, it is just a list of Paras, which can be either.
- ProcessPara, is ProcDecl, potentially, defining a process paragraph.
- We are using, in EOL, no c : ANNController, that is the self, that is the context of the operation, all the semantics are operations on ANNController objects.
- CircusProcess, absrtact, of BasicProcess, with parameters, mainAction, ontheFlyParagraphs, paragraph lists, Axiom paragraphs, state para. State paragraph list. Them has local paragraphs.

- ProcessPara, has a CircusProcess, a namelist, a name, and if it is a basic process or not.
- The AnglePIDANN, overall, is a process pargraph, not a circusprocess, then the basic process, then basic process has a list of paragraphs.
- Then, each is an action paragraph, each CSP process,
- Treat the $Value == \mathbb{R}$, horizontal definition paragraph, as Part of the Toolkit, as imported in CZT.
- Not using explicit paragraph lists, generating a document with just using, the actual paragraphs. They aren't grouped in the CZT file anyway, it was grouped, in the metalanguge, in the BNF, more just to show, to describe what they are.
- We don't have a section, that would have a list of paragraphs, we just have the list of paragraphs, doesn't matter. Still automatically generates them.
- Do we need the explicit import? and Section header? in the M2M? Does that exist yet?
- Each channel is declared in its own channel paragraph, one channel paragraph per channel declaration.
- layerRes and nodeOut are declared as STRINGS, not channels, as easier just to get the names, then call the functions to get the channel paragraphs, and declarations, from other places.
- Process Paragraph, top level, name "AnglePIDANN". Basic Process, then has list of paragraphs, aciton paragraphs, then horizontal definition paragraph.
- Axiomatic description paragraph relu. That can be in the toolkit as well.
- createProcessPara, needs a Z!CircusProcess, sets the circusprocess, to that, name: and is BasicProcess, isBasicProc, true,
- createParallelProcess, name, left, and right. cs_name, reference to channelset name, creates a channel set, with the reference expression, of cs_name.
- Can also, createParallelProcess, with $channel_names$ as a set, left, and right.
- createCallProcess, call expression,
- createBasicProcess, mainActionName, sets the main action.
- give a sequence of
- the basic process, paragraph lists, is the list.

- paragraphs, is the sequence.
- From a Z!Para, can create a basic process, with a main action name, this is a Z!BasicProcess.
- create action, createPrefixingAction, createAction1.
- mechanisation of process, top level: Process Paragraph, then has a name, the process paragraphs name is "AnglePIDANN", the name of the RC component. That is just the anglepidann.name, it does work.
- then has a basic process, then in this basic process, has a pargraph list, then it has ACTION PARAGRAPHS, then a main action, then a horizontal definition paragraph, default?
- Fang's mechanisation, find process paragraphs, then basic processes, and action paragraphs.
- createProcessPara(name: String, isBasicProc: Boolean), context of a CircusProcess, creates a ProcessPara. sets the name to name, and isBasicProcess,
- createCallProcess(), creates a Ref Expression, a process that calls a reference, a refernece expression.
- createParallelProcess,
- createActionPara(), Circus Action, returns an ActionPara, takes a CircusAction, creates an action paragraph. with the circus action = to self.
- createCircusAction,
- how the events are represented in memory,
- Events are not ordered, in EMF RoboChart models, based on when users, I saw that somewhere they were? But not in EOL itself, not ordered.
- ordered is FALSE on events, saw in representation, in parts of xtext code, not in EMF, not an ordered.
- ORDERING WORKS, NOT SURE WHY, SAYS UNORDERED, IN EMF, BUT IT SEEMS TO WORK, EVEN IN INTERFACES. MULTIPLE, ONE AT A TIME, SURELY.

A AnglePIDANN2 Circus Program

Used to make an example with more than one hidden layer, and more than one layer per node.

B CSP Semantics Sketch

$$Value == \mathbb{R}$$

channel $layerRes01 : Value$
channel $layerRes02 : Value$
channel $layerRes11 : Value$
channel $layerRes21 : Value$
channel $nodeOut111 : Value$
channel $nodeOut112 : Value$
channel $nodeOut211 : Value$
channel $terminate$

channel $adiff\_in : \mathbb{R}$
channel $anewError\_in : \mathbb{R}$
channel $angleOutputE\_out : \mathbb{R}$

channelset $ANNHiddenEvts == \{\!|\ layerRes11\ |\!\}$

DON'T NEED THIS, for the meta-language.

$$relu : Value \rightarrow Value$$

$$\forall x : Value \bullet (x < 0 \Rightarrow (x, 0) \in relu) \land (x \geq 0 \Rightarrow (x, x) \in relu)$$

Figure 3: The preliminary Circus paragraphs, for the *AnglePIDANN* example.

$$
\begin{array}{l}
\textbf{process}\ \mathit{AnglePIDANN} \mathrel{\widehat=} \\
\quad \textbf{begin} \\
\quad \mathit{Collator} \mathrel{\widehat=} l, n, i : \mathbb{N};\ \mathit{sum} : \mathit{Value} \bullet \\
\qquad \big(l = 1 \land n = 1 \land i = 0\big)\ \&\ \mathit{layerRes}11\,!(\mathit{relu}(\mathit{sum} + (0.125424))) \longrightarrow \textbf{Skip} \\
\qquad \Box\,\big(l = 1 \land n = 1 \land i = 1\big)\ \&\ \mathit{nodeOut}111\,?x \longrightarrow \mathit{Collator}(l, n, (i-1), (\mathit{sum} + x)) \\
\qquad \Box\,\big(l = 1 \land n = 1 \land i = 2\big)\ \&\ \mathit{nodeOut}112\,?x \longrightarrow \mathit{Collator}(l, n, (i-1), (\mathit{sum} + x)) \\
\qquad \Box\,\big(l = 2 \land n = 1 \land i = 0\big)\ \&\ \mathit{layerRes}21\,!(\mathit{relu}(\mathit{sum} + (-0.107753))) \longrightarrow \textbf{Skip} \\
\qquad \Box\,\big(l = 2 \land n = 1 \land i = 1\big)\ \&\ \mathit{nodeOut}211\,?x \longrightarrow \mathit{Collator}(l, n, (i-1), (\mathit{sum} + x)) \\
\quad \mathit{NodeIn} \mathrel{\widehat=} l, n, i : \mathbb{N} \bullet \\
\qquad \big(l = 1 \land n = 1 \land i = 1\big)\ \&\ \mathit{layerRes}01\,?x \longrightarrow \mathit{nodeOut}111\,!(x * (1.22838)) \longrightarrow \textbf{Skip} \\
\qquad \Box\,\big(l = 1 \land n = 1 \land i = 2\big)\ \&\ \mathit{layerRes}02\,?x \longrightarrow \mathit{nodeOut}112\,!(x * (0.132874)) \longrightarrow \textbf{Skip} \\
\qquad \Box\,\big(l = 2 \land n = 1 \land i = 1\big)\ \&\ \mathit{layerRes}11\,?x \longrightarrow \mathit{nodeOut}211\,!(x * (0.744636)) \longrightarrow \textbf{Skip} \\
\quad \mathit{Node} \mathrel{\widehat=} l, n, \mathit{inpSize} : \mathbb{N} \bullet \\
\qquad \big(l = 1 \land n = 1\big)\ \&\ \ ((\Big|\Big|\Big|\ i : 1 \mathinner{..} \mathit{inpSize} \bullet \mathit{NodeIn}(l, n, i)) \\
\qquad\qquad\qquad\qquad [\![\ |\ \{\!|\ \mathit{nodeOut}111, \mathit{nodeOut}112\ |\!\}\ |\ ]\!] \\
\qquad\qquad\qquad\qquad \mathit{Collator}(l, n, \mathit{inpSize}, 0) \setminus \{\!|\ \mathit{nodeOut}111, \mathit{nodeOut}112\ |\!\}) \\
\qquad \Box\,\big(l = 2 \land n = 1\big)\ \&\ \ ((\Big|\Big|\Big|\ i : 1 \mathinner{..} \mathit{inpSize} \bullet \mathit{NodeIn}(l, n, i)) \\
\qquad\qquad\qquad\qquad [\![\ |\ \{\!|\ \mathit{nodeOut}211\ |\!\}\ |\ ]\!] \\
\qquad\qquad\qquad\qquad \mathit{Collator}(l, n, \mathit{inpSize}, 0) \setminus \{\!|\ \mathit{nodeOut}211\ |\!\}) \\
\quad \mathit{HiddenLayer} \mathrel{\widehat=} l, s, \mathit{inpSize} : \mathbb{N} \bullet \\
\qquad ([\![\{\!|\ \mathit{layerRes}01, \mathit{layerRes}02\ |\!\}\ ]\!]\ i : 1 \mathinner{..} s \bullet \mathit{Node}(l, i, \mathit{inpSize})) \\
\quad \mathit{HiddenLayers} \mathrel{\widehat=} \\
\qquad \mathit{HiddenLayer}(1, 1, 2) \\
\quad \mathit{OutputLayer} \mathrel{\widehat=} \\
\qquad [\![\{\!|\ \mathit{layerRes}11\ |\!\}\ ]\!]\ i : 1 \mathinner{..} 1 \bullet \mathit{Node}(2, i, 1) \\
\quad \mathit{ANN} \mathrel{\widehat=} \\
\qquad ((\mathit{HiddenLayers}[\![\ |\ \{\!|\ \mathit{layerRes}11\ |\!\}\ |\ ]\!] \mathit{OutputLayer}) \setminus \mathit{ANNHiddenEvts})\ ;\ \mathit{ANN} \\
\quad \mathit{ANNRenamed} \mathrel{\widehat=} \\
\qquad (\mathit{ANN})\ [\mathit{layerRes}01, \mathit{layerRes}02, \mathit{layerRes}21 := \\
\qquad\qquad\qquad \mathit{anewError\_in}, \mathit{adiff\_in}, \mathit{angleOutputE\_out}] \\
\qquad\qquad \triangle\ \mathit{terminate} \longrightarrow \textbf{Skip} \\
\quad \bullet\ \mathit{ANNRenamed} \\
\quad \textbf{end}
\end{array}
$$

Figure 4: *AnglePIDANN* example, in Circus

process $AnglePIDANN2 \mathrel{\widehat=}$ **begin**
$Collator \mathrel{\widehat=} l, n, i : \mathbb{N};\ sum : Value \bullet$
$\big(l = 1 \land n = 1 \land i = 0\big)\ \&\ layerRes11\,!(sign(sum + (0))) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 1 \land n = 1 \land i = 1\big)\ \&\ nodeOut111\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 1 \land n = 1 \land i = 2\big)\ \&\ nodeOut112\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 1 \land n = 2 \land i = 0\big)\ \&\ layerRes12\,!(sign(sum + (0))) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 1 \land n = 2 \land i = 1\big)\ \&\ nodeOut121\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 1 \land n = 2 \land i = 2\big)\ \&\ nodeOut122\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 1 \land n = 3 \land i = 0\big)\ \&\ layerRes13\,!(sign(sum + (0))) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 1 \land n = 3 \land i = 1\big)\ \&\ nodeOut131\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 1 \land n = 3 \land i = 2\big)\ \&\ nodeOut132\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 2 \land n = 1 \land i = 0\big)\ \&\ layerRes21\,!(sign(sum + (0))) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 2 \land n = 1 \land i = 1\big)\ \&\ nodeOut211\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 2 \land n = 1 \land i = 2\big)\ \&\ nodeOut212\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 2 \land n = 1 \land i = 3\big)\ \&\ nodeOut213\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 3 \land n = 1 \land i = 0\big)\ \&\ layerRes31\,!(sign(sum + (0))) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 3 \land n = 1 \land i = 1\big)\ \&\ nodeOut311\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 4 \land n = 1 \land i = 0\big)\ \&\ layerRes41\,!(sign(sum + (0))) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 4 \land n = 1 \land i = 1\big)\ \&\ nodeOut411\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$\Box\big(l = 4 \land n = 2 \land i = 0\big)\ \&\ layerRes42\,!(sign(sum + (0))) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 4 \land n = 2 \land i = 1\big)\ \&\ nodeOut421\,?x \longrightarrow Collator(l, n, (i-1), (sum + x))$
$NodeIn \mathrel{\widehat=} l, n, i : \mathbb{N} \bullet$
$\big(l = 1 \land n = 1 \land i = 1\big)\ \&\ layerRes01\,?x \longrightarrow nodeOut111\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box$
$\big(l = 1 \land n = 1 \land i = 2\big)\ \&\ layerRes02\,?x \longrightarrow nodeOut112\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box$
$\big(l = 1 \land n = 2 \land i = 1\big)\ \&\ layerRes01\,?x \longrightarrow nodeOut121\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box$
$\big(l = 1 \land n = 2 \land i = 2\big)\ \&\ layerRes02\,?x \longrightarrow nodeOut122\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box$
$\big(l = 1 \land n = 3 \land i = 1\big)\ \&\ layerRes01\,?x \longrightarrow nodeOut131\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box$
$\big(l = 1 \land n = 3 \land i = 2\big)\ \&\ layerRes02\,?x \longrightarrow nodeOut132\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box$
$\big(l = 2 \land n = 1 \land i = 1\big)\ \&\ layerRes11\,?x \longrightarrow nodeOut211\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 2 \land n = 1 \land i = 2\big)\ \&\ layerRes12\,?x \longrightarrow nodeOut212\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 2 \land n = 1 \land i = 3\big)\ \&\ layerRes13\,?x \longrightarrow nodeOut213\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 3 \land n = 1 \land i = 1\big)\ \&\ layerRes21\,?x \longrightarrow nodeOut311\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 4 \land n = 1 \land i = 1\big)\ \&\ layerRes31\,?x \longrightarrow nodeOut411\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$\Box\big(l = 4 \land n = 2 \land i = 1\big)\ \&\ layerRes31\,?x \longrightarrow nodeOut421\,!(x * (1)) \longrightarrow \mathbf{Skip}$
$Node \mathrel{\widehat=} l, n, inpSize : \mathbb{N} \bullet$
$\big(l = 1 \land n = 1\big)\ \&\ (($
$\Big|\Big|\Big|\ i : 1\,..\,inpSize \bullet NodeIn(l, n, i))$
$[\![\ |\ \{\!|\ nodeOut111, nodeOut112\ |\!\}\ |\ ]\!]$
$Collator(l, n, inpSize, 0) \setminus \{\!|\ nodeOut111, nodeOut112\ |\!\}$
$)$
$\Box$
$\big(l = 1 \land n = 2\big)\ \&\ (($
$\Big|\Big|\Big|\ i : 1\,..\,inpSize \bullet NodeIn(l, n, i))$
$[\![\ |\ \{\!|\ nodeOut121, nodeOut122\ |\!\}\ |\ ]\!]$
$Collator(l, n, inpSize, 0) \setminus \{\!|\ nodeOut121, nodeOut122\ |\!\}$
$)$
$\Box$
$\big(l = 1 \land n = 3\big)\ \&\ (($
$\Big|\Big|\Big|\ i : 1\,..\,inpSize \bullet NodeIn(l, n, i))$

$$
\begin{array}{l}
ANNRenamed = ANN \triangle end \longrightarrow SKIP \\
ANN = \\
\quad ((HiddenLayers \parallel [\{\!|\ layerRes.(layerNo - 1)\ |\!\}] OutputLayer) \setminus ANNHiddenEvts) \\
\quad ;\ ANN \\
ANNHiddenEvts = \Sigma \setminus \{\!|\ layerRes.0, layerRes.layerNo, end\ |\!\} \\
HiddenLayers = \|\, i : 1 \,..\, layerNo - 1 \bullet [\,\{\!|\ layerRes.(i-1), layerRes.i\ |\!\}\,] \\
\qquad\qquad HiddenLayer(i, layerSize(i), layerSize(i-1)) \\
HiddenLayer(l, s, inpSize) = \|\, i : 1 \,..\, s \bullet [\{\!|\ layerRes.(l-1)\ |\!\}] Node(l, i, inpSize) \\
Node(l, n, inpSize) = \\
\quad (\ (\|\|\|\, i : 1 \,..\, inpSize \bullet NodeIn(l, n, i)) \\
\qquad \parallel [\ \{\!|\ nodeOut.l.n\ |\!\}\ ] \\
\qquad Collator(l, n, inpSize)\ ) \setminus \{\!|\ nodeOut\ |\!\} \\
NodeIn(l, n, i) = layerRes.(l-1).n?x \longrightarrow nodeOut.l.n.i!(x * weight) \longrightarrow \mathbf{Skip} \\
Collator(l, n, inpSize) = \mathbf{let} \\
\qquad\qquad C(l, n, 0, sum) = layerRes.l.n!(ReLU(sum + bias)) \longrightarrow \mathbf{Skip} \\
\qquad\qquad C(l, n, i, sum) = nodeOut.l.n.i?x \longrightarrow C(l, n, (i-1), (sum + x)) \\
\qquad \mathbf{within} \\
\qquad\qquad C(l, n, inpSize, 0) \\
OutputLayer = \|\, i : 1 \,..\, layerSize(layerNo) \bullet \ [\,\{\!|\ layerRes.(layerNo - 1)\ |\!\}\,] \\
\qquad\qquad Node(layerNo, i, layerSize(layerNo - 1))
\end{array}
$$

Figure 6: CSP ANN Semantic Pattern.