A compositional approach using Keras for neural networks in real-time systems

Xin Yang
The University of Auckland
Auckland, New Zealand
xyan510@aucklanduni.ac.nz

Partha Roop
The University of Auckland
Auckland, New Zealand
p.roop@auckland.ac.nz

Hammond Pearce
The University of Auckland
Auckland, New Zealand
hammond.pearce@auckland.ac.nz

Jin Woo Ro
The University of Auckland
Auckland, New Zealand
jro002@aucklanduni.ac.nz

***Abstract*—Real-time systems are designed using model-driven approaches, where a complex system is represented as a set of interacting components. Such a compositional approach facilitates design of simpler components, which are easier to validate and integrate with the overall system. In contrast to such systems, data-driven systems like neural networks are designed as monolithic black-boxes to capture the non-linear relationship from inputs to outputs. Increasingly, such systems are being used in safety-critical real-time systems. Here, a compositional approach would be ideal. However, to the best of our knowledge, such a compositional approach is lacking while designing data-driven components based on neural networks.**

This paper formalises this problem by developing the concept of Composed Neural Networks (CpNNs) by extending the well known Keras python framework. CpNNs formalise the synchronous composition of several interacting neural networks in Keras. Further, using the developed semantics, we enable modular compilation from a given CpNN to C code. The generated code is suitable for the Worst-Case Execution Time (WCET) analysis. Using several benchmarks we demonstrate the superiority of the developed approach over a recently proposed approach using Esterel, as well as the popular Python package Tensorflow Lite. For the given benchmarks, our approach is superior to Esterel with an average WCET reduction of 64.06%, and superior to Tensorflow Lite with an average measured WCET reduction of 62.08%.

***Index Terms*—Deep neural networks, Cyber Physical Systems, Real-Time**

I. Introduction

Software Engineering (SE) is founded on the principles of abstraction and composition. Paranas's pioneering ideas espoused through the concept of *abstract interfaces* [11] led to the development of object-oriented methods, such as encapsulation and inheritance, now common place in many object oriented programming languages. Likewise, the compositional approach is also widespread in many other SE applications, including in safety-critical software used in Cyber-Physical Systems (CPSs) [2], as nicely summarised in a more recent survey by Tripakis [21]. Most of these systems use techniques founded on formal models, which aid in both analysis and automated code generation through semantic preserving translations, particularly in tools such as Ansys SCADE [10]. SCADE is used for the design of safety-critical software for aerospace applications. Being founded on formal models, these techniques are known as *model-driven design*.

An orthogonal paradigm for automated reasoning over unstructured data has come to the fore in recent times, which may be termed as *data-driven design* [22]. Artificial Neural Networks (ANNs) are a specific case that is widely adopted to determine the non-linear relationships between a given set of inputs and their corresponding outputs. Typically, ANN operations are broken over two distinct phases: *training* and *inference*. During the training phase, the weights of connections between neurons are learned using a set of training data. Once trained over this dataset, the networks are used for making inferences when exposed to new input data. These data-driven techniques work reliably when used in advisory roles, and provide a confidence interval that is indicative of their accuracy.

While model-driven and data-driven techniques work quite well in their respective domains, problems may arise when they are combined. For example, ANNs are being used recently in Autonomous Vehicles (AVs), where their decision influences control actions. For example, an AV may automatically detect traffic signs and pedestrians using computer vision algorithms that are based on neural networks. These data-driven tasks can have significant safety implications when they influence control driven tasks such as steering, adaptive cruise control, and/or vehicle braking. In a recent survey Tripakis [22] outlines the challenges in designing such systems.

Considering these challenges, there have been several recent efforts to combine formal methods based techniques [18] with ANN-based solutions. For example, there is a promising approach that uses abstract interpretation to deal with adversarial perturbations of Neural Network (NN) based image classification techniques [7]. However, most formal techniques for analysing the safety of NNs are developed to provide functional guarantees rather than timing guarantees. This paper, on the other hand, focuses on real-time systems and hence will consider the issue of providing timing guarantees.

Recently, Roop et al. [16] proposed synchronous NNs specifically for designing real-time applications. Their approach uses the Esterel [4] language for designing ANNs. They combine the Darknet [14] C-library for NNs to develop complex applications. However, their approach has several limitations. First, industry practice for developing neural networks is based on libraries such as Tensorflow or Keras. These provide many features for rapid prototyping as well as

978-3-9819263-4-7/DATE20/©2020 EDAA

efficient implementation. It is unlikely that ANN developers can be motivated to switch to a synchronous language such as Esterel, which provides minimal support for AI. Secondly, the Esterel-approach relies on the Darknet software library for the development of complex NNs. This ironically is self-defeating, as the Darknet libraries are not themselves amenable to timing analysis. Hence, the complex benchmarks in [16] were not statically analysed for their WCET.

In this paper, we overcome the limitations of [16] by developing a new compiler for the popular, open-source Keras NN library for ANNs based on Multi-Layer Perceptrons (MLPs). Our tool allows for ANNs which have been trained in Keras to be exported and then synchronously composed with other networks as well as with custom functions in C. These custom functions may be used for modelling the physics of the controlled plant, or for implementing other heuristic-based functionality. Thus, the developed approach is an ideal fit for the design of AI-based CPS.

Then, using new compositional semantics enabling communication between the different components of the designed system, our compiler will generate correct-by-construction C code. The produced code is amenable for the execution of any embedded target platform without any run-time support, and is suitable for static timing analysis (a requirement for formally deriving Worst-Case-Execution-Time i.e. WCET bounds). Using a new case study as well as the original set of Esterel benchmarks from [16], we demonstrate the relative superiority of our proposed approach, which is termed as Composed Neural Networks (CpNN). We have also compared with the well known Tensorflow Lite framework for embedded platforms. **To the best of our knowledge, our approach is the first framework for machine learning of real-time systems modelled using a composition of NNs with their adjoining plant**.

The rest of the paper is organised as follows: In Section II we introduce a composed NN controller example that illustrates the synchronous control in CPSs. In Section III, we provide the semantics to show how a CpNN may be compiled to low-level Kernel statements. Based on the semantics, we present the compositional design and automated code generation of CpNNs in Section IV, as well as a methodology for statically determining the WCET of this code in Section V. In Section VI, we compare the developed approach relative to the Esterel [16] and the Tensorflow Lite framework from Google. Finally, we provide concluding remarks in Section VII.

II. Motivating Example

In general, road vehicles are most efficient when they follow one another like a train [23]. The role of an ideal Adaptive Cruise Control (ACC) controller would be to automatically adjust the vehicle acceleration and velocity while driving to achieve such an optimum behaviour.

A possible design of an ACC controller is to mimic human driving [19]. However, the conventional way of capturing human driving behaviour through the calibration of a specific model has several drawbacks on system flexibility. Thus, NNs, which are trained using actual human driver data, are increasingly used for ACC.

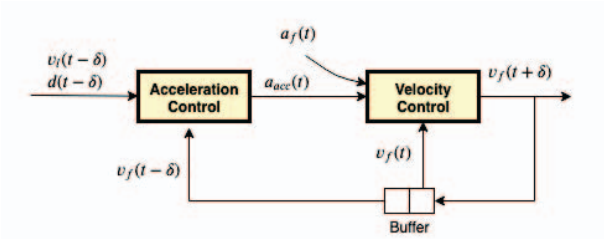


Fig. 1. Motivating example ACC design.

We present the design of an ANN-based ACC controller in Figure 1. As there are two NNs in this system, it provides an ideal pedagogic use-case for the CpNN approach. As can be seen, it consists of two NNs, termed the *acceleration controller* and the *velocity controller*. The variables used in this example are listed as follows:

- t: time (s).
- $d(t)$: spacing between the vehicles (m) at time t.
- $v_l(t)$: leading vehicle velocity (ms^{-1}) at time t.
- $v_f(t)$: following vehicle velocity (ms^{-1}) at time t.
- $a_f(t)$: following vehicle acceleration (ms^{-2}) at time t.
- $a_{acc}(t)$: desired ACC acceleration (ms^{-2}) at time t.
- δ: unit delay (s).

The acceleration control NN requires three input values, $\{d(t-\delta), v_l(t-\delta), v_f(t-\delta)\}$, to generate $a_{acc}(t)$ as the output. Intuitively, the previous values of d, v_l, and v_f are used to compute the desired acceleration a_{acc}. On the other hand, the velocity control NN takes three input values of $a_{acc}(t)$, $v_f(t)$, and $a_f(t)$, and outputs the next velocity $v_f(t+\delta)$. Considering that the unit delay can be small, the desired ACC acceleration is computed not too significantly different from the current acceleration. Thus, in this case study, we assume that the desired acceleration is reached after the unit delay, hence, the next acceleration value is $a_f(t+\delta) = a_{acc}(t)$.

The experimental car-following data, obtained from human drivers from a well known study [13], is used to train the NNs. In this experiment, nine vehicles are driven in a circular track as a platoon of vehicles. With the sampling resolution of 0.1 seconds, the spacing, velocities, and accelerations of each vehicle are recorded. Specifically, the leader vehicle is requested to drive as if there are virtual traffics in front of him/her, for instance, approaching to an intersection. Thus, several car-following patterns are recorded.

For a "human-like" ACC, the acceleration control NN needs to produce the desired acceleration $a_{acc}(t)$ very close to the acceleration values in the human dataset [13]. More precisely, the value of $a_{acc}(t)$ needs to be the next acceleration value $a_f(t+\delta)$. Since the sampling resolution is 0.1 seconds, the unit delay is set $\delta = 0.1$. Likewise, the velocity control NN is trained by setting $a_{acc}(t) = a_f(t+\delta)$.

A. ANN architecture details

The *acceleration controller* is structured following the network in [19], which contains three neurons in the input layer to receive v_l, d, and v_f. The input layer is followed by one hidden layer with ten neurons. There is one neuron at the output layer which produces a_{acc}. For the *velocity controller*, there are three neurons in its input layer, v_f, a_{acc} and a_f. There is only one hidden layer with ten neurons, and one neuron in the output layer to generate v_f. Both NNs are trained in Keras, using the data from [13].

III. Operational Semantics

In general, we adopt synchronous semantics for the execution of CpNNs. Here, each component (i.e. any function or ANNs) is viewed as executing in a logically concurrent manner with its operation broken over a series of *ticks*. To simplify the process, we adopt *delayed synchronous* communication, which means that any communication between producer/consumer components may only occur at tick boundaries. Due to this delayed synchronous composition of components, the overall system is deterministic and deadlock free by design [3].

As an example, consider the running case study of the ACC. It is possible to construct this as a synchronous system pseudocode as shown below. Here, the AccelerationController and the VelocityController operate with logical concurrency.

```
1 while (true):
2   get v_l, d from sensors
3   a_acc = AccelerationController(v_l, d, v_f)
4   store a_acc
5   assign a_f with previous a_acc
6   v_f = VelocityController(a_acc, a_f, v_f)
```

Listing 1. ACC in Keras Semantics

A. Semantics

To define CpNN operation we must first define a minimal set of kernel statements.

Definition 1 (CpNN component): We define any component in a CpNN (e.g. an ANN or an arbitrary function) as a tuple $C = \langle I, O, \eta \rangle$ where

- I is a finite collection of input variables with its domain being $\mathbf{I} = \mathbb{R}^n$.
- O is a finite collection of output variables with its domain being $\mathbf{O} = \mathbb{R}^m$.
- $\eta : \mathbf{I} \rightarrow \mathbf{O}$ is the non-linear function (termed the component function) that provides the behaviour of a given component i.e. when provided a vector of input of size n produces an output vector of size m.

It is trivial to execute C synchronously: within each reaction (or tick), outputs O are generated from inputs I using function η. We define this operation as a kernel statement $C()$, which is the simplest operation. This execution could also be represented as $\mathbf{O} \longleftarrow C(\mathbf{I})$.

Table I presents the full set of kernel statements. These are inspired by the synchronous programming language Esterel. Note the usage of the *pause* operator. In synchronous languages, an operator like *pause* is used to denote state boundaries. This allows us to use the progression of ticks as a description of the passage of time. In addition, tick boundaries is where communication between different components can occur.

Kernel Statements	Description
$C()$	Execute component C
$x \longleftarrow C(y)$	Data assignment to x from C with input y
pause	A tick boundary
loop ... end	Endless loop
a ; b	Sequential execution of a followed by b
$a \parallel b$	Concurrent execution of a and b
[...]	Denote a block of kernel statements

TABLE I
Our Kernel statements

B. Sequential and Parallel Operators

In addition to running components, it is possible to combine compositions of components using the *Sequential* and *Parallel* kernel operators. Kernel statements separated by ; will run in a *sequence*, that is, one after the other. Execution of a block of kernel statements separated by ; completes in order.

On the other hand, kernel statements separated by || will operate concurrently. Execution of blocks of kernel statements separated by || is not complete until all statements are completed, but they may be completed in any order.

C. ACC Example

These Kernel statements allow us to construct complex systems of interconnected components. For instance, the ACC controller could be represented with the following kernel statements.

```
1  loop
2    pause;
3    v_l, d ⟵ ReadInputs();
4    [
5      a_acc ⟵ NN_acc(v_l, d, v_f);
6      pause;
7    ] || [
8      a_f ⟵ store_a(a_acc);
9      pause;
10   ] || [
11     v_f ⟵ NN_vel(a_acc, a_f, v_f);
12     pause;
13   ]
14   ;
15   EmitOutputs(v_f);
16   pause;
17 end
```

Listing 2. ACC Mapping Using Kernel Statements

Here, we see there are two functions $ReadInputs$ and $EmitOutputs$, each of which is responsible for environmental interaction. Then, the two internal ANNs are encapsulated as components NN_{acc} and NN_{vel}, respectively. A customer function $store\_a$ is also involved in this CpNN to store the previous a_{acc} value. The $pause$ statements control the overall

flow of this system: it will take three ticks to complete its computation. Let us consider the three inputs to NN_{acc}, V_l and d may come from different sensors, whereas V_f is generated by NN_{vel}. Since the inputs are from different sources, they may arrive at a different time. Hence, $pause$ provides a concurrency control to the CpNN.

IV. Code Generation

Our CpNN compiler generates C code, based on the semantics from Section III. The code generation for CpNN consists of two parts, namely Keras2C and CpNN2C, as shown in Figure 2

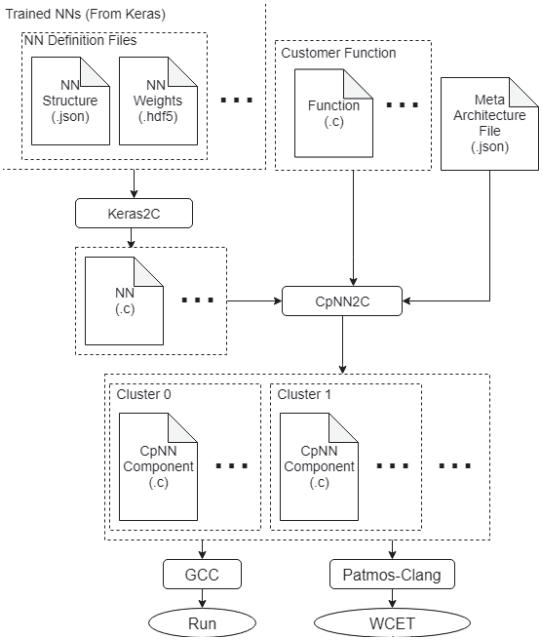


Fig. 2. An Overview of CpNN Compiler

After we train a neural network in Keras, we can save the pre-trained model into two files, one is a structure file in *json* format, and the other one is a weights file in *hdf5* format. Keras2C first reads these two files and extracts information such as the neural network structure, the weights and the bias. Then, the extracted information is passed to Keras2C's code generation template. The template is developed using Jinja2, which is a general purpose templating language which allows users to design templates for automatic code generation. [15]

Further, the generated NN in C can be fed into CpNN2C as a CpNN component. CpNN2C receives a collection of CpNN components, including both generated NNs from Keras2C and customer functions such as *store_a*. They are composed to create the CpNN. While in this paper we treat each ANN as a "black-box" component, we can also configure the compiler to generate synchronous code which executes the networks in a "layer-by-layer" approach as suggested in [16].

In addition, CpNN2C also receives a *meta architecture file*, which identifies how the CpNN components are composed. The *meta architecture file* has three sections, namely the *Type* section, the *Instance* section and the *Connection* section. The *Type* section contains the information of each CpNN component, such as a component's name, input/output size and its *.c* file location. The *Instance* section locates the CpNN component into clusters. Within the same cluster, CpNN components are executed sequentially. For different clusters, they are executed concurrently. This parallel execution uses logical concurrency, where the NNs in different clusters are logically running at the same time but processing the data from different ticks. Therefore, the C code is a serialised representation of the parallel execution. Hence, clusters implement the $a; b$ and $a||b$ semantics from Section III.

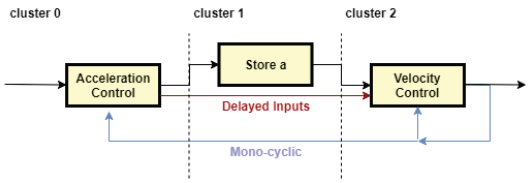


Fig. 3. ACC model connection types

Next, we define how the data is transferred between the CpNN components in the *Connection* section. Usually, this is a simple copy from outputs to inputs. However, it may be the case that networks require internal delays. Our compiler provides a pre function to implement this: i.e. if data needs to cross multiple cluster and $pause$ boundaries, it can be obtained using pre. In other words, CpNN supports different synchronous connection types. For example, the ACC model involves two synchronous connection types, delayed inputs and mono-cyclic connection as shown in Figure 3. In ACC, a_f is produced by $store\_a$, where a_{acc} is fed directly from $Acceleration\ Control$ to $Velocity\ Control$. The a_{acc} input needs to be delayed to ensure that it will arrive in the *velocity control* with a_f concurrently. Therefore, $store\_a$ and the data transfer of a_{acc} are grouped in an individual cluster, with a pre to their outputs. Hence, a_{acc} is delayed by one tick to wait for the execution of $store\_a$, and the data from both components are now guaranteed to arrive in *velocity control* at the same time.

The mono-cyclic connection is similar in spirit to *recurrent neural networks*, which considers the impact of historical patterns. In the ACC model [6], the output V_f (see notation earlier for case study) from $Velocity\ Control$ is fed to both $Acceleration\ Control$ and $Velocity\ Control$ itself.

CpNN2C will automatically generate C code to connect the CpNN components based on the *meta architecture file*. Further, the Make tool from build-essential package will compile the generated C code to either an executable binary or a series of files suitable for WCET analysis. For the binary compilation, a GNU Compiler Collection (GCC) is applied [20]. For the WCET analysis, we used *patmos-clang* [17], which is a compiler designed to support the timing analysis of C programs, elaborated in Section V.

V. WCET Analysis

Timing analysis of software requires the model of the underlying architecture [9]. In our case, we target the time predictable soft-core Patmos architecture [17] along with the associated timing analyser Platin [8]. As a soft-core processor, we implement Patmos using the Altera DE2-115 FPGA board with a 50MHz clock.

In addition to a hardware model, code that is to be subject to static timing analysis must itself avoid certain software features. For instance, unbounded loops, function pointers, and interrupt-driven program flow can all result in problematic control flow graphs. As such, CpNN2C (and its components) avoid these constructs, and generate simple, statically scheduled execution code. As such, all compiled code using our tool can be subject to static timing analysis.

Recall that a CpNN is a collection of clusters running sequentially. Within each cluster, the path with the largest cycle delays (as discovered by Platin) will become the WCET of that cluster. Then, the WCET of the whole CpNN system is the sum of the WCET of all the clusters.

In the model for our ACC case study, there is one component in each cluster. Therefore, the execution time of each cluster is equivalent to the execution time of the corresponding components. The WCET for ACC model is the sum of the execution times for all three clusters. When analysed with Platin, this will give 80871 cycles. Executed at 50 MHz each cycle is 20 ns, so the total WCET is $80871 * 20ns = 1.62ms$.

VI. Results

Benchmark		CpNN Contents	#Connections	SANN WCET (ms) (from [16])	CpNN WCET (ms) (our result)
A	XOR	MLP	9	0.82	0.10
B	Adder	MLP	15	0.49	0.10
C	AIBRO	MLP * 3	18, 18, 28	2.80	2.24
D	ESS	MLP * 3	110, 55, 620	14.00	4.36
E	RABBIT WOLF	MLP * 3	576, 840, 840	Unavailable	10.98
Motivating example:					
F	ACC	MLP * 2, Customer Function	40, 40, N/A	N/A	1.62
Average WCET Reduction Percentage					64.06%

TABLE II
Comparing CpNN2C to [16]

To demonstrate the benefits of our approach compared to the synchronous NN approach in [16], we reimplemented their suite of open-source benchmarks (sourced from [12]) using our tool CpNN2C. As [16] used the open-sourced data from the TCREST project, we were able to both reproduce their results and compare with the proposed approach by using the same FPGA platform and the same clock frequency for a direct comparison. All NNs were trained offline (i.e. pre-trained) using the Keras NN library [5].

Our comparison results are presented in Table II. As can be seen, we outperform the results of [16], with a 64.06% average WCET reduction. The main reason is due to the overhead induced by the Esterel compilerrelative to this simple synchronous C code we generate.

Benchmark	#Connections	Tensorflow Lite (ms)	CpNN (ms)
Xor	9	1.193	0.05
Adder	15	1.3179779	0.087
Aibro	64	5.3436756	0.068
Acc	80	2.5730133	0.08
Rabbit	576	3.6201477	0.139
Ess	785	1.99437	0.14
Wolf	840	3.15499	0.163
Benchmark-a	2110	0.352859	0.312
Benchmark-b	5500	2.3689	0.551
Benchmark-c	20300	3.18574905	1.009
Benchmark-d	30350	0.66375732	1.61
Average Measured WCET Reduction Percentage			62.08%

TABLE III
Comparing CpNN2C to Tensorflow Lite

A. Comparison with Tensorflow Lite

In this section, we compare CpNN2C with Tensorflow Lite [1], a popular library for embedded system using NNs. In addition to the synchronous NNs in [16], the WCETs analysis of four synthetic benchmarks are included for testing the performance of CpNN with larger NN connection ranges. The synthetic benchmarks all receive one input and produce one output. They contain two hidden layers, with softmax and linear activation functions respectively. All the NNs were pre-trained in Keras, which can be read directly for CpNN.

It is worth noting that Tensorflow Lite provides a converter to translate the NNs in Keras to a Tensorflow Lite format. As it does not support a compositional design, the synchronous NNs from [16] were constructed manually in *Python*. In this way we ensured that equivalent NNs were implemented in both frameworks.

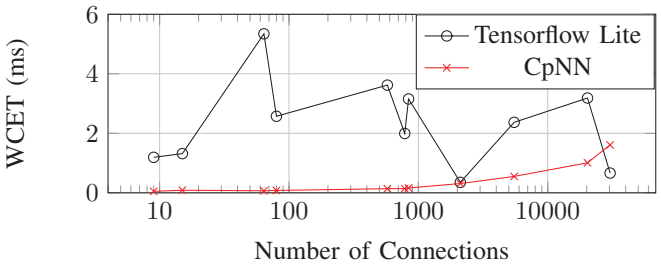
In the previous comparison with [16], the WCET analysis was done by Platin. However, unlike our tool, Tensorflow Lite does not produce code amenable to static timing analysis. This is mainly because Tensorflow Lite executes networks within an interpreter written in C++.

Hence, we use *measurement based* timing analysis methodology for comparison. Here, we execute both our code and the Tensorflow Lite code on a Raspberry Pi 3 Model B+. The NNs are executed for one million *ticks* with random input traces, and the iteration with the longest execution time is recorded as the measured WCET for that NN. To ensure fairness, the measured WCET measurements of Tensorflow Lite's benchmarks recorded only the execution time of the internal interpreter - we did not include the initialisation time for the system (our methodology profiled the execution time of *Interpreter.invoke()*).

The results for this comparison are shown in Table III and Figure 4. As we can see, CpNN outperformed Tensorflow Lite in most benchmarks, with a 62.08% average WCET reduction as well as supporting large scale NNs such as RABBIT WOLF. Additionally, the measured WCETs of the benchmarks from CpNN increase steadily as the connections increase. However, the measured executions of Tensorflow Lite do not have a clear trend, and show very unpredictable and inconsistent measured WCET times. This is likely because Tensorflow Lite

is optimised to improve the average case [1].

Fig. 4. CpNN Compiler vs Tensorflow Lite for measured WCET



B. ACC Case Study Discussion

It is important to know the WCET of the ACC case study, as a *safety-critical systems*. Here, any functional *or timing* error could cause catastrophic consequences. While many approaches for ANNs can provide analysis of functional correctness, few examine timing properties. However, using our CpNN2C compiler with the Platin timing analyser for Patmos architectures we can guarantee the timing bounds of the dual-ANN ACC cruise controller. More precisely, the ACC model ensures that appropriate acceleration and velocity are evaluated within $1.62ms$ on the 50 MHz Patmos core.

VII. Conclusions

Neural networks are increasingly used to achieve control tasks in CPSs. Such controllers must meet stringent functional and timing constraints. However, there is minimal work on the timing validation of neural network controllers used in CPS. One proposed approach is to use Esterel [16], but Esterel is a language with minimal support for the design of neural networks. Instead, in this paper, we adopt the well known Keras based neural network library. Thus, in this paper we present CpNNs and our compiler CpNN2C, which together provide a pathway for the synchronous design of composed neural networks for real-time systems. Our approach has a compositional semantics which enables modular compilation to deterministic C code, which is amenable to static timing analysis for determining WCET. In our benchmarks, CpNNs have an average WCET reduction of 64.06% compared to the Esterel approach in [16], and an average measured WCET reduction of 62.08% compares to Tensorflow Lite. Our framework currently supports MLP. Future work will examine other type of NNs, such as CNN and RNN.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 265–283.

[2] Alur, R. Principles of cyber-physical systems, 2015.

[3] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Le Guernic, P., and De Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE 91*, 1 (2003), 64–83.

[4] Berry, G. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner* (Cambridge, MA, USA, 2000), G. Plotkin, C. Stirling, and M. Tofte, Eds., MIT Press, pp. 425–454.

[5] Chollet, F., et al. Keras, 2015.

[6] Elman, J. L. Finding structure in time. *Cognitive science 14*, 2 (1990), 179–211.

[7] Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. Ai2: Safety and robustness certification of neural networks with abstract interpretation. 3–18.

[8] Hepp, S., Huber, B., Knoop, J., Prokesch, D., and Puschner, P. P. The platin tool kit-the t-crest approach for compiler and wcet integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS* (2015), pp. 5–7.

[9] Hergenhan, A., and Rosenstiel, W. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)* (March 2000), pp. 552–559.

[10] Inc, A. SCADE. https://www.ansys.com/products/embedded-software. Accessed: 2019-09-14.

[11] Parnas, D. L. Use of abstract interfaces in the development of software for embedded computer systems. Tech. rep., NAVAL RESEARCH LAB WASHINGTON DC, 1977.

[12] PRETgroup. sann. https://github.com/PRETgroup/sann.

[13] Ranjitkar, P., Nakatsuji, T., and Kawamua, A. Car-following models: an experiment based benchmarking. *Journal of the Eastern Asia Society for Transportation Studies 6* (2005), 1582–1596.

[14] Redmon, J. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[15] Ronacher, A. Jinja2 documentation. *Welcome to Jinja2-Jinja2 Documentation (2.8-dev)* (2008).

[16] Roop, P. S., Pearce, H., and Monadjem, K. Synchronous neural networks for cyber-physical systems. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)* (2018), IEEE, pp. 1–10.

[17] Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C. W., Karlsson, S., and Thorn, T. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems* (2011), OASICS.

[18] Seshia, S. A., Sadigh, D., and Sastry, S. S. Towards verified artificial intelligence. *arXiv preprint arXiv:1606.08514* (2016).

[19] Simonelli, F., Bifulco, G. N., De Martinis, V., and Punzo, V. Human-like adaptive cruise control systems through a learning machine approach. In *Applications of Soft Computing*. Springer, 2009, pp. 240–249.

[20] Stallman, R. M., and DeveloperCommunity, G. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[21] Tripakis, S. Compositionality in the science of system design. *Proceedings of the IEEE 104*, 5 (2016), 960–972.

[22] Tripakis, S. Data-driven and model-based design. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)* (May 2018), pp. 103–108.

[23] Xiao, L., and Gao, F. Practical string stability of platoon of adaptive cruise control vehicles. *IEEE Transactions on intelligent transportation systems 12*, 4 (2011), 1184–1194.