Towards Verified Computation Graphs for Safety-Critical ML: From Static Model to Execution Semantics

João Machado[1,2,*], Ricardo Silva[1,2,*], João Galego[2,3], Eric Jenn[4], Hugo Daniel Macedo[1,*] and Jorge Sousa Pinto[1,*]

[1]*HASLab/INESC TEC & Universidade do Minho, Portugal,*
[2]*Critical Software, Portugal,*
[3]*ISEG Executive Education, Portugal*
[4]*IRT Saint-Exupéry, Toulouse, France*

Abstract

Although the integration of machine learning into safety-critical systems is growing, a fundamental tension remains: performance-oriented techniques such as parallelism and optimized execution can introduce opacity and nondeterminism, while safety-critical systems demand transparency and reproducibility. This paper provides a formalization of ONNX computation graphs that addresses these requirements by focusing on the abstract modeling and verification of graph-based machine learning representations. The proposed workflow, developed within the SONNX initiative, combines the exploration and validation of structural properties of the graph using the Alloy Analyzer — used to explore and validate structural properties via model checking — with a subsequent refinement of the model in the Why3 verifier, enabling deductive reasoning over type invariants and functional properties. We further introduce an execution semantics for the graph model, defined as a state machine where graph configurations correspond to states and node execution induces state transitions. Overall, our proposed model is thus defined as a set of well-formed graph configurations equipped with a single-rule, transition relation.

Keywords

Formal Methods, Machine Learning, Model Checker, Deductive Verification

1. Introduction

The adoption of machine learning (ML) components in safety-critical systems is increasing across domains such as avionics, automotive, and industrial automation. In these contexts, certification requires not only functional performance, but also strong guarantees of determinism, traceability, and correctness. While considerable effort has been devoted to verifying properties of trained neural networks, less attention has been given to the correctness of the execution infrastructure that evaluates such models.

The Open Neural Network Exchange[1] (ONNX) format has become a widely adopted standard for representing ML models as directed acyclic computation graphs of operators consuming and producing tensors. Although widely adopted in the AI community, the lack of formal verification tools has limited the use of ONNX in safety-critical applications, which lead to the creation of a working group with the goal of providing a safe-oriented profile built upon ONNX, the SONNX initiative [1].

Our previous work within the SONNX initiative [2] outlined a complete pipeline targeting ONNX operators, specifying them at both abstract and concrete levels in WhyML and ultimately culminating in a reference C implementation. However, operator correctness alone is insufficient to characterise the behaviour of complete models, because the ONNX format represents machine learning models

8th International Workshop on Artificial Intelligence and fOrmal VERification, Logic, Automata, and sYnthesis. OVERLAY 2026 @ FLoC, July 18–19, Lisbon, Portugal

*Corresponding author.
joao.costa.machado@criticalsoftware.com (J. Machado); ricardo.moreira.silva@criticalsoftware.com (R. Silva); joao.galego@criticalsoftware.com (J. Galego); eric.jenn@irt-saintexupery.com (E. Jenn); hdm@di.uminho.pt (H. D. Macedo); jsp@di.uminho.pt (J. S. Pinto)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]https://onnx.ai/

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-09

CEUR-WS.org/Vol-4324/paper4.pdf

as directed acyclic computational graphs that have a runtime semantics beyond the correctness of individual operators.

This paper addresses that gap by proposing a formal model of ONNX computation graphs together with an operational semantics for graph execution. Graph configurations are represented as states, and execution is defined as a transition relation in which any enabled computation node may fire. This captures the data-flow parallelism inherent in ONNX while preserving deterministic outcomes under dependency constraints.

Our approach is mechanised in a stepwise manner using complementary formal methods tools. Alloy [3] is used to validate structural constraints and generate bounded graph instances, while Why3 [4] is used to encode the model, specify dynamic invariants following the approach in [5], and discharge verification conditions through automated theorem provers.

In related works, the verification of neural networks has been an active research area. Liu et al. [6] provide a comprehensive survey of verification algorithms for deep neural networks, and Huang and colleagues [7] survey safety and trustworthiness aspects including verification, testing, and adversarial robustness. Zhang et al. [8] survey robustness assessment techniques for image recognition networks. These approaches, which rely primarily on constraint solving, abstract interpretation, and bound propagation, verify properties of trained networks (such as robustness to adversarial perturbations) but do not employ deductive verification. Our work is complementary to network-level verification, which assumes correct implementations – our methodology aims to provide them using deductive verification techniques.

The main contributions of this paper are: A mathematical formalisation of ONNX computation graphs and their well-formedness conditions; A state-machine execution semantics for graph evaluation; A collection of inductive invariants capturing execution safety properties; A mechanised verification workflow combining Alloy and Why3.

The remainder of the paper is organised as follows. Section 2 presents the graph model, execution semantics, and invariants. Section 3 describes the mechanisation in Alloy and Why3. Section 4 concludes and outlines future work.

2. Formalization of ONNX graphs

We define a formal model of ONNX graphs as a mathematical structure, provide a model of computation as a state machine between graph configurations, and formalize its mathematical properties as a set of inductive invariants that assert the safety of the graph execution.

The ONNX format represents machine learning models as directed acyclic computational graphs in which edges denote the flow of tensors between nodes, and the nodes, can be divided into three categories: *graph inputs*, which feed the externally provided input tensors to the model as source vertices; *graph outputs*, which correspond to the resulting tensors computed by the model; and *computation nodes*, that perform the intermediate computations by consuming input and producing output tensors.

To aid in the understanding of the following formalization, Figure 1 shows a simple ONNX network representing a logistic regression model. It is composed of three *graph inputs* (X, W and B), three *computation nodes* ($MatMul$, Add and $Sigmoid$), and one *graph output* (y_{out}).

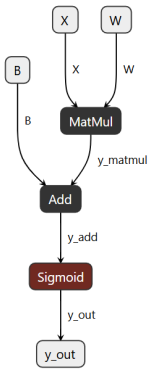


Figure 1: Example ONNX graph

2.1. Graph definitions

We define graphs as a mathematical structure where vertices are either input/output tensors or operators, and edges represent the tensor input/output relationships between them. The graph structure is subject to a set of well-formedness conditions, which are used in the mechanization of the theory to type-check the graph and ensure its correctness.

Definition 1 (Tensor). *A* tensor *is an element of an abstract set $\mathcal{T}$. We define the* lifted *set $\mathcal{T}_\perp = \mathcal{T} \cup \{\perp\}$, where $\perp$ denotes an undefined tensor value.*

Definition 2 (Tensor Node). *A* tensor node *is a pair $\langle id, val \rangle$, where $id \in \mathbb{Z}$ is a unique node identifier and $val \in \mathcal{T}_\perp$.*

Definition 3 (Operator). *An* operator *is an element of an abstract set $\mathcal{O}$ (e.g.* Add, Relu, Conv, *...). Each operator $k \in \mathcal{O}$ is equipped with:*

(a) a semantics function $[\![k]\!] : \mathcal{T}^* \to \mathcal{T}^*$, *mapping an ordered list of input tensor values to an ordered list of output tensor values;*

(b) a constraint predicate $\mathrm{cst}_k(\cdot) : \mathcal{T}^* \to \mathbb{B}$, *expressing specific requirements on the inputs in order for the operator to execute; and*

(c) the functions `n_inputs` *and* `n_outputs` *that return, respectively, the number of inputs consumed and outputs produced by a given operator.*

Definition 4 (Computational Nodes). *A* computation node *is a tuple $\langle id,\ inp,\ out,\ op \rangle$ where $id \in \mathbb{Z}$ is a unique identifier, $inp \in \mathbb{Z}^*$ is a list of* input *tensor node identifiers, $out \in \mathbb{Z}^*$ is a list of* output *tensor node identifiers, and $op \in \mathcal{O}$ is the operation performed by this node.*

Computation nodes well-formedness is defined as a `type invariant` enforcing compliance with the input and output arities of the the associated operator:

(WF1) **Input and output arity.** $|c.inp| = \mathrm{n_inputs}(c.op) \ \wedge \ |c.out| = \mathrm{n_outputs}(c.op)$

Definition 5 (Graph). *A graph is a tuple $G = \langle I,\ O,\ C,\ T \rangle$ where: I is the set of graph inputs; O is the set of graph outputs; C is the set of graph computation nodes; and T is the set of graph intermediate tensor nodes. T, I and O are, respectively, defined by:*

$$T = \bigcup \{ j \in \mathcal{T}_\perp \mid \exists c_1, c_2 \in C : \ \ j.id \in c_1.inp \wedge j.id \in c_2.out \}$$

$$I = \bigcup \{ j \in \mathcal{T}_\perp \mid \exists c \in C : \ \ j.id \in c.inp \wedge j \notin T \} \qquad O = \bigcup \{ j \in \mathcal{T}_\perp \mid \exists c \in C : \ \ j.id \in c.out \wedge j \notin T \}$$

The following notation is used throughout. Let:

1. $\mathrm{all}(G) = I \cup O \cup T$ be the collection of all tensor nodes of G;
2. For any set of nodes L, $\mathrm{ids}(L) = \bigcup \{ l.id \mid l \in L \}$ denotes the set of identifiers of L
3. For $c \in C, \mathrm{out}(c) = \bigcup \{ t.val \mid t.id \in c.out \ \wedge t \in \mathrm{all}(G) \}$ denotes the set c output tensors;
4. The set of tensor node ids that are, input and output of some computation node are defined, respectively, by:

$$\mathrm{inTids}(C) = \bigcup \{ c.inp \mid c \in C \} \qquad \textit{and} \qquad \mathrm{outTids}(C) = \bigcup \{ c.out \mid c \in C \}.$$

Definition 6 (Well-formed Graph). *A graph G is* well-formed *if the following conditions in the form of type invariants hold:*

(WF2) ***Unique tensor-node identifiers.*** $|ids(all(G))| = |I| + |O| + |T|$

(WF3) ***Unique computation-node identifiers.*** $|ids(C)| = |C|$

(WF4) ***Input closure.*** $inTids(C) \subseteq ids(I \cup T)$.

(WF5) ***Output closure.*** $outTids(C) \subseteq ids(O \cup T)$.

(WF6) ***Unique producer.*** $\forall j \in outTids(C) : |\bigcup \{c \mid j \in c.out \wedge c \in C\}| = 1$

(WF7) ***Tensor coverage.*** $ids(all(G)) \subseteq inTids(C) \cup outTids(C)$.

(WF8) ***Output tensor-node condition.*** $\forall j \in ids(all(G))$: $j \notin inTids(C) \iff j \in \mathrm{ids}(O)$.

(WF9) ***Input tensor-node condition.*** $\forall j \in ids(all(G))$: $j \in \mathrm{outTids}(C) \iff j \in \mathrm{ids}(I)$.

(WF10) ***Valued graph inputs.*** $\forall t \in I, \quad t.val \neq \bot$.

(WF11) ***Atomic output production*** $\forall c \in C : \ out(c) \cap \bot \in \{\emptyset, \ out(c)\}$.

Conditions (WF7), (WF8), and (WF9) together partition the tensor nodes of G into three disjoint classes. A tensor node is a graph input *if and only if it is not produced by any computation node (its value is externally provided, and by (WF10) it is always defined). It is a* graph output *if and only if it is not consumed by any computation node. Every remaining tensor node is an* intermediate *node, both produced and consumed by computation nodes; these are exactly the elements of T.*

Definition 7 (Edge). *Let G be a well-formed graph. A vertex is an element of an abstract set $\mathbb{V}_G = G.I \cup G.O \cup G.C$. The edge relation $\rightarrow_G \subseteq \mathbb{V}_G \times \mathbb{V}_G$ can be defined as:*

$$\begin{aligned} g_i \rightarrow_G c \quad &\equiv \quad g_i \in G.I \wedge c \in G.C \wedge g_i.id \in c.inp \\ c_1 \rightarrow_G c_2 \quad &\equiv \quad c_1 \in G.C \wedge c_2 \in G.C \wedge (\exists id : id \in c_1.out \ \wedge \ id \in c_2.inp) \\ c \rightarrow_G g_o \quad &\equiv \quad c \in G.C \wedge g_o \in G.O \wedge g_o.id \in c.out \end{aligned}$$

Definition 8 (Path). *Let G be a well-formed graph. Exists a path from vertex v to vertex v' if exists any list of vertices $\pi = [v_0, ..., v_n]$ such that $v_i \in \mathbb{V}_G$ and:*

$$v = v_0 \wedge v' = v_n \wedge (\forall i : 0 \leq i < n \implies v_i \rightarrow_G v_{i+1})$$

To denote that v and v' are connected by a path we write $v \twoheadrightarrow_G v'$.

Definition 9 (ONNX Graph). *An* ONNX graph *is a well-formed graph that satisfies the following additional constraints:*

(WF11) ***Acyclicity.*** $\forall v \in \mathbb{V}_G : \neg \ (v \twoheadrightarrow_G v \ \wedge \ \twoheadrightarrow_G \ \neq [\,])$

(WF12) ***Reachability.*** $\forall v \in (I \cup C) : (\exists v_{out} \in O : v \twoheadrightarrow_G v_{out})$

We write Σ for the set of all well-formed ONNX graphs.

2.2. Execution semantics

The dynamic behavior of the computation of an ONNX model is given as a state-machine over configurations corresponding to the state of the tensors in the associated graph. The initial state of the graph is defined by the assignment of values to the graph inputs' tensors and the absence of values for the output and intermediate tensor nodes. A state transition is induced by non-deterministically selecting an executable computational node - a node with all outputs tensors set as undefined, all input tensors assigned, which satisfy all the operator's constraints. State transitions entail a transition relation, and its transitive closure defines the graph trace associated with the model execution.

Definition 10 (Initial state)**.** *A well-formed graph $G \in \Sigma$ is an* initial state *if every tensor node, which is not a graph input, is undefined:*

$$\forall t \in G.O \cup G.T, \quad t.val = \bot.$$

We write $\Sigma_0 \subseteq \Sigma$ for the set of all initial states.

Notation Let G be a well-formed graph where $c \in G.C$, let:

1. $\text{inNodes}(c,\, G) \;=\; \big[\, t \;\big|\; t.id \in c.inp \;\;\wedge\;\; t \in G.I \cup G.T \,\big]$ denote a list containing all tensor nodes which are input of c;
2. $\text{outNodes}(c,\, G) \;=\; \big[\, t \;\big|\; t.id \in c.out \;\wedge\; t \in G.O \cup G.T \,\big]$ denote the dual of `inNodes`;
3. $\text{vals}(l) \;=\; \big[\, t.val \;\big|\; t \in l \,\big]$ be the list of values of each tensor node in the list l.

We assume each of the lists preserves the order of $c.inp$ (resp. $c.out$). The existence of all referenced tensor nodes in the appropriate sublists is guaranteed by *(WF4)* and *(WF5)*.

4. $\text{ready}(l) \;=\; \forall t \in l, t.val \neq \bot$, be a predicate checking all tensor nodes in l are assigned;
5. $\text{unexec}(l) \;=\; \forall t \in l, t.val = \bot$, be the dual of `ready` (checking all tensors are undefined).

Definition 11 (Executability)**.** *A computation node c is* executable *in a graph $G \in \Sigma$, written* $\text{isExec}(c,\, G)$*, if and if only:*

(i) $c \in G.C$ *- ensures that only computation nodes of the graph may fire;*

(ii) $\text{ready}(\text{inNodes}(c,\, G))$ *- all inputs must already be computed;*

(iii) $\text{unexec}(\text{outNodes}(c,\, G))$ *- prevents a node from firing more than once; and*

(iv) $\text{cst}_{op}(\text{vals}(\text{inNodes}(c,\, G)))$ *- checks that the input tensors satisfy the operator's constraints.*

Definition 12 (Execution function)**.** *Let $G \in \Sigma$ and let c be a computation node that is executable and let:*

1. $vs \;=\; [\![c.op]\!]\big(\text{vals}(\text{inNodes}(c,\, G))\big)$ *be the list of output tensors returned by the operator*
2. $\text{zip}(tn,\, t) = \big[\, \langle tn_i.id,\, t_i\rangle \mid 0 \leq i < |tn| \wedge |tn| = |t| \,\big]$ *pairs the i-th tensor node id ($tn.id$) with the i-th tensor (t)*
3. $newNodes \;=\; zip\big(\text{outNodes}(c,\, G),\; vs\big)$ *be an updated output tensor nodes list, after execution*

Both vs and $outNodes(c, G)$ share the same length due to computational node type invariant (WF1) and graph type invariant (WF5).

4. $\text{upd}(v,\, l) = [\, \langle v' \text{ if } l'.id = v'.id \text{ else } l' \mid l' \in \ell \;\wedge v' \in v, \,]$ *be the function that updates a tensor node list based on the outputs produced by an operator execution*

The execution *of c in G is then defined by:*

$$\text{exec}(c,\, G) \;=\; \Big\langle\, G.I; \quad \text{upd}(newNodes,\, G.O), \quad G.C, \quad \text{upd}(newNodes,\, G.T) \,\Big\rangle$$

The only structures that are modified during execution are the intermediate tensor nodes, $G.T$, and the graph outputs, $G.O$. Output tensor nodes of some computation node can belong either to the set of intermediate tensors or to the graph outputs; the simultaneous update on both these structures accounts for both cases. Moreover, the number of nodes with an assigned value strictly monotonically increases throughout graph execution. The graph execution semantics can then be defined in terms of a state machine specification.

Definition 13 (Single-step execution relation)**.** *The* single-step execution relation $\rightsquigarrow \subseteq \Sigma \times \Sigma$ *is a state transition relation, defined by the following inference rule:*

$$\frac{\text{isExec}(c,\, G) \qquad G' = \text{exec}(c,\, G)}{G \;\rightsquigarrow\; G'}\ \textsc{Execute}$$

That is, $G \rightsquigarrow G'$ *if and if only there exists a computation node* c *that is executable in* G *(Definition 11) and whose execution (Definition 12) yields* G'*.*

The non-determinism is in the choice of c: when several computation nodes are simultaneously executable in G, rule Execute may be applied with any one of them, yielding (possibly) different successor states. Each such choice models a different firing order of data-flow operations. From a practical perspective, however, any concrete execution algorithm that conforms to this semantics must be deterministic: the apparent non-determinism arises only when executable nodes are independent, that is, when they do not share data dependencies. In such cases, the order in which these nodes are executed does not affect the final outcome of the computation, since their executions commute.

Definition 14 (Transitive closure execution relation)**.** *The* multi-step execution relation $\rightsquigarrow^* \subseteq \Sigma \times \Sigma$ *is the reflexive-transitive closure of* $\rightsquigarrow$*, defined by the following two rules:*

$$\frac{}{G \;\rightsquigarrow^*\; G}\ \textsc{Refl} \qquad\qquad \frac{G_1 \;\rightsquigarrow\; G_2 \qquad G_2 \;\rightsquigarrow^*\; G_3}{G_1 \;\rightsquigarrow^*\; G_3}\ \textsc{Trans}$$

Definition 15 (Execution state machine)**.** *The graph execution is described by the state machine specification* $\langle \Sigma,\ \Sigma_0,\ \rightsquigarrow \rangle$*, where* Σ *is the set of all well-formed graphs (Definition 5),* Σ_0 *is the set of initial states (Definition 10), and* $\rightsquigarrow$ *is the single-step execution relation (Definition 13).*

An *execution* of the state machine is any sequence of transition steps $G_0 \rightsquigarrow G_1 \rightsquigarrow \cdots$ starting from some $G_0 \in \Sigma_0$. A *terminal* state is a graph $G \in \Sigma$ from which no further step is possible, i.e. no computation node is executable in G. Because each application of Execute strictly reduces the number of undefined tensor nodes, and the graph is finite, every execution is finite and reaches a terminal state.

2.3. Inductive invariants

Given the state machine specification of the graph execution, that ensures static well-formedness, we can now define and prove dynamic safety properties that hold in all reachable configurations. The safety properties are expressed as inductive invariants following the approach in [5], which are predicates that hold for all initial states and are preserved by the execution steps.

Definition 16 (Inductive Invariant)**.** *For every predicate* Φ *over any valid state machine configuration* $\langle \Sigma,\ \Sigma_0,\ \rightsquigarrow \rangle$*,* Φ *is said to be an inductive invariant if:*

(i) Φ *holds for every* $w_0 \in \Sigma_0$

(ii) $\forall w, w' \in \Sigma : w \rightsquigarrow w' \wedge \Phi(w) \implies \Phi(w')$

A fundamental property with respect to the graph execution is ensuring that a node can only trigger if and if only, all of its predecessors (dependencies) have already been computed. Let $w \in \Sigma$ be a valid graph state and let:

1. $\text{hasExec}(c,\, w) = \text{ready}(\text{outNodes}(c,\, w))$, denote the predicate stating that computational node c has already been executed, i.e., all of its outputs have been defined.

(II1) **Computed Dependencies.** $\forall c, p \in w.C : \text{isExec}(c,\, w) \implies (p \twoheadrightarrow_w c \implies \text{hasExec}(p,\, w)))$

This invariant captures a fundamental safety property of the execution model: a computation node may fire only after all of its dependencies have been successfully computed. A natural dual, forward-looking reading of this property is equally informative, namely that whenever some predecessor of a computation node c remains unexecuted, c itself cannot yet be regarded as ready to fire. The equivalence between the two formulations follows from a short chain of purely logical manipulations.

Let $w \in \Sigma$ in:

$$
\begin{aligned}
& \forall c, c' \in w.C.\ c' \to_w c \implies \big(\mathrm{isExec}(c,\, w) \implies \mathrm{hasExec}(c',\, w)\big) \\
\equiv\ & \text{(currying)} \\
& \forall c, c' \in w.C.\ \big((c' \to_w c) \wedge \mathrm{isExec}(c,\, w)\big) \implies \mathrm{hasExec}(c',\, w) \\
\equiv\ & \text{(material implication)} \\
& \forall c, c' \in w.C.\ \neg\big((c' \to_w c) \wedge \mathrm{isExec}(c,\, w)\big) \vee \mathrm{hasExec}(c',\, w) \\
\equiv\ & \text{(De Morgan's law)} \\
& \forall c, c' \in w.C.\ \big(\neg(c' \to_w c) \vee \neg\mathrm{isExec}(c,\, w)\big) \vee \mathrm{hasExec}(c',\, w) \\
\equiv\ & \text{(assoc./comm.)} \\
& \forall c, c' \in w.C.\ (\neg(c' \to_w c) \vee \mathrm{hasExec}(c',\, w)) \vee \neg\mathrm{isExec}(c,\, w) \\
\equiv\ & \text{(De Morgan's law)} \\
& \forall c, c' \in w.C.\ \neg((c' \to_w c) \wedge \neg\mathrm{hasExec}(c',\, w))) \vee \neg\mathrm{isExec}(c,\, w) \\
\equiv\ & \text{(material implication)} \\
& \forall c, c' \in w.C.\ ((c' \to_w c) \wedge \neg\mathrm{hasExec}(c',\, w))) \implies \neg\mathrm{isExec}(c,\, w)
\end{aligned}
$$

Resulting in the following invariant:

(II2) **Successors Undefined.** $\forall c, p \in w.C : (p \twoheadrightarrow_w c \ \wedge\ \neg\mathrm{hasExec}(p,\, w)) \implies \neg isExec(c)$

which essentially states that if a given node is not yet computed then neither of its successors are ready for execution.

Invariant *(II1)* is non-trivial. For proof purposes, it is useful to decomposite into auxiliar invariants:

(II3) **Input Defined.** $\forall c \in w.C : \mathrm{hasExec}(c,\, w) \implies ready(inNodes(c, w))$

(II4) **Parents Computed 1.** $\forall c, p \in w.C : (p \to_w c \ \wedge\ isExec(c)) \implies \mathrm{hasExec}(p,\, w)$

(II5) **Parents Computed 2.** $\forall c, p \in w.C : (p \to_w c \wedge\ \mathrm{hasExec}(c,\, w)) \implies \mathrm{hasExec}(p,\, w)$

Together with the inductive hypothesis, invariants *(II4)* and *(II5)* suffice to establish the target property *(II1)*. Invariant *(II3)* is additionally useful as a supporting lemma in the proof of *(II5)*.

3. Mechanization and tooling

Our formalization of the ONNX graph model and its execution semantics was mechanized with the aim of providing a machine-checked specification and verifying the associated safety properties. The approach was stepwise, starting with an abstract model in Alloy, which was then refined and fully formalized in Why3. The choice of these two frameworks was motivated by their complementary strengths in terms of modelling and verification capabilities.

Alloy [3] is a model checker built on a first-order relational logic well-suited to specifying both the structural and the behavioural aspects of a system. Given a bounded scope, the Alloy Analyzer compiles the specification into SAT and exhaustively searches for instances or counter-examples, displaying them graphically.

Why3 [4] is a platform for deductive program verification centred on the WhyML specification and programming language. It allows developers to annotate programs with preconditions, postconditions and invariants, from which the platform automatically generates the corresponding verification conditions (VCs). These VCs are then discharged through a back-end of automated theorem provers and SMT solvers, with interactive proof assistants available as a fallback whenever full automation is not achievable.

While Alloy supported structural reasoning, contributing to earlier error detection, Why3 came as natural choice given our previous work [2] and current goal of developing a full fledged verified reference runtime for ONNX models.

Why3 formalization All entities introduced in Section 2 are modelled as records, each faithfully reflecting its corresponding mathematical definition. Whenever an entity is required to satisfy specific safety properties, those conditions are encoded as `type invariants`, ensuring that the properties are preserved throughout graph construction and execution. A `type invariant` is a logical condition that every instance of a given type must satisfy by construction.

Tensors are represented by the polymorphic type `tensor 'a`, encoded as a tuple comprising a list of `dimensions`, an `infinite map`, and a `background value` to which every invalid coordinate is mapped. The inputs and outputs of a computation node are represented as lists, reflecting the fact that tensors are consumed and produced in a non-arbitrary order. A detailed account of tensors and operators, together with their formalization, has been presented in our previous work [2].

To execute the operators carried by the computation nodes, the data type of the tensors must be fixed and every admissible variant must be wrapped in a single, uniform type, since the operations performed at the computation nodes are, in general, not polymorphic. To this end, the wrapper type `wrapper_tensor` makes the distinction between integer and real tensors explicit through the constructors `I tensor_int` and `R tensor_real`, while preserving a single homogeneous type at the level of the graph. Tensors are then encapsulated in `optional structures`, which allows distinguishing between computed tensors (assigned a value) and uncomputed ones (left unassigned).

The `graph` type is modelled using lists rather than maps, with the goal of simplifying the proofs of its invariants. Although maps would arguably provide more efficient access to elements by identifier, lists — being a simple inductive type — allow most invariants to be discharged by structural induction with little to no manual intervention. Invariants are central to the graph model : they encode the properties that must hold for a graph to be considered well-formed. To express them, we rely exclusively on *predicates* and *pure functions*. This choice is particularly important from the perspective of proof automation: since these definitions are purely logical and side-effect-free, the underlying SMT solvers can unfold them transparently during verification, thereby easing the discharge of any obligation that depends on such invariants.

The notion of *path* is defined by means of an inductive predicate, yielding a natural and compositional definition. Inductive definitions are particularly well-suited for this kind of property, since the relation "there exists a path from v to v'" is intrinsically given by the least fixed point of a set of construction rules: a path is either empty (base case `Path_empty`) or extends an edge with a further path (inductive case `Path_cons`).

Execution is specified in WhyML by a direct transcription of the predicates and functions of Section 2.2. Auxiliary `lemmas` are nevertheless required to discharge the associated proof obligations, since SMT solvers do not automatically derive that execution does not introduce any structural change, only the tensors produced by the firing computation node are affected. The accompanying repository (https://github.com/CSW-RS/OVERLAY26-SONNX) contains the complete Why3 developments described above.

4. Conclusion

This paper presented a formal foundation for the verification of ONNX computation graphs. We defined a mathematical model capturing the graph structure, its execution behaviour, through a state-machine semantics in which graph evaluation proceeds through enabled node transitions. The model has been analysed using Alloy for bounded structural validation and a Why3 mechanization is underway.

The structural model and the execution semantics are already fully formalised in Why3, with all the corresponding well-formedness conditions and execution-related obligations discharged through automated solvers. The inductive invariants introduced in Section 2.3, in turn, are not yet completely covered in Why3, which are left for future work among two other goals: the strengthening of the refinement chain and extraction of executable C code, yielding a certifiable reference runtime for ONNX-based systems.

Declaration on Generative AI

The authors used ChatGPT and Claude in order to: Grammar and spelling check, Paraphrase and reword, Drafting content and Formatting assistance. The authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] E. Jenn, J. Souyris, H. Belfy, L. Correnson, M. Turki, N. Valot, F. Védrine, SONNX: Towards an ONNX profile for critical systems, in: European Congress of Embedded Real Time Systems (ERTS), 2026. ISSN 2680-0918.

[2] Loïc Correnson and Eric Jenn and Hugo Daniel Macedo and João Machado and Ricardo Silva and Jorge Sousa Pinto, Verified Tensor Operators for Safety-Critical ML: From Specification to Reference Implementation, 2026. URL: https://doi.org/10.1007/978-3-032-32357-6_2.

[3] D. Jackson, Alloy: A language and tool for exploring software designs, Commun. ACM 62 (2019) 66–76. URL: https://doi.org/10.1145/3338843. doi:10.1145/3338843.

[4] J.-C. Filliâtre, A. Paskevich, Why3 — where programs meet provers, in: Programming Languages and Systems (ESOP 2013), volume 7792 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 2013, pp. 125–128. doi:10.1007/978-3-642-37036-6_8.

[5] C. Belo Lourenço, J. Sousa Pinto, Auto-active verification of distributed systems and specification refinements with why3-do, Science of Computer Programming 247 (2026) 103352. doi:https://doi.org/10.1016/j.scico.2025.103352.

[6] C. Liu, T. Arnon, C. Lazarus, C. Strong, C. W. Barrett, M. J. Kochenderfer, Algorithms for verifying deep neural networks, Foundations and Trends in Optimization 4 (2021) 244–400. doi:10.1561/2400000035.

[7] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, X. Yi, A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability, Computer Science Review 37 (2020) 100270. doi:10.1016/j.cosrev.2020.100270.

[8] J. Zhang, J. Liu, Y. Dong, H. Su, A survey of neural network robustness assessment in image recognition, arXiv preprint arXiv:2404.08285 (2024). URL: https://arxiv.org/abs/2404.08285.