HAL
open science

Formal description of ML models for unambiguous implementation

Adrien Gauffriau, Iryna De Albuquerque Silva, Claire Pagetti

▶ To cite this version:

Adrien Gauffriau, Iryna De Albuquerque Silva, Claire Pagetti. Formal description of ML models for unambiguous implementation. 12th European Congress on Embedded Real Time Software and Systems (ERTS 2024), Jun 2024, Toulouse, France. hal-04167435v2

HAL Id: hal-04167435
https://hal.science/hal-04167435v2

Submitted on 17 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal description of ML models for unambiguous implementation

Adrien Gauffriau *Airbus, France* Iryna De Albuquerque Silva *ONERA, France* Claire Pagetti *ONERA, France*

I. Introduction

Machine learning (ML) applications have been gaining considerable attention in the field of transportation. However, their use in real-life operational safety-critical products, in particular in the aeronautical domain subject to stringent certification, raises several issues regarding functional correctness, compliance with requirements, formal verification, safety or implementation. In order to tackle those issues, new guidelines – named ED 324/ ARP 6983 standard [EUR21] – are currently drafted by the EUROCAE WG-114/SAE G-34 joint working group that cover the whole spectrum of the system development including the data sets composition, the ML model design and its implementation. In this paper, we focus only on the implementation of the ML model.

A. Context

In the ML current practices, a *training framework* is used to design an ML model and the resulting ML model is then deployed on the target with a *deployment framework*. It is up to the training framework or a designer to export the trained model description in an exchange format and up to the deployment framework to import the ML model description. The left part of figure 1 shows those practices. The deployment framework is most of the time an ML model interpreter, that can accommodate any type of ML model architecture, and that allocates at runtime the execution on the different available accelerators (e.g. GPU, FPGA) of the target. These ML frameworks have been designed 1) to ease as much as possible the deployment of models for the users and 2) to optimize as much as possible the execution performance (usually expressed in trillion operations per second – TeraOp/s). As a result, they are very impressive and allow for complex deployments and optimizations. Those are hard, if not impossible, to reproduce for a programmer without using ML libraries (e.g. CUDNN on NVIDIA).

The counterpart of such an approach is 1) the absence or limited information of internal computation and allocation; 2) small modifications and adaptations of the ML model (e.g. when exporting the ML model description or when quantizing on the fly some matrix multiplication to execute on deep learning accelerator – DLA). If this grey/black box approach is acceptable and suitable for mainstream applications, it is a blocking point for highly safety-critical applications. As a result, the main objective of the ARP 6983 standard with respect to the implementation process is the *semantics preservation* of the off-line trained model on the final hardware platform. This means that the execution of the ML model in the training framework should be exactly reproduced on the embedded target during execution. To reach that objective, an alternative development process is proposed as illustrated in the right hand part of figure 1.

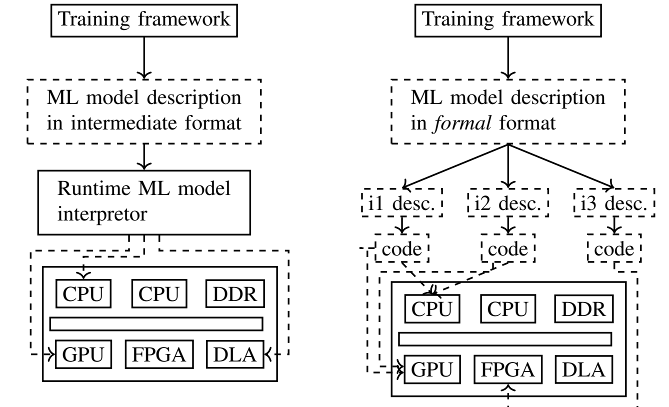


Figure 1. Left part: current ML deployment practice and right part: proposed aeronautical practice.

B. Assurance development process

The principle of this trustable development process (figure 1) is to ensure the semantic preservation at each step of the development cycle (e.g. between training framework and description, between description and code, between code and executable). This is guaranteed thanks to a series of requirements provided in the ARP 6983.

Requirement 1. First, the trained model must be formally and unambiguously defined in what we call an adequate *format*. Such a *format* must come with a formal syntax and semantic, and should be agnostic of any (training and/or deployment) framework.

Requirement 2. Second, the implementation process should allow several deployments on hardware platforms and it must be known beforehand how the ML model will be mapped on the accelerators and when. This entails in particular that the *format* should allow several types of deployment such as distribution, parallelization or pipelining. Thus, there is a step that consists in splitting the ML model description (in the chosen format) as a series of *item* descriptions. Indeed, in the avionics domain, a target processor is decomposed as a set of software (SW) and/or hardware (HW) *items*. Let us consider for instance an ULTRASCALE+ (ZCU102) platform [Xil19]: it is composed of several ARM cores, a GPU and an FPGA. If the ML model is spread over the different components, in

particular on one ARM and the FPGA, there will necessarily be several items. Indeed, the ARM associated code will be considered as one SW item and will go through the ED-12C/DO-178C [RTC11] development process. Whereas the hardware design of the FPGA will be considered as another item (HW) and will go through the ED-80/DO-254 [RTC00] development process.

Requirement 3. Third, the implementation has to follow the usual aeronautical development standards (e.g. ED-12C/DO-178C [RTC11] for software). Thus, the description of a model within the format must be compliant with a certified implementation process. This last objective concerns the capability to implement an item description following standards such as ED-12C/DO-178C [RTC11] or ED-80/DO-254 [RTC00]. Among the requirements from those standards, two are related to the format. First, it must offer full traceability: looking at the generated code (e.g. C or CUDA), it is humanly possible to trace back to the original exported model. Second, the execution must be *predictable*, meaning that it is possible to assess a WCET (Worst Case Execution Time) [WEE+08]. This entails that the code is expected to be allocated statically on the resources, all the memory allocations are static and the schedule (here the sequence of operations) is also static.

C. Contributions

Our general objective is to define an approach compatible with the ARP 6983 requirements presented in the prior section. We focus on a representative subset of deep neural networks (DNNs) that is feed-forward neural networks trained off-line. Our main goal is the definition of an adequate format, with a formal syntax and semantics, able to describe both 1) a global ML model, and 2) any parallelized allocation on several items, the behaviour of which is equivalent to the global model.

For requirement 1. There are several initiatives to propose an intermediate format between trained models and their implementation such as ONNX [BLZ+19] (Open Neural Network Exchange). After a thorough evaluation of different existing formats, we identified NNEF (Neural Network Exchange Format) [The22] as the most suited for our purpose: syntax and semantics are public and moreover the authors made a strong effort to provide a formal specification.

Since NNEF provided a potential candidate, we decided to construct our format on top of it. As it is now, NNEF describes formally the global ML model. Indeed, the semantics of NNEF is almost fully defined (see section II). What is however missing in NNEF is the clear formalization of the *execution model*, that is the formal behaviour behind a series of NNEF *instructions*. To fix this missing element, we rely on Petri nets, a usual representation of program behaviours [Pet77]. This choice is also consistent with the need to express distributed behaviour on items, as Petri nets also allow to model all combinations of execution: sequence, pipeline, recursion or parallel [Old86].

For requirement 2. We illustrate in section III why decomposing ML model into items is of interest. To that end, we extend the syntax and semantics of NNEF. We rely in particular on *logical data exchange* among items to distribute the computation and express the semantics with *coloured Petri nets*. We formally show that the synchronization of the Petri nets behaves as the *original* non distributed Petri net.

For requirement 3. We do not propose a complete DO-178C compatible approach but instead show a reasonable implementation approach on the XAVIER platform [NVI19a] that we believe could be with some effort compliant with the ED-12C/DO-178C [RTC11].

II. FORMAT OF DESCRIPTION – NNEF

KHRONOS standardization group[1] has defined the NNEF (Neural Network Exchange format) format with a specification that provides a syntax and a semantic. We focus on the NNEF syntax and semantics elements needed for our purpose. The reader can refer to [The22] for a complete description of NNEF.

A. Brief Reminder on Neural Network

The field of artificial intelligence has gained much research attention in the past years. The power of AI resides in the capacity of solving highly complex problems [GBC16]. Machine learning domain describes the study and development of statistical algorithms that are able to efficiently generalize on unknown input data after the extraction of patterns from a similar, and representative, data set. Neural networks are a class of ML algorithms. A neural network implements a mathematical function $F_{\mathcal{N}}$ that aims at approximating a continuous real-valued function [HSW89], [SZ06]. $F_{\mathcal{N}}$ is composed of different mathematical functions called *layers*.

There are two types of deep neural networks: feed-forward neural networks and recurrent neural networks. Recurrent neural networks (RNN) feature layers that take as input some of their output (or the output of a successor layer), thus creating cycles. In feed-forward variants it is not true. We are not addressing RNN in this paper. A common representation of a feed-forward deep neural network (FDNN) is in the form of a *directed acyclic graph* (DAG) defining how its layers are connected together.

***Definition** 1 (Feed-forward Deep Neural Network): A* feed-forward deep neural network $\mathcal{N} = (V, E)$ *is a directed acyclic graph, wherein:*

- V *is the finite set of vertices of the graph, which represent its layers* $l \in V$*;*
- $E \subseteq V \times V$ *is the set of edges of the graph, representing the data flow within the neural network.*

In order to construct the possible flows of data within the feed-forward deep neural network, it is necessary to define what are the predecessors and successors of a vertex, or layer.

***Definition** 2 (Predecessors / successors of a layer): The direct predecessors (resp. successors) of a layer* l *are defined as the layers of the set* $Pre(l) = \{l' \in V \mid (l', l) \in E\}$ *(resp.* $Succ(l) = \{l' \in V \mid (l, l') \in E\}$*). The predecessor transitive closure of* l *is the set of all its predecessors layers noted* $\mathrm{Pre}^*(l) = \bigcup_{n=1}^{k} \mathrm{Pre}^n(l)$*, wherein:*

$$\mathrm{Pre}^n(l) = \begin{cases} \mathrm{Pre}(l), & \textit{if } n = 1 \\ \bigcup_{l' \in \mathrm{Pre}^{n-1}(l)} (\mathrm{Pre}(l') \cup \{l'\}), & \textit{otherwise} \end{cases}$$

[1]https://www.khronos.org/

A layer can be classified into input, output and intermediate, or hidden. An input layer l only consumes input data, i.e., $Pre(l) = \emptyset$. Similarly, a final layer l only produces output data, i.e., $Succ(l) = \emptyset$. We represent $V_I \subseteq V$ as the set of input layers and $V_O \subseteq V$ as the set of output layers. Note that $V_I \bigcap V_O = \emptyset$. The remaining layers, $l \notin V_I \bigcup V_O$, are the hidden layers. Finally, as a straight consequence of directed acyclic graphs properties, $l \notin Pre^*(l)$.

The function performed by a layer is of the form $f_l : \mathbb{R}^m \rightarrow \mathbb{R}^n$, wherein m and n represent respectively the input and output dimensions of the given layer's function.

***Definition** 3 (Function associated to a FDNN): Let $\mathcal{N} = (V, E)$ be a feed-forward deep neural network and $V_O \subseteq V$ be the set of output layers. Let us denote the function associated to a set of layers such that:*

$$\forall U \subseteq V, F_U = \begin{cases} (F_{l_1}, \dots, F_{l_n}), & \textit{if } U = \{l_1, ..., l_n\} \\ F_\emptyset, & \textit{if } U = \emptyset. \end{cases} \quad (1)$$

wherein:

$$\forall l \in V, \quad F_l = f_l(F_{Pre(l)}) \quad (2)$$

We note $F_\mathcal{N} = F_{V_O}$ the function associated to a feed-forward deep neural network.

***Example** 1 (*Single-path *feed-forward deep neural network): It corresponds to the particular case of a feed-forward deep neural network, wherein: $V = \{l_1, \dots, l_n\}$ and $\forall i \geq 2$,* $\text{Pre}(l_i) = \{l_{i-1}\}$. *Therefore, $V_I = \{l_1\}$, $V_F = \{l_n\}$. Such an example is the* LENET-5 *shown in figure 2.*

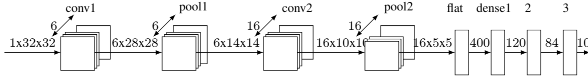


Figure 2. LENET-5 neural network

According to Definition 3 the function associated to a single-path DNN is the composition function: $F_\mathcal{N}(x) = \circ f_{l_{n-1}} \circ \dots \circ f_{l_1}(x)$ wherein m_{l_1} in the input dimension of f_{l_1} and p_{l_n} is the output dimension of f_{l_n}.

B. Neural Network Description in NNEF

Our first goal concerns the definition of a format, with a formal syntax and semantics, able to describe any ML model such as the LENET-5 of example 1. Let us explain why NNEF fulfils this goal. An NNEF description is composed of two parts: 1) A computation graph described in a human readable text file; and 2) the parameters provided in multiple raw data file. The fact that the description is textual is important for the traceability between the specification (output of the training framework) and the embedded code.

The computation graph file describes all parameters needed and operations to be done. More precisely, each line of the computation graph is an elementary instruction (NNEF *compound fragment*) that may be split into several atomic operations (NNEF *primitives*). The result of each instruction is stored in a named variable, that represents a tensor, and which can be used as input for other instruction(s). To compute an operation all its inputs variables shall be available.

***Remark** 1:* NNEF description follows a SSA (static single assignment) form [CFR+89] which helps the implementation process. It is usual to translate a program in its SSA form before compilation or optimization passes [BBD+17].

***Example** 2: Let us illustrate how to specify a neural network with an example. The* NNEF *textual specification of the* LENET-5 *detailed in example 1 is given in the listing 1. First, all parameters are declared and stored as variables.* e1 *is the input tensor of size $1 \times 32 \times 32$,* v1 *contains the 6 kernels of size $1 \times 5 \times 5$ of the first convolution and* v2 *is the bias applied at the end of the first convolution. The parameters needed by the layers should all be defined as variables in the description file.*

```
graph torch_jit_export(e1) -> (out)
{
    e1 = external<scalar>(shape = [1, 1, 32, 32]);
    v1 = variable<scalar>(shape = [6, 1, 5, 5],
        label = 'v1');
    v2 = variable<scalar>(shape = [1, 6], label = '
        v2');
    v3 = variable<scalar>(shape = [16, 6, 5, 5],
        label = 'v3');
    v4 = ...; v5 = ...; v6 = ...; v7 = ...;
    v8 = ...; v9 = ...; v10 = ...;

    o1 = conv(e1, v1, v2, stride = [1, 1], dilation
        = [1, 1], padding = [(0, 0), (0, 0)], groups
         = 1);
    o2 = relu(o1);
    o3 = max_pool(o2, size = [1, 1, 2, 2], stride =
        [1, 1, 2, 2], dilation = [1, 1, 1, 1],
        padding = [(0, 0), (0, 0), (0, 0), (0, 0)],
        border = 'ignore');
    o4 = conv(o3, v3, v4, ...;   o5 = relu(o4);
    o6 = max_pool(o5, ...;  o7 = reshape(o6, shape =
          [0, -1]);
    o8 = linear(o7, v5, v6);  o9 = relu(o8);
    o10 = linear(o9, v7, v8);  o11 = relu(o10);
    o12 = linear(o11, v9, v10);
    out = softmax(o12, axes = [1]);
}
```

Listing 1. LENET-5 with NNEF syntax

After the variables declaration, comes the computation graph itself. The output of the first convolution is stored in the variable o1. *When calling the function / compound fragment* conv, *the user must instantiate the full set of parameters for this type of layer: input tensor, kernels, bias, stride, dilation, padding and groups. Every parameter appears explicitly in the definition and there is no ambiguity. For instance, the way to declare the padding explicitly states how the padding applies on top / bottom / right / left. After the convolution, the activation function has to be explicitly applied to* o1 *and thus is not hidden in the convolution, ensuring again an unambiguous description. The first pooling layer results in variable* o3. *Reading the description, we recognize the* LENET-5 *detailed before. The flat layer is encoded with a more expressive function* reshape *that allows several reshaping. The* dense *layer is called* linear. *The post-processing* softmax *is also explicit.*

The NNEF specification also provides the link between instructions (e.g. *conv*) appearing in the file and their associated mathematical functions. Subsequently, we will not use NNEF terms, because the NNEF standard uses the generic term *fragment* for both. We illustrate this with the max pooling

layer only.

C. Max Pooling Layer Semantics

Let us illustrate issues that may arise without a formal description. Let us first remind the functional semantics of a max pooling layer where a padding (and no dilatation) is applied. Thus, the function is defined by $\mathcal{P}ool_{k,s} \circ \mathcal{P}_{p,v}$ where each function is defined below.

***Definition** 4 (Padding associated function – $\mathcal{P}_{p,v}$): Let $p = (p_t, p_b, p_l, p_r)$ be a 4-tuple of integers representing the padding to be applied on each border of a 3D-tensor and v the float value to be used for the padding. The padding function $\mathcal{P}_{p,v}$ applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{P}_{p,v}(I)$ of size (o_h, o_w, o_c) with $o_h = n_h + p_t + p_b$, $o_w = n_w + p_l + p_r$ and $o_c = n_c$ such that*

$$O_{x,y,z} = \begin{cases} v & \textbf{\textit{if}}\ (x \leq p_t) \text{ or } (x > n_h + p_t) \text{ or} \\ & \quad (y \leq p_l) \text{ or } (y > n_w + p_l) \\ I_{x-p_t, y-p_l, z} & \textbf{\textit{otherwise}} \end{cases}$$

***Definition** 5 (Pooling layer associated function – $\mathcal{P}\text{ool}_{k,s}$):* Let $s = (s_h, s_w)$ be the *stride* parameters and $k = (k_h, k_w)$ be the height and width of the *window*. The pooling applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{P}ool_{k,s}(I)$ of size (o_h, o_w, o_c) with $o_h = \left\lfloor \frac{n_h - k_h}{s_h} + 1 \right\rfloor$, $o_w = \left\lfloor \frac{n_w - k_w}{s_w} + 1 \right\rfloor$ and $o_c = n_c$ with $O_{x,y,z} = \mathit{max}(I[s_h \cdot (x-1)+1 : s_h \cdot (x-1)+k_h+1][s_w \cdot (y-1)+1 : s_w \cdot (y-1)+k_w+1][z])$. Here, $I[s_{11} : s_{21}, ..., s_{1k} : s_{2k}]$ represents the *slice* of I of all the values $I_{s_{11}+x_1, \ldots, s_{1k}+x_k}$ with $i \in [1, k]$ and $x_i \in [1, s_{2i} - s_{1i}]$.

The NNEF syntax of a max pooling layer describes the padding values with an enumerate string. Ignoring the border for a max pooling layer is equivalent to pad with the minimum float value (*MIN_F*). This corresponds to the neutral operator of the max function. Looking now at the *max_pool* elementary instruction according to NNEF documentation [The22], it is defined in the pseudo-code by 2 atomic operations. 1) *argmax_-pool* that computes an array of index (corresponding to the max in each pool); 2) *sample* that returns for an array of index, an array with corresponding values. This pseudo-code indeed encodes the expected function $\mathcal{P}ool_{k,s} \circ \mathcal{P}_{p,v}$.

Of the importance of unambiguous description. We propose to highlight the importance of making unambiguous textual descriptions of ML models by comparing two state-of-the-art training frameworks, namely PyTorch and Keras (with TensorFlow).

- In Keras, a padding inside a max pooling layer can only be declared by a string ∈ {"valid", "same"}. "valid" means no elements to add, while "same" means that padding is added on the right and on the bottom borders only to fit the size of the pool.

```
# KERAS SYNTAX for pool1
    MaxPool = tf.keras.layers.MaxPooling2D(
        pool_size=(2,2),strides=(2,2),padding
        = 'same')

# NNEF for KERAS SYNTAX
    max_pool_keras = max_pool(input, size =
        [1, 1, 2, 2], stride = [1, 1, 2, 2],
        dilation = [1, 1, 1, 1], padding =
        [(0, 0), (0, 0), (0, 1), (0, 1)],
        border = 'ignore');
```

This corresponds to $\mathcal{P}ool_{[2,2],[2,2]} \circ \mathcal{P}_{[0,1,0,1],\text{MIN_F}}$.

- In PyTorch, a padding inside a max pooling layer can only be declared by one or two integers. In case of one integer, this defines the number of elements to add to each border (top, bottom, left and right). In presence of 2 integers, the first gives the number of elements to add at the top and bottom, and the second at the left and right.

```
#PYTORCH SYNTAX for pool1
    MaxPool = NN.MaxPool2d(2,           2,
        1)
                        # Kernel Size,  Stride,
                            Padding

#NNEF for PYTORCH SYNTAX
    max_pool_torch = max_pool(input, size = [1,
        1, 2, 2], stride = [1, 1, 2, 2],
        dilation = [1, 1, 1, 1], padding = [(0,
        0), (0, 0), (1, 1), (1, 1)], border =
        'ignore');
```

This corresponds to $\mathcal{P}ool_{[2,2],[2,2]} \circ \mathcal{P}_{[1,1,1,1],\text{MIN_F}}$

The semantics of Keras and PyTorch are not equivalent, and there is no possibility to convert one into another at once. The only way to make a valid conversion is to explicitly add a padding layer before the max pooling.

D. NNEF *Execution Model Semantics*

The semantics of the *execution model*, that is the formal behaviour behind a series of NNEF *instructions*, is not explicitly given by the standard. It assumes that one instruction can be executed when all its inputs are computed and available. Thus, executing the instructions in sequence following the order of the NNEF guarantees a correct execution. There are two types of instructions: those reading parameters from binary files or input tensors and layer-associated instructions based on one or several atomic operations. An instruction is always of the form

$$\mathit{var} = \mathit{operation}(v_1, \ldots, v_k, \mathit{cst}_1, \ldots, \mathit{cst}_j);$$

where *var* is a variable computed by the *operation* (any NNEF *fragment*), v_i are either variables computed beforehand, the input tensor or fixed parameters (e.g. kernels), and cst_i are constant (e.g. stride). The NNEF execution model can be formally expressed using the Petri net formalism [Pet77].

***Translation** 1:* A NNEF description, composed of n instructions, generates a Petri net (P, T, V) where:

- the set of places P corresponds to all variables appearing in the NNEF description (i.e. n places for a description of n instructions);
  - a token in a place means that the associated variable is available for computation;
  - initially there are as many tokens in each parameter-associated place as the parameter is needed in the instructions and as many tokens in the input tensor place as the input tensor is used by the instructions;
  - there is a unique final place corresponding to the last variable computed in the NNEF file;

- the set of transition names T corresponds to all instructions appearing in the NNEF description;
- $V \subseteq 2^P \times T \times \mathbb{N} \times P$ defines the set of transitions. A transition can be fired if there is a token in all input places. When it fires, the transition removes a token from each of these places and generates as many tokens as defined on the edge in the unique output place.
  - each instruction $var = op(v_1, \ldots, v_k, cst_1, \ldots, cst_j)$ generates the transition given in figure 3 where p is the number of time *var* will be consumed by other instructions;
  - when only one token is produced by a transition, we omit the integer.

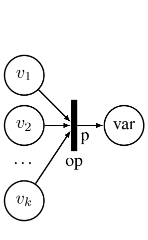


Figure 3. Translation of one instruction

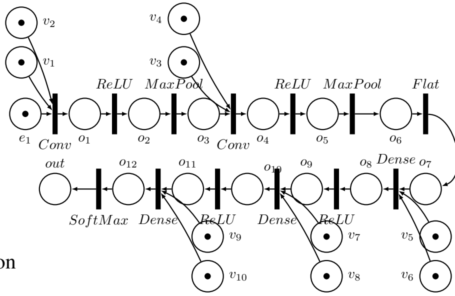


Figure 4. NNEF semantics of the LENET-5 express with Petri net. Initial marking

The semantic of the Petri net clarifies the execution order that is unclear in the NNEF formalism. Especially, the order of the textual file should not impose a unique order when several valid ones may exist.

***Example** 3: The* LENET-5 *model of figure 2 with its associated* NNEF *description in listing 1 has the associated Petri net given in figure 4. We recognize the instructions sequence that describes the computation of the neural network graph. There is a unique possible schedule for this* NNEF *description, but we will see later other* NNEF *models that accept several schedules (see section III).*

***Remark** 2:* The Petri net of figure 4 only defines the semantics of a single inference pass. It is usual to repeat the inference pass in order to process new inputs (e.g. in a periodic manner). This can also be represented using the Petri formalism by sending back tokens to $e1$ and v_k places. For demonstration and clarity, subsequently in the paper, we always consider the semantics of a single inference.

A way to define the semantics of Petri nets is to compute the set of reachable *markings*, where a marking defines the number of tokens in each place at a given instant.

***Definition** 6 (Marking): Considering a Petri net with n places, a marking is defined as a vector $v \in \mathbb{N}^n$ giving the number of tokens $v[i]$ in the i-th place. This initial location of tokens is called the* initial marking*, this represents the starting state of the system. A* final marking *is a marking such that there is one token only in each final place and from which no transition can be fired any more.*

***Example** 4: In the Petri net of figure 4, there are 24 states. There are 11 tokens in the initial making and there is a single token in the unique final marking (in the place* out*).*

***Property** 1 (Initial marking and unique final marking):* The unique initial marking is defined by the translation and consists of token(s) in the input tensor variable place and parameters-associated places. Because we consider feed-forward neural networks, there is unique *final marking*.

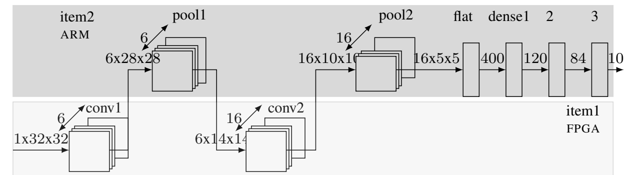
***Definition** 7 (Paths and semantics): A* valid path *starts from the initial marking m_i, lists a series of fireable transitions and ends in a final marking m_f, i.e. $m_i \longrightarrow^{t_1} m_1 \longrightarrow^{t_2} \ldots \longrightarrow^{t_l} m_f$. The semantics of a Petri net is the set of valid paths.*

***Property** 2 (Possible executions of an* NNEF *description):* Because we consider feed forward neural networks, the number of valid paths is finite and the valid paths correspond to all possible execution orders respecting the semantics of the ML model.
Semantics preserving code generation could lead to any implementation the path of which is valid. Full sequential code following the order of instructions of the NNEF file is one of them.

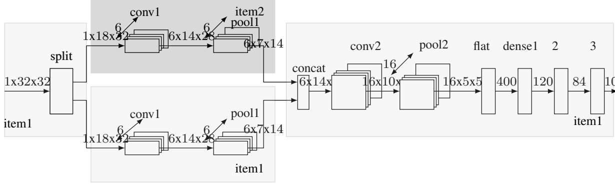
III. IS DISTRIBUTION NEEDED?

Because we consider highly distributed platforms, a designer may choose to split the ML model into several parts in order to accelerate the execution and reduce the execution time. In particular, it could lead to developing parts independently and on different *items* following the aeronautics standards [SAE10]. In such a case, there should be a formal description for each item that becomes the input specification for HW/SW item implementation. We identified 3 different needs for distribution to be addressed that we illustrate on the LENET-5 example.

***Remark** 3:* Note that if the designer considers its platform as a unique item, the NNEF description will be the specification.
Off-loading computation. Let us consider for instance the implementation of the LENET-5 on an ULTRASCALE+ (ZCU102) platform [Xil19] composed of several ARM cores, a GPU and an FPGA. Let us assume that we choose to execute the convolution on the FPGA and all other layers on one ARM. The idea will be to offload the input tensor of each convolution on the FPGA and retrieve the feature maps from the FPGA (see figure below).

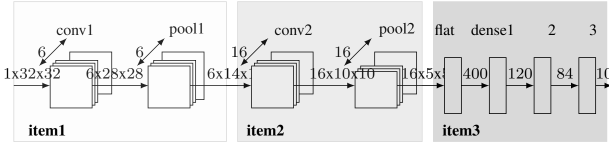


Parallelizing the layers. A second type of distribution could be to refine the layers and exhibit more parallelism by distributing the computation of a layer across several items. It will be up to the designer to show the semantic preservation at this refinement level. Looking again at the LENET-5, we can split the computation of the first two layers on two different items. Each item will do the convolution+maxpool on a part of the input image. To do so, the input image is split along the height and two sub-parts are executed on two different



items (see figure below). In order to keep the semantics for the convolution, some overlap exists between the two sub-images.

Pipelining the computation. A third type of distribution is the pipelining of the DNN execution. In this case, each item is in charge of computing a specific layer (or a group of layers). The first item computes the first layer(s) on the first tensor input and sends its output to the second item that will compute the second group of layer(s) while the first item starts processing the second input tensor. This classical mechanism enables to reduce (e.g. for video processing) the frame rate while degrading the latency. The depth of the pipeline is the number of inputs that can be handled at the same time among the pipeline.



IV. NNEF EXTENSION FOR MULTIPLE ITEMS

The purpose of this section is to propose a manner to separate the specification of each item so that any execution of the items respecting the specification properly encode the global ML model. To that extent, we propose first to extend the syntax of NNEF to allow explicit parallelization and then to express the associated semantics with colored Petri nets.

A. *Extension for item splitting*

We first need to specify the item on which the description will be implemented. To that end, we enrich the graph definition with the keyword *graphitem* to provide the logical id of the HW or SW item.

Syntax1 GraphItem

```
<graph-definition> ::= <graph-declaration>
  <graph-declaration-item> <body>

<graph-declaration-item> ::= "graphitem"
  <identifier> <identifier>"("<identifier-list>")"
  "-$>$" "("<identifier-list>")"
```

***Semantics** 1:* The first <*identifier*> refers to the item id, the second <*identifier*> is the name of the local node and elements of the <*identifier-list*> will be input or output of the graphitem. The semantic of the <*graph-declaration-item*> is such that all instructions within the <*body*> are executed by the item.

We also need to *exchange data* between several items and ensure that those exchanged data are available before computation. To that end, we introduce a new type of variable, namely *variablesync*. This references a variable that could be read from or write to another *graphitem*. We use a *fragment* to define this new type.

Syntax2 VariableSync

```
fragment variablesync<? = scalar>
    (shape: integer[])  -> ( output: tensor<?> )
```

***Semantics** 2:* Each *variablesync* is a shared variable with a unique writer and possibly multiple readers. Writer is in charge to transmit the variable via the instruction *send_var* and each reader can access this data via *get_var* instruction.

We then define new NNEF instructions to send or get data between several *graphitem*.

Syntax3 get_var

```
fragment get_var<? = scalar>(source : graphitem,
    data  : variablesync) -> ( output: tensor<?> )
```

***Semantics** 3: get_var* appears in each reader description. The output of this instruction is a local variable that gets the content of the shared variable and which is available for the caller item. *Source* is the item that writes and provides the shared variable the name of which is given by *data*.

Similarly, the writer must define the instruction to send a shared variable to other items.

Syntax4 send_var

```
fragment send_var<? = scalar>
    (  dest   : graphitem[], data   : scalar)
    -> ( output: variablesync )
```

***Semantics** 4: send_var* appears in the writer description only. It takes as input the list of reader items and the name of the tensor that shall be sent. The output tensor is a global *variablesync* that will support the *synchronization*.

The rest of the NNEF syntax remains unchanged.

B. *Splitting an* NNEF *description into multi-item descriptions*

Initially, the DNN is described in a unique NNEF description as shown in section II. Such a description contains 3 types of instructions:

- definition of input tensor(s);
- definition of fixed parameters;
- variables computed by each layer.

A splitting consists in partitioning the last type of instructions among the items, adding the adequate definition(s) of tensors / fixed parameters and adding the adequate *send_var* / *get_-var*. The composition of item descriptions shall respect the semantics of the full NNEF description.

***Example** 5: Let us consider the DNN of listing 2 with its associated Petri net (figure 5). Let us assume that the DNN is allocated on 3 items such that o_1, o_6, o_7 and out are computed on item 1; o_2, o_3 are computed on item 2 and o_4, o_5 are computed on item 3. Thus, the description on the items is given in Listing 3.*

The union of the instructions of each item corresponds to the complete NNEF *description with the additional* variablesync *and the communication instructions. Locally in the item, the*

```
graph DNN(e1) -> (out)
{
   e1 = external..
   v1 = variable... 'variable1'); v2 = ...; v3 =
       ...; v4 = ...;
   v5 = ...; v6 = ...; v7 = ...; v8 = ...; v9 = ...;
        v10 = ...;
   o1 = conv(e1,v1,v2,... ;
   o2 = conv(o1,v3,v4,...; o3 = conv(o2,v5,v6,...;
   o4 = conv(o1,v7,v8,...; o5 = conv(o4,v5,v6,...;
   o6 = concat (o3, o5); o7 = flatten(o6,...;
   out = gemm(o7, v9, v10...;
}
```

Listing 2. Complete DNN NNEF

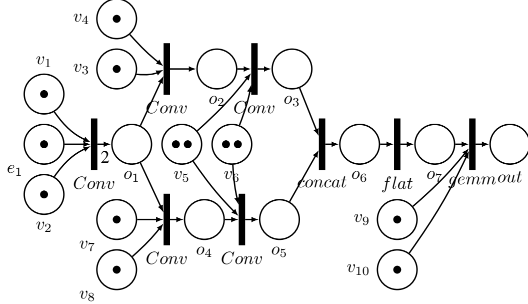


Figure 5. Petri associated to DNN of listing 2. Initial marking

pointers to the input tensor and fixed parameters must also be declared.

```
graphitem ITEM1 DNN1([e1, vsync2, vsync3]) -> ([
    vsync1, out])
{
   e1 = external...;
   v1 = variable... 'variable1'); v2 = ...; v9 =
       ...;  v10 = ...;
   vsync1 = variablesync<scalar> (shape ...;
   vsync2 = variablesync ...;   vsync3 =
       variablesync ...;

   o1 = conv(e1,v1,v2,...;
   vsync1 = send_var ([ITEM2, ITEM3], o1);
   o3 = get_var(ITEM2, vsync2);
   o4 = get_var(ITEM3, vsync3);
   o6 = concat (o3, o4); o7 = flatten(o4,...;
   out = gemm(o7, v9, v10...;
}

graphitem ITEM2 DNN2([vsync1]) -> ([vsync2])
{
   v3 = variable... 'variable3'); v4 = ...; v5 =
       ...;  v6 = ...;
   vsync1 = variablesync ...;  vsync2 = variablesync
        ...;

   o1 = get_var(ITEM1, vsync1)
   o2 = conv(e1,v3,v4,...; o3 = conv(o2,v5,v6,...
   vsync2 = send_var ([ITEM1], o3);
}

graphitem ITEM3 DNN3([vsync1]) -> ([vsync3])
{
   v5 = ...; v6 = ...; v7 = ...; v8 = ...
   vsync1 = variablesync...;  vsync3 = variablesync
       ...;

   o1 = get_var(ITEM1, vsync1)
   o4 = conv(o1,v7,v8,...; o5 = conv(o1,v5,v6,...
   vsync3 = send_var ([ITEM1], o5);
}
```

Listing 3. NNEF for all items

C. Petri-based semantic

We define the execution model semantics of multi-item descriptions using coloured Petri nets [JK09]. We associate a colour to each item where the colour is set to the tokens and edges (on which the coloured tokens transit).

***Translation** 2:* Let assume there are N items. We first apply the translation 1 for each item leading to N independent Petri nets (P_i, T_i, V_i), each with a unique and distinct colour. For the new instruction, the translation is extended as follows:

- a *varsync* does not generate any place;
- a *get_var* does not generate any transition;
- a *send_var* produces a transition *sync* with an incoming edge from the variable the content of which is transmitted.

The set of NNEF item descriptions generates a Petri net (P, T, V) which is roughly speaking the union of the N Petri nets (P_i, T_i, V_i). More precisely,

- any input tensor or fixed parameter that is duplicated in the NNEF files appears in the Petri net of the item. Those duplicated places are merged. Because we use the same naming convention, $P = \cup P_i$;
- the initial tokens in the places are also merged leading to places with possibly multiple tokens and multiple colours;
- $T = \cup T_i \cup T'$ where T' are the transitions connecting places of one item to other items thanks to the *sync* transition. More precisely,
  - for each writer, there are k edges back from the *sync* label where k is the number of readers. The colour of the each arrow is the one of the reader and the number of tokens sent back corresponds to the number of time the shared variable appears in the reader item instructions;
  - for each reader, there is an edge from the writer place before *sync* to each transition requiring the shared data;
- As before, when a coloured token is present in a place, it means that the associated variable is available for the item identified by the colour and can be used by transition.

***Example** 6: The Petri net associated to the example 5 is given in figure 6. We present the initial marking with colored tokens. Each color represents the state of one item. Compare to the figure 4, we express here the multi items semantics with synchronizations.*

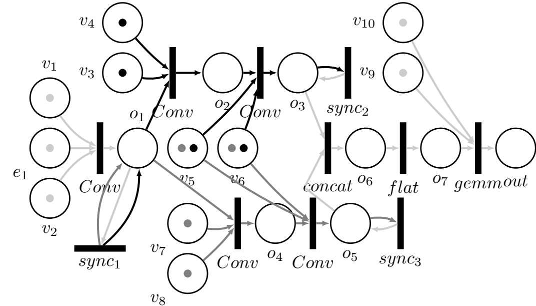


Figure 6. Semantic of the items synchronization

***Property** 3 (Equivalence between Petri nets):* The semantics of the multi-items behaviour is equivalent to the complete original ML model.

***Remark** 4:* It is important to explicitly describe the *send_var* and *get_var* either in the NNEF files but also in the Petri net because items are supposed to be independent and segregated. In particular, an item X is not allowed to access the memory space of an item Y and interfere with its execution.

This is classical in aeronautics, see Arinc 653 specification. The XTRATUM hypervisor [CRM+09] is an example of *time and space partitioning* hypervisor that provides communication with *sampling* and *queuing* ports (close to Arinc 653 requirements).

V. IMPLEMENTATION AND EXPERIMENTS

The previous sections showed how to fulfill the requirements 1 and 2 of the introduction. The purpose of this section is to give some hint of how a DO-178C compatible implementation process, as required per requirement 3, could be defined taking as input an extended NNEF specification. The considered target, a Jetson XAVIER TX system-on-chip, is composed of 6 Carmel ARM cores, a GPU, 2 deep learning accelerators (NVDLA) and other dedicated circuits. The use of a NVIDIA platform is mainly motivated by its availability and the ease to quickly deploy neural networks application. We will not discuss the adequation of GPU and CUDA implementation with DO-178C objectives because it is an open problem.

The *sync* implementation relies on barrier mechanism and each item NNEF description is manually coded. In order to validate the semantic preservation, we made some instrumentation to show that: (i) the execution trace is included in all possible execution traces defined by the Petri net; (ii) the numerical precision is kept; (iii) the measured execution time does not vary. We will use the multi-items example presented in example 5 as the specification. For our experiments, each item is allocated to one CPU and all the CUDA cores of the GPU (grouped in a CUDA stream). As a consequence, we do not guarantee a segregation between items (as they share the GPU) we instead focus on a way to implement parallel operations of neural networks with a static code scheduling while preserving the semantics. As for the Petri semantics, we only developed a code for a single inference. Nevertheless, it is easy to slightly modify the code by adding loops to handle several input tensors.

A. Get/Send specification

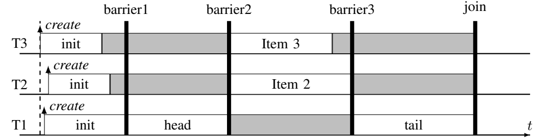
We chose to implement *get_var* and *send_var* with 1) global variables stored in the SRAM of the XAVIER and 2) the POSIX barrier mechanism of the *pthread* library. A barrier *b*, shared among several processes, will block them as long as not all of them reach *b*. Such a behavior is strictly included in the semantics of the *sync* transition within the Petri net. However, it is not the most efficient as it prevents the sending item to proceed until all the receiving items reach *b*, whereas the semantics of *sync* transition only requires a receiver to wait for the sender (not the sender to wait for all receivers). Nonetheless, the barrier mechanism is optimal for our example because no sender has to process any instruction before a further *get_var* or stop execution.

B. Manual code generation

There are C and PYTHON interpreters of the NNEF format [NNE18] but only for traditional CPU target. Consequently, no existing tool supports our syntax extension nor state-of-the-art GPU. Thus we developed the code for each item using C++ and CUDA using the CUDNN library. Basically the C++ code is executed by the ARM processor whereas CUDA allows the definition of kernels that are executed synchronously by all CUDA cores. The CUDNN library is built on top of CUDA for executing common neural networks layers.

1) Software architecture: Practically, each type of layer is implemented using a dedicated C++ class that inherits from the abstract *Layer* class that defines common attributes and methods to be implemented (*init()* and *forward()*) by child classes. In effect, *init* statically allocates tensors and CUDNN descriptors while *forward* launches the layer computation based on CUDNN for Convolution and max pooling layers. Each item contains one object implementing a static scheduler. More precisely, during the *init* phase, each item creates an object for each layer which are stored in ordered lists. Thus, items 2 and 3 need one single ordered list whereas item 1 needs two ordered lists (one for the first part and one for the second part). During *forward*, the scheduler calls in order the *forward* of objects stored in ordered lists.

2) Scheduling: We define one separate thread for each item allocated to one CPU + CUDA cores. More precisely, synchronizations between threads use `pthread_barrier_t` and associated APIs (`barrier_init` and `barrier_wait`). Barriers synchronize accesses to shared variables.



The execution sequence starts with the 3 threads creation on the CPUs and then reaches the first synchronization barrier. Then Item1 thread calls the *forward* method of layers of the head (until *send_var*) while Items 2 and 3 threads wait for the second synchronization barrier. After, the second synchronization barrier, Items 2 and 3 threads call *forward* method of their layers while Item1 thread waits for the third synchronization barrier. After the third synchronization barrier, Item 1 thread calls *forward* method of layers of the tail. At last, threads join and exit.

C. Semantic preservation of the Petri net

The first analysis aims at verifying that all observed scheduling of layers on the XAVIER respects the Petri net semantics. Because we use a static scheduler, all schedules should behave as shown in section V-C which is included in the semantics of coloured Petri net of figure 6. For that, we logged each start/end of branches and layers and we stressed the robustness of the implementation by addind some temporal noises (sleep in the code).

All observed traces respected the schedule of section V-C with some timing variations. When observing the implementation with no noise, execution traces of Item 2 and 3 are interleaved on the GPU. When adding a wait of 1s at the beginning of Item 2 (just after barrier1), all layers of Item 3 were executed before those of Item 2.

D. Semantics preservation of the function

The second instrumentation mechanism aims at checking that the functional semantics of the DNN is preserved. We achieve this by re-implementing the NNEF specification in PYTORCH. Then we define 100 random vectors that we run both on the PYTORCH implementation and on the C++ implementation on the NVIDIA target. Finally, we compute the overall average error mean between both executions for the 100 runs.

We were not able to find the exact convolution algorithm of PYTORCH. We think that it exists a non documented heuristic that calls the best algorithm (considering execution time) depending of the convolution parameters and available hardware. According to the CUDNN documentation [NVI19b], it is possible to select the convolution algorithm among a list, but details of the implementation are not given. Thus, there may be a discrepancy between convolutions that we cannot fix. The average error mean is extremely small 1.10^{-7} for FLOAT32 using 3 CUDNN algorithms (namely gemm, Winograd and direct). Nevertheless numeric precision results for this experiment are in an acceptable range that is very close to the available numeric precision of floating point representation and this also is observed by other frameworks [SCGP22].

E. Measured Execution Time (MET)

One objective of the DO-178C that we did not mention until now is the capacity to estimate the Worst Case Execution Time (WCET). Due to the complexity of NVIDIA target, a formal demonstration using static analysis may be difficult. But at least, a good property is a low variation of the measured execution time among several executions. In our case, the generated code does not contain any IF-THEN-ELSE patterns or dynamic loop conditions. Thus, the variability is only linked to the hardware behavior. We measured the MET of the complete DNN and of the first convolution of Item 1 over 10 runs. We rely on the *nsys* tool from NVIDIA to get timing measurements.

	Mean(MET)	**MIN(MET)**	**(MAX(MET))**	**STD(MET)**
First Conv	324 2976 ns	322 688 ns	326 656 ns	1.45 ns
DNN	24 257 μs	16 285 μs	53 950 μs	13 526 μs

The MET of the first convolution is very stable with a very low jitter. The MET distribution of the DNN is large and to understand why, we need to investigate the low level behaviour. NVIDIA GPUs are black-boxes processors on which we cannot guarantee worst-case execution time [AA21], [ACR22].

VI. Related work

There are plenty of different formats but no consensus within the community. State-of-the-art frameworks like PYTORCH, or TENSORFLOW rely on custom black-box formats built on top of *protocol buffers* [Goo01] developed by Google. A *protocol buffer* is a structured binary format that is not human readable and requires conversion tools and template files to be interpreted. Thus, the syntax is not formally defined and specification of layers are only available through documentation website. For example, TENSORFLOW proposes the *.h5* [HDF01] format and KERAS format [Ten15] builds on top of protocol buffer. Moreover the way to save a neural network is not unified among frameworks and may evolve every updated version with poor backward compatibility. When moving from caffe to caffe2 (known today as PYTORCH), the caffe [JSD+14] format was not supported anymore.

All previous formats were developed specifically for training frameworks (open source or proprietary) without any objective for sharing models. Their main purpose was to allow saving and reloading previous trained models without too much consideration on syntax and semantics. ONNX [BLZ+19] and NNEF [The22] were developed with the objective to be independent from frameworks. ONNX is still based on binary protocol buffers [Goo01] (so without a textual syntax) but is proposing a functional semantics through its github site. On the contrary NNEF is proposing a textual format with a syntax and a semantics that is formally defined in a specification. Unfortunately, the NNEF format suffers from a small community and tools supporting the format.

NNet [The22] format is an example of ad-hoc format. RELUPLEX neural networks examples [KBD+17] are in NNet. It is based on textual files but without definition of syntax and semantics. Import and export tools are provided, but its utilization for sharing neural networks between teams remains supremely painful.

Some other formats like [CE-17], [Apa18], [LAB+21a] tackle the need for intermediate representation between a neural network description and an implementation on a target. Especially this supports different optimizations passes like layers folding or low level tensor manipulation description like in LLVM [Lat02]. We consider that we are closest to programming language than from a neural network description format. Most of the time ONNX or NNEF are used as input like in [LAB+21b], [PBCPB20], [JBL+20].

Because DNN are data-flow, it is natural to translate them into *synchronous languages*. There are some works such as [LFG20] that proposed to encode them as Synchronous Dataflow Graphs or SCADE tool suite [CPPP18] which is currently developing a DNN libraries. Once the translation is done, it is then possible to reuse all the qualified code generation tools. This is complementary to our work as we could use the NNEF description to generate the SCADE program for instance. To the best of our knowledge, none of actual available neural network description formats propose solution for describing multi items implementation with concerns on sharing variables among them.

VII. Conclusion

We have proposed a formal extension of NNEF that takes into account the *execution model* of a description and allows for the modification of a description of a trained model to define traceable distribution and parallelisation optimizations that preserves the semantics while improving the execution time compared to a pure sequential aproach. We have also proposed a code generation strategy based on barriers for exchanging data between items. As a future work, a working group has been set up to propose an ONNX aeronautics profile.

Acknowledgments

This work has benefited from the AI Interdisciplinary Institute ANITI, funded by the "Investing for the Future – PIA3" program Grant agreement ANR-19-P3IA-0004 and from the PHYLOG 2 project funded by the French government through the France Relance program, based on the funding from and by the European Union through the NextGenerationEU program.

References

[AA21] Tanya Amert and James H. Anderson. Cupid^{rt}: Detecting improper GPU usage in real-time applications. In *24th IEEE International Symposium on Real-Time Distributed Computing, ISORC 2021, Daegu, South Korea, June 1-3, 2021*, pages 86–95. IEEE, 2021.

[ACR22] Michaël Adalbert, Thomas Carle, and Christine Rochange. PasTiS: building an NVIDIA Pascal GPU simulator for embedded AI applications. In *11th European Congress on Embedded Real-Time Systems (ERTS 2022)*, 2022.

[Apa18] Apache. TVM, 2018. https://tvm.apache.org/.

[BBD+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 586–601. ACM, 2017.

[BLZ+19] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://onnx.ai/, 2019.

[CE-17] CE-List. N2D2 frameworks, 2017. https://github.com/CEA-LIST/N2D2.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, 1989.

[CPPP18] Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. Scade 6: From a kahn semantics to a kahn implementation for multicore. In Hiren Patel, Tom J. Kazmierski, and Sebastian Steinhorst, editors, *2018 Forum on Specification & Design Languages, FDL 2018, Garching, Germany, September 10-12, 2018*, pages 5–16. IEEE, 2018.

[CRM+09] Alfons Crespo, I. Ripoll, M. Masmano, P. Arberet, and Metge Jean-Jacques. Xtratum: An open source hypervisor for tsp embedded systems in aerospace. pages 31–, 05 2009.

[EUR21] EUROCAE WG-114/SAE joint group. Certification/approval of aeronautical systems based on AI, 2021. on going standardization.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[Goo01] Google. Protocol Buffers, 2001. https://portal.hdfgroup.org/display/HDF5/Introduction+to+HDF5.

[HDF01] HDF group. Introduction to HDF5, 2001. hhttps://www.hdfgroup.org/solutions/hdf5/.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[JBL+20] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. Compiling onnx neural network models using mlir, 2020.

[JK09] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[KBD+17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017.

[LAB+21a] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[LAB+21b] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, et al. MLIR: scaling compiler infrastructure for domain specific computation. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[LFG20] Daniel Luenemann, Maher Fakih, and Kim Gruettner. Capturing neural-networks as synchronous dataflow graphs. In *MBMV 2020 - Methods and Description Languages for Modelling and Verification of Circuits and Systems; GMM/ITG/GI-Workshop*, pages 1–10, 2020.

[NNE18] NNEF project. NNEF GitHub tools repository, 2018. https://github.com/KhronosGroup/NNEF-Tools/.

[NVI19a] NVIDIA - Dustin Franklin. Introducing the Jetson Xavier Nx, 2019. https://developer.nvidia.com/blog/jetson-xavier-nx-the-worlds-smallest-ai-supercomputer/.

[NVI19b] NVIDIA developer team. CudNN library documentation, 2019. https://docs.nvidia.com/deeplearning/cudnn/api/index.html.

[Old86] Ernst-Rüdiger Olderog. Operational petri net semantics for ccsp. In *European Workshop on Applications and Theory in Petri Nets*, pages 196–223. Springer, 1986.

[PBCPB20] Hugo Pompougnac, Ulysse Beaugnon, Albert Cohen, and Dumitru Potop-Butucaru. From SSA to Synchronous Concurrency and Back. Research Report RR-9380, INRIA Sophia Antipolis - Méditerranée (France), December 2020.

[Pet77] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[RTC00] RTCA/EUROCAE. DO-254/ED-80 - Design Assurance Guidance For Airborne Electronic Hardware, 2000.

[RTC11] RTCA/EUROCAE. DO-178C/ED-12C - Software Considerations in Airborne Systems and Equipment Certification, 2011.

[SAE10] SAE/EUROCAE. Aerospace Recommended Practices ARP4754a/ed-79a- development of civil aircraft and systems, 2010.

[SCGP22] Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, and Claire Pagetti. ACETONE: predictable programming framework for ML applications in safety-critical systems. In *34th Euromicro Conference on Real-Time Systems ECRTS 2022*, pages 3:1–3:19, 2022.

[SZ06] Anton Maximilian Schäfer and Hans Georg Zimmermann. Recurrent neural networks are universal approximators. In Stefanos D. Kollias, Andreas Stafylopatis, Włodzisław Duch, and Erkki Oja, editors, *Artificial Neural Networks – ICANN 2006*, pages 632–640, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Ten15] TensorFlow. Keras Model, 2015. https://www.tensorflow.org/api_docs/python/tf/keras/Model.

[The22] The Khronos NNEF Working Group. Neural Network Exchange Format – Version 1.0.5, 2022. https://registry.khronos.org/NNEF/specs/1.0/nnef-1.0.5.pdf.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.

[Xil19] Xilinx. ZCU102 Evaluation Board – User Guide, 2019. https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd.