NeuroPDR: Integrating Neural Networks in the PDR Algorithm for Hardware Model Checking

Guangyu Hu[*†], Wei Zhang[*], Hongce Zhang[*†]
[*]Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
[†]Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China
Email: ghuae@connect.ust.hk, {wei.zhang, hongcezh}@ust.hk

Abstract—**The property-directed reachability (PDR) algorithm has been one of the most successful hardware model checking algorithms powering modern formal property verification tools. Inductive generalization is the key to the efficiency of the PDR algorithm. In this paper, we present NeuroPDR, a message-passing graph neural network that learns to generalize inductive clauses to accelerate the PDR algorithm. Experiments show that on average, the integration of NeuroPDR reduces around 26.4% of the time of convergence, and the network trained on one set of benchmarks can also benefit solving another. The speed-up and transferability indicate a promising future for hybrid model checkers with machine learning methods integrated.**

Index Terms—**formal verification, model checking, property directed reachability, inductive generalization**

I. Introduction

Hardware verification plays a crucial role in the modern chip design process. Formal property verification tools are commonly used to spot IC logic design bugs. Internally, these tools use model checking algorithms to check for violations of user-specified properties.

IC3 is one of the most successful SAT-based hardware model checking algorithms, proposed by A. Bradley [3]. It was further developed by Een *et al.* into the property-directed reachability (PDR) algorithm [7]. The key idea of the PDR algorithm is to incrementally build an inductive invariant through iteratively blocking bad states (those that violate a safety property) and their predecessors with relatively inductive clauses. These clauses are generated via a process called inductive generalization, one key procedure of the algorithm. The most commonly-used inductive generalization method is the minimal-inductive-clause (MIC) method [4]. Some variants may also factor in the counterexample-to-generalization (CTG) [13] or use an iterative inductive generalization algorithm [7] to obtain an inductive clause that will more likely constitute an inductive invariant. Despite these proposed algorithmic improvements, the inductive generalization methods so far are greedy in nature. They only have a local view of the current reachable states and therefore, may produce globally sub-optimal clauses which do not generalize well. Prior research on software model checking has pointed out similar issues, for example, myopic generalization and excessive generalization in a PDR implementation for software verification [21]. As a workaround, heuristic rules were added to serve as global guidance to lead the PDR algorithm out of the local optima. In this paper, we addressed a similar issue in hardware model checking while using machine learning methods to provide such global guidance.

There have been several existing works that use machine learning to guide the classic formal verification algorithms. For example, in Boolean satisfiability (SAT) solving, the NeuroCore [17] solver employed a graph neural network to predict the scores for variable selection in the classic conflict-driven-clause-learning algorithm framework. The integration of machine learning methods provides better heuristics while preserving the soundness and completeness of the algorithm. In this work, our NeuroPDR algorithm follows the same paradigm and takes advantage of a graph neural network to learn the circuit structure of the logic design under verification and propose heuristics for inductive generalization in the PDR algorithm. The soundness of the inferred clauses will be checked before acceptance so the prediction accuracy will at most affect the speed rather than the correctness. Our experiments show that the integration of NeuroPDR on average saves around 26.4% on a test set collected from the hardware model checking competition (HWMCC) benchmarks. To our best knowledge, this is the first work that integrates neural networks at the core of the PDR algorithm for hardware model checking. Specifically, in this paper, we make the following contributions:

- We propose a method to encode inductive generalization problems in hardware model checking into graphs.
- We design a message-passing graph neural network to infer the inductive clauses in the PDR algorithm.
- We propose a method to integrate the neural inductive generalization process into the classic PDR algorithm framework. This integration preserves soundness and completeness. It is also asynchronous — the inference process is kept away from the critical path of the algorithm to minimize the overhead of machine learning.
- We conduct experiments on a test set collected from the hardware model checking benchmarks. Our experiments show that the integration on average brings about 26.4%

The work has been supported in part by Guangdong Basic and Applied Basic Research Foundation no. 2022A1515110178 and by the Guangzhou-HKUST(GZ) Joint Funding Program (No. 2023A03J0013).

solving time reduction, and the network trained on one set of benchmarks can also accelerate solving another.

- We obtain a comprehensive dataset for neural inductive generalization, which is made publicly available at https://github.com/Gy-Hu/ML4PDR.

The rest of the paper is organized as follows. The next section introduces the technical background. Section III illustrates how inductive generalization can be improved. Section IV presents our NeuroPDR algorithm and how it is used to guide inductive generalization in PDR. Section V is the experiment, followed by related works and the conclusion.

II. Preliminaries

Hardware model checking examines whether a safety property P is valid on a state transition system $\langle \mathit{Var}, \mathit{Init}, \mathit{Tr} \rangle$, where Var is the set of state variables (along with the primed version of variables Var' that denote the next state), Init is a state predicate representing the initial states, and Tr is a transition relation. An input variable can be treated in the same way as a free state variable — one without specifying its next state function, therefore, we omit the input variables in this formulation. In this paper, we focus on the bit-level hardware model checking problems, usually presented in the AIGER format [1]. For a bit-level logic circuit, each state variable is a single-bit Boolean variable. Variables or their negations are called the literals. A *clause* is a disjunction of literals, while the conjunction of literals is called the *cube*.

When the PDR algorithm attempts to solve the hardware model checking problem, it incrementally constructs a sequence of over-approximation of reachable states within a bound of state transitions. This sequence is called the forward reachable series (FRS): $\{F_i\}$. Each F_i is a state predicate that describes all reachable states within i steps. F_i is also referred to as the *frame* and is often represented by a conjunction of clauses. FRS are constructed by iteratively checking whether bad states or their predecessors are reachable in bounded steps. If a bad state is found to be reachable from any initial state, the algorithm terminates with the result `UNSAFE` indicating there is a violation of property P. Otherwise, a clause is added to the frame to "remember" that the states are not reachable within the current bound. This essentially blocks (removes) some states in F. Whenever the FRS is found to be inductive ($\exists i. F_{i+1} \models F_i$), we say that the FRS converges and the algorithm terminates with result `SAFE`. Upon termination, the last frame becomes a safe inductive invariant that certifies the validity of property P.

A. Inductive Generalization in IC3/PDR

In the aforementioned process, when a bad state s is checked to be unreachable within a bound i, we need to come up with an inductive clause to block s. As s is usually a cube, $\neg s$ is in fact a clause, which could be used to block s on frame F_i. However, $\neg s$ only removes a single state. In practice, for faster convergence, we would like to generalize the clause $\neg s$ to block more states at a time.

Specifically, we would like to find a new clause c with fewer literals than $\neg s$ while preserving the validity of the following three formulas:

$$F_{i-1} \wedge c \wedge \mathit{Tr} \models c' \tag{1}$$

$$\mathit{Init} \models c \tag{2}$$

$$c \models \neg s \tag{3}$$

Here c' denotes the clause where all variables in c are replaced with their counterparts in the set of next state variables Var' and F_{i-1} is the frame prior to the bound. These three formulas essentially specify that c is relatively inductive, is a superset of the initial states and does not intersect with the bad state s.

To ensure formula (3) is valid, the inductive generalization procedure picks a subset of the literals in $\neg s$ with various heuristics. For example, the commonly-used minimal-inductive-clause (MIC) method [13] tries to remove as many literals as possible following a certain literal selection order. The clause generated from MIC is minimal because no more literals can be further removed. However, the clause is not necessarily minimum — there could be other literal selection ordering that results in even fewer literals.

B. Message Passing Graph Neural Network

Graph neural networks (GNNs) have been widely used in machine-learning applications like natural language processing and computer vision. Recent research has found that graph neural networks are also useful in formal verification. For instance, Selsam *et al.* used GNN to learn the structure of a conjunctive normal form (CNF) formula to predict its satisfiability [18]. GNN employs a message passing mechanism to gather useful structural information of the graph. Every node in the graph is given a hidden state, a d-dimensional vector $\boldsymbol{n}$. In each round, the hidden state of a node v is updated by aggregating messages from its neighboring nodes as shown by Equation (4):

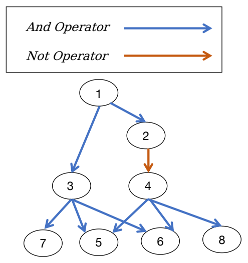
$$\begin{aligned} \boldsymbol{n}_v \leftarrow \mathit{Update}\big(&\boldsymbol{n}_v, \\ &\mathit{Agg}\left(\{\mathit{Msg}\left(\boldsymbol{n}_u\right) | u \in \mathit{Neighbor}(v)\}\right)\big) \end{aligned} \tag{4}$$

After several rounds of message passing, the hidden state vectors are expected to encode structural informatino of the graph and are finally used to predict labels specific to nodes (in the case of node classification) or the whole graph (graph classification) via a mapping $\mathit{Predict}$. The mapping Msg, $\mathit{Predict}$, and the hidden state update function Update are all trainable. The aggregation function Agg is chosen to be invariant to permutation.

III. Improving Inductive Generalization

The main idea of our work is to improve inductive generalization in the PDR algorithm by additional guidance from a neural network. Here we use a simple example as an illustration of this method.

Suppose we have a simple state transition system to verify, a 7-bit counter that counts from 0 to 100 (in decimal). We denote the state bits in the counter as $\mathit{cnt}[6..0]$, where $\mathit{cnt}[6]$



ID	Formula
1	$\neg(v_{14} \wedge v_{18} \wedge v_{22}) \wedge (v_{18} \wedge v_{22} \wedge v_{26})$
2	$\neg(v_{14} \wedge v_{18} \wedge v_{22})$
3	$v_{18} \wedge v_{22} \wedge v_{26}$
4	$v_{14} \wedge v_{18} \wedge v_{22}$
5	v_{22}
6	v_{18}
7	v_{26}
8	v_{14}

Fig. 1. Graph representation of formula $\neg(v_{14} \wedge v_{18} \wedge v_{22}) \wedge v_{18} \wedge v_{22} \wedge v_{26}$

is the most significant bit. The counter will be disabled when $cnt[6..0]$ is equal to $(1100100)_2$, so it will stop at $(100)_{10}$. We state a safety property P that $cnt[6..0]$ will not reach $(127)_{10}$, which is indeed valid. The bad state s (namely, $\neg P$) is when $cnt[6..0]$ is equal to $(1111111)_2$.

In order to show our property is valid, the PDR algorithm recursively blocks the bad state s in the construction of FRS. It first blocks s on F_1 by finding a clause that indicates s is not reachable in 1-step transition from the initial state. As stated in Section II, the clause is generated by selecting a subset of literals in $\neg s$ (namely, $\neg cnt[6] \vee \neg cnt[5] \vee ... \vee \neg cnt[0]$) such that the new clause c ensures Equation (1)-(3) hold. For the first frame, F_{i-1} is the initial state predicate ($\neg cnt[6] \wedge \neg cnt[5] \wedge ... \wedge \neg cnt[0]$) indicating all state bits are 0. As a typical inductive generalization will try to remove as many literals as possible, the resulting clause could simply be $\neg cnt[6]$, which is a minimal clause (and also the minimum clause in this case). $\neg cnt[6]$ indicates the counter will not transit to a state with a 1 in the most significant bit of cnt. This is true for the first frame, but it is not a universally correct statement. When the counter reaches 64, $cnt[6]$ will become 1, so the clause $\neg cnt[6]$ will be discarded eventually, but only after the algorithm has explored the first 64 transitions! This is a major source of inefficiency in the algorithm due to the improper inductive generalization that aims to find the minimal clauses. The clause is chosen based on the reachable states contained in merely the prior frame F_{i-1} without considering the global reachability of the state transition system. Therefore, it is a local generalization.

In our work, we aim to provide global guidance to the inductive generalization procedure in the PDR algorithm. Specifically, we would like to use a neural network to look into the structure of the circuit and predict the selection of literals based on the (inferred) global reachability. For this specific example, if the algorithm is aware that the counter possibly ranges from 0 to 100, it will keep more literals in c, for example, $\neg cnt[6] \vee \neg cnt[5] \vee cnt[4]$, which is indeed a valid statement and will make up as part of the global inductive invariant.

IV. GNN-guided Inductive generalization

In this section, we introduce how we encode the inductive generalization problem into a graph and use GNN to predict generalized clauses.

A. *Graph Encoding of Logic Formulas*

The idea of inductive generalization is to find a clause that satisfies Formula (1)-(3) and is also part of an inductive invariant. In order to find the suitable clause, we need to convey the relevant information about these formulas to the neural network. Prior works [17], [18] have shown that it is possible to represent a CNF formula using a graph. We further extend this idea to representing arbitrary Boolean logic formulas using a directed acyclic graph (DAG). Figure 1 illustrates the DAG representation of a Boolean logic formula, where each node represents either a Boolean operator or a variable. Operator nodes have non-zero outdegrees whereas variable nodes have zero outdegrees.

Because the AIGER format requires all circuits must be first converted into an equivalent zero-initialized state transition system, the initial predicate is not specific to the circuit. Therefore, we do not need to encode $Init$. As we are selecting a subset of literals from $\neg s$, we can consider encoding the following formula: $F_{i-1} \wedge \neg s \wedge Tr \wedge s'$ into a DAG, and have the neural network predict which literals in s (as well as s') can be dropped while maintaining the unsatisfiability of the formula. In fact, the formula to encode can be further simplified. As logic circuits are all *functional* transition systems (namely, Var' can be computed from a function over Var), we can substitute the primed variables in s' using the transition function (denoted as $s'|_{Tr}$) and omit Tr in the original formula. Moreover, as we are interested in global inductive generalization, we don't have to encode information about local reachability specific to a frame. So we may remove F_{i-1} and the remaining one is $(\neg s) \wedge (s'|_{Tr})$. These simplifications effectively reduce the size of the converted DAG and help to scale our method onto larger problems.

For example, suppose there is a cube $s := v_{14} \wedge v_{18} \wedge v_{22}$ to block, and within the transition relation, state variable v_{14} is updated by v_{18} in the next cycle, and v_{18} and v_{22} are updated by v_{22} and v_{26}. Following the above steps, in NeuroPDR we will construct the formula $(\neg s) \wedge (s'|_{Tr}) := \neg(v_{14} \wedge v_{18} \wedge v_{22}) \wedge v_{18} \wedge v_{22} \wedge v_{26}$ and convert it into DAG shown in Figure 1, which will be used to predict the inductive clause to block s.

B. *GNN Architecture*

As stated above, our GNN accepts the DAG of the formula $(\neg s) \wedge (s'|_{Tr})$ as the input. It predicts a 0-1 label for each variable (leaf) node. The label indicates if the variable (and thus the literal) can be removed from $\neg s$. This is essentially a node classification task.

1) Notation: We denote the number of variable nodes in the DAG as a, and the number of operator nodes as b. We use M_{adj} to refer to the adjacency matrix of the DAG. $M_{adj}(i,j) = 1$ means there is a directed edge from node i to node j. As variable nodes have 0-outdegree, we omit those rows in M_{adj}. Therefore, $M_{adj} \in \mathbb{R}^{b \times (a+b)}$. Each graph node is associated with a d-dimensional hidden state vector (the embedding). We use the matrix $N^{(t)} \in \mathbb{R}^{(a+b) \times d}$ to refer to the stacking of all node embeddings at an iteration t.

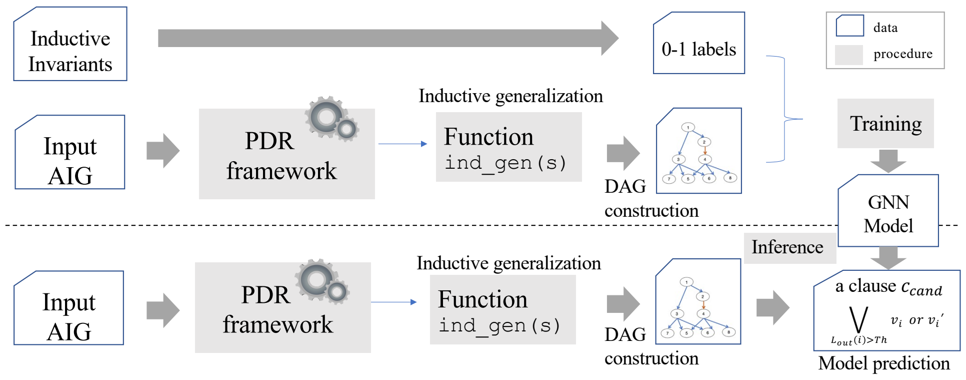


Fig. 2. Training and inference in NeuroPDR

2) NeuroPDR Architecture: NeuroPDR follows the classic message passing paradigm. Structural information of the graph will be embedded in the hidden state vectors after a number of iterations. Messages flow along the directed edges in the forward pass and against the edges in the backward pass.

In NeuroPDR, there are three multi-layer perceptrons (MLPs): $(\mathbf{C}_{\text{msg}}, \mathbf{B}_{\text{msg}}, \mathbf{L}_{\text{out}})$, and two Long Short Term Memory (LSTM) networks: $(\mathbf{C}_{\text{u}}, \mathbf{B}_{\text{u}})$. $\mathbf{C}_{\text{msg}}$ and $\mathbf{B}_{\text{msg}}$ map a node's embedding to a message in the forward/backward pass. $\mathbf{C}_{\text{u}}$ and $\mathbf{B}_{\text{u}}$ serve as the update functions of node embeddings in the forward/backward pass. A single iteration applies the following two updates for a backward pass and a forward pass:

$$\left(M_c^{(t+1)}, M^{(t+1)}\right) \leftarrow \mathbf{B}_{\text{u}}\left(\left[M_{adj}\mathbf{B}_{\text{msg}}\left(N^{(t)}\right), M_c^{(t)}\right]\right) \tag{5}$$

$$\left(N_c^{(t+1)}, N^{(t+1)}\right) \leftarrow \mathbf{C}_{\text{u}}\left(\left[M_{adj}^T\mathbf{C}_{\text{msg}}\left(M^{(t+1)}\right), N^{(t)}\right]\right) \tag{6}$$

Formula (5) and (6) are the results of instantiating Formula (4). $M_c^{(t)} \in \mathbb{R}^{b\times d}$ and $N_c^{(t)} \in \mathbb{R}^{(a+b)\times d}$ are the cell states for the two LSTMs: $\mathbf{B}_{\text{u}}$ and $\mathbf{C}_{\text{u}}$, both initialized to zero matrices. We choose LSTMs as the update function of hidden state vectors, as each node will go through a sequence of updates and will require sequential processing by recurrent neural networks. $M^{(t+1)} \in \mathbb{R}^{b\times d}$ holds the stacking of the intermediate results on operator node embeddings between the two passes. The multiplication with the adjacency matrix (or its transpose) in (5) and (6) indicates that the aggregation function we use is simply the sum of all incoming messages. The initial node embeddings are set to two trainable parameters $\boldsymbol{n}_\top$ or $\boldsymbol{n}_\bot$ depending on the nodes' Boolean results evaluated under the variable assignment in s.

After in total T iterations, we collect all embeddings on variable nodes and use $\mathbf{L}_{\text{out}}$ to map each into a scalar in $[0, 1]$, indicating the probability of keeping the variable in the result clause c. During inference, we only keep those variables whose probabilities are above a threshold Th. During training, we use the binary cross-entropy between the output probabilities and the training labels as the loss function. The training labels are collected from the clauses in the inductive invariants generated by an oracle PDR model checker. Training and inference of NeuroPDR are shown in Figure 2.

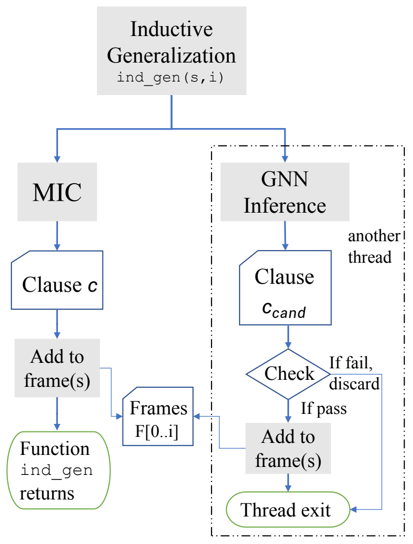


Fig. 3. Asynchronous integration of NeuroPDR

C. The Integration of NeuroPDR

Because the inductive generalization procedure is at the core of the PDR algorithm and being invoked frequently, extra overhead in the procedure may significantly slow down the overall algorithm. As the main PDR algorithm is usually running on a CPU, while the neural network inference usually takes place on GPUs, it takes time to transfer data between CPUs and GPUs. The network inference and data exchange will incur extra overhead to the inductive generalization procedure and therefore, should not be placed on the critical path. Here, we propose an asynchronous integration of NeuroPDR to hide the latency of invoking a neural network.

This asynchronous integration (shown in Figure 3) places the neural network inference on a separate thread, while having the main thread running the classic inductive generalization algorithm (e.g., MIC). It supplies extra global inductive clauses, in addition to the existing ones computed by MIC. In some sense, this is similar to clause sharing in the parallel PDR implementation [15], except that here the clauses to share come from a neural network. The PDR algorithm will not stop waiting for the neural network's output. When finished, the output clause c will be inserted into the frames at a later time.

For each output clause, we explicitly check if Formula (1) and (2) hold (as we are selecting from the variable subset, the requirement (3) is guaranteed automatically). We will discard the output clause upon failure. NeuroPDR will dynamically adjust the threshold Th based on the ratio of how many output clauses pass the checks. A lower passing ratio means variable removal is too aggressive and will result in a decrease in Th.

In this section, we present the architecture of NeuroPDR and how we design its integration with the core procedure of the PDR model checking algorithm.

V. Experiment

A. Experiment Setup

As machine learning frameworks are mostly available in Python, to allow an easier integration, in our prototype we make a Python implementation of the PDR algorithm. Our re-implementation follows the C++ code in IC3ref [2]. All experiments are performed on a Ubuntu 20.04.4 LTS x86_64 server equipped with Intel Xeon Platinum 8375C, an NVIDIA 3090 GPU, and 128GB memory. The configuration and hyper-parameters of our neural networks are listed in Table I.

B. Data Collection

As presented in Section IV-B, our NeuroPDR requires clauses from inductive invariants to serve as the training labels. We use Berkeley-ABC [5] as the oracle model checker and collect inductive invariants on 24 small-size safe AIGER problems from the benchmark of HWMCC07. The reasons for choosing HWMCC07 are (1) the problem sizes in the early competitions are relatively small and therefore, easier to train on; (2) we will later test our network also on HWMCC20 to see if NeuroPDR can transfer its knowledge learned from a simpler benchmark. On each of the 24 problems, we run the vanilla PDR algorithm and collect all inductive generalization queries when solving these cases. For each cube s to block, we find one clause c in the inductive invariant such that $c \models \neg s$ (this also guarantees that c contains no more variables than s), and mark the variables in c as the training label for s. In total, we collect 974 (s, c) pairs as the dataset.

C. Training

We divide the dataset into training, validation, and test sets following the ratio of 3:1:1. We add weight for the positive cases in the BCE loss to counteract the unbalanced 0-1 labels

TABLE I
CONFIGURATIONS AND HYPERPARAMETERS

Parameter / Configuration	Value
MLP architecture	128-128-128
Activation function	ReLU
Embedding dimension (d)	128
#. rounds of message passing (T)	512
#. training epochs	256
Learning rate	1e-4
Weight decay	1e-10
Weights for 0-1 labels	1:2
Optimizer	Adam

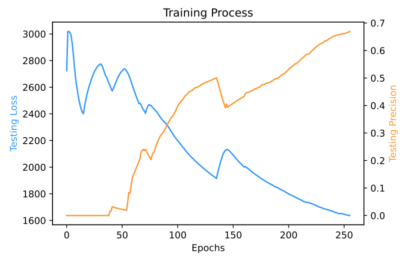


Fig. 4. Loss and precision during training

in the dataset. We count the precision (true positives divided by the sum of true and false positives) during training. The statistics in Figure 4 show NeuroPDR reaches 0.7 precision in the end.

D. Model Checking Results

Although our Python implementation is incomparable to a C++ model checker, we can still measure the relative speed before and after integrating NeuroPDR on the Python implementation. In total, we use 42 cases from HWMCC07 and HWMCC20 that can be solved by the Python implementation in one hour, and seven of them are industry cases. We also count the problem sizes in terms of AIG nodes of these cases to show the scale of the problems. The results are listed in Table II. In the table, $t_{vanilla}$ is the solving time of PDR algorithm without NeuroPDR integration. It uses

TABLE II
RESULTS OF EXPERIMENTS

Benchmark	Name	AIG Size	$t_{vanilla}$	$t_{NeuroPDR}$
HWMCC07	eijk.S1196.S	552.0	8.28	**7.48**
	eijk.S1238.S	580.0	8.29	**7.43**
	eijk.S208c.S	90.0	**26.28**	35.80
	eijk.S208o.S	86.0	9.92	**8.84**
	eijk.S344.S	171.0	633.28	**567.85**
	eijk.S386.S	201.0	38.28	**35.05**
	eijk.S510.S	299.0	1076.02	**935.96**
	eijk.S641.S	228.0	12.08	**8.73**
	eijk.S713.S	227.0	10.84	**9.57**
	eijk.S820.S	477.0	95.37	**83.75**
	eijk.S832.S	507.0	101.70	**92.85**
	eijk.S953.S	489.0	246.83	**215.43**
	nusmv.guidance^1.C	952.0	19.32	27.62
	nusmv.guidance^5.C	960.0	652.23	**638.03**
	nusmv.guidance^8.C	958.0	2849.75	**1723.15**
	nusmv.reactor^3.C	687.0	118.11	**105.77**
	nusmv.syncarb10^2.B	93.0	17.45	**13.04**
	nusmv.tcas-t^2.B	1552.0	2137.45	**1499.19**
	nusmv.tcas^2.B	1508.0	2135.46	**931.82**
	texas.PI_main^01.E	4120.0	**24.00**	37.55
	texas.PI_main^05.E	4125.0	46.95	**26.65**
	texas.PI_main^15.E	4123.0	7.48	**5.01**
	vis.4-arbit^1.E	174.0	**41.89**	44.57
	vis.arbiter.E	232.0	16.17	**10.44**
	vis.coherence^2.E	828.0	35.41	**32.07**
	vis.coherence^3.E	827.0	88.67	**56.15**
HWMCC20	**cal159 (industry)**	22277.0	240.47	**62.58**
	cal161 (industry)	22308.0	117.40	**100.56**
	cal162 (industry)	22278.0	121.51	**62.30**
	elevator.4.prop1-func-interl	1387.0	117.37	**52.63**
	gen10 (industry)	3964.0	21.03	**15.57**
	gen12 (industry)	4139.0	21.94	**16.66**
	gen14 (industry)	4139.0	22.56	**16.62**
	gen21 (industry)	3953.0	20.96	**15.46**
	picorv32-check-p05	11958.0	1726.59	**1457.06**
	picorv32-check-p09	11960.0	76.05	**54.96**
	picorv32-check-p20	11997.0	1159.78	**824.31**
	qspiflash_dualflexpress_divfive-p016	926.0	**8.19**	8.44
	qspiflash_dualflexpress_divfive-p022	926.0	16.92	**11.40**
	qspiflash_dualflexpress_divthree-p111	1014.0	**1242.22**	1690.66
	rast-p11	29601.0	62.90	**25.62**
	stack-p2	21022.0	680.30	**281.86**

counterexample-to-generalization (CTG) in the minimal inductive clause (MIC) method [13] for inductive generalization in PDR. The column $t_{NeuroPDR}$ measures the solving time of our proposed method. We can see, NeuroPDR saves the solving time in most of the cases. Only in a few cases, for example `qspiflash_dualflexpress_divthree-p111`, there is a slow-down. We investigate these cases and find that the slow-down is mainly due to the extra clauses generated and inserted into the frames. Although the actual SAT solving time only marginally increases, the time of converting a frame into SAT queries grows significantly. This is problem specific to Python implementation of PDR. As a C++ implementation is usually more efficient in such conversion, we expect the overhead of extra clauses to be a lesser issue for a C++ PDR implementation.

Among the test cases, the integration with NeuroPDR on average brings about 26.4% solving time reduction. For the test cases in HWMCC20, the average is about 17.0%. This shows a good transferability of our method on unseen cases. As the future work, we will integrate NeuroPDR into a C++ model checker to support efficiently testing more problems.

VI. Related Works

A. Machine Learning for Model Checking

There have been several existing works that apply machine learning methods to model checking, mainly in the domain of software verification. For example, Zhu *et al.* proposed a data-driven CHC solver [23] that uses linear classification and decision trees to synthesize invariants for programs. ICE [10], [11] is another data-driven method for learning inductive invariants from positive, negative, and implication samples. It has also inspired quite a few later works [6], [22]. Code2Inv employed graph neural networks and reinforcement learning for loop invariant synthesis [20]. In addition, there were also other works that take advantage of machine learning methods in other formal applications [8], [9], [16], [19]. Among them, ROPEY [14] is the closest work compared to this paper as it also focused on improving inductive generalization in the PDR algorithm. It used techniques borrowed from natural language processing (NLP) and worked on software verification problems encoded in linear integer/real arithmetics (LIA/LRA), a completely different setting from hardware verification, where the NLP encoding for LIA/LRA formulas cannot be easily extended to logic circuits. Whereas, this paper considers hardware verification, and uses graph neural networks to infer a better solution of inductive generalization in the domain of hardware model checking.

B. Optimizing Inductive Generalization in the PDR Algorithm

Inductive generalization is a key factor to the overall performance of the PDR algorithm. Prior research has presented various inductive generalization methods [4], [7], [13]. Alberto Griggio *et al.* compared six different inductive generalization methods with experiments showing that a suitable inductive generalization method can reduce around 12% cumulative time and solve 11% more problems [12]. As it is hard to find the best inductive generalization method for a specific set of model checking problems, it may take considerable efforts to manually tune the inductive generalization heuristics. Therefore, in our work, we propose to use machine learning methods to automatically learn such heuristics as an alternative.

VII. Conclusion

Machine learning techniques have already been adopted in many EDA applications. However, in the domain of formal verification, it is still unclear how to best leverage machine learning in formal property verification tools. This paper investigates one way of integrating graph neural networks to a core hardware model checking algorithm — the PDR algorithm. We hope our work will help open up the possibility of AI in the domain of IC formal verification.

References

[1] Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond (2011)
[2] Bradley, A.R.: IC3 reference implementation. https://github.com/arbrad/IC3ref, accessed: 2022-11-11
[3] Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI. pp. 70–87. Springer (2011)
[4] Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD. pp. 173–180. IEEE (2007)
[5] Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: CAV. pp. 24–40. Springer (2010)
[6] Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. TACAS (2018)
[7] Eén, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134. IEEE (2011)
[8] Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: FMCAD. pp. 1–9. IEEE (2018)
[9] Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: CAV. pp. 813–829. Springer (2013)
[10] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV. pp. 69–87. Springer (2014)
[11] Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. POPL **51**(1), 499–512 (2016)
[12] Griggio, A., Roveri, M.: Comparing different variants of the ic3 algorithm for hardware model checking. TCAD **35**(6), 1026–1039 (2015)
[13] Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD. pp. 157–164. IEEE (2013)
[14] Le, N., Si, X., Gurfinkel, A.: Data-driven optimization of inductive generalization. In: FMCAD. p. 86 (2021)
[15] Marescotti, M., Gurfinkel, A., Hyvärinen, A.E., Sharygina, N.: Designing parallel PDR. In: FMCAD. pp. 156–163. IEEE (2017)
[16] Prabhu, S., Madhukar, K., Venkatesh, R.: Efficiently learning safety proofs from appearance as well as behaviours. In: International Static Analysis Symposium. pp. 326–343. Springer (2018)
[17] Selsam, D., Bjørner, N.: Guiding high-performance sat solvers with unsat-core predictions. In: SAT. pp. 336–353. Springer (2019)
[18] Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. In: ICLR (2019)
[19] Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: European Symposium on Programming. pp. 574–592. Springer (2013)
[20] Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2inv: A deep learning framework for program verification. In: CAV. pp. 151–164. Springer (2020)
[21] Vediramana Krishnan, H.G., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: CAV. pp. 101–125. Springer (2020)
[22] Vizel, Y., Gurfinkel, A., Shoham, S., Malik, S.: IC3-flipping the E in ICE. In: VMCAI. pp. 521–538. Springer (2017)
[23] Zhu, H., Magill, S., Jagannathan, S.: A data-driven chc solver. PLDI **53**(4), 707–721 (2018)