Let a Neural Network Be Your Invariant

Mirco Giacobbe*
University of Birmingham, UK
Zeroth Research, UK

Daniel Kroening*
Amazon Web Services, USA

Abhinandan Pal*
University of Birmingham, UK
National Institute of Informatics Tokyo, Japan

Michael Tautschnig*
Amazon Web Services, USA
Queen Mary University of London, UK

Abstract

Safety verification ensures that a system avoids undesired behaviour. Liveness complements safety, ensuring that the system also achieves its desired objectives. A complete specification of functional correctness must combine both safety and liveness. Proving with mathematical certainty that a system satisfies a safety property demands presenting an appropriate inductive invariant of the system, whereas proving liveness requires showing a measure of progress witnessed by a ranking function. Neural model checking has recently introduced a data-driven approach to the formal verification of reactive systems, albeit focusing on ranking functions and thus addressing liveness properties only. In this paper, we extend and generalise neural model checking to additionally encompass inductive invariants and thus safety properties as well. Given a system and a linear temporal logic specification of safety and liveness, our approach alternates a learning and a checking component towards the construction of a provably sound neural certificate. Our new method introduces a neural certificate architecture that jointly represents inductive invariants as proofs of safety, and ranking functions as proofs of liveness. Moreover, our new architecture is amenable to training using constraint solvers, accelerating prior neural model checking work otherwise based on gradient descent. We experimentally demonstrate that our method is orders of magnitude faster than the state-of-the-art model checkers on pure liveness and combined safety and liveness verification tasks written in SystemVerilog, while enabling the verification of richer properties than was previously possible for neural model checking.

1 Introduction

Model checking addresses the question of whether a reactive system $\mathcal{M}$ meets a specification of intended behaviour Φ [42]. Reactive systems are systems that continually respond to inputs and operate over infinite executions. Exemplars include hardware designs implementing processing units, communication and arbitration protocols, as well as digital controllers for automotive, aerospace, or medical applications, where safety is paramount. Safety properties require that *nothing bad ever happens*. A safety violation is a finite execution the last state of which falsifies Φ and the damage is irredeemable [77]. Safety properties are dual to liveness properties, which require that *something good eventually happens*. A liveness violation is evidenced by an infinite execution that falsifies Φ, as any finite violating prefix could in principle still be extended to a compliant execution [77]. Every linear-time specification is the composition of a safety and a liveness property [8], and linear temporal logic (LTL) provides an expressive language for specifying linear-time properties of reactive systems.

*The authors are listed alphabetically.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

https://doi.org/10.52202/085713-2509

SystemVerilog Assertions (SVA) bring linear-time logic into commercial hardware design flows [56], and electronic design automation vendors have spent decades optimising symbolic model checkers for them. Testing and simulation alone cannot fully guarantee that a system meets its specifications, and explicitly enumerating every possible behaviour is impractical, except for the most trivial hardware designs. Symbolic model checking has made it feasible to handle large state spaces. Two symbolic paradigms dominate. Algorithms based on binary decision diagrams (BDDs) provide sound and complete procedures for LTL model checking, yet BDDs often explode in size when faced with arithmetic data paths ubiquitous in modern hardware design [26, 52]. Bounded model checking (BMC) algorithms enjoy the much more scalable combinatorial and arithmetic reasoning of SAT solvers, but only explore the state space up to a finite depth, sacrificing completeness over unbounded execution [17]. Incremental algorithms for the construction of exploration frontiers (IC3) bridge that gap in unbounded safety verification [20], but fall short on liveness.

Neural model checking is an automated approach that leverages *neural certificates* to verify reactive systems against LTL specifications, but prior work has exclusively focused on liveness properties [63]. This prior cognate work learns ranking functions represented as neural networks—which serve as proof of liveness—from random executions and then checks their validity using symbolic reasoning, thereby providing formal guarantees. However, specifications that characterise pure safety or combinations of safety and liveness additionally demand the construction of inductive invariants—which serve as proofs of safety. Prior work does not construct such invariants, leaving safety out of reach.

In this paper, we demonstrate that neural networks are an effective representation for inductive invariants as well. Our novel method leverages neural certificates to simultaneously represent ranking functions and inductive invariants. Given a hardware design $\mathcal{M}$ and a non-deterministic Büchi automaton $\mathcal{A}_{\neg\Phi}$ that recognises the violations to an LTL formula Φ, we train a neural certificate on sample transitions of the synchronous composition $\mathcal{M} \parallel \mathcal{A}_{\neg\Phi}$. This network (i) classifies each state as inside or outside the invariant and (ii) ranks the states, within the invariant, to strictly decrease whenever an accepting state is encountered. We then pose a one-step BMC query to check if the neural certificate is true over the entire state space. On affirmation, this proves every accepting state is either (i) unreachable or (ii) visited at most finitely many times. This implies that no reachable behaviour of $\mathcal{M}$ satisfies $\neg\Phi$, thereby proving that every reachable behaviour of $\mathcal{M}$ satisfies Φ.

We implemented our method using a new neural ranking function architecture that enables training using constraint solvers, in contrast to prior work that exclusively relies on gradient descent for training [63]. Our new architecture yields more compact representations and it is much more efficient and numerically stable than prior work. We extended the prior benchmark with standard safety and combined liveness and safety verification problems, and have compared our method with the state-of-the-art hardware model checkers. Measuring against the per-task fastest on pure-liveness, our method is faster on $66\,\%$ of tasks, $10\times$ faster on $46\,\%$, $10^3\times$ on $11\,\%$, and $10^4\times$ on $4\,\%$. On pure safety tasks, while not outperforming, we match the best symbolic tools: $80\,\%$ of instances complete for all approaches in under $1\,$s. The picture changes dramatically for properties that demand *both* an inductive invariant and a ranking function: our method is faster than the leading model checkers on $61\,\%$ of tasks, running $10^2\times$ on $43\,\%$, $10^4\times$ on $27\,\%$, and $10^5\times$ on $6\,\%$.

Our contribution is threefold. First, by simultaneously representing inductive invariants and ranking functions using neural networks, we extend and generalise neural model checking to verify arbitrary LTL properties encompassing both safety and liveness. Second, by introducing a neural architecture amenable to training using constraint solvers, we achieve both superior runtime efficiency and numerical stability. Third, by evaluating our method across a 634-task benchmark suite, we demonstrate that our method outperforms the state of the art in automated formal verification.

2 Model Checking Safety and Liveness

We consider specifications Φ in LTL over a set of atomic propositions Π. LTL extends propositional logic with temporal operators: $\mathsf{X}\,\Phi$ meaning Φ holds at the next step, $\mathsf{F}\,\Phi$ meaning Φ eventually holds, $\mathsf{G}\,\Phi$ meaning Φ always holds, $\Phi_1 \,\mathsf{U}\, \Phi_2$ meaning Φ_1 holds at all times until Φ_2 necessarily holds at some future step, and $\Phi_1 \,\mathsf{W}\, \Phi_2$ meaning Φ_1 holds at all time before Φ_2 possibly occurs [102]. LTL naturally expresses safety, liveness, and their combinations for reactive systems. For $\Pi = \{\mathtt{a}, \mathtt{b}\}$, the formula $\mathsf{G}\ \mathtt{a}$ is a safety property: a single finite prefix whose last state violates $\mathtt{a}$ refutes it. By contrast, $\mathsf{F}\ \mathtt{b}$ is a liveness property: no finite execution suffices to disprove it, since $\mathtt{b}$ may still occur later. The formula $\mathtt{a}\,\mathsf{U}\,\mathtt{b}$ combines both: its safety component forbids any prefix containing $\neg\mathtt{a}\wedge\neg\mathtt{b}$ before the first $\mathtt{b}$, while its liveness component requires that $\mathtt{b}$ eventually occurs. Equivalently, $\mathtt{a}\,\mathsf{U}\,\mathtt{b} \;=\; \mathtt{a}\,\mathsf{W}\,\mathtt{b} \;\wedge\; \mathsf{F}\ \mathtt{b}$, where $\mathtt{a}\,\mathsf{W}\,\mathtt{b}$ is pure safety and $\mathsf{F}\ \mathtt{b}$ is pure liveness.

We model a reactive system $\mathcal{M}$ over a finite variable set $X_{\mathcal{M}}$ as a state transition system by an initial condition $\mathrm{Init}_{\mathcal{M}}$ and a sequential update relation $\mathrm{Update}_{\mathcal{M}}$. The variables are split into inputs $\mathrm{inp}\,X_{\mathcal{M}}$ and state-holding registers $\mathrm{reg}\,X_{\mathcal{M}}$. $\mathrm{Init}_{\mathcal{M}}$ is a first-order predicate over $\mathrm{reg}\,X_{\mathcal{M}}$ and $\mathrm{Update}_{\mathcal{M}}$ is a first-order predicate over $X_{\mathcal{M}} \cup \mathrm{reg}\,X'_{\mathcal{M}}$, where the set of primed variables denotes the next-cycle values. A reactive system $\mathcal{M}$ induces a transition system with state space S of valuations of $X_{\mathcal{M}}$, initial states $S_0 \subseteq S$ with $\boldsymbol{s} \in S_0 \iff \mathrm{Init}_{\mathcal{M}}(\mathrm{reg}\,\boldsymbol{s})$, and transition relation $\rightarrow_{\mathcal{M}} \subseteq S \times S$ with $\boldsymbol{s} \rightarrow_{\mathcal{M}} \boldsymbol{s}' \iff \mathrm{Update}_{\mathcal{M}}(\boldsymbol{s}, \mathrm{reg}\,\boldsymbol{s}')$. The executions of $\mathcal{M}$ are the infinite sequences $(\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{s}_2 \ldots)$ where $\boldsymbol{s}_0 \in S_0$ and $\boldsymbol{s}_i \rightarrow_{\mathcal{M}} \boldsymbol{s}_{i+1}$ for all $i \in \mathbb{N}$.

We cast our model checking question as a language inclusion question. Given a system $\mathcal{M}$, we define as $\mathrm{obs}\,X_{\mathcal{M}} \subseteq X_{\mathcal{M}}$ its set of observables, and in turn define the language $L_{\mathcal{M}}$ of $\mathcal{M}$ as the set of sequences $(\mathrm{obs}\,\boldsymbol{s}_0, \mathrm{obs}\,\boldsymbol{s}_1 \ldots)$—which we call the traces of $\mathcal{M}$—induced by the executions $(\boldsymbol{s}_0, \boldsymbol{s}_1 \ldots)$ of $\mathcal{M}$. Given an LTL formula Φ over atomic propositions Π and the alphabet $\Sigma = 2^{\Pi}$ of valuations of Π, we define the language $L_{\Phi} \subseteq \Sigma^{\omega}$ of Φ as the set of all possible traces $(\sigma_0, \sigma_1 \ldots) \in \Sigma^{\omega}$ satisfying Φ. Provided that $\Pi = \mathrm{obs}\,X_{\mathcal{M}}$ (which we require to be Boolean), the model checking question asks whether $L_{\mathcal{M}} \subseteq L_{\Phi}$, i.e., all traces of $\mathcal{M}$ satisfy Φ. This is equivalent to the language emptiness question $L_{\mathcal{M}} \cap L_{\neg\Phi} = \emptyset$, i.e., no trace of $\mathcal{M}$ violates Φ [11, 42].

As is standard in automata-theoretic verification, we reduce the language emptiness question to an equivalent *fair emptiness* question. A non-deterministic Büchi automaton $\mathcal{A}$ consists of a finite set of states Q, an initial state $q_0 \in Q$, input alphabet Σ, transition relation $\delta \subseteq Q \times \Sigma \times Q$, and a set of fair states $F \subseteq Q$ [8, 39, 125]. This can be viewed as a reactive system with a single variable $X_{\mathcal{A}} = \{\mathtt{q}\}$ over Q, where $\mathrm{Init}_{\mathcal{A}}(q) \iff q = q_0$ and $\mathrm{Update}_{\mathcal{A}}(\sigma, q, q') \iff (q, \sigma, q') \in \delta$. We define the fair language $L^{\mathrm{f}}_{\mathcal{A}}$ of $\mathcal{A}$ as the set of traces of $\mathcal{A}$ whose respective executions visit F infinitely often. We rely on the fact that every LTL formula Φ admits a non-deterministic Büchi automaton $\mathcal{A}_{\Phi}$ for which $L^{\mathrm{f}}_{\mathcal{A}_{\Phi}} = L_{\Phi}$. Our reduction to fair emptiness thus constructs an automaton $\mathcal{A}_{\neg\Phi}$ and forms the synchronous product $\mathcal{M} \parallel \mathcal{A}_{\neg\Phi}$ between $\mathcal{M}$ and $A_{\neg\Phi}$ with the fair states defined as $S \times F$. We refer the reader to the literature for a full formal definition of the synchronous product [9, 11, 42]; the key property to note is that $L^{\mathrm{f}}_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}} = L_{\mathcal{M}} \cap L^{\mathrm{f}}_{\mathcal{A}_{\Phi}}$. Hence, our language emptiness question and, in turn, our model checking question amount to deciding the fair emptiness question $L^{\mathrm{f}}_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}} = \emptyset$.

We answer the fair emptiness question by presenting a certificate consisting of two components: (i) an inductive invariant defined as a set $I \subseteq \mathrm{reg}\,S \times Q$ that captures (or over-approximates) the reachable states of $\mathcal{M} \parallel \mathcal{A}_{\neg\Phi}$, and (ii) a ranking function $V : I \to \mathbb{W}$ over a well-founded relation $(\mathbb{W}, \prec)$ that assigns ranks to all states within the invariant that prove that all reachable executions visit fair states at most finitely many times. Concretely, for all $\boldsymbol{s}, \boldsymbol{s}' \in S$ and $q, q' \in Q$,

$$
\begin{aligned}
&\boldsymbol{s} \in S_0 && \Longrightarrow (\mathrm{reg}\,\boldsymbol{s}, q_0) \in I, && (1)\\
&(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \;\wedge\; (\mathrm{reg}\,\boldsymbol{s}, q) \in I && \Longrightarrow (\mathrm{reg}\,\boldsymbol{s}', q') \in I, && (2)\\
&(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \;\wedge\; (\mathrm{reg}\,\boldsymbol{s}, q) \in I && \Longrightarrow V(\mathrm{reg}\,\boldsymbol{s}, q) \succeq V(\mathrm{reg}\,\boldsymbol{s}', q'), && (3)\\
&(\boldsymbol{s}, q) \rightarrow_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}} (\boldsymbol{s}', q') \;\wedge\; (\mathrm{reg}\,\boldsymbol{s}, q) \in I \;\wedge\; q \in F && \Longrightarrow V(\mathrm{reg}\,\boldsymbol{s}, q) \succ V(\mathrm{reg}\,\boldsymbol{s}', q'). && (4)
\end{aligned}
$$

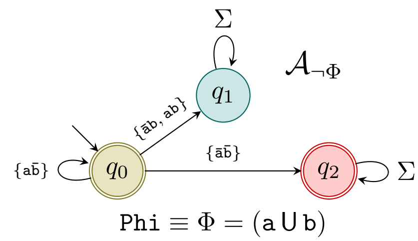
Clauses (1) and (2) state that initial states lie in I and that I is closed under transitions, so I forms an over-approximation of the set of reachable states. Clauses (3) and (4) require that, along every transition whose source is in I, the rank V never increases, and it decreases strictly whenever the source state is fair ($q \in F$). Because $(\mathbb{W}, \prec)$ is well founded, only finitely many strict decreases are possible; hence, fair states must be visited at most finitely many times. This implies that all executions are unfair, i.e., $L^{\mathrm{f}}_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}} = \emptyset$, which is equivalent to saying all traces of $\mathcal{M}$ satisfy Φ [125].

Figure 1 illustrates our workflow on an example. Figure 1a gives a SystemVerilog module $\mathcal{M}$ that satisfies the LTL property $\Phi = \mathtt{a}\ \mathsf{U}\ \mathtt{b}$. Figure 1b gives a Büchi automaton $\mathcal{A}_{\neg\Phi}$ recognising $\neg\Phi$. Figure 1c depicts the state space of the synchronous product $\mathcal{M} \parallel \mathcal{A}_{\neg\Phi}$: product states are arranged on a two-dimensional grid, each state corresponding to the pair of a state s_i of $\mathcal{M}$ (where i is the value of the register $\mathtt{c}$) and a state q_0, q_1, or q_2 of $\mathcal{A}_{\neg\Phi}$. The sole initial state is (s_0, q_0), and the dotted region indicates the inductive invariant I. Each state in I indicates in its upper-right corner the respective rank, taken from $\mathbb{N}$. Along each transition in I, the rank strictly decreases when leaving a fair state and never increases otherwise, ensuring that the sole fair cycle within I terminates. Notably,

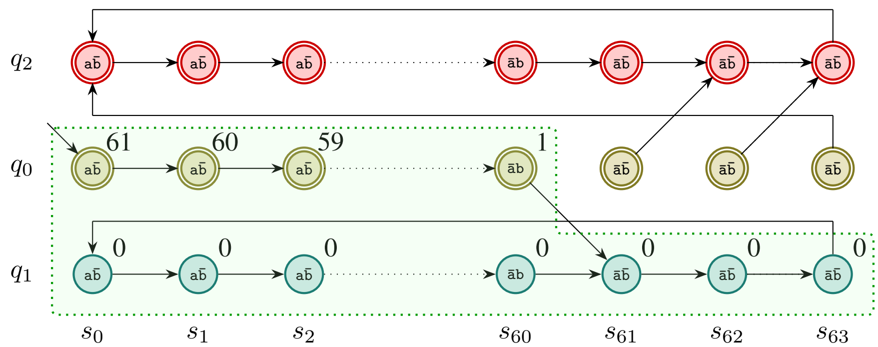
 https://doi.org/10.52202/085713-2509

```
module MODEL (input clk, output reg a, b);
  reg [5:0] c = 0;
  assign a = (c < 60);
  assign b = (c == 60);
  always @(posedge clk) begin
      c <= c + 1;
  end
  Phi: assert property
    (@(posedge clk) a s_until b);
endmodule
```

(a)



(b)



(c)

Figure 1: Model checking safety and liveness on an illustrative example

the system has a non-terminating fair cycle (e.g., the cycle on q_2), but this is unreachable and excluded by I. In general, invariants may include unreachable states, provided none can lead to a fair cycle.

3 Neural Partially-Ranking Functions

We reduce the fair emptiness problem to a machine learning task by (i) modelling the invariant I as a *classifier* over-approximating the reachable states, and (ii) modelling the ranking function V as a *regressor*. In this section, we present our learn-check workflow which, upon termination of our procedure, formally ensures that these two components jointly satisfy conditions (1)–(4). Crucially, we introduce the *neural partially-ranking function*, which combines the definition of inductive invariant and ranking function into a single neural network.

We define a neural partially-ranking function over discrete parameter space Θ as the function

$$\bar{V} : \mathrm{reg}\, S \times \Theta \to \mathbb{Z} \tag{5}$$

Our objective is to learn an upper bound $\kappa \in \mathbb{Z}$ and a set of parameters $\{\theta_q \in \Theta\}_{q \in Q}$, such that for all $s, s' \in S$ and $q, q' \in Q$, the following two conditions hold:

$$s \in S_0 \quad\Longrightarrow\quad \kappa \geq \bar{V}(\mathrm{reg}\, s; \theta_{q_0}), \tag{6}$$

$$(s,q) \to_{\mathcal{M} \| \mathcal{A}_{\neg\phi}} (s', q') \wedge \kappa \geq \bar{V}(\mathrm{reg}\, s; \theta_q) \quad\Longrightarrow\quad \bar{V}(\mathrm{reg}\, s; \theta_q) \;\geq\; \bar{V}(\mathrm{reg}\, s'; \theta_{q'}) + \mathbf{1}_F(q). \tag{7}$$

where $\mathbf{1}_F(q) = 1$ if $q \in F$ else 0. The trainable network parameters are in Θ and are distinct for each automaton state $q \in Q$, while κ is global. The *initiation condition* of Eq. (6) ensures that all initial states receive a rank bounded above by κ. The *ranking condition* of Eq. (7) ensures that states bounded above by κ are inductively assigned a rank. Ultimately, a solution $\boldsymbol{\kappa}$ and $\{\boldsymbol{\theta}_q\}_{q \in Q}$ induces an invariant I and ranking function V satisfying the clauses (1)–(4), respectively defined as follows:

$$I = \left\{(\mathrm{reg}\, s, q) \,:\, \boldsymbol{\kappa} \geq \bar{V}(\mathrm{reg}\, s; \boldsymbol{\theta}_q)\right\}, \qquad V(\mathrm{reg}\, s, q) \;=\; \bar{V}(\mathrm{reg}\, s; \boldsymbol{\theta}_q) \text{ if } (\mathrm{reg}\, s, q) \in I. \tag{8}$$

Since the hardware state space S and Q are finite, the image $\bar{\mathbb{W}} = \{\bar{V}(\mathrm{reg}\, s; \boldsymbol{\theta}_q) \colon s \in S, q \in Q\}$ of $\bar{V}$ for a constant parameter $\{\boldsymbol{\theta}_q\}_{q \in Q}$ is finite as well, having a least element under the strict inequality $<$ over the integers. This makes $(\bar{\mathbb{W}}, <)$ well founded, having no infinite descending chain.

We train $\bar{V}$ in a counterexample-guided inductive synthesis loop as illustrated in Figure 2a [1, 117]. Our procedure incrementally constructs two datasets D_{init} and D_{trans} to train $\bar{V}$ to respectively satisfy

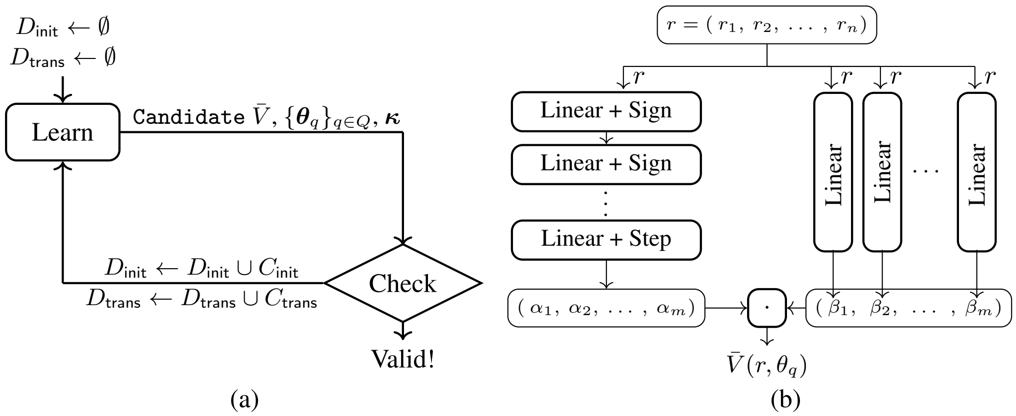


Figure 2: Our learn-check workflow (a) and our architecture for neural partially-ranking functions (b)

conditions (6) and (7). The dataset D_{init} contains sample initial register values $\operatorname{reg} \boldsymbol{s}$ of $\mathcal{M}$ where $\boldsymbol{s} \in S_0$, and the dataset D_{trans} contains sample transitions of $\mathcal{M} \parallel \mathcal{A}_{\neg\phi}$ in the form of quadruples $(\operatorname{reg} \boldsymbol{s}, \boldsymbol{q}, \operatorname{reg} \boldsymbol{s}', \boldsymbol{q}')$ where $(\boldsymbol{s}, \boldsymbol{q}) \rightarrow_{\mathcal{M}\parallel\mathcal{A}_{\neg\phi}} (\boldsymbol{s}', \boldsymbol{q}')$. Initially, we assume that D_{init} and D_{trans} are empty. Our learning component computes a set of parameters $\{\boldsymbol{\theta}_q\}_{q\in Q}$ to hold over the set of samples D_{init} and D_{trans}. Our checking component formally verifies whether conditions (6) and (7) hold over the entire state space. If the checker confirms the conditions, then the procedure halts successfully. In the converse case, it provides sets of counterexamples C_{init} and C_{trans}, which are respectively added to D_{init} and D_{trans}, and learning and checking are repeated in a loop.

Learning Our learning phase computes a set of parameters for which our initiation and ranking conditions are valid across all samples in the dataset. More precisely, enforcing our initiation condition amounts to satisfying the following constraint:

$$\operatorname{LearnInit}(\theta, \kappa) = \bigwedge_{\boldsymbol{r} \in D_{\mathsf{init}}} \big(\kappa \geq \bar{V}\big(\boldsymbol{r}; \theta_{q_0}\big)\big) \tag{9}$$

Similarly, satisfying our ranking condition amounts to satisfying the following constraint:

$$\operatorname{LearnRank}(\theta, \kappa) = \bigwedge_{(\boldsymbol{r},\boldsymbol{q},\boldsymbol{r}',\boldsymbol{q}') \in D_{\mathsf{trans}}} \big(\kappa \geq \bar{V}(\boldsymbol{r}; \theta_{\boldsymbol{q}}) \implies \bar{V}(\boldsymbol{r}; \theta_{\boldsymbol{q}}) \geq \bar{V}(\boldsymbol{r}'; \theta_{\boldsymbol{q}'}) + \mathbf{1}_F(\boldsymbol{q})\big) \tag{10}$$

Overall, our learning phase is tasked to solve the following question which, as we show in Section 4, we cast to a constraint satisfaction problem:

$$\exists \theta, \kappa \colon \operatorname{LearnInit}(\theta, \kappa) \wedge \operatorname{LearnRank}(\theta, \kappa) \tag{11}$$

If a valid set of parameters exists, then this is passed on to the checking phase. If a valid set of parameters does not exist, this indicates a larger network is needed or no solution exists at all. In this case, our strategy is to progressively add a hidden layer or widen it and retry until a preset limit.

Checking Our checking phase determines the presence or the absence of counterexamples to our initiation and ranking conditions for the candidate parameters $\boldsymbol{\kappa}, \{\boldsymbol{\theta}_q\}_{q\in Q}$. More precisely, a counterexample for the initiation condition must satisfy the following constraint:

$$\operatorname{CheckInit}(s) = \operatorname{Init}_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}}(s, q_0) \wedge \boldsymbol{\kappa} < \bar{V}(\operatorname{reg} s; \boldsymbol{\theta}_{q_0}) \tag{12}$$

Similarly, a counterexample for the ranking condition must satisfy the following constraint:

$$\begin{aligned} \operatorname{CheckRank}(s, q, r', q') = {} & \operatorname{Update}_{\mathcal{M}\parallel\mathcal{A}_{\neg\Phi}}(s, q, r', q') \wedge \\ & \boldsymbol{\kappa} \geq \bar{V}(\operatorname{reg} s; \boldsymbol{\theta}_q) \wedge \bar{V}(\operatorname{reg} s; \boldsymbol{\theta}_q) < \bar{V}(r'; \boldsymbol{\theta}_{q'}) + \mathbf{1}_F(q) \end{aligned} \tag{13}$$

Overall, our checking phase asks whether the following statement holds:

$$\exists s \colon \operatorname{CheckInit}(s) \quad \vee \quad \exists s, q, r', q' \colon \operatorname{CheckRank}(s, q, r', q') \tag{14}$$

We check the initialisation and ranking conditions independently. If both queries are unsatisfiable, then we conclude that $\bar{V}$ is a valid neural partially-ranking function. If either or both are satisfiable,

 https://doi.org/10.52202/085713-2509

then the respective assignments constitute counterexamples. To produce multiple counterexamples C_{init} or C_{trans} in each iteration, we check (13) over explicitly enumerated automaton pairs (q, q') independently. This accelerates convergence, and gives equal emphasis to each transition of $\mathcal{A}_{\neg\Phi}$.

Our counterexample-guided inductive synthesis procedure captures task-specific edge cases that yield succinct datasets. This is in contrast to prior work that benefits from random initialisation of the datasets before the first iteration [63]. Moreover, as we show in Section 4, we further optimise our procedure by proposing a specialised architecture that is amenable not only to checking but also to learning using efficient and numerically stable constraint satisfaction solvers.

4 Learning Neural Partially-Ranking Functions Using MILP Solvers

We present a specialised architecture for neural partially-ranking functions amenable to training using constraint satisfaction solvers. Our architecture is illustrated in Figure 2b. Given a vector of word-level register values $r = (r_1, \dots, r_n)$ with $n = |\operatorname{reg} X_{\mathcal{M}}|$, our architecture feeds r into two branches. The left component $\alpha^{(r,q)}$ consists of L fully connected layers $(W_1^{(q)}, b_1^{(q)}, \dots, W_L^{(q)}, b_L^{(q)})$ with $N_1, \dots, N_L$ neurons, respectively, sign activation functions on the hidden layers, and step function on the output layer featuring $m = N_L$ neurons. The right component consists of m linear functions with n inputs each: $\beta^{(r,q)} = A^{(q)} r + c^{(q)}$. The left component α acts as a *mask* that selects or excludes the linear functions of the right component β; in other words, $\bar{V}(r, \theta_q) = \alpha^{(r,q)} \cdot \beta^{(r,q)}$.

Every application of arguments $(\boldsymbol{r}, \theta_{\boldsymbol{q}})$ to $\bar{V}$ in Eqs. (9)–(11) requires encoding. We build upon the result for which feed-forward neural networks with sign activation functions are amenable to training with mixed-integer linear programming (MILP) solvers [67, 121]. Henceforth, we use i, j to index individual elements of linear components, specifically i for the inputs elements and j for output elements, and we use l to index layers of the network. A single hyperparameter $P \in \mathbb{N}_{>0}$ bounds the magnitude of all decision variables in the encoding. Decision variables shared across multiple occurrences of $\boldsymbol{r}$ and associated to a common $\boldsymbol{q}$ are denoted as $w_{i\ell j}^{(\boldsymbol{q})}, b_{\ell j}^{(\boldsymbol{q})}, a_{ij}^{(\boldsymbol{q})}, c_j^{(\boldsymbol{q})} \in [-P, P]$ to respectively refer to the elements of $W_l^{(\boldsymbol{q})}$, $b_l^{(\boldsymbol{q})}$, $A^{(\boldsymbol{q})}$ and $c^{(\boldsymbol{q})}$; the threshold $\kappa \in [-P, P]$ is also a common decision variable in the learning phase. Decision variables associated to specific occurrences of $\boldsymbol{r}$ and $\boldsymbol{q}$ are $u_{\ell j}^{(\boldsymbol{r},\boldsymbol{q})} \in \{0, 1\}$ to indicate the activation status of the corresponding neurons, $z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})}$ to represent neuron-weight products where $z_{i1j}^{(\boldsymbol{r},\boldsymbol{q})} \in [-PM_{\boldsymbol{r}}, PM_{\boldsymbol{r}}]$ with $M_{\boldsymbol{r}} = \max(|\boldsymbol{r}_1|, \dots, |\boldsymbol{r}_n|)$ and $z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} \in [-P, P]$ with $l \geq 2$, and finally $v_j^{(\boldsymbol{r},\boldsymbol{q})} \in [-PM_{\boldsymbol{r}}, PM_{\boldsymbol{r}}]$ to represent the overall output of the network. First, we encode the relationship between z, $\boldsymbol{r}$ and w at layer 1 as

$$z_{i1j}^{(\boldsymbol{r},\boldsymbol{q})} = \boldsymbol{r}_i w_{i1j}^{(\boldsymbol{q})} \tag{15}$$

Then, for $l = 1, \dots, L$, we encode the relationship between u, z and b at layer l (where $\epsilon > 0$) as

$$\left(u_{\ell j}^{(\boldsymbol{r},\boldsymbol{q})} = 1 \Rightarrow b_{\ell j} + \textstyle\sum_{i \in N_{\ell-1}} z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} \geq \epsilon\right) \wedge \left(u_{\ell j}^{(\boldsymbol{r},\boldsymbol{q})} = 0 \Rightarrow b_{\ell j} + \textstyle\sum_{i \in N_{\ell-1}} z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} \leq -\epsilon\right) \tag{16}$$

For $l = 2, \dots, L$, we encode the relationship between u at layer $l-1$ and z and w at layer l as

$$\begin{aligned}
&(z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} - w_{i\ell j}^{(\boldsymbol{q})} + 2P\, u_{(\ell-1)i}^{(\boldsymbol{r},\boldsymbol{q})} \leq 2P) \wedge (z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} + w_{i\ell j}^{(\boldsymbol{q})} - 2P\, u_{(\ell-1)i}^{(\boldsymbol{r},\boldsymbol{q})} \leq 0) \wedge \\
&(z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} - w_{i\ell j}^{(\boldsymbol{q})} - 2P\, u_{(\ell-1)i}^{(\boldsymbol{r},\boldsymbol{q})} \geq -2P) \wedge (z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} + w_{i\ell j}^{(\boldsymbol{q})} + 2P\, u_{(\ell-1)i}^{(\boldsymbol{r},\boldsymbol{q})} \geq 0)
\end{aligned} \tag{17}$$

which represents $z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} = w_{i\ell j}$ if $u_{(\ell-1)j}^{(\boldsymbol{r},\boldsymbol{q})} = 1$, and $z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})} = -w_{i\ell j}$ otherwise. Finally, we encode the relationship between v, a, c and u at layer L as

$$\left(u_{Lj}^{(\boldsymbol{r},\boldsymbol{q})} = 1 \Rightarrow v_j^{(\boldsymbol{r},\boldsymbol{q})} = c_j^{(\boldsymbol{q})} + \textstyle\sum_{i \in n} \boldsymbol{r}_i a_{ij}^{(\boldsymbol{q})}\right) \wedge \left(u_{Lj}^{(\boldsymbol{r},\boldsymbol{q})} = 0 \Rightarrow v_j^{(\boldsymbol{r},\boldsymbol{q})} = 0\right) \tag{18}$$

As a result, each occurrence of $\bar{V}(\boldsymbol{r}, \theta_{\boldsymbol{q}})$ in Eqs. (9)–(11) corresponds to the linear term $\sum_{i \in m} v_i^{(\boldsymbol{r},\boldsymbol{q})}$.

Notably, a naïve encoding that represents $z_{i\ell j}^{(\boldsymbol{r},\boldsymbol{q})}$ as a product between a neuron variable and $w_{i\ell j}^{(\boldsymbol{q})}$ would induce bilinear constraints; our encoding avoids this and produces an equivalent encoding with linear constraints only. This enables using efficient MILP (and SMT) solvers in the learning phase.

Essentially, our architecture represents a piecewise function with 2^m configurations, given by $\alpha \in \{0, 1\}^m$, each of which induces a specific summation of linear functions, given by $\beta \in \mathbb{Z}^m$, in the spirit of piecewise-linear ranking functions [76, 123, 124]. In our implementation, we restrict all decision variables to the *integer* type, which results in neural networks that are naturally quantised.

Table 1: Tasks completed per tool, per design; in bold the max per design, per spec-type

	LS	LCD	Tmcp	i2cS	7-Seg	PWM	VGA	UARTt	Delay	Gray	Blink	Total
Pure Safety												
Tasks	16	28	17	20	15	12	20	10	32	11	25	206
`Our`	**16**	**28**	**17**	4	**15**	**12**	**20**	**10**	**32**	**11**	**25**	190
`ABC`	**16**	**28**	**17**	**20**	**15**	**12**	**20**	**10**	**32**	**11**	**25**	206
`nuXmv`	**16**	**28**	**17**	**20**	**15**	**12**	**20**	**10**	**32**	**11**	**25**	206
`rIC3`	**16**	**28**	**17**	**20**	**15**	2	**20**	**10**	**32**	**11**	**25**	196
Pure Liveness												
Tasks	16	14	17	20	30	12	10	10	32	33	25	219
`Our`	0	**14**	**17**	**20**	**30**	**12**	**10**	**10**	**32**	**33**	**14**	192
`ABC`	3	5	9	4	10	3	5	**10**	8	16	5	78
`nuXmv`	12	13	**17**	16	28	7	4	**10**	**32**	**33**	8	180
`rIC3`	7	7	13	7	17	4	6	**10**	14	19	6	110
`NMC'24`	**15**	3	10	**20**	**30**	10	9	**10**	**32**	**33**	0	172
Safety + Liveness												
Tasks	32	14	17	20	15	12	10	10	32	22	25	209
`Our`	**32**	**14**	**17**	3	9	**12**	**10**	**10**	**32**	**22**	11	172
`ABC`	7	12	11	**20**	5	3	**10**	7	6	11	4	96
`nuXmv`	11	12	14	16	**15**	6	**10**	**10**	26	14	**20**	154
`rIC3`	12	**14**	14	**20**	8	4	**10**	9	18	14	6	129

5 Experimental Evaluation

Implementation We implemented a prototype using a modular `Python` 3.12 pipeline [2]. Given an LTL formula Φ, we obtained $\mathcal{A}_{\neg\Phi}$ with `Spot` 2.11.6 [51], converted SystemVerilog to bit-vector SMT using `EBMC` 5.6 [91], performed validity queries and computed counterexamples using `Bitwuzla` 0.7.0 [96], and trained a neural partially-ranking function using the MILP solver `Gurobi` 12.0.1 [65]. Our method relies on a single global hyperparameter configuration, besides the integer bound P (see Section 4). We chose P from a finite set of configurations $\{1, 5, 10, \lfloor M/10 \rfloor, \lfloor M/2 \rfloor, M, M{+}1, 2M\}$, where M is the largest value any register can take in the hardware design; this renders the bounds for z_{i1j} and v_j quadratic in M except when $P = 1$, 5, or 10. We configured the architecture to start with a linear model (per $q \in \mathcal{A}_{\neg\phi}$) and add one hidden layer with a single neuron (left branch of Figure 2a) upon the first failure of the learning phase, subsequently increasing its width by one neuron upon subsequent failures, up to 5 attempts.

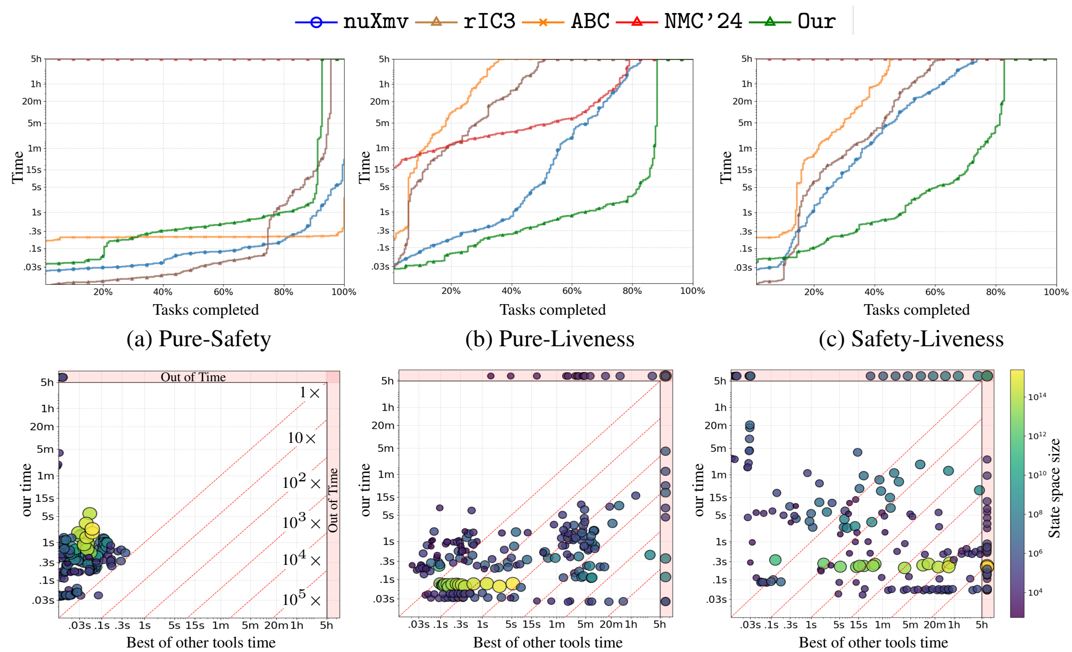
Experimental Setup Building on the ten parameterised SystemVerilog designs and 194 pure-liveness tasks of prior work [63], we add an eleventh design and, for all eleven, contribute additional pure-safety and combined safety–liveness specifications on each task, yielding a 634-task suite that retains the original diversity (Appendix A). We compared against (i) the gradient-based neural model checking approach, labelled `NMC'24` (liveness only) [63], and (ii) the winners of recent HWMCC editions [107]: `nuXmv` 2.1.0 [28], `rIC3` 1.5.1 [119], and `ABC/Super-Prove` [24]. For `nuXmv` we translated each design with its assertions to SMV using `EBMC`; then, we converted SMV to AIG for `ABC` and `rIC3` with `AIGER` [18], yielding a fully automated flow. All experiments ran on an AWS m6a.4xlarge instance (16 vCPUs, 64 GB RAM, Ubuntu 24.04). Each task had a $5\,\mathrm{h}$ timeout per tool; in aggregate, the experiments consumed $9083\,\mathrm{h}$ ($\approx 378\,\mathrm{d}$) of compute time.

Overall Efficacy Table 1 presents all solved instances by property class. For *pure-safety*—historically emphasised in HWMCC [16, 107]—symbolic tools clear $\geq 95\,\%$ of tasks, and we remain close to decades-mature tools at $92\,\%$. For *pure-liveness*, performance spreads: `nuXmv` $82\,\%$, `ABC` $36\,\%$, `rIC3` $50\,\%$, `NMC'24` $79\,\%$, while we complete $88\,\%$. On *safety–plus-liveness* properties, the gap widens, `nuXmv` solves $74\,\%$, `ABC` $46\,\%$, `rIC3` $61\,\%$, and our approach reaches $82\,\%$.

Comparison on Pure Safety We generally perform on par with leading symbolic checkers with decades of optimisation, as Figure 3a indicates, while being faster than `NuXmv` on $17\,\%$, `rIC3` on

[2]`https://github.com/aiverification/neuralmc`

https://doi.org/10.52202/085713-2509



(a) Pure-Safety (b) Pure-Liveness (c) Safety-Liveness

(d) Pure-Safety (e) Pure-Liveness (f) Safety-Liveness

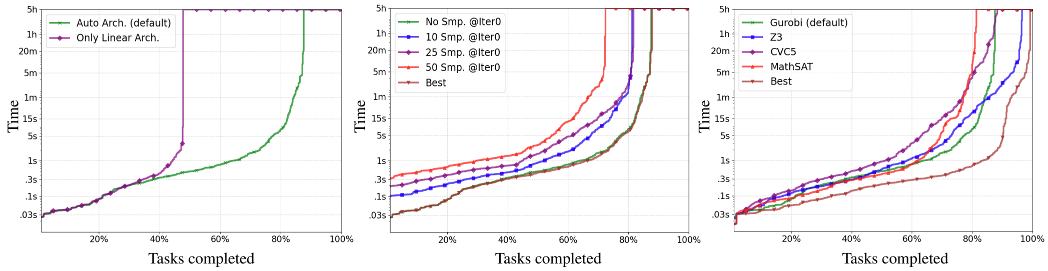
Figure 3: Runtime comparison with the state of the art (all times are in *log scale*)

$25\,\%$, and ABC on $32\,\%$ of safety tasks. Figure 3d plots each individual task by time, with the per-task fastest symbolic solver on the x-axis and our method on the y-axis; point size and brightness reflect state-space size. While symbolic tools often lead on individual instances, in $80\,\%$ of tasks both finish within $1\,$s, indicating that a our method can meaningfully match a portfolio of symbolic safety checkers, besides being the first neural model checking framework to support general safety.

Comparison on Pure Liveness Figure 3b indicates that our method is consistently faster than the others. Within $1\,$s we solve $24\,\%$ more tasks than `nuXmv`, $59\,\%$ more tasks than `rIC3`, $59\,\%$ more than `ABC`, and $64\,\%$ more than `NMC'24`; at $1\,\mathrm{min}$ the margins widen to $31\,\%$, $68\,\%$, $76\,\%$, $71\,\%$, and they still measure $12\,\%$, $45\,\%$, $59\,\%$, $13\,\%$ after $1\,$h. The percentage of tasks completed in $5\,$h per task by `nuXmv`, `NMC'24`, `rIC3`, and `ABC` are completed in under $7\,$s, $3\,$s, $0.6\,$s, $0.3\,$s by our method. Figure 3d shows the per-task runtime gains against the fastest competing method per task: our method is faster on $66\,\%$ of tasks, at least $10\times$ faster on $46\,\%$, $10^2\times$ faster on $29\,\%$, $10^3\times$ faster on $11\,\%$, $10^4\times$ faster on $4\,\%$, and $10^5\times$ faster on $1\,\%$ with respect to the whole state of the art. On a specific comparison with `NMC'24` we obtain speedups of at least $10\times$ on $88\,\%$, $10^2\times$ on $79\,\%$, $10^3\times$ on $45\,\%$, $10^4\times$ on $18\,\%$, and $10^5\times$ on $5\,\%$. Finally overall, $5\,\%$ of tasks timeout for all tools except ours.

Comparison on Combined Safety and Liveness Figure 3c indicates that the performance gap between our method and the alternatives further widens on safety-liveness tasks. Within $1\,$s our approach completes $31\,\%$ more tasks than `nuXmv`, $34\,\%$ more than `rIC3` and $36\,\%$ more than `ABC`; after $1\,\mathrm{min}$ the gaps grow to $38\,\%$, $43\,\%$, $54\,\%$, and persist at $20\,\%$, $31\,\%$, $41\,\%$ after $1\,$h. Benchmarks that exhaust `nuXmv`, `ABC` or `rIC3` in $5\,$h finish in $56\,$s, $6\,$s, $0.8\,$s per task with our approach. Figure 3f shows that we are faster on $61\,\%$, $10\times$ faster on $52\,\%$, $10^2\times$ faster on $43\,\%$, $10^3\times$ faster on $36\,\%$, $10^4\times$ faster on $27\,\%$, and $10^5\times$faster on $6\,\%$ of the tasks; all tools, besides ours, hit the $5\,$h timeout on $18\,\%$ of the task suite, bringing arbitrary LTL verification to runtime parity with pure-safety.

Neural Certificates Expressivity In Figure 4a, `Auto` is our default configuration that starts with linear for each state q of $\mathcal{A}_{\neg\Phi}$ and escalates to neural models only if linear is infeasible; `Only Linear` is the ablation that *exits* immediately when the linear model fails. The linear model completes $48\,\%$ of tasks within $9\,$s per task and makes no progress thereafter; by contrast, our default finishes $87\,\%$ overall. Since the default always tries the linear model first, it never trails the ablation; when linear



(a) Linear Architecture Ablation (b) Uniformly Random Samples (c) Other Learning Engines

Figure 4: Runtime analysis of our approach (all times are in *log scale*)

fails, automatic widening provides the expressivity needed to solve a lot more instances, showing that purely affine certificates are insufficient while our neural-certificate architecture succeeds.

Comparison with Random Dataset Generation Our iterative procedure presented in Figure 2a starts with empty datasets D_{init} and D_{trans} and populates them purely via a counterexample-guided procedure, which identifies edge-cases one by one towards the synthesis of neural partially-ranking functions. This is in contrast with the cognate neural model checking approach based on gradient descent, which benefits from random test generation to initially populate the datasets and rarely relies on the generation of data by means of counterexamples [63, Section 5]. Figure 4b evaluates our MILP-based approach under multiple random initialisation sizes with 0 (our default), 10, 25, and 50 randomly generated samples, respectively. As our experiment indicates, our default approach is nearly indistinguishable from the *Best* curve (which indicates the fastest configuration per instance). This indicates that our MILP-based approach does not benefit from unguided random data generation.

Comparison with SMT Learning Engines Our new architecture makes learning amenable to MILP solving and, more generally, to decision procedures for linear theories implemented in SMT solvers. Figure 4c compares alternative engines for our learn phase, where we replace the MILP solver `Gurobi` with the SMT solvers `CVC5` 1.1.2 [14], `MathSAT` 5.6.10 [36], and `Z3` 4.13.0 [49] via `PySMT` [59]. Within the $5\,\mathrm{h}$ timeout, `MathSAT` completed $81\,\%$ of tasks, `Gurobi` completed $87\,\%$, `CVC5` completed $88\,\%$, and `Z3` completed $96\,\%$ of tasks. `Gurobi` and `Z3` offer the best compromise between speed and efficacy, with `Z3` demonstrating superior coverage and `Gurobi` shorter runtimes: `Gurobi` was on average $7\times$ faster than `Z3` on successfully solved instances. However, `Gurobi` failed to return a result in $13\,\%$ of cases because of memory segmentation faults, model infeasibility for our hyper-parameters, numerical mismatches after evaluation of the result with `NumPy`; only 2 tasks were timeouts. The *Best* curve selects the fastest engine per instance—obviously outperforming every other solver, providing a baseline for a potential portfolio implementation. We remark that all experiments besides this comparison (Figure 4c) use only `Gurobi` as its learning engine.

Strengths and Limitations Our approach is much more straightforward in terms of parameter tuning then the prior neural model checking approach [63], owing to the theoretical completeness and the superior numerical stability of MILP and SMT algorithms over stochastic gradient descent. However, this strongly relies on sign-activated feed-forward neural networks; the generalisation to further architectures is an open problem for future investigation. Our method is amenable to training over parameters of integer type, which naturally results in quantised neural partially-raking functions. On the other hand, our approach is limited to hardware and reactive systems without arrays or strings of parametric length, procedure calls or dynamic data structures, which are topics for future work.

Threats to Validity Our empirical evaluation has demonstrated the superior performance of our approach over the state of the art on verification tasks derived from standard textbook level hardware designs. Our benchmark suite encompasses word-level designs with safety and liveness properties, unlike the standard HWMCC'24 benchmarks which focus on pure safety properties [19]. Our benchmark is arguably the hardest (as of today) for word-level liveness and combined safety-liveness hardware verification. Yet, our evaluation may not be representative of verifications tasks other than our suite and further research is required to assess the generalisability and the scalability of our approach to other workloads. Future work includes the integration of neural and compositional model checking towards the scalable formal verification of industrial-scale designs [40, 88, 134].

https://doi.org/10.52202/085713-2509

6 Related Work

Model Checking Linear Temporal Logic Formal hardware verification relies on model checking LTL specifications—codified, e.g., as SystemVerilog Assertions—to guarantee safety and liveness properties [8, 56, 77, 102]. Symbolic engines based on BDD fixed-points [10, 41, 75] or SAT/IC3 reasoning [17, 20, 73, 87, 114] among other methods have evolved over fifty years, with key contributions to formal verification honoured by the ACM Turing Awards of 1996, 2007, and 2013. The automata-theoretic approaches to model checking often reduce model checking to fair-emptiness. This has been tackled via k-liveness bounded model checking [38, 69], IC3 with strongly connected components [23], or BDDs with Emerson–Lei fixed-point computation [52]. Ranking functions were originally devised for program termination [55] and later generalised to liveness certificates [5, 6, 44, 50, 64, 81, 125], may be linear [21, 104], piecewise-defined [76, 123, 124], word-level [32, 47], lexicographic [22, 82], or represented as disjunctive well-founded relations [37, 43, 45, 74, 103]. Traditional algorithms for the automated construction of ranking functions and inductive invariants usually rely on constraint solving (using Farkas' lemma or Positivstellensatz results) or abstract interpretation [31, 43, 45, 74, 103, 105].

Machine Learning for Automated Reasoning Neural methods have permeated every layer of formal reasoning. In theorem proving, they guide clause selection [53, 84], premise selection [13, 68, 89, 127], tactic prediction [100, 132], and even end-to-end proof search [54, 106]. In constraint solving, neural approaches have been integrated into SMT [12, 85, 112] and SAT solving [61, 79, 113, 128], and have been further extended to combinatorial optimisation [83] and MILP [60, 93]. Similar ideas have been applied to program synthesis [15, 33, 72, 122], algorithm selection [80, 86], termination analysis [7], circuit synthesis and repair from temporal logic specifications [46, 111], and generation of satisfying traces for given LTL properties [66]. A large body of work focussed on learning inductive invariants via teacher–learner loops [57], decision trees [58, 76], random search [115], data-driven templates [98], neural networks [109, 133], reinforcement learning [116, 131] and, most recently, large language models [29, 71, 101, 130]. These techniques target safety alone; LTL specifications that encompass both safety *and* liveness are out of scope. Handling properties that combine safety and liveness using machine learning fills a significant gap in the field.

Neural Certificates As opposed to using neural networks to *generate* formal proof certificates, our approach uses neural networks to *represent* them (along with mathematical guarantees), falling within the spectrum of neural certificates. This extends recent work on control [2, 30, 48, 78, 92, 95, 108, 135–137], formal verification of software and probabilistic programs [3, 4, 62], and model checking reactive systems under *pure liveness* LTL properties [63]. Our work has enabled neural certificates to effectively prove *arbitrary* LTL properties, orders of magnitude faster than the state of the art.

7 Conclusion

We have presented a novel and improved neural model checking approach for arbitrary compositions of safety and liveness specifications. By introducing a specialised neural partially-ranking function architecture that (i) simultaneously represents inductive invariants and ranking functions and (ii) is amenable to training with numerically stable MILP solvers in a counterexample-guided procedure, our method yields compact and naturally quantised certificates. We implemented a lightweight Python prototype that combines off-the-shelf MILP (`Gurobi`), SMT (`Bitwuzla`), and BMC (`EBMC`) to train and certify the network in an iterative approach that provides formal guarantees [65, 91, 96]. Our improved technique attains speedups up to $10^5\times$ and $10\,\%$ higher completion (within a $5\,\mathrm{h}$ time limit) over the state-of-the-art symbolic model checkers and prior work on neural model checking [24, 28, 63, 119], on a benchmark suite of 634 hardware model checking tasks encompassing safety, liveness and combined safety-liveness verification tasks written in SystemVerilog.

A complete specification for the functional correctness of reactive systems must capture *both* safety and liveness: the former guarantees the absence of undesired behaviour; the latter guarantees the presence of desired behaviour. Our approach has extended neural model checking to capture both safety and liveness, simultaneously. This paves a path towards the provably safe application of artificial intelligence and machine learning to the design of high-assurance industrial hardware, with impact to avionics [90, 118], automotive [97, 120], medical devices [70, 126], nuclear control [27, 99], rail signalling [34, 35], robotics [110, 129], utility grids [25, 94], and more.

Acknowledgments

We thank Matthew Leeke, Sonia Marin, and Mark Ryan for their feedback and the anonymous reviewers for their comments and suggestions on this manuscript. This work was supported in part by the Amazon Research Award *Neural Software Verification* (Fall 2024), and by the Advanced Research and Invention Agency (ARIA) under the *Safeguarded AI* programme.

References

[1] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. Counterexample guided inductive synthesis modulo theories. In *CAV (1)*, volume 10981 of *LNCS*, pages 270–288. Springer, 2018.

[2] A. Abate, D. Ahmed, M. Giacobbe, and A. Peruffo. Formal synthesis of Lyapunov neural networks. *IEEE Control. Syst. Lett.*, 5(3):773–778, 2021.

[3] A. Abate, M. Giacobbe, and D. Roy. Learning probabilistic termination proofs. In *CAV (2)*, volume 12760 of *LNCS*, pages 3–26. Springer, 2021.

[4] A. Abate, A. Edwards, M. Giacobbe, H. Punchihewa, and D. Roy. Quantitative verification with neural networks. In *CONCUR*, volume 279 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[5] A. Abate, M. Giacobbe, and D. Roy. Stochastic omega-regular verification and control with supermartingales. In *CAV (3)*, volume 14683 of *LNCS*, pages 395–419. Springer, 2024.

[6] A. Abate, M. Giacobbe, and D. Roy. Quantitative supermartingale certificates. In *CAV (2)*, volume 15932 of *LNCS*, pages 3–28. Springer, 2025.

[7] Y. Alon and C. David. Using graph neural networks for program termination. In *ESEC/SIGSOFT FSE*, pages 910–921. ACM, 2022.

[8] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Comput.*, 2(3): 117–126, 1987.

[9] R. Alur and T. A. Henzinger. Reactive modules. In *LICS*, pages 207–218. IEEE, 1996.

[10] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Reveal: A formal verification tool for Verilog designs. In *LPAR*, volume 5330 of *LNCS*, pages 343–352. Springer, 2008.

[11] C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[12] M. Balunovic, P. Bielik, and M. T. Vechev. Learning to solve SMT formulas. In *NeurIPS*, pages 10338–10349, 2018.

[13] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019.

[14] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, et al. cvc5: A versatile and industrial-strength SMT solver. In *TACAS*, LNCS, pages 415–442. Springer, 2022.

[15] S. Barke, E. A. Gonzalez, S. R. Kasibatla, T. Berg-Kirkpatrick, and N. Polikarpova. HYSYNTH: context-free LLM approximation for guiding program synthesis. In *NeurIPS*, 2024.

[16] A. Biere, N. Froleyks, and M. Preiner. Hardware model checking competition (HWMCC) 2020. URL `https://hwmcc.github.io/2020/`.

[17] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

 https://doi.org/10.52202/085713-2509

[18] A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[19] A. Biere, N. Froleyks, and M. Preiner. Hardware model checking competition 2024. In *FMCAD*, pages 7–7. IEEE, 2024.

[20] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[21] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, volume 3576 of *LNCS*, pages 491–504. Springer, 2005.

[22] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, volume 3580 of *LNCS*, pages 1349–1361. Springer, 2005.

[23] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang. An incremental approach to model checking progress properties. In *FMCAD*, pages 144–153. FMCAD Inc., 2011.

[24] R. K. Brayton and A. Mishchenko. ABC: an academic industrial-strength verification tool. In *CAV*, volume 6174 of *LNCS*, pages 24–40. Springer, 2010.

[25] T. Brihaye, M. Jungers, S. Lasaulce, N. Markey, and G. Oreiby. Using model checking for analyzing distributed power control problems. *EURASIP Journal on Wireless Communications and Networking*, 2010:1–13, 2010.

[26] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439. IEEE Computer Society, 1990.

[27] I. Buzhinsky and A. Pakonen. Model-checking detailed fault-tolerant nuclear power plant safety functions. *IEEE Access*, 7:162139–162156, 2019.

[28] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In *CAV*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.

[29] S. Chakraborty, S. K. Lahiri, S. Fakhoury, A. Lal, M. Musuvathi, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy. Ranking LLM-generated loop invariants for program verification. In *EMNLP (Findings)*, pages 9164–9175. Association for Computational Linguistics, 2023.

[30] Y. Chang, N. Roohi, and S. Gao. Neural Lyapunov control. In *NeurIPS*, pages 3240–3249, 2019.

[31] K. Chatterjee, H. Fu, and A. K. Goharshady. Termination analysis of probabilistic programs through Positivstellensatz's. In *CAV (1)*, volume 9779 of *LNCS*, pages 3–22. Springer, 2016.

[32] H. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Bit-precise procedure-modular termination analysis. *ACM Trans. Program. Lang. Syst.*, 40(1):1:1–1:38, 2018.

[33] X. Chen, D. Song, and Y. Tian. Latent execution for neural program synthesis beyond domain-specific languages. In *NeurIPS*, pages 22196–22208, 2021.

[34] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal verification of a railway interlocking system using model checking. *Formal Aspects Comput.*, 10(4):361–380, 1998.

[35] A. Cimatti, R. Corvino, A. Lazzaro, I. Narasamdya, T. Rizzo, M. Roveri, A. Sanseviero, and A. Tchaltsev. Formal verification and validation of ERTMS industrial railway train spacing system. In *CAV*, volume 7358 of *LNCS*, pages 378–393. Springer, 2012.

[36] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *TACAS*, LNCS, pages 93–107. Springer, 2013.

[37] A. Cimatti, A. Griggio, C. Johannsen, K. Y. Rozier, and S. Tonetta. Infinite-state liveness checking with rlive. In *CAV (1)*, volume 15931 of *LNCS*, pages 215–236. Springer, 2025.

[38] K. Claessen and N. Sörensson. A liveness checking algorithm that counts. In *FMCAD*, pages 52–59. IEEE, 2012.

[39] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[40] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS*, pages 353–362. IEEE Computer Society, 1989.

[41] E. M. Clarke, R. P. Kurshan, and H. Veith. The localization reduction and counterexample-guided abstraction refinement. In *Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 61–71. Springer, 2010.

[42] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith. *Model Checking, 2nd Edition*. MIT Press, 2018.

[43] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426. ACM, 2006.

[44] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL*, pages 265–276. ACM, 2007.

[45] B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS*, volume 7795 of *LNCS*, pages 47–61. Springer, 2013.

[46] M. Cosler, F. Schmitt, C. Hahn, and B. Finkbeiner. Iterative circuit repair against formal specifications. In *ICLR*. OpenReview.net, 2023.

[47] C. David, D. Kroening, and M. Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In *ESOP*, volume 9032 of *LNCS*, pages 183–204. Springer, 2015.

[48] C. Dawson, Z. Qin, S. Gao, and C. Fan. Safe nonlinear control using robust neural Lyapunov-barrier functions. In *CoRL*, volume 164 of *Proceedings of Machine Learning Research*, pages 1724–1735. PMLR, 2021.

[49] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, LNCS, pages 337–340. Springer, 2008.

[50] R. Dimitrova, L. M. F. Fioriti, H. Hermanns, and R. Majumdar. Probabilistic CTL*: The deductive way. In *TACAS*, volume 9636 of *LNCS*, pages 280–296. Springer, 2016.

[51] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko. From Spot 2.0 to Spot 2.10: What's new? In *CAV (2)*, volume 13372 of *LNCS*, pages 174–187. Springer, 2022.

[52] E. A. Emerson and C. Lei. Modalities for model checking: Branching time strikes back. In *POPL*, pages 84–96. ACM Press, 1985.

[53] V. Firoiu, E. Aygün, A. Anand, Z. Ahmed, X. Glorot, L. Orseau, L. M. Zhang, D. Precup, and S. Mourad. Training a first-order theorem prover from synthetic data. *CoRR*, abs/2103.03798, 2021.

[54] E. First, M. N. Rabe, T. Ringer, and Y. Brun. Baldur: Whole-proof generation and repair with large language models. In *ESEC/SIGSOFT FSE*, pages 1229–1241. ACM, 2023.

[55] R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

[56] H. Foster. The 2022 Wilson research group functional verification study, 2022. URL `https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/`.

[57] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV*, volume 8559 of *LNCS*, pages 69–87. Springer, 2014.

 https://doi.org/10.52202/085713-2509

[58] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, pages 499–512. ACM, 2016.

[59] M. Gario and A. Micheli. PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*, 2015.

[60] Z. Geng, X. Li, J. Wang, X. Li, Y. Zhang, and F. Wu. A deep instance generative framework for MILP solvers under limited data availability. In *NeurIPS*, 2023.

[61] M. Ghanem, F. Schmitt, J. Siber, and B. Finkbeiner. Learning better representations from less data for propositional satisfiability. In *NeurIPS*, 2024.

[62] M. Giacobbe, D. Kroening, and J. Parsert. Neural termination analysis. In *ESEC/SIGSOFT FSE*, pages 633–645. ACM, 2022.

[63] M. Giacobbe, D. Kroening, A. Pal, and M. Tautschnig. Neural model checking. In *NeurIPS*, 2024.

[64] O. Grumberg, N. Francez, J. A. Makowsky, and W. P. de Roever. A proof rule for fair termination of guarded commands. *Inf. Control.*, 66(1/2):83–102, 1985.

[65] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL `https://www.gurobi.com`.

[66] C. Hahn, F. Schmitt, J. U. Kreber, M. N. Rabe, and B. Finkbeiner. Teaching temporal logics to neural networks. In *ICLR*. OpenReview.net, 2021.

[67] R. T. Icarte, L. Illanes, M. P. Castro, A. A. Ciré, S. A. McIlraith, and J. C. Beck. Training binarized neural networks using MIP and CP. In *CP*, volume 11802 of *LNCS*, pages 401–417. Springer, 2019.

[68] G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban. Deepmath – deep sequence models for premise selection. In *NIPS*, pages 2235–2243, 2016.

[69] A. Ivrii, Z. Nevo, and J. Baumgartner. k-FAIR = k-LIVENESS + FAIR—revisiting SAT-based liveness algorithms. In *FMCAD*, pages 1–5. IEEE, 2018.

[70] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam. Modeling and verification of a dual chamber implantable pacemaker. In *TACAS*, volume 7214 of *LNCS*, pages 188–203. Springer, 2012.

[71] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma. Finding inductive loop invariants using large language models. *CoRR*, abs/2311.07948, 2023.

[72] N. Kobayashi, T. Sekiyama, I. Sato, and H. Unno. Towards neural-network-guided program synthesis and verification. *Formal Methods in System Design*, pages 1–33, 2025.

[73] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *VMCAI*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.

[74] D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV*, volume 6174 of *LNCS*, pages 89–103. Springer, 2010.

[75] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *CAV*, volume 1633 of *LNCS*, pages 172–183. Springer, 1999.

[76] S. Kura, H. Unno, and I. Hasuo. Decision tree learning in CEGIS-based termination analysis. In *CAV (2)*, volume 12760 of *LNCS*, pages 75–98. Springer, 2021.

[77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3 (2):125–143, 1977.

[78] M. Lechner, D. Žikelić, K. Chatterjee, and T. A. Henzinger. Stability verification in stochastic control systems via neural network supermartingales. In *AAAI*, pages 7326–7336. AAAI Press, 2022.

[79] G. Lederman, M. N. Rabe, S. Seshia, and E. A. Lee. Learning heuristics for quantified boolean formulas through reinforcement learning. In *ICLR*. OpenReview.net, 2020.

[80] W. Leeson and M. B. Dwyer. Algorithm selection for software verification using graph neural networks. *ACM Trans. Softw. Eng. Methodol.*, 33(3):76:1–76:36, 2024.

[81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *ICALP*, volume 115 of *LNCS*, pages 264–277. Springer, 1981.

[82] J. Leike and M. Heizmann. Ranking templates for linear loops. *Log. Methods Comput. Sci.*, 11(1), 2015.

[83] Y. Li, J. Guo, R. Wang, and J. Yan. From distribution learning in training to gradient search in testing for combinatorial optimization. In *NeurIPS*, 2023.

[84] S. M. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017.

[85] Z. Lu, S. Siemer, P. Jha, F. Manea, J. Day, and V. Ganesh. Z3-alpha: a reinforcement learning guided SMT solver. *System Description: SMT-COMP*, 2023.

[86] Z. Lu, P. Chien, N. Lee, and V. Ganesh. Algorithm selection for word-level hardware model checking (student abstract). In *AAAI*, pages 29426–29427. AAAI Press, 2025.

[87] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV*, volume 2404 of *LNCS*, pages 250–264. Springer, 2002.

[88] K. L. McMillan. Toward liveness proofs at scale. In *CAV (1)*, volume 14681 of *LNCS*, pages 255–276. Springer, 2024.

[89] M. Mikula, S. Tworkowski, S. Antoniak, B. Piotrowski, A. Q. Jiang, J. P. Zhou, C. Szegedy, L. Kucinski, P. Milos, and Y. Wu. Magnushammer: A transformer-based approach to premise selection. In *ICLR*. OpenReview.net, 2024.

[90] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or formal verification: DO-178C alternatives and industrial experience. *IEEE Softw.*, 30(3):50–57, 2013.

[91] R. Mukherjee, D. Kroening, and T. Melham. Hardware verification using software analyzers. In *ISVLSI*, pages 7–12. IEEE Computer Society, 2015.

[92] A. Nadali, V. Murali, A. Trivedi, and M. Zamani. Neural closure certificates. In *AAAI*, pages 21446–21453. AAAI Press, 2024.

[93] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. Li, O. Vinyals, and Y. Zwols. Solving mixed integer programs using neural networks. *CoRR*, abs/2012.13349, 2020.

[94] Y. Nemouchi, S. Etigowni, A. J. Zolan, and R. Macwan. Formally verified ZTA requirements for OT/ICS environments with Isabelle/HOL. In *VSTTE*, volume 14095 of *LNCS*, pages 64–84. Springer, 2023.

[95] G. Neustroev, M. Giacobbe, and A. Lukina. Neural continuous-time supermartingale certificates. In *AAAI*, pages 27538–27546. AAAI Press, 2025.

[96] A. Niemetz and M. Preiner. Bitwuzla. In *CAV (2)*, volume 13965 of *LNCS*, pages 3–17. Springer, 2023.

[97] M. Nyberg, D. Gurov, C. Lidström, A. Rasmusson, and J. Westman. Formal verification in automotive industry: Enablers and obstacles. In *ISoLA (4)*, volume 11247 of *LNCS*, pages 139–158. Springer, 2018.

 https://doi.org/10.52202/085713-2509

[98] S. Padhi, R. Sharma, and T. D. Millstein. Data-driven precondition inference with learned features. In *PLDI*, pages 42–56. ACM, 2016.

[99] A. Pakonen, T. Tahvonen, M. Hartikainen, and M. Pihlanko. Practical applications of model checking in the Finnish nuclear industry. In *10th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies (NPIC & HMIT 2017). American Nuclear Society*, pages 1342–1352, 2017.

[100] A. Paliwal, S. M. Loos, M. N. Rabe, K. Bansal, and C. Szegedy. Graph representations for higher-order logic and theorem proving. In *AAAI*, pages 2967–2974. AAAI Press, 2020.

[101] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin. Can large language models reason about program invariants? In *ICML*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR, 2023.

[102] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[103] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE, 2004.

[104] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[105] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. *ACM Trans. Program. Lang. Syst.*, 29(3):15, 2007.

[106] S. Polu and I. Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020.

[107] M. Preiner, N. Froleyks, and A. Biere. Hardware model checking competition (HWMCC) 2024. URL `https://hwmcc.github.io/2024/`.

[108] Z. Qin, K. Zhang, Y. Chen, J. Chen, and C. Fan. Learning safe multi-agent control with decentralized neural barrier certificates. In *ICLR*. OpenReview.net, 2021.

[109] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana. CLN2INV: learning loop invariants with continuous logic networks. In *ICLR*. OpenReview.net, 2020.

[110] S. Scherer, F. Lerda, and E. M. Clarke. Model checking of robotic control systems. In *ISAIRAS*, 2005.

[111] F. Schmitt, C. Hahn, M. N. Rabe, and B. Finkbeiner. Neural circuit synthesis from specification patterns. In *NeurIPS*, pages 15408–15420, 2021.

[112] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh. MachSMT: A machine learning-based algorithm selector for SMT solvers. In *TACAS (2)*, volume 12652 of *LNCS*, pages 303–325. Springer, 2021.

[113] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a SAT solver from single-bit supervision. In *ICLR (Poster)*. OpenReview.net, 2019.

[114] T. Seufert, F. Winterer, C. Scholl, K. Scheibler, T. Paxian, and B. Becker. Everything you always wanted to know about generalization of proof obligations in PDR. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 42(4):1351–1364, 2023.

[115] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. *Formal Methods Syst. Des.*, 48(3):235–256, 2016.

[116] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. Learning loop invariants for program verification. In *NeurIPS*, pages 7762–7773, 2018.

[117] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[118] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *FM*, volume 5850 of *LNCS*, pages 532–546. Springer, 2009.

[119] Y. Su, Q. Yang, Y. Ci, T. Bu, and Z. Huang. The rIC3 hardware model checker. In *CAV (1)*, volume 15931 of *LNCS*, pages 185–199. Springer, 2025.

[120] X. Sun, H. Khedr, and Y. Shoukry. Formal verification of neural network controlled autonomous systems. In *HSCC*, pages 147–156. ACM, 2019.

[121] T. Thorbjarnarson and N. Yorke-Smith. On training neural networks with mixed integer programming. *CoRR*, abs/2009.03825, 2020.

[122] D. Trivedi, J. Zhang, S. Sun, and J. J. Lim. Learning to synthesize programs as interpretable and generalizable policies. In *NeurIPS*, pages 25146–25163, 2021.

[123] C. Urban. The abstract domain of segmented ranking functions. In *SAS*, volume 7935 of *LNCS*, pages 43–62. Springer, 2013.

[124] C. Urban. FuncTion: An abstract domain functor for termination (competition contribution). In *TACAS*, volume 9035 of *LNCS*, pages 464–466. Springer, 2015.

[125] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Ann. Pure Appl. Log.*, 51(1-2):79–98, 1991.

[126] D. A. Vogel. *Medical Device Software Verification, Validation and Compliance*. Artech House, 2011.

[127] M. Wang, Y. Tang, J. Wang, and J. Deng. Premise selection for theorem proving by deep graph embedding. In *NIPS*, pages 2786–2796, 2017.

[128] W. Wang, Y. Hu, M. Tiwari, S. Khurshid, K. L. McMillan, and R. Miikkulainen. NeuroBack: Improving CDCL SAT solving using graph neural networks. In *ICLR*. OpenReview.net, 2024.

[129] M. Weißmann, S. Bedenk, C. Buckl, and A. C. Knoll. Model checking industrial robot systems. In *SPIN*, volume 6823 of *LNCS*, pages 161–176. Springer, 2011.

[130] G. Wu, W. Cao, Y. Yao, H. Wei, T. Chen, and X. Ma. LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In *ASE*, pages 406–417. ACM, 2024.

[131] M. Wu, T. Tsukada, H. Unno, T. Sekiyama, and K. Suenaga. Learning heuristics for template-based CEGIS of loop invariants with reinforcement learning. *arXiv:2107.09766*, 2021.

[132] K. Yang and J. Deng. Learning to prove theorems via interacting with proof assistants. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019.

[133] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *PLDI*, pages 106–120. ACM, 2020.

[134] J. Yao, R. Tao, R. Gu, and J. Nieh. Mostly automated verification of liveness properties for distributed protocols with ranking functions. *Proc. ACM Program. Lang.*, 8(POPL):1028–1059, 2024.

[135] H. Zhao, X. Zeng, T. Chen, and Z. Liu. Synthesizing barrier certificates using neural networks. In *HSCC*, pages 25:1–25:11. ACM, 2020.

[136] D. Zikelic, M. Lechner, T. A. Henzinger, and K. Chatterjee. Learning control policies for stochastic systems with reach-avoid guarantees. In *AAAI*, pages 11926–11935. AAAI Press, 2023.

[137] D. Zikelic, M. Lechner, A. Verma, K. Chatterjee, and T. A. Henzinger. Compositional policy learning in stochastic control systems with formal guarantees. In *NeurIPS*, 2023.

https://doi.org/10.52202/085713-2509

Model	LTL Specification (Pure-Liveness)
DELAY	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{sig}$
	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\left(\texttt{sig} \wedge \mathsf{X}\ \texttt{!sig}\right)$
LCD Controller	$\mathsf{FG}\ \texttt{lcd\_enable} \rightarrow \mathsf{GF}\ \texttt{ready}$
Blink	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{ledON}$
Thermocouple	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{get\_data}$
7-Segment	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{disp = 1}$
	$\mathsf{FG}\ \texttt{!rst} \rightarrow \left(\mathsf{GF}\ \texttt{disp = 0} \wedge \mathsf{GF}\ \texttt{disp = 1}\right)$
i2c Stretch	$\mathsf{FG}\left(\texttt{!rst} \wedge \texttt{ena}\right) \rightarrow \mathsf{GF}\ \texttt{strc}$
Pulse Width Modulation	$\mathsf{GF}\ \texttt{!pulse}$
VGA Controller	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{disp\_ena}$
UART Transmitter	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{wait}$
Load-Store	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{sig}$
Gray Counter	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{sig}$
	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\left(\texttt{sig} \wedge \mathsf{X}\ \texttt{!sig}\right)$
	$\mathsf{FG}\ \texttt{!rst} \rightarrow \left(\mathsf{GF}\ \texttt{sig} \wedge \mathsf{GF}\ \texttt{!sig}\right)$

Table 2: Model Name and LTL Specification in our Benchmark (Pure-Liveness)

Model	LTL Specification (Pure-Safety)
DELAY	$\mathsf{G}\ \texttt{!err}$
	$\mathsf{G}\left(\texttt{sig} \rightarrow \mathsf{X}\ \texttt{!sig}\right)$
LCD Controller	$\mathsf{G}\left(\left(\texttt{!lcd\_enable} \wedge \texttt{ready}\right) \rightarrow \mathsf{X}\ \texttt{ready}\right)$
	$\mathsf{G}\left(\mathsf{X}\ \texttt{wait} \rightarrow \texttt{busy}\right)$
Blink	$\mathsf{G}\left(\left(\texttt{mode1} \wedge \mathsf{X}\ \texttt{mode1}\right) \rightarrow \mathsf{X}\left(\texttt{cnt!=0}\right)\right)$
Thermocouple	$\mathsf{G}\left(\left(\texttt{ !spi\_busy } \wedge \texttt{ get\_data } \wedge \texttt{ !rst }\right) \rightarrow \mathsf{X}\ \texttt{get\_data}\right)$
7-Segment	$\mathsf{G}\left(\left(\texttt{!sig} \wedge \mathsf{X}\ \texttt{!sig} \wedge \texttt{!rst}\right) \rightarrow \left(\left(\texttt{ds0} \wedge \mathsf{X}\ \texttt{ds0}\right) \vee \left(\texttt{ds1} \wedge \mathsf{X}\ \texttt{ds1}\right)\right)\right)$
i2c Stretch	$\mathsf{G}\left(\left(\texttt{!strc} \wedge \mathsf{X}\ \texttt{strc}\right) \rightarrow \mathsf{X}\ \texttt{switch}\right)$
Pulse Width Modulation	$\mathsf{G}\left(\left(\texttt{pulseLB} \rightarrow \texttt{pulse}\right) \wedge \left(\texttt{!pulseUB} \rightarrow \texttt{!pulse}\right)\right)$
VGA Controller	$\mathsf{G}\left(\left(\texttt{en} \wedge \texttt{!rst}\right) \rightarrow \left(\left(\texttt{hs} \leftrightarrow \mathsf{X}\ \texttt{!hs}\right) \vee \left(\texttt{!hs} \leftrightarrow \mathsf{X}\ \texttt{hs}\right)\right)\right)$
	$\mathsf{G}\left(\left(\texttt{en} \wedge \texttt{!rst}\right) \rightarrow \left(\left(\texttt{vs} \leftrightarrow \mathsf{X}\ \texttt{!vs}\right) \vee \left(\texttt{!vs} \leftrightarrow \mathsf{X}\ \texttt{vs}\right)\right)\right)$
UART Transmitter	$\mathsf{G}\left(\left(\texttt{wait} \rightarrow \texttt{!busy}\right) \wedge \left(\texttt{transmit} \rightarrow \texttt{busy}\right)\right)$
Load-Store	$\mathsf{G}\left(\texttt{sig} \rightarrow \mathsf{X}\,\mathsf{X}\ \texttt{!sig}\right)$
Gray Counter	$\mathsf{G}\left(\left(\texttt{sig} \wedge \texttt{!rst}\right) \rightarrow \mathsf{X}\ \texttt{!sig}\right)$

Table 3: Model Name and LTL Specification in our Benchmark (Pure-Safety)

Model	LTL Specification (Safety-Liveness)
Delay	$\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{XG}\ (\ cnt < N\ \mathsf{U}\ \texttt{sig}\ )$
	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{FG}\ (\ cnt < N\ \mathsf{U}\ \texttt{sig}\ )$
LCD Controller	$\mathsf{G}\ \Big(\ \texttt{receive} \rightarrow\ (\ \texttt{receive}\ \mathsf{U}\ \texttt{ready}\ )\Big)$
Blink	$\mathsf{G}\ \texttt{!rst} \rightarrow\ \mathsf{G}\ \Big(\ \texttt{ledON} \rightarrow\ (\ \texttt{ledON}\ \mathsf{U}\ \texttt{mode0}\ )\Big)$
Thermocouple	$\mathsf{G}\ \texttt{!rst} \rightarrow\ \mathsf{G}\ \Big(\ \texttt{pause} \rightarrow\ (\ \texttt{pause}\ \mathsf{U}\ \texttt{get\_data}\ )\Big)$
7-Segment	$\mathsf{G}\ \texttt{!rst} \rightarrow\ \mathsf{G}\ \Big(\big((\ \texttt{ds0}\ \wedge\ \mathsf{X}\ \texttt{ds0})\ \vee\ (\ \texttt{ds1}\ \wedge\ \mathsf{X}\ \texttt{ds1})\big)\ \mathsf{U}\ \texttt{sig}\Big)$
i2c Stretch	$\mathsf{G}\ \texttt{!rst}\ \rightarrow\ \mathsf{G}\ \Big(\ \texttt{!strc} \rightarrow\ (\ \texttt{!strc}\ \mathsf{U}\ \texttt{switch}\ )\Big)$
Pulse Width Modulation	$\mathsf{GF}\ \texttt{!pulseUB}\ \wedge\ \mathsf{XG}\ \Big(\ \texttt{!pulseUB} \rightarrow\ \texttt{!pulse}\ \Big)$
VGA Controller	$\mathsf{G}\ \Big(\big(\ \texttt{Vcnt == 0}\ \big) \rightarrow\ \big((\ \texttt{Vcnt == 0}\ )\ \mathsf{U}\ (\ \texttt{Hcnt == 0}\ )\big)\Big)$
UART Transmitter	$\mathsf{G}\ \texttt{!rst} \rightarrow\ \mathsf{G}\ \Big(\ \texttt{busy} \rightarrow\ (\ \texttt{busy}\ \mathsf{U}\ \texttt{wait}\ )\Big)$
Load-Store	$\mathsf{G}\ \texttt{!rst} \rightarrow\ \mathsf{G}\ \Big(\ \texttt{modeUP} \rightarrow\ (\ \texttt{modeUP}\ \mathsf{U}\ \texttt{sig}\ )\Big)$
	$\mathsf{G}\ \texttt{!rst} \rightarrow\ \mathsf{X}\,\mathsf{G}\ \Big(\ \texttt{modeUP} \rightarrow\ (\ \texttt{!sig}\ \mathsf{U}\ \texttt{modeDOWN}\ )\Big)$
Gray Counter	$\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{XG}\ (\ cnt > 0\ \mathsf{U}\ \texttt{sig}\ )$
	$\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{FG}\ (\ cnt > 0\ \mathsf{U}\ \texttt{sig}\ )$

Table 4: Model Name and LTL Specification in our Benchmark (Safety + Liveness)

A Details of the Case Studies

We benchmark our tool on eleven RTL designs. Ten are adopted from the neural model-checking study of [63], which provided *only* pure-liveness properties. We extend every design with semantically natural pure safety and safety plus liveness specifications. Table 2 lists all pure-liveness formulas, Table 3 the pure-safety formulas, and Table 4 the safety-liveness formulas, all in Linear Temporal Logic (LTL). Below we discuss each design together with its full specification suite.

Delay (DELAY). A counter cnt raises a pulse sig after a fixed delay and is reset by rst. Liveness requires $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{sig}$ and, to prevent sig from sticking, $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ (\texttt{sig} \wedge \mathsf{X}\ \texttt{!sig})$. Safety bounds the counter, $\mathsf{G}\ \texttt{cnt} <= N$, and clears sig in the next cycle, $\mathsf{G}\ (\texttt{sig} \rightarrow \mathsf{X}\ \texttt{!sig})$. Safety-Liveness formulas constrain cnt until sig occurs, $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{XG}\ \big(\texttt{cnt} < N\ \mathsf{U}\ \texttt{sig}\big)$ and its eventually variant $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{FG}\ \big(\texttt{cnt} < N\ \mathsf{U}\ \texttt{sig}\big)$.

LCD Controller (LCD). After initialisation the controller waits for lcd_enable, switches from ready to send, and returns to ready. Liveness demands $\mathsf{FG}\ \texttt{lcd\_enable} \rightarrow \mathsf{GF}\ \texttt{ready}$. Safety preserves ready when lcd_enable is low, $\mathsf{G}\big((\texttt{!lcd\_enable} \wedge \texttt{ready}) \rightarrow \mathsf{X}\ \texttt{ready}\big)$, and asserts busy in wait, $\mathsf{G}\big(\mathsf{X}\ \texttt{wait} \rightarrow\ \texttt{busy}\big)$; for safety-liveness a receive phase persists until ready, $\mathsf{G}\big(\texttt{receive} \rightarrow (\texttt{receive}\ \mathsf{U}\ \texttt{ready})\big)$.

Blink (BLINK). A counter toggles led on every wrap and raises a one-cycle flag flg. Liveness asserts $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{ledON}$, while safety enforces staying in mode1 if cnt has not wrapped: $\mathsf{G}\big((\texttt{mode1} \wedge \mathsf{X}\ \texttt{mode1}) \rightarrow \mathsf{X}(\texttt{cnt} \neq 0)\big)$. The safety–liveness specification ensures that led remains on until mode0 is reached: $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{G}\big(\texttt{ledON} \rightarrow (\texttt{ledON}\ \mathsf{U}\ \texttt{mode0})\big)$.

Thermocouple (Tmcp.). The FSM cycles through start, get_data, pause. Liveness requires $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{get\_data}$. Safety keeps get_data active on an idle bus and safety-liveness keeps pause active until get_data becomes reachable: $\mathsf{G}\big(\texttt{!spi\_busy} \wedge \texttt{get\_data} \wedge \texttt{!rst} \rightarrow \mathsf{X}\ \texttt{get\_data}\big)$ and $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{G}\big(\texttt{pause} \rightarrow (\texttt{pause}\ \mathsf{U}\ \texttt{get\_data})\big)$.

 https://doi.org/10.52202/085713-2509

7-Segment (7-Seg). Two displays alternate unless reset. Liveness enforces $\mathsf{FG}\ \texttt{!rst} \rightarrow (\mathsf{GF}\ \texttt{disp = 0} \wedge \mathsf{GF}\ \texttt{disp = 1})$ and the single-display variant $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{disp = 1}$. Safety freezes the display in the absence of sig: $\mathsf{G}\big(\texttt{!sig}\wedge\mathsf{X}\texttt{!sig}\wedge\texttt{!rst} \rightarrow ((\texttt{ds0}\wedge\mathsf{X}\texttt{ds0})\vee(\texttt{ds1}\wedge\mathsf{X}\texttt{ds1}))\big)$ while safety-liveness holds the display until sig, $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{G}\big(((\texttt{ds0} \wedge \mathsf{X}\texttt{ds0}) \vee (\texttt{ds1} \wedge \mathsf{X}\texttt{ds1}))\ \mathsf{U}\ \texttt{sig}\big)$.

i2c Stretch (i2cS). Timing signals scl_clk and data_clk are generated according to bus frequency; stretch handles clock stretching. Liveness states $\mathsf{FG}\ (\texttt{!rst} \wedge \texttt{ena}) \rightarrow \mathsf{GF}\ \texttt{strc}$. Safety links a rising stretch to switch: $\mathsf{G}\big(\texttt{!strc} \wedge \mathsf{X}\texttt{strc} \rightarrow \mathsf{X}\texttt{switch}\big)$ meanwhile safety-liveness, under permanent reset-low, keeps stretch low until switch, $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{G}\big(\texttt{!strc} \rightarrow (\texttt{!strc}\ \mathsf{U}\ \texttt{switch})\big)$.

Pulse Width Modulation (PWM). An N-bit counter drives pulse; the design must regularly unset the pulse. Accordingly, liveness requires $\mathsf{GF}\ \texttt{!pulse}$. Two guard signals bound the duty-cycle window: pulseLB (minimum width) and pulseUB (maximum width). Safety enforces these bounds at every cycle, $\mathsf{G}\big((\texttt{pulseLB}\rightarrow\texttt{pulse}) \wedge (\texttt{!pulseUB}\rightarrow\texttt{!pulse})\big)$, and safety-liveness strengthens them with a fairness clause that forces the upper bound to be violated—and thus pulse to be low—infinitely often: $\mathsf{GF}\ \texttt{!pulseUB} \wedge \mathsf{XG}\big(\texttt{!pulseUB}\rightarrow\texttt{!pulse}\big)$.

VGA Controller (VGA). Horizontal and vertical counters drive disp_ena. Liveness: $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{disp\_ena}$. Safety flips hs and vs under disp_ena: $\mathsf{G}\big((\texttt{en} \wedge \texttt{!rst}) \rightarrow ((\texttt{hs} \leftrightarrow \mathsf{X}\texttt{!hs}) \vee (\texttt{!hs} \leftrightarrow \mathsf{X}\texttt{hs}))\big)$, $\mathsf{G}\big((\texttt{en} \wedge \texttt{!rst}) \rightarrow ((\texttt{vs} \leftrightarrow \mathsf{X}\texttt{!vs}) \vee (\texttt{!vs} \leftrightarrow \mathsf{X}\texttt{vs}))\big)$, and safety-liveness asserts a vertical–horizontal counter relationship $\mathsf{G}\big(\texttt{Vcnt==0} \rightarrow (\texttt{Vcnt==0}\ \mathsf{U}\ \texttt{Hcnt==0})\big)$.

UART Transmitter (UARTt). The FSM toggles between wait and transmit. Liveness: $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{wait}$. Safety couples busy with required modes: $\mathsf{G}\big((\texttt{wait} \rightarrow \texttt{!busy}) \wedge (\texttt{transmit} \rightarrow \texttt{busy})\big)$, and safety-liveness under permanent reset-low holds busy until wait, $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{G}\big(\texttt{busy} \rightarrow (\texttt{busy}\ \mathsf{U}\ \texttt{wait})\big)$.

Load–Store (LS). A counter counts up in load, down in store; sig triggers the switch. Liveness: $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{sig}$. Safety clears sig after two cycles: $\mathsf{G}\big(\texttt{sig} \rightarrow \mathsf{XX}\texttt{!sig}\big)$. The safety-liveness formulas hold modeUP until sig and, symmetrically, !sig until modeDOWN: $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{G}\big(\texttt{modeUP} \rightarrow (\texttt{modeUP}\ \mathsf{U}\ \texttt{sig})\big)$ and $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{XG}\big(\texttt{modeUP} \rightarrow (\texttt{!sig}\ \mathsf{U}\ \texttt{modeDOWN})\big)$.

Gray Counter (Gray). Counts in Gray code to minimise bit flips. Liveness: $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{GF}\ \texttt{sig}$, $\mathsf{FG}\texttt{!rst} \rightarrow \mathsf{GF}(\texttt{sig} \wedge \mathsf{X}\texttt{!sig})$, and $\mathsf{FG}\texttt{!rst} \rightarrow (\mathsf{GF}\texttt{sig} \wedge \mathsf{GF}\texttt{!sig})$. Safety clears sig one step later: $\mathsf{G}\big(\texttt{sig} \wedge \texttt{!rst} \rightarrow \mathsf{X}\texttt{!sig}\big)$. Safety-liveness bind the counter until sig: $\mathsf{G}\ \texttt{!rst} \rightarrow \mathsf{XG}(cnt > 0\ \mathsf{U}\ \texttt{sig})$ and its eventually counterpart $\mathsf{FG}\ \texttt{!rst} \rightarrow \mathsf{FG}(cnt > 0\ \mathsf{U}\ \texttt{sig})$.

Collectively, these benchmarks exercise handshake logic, clock stretching, display timing, PWM generation, UART serialisation, counter bounds, and Gray-code sequencing. Each property appears in pure-liveness, pure-safety, and safety-liveness form, enabling a comprehensive assessment across the complete verification spectrum.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims of this paper are stated in the abstract and elaborated in Section 1. There, we briefly outline the theoretical underpinnings of these claims and summarise experimental results that quantify the scalability of our approach.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalise to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 5 includes a dedicated *Strengths and Limitations* subsection, where we explicitly state our drawbacks and justify their reasonableness, along with a discussion of theoretical and practical constraints. Section 5 contains a class of problems (pure-safety) where we underperform. The accompanying *Threats to Validity* subsection critically evaluates the scope of our claims by motivating the benchmark selection used for comparison against alternative verification methods.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

https://doi.org/10.52202/085713-2509

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: We provide references for all theoretical foundations underpinning our work, including automata-theoretic LTL model checking, fair termination, inductive invariant and the use of ranking functions and inductive invariants in sec 2.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The *Implementation* subsection of Section 5 describes the full pipeline used in our prototype and specifies the versions of all external dependencies. It outlines the neural network architecture under both linear and automatic configurations, as well as the learning engines used. This section also states the configurations employed in our main experiments and ablation studies. In contrast, Section 3 addresses the theoretical underpinnings of the learning procedure, including dataset generation, the design of the neural architecture, and the SMT-based check step, as part of our iterative approach. Together, these sections provide sufficient detail to replicate our method for neural model checking of safety and liveness properties. Additionally, the *Experimental Setup* subsection discusses the versions of competing tools, the hardware environment used for evaluation, and the translation procedure from SystemVerilog to the appropriate input formats for each tool. For reproducibility and further implementation details, our code and benchmarks will be made publicly available.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example

(a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
(b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Alongside the paper, we provide a zip archive as per NeurIPS guidelines. It includes all benchmarks, experiment scripts, and a `README.md` with detailed instructions for reproducing our results.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 5, particularly the *Implementation* subsection, discusses our use of a single hyperparameter and outlines how it is selected. We also detail the tools used for the *learn* and *check* phases. Our approach to dataset generation is described in Section 3, and a full description of the benchmark suite is provided in Appendix A.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.

https://doi.org/10.52202/085713-2509

- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Because formal-verification tools are deterministic and must return provably correct answers, traditional variance measures—error bars and confidence intervals—are not meaningful. Instead for statistical significance, we run each tool once on a substantial benchmark suite of 634 instances and provide cactus plots, an experimental setup consistent with community practice in formal verification competitions such as HWMCC. Statistical evidence is conveyed by cactus plots (Figure 3(a–c); Figure 4(b-c)), which report, for every time cap up to $15\,\mathrm{min}$, the cumulative number of tasks each competitor completes. Scatter plots (Figure 3(d–f)) complement this view by plotting one point per benchmark task, exposing the full runtime distribution rather than aggregate summaries. We additionally annotate salient figures in our text—e.g., the percentage of tasks finished within a given cap and the count lying to the right of the $10\times$ diagonal—to underscore the consistent performance gaps observed across tools and throughout our ablation study.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The compute environment is described in the *Experimental Setup* subsection of Section 5, where we specify our Amazon EC2 instance (type and configuration) and report the total wall-clock time across all experiments. To avoid cluttering the paper with over 7 000 individual measurements (634 tasks × multiple settings), we provide the complete runtime data in the supplementary ZIP archive and through cactus plots. In the main paper, one can visualise the per-task runtime through the cactus (Figure 3(a–c);Figure 4(b-c)) and scatter plots 3(d-f)).

   Guidelines:

   - The answer NA means that the paper does not include experiments.

- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We affirm that all content presented in this work is disclosed with full academic integrity. All original contributions are properly cited, and our methodology is described in detail throughout the paper to ensure reproducibility. In addition, we provide our full implementation and experimental setup as part of the supplementary materials. Our experiments exclusively use synthetic data, with no human subjects involved, and are therefore fully aligned with ethical research standards. The goal of our method is to enhance the reliability and safety of computer systems. The benchmark hardware designs are drawn from prior work, which is cited accordingly, with one additional design introduced by us. While the pure-liveness specifications are taken from the same work, we developed additional pure-safety and safety-liveness specifications based on the semantics of each design. We have reviewed the NeurIPS Code of Ethics and confirm our strict adherence to its principles.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: As discussed in the introduction and conclusion, improving the correctness of hardware designs prior to fabrication contributes to the development of safer and more reliable systems. Avoiding the manufacture of flawed silicon also reduces material waste. We are not aware of any direct path by which our method could be applied to harmful ends.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.

https://doi.org/10.52202/085713-2509

- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: Our work contributes to improving the correctness of hardware systems. The benchmark designs used in our evaluation are extensions of a prior benchmark suite and consist of standard hardware modules drawn from well-known literature. These designs are in the public domain and do not pose any risk of misuse.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We include the licenses for all tools stated in section 5, and provide appropriate citations for each.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

Justification: We include the source code of our prototype and the extended benchmark suite as supplementary material to this paper, released under the MIT License.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: This paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: This paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

https://doi.org/10.52202/085713-2509

Justification: The core method development in this research does not involve large language models (LLMs) as any important, original, or non-standard components.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.