A study of an ACAS-Xu exact implementation using ED-324/ARP6983

Christophe Gabreau, Marie-Charlotte Teulières, Eric Jenn, Augustin Lemesle, Dumitru Potop-Butucaru, Floris Thiant, Lucas Fischer, Mariem Turki

A study of an ACAS-Xu exact implementation using ED-324/ARP6983

Christophe Gabreau*, Marie-Charlotte Teulières†, Eric Jenn ‡, Augustin Lemesle ¶, Dumitru Potop Butucaru ‖, Floris Thiant §, Lucas Fischer **, Mariem Turki ‡

* *Airbus,* † *Airbus Protect,* ‡ *IRT Saint-Exupéry,* § *IRT System X,* ¶ *CEA,* ‖ *INRIA,* ** *Datakalab*

Abstract: *This paper studies the exact implementation of the ACAS-Xu ML models (designed using Machine Learning technique) on several hardware platforms while ensuring some properties: ML model full semantics description, memory footprint optimisation, integer representation, formal verifiability. Certification aspects are also addressed using the EUROCAE/SAE joint group WG-114/G-34 current draft of the future standard ED-324/ARP6983 for embedding ML technology in aeronautical systems.*

Keywords: *Machine learning, Quantization, Formal verification, Implementation, ED-324/ARP6983, Certification.*

Disclaimer: *This paper is based on preliminary results of the EUROCAE WG-114/SAE G-34 working group and only reflect authors' view.*

I. Introduction

A. Context

In the airborne context, a safety-critical system cannot be certified as long as it is not demonstrated that this system safely performs its intended function under all foreseeable operating and environmental conditions. This demonstration only holds when the intent, along with the safety objectives, are satisfied in the target environment. When it comes to embed systems based on Machine Learning (ML) technology, the use of formal methods at design level seems very promising to support this demonstration and therefore, alleviate massive and costly testing activities on the selected HW platform. In addition, formal verification covers some of the learning assurance objectives from the novel ML standard ED-324/ARP6983 (ongoing work from the joint EUROCAE/SAE working groups WG-114/G-34).

This paper extends the previous works already performed on the ACAS-Xu case study in [3],[7] and [8]. While [3] and [7] were elaborating the design aspect of the development, [8] was focusing on the implementation part by proposing a method to implement a certifiable system with respect to ED-324/ARP6983 objectives. This paper proposes to study the capability to apply this method on several target platforms and to support the exact replication of formally proven ML models. This work has been performed in the frame of the Confiance.ai project[1] and the DeepGreen project[2].

B. Contributions

The contributions are the methods to implement a certifiable ACAS-Xu system using surrogate ML models. These methods are supported by the following activities:

- Design the models, i.e. quantify, formally verify the quantified models and provide a specification of the quantified models' semantics
- Implement the quantified models, i.e. study the capability of an exact replication by respecting the specification of the designed models on several targets
- Contribute to certification, i.e. provide elements of a certification argumentation to demonstrate the conformity with the current recommendations of the WG-114/G-34 joint working group.

The abstract is structured as follows: section II describes the related work in the field of implementation of embedded ML; section III gives a brief description of the use case used to illustrate the approach and introduces the overall implementation strategy; Sections IV, V, and VI respectively address the design, implementation and certification aspects of the development.

II. Related work

There is an important survey [16] that structures and analyzes challenges, techniques, and methods for developing AI-based safety-critical systems. In particular, it addresses the development and the integration of a ML-based function as part of a safety critical system hosted on hardware platforms. References to development are mainly based on Autonomous Driving (AD) software frameworks using existing automotive standard guidance. To our knowledge, there is no such work in the airborne context, about the implementation of a ML model on hardware targets following the guidance of the future aeronautical standard ED-324/ARP6983. In an airborne context, there are some previous works dealing with the implementation of a DNN model on a recent CPU processor [1], studies about the prevention of the propagation of hardware errors that can lead to catastrophic failures in DNN

[1]Web site: confiance.ai
[2]Web site: deepgreen.ai

accelerator systems [14], or some guaranties of robustness against hardware soft errors corrupting the target memory during implementation [9]. There are also work on other domains, for instance robotics, where implementation is also a challenge when it comes to embed DNN model for computer vision and system control purposes in different target hardware such as CPUs, GPUs or Intel's neuromorphic chip [4].

III. Case Study and Implementation Strategy

A. The ACAS-XU case study

The ACAS-XU case study is described in details in [3]. Basically, the ACAS-XU system contains a specific function which computes the correct maneuver in order to prevent collision between UAVs for a set of input data (geometrical configuration of the ownship and the intruder UAVs). The implementation of the function is based on standardized lookup tables which provide the best advisory maneuver according to the geometry of the system given in Figure 1. The parameters are:

- ρ (ft), the distance from ownship to intruder
- θ (rad), the angle to intruder relative to ownship heading
- ψ (rad), the heading angle of intruder relative to ownship heading direction
- v_{own} (ft/s), the speed of ownship
- v_{int} (ft/s), the speed of intruder
- τ (s), the time until loss of vertical separation.

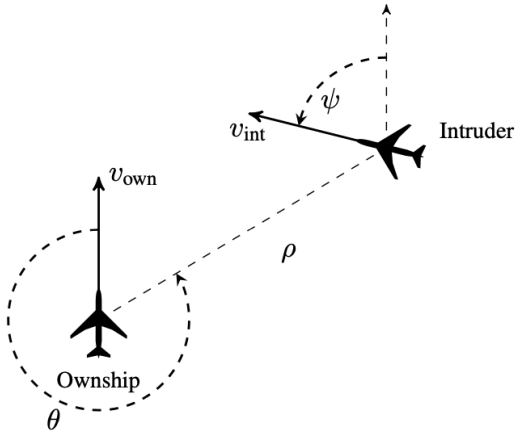

Fig. 1. ACAS Xu geometry [11]

The ACAS-XU function is implemented using ML. The ML part of the system, or ML Constituent (MLC) as per WG-114/G-34, is composed of ML model(s) and associated pre/post data processing that can be deployed on one or several *items*. The use of the term "item" complies with the definition given in the existing airborne system development guidance ED-79B/ARP4754B [19].

B. Implementation strategy

We decided to reuse the safe hybrid architecture that was used to develop the ACAS-Xu ML-based function[3], which is composed of the *NN-based controller* (basically the MLC contains the models and the pre/post processing code), the *safety net* and a *check module* which, in real time, conditions the execution path to provide the correct advisory (cf. Figure 2).

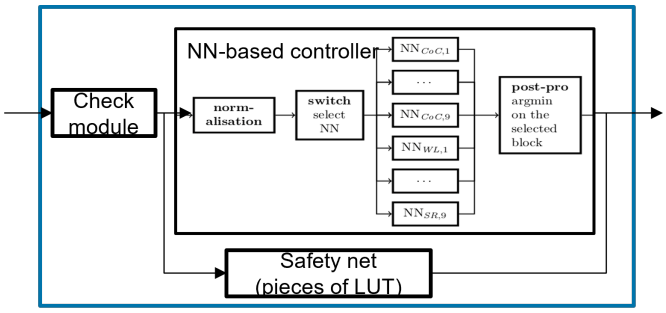

Fig. 2. Hybrid Architecture Overview [3]

We applied the method proposed in [8] to define an implementation strategy with the following main phases:

- **Specification**: We specified some requirements at MLC level (performance, functional, safety, operational), and we reused the similarity property defined in [3] (*"decisions of the model are identical to the ones obtained from the tables"*). In addition, we defined some specific requirements to cover the pre/post processing of the data. This phase is not developed further.
- **MLC Design/Quantization**: The full MLC logical architecture is decomposed into 45 models (and 45 Operational Design Domains-ODDs) as per the original study [3]. We simplified the problem by only considering avoidance maneuvers in the horizontal plan τ=1. The 5 considered models (with their own ODD) are quantified in order to meet the implementation constraints, indeed integer representation is preferred for memory footprint and inference latency reduction purposes. Once quantified, the models are exported using the ONNX format. These models become the reference for both the formal verification and the implementation.
- **Model Design/ODD partitioning**: Each of the 5 ODDs is partitioned into two domains: one domain in which each quantified model is *proved* to hold the similarity property and one domain in which it does not. In the latter case, a safety net is used instead (cf Figure 2). These proofs are realized by using formal verification methods. At the end of the formal verification activity, we know for sure where the quantified models correctly perform.
- **Model Design/Description**: As per ED-324/ARP6983, the outcomes of the design phase is the ML Model Description (MLMD). This MLMD should express the full semantics of the quantified model to enable the capability of exact replication during implementation. This paper describes the semantics of the quantified models and stresses the gap between the current existing format and what is really needed in the avionic field.

- **Implementation**: This phase starts with the architecture of the ML Model, i.e. its decomposition into multiple ML Model Item Descriptions (MLMIDs), each MLMID corresponding to one item of the physical architecture. Then MLMIDs are transformed into implementation models for the different targets. In this paper, we consider the *exact* replication of the ML models, i.e. the inference ML model correctly and completely implements the specification of the ML model semantics expressed in the MLMD. Specific techniques (MLIR/iree tools suite) are used to control the transformations from MLMID to executable code in order to demonstrate the full preservation of the quantified model semantics and support the exact replication.
- **Exact replication study**: According to the obtained replication level (related to the used target), the performance of the implemented models are verified against the performance of the designed models.
- **Certification argumentation**: We built an argumentation which demonstrates the objectives defined in the EUROCAE/SAE WG-114/G-34 ongoing standardization work (ED-324/ARP6983). This argumentation contributes to guarantee that the implementation preserves the model properties and do not introduce any unacceptable unintended behaviours. The argumentation is developed using the GSN assurance case notation.

IV. Design

A. Quantization

Quantization plays a pivotal role in the optimization of machine learning models, with a specific focus on the Data-free method [15], [6]. This method aims to preserve the mathematical function of Deep Neural Networks (DNNs), making it particularly relevant in the context of the ACAS-Xu case study. Here, we consider a subset of five models at τ=1. The notion of data free quantization involved no use of the data to compute a metric to optimize. In [6], Residual error has been introduced, and represents the error between the original weights, and the quantized weights. Let $W - Q^{-1}(W^q)$, with $R^1 = W^q$ be the residual error, and R^k be the K^{th} residual expansion term such that :

$$R^{(K)} = Q\left(W - \sum_{k=1}^{(K-1)} Q^{-1}(R^{(k)})\right) \quad (1)$$

The maximal error between the original weight and the quantized weight decreases exponentially with the expansion order. The quantization process ensures a high preservation of the accuracy.

Table I summarises the results obatined by the quantization process, compared to the original models (with τ = 1). An overall prediction accuracy of under 2% compared to the full precision model has been attained. The accuracy on next state prediction remained consistently higher than 95%, highlighting the effectiveness of the applied quantization techniques.

The quantization process achieves precise activation using diverse calibration methods. Surprisingly, despite numerous attempts, the introduced bias correction by [15] did not contribute significantly to this task.

Previous advisory	**R2 Score**		**Argmin truth accuracy**	
	Original	Quantized	Original	Quantized
model_CoC	99,82	99,1	96,05	95,35
model_WL	99,48	97,9	96,5	95,25
model_SL	99,52	98,02	96,03	95,19
model_WR	99,51	97,57	96,88	95,04
model_SR	99,57	97,98	96,15	95,15

TABLE I
COMPARISON OF METRICS BETWEEN ORIGINAL MODEL, AND QUANTIZED MODELS, WITH τ=1 AND DEPENDING ON THE PREVIOUS ADVISORY.

B. ODD partitioning

The partitioning of the ODD is realized by verifying where the similarity property holds. This leads to the partitioning of the ODD into 2 classes of input space: where the NN-based controller performs correctly and where it does not (in this case the safety net is used as per Figure 2). The property is then verified using formal methods.

Following the same approach as [3], we decompose the input space of the neural network in p-dimensional boxes. As we selected a subset of the models with $\tau = 1$, we consider in this section only five models for the formal verification, each depending on the previous advisory and taking inputs in a five dimensional space with $(\rho, \omega, \psi, v_{own}, v_{int})$. The decomposition of the input space is thus a set of 7 356 800 5-dimensional boxes for each model, for a total of 36 784 000 boxes.

For each of these boxes, we aim to determine that the decision of the neural network are included in the decisions of the look up table (LUT) as formalised in [3]:

$$\forall l \subseteq \mathbb{R}^5, \text{decisions NN}(l) \subseteq \text{decisions LUT}(l)$$

For the formal verification, PyRAT [12] is used. PyRAT is based on abstract interpretation [2] and relies on abstract domains such as Zonotopes [10] to soundly overapproximate all possible outputs of a neural network. PyRAT computation and domains are sound w.r.t. floating point arithmetics, correctly rounding the different variables towards minus and plus infinity. Due to the number of verifications to perform and the constraints of the quantification, we chose to use only the Zonotope domain for the verification in a correct but incomplete way. Indeed, as the ReLU activation functions of the network have been replaced by quantisation operations, classical branch and bounds approaches by case disjunction on ReLU cannot be performed or would need to be adapted.

The formal verification is done by PyRAT on an implementation of the operators of the neural network reproducing exactly the ONNX reference implementation. On the ACAS-Xu case study, while the models are partially quantified (all the weights and bias of the models are converted to 8-bits unsigned integer), some rescaling operations with floating points are

still occurring in the QGemm operators that constitute the network. Additionally, a first conversion from float32 to uint8 is applied by a QuantizeLinear operator as the input of the network are float32 numbers. Following this, the inputs of a QGemm operator are first cast to 32-bits integer on which the the classical Gemm is applied (with weights and biases also in int32). The results of which is cast to float32 and rescaled before being cast back into 8-bits unsigned integer. Thus, while the integer operations preclude any floating point inaccuracy and rounding considerations, the rescaling operations (multiplication by scalar and addition with a bias in float32) may lead to floating point errors.

As we aim to have a verification correct w.r.t. the floating point implementation, *i.e.*, taking into account all possible floating point errors, the errors introduced by the rescaling in float32 (and more specifically in the addition) have been integrated in PyRAT in the Zonotope abstract domain. In addition to this, a specific abstract transformer has been defined for the cast function from float32 to uint8. Indeed, this function is not linear and must thus be overapproximated by a linear function to be used with an abstract domain like the Zonotopes. For an input x, we define the abstract transformer of the cast function as:

$$cast^{\#}(x) = x + 0.5 * \epsilon_{noise}$$

with $\epsilon_{noise} \in [-1, 1]$ the approximation introduced by the operation. This is illustrated in Figure 3.

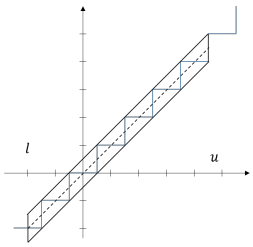

Fig. 3. Abstraction of the cast function from float32 to uint8 for the Zonotope abstract domain in PyRAT over an input interval $[l, u]$. In blue the function itself and in black its abstraction.

Table II summarises the results of the verification in function of the previous advisory. The total analysis time on the 36 millions of boxes with PyRAT was of approximately **40 hours**. Safe boxes means the similarity property holds, for unsafe boxes a counter example to the similarity property was found and unknown means we cannot conclude. As it is usually difficult to find counter examples with simple abstract interpretation based approaches, we rely on counter examples search using adversarial attacks inspired functions. We see that a large proportion of unknown results still remains as we used a fast but incomplete method. Using more precise domains or a complete approach may improve these results. However, considering the running time of complete methods and the number of points to verify (even if only 30% of the points are remaining) the computational cost of a complete verification remains prohibitive.

Previous advisory	Safe boxes	Unsafe boxes	Unknown
CoC	73%	0%	27%
WL	56%	0%	44%
SL	73%	3%	23%
WR	66%	4%	30%
SR	74%	4%	22%

TABLE II
PERCENTAGE OF SAFE, UNSAFE AND UNKNOWN BOXES IN THE ODD PARTITION AS VERIFIED BY PYRAT.

In order to further improve these results, the p-box partitioning might be delegated directly to PyRAT, instead of hard-coding it. This could allow a more dynamic p-box splitting, leveraging the fact that the similarity property might have been proven on already larger parts the input space than our current p-box. Thus allowing a finer partitioning only when needed with heuristics already existing in PyRAT [5]. On the other hand, for this verification the Zonotope domain was only adapted to the integer values with overapproximation. It might benefits from more in-depth changes, allowing to handle the cast operations with more precision.

C. Expression of the ML Model Description (MLMD)

As per ED-324/ARP6983 guidance, the MLMD is the input of both the model verification and the MLC implementation process. It is key for the exact replication approach because it provides the capability to describe the complete semantics of the designed ML model, and enables to demonstrate that the implementation process does not alter this semantics and therefore, that the ML model properties which have been verified at design level, still hold.

In the context of ACAS-Xu case study, the MLMD is based on the existing ONNX format. This format is not sufficient to meet the "complete semantic description" requirement of the MLMD. Several elements are missing and are currently being addressed in a working group (connected to the ONNX SIG Operator). These elements are:

a) Operators full specification: The objective is to propose a complete specification of operators, unambiguous, no subject to interpretation or approximation. The specification should use a set of well-defined operators/primitives. The ACAS-XU model is a model composed of 9 fully connected layers represented by QGEMM as the model has been quantized. A complete specification of the QGEMM operator would be as follows:

- **Definition of the operator** : QGEMM is the quantized form of the GEMM (GEneral Matrix Multiply) operator
- **Description of inputs and attributes** : it includes roles description, data type, default value, constraints, description of the impact on the semantic of the operator.
  Example on Q_a input :
  Description : Q_A is the first input tensor involved in the multiplication.
  Data Type : tensor (integer)
  Default : No default, input required
  Constraints : If $transQ_A \neq 0$, Q_A should be transposed.

Transposition should be performed before computation of the operation. If $transQ_A = 0$, the shape of Q_A should be (M,K). If $transQ_A \neq 0$, the shape of Q_A should be (K, M).

- **Specification**
  Textual :

$$Y_q[i,k] = \frac{S_A \times S_B}{S_Y}\left(\sum_{j=1}^{N} Q_A[i,j] \times Q_B[j,k] - Z_B \times \sum_{j=1}^{N} Q_A[i,j] - Z_A \times \sum_{j=1}^{N} Q_B[j,k] + N \times Z_A \times Z_B + Q_C[i,k]\right) + Z_Y \quad (2)$$

  Q_A, Q_B, and Q_C are the quantized input tensors of QGEMM, and Q_Y is quantized output. Z_A, Z_B, and Z_Y (resp. S_A, S_B, and S_Y), the zero points (resp. scaling factors) of the quantization[3].

- **Reference implementation** Based on the C++ implementation proposed by ONNX Runtime.
  To establish a correspondence between the equation and the QGEMM documentation in ONNX Runtime, we can equate Q_A, Q_B, and Q_C to matrices A, B, and C, respectively. Correspondingly, Z_A, Z_B, and Z_Y can be associated with the parameters a_zero_point, b_zero_point, and y_zero_point, while S_a, S_b, and S_y align with the scales a_scale, b_scale, and y_scale, respectively.

b) Control flow: The control flow introduces the scheduling, the decomposition of the MLMD into several items with its interactions. Items could be hardware or software. In the case of the ACAS XU, three items are considered as shown in figure 4 on the implementation on TDA4VM Jacinto SoC: one hardware item, the Jacinto SoC, and two software items CPU and DSP (or GPU in the case of NVIDIA Xavier TX2). Several tools are being investigated within the MLMD community to support the expression of the control flow.

c) Syntax: The syntax should be textual and human readable in order to be open to manual code, and auditable. ONNX is based on protocol buffers, natively binary. The ONNX format is not human readable, this point is also adressed in the MLMD community

V. Implementation

A. Physical design

As per ED-324/ARP6983 guidance, this phase aims at defining the appropriate architecture of the MLC supporting the deployment of the ML Models on possibly one or several resources of the selected target. This implementation activity deals with the both components of the design outcome:

- MLMD part (ML model): The appropriate architecture is designed with respect to the resources available on the selected hardware platform. This activity will identify all the items which are necessary for the deployment of the model(s) onto the target. The MLMD is decomposed into ML Model Item Descriptions (MLMIDs). There is one MLMID per item.
- Non-MLMD part: the specific software parts needed for the pre/post data processing and the selection of the proper model execution (*Check Module* in Figure 2) according to operational inputs.

The ACAS-Xu model has been deployed on several targets:

- **a** TDA4VM Jacinto SoC fitted with 2 Cortex A72, 4 Cortex R5F, 1 C7X DSP and 2 C66X DSPs
- **a** NVIDIA Xavier TX2 fitted with 8 NVIDIA Carmel cores and a Volta GPU with 512 CUDA cores and 64 Tensor cores,
- **a** Xilinx Kria KV260 fitted with a K26 SoM embedding a ZYNQ UltraScale+ with 4 Cortex-A53 cores, 2 Cortex R5F cores, and a 256K logic cells FPGA fabric.

In all three cases, the ML model is implemented partially on CPU cores and an accelerator (respectively: a C7X DSP, a GPU, a DPU IP). The ML model (MLMD) is decomposed into 2 SW items (MLMIDs) hosted on one HW item (the SOC target):

- SW item 1: deployed on the CPUs (pre/post processing code wrt Figure 2)
- SW item 2: deployed on the accelerator (models)

For instance and with reference to Figure 4, in the Jacinto (HW item) implementation, each model is implemented using two kernels, one doing the quantization and the other doing the QGEMM operation. Kernels deployed on the C7X DSP accelerator using TI's TIOVX workflow that implements the OpenVX standard.

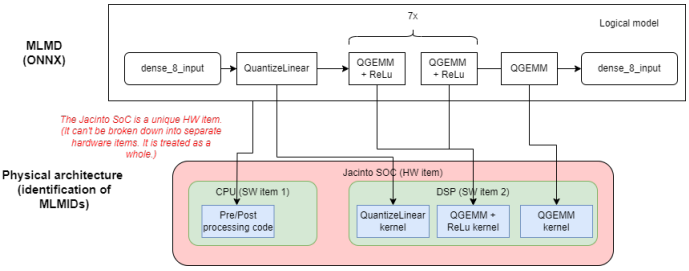

Fig. 4. Physical architecture on Jacinto target

B. Items development

In this phase, each item is implemented (coding, compiling, linking, loading and integrating) for the platform resource to which it is deployed. For instance the Jacinto implementation is described in Figure 5.

C. Exact replication study

a) Verification strategy.: Two verification methods can be applied:

1) Option 1: demonstrate that the series of transformations performed by the tool chains actually preserve the model semantics

[3]All attributes,inputs, outputs are supposed to be well-defined as described with Q_A

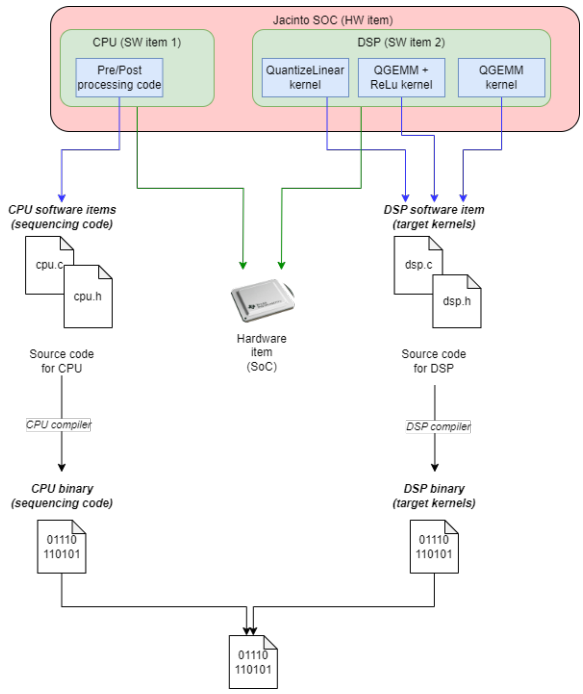

Fig. 5. SW items implementation on Jacinto target

2) Option 2: verify formally or by testing that the series of transformations preserves the model semantics.

When using complex COTS implementation tool chains, the first option is generally not an option, because the series of transformations is too complex to be analyzed and demonstrated to be correct (see paragraph about the VITIS AI and TIDL implementations later in this section), or simply because the details of the implementation are not available. This is to some extent similar to the case of the compilation of source code: except in some very specific cases (e.g, CompCert[13]), the complex series of transformation performed by a compiler cannot be demonstrated to preserve the semantics of the input code. In that case, the solution consists to verify the implementation by testing. For testing to be applicable, a correctness criterion must be defined and an oracle providing the expected value is needed. A correctness criterion can be (i) the identity of the final output of the network (the advisory to be applied), (ii) the identity of the last layer of the network (costs provided by the LUT), or (iii) the identity of all activations of the network. If testing is exhaustive, then all criterion are equal so the simplest (i.e., (i)) is sufficient: indeed, if the outputs of the implementation under test and the output of the reference implementation are identical for any input, there is no need to check what is going on in the intermediate layers. However, if testing cannot be exhaustive, which is generally the case, more confidence can be achieved by ensuring that intermediate results, e.g., activations, are actually identical.

This means that those activation values must be computed, which also means that the execution infrastructure (software and hardware) is also part of the oracle.

b) Application to the ACAS-Xu use case: As shown on Figure 6, the ACAS-xu model is a 9-layer fully connected network using essentially QGEMM operators, a quantified version of the classical GEMM (Generalized Matrix Multiplication) operator.

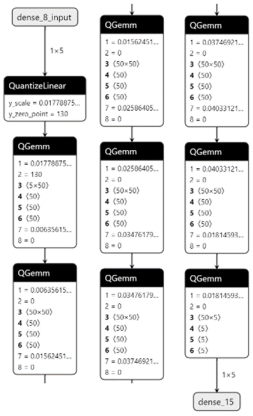

Fig. 6. ACAS-xu neural network (represented using Netron)

The ONNX runtime provides several optimized implementations of the QGEMM operator for different hardware targets. In our context, we implemented the QGEMM operator in the most straightforward way with respect to the mathematical expression of the operation shown on Eq. 2.

It is worth noting that the QGEMM documentation provided by ONNX gives a minimalist description of the semantics of the operator: "Quantized Gemm". In particular, no part of the specification mentions anything about the accuracy of the operator. In our context, the ONNX-runtime implementation was considered as the reference implementation, i.e., the one defining the actual and exact semantics of the operator.

Our first implementations on the TDA4VM Jacinto SoC through TIOVX and the NVIDIA Xavier GPU using cutlass and CUDA were based on a direct implementation of Eq. 2.

Excerpts of the cutlass[4] and Cuda implementations of Eq. 2 are given hereafter.

```
void qgemm(...) {
 [...]
 if (num_layer == 0) {
  sz_in = SIZE_MODEL_IN;
  quantize(r1, s1, z1, q1, size_layer_in); }
  else if (num_layer == NB_LAYER-1) {
   sz_out = SIZE_MODEL_OUT; }
   // q1 = q1 x q2 + q3
   result = cutlass_gemm(1, sz_out, sz_in, 1,
                q1, q2, 1, q3, q1);
   [...]
   if (result != cudaSuccess) {...}
   // q1 = q1 - z1 x ones * q2
   result = cutlass_gemm(1, sz_out, sz_in, -z1,
     ones, q2, 1, q1, q1);
   [...]
   // s2 = s1 x s2 / s3
   scaleOp(s1, s2, s3, s2, sz_out);
   // r1 = q1 x s2
   elementwise_matmul(q1, s2, r1, sz_out);
   if (num_layer != NB_LAYER-1) {
    relu(r1, q1, sz_out); }
```

```
}
```

`cutlass` provides highly optimized versions of gemm operators. Even though this "library" is open source, traceability between the operators and their implementation is not straightforward (for the very reason that it is *optimized* and optimization usually increases complexity and lowers traceability), we also considered a lower-level and naive CUDA implementation:

```
/* Non-optimized GEMM CUDA kernel */
__global__ void kernel_cuda_naive_gemm(...) {
 int i = blockIdx.x;
 int j = threadIdx.x;
 if (i < M && j < N) {
  float accumulator = 0;
   for (int k = 0; k < K; ++k) {
     accumulator += A[i*K + k]*B[k*N +j] ; }
   D[i*N + j] = alpha * accumulator +
    beta * C[i*N + j]; }
}
/* Non-optimized GEMM implementation */
void cuda_naive_gemm(...) {
 dim3 grid(M, 1, 1);
 dim3 block(N, 1, 1);
  kernel_cuda_naive_gemm<<< grid, block >>>
   (M, N, K, alpha, A, B, beta, C, D);
}
/* QGEMM implementation */
void qgemm(...) {
 [...]
 if (num_layer == 0) {
  sz_in = SIZE_MODEL_IN;
  quantize(r1, s1, z1, q1, size_in); }
  else if (num_layer == NB_LAYER-1) {
   sz_out = SIZE_MODEL_OUT; }
  // q1 = q1 x q2 + q3
  cuda_naive_gemm(1, sz_out, sz_in,
   1, q1, q2, 1, q3, q1);
  // q1 = q1 - z1 x ones x q2
  cuda_naive_gemm(1, sz_out, sz_in,
   -z1, ones, q2, 1, q1, q1);
  // s2 = s1 x s2 / s3
  scale(s1, s2, s3, s2, sz_out);
  // r1 = q1 x s2
  elementwise_matmul(q1, s2, r1, sz_out);
  if (num_layer != NB_LAYER-1) {
   relu(r1, q1, sz__out); }
}
```

Those implementation intially showed differences with the ONNX runtime implementation, from the very first layer of the network whereas the CPU and GPU variants of the ONNX-runtime gave identical results. Investigation where carried out both by analysing and instrumenting the ONNX source code in order to track the origins of the differences. In order to get the details about its actual behaviour, it was necessary to look to the source code[5]

In particular, the scale ratio and accumulator products were observed for they were the only place involving floating point computations. And, indeed, differences were observed that were tracked down to the different ways to perform rounding between our implementation (rounding to the nearest integer value) and ONNX' (rounding to the nearest even integer).

Note that finding the origin of these discrepancies required a thorough analysis and time consuming analysis of the ONNX implementation that was hopefully available. After those investigations, our two implementations (CPU with DSP and GPU) provided the very same results for all test cases (around 100000) as the ONNX runtime implementation. Indeed, the results for each layer are stored in a binary file and subsequently compared to the output binary file generated by the reference implementation. This process ensures identical implementations and consistent inference results.

Nevertheless, since there is not formal specification of the expected result, there is obviously no guarantee that our implementation will provide the same result as ONNX' implementation for another target, even for this simple operator. Other discrepancies may show up depending on the actual execution order of mathematical operations, from one implementation to another or, even worse, from one execution to another for the same implementation. This is clearly not acceptable for a test oracle.

A third implementation of the same model was done on a Xilinx' Krya KV260 board fitted with an UltraScale+ MPSoC chip. In that case, we were not able to ensure the strict equivalence of the deployed model with respect to the reference ONNX model. Indeed, the VITIAS-AI toolchain used to deploy the model does not accept the ONNX quantized model. Indeed, it provides its own optimisation tool (Vitis AI Optimizer) that implements specific quantization techniques. Therefore, we reimplemented the ACAS-Xu non quantified model using Keras, and used Vitis AI quantizer to generate the quantized model. Formal verification could possibly be performed on this quantized model, but the lack of documentation about the generated model prevented us to do so. Nevertheless, given the open-source nature of Vitis AI Quantizer, an in-depth reverse engineering of the format generated could be considered.

The same conclusion applies to a fourth implementation done using Texas instruments' TIDL toolchain for the TDA4VM: the toolchain provides its own quantization tool, and the binary files generated by the tool would have to be reversed engineered to get the weight and perform formal verification.

D. Specification and implementation using MLIR and iree

While the first exact replication study of Section V-C has relied on significant manual implementation phases, **we have conducted a second study aiming at automating the implementation process**. This study uses the MLIR compiler infrastructure and IR[6] for ML and HPC, and the MLIR-based iree runtime and compiler for mobile and edge applications.

MLIR's key originality is to use a general SSA-based syntax [18] to allow mixing aspects of a system (data types, operators) belonging to multiple representation levels. The types and operators are grouped into domain-specific *dialects* allowing the high-level representation of an ML algorithm (using dialects such as `tosa` or `hlo`) and then the progressive and seamless *lowering* of the abstraction level all the way to

[5]The source code is available at .

[6]Intermediate Representation used during compilation.

```
1  func.func @qgemm(
2    %a:tensor<?x?xui8>,%a_scale:tensor<f32>,
3    %a_zp:tensor<ui8>, %b:tensor<?x?xi8>,
4    %b_scale:tensor<f32>,%y_scale:tensor<f32>,
5    %y_zp:tensor<ui8>)->(tensor<?x?xf32>) {
6    %b1 = mhlo.convert %b
7        : (tensor<?x?xi8>) -> tensor<?x?xi32>
8    %1 = mhlo.convert %a
9        : (tensor<?x?xui8>) -> tensor<?x?xi32>
10   %q1q2 = mhlo.dot %1, %b1 : (tensor<?x?xi32>,
11         tensor<?x?xi32>) -> tensor<?x?xi32>
12   %2 = shape.shape_of %a :...
13   %3 = arith.constant 0 : index
14   %4 = arith.constant 1 : index
15   %5 = tensor.extract %2[%3] : tensor<2xindex>
16   %6 = tensor.extract %2[%4] : tensor<2xindex>
17   %7 = mhlo.convert %a_zp
18       : (tensor<ui8>) -> tensor<i32>
19   %o1 = tensor.splat %7[%5,%6]:tensor<?x?xi32>
20   %zp1 = mhlo.dot %o1, %b1 :...
21   %8 = arith.subi %q1q2, %zp1 :...
22   %acc = mhlo.convert %8
23       : tensor<?x?xi32> -> tensor<?x?xf32>
24   %9 = arith.mulf %a_scale,%b_scale :...
25   %M = arith.divf %9,%y_scale : tensor<f32>
26   %10 = func.call @scalar_tensor(%M,%acc):...
27   func.return %10 : tensor<?x?xf32>
28 }
```

Fig. 7. High-level MLIR representation of the qgemm function

optimized GPU, CPU, TPU or even FPGA-based implementations. This process typically passes through general linear algebra representations (dialect `linalg`), affine loop nests (dialect `affine`), or vectorized code (dialect `vector`) where domain-specific optimizations are applied before reaching low-level dialects such as `llvmir` or `amdgpu`. When necessary, the compilation process will partition the code between multiple devices (e.g. GPU kernels vs. host CPU control code).

MLIR allows the representation of MLMD specifications, of the MLMID-level information, and ultimately the implementation code. The format is textual, allowing inspection after each lowering or optimization phase, making it possible to trace the transformation of high-level operators such as `Conv2D` into, for instance, loop nests that have been fused with other loops, tiled, vectorized and parallelized on a multi-core.

We provide in Fig. 7 an MLIR encoding of the qgemm function.[7] Functionally equivalent Python code is provided for reference in Fig. 8. To facilitate understanding, the MLIR code closely follows the Python reference, all Python variables having counterparts in the MLIR code. For conciseness, we assumed that scalar-tensor product is implemented under the form of external function `@scalar_tensor`, called from function `@qgemm`. The MLIR implementation is naturally more verbose, as the language is designed as a compiler IR. In particular, the type of each variable and operation is explicitly defined (in gray in Fig. 7).

Each MLIR operation (in blue in Fig. 7) belongs to a domain-specific dialect, represented as a prefix to the operation name. In addition to function `@qgemm` (and the scalar-tensor

[7]Under simplifying assumptions considered previously in the paper: initialization with zeroes, zero point for the second argument equal to 0, float output...

```
import numpy as np
def qgemm(a,a_scale,a_zp,b,b_scale,
          y_scale,y_zp):
    b1 = b.astype(np.int32)
    q1q2=a.astype(np.int32)@b1
    o1 = np.full(a.shape,a_zp,dtype=int32)
    zp1 = o1@b1
    acc = (q1q2-zp1).astype(np.float32)
    M = a_scale*b_scale/y_scale
    return M*acc
```

Fig. 8. Python functional reference for the MLIR code in Fig. 7

product functions it calls), the high-level MLIR specification of the ACAS-Xu neural network (of Fig. 6) also includes a `@relu` function, and the top-level model function that calls `@qgemm` and `@relu` multiple times.

Note that the semantics of all operations is fully defined, which requires the specification of all data types. In function `@qgemm` this requires the specification of all data conversions, *e.g.* that of 8-bit unsigned integers to 32-bit signed integers (in lines 8-9). Notice how `@qgemm` allows mixing high-level data processing operations of the `mhlo` dialect, simple arithmetic operations applied pointwise on tensors of the same size (of dialect `arith`), general tensor manipulations of the dialects `tensor` and `shape`, and classical function constructs of the `func` dialect (function definition, call, return).

Importing into MLIR is possible from a multitude of formats including Jax, Tensorflow, or Pytorch, or ONNX. However, `qgemm` involves complex type conversions produced by the automatic quantization process described in Sec. IV-A, meant to reduce execution time and parameter size without penalizing ML performance. Thus, the implementation of a single Dense layer involves int8, uint8, int32, and float32 tensors. To make sure these conversions are preserved, we have manually produced the code of Fig. 7.

The output of the quantization process has the important property that floating point operations (which are not associative) are not used for the performance-critical matrix multiplications, thus allowing their optimization using the full power of parallelization algorithms (e.g. tiling), without affecting the output. Thus, exact replication can be attained without restricting optimization.

The dialect mechanism is designed for extensibility. For instance, in previous work [17] we have extended MLIR with a dialect providing the dataflow control primitives of Simulink and Lustre/SCADE, thus allowing the specification of stateful cyclic controllers (allowing the representation of stateful neural networks or reinforcement learning algorithms) and then their direct compilation into high-performance cyclic controllers.

But for the context of this paper, the extensibility of MLIR is best demonstrated by the experimentation platform we used, which is iree—an ML run-time and compiler producing code tailored for the run-time, to run on various targets : CPU (single- or multi-core), GPUs, and other accelerators. The iree compiler extends "vanilla" MLIR with new dialects allowing the description of a hardware abstraction, of execution man-

agement, of partitioning, allocation and scheduling. It also streamlines compilation for the run-time. On the resulting implementations, we apply fine-grain performance tracing using the Tracy profiling tool, to gain insight into performance bottlenecks.

Note that, unlike other ML compilation infrastructures, MLIR allows adopting a white-box approach to the design of the compilation pipeline, which can be completely customized. Along with the ability to trace source code transformation between compilation passes, this allows exposing critical transformations such as XLA-level linear algebra optimizations, tiling, buffer allocation, identification of computational kernels (to be executed on GPUs), parallelization, etc., which in turn allows (1) making performance trade-offs by activating/parameterizing compilation passes depending on the application and target architecture and (2) incrementally establishing the equivalence between the source code and the implementation.

VI. Certification

Regarding the certification aspects, we used the concept of assurance case developed in [7] and followed the method and the notation developed in [8] to elaborate the argumentation. Indeed, we built an assurance case based on the development assurance principles of the last draft version of the future ML standard ED-324/ARP6983 available at the time we wrote the paper. Doing so, we anticipate that the airborne authorities (e.g. EASA) will recognize this standard as an acceptable means of compliance to the regulation requirements.

The standard ED-324/ARP6983 is an end-to-end standard that provides guidance to develop and certify an airborne system whose at least one function is ML-based. The figure 9 proposes the first level of the assurance case encompassing all the processes that have been standardized. The ED-324/ARP6983 contains all the usual processes included in a development assurance standard (cf CONFIDENCE module): planning, development, validation, verification, configuration management, process assurance and certification liaison. However the assurance case only focuses on the development, validation and verification processes. The whole assurance case has been developed but there is no enough space in the paper to display all the flow diagrams, therefore we will detail only some aspects textually. The underneath subsections textually develop the assurance case from the the goals defined in figure 9.

A. ED-324/ARP6983 - [GOAL] The MLC requirements are a satisfactory refinement of the allocated system requirements

[SOLUTION]: The ACAS-Xu function is fully specified by the standardized lookup tables developed in the EUROCAE WG 75.1/RTCA SC-147 MOPS for ACAS-Xu. Therefore, there is no need for any other MLC requirements except some operational requirements (timing, similarity property, target definition). It includes the partitioning of the input space into 5 ODDs to accommodate the 5 previous advisories and the definition of the similarity property.

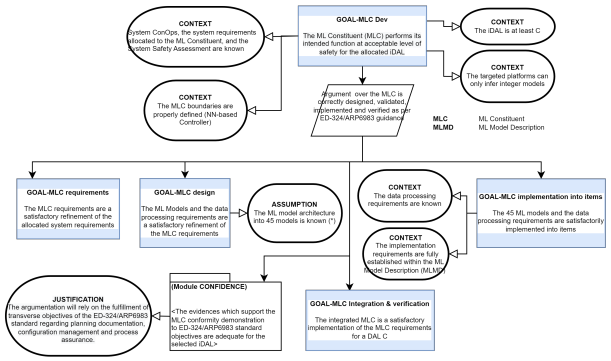

Fig. 9. Upper-level assurance case

B. ED-324/ARP6983 - [GOAL] The ML Models and the data processing requirements are a satisfactory refinement of the MLC requirements

[STRATEGY]: Argument over the datasets and the design models satisfy the ED-324/ARP6983 objectives. For any reference to initial development, refer to [3].

1) *[SUBGOAL] The ML datasets comply and are traceable to the MLC requirements* - [SOLUTION]: The 5 datasets are extracted from the standardized lookup tables developed in the EUROCAE WG 75.1/RTCA SC-147 MOPS for ACAS-Xu.
2) *[SUBGOAL] The 5 ML models comply with the MLC requirements, generalize out-of-sample, are stable and robust within their ODD* - [SOLUTION]: The models are quantized from the existing float models due to implementation constraints (targets platforms are integer-based). The compliance to MLC requirements is met by formally verifying that the similarity property holds. Wherever the quantized models (defined through their MLMD) do not hold the similarity property, the related part of the ODD is captured as a derived requirement and passed up to the system engineering level for safety net mitigation.
3) *[SUBGOAL] The ML data processing complies with the MLC requirements and ML Model architecture, and is traceable from the ML data processing description* - [SOLUTION]: The ML data processing code is developed from the MLC logical architecture (5 ML models in parallel) and the MLC requirements.
4) *[SUBGOAL] The 5 MLMDs are traceable to the 5 quantized ML models* - [SOLUTION]: Each MLMD is defined using the future ONNX extended format that expresses the full semantics of the model.

C. ED-324/ARP6983 - [GOAL] The 5 ML models and the data processing requirements are satisfactorily implemented into items

[STRATEGY]: Argument over the MLC is deployed onto traditional SW/HW item(s)

1) *[SUBGOAL] The MLC architecture into MLMIDs is developed wrt the ED 324/ARP6983 objectives* - [SO-

LUTION]: Each of the 5 models is deployed onto 2 SW items (CPU+Accelerator). U(MLMIDs) = MLMD is easily demonstrated using the description of data/control flow between items.

2) *[SUBGOAL] The implementation of MLMIDs, the ML Constituent architecture requirements and the refined ML Constituent implementation requirements complies with applicable item level standard* - [SOLUTION]: The 5 models are manually coded from the specification of their semantics (extended ONNX format) using target platform libraries. The ED-12C/DO-178C guidance is used to demonstrate that the items development does not introduce any unacceptable error for DAL C software.

D. ED-324/ARP6983 - [GOAL] The integrated MLC is a satisfactory implementation of the MLC requirements for a DAL C

[STRATEGY]: Argument over the MLC integration and verification activities satisfy the ED-324/ARP6983 guidance

1) *[SUBGOAL] ML and Traditional Item Implementations are produced and loaded onto the target platform for ML Constituent integration and verification* - [SOLUTION]: The 5 ML models and pre/post processing code are integrated on the selected platforms (TI Jacinto, nVIDIA Xavier).
2) *[SUBGOAL] ML Training and Target Environment differences are identified and assessed for their impact on stability and generalization* - [SOLUTION]: The exact replication is supported by the verification that the predictions are binary identical (checked on the whole input space).
3) *[SUBGOAL] ML Constituent performance is verified* - [SOLUTION]: Verification credit can be sought for formal verification activity performed during design phase.
4) *[SUBGOAL] Integrated ML Constituent complies and is robust with MLC reqs* - [SOLUTION]: Operational requirements verification is in progress.

VII. Conclusions

In this paper, we managed to optimize pre-designed ML models in float format (from [3]) into models that we can embed in integer-based targets. The ONNX format of these models have been formally verified and exactly implemented onto 2 target platforms (TI-Jacinto and nVDIA Xavier). At last we have demonstrated the compliance with the main development assurance objectives of ML future avionic ML standard ED-324/ARP693, making the ACAS-Xu system implementation certifiable.

It shall be noted that the exact replication is possible only when the ML model semantics and dynamics are completely and unambiguously specifiable. This makes the achievement of such a MLMD format an enabler of such technique.

If manual development is possible for this kind of non-complex model development, there is a need to automate the demonstration of the exact replication of the reference implementation in order to scale up the method to complex models. The MLIR/iree tools suite sounds a very promising technique for this purpose.

As a generalization of the ACAS-Xu case study, the certification of such ML surrogate modelling technique seems doable using the ED-324/ARP6983 guidance. Indeed, the use of such technique to approximate a function that can be specified with physical equations (or any other Oracle like ACAS-Xu lookup tables), makes the demonstration of conformity to ED-324/ARP6983 objectives very attainable.

Going further in the context of the use of surrogate modelling technique, the effort to comply with ARP6983/ED-324 objectives is significantly reduced for a DAL C development. With the same level of effort, it may be contemplated to comply with upper DALs.

VIII. Acknowledgment

This work has been supported by the French government under the "France 2030" program, as part of the SystemX Technological Research Institute, and as part of the DeepGreen project with grant ANR-23-DEGR-0001

References

[1] S. CHICHIN, D. PORTES, M. BRUNDLER, and V. JEGU. Capability to embed deep neural networks: Study on cpu processor in avionics context. In *ERTS 2020*, 2020.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[3] M. Damour, F. de Grancey, C. Gabreau, A. Gauffriau, and J.-B. Ginestet. Towards certification of a reduced footprint acas-xu system: a hybrid ml-based solution. In *Computer Safety, Reliability, and Security 40th International Conference SAFECOMP 2021*, pages pp.34–48, 2021, 978–3–030–83903–1. https://hal.archives-ouvertes.fr/hal-03355299v2), 2021.

[4] T. DeWolf, P. Jaworski, and C. Eliasmith. Neurorobotics nengo and low-power ai hardware for robust, embedded neurorobotics. Frontiers in Neurorobotics (2020).

[5] S. Durand, A. Lemesle, Z. Chihani, C. Urban, and F. Terrier. ReCIPH: Relational Coefficients for Input Partitioning Heuristic. 1st Workshop on Formal Verification of Machine Learning (WFVML 2022), July 2022.

[6] M. C. Edouard Yvinec, Arnaud Dapogny and K. Bailly. Rex: Data-free residual quantization error expansion. arXiv preprint arXiv:2203.14645, 2022.

[7] C. Gabreau, A. Gauffriau, F. de Grancey, J.-B. Ginestet, and C. Pagetti. Toward the certification of safety-related systems using ml techniques: the acas-xu experience. In *ERTS 2022*, page Session Th.4.C Assurance & Certification, 2022.

[8] C. Gabreau, A. Gauffriau, M.-C. Teulières, D. Marandas, C. Pagetti, and C. Maxim). Implementation using ed-xxx/arp6983: the acas xu experience. In *CTIC 2023*, 2023.

[9] F. Geissler, S. Qutub, and S. Roychowdhury. Towards a safety case for hardware fault tolerance in convolutional neural networks using activation range supervision.

[10] E. Goubault and S. Putot. A zonotopic framework for functional abstractions, 2009.

[11] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017.

[12] A. Lemesle. PyRAT Analyzer website. https://pyrat-analyzer.com/. Accessed: March 18th, 2023.

[13] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. Compcert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 2016.

[14] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications.

[15] e. a. Markus Nagel, Mart van Baalen. Data-free quantization through weight equalization and bias correction. In *ICCV, pp. 1325–1334, 2019*, page 1325–1334, 2019.

[16] J. Perez-Cerrolaza and al. Artificial intelligence for safety-critical systems in industrial and transportation domains: A survey. ACM Computing Surveys https://dl.acm.org/doi/10.1145/3626314, 2023.

[17] H. Pompougnac, U. Beaugnon, A. Cohen, and D. P. Butucaru. Weaving synchronous reactions into the fabric of ssa-form compilers. *ACM Trans. Archit. Code Optim.*, 19(2), mar 2022.

[18] F. Rastello and F. Bouchez Tichadou (eds.). *SSA-based Compiler Design*. Springer, 2022.

[19] SAE/EUROCAE. Aerospace Recommended Practices ARP4754B/ED-79B- development of civil aircraft and systems, 2023.