Boosting Constrained Horn Solving by Unsat Core Learning

Parosh Aziz Abdulla*[1], Chencheng Liang*[1], and Philipp Rümmer*[1,2]

[1] Uppsala University, Uppsala, Sweden
chencheng.liang@it.uu.se
parosh.abdulla@it.uu.se
[2] University of Regensburg, Regensburg, Germany
philipp.ruemmer@ur.de

Abstract. The Relational Hyper-Graph Neural Network (R-HyGNN) was introduced in [1] to learn domain-specific knowledge from program verification problems encoded in Constrained Horn Clauses (CHCs). It exhibits high accuracy in predicting the occurrence of CHCs in counterexamples. In this research, we present an R-HyGNN-based framework called MUSHyperNet. The goal is to predict the Minimal Unsatisfiable Subsets (MUSes) (i.e., unsat core) of a set of CHCs to guide an abstract symbolic model checking algorithm. In MUSHyperNet, we can predict the MUSes once and use them in different instances of the abstract symbolic model checking algorithm. We demonstrate the efficacy of MUSHyperNet using two instances of the abstract symbolic model-checking algorithm: Counter-Example Guided Abstraction Refinement (CEGAR) and symbolic model-checking-based (SymEx) algorithms. Our framework enhances performance on a uniform selection of benchmarks across all categories from CHC-COMP, solving more problems (6.1% increase for SymEx, 4.1% for CEGAR) and reducing average solving time (13.3% for SymEx, 7.1% for CEGAR).

Keywords: Automatic program verification · Constrained Horn clauses · Graph Neural Networks.

1 Introduction

Constrained Horn Clauses (CHCs) [2] are logical formulas that can describe program behaviors and specifications. Encoding program verification problems in CHCs and solving them (checking CHCs' satisfiability) has been an active research area for a number of years [3,4]. If the encoded CHCs are satisfiable, the corresponding program verification problem is safe; if not, it is unsafe.

Solving a set of CHCs means that we either find an interpretation for the predicate (relation) symbols and variables that satisfies all the clauses, or prove

* Author names in alphabetical order. The theory presented in the paper was developed by Abdulla, Liang, and Rümmer, the implementation is by Liang and Rümmer, evaluation was done by Liang.

that no such interpretation exists. Various techniques, such as Counterexample Guided Abstraction Refinement (CEGAR) [5] and IC3 [6], have been utilized for this purpose. However, due to the undecidability of solving CHCs, we need carefully designed or tuned heuristics for specific instances.

In this paper, we consider Minimal Unsatisfiable Subsets (MUSes) [7] of sets of CHCs to support the solving process. Given an unsatisfiable set of CHCs, each MUS is a subset that is again unsatisfiable, but removing any CHC from an MUS makes it satisfiable. Understanding MUSes of a set of CHCs can guide solvers to focus on error-prone clauses [8, Section 3.1]: for an unsatisfiable set of CHCs, evaluating the satisfiability of MUSes first can quickly identify unsatisfiable CHCs, eliminating the need for a comprehensive check. In a satisfiable set of CHCs, examining MUSes first can provide a better starting point for refining potentially problematic constraints. This guidance can help the solver converge towards a solution more efficiently. Manually designed heuristics to find MUSes involves summarizing and generalizing the features of a set of CHCs from examples, which can be replaced by data-driven methods.

Various studies that apply deep learning to formal methods for verification have been published in the recent past, e.g., [9,10,11,12]. Primarily, deep learning serves as a feature extractor, which can automatically summarize and generalize program features from examples, alleviating the need for manual crafting and tuning of various heuristics. In addition, the idea of representing logic formulas as graphs and using Graph Neural Networks (GNNs) [13] to guide solvers has been employed in many successful studies [14,15,16,17].

However, we are not aware of any study that applies GNNs to guide CHC-based symbolic model checking techniques. The main contribution of this paper is to train a GNN model to predict MUSes of a set of CHCs. We train a GNN using the CHC-R-HyGNN framework [1] to predict values between 0 and 1, representing the probabilities of CHCs being elements of MUSes. For example, we assume that a set $\mathcal{C} = \{c_1, c_2, c_3\}$ of CHCs has one MUS $\{c_1, c_2\}$. The predicted probabilities for c_1, c_2, and c_3 being in the MUS could be 0.9, 0.8, and 0.1, respectively. We propose several strategies that use the predicted probability to guide an abstract symbolic model checking algorithm. The strategies can be instantiated for different algorithms on CHCs, such as CEGAR- and symbolic execution-based satisfiability checkers.

Figure 1 depicts an overview of our framework MUSHyperNet. Firstly, we encode the program verification problem into a set of CHCs. Secondly, we use the graph encoder in the CHC-R-HyGNN framework [1] to convert the set of CHCs into a graph format. Then, we train a GNN model named Relational Hypergraph Neural Network (R-HyGNN) to predict the probability of each CHC occurring in MUSes. Finally, we employ the predicted probabilities to guide the abstract symbolic model checking algorithm by determining the sequence for processing each CHC in the set of CHCs.

We utilize the same benchmarks as in [1], comprising 17 130 Linear Integer Arithmetic (LIA) problems from the CHC-COMP 2021 repository [18] for training and evaluating our approach. Further details about the benchmark can

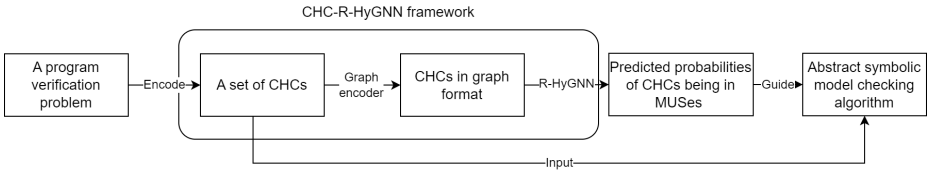


Fig. 1: The CHC-R-HyGNN framework [1] (represented by the round box) comprises a CHC graph encoder and a GNN known as the Relational Hypergraph Neural Network (R-HyGNN), which is capable of handling hypergraphs. In our previous work, we introduced the CHC-R-HyGNN framework that employs various proxy tasks to generalize valuable information for constructing heuristics for CHC-encoded program verification problems.

be found in [19, Table 1]. The problems for evaluation are uniformly selected from this benchmark and can be found in a public repository [20] The experimental results show an improvement of up to 4.1% and 6.1% in the number of solved problems for the CEGAR and SymEx algorithms, respectively. Additionally, the average solving time demonstrates enhancements of 7.1% and 13.3% for the CEGAR and SymEx algorithms, respectively. In other words, MUSHyperNet can increase the number of solved problems and decrease the solving time for problems similar to those in the CHC-COMP benchmark. To the best of our knowledge, this is the first time unsat core learning has been used successfully in the context of CHCs.

In summary, our contributions are as follows:

- We develop a GNN-based framework named MUSHyperNet, which trains a GNN to predict the MUSes of CHCs and utilizes the predicted probabilities to guide an abstract symbolic model checking algorithm.
- We explore GNN models trained on different datasets and methods for applying predicted MUSes to guide two instances of the model checking algorithm.
- We evaluate MUSHyperNet on 383 linear and 488 non-linear LIA problems, uniformly sampled from the CHC-COMP benchmark [19]. The improvements in the number of solved problems and average solving time are up to 6.1% and 13.3% for the SymEx, and 4.1% and 7.1% for the CEGAR.

2 Preliminaries

We first introduce required notation for multi-sorted first-order logic, and define Constrained Horn Clauses (CHCs) and the encoding of a program verification problem as CHCs. Finally, we explain basic concepts of Graph Neural Network (GNN) and introduce Relational Hyper Graph Neural Network (R-HyGNN).

2.1 Notations

We assume familiarity with standard multi-sorted first-order logic (e.g., see [21]). A first-order language $\mathcal{L}$ is defined by a signature $\Sigma = (\mathcal{S}, \mathcal{R}, \mathcal{F}, \mathcal{X})$, where $\mathcal{S}$ is

a non-empty set of sorts; $\mathcal{R}$ is a set of fixed-arity predicate (relation) symbols, each of which is associated with a list of argument sorts; $\mathcal{F}$ is a set of fixed-arity function symbols, each of which is associated with a list of argument sorts and a result sort; and $\mathcal{X} = \bigcup_{s \in \mathcal{S}} \mathcal{X}_s$ is a set of sorted variables, where $\mathcal{X}_s$ are the variables of sort s. A *term* t is a variable from $\mathcal{X}$, or an n-ary function symbol $f \in \mathcal{F}$ applied to terms $t_1, \ldots, t_n$ of the right sorts. An *atomic formula* (*atom*) of $\mathcal{L}$ is of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{R}$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms of the right sorts. A *formula* is a Boolean literal *true*, *false*, an atom, or obtained by applying logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$ and quantifiers $\forall$, $\exists$ to formulas. We write implications both left-to-right ($\varphi \rightarrow \psi$) and right-to-left ($\psi \leftarrow \varphi$). A formula is *closed* if all variables occurring in the formula are bound by quantifiers.

A *multi-sorted structure* $\mathcal{M} = (\mathcal{U}, I)$ for $\mathcal{L}$ consists of a set $\mathcal{U} = \bigcup_{s \in \mathcal{S}} \mathcal{U}_s$, being the union of non-empty domains $\mathcal{U}_s$ of each sort $s \in \mathcal{S}$, and an interpretation I such that $I(s) = \mathcal{U}_s$ for every sort $s \in \mathcal{S}$; for each n-ary predicate symbol $p \in \mathcal{R}$ with argument sorts $s_1, \ldots, s_n$, $I(p) \subseteq U_{s_1} \times \cdots \times U_{s_n}$; and for each n-ary function symbol $f \in \mathcal{F}$ with argument sorts $s_1, \ldots, s_n$ and result sort s, $I(f) \in U_{s_1} \times \cdots \times U_{s_n} \rightarrow U_s$. A *variable assignment* β for the structure $\mathcal{M} = (\mathcal{U}, I)$ is a function $\mathcal{X} \rightarrow \mathcal{U}$ that maps each variable $x \in \mathcal{X}_s$ to an element of the corresponding domain $\mathcal{U}_s$. Given $\mathcal{L}$, a structure $\mathcal{M} = (\mathcal{U}, I)$, and a variable assignment β, the *evaluation* of a term or formula is performed by the function $val_{\mathcal{M},\beta}$, defined by $val_{\mathcal{M},\beta}(x) = \beta(x)$ for a variable $x \in \mathcal{X}$; $val_{\mathcal{M},\beta}(f(t_1, \ldots, t_n)) = I(f)[val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n)]$ for a function $f \in \mathcal{F}$; and $val_{\mathcal{M},\beta}(p(t_1, \ldots, t_n)) = \mathit{true}$ iff $(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n)) \in I(p)$. The evaluation of compound formulas is defined as is common. When $\mathcal{M}$ is clear from the context, we also write val_β instead of $val_{\mathcal{M},\beta}$.

We say that a formula φ is *satisfied* in $\mathcal{M}, \beta$ if $val_{\mathcal{M},\beta}(\varphi) = \mathit{true}$, and that it is *satisfiable* (SAT) if it is satisfied by some $\mathcal{M}, \beta$. We say a set Γ of formulae *entails* a formula φ, denoted $\Gamma \models \varphi$, if φ is satisfied whenever all formulas in Γ are satisfied.

Example 1. We assume a language $\mathcal{L}$ consisting of a sort s, two constants a and b, a unary function symbol f with argument and result of sort s; a binary predicate symbol p with two arguments of sort s. A structure $\mathcal{M} = (U_s, I)$ can be defined by $U_s = \mathbb{Z}$, $I(a) = 1$, $I(b) = 2$, $I(f)[x] = x + 2$, and $I(p)[x, y] = x \leq y$. The formula $\varphi = p(a, b) \rightarrow p(x, f(a))$ is satisfied in $\mathcal{M}$ by a variable assignment $\beta(x) = 1$ since $val_\beta(\varphi) = I(p)[val_\beta(a), val_\beta(b)] \rightarrow I(p)[val_\beta(x), I(f)[val_\beta(x)]] = \neg(1 \leq 2) \vee 1 \leq (1 + 2)$ is true. The formula φ is satisfiable since $val_\beta(\varphi) = \mathit{true}$ for $\mathcal{M}$ and a variable assignment $\beta(x) = 1$.

2.2 Constraint Horn Clauses

To introduce the notion of constrained Horn clauses, we assume a fixed base signature $\Sigma = (\mathcal{S}, \mathcal{R}, \mathcal{F}, \mathcal{X})$, as well as a unique structure $\mathcal{M}$ over this signature, forming the background theory. In this paper, we mainly consider the background theory of Linear Integer Arithmetic (LIA), following the SMT-LIB

standard [22]. We further assume a set $\mathcal{R}_C$ of additional relation symbols that is disjoint from $\mathcal{R}$, which will be used to formulate the head and body of clauses.

Definition 1 (Constrained Horn clause). *Given signature Σ and the set $\mathcal{R}_C$, a* Constrained Horn Clause *(CHC) is a closed formula in the form*

$$\forall \bar{x}.\ H \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi, \tag{1}$$

where $\bar{x}$ is a vector of variables; H is either false or an atom $p(t_1, \ldots, t_n)$ with $p \in \mathcal{R}_C$; the relation symbols $p_1, \ldots, p_n$ are elements of $\mathcal{R}_C$; and $\bar{t}_1, \ldots, \bar{t}_n$ and φ are vectors of terms and a formula over Σ, respectively. We call H the head *and $p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi$ the* body *of the clause, respectively. We call the formula φ in the body the* constraint *of the clause.*

For convenience, in many places we leave out the quantifiers $\forall \bar{x}$ when writing clauses. A CHC without atoms in its body (the case $n = 0$) is called a *fact*. If the body of a CHC contains zero or one atom, the CHC is called *linear*. Otherwise, it is called *non-linear*.

Solving CHCs boils down to searching interpretations of the relation symbols $\mathcal{R}_C$ that satisfy the CHCs, assuming that all background symbols from Σ are interpreted by the fixed structure $\mathcal{M}$:

Definition 2 (Satisfiability of a CHC). *A CHC h is* satisfiable *if there is a structure $\mathcal{M}_C = (\mathcal{U}, I_C)$ for the extended signature $\Sigma_C = (\mathcal{S}, \mathcal{R} \cup \mathcal{R}_C, \mathcal{F}, \mathcal{X})$ such that (i) I_C coincides with I on Σ, and (ii) $\mathcal{M}_C$ satisfies h. A set $\mathcal{C}$ of CHCs is* satisfiable *if there is an extended structure $\mathcal{M}_C$ simultaneously satisfying all clauses in $\mathcal{C}$.*

Encoding Program Verification Problems using CHCs. A program verification problem involves checking whether a program adheres to its specified behavior. One approach to verification is to transform the problem into determining the satisfiability of a set of CHCs. This can be done, for instance, by encoding the partial correctness of a procedural imperative program into a negated Existential positive Least Fixed-point Logic (E+LFP) formula [4] using the weakest precondition calculus. Generally, encodings are designed such that the set of CHCs is satisfiable if and only if a program is safe. Various encoding schemes for different programming languages have been introduced in the literature, e.g., [4].

2.3 Graph Neural Networks

A Graph Neural Network (GNN) [13] is a type of neural network that consists of Multi-Layer Perceptrons (MLPs) [23]. GNNs operate on graph-structured data with nodes and edges, making them suitable for logic formulas which can naturally be represented as graphs. A GNN can take a set of typed nodes and edges as input and output a set of feature representations (vectors of real numbers) associated with the properties of the nodes. We refer to them as *node representations* in the rest of the sections.

The Message-Passing based GNN (MP-GNN) [24] is a type of GNN model. It utilizes an iterative message-passing algorithm in which each node in the graph aggregates messages from its neighboring nodes to update its own node representation. This mechanism assists in identifying the inner connections within substructures, such as terms and atoms, in graph represented logic formulae.

Formally, let $G = (V, E)$ be a graph, where V is the set of nodes and E is the set of edges. Let x_v be the initial node representation (a vector of random real numbers) for node v in the graph, and let N_v be the set of neighbors of node v. An MP-GNN consists of a series of T message-passing steps. At each step t, every node v in the graph updates its node representation as follows:

$$h_v^t = \phi_t(\rho_t(\{h_u^{t-1} \mid u \in N_v\}), h_v^{t-1}), \tag{2}$$

where $h_v^t \in \mathbb{R}^n$ is the updated node representation for node v after t iterations. The initial node representation, h_v^0, is usually derived from the node type and given by x_v. The node representation of u in the previous iteration $t-1$ is h_u^{t-1}, and node u is a neighbor of node v. $\rho_t : (\mathbb{R}^n)^{|N_v|} \to \mathbb{R}^n$ is a aggregation function with trainable parameters (e.g., a MLP followed by sum, min, or max) that aggregates the node representations of v's neighboring nodes at the t-th iteration. $\phi_t : \mathbb{R}^n \to \mathbb{R}^n$ is a function with trainable parameters (e.g., a MLP) that takes the aggregated node representation from ρ_t and the node representation of v in previous iteration as input, and outputs the updated node representation of v at the t-th iteration. MP-GNN assumes a node can capture local structural information from t-hop's neighbors by updating the node representation using aggregated representations of the neighbor nodes.

The final output of the MP-GNN could be the set of updated node representations for all nodes in the graph after T iterations. These node representations can be used for a variety of downstream tasks, such as node classification or graph classification.

Relational Hyper-Graph Neural Network (R-HyGNN) [1] is an extension of one MP-GNN called Relational Graph Convolutional Networks (R-GCN) [25], and it is specifically designed to handle labeled hypergraphs.

A labeled (typed) hypergraph is a hypergraph where each vertex (node) and hyperedge is assigned a type from a predefined set of types. Formally, a labeled hypergraph LHG is defined as a tuple $LHG = (V, E, \lambda_V, L_V, L_E)$, where V is a set of elements called vertices (or nodes), L_E is a set of pair consisting of a label (type) r and the number of nodes k under the label r, $E \subseteq V^* \times L_E$ is a set of hyperedges in which each hyperedge consists of a non-empty subsets of V and a pair $(r, k) \in L_E$. Here, $\lambda_V : V \to L_V$ is a labeling function that assigns a type from the set L_V to each vertex in V. The L_V is a set of possible types (labels) for the vertices V.

The node representation updating rule of R-HyGNN for one node v at timestep t is

$$h_v^t = \mathrm{ReLU}\Big(\sum_{\substack{(w_1,\ldots,w_k,(r,k)) \\ \in E}} \sum_{\substack{i \in \{1,\ldots,k\}, \\ w_i = v}} W_{r,i}^t \cdot \|(h_{w_1}^{t-1}, \ldots, h_{w_{i-1}}^{t-1}, h_{w_{i+1}}^{t-1}, \ldots, h_{w_k}^{t-1})\Big), \tag{3}$$

where the pair $(r, k) \in L_E$ is the edge type (relation) and the number of node for a edge $(w_1, \ldots, w_k, (r, k)) \in E$, $W^t_{r,i}$ is a matrix of learnable parameters in time step t for node $v = w_i$ in the edge with type r. There are $|L_E| \times \sum_{(r,k)\in L_E}(k) \times t$ matrices of learnable parameters in total. Here, $\|(h^{t-1}_{w_1}, \ldots, h^{t-1}_{w_{i-1}}, h^{t-1}_{w_{i+1}}, \ldots, h^{t-1}_{w_k})$ means concatenate v' neighbour node representations in time step $t-1$. The initial node representation h^0_v is derived from the node types L_V.

Intuitively, to update the representation of a node, R-HyGNN first concatenates the neighbor representations of node v for each edge from the previous time step $t-1$. It then multiplies the concatenated neighbor representations by the corresponding matrix of trained parameters (i.e., $W^t_{r,i}$) to derive a local representation of v. Next, it aggregates the local node representations (e.g., by addition). In other words, $\sum_{\substack{(w_1,\ldots,w_k,(r,k))\\ \in E}} \sum_{\substack{i\in\{1,\ldots,k\},\\ w_i=v}} W^t_{r,i} \cdot \|(h^{t-1}_{w_1}, \ldots, h^{t-1}_{w_{i-1}}, h^{t-1}_{w_{i+1}}, \ldots, h^{t-1}_{w_k})$ can be abstract to $\rho_t(\|(h^{t-1}_{w_1}, \ldots, h^{t-1}_{w_{i-1}}, h^{t-1}_{w_{i+1}}, \ldots, h^{t-1}_{w_k}))$ where ρ_t is an aggregation function with trainable parameters in $W^t_{r,i}$. Finally, it applies a ReLU function [26] as the update function ϕ_t. This function takes the aggregated local node representations as input and produces the final node representation h^t_v. This updating process for a single node recurs t times.

3 Abstract Symbolic Model Checking for CHCs

The goal of our work is to utilize GNNs to obtain improved state space exploration methods for CHCs. To this end, in this section we introduce an abstract formulation of CHC state space exploration, covering both the classical CEGAR approach and exploration in the style of symbolic execution.

3.1 Satisfiability Checking for CHCs

Our Algorithm 1 checks the satisfiability of a given a set $\mathcal{C}$ of CHCs by constructing an abstract reachability hyper-graph (ARG). An ARG is an over-approximation of the facts $p(\bar{a})$ that are logically entailed by $\mathcal{C}$; by demonstrating that the atom *false* does not follow from $\mathcal{C}$, it can be shown that $\mathcal{C}$ is satisfiable. Since each node of an ARG can represent a whole set of facts, a finite ARG can be a representation even of infinite models of a set $\mathcal{C}$.

We first give an abstract definition of ARGs that does not mandate any particular symbolic representation of sets of $p(\bar{a})$. We later introduce two instances of this abstract framework.

Like in Section 2.2, we denote the set of relation symbols used in CHCs by $\mathcal{R}_C$. For a k-ary relation symbol $p \in \mathcal{R}_C$ with argument sorts $s_1, \ldots, s_k$, we write $\mathcal{R}_p = \mathcal{P}(U_{s_1} \times \cdots \times U_{s_k})$ for the set of possible relations represented by p.

Definition 3. *An* abstract reachability graph *for a set $\mathcal{C}$ of CHCs is a hyper-graph (V, E), where*

- *the set V of nodes is a set of pairs (p, R), with p being a relation symbol and $R \in \mathcal{R}_p$;*

- $E \subseteq V^* \times \mathcal{C} \times V$ *is a set of hyper-edges labelled with clauses. For every edge* $(v_1, \ldots, v_n, h, v_0) \in E$*, it is the case that:*
  - *the head of a clause* h *is not false, i.e.,* h *is of the form* $\forall \bar{x}.\ p_0(\bar{t}_0) \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi$*;*
  - *nodes* $v_0, \ldots, v_n$ *correspond to the head and the body of* h*, i.e., for* $i \in \{0, \ldots, n\}$ *it is the case that* $v_i = (p_i, R_i)$ *for some* $R_i \in \mathcal{R}_{p_i}$*;*
  - R_0 *over-approximates the facts implied by the clause* h*:*

$$R_0 \supseteq \left\{ val_\beta(\bar{t}_0) \mid \begin{array}{l} val_\beta(\varphi) = true \text{ and } val_\beta(\bar{t}_i) \in R_i \text{ for } i = \{1, \ldots, n\} \\ \text{for some variable assignment } \beta \end{array} \right\}. \tag{4}$$

The algorithm starts with an empty ARG, and then adds nodes and edges to it until the ARG is *complete,* which intuitively means that all possible non-trivial edges are present in the graph. To define the notion of a complete ARG, we first need to characterize what it means for a clause to correspond to a feasible edge of the ARG:

Definition 4. *A clause* $h \in \mathcal{C}$ *is* feasible *for nodes* $v_1, \ldots, v_n \in V$ *of an ARG* (V, E) *if*

- h *is of the form* $\forall \bar{x}.\ H \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi$*;*
- *nodes* $v_1, \ldots, v_n$ *correspond to the body of* h*, i.e., for* $i \in \{1, \ldots, n\}$ *it is the case that* $v_i = (p_i, R_i)$ *for some* $R_i \in \mathcal{R}_{p_i}$*;*
- *the constraints imposed by* φ *and* $v_1, \ldots, v_n$ *are not contradictory, i.e., there is a variable assignment* β *such that* $val_\beta(\varphi) = true$ *and* $val_\beta(\bar{t}_i) \in R_i$ *for* $i = \{1, \ldots, n\}$*.*

Definition 5. *An ARG* (V, E) *is complete for a set* $\mathcal{C}$ *of CHCs if*

- *for every CHC* $h \in \mathcal{C}$ *that has a head different from false, and that is feasible for* $v_1, \ldots, v_n \in V$*, there is some edge* $\langle (v_1, \ldots, v_n), h, v_0 \rangle \in E$*;*
- *there is no CHC* $h \in \mathcal{C}$ *with head false that is feasible for any* $v_1, \ldots, v_n \in V$*.*

We can finally observe that complete ARGs correspond to models of the clause set $\mathcal{C}$.

Lemma 1. *A set* $\mathcal{C}$ *of CHCs has a complete ARG iff* $\mathcal{C}$ *is satisfiable.*

Algorithm 1 describes how ARGs can be constructed for a given clause set $\mathcal{C}$. The algorithm starts with an empty ARG (V, E), and maintains a queue $Q \subseteq \mathcal{C} \times V^*$ of feasible edges to be added to the graph next. The queue is initialized with the clauses with empty body, representing the initial states of a program. Once the queue runs empty, the constructed ARG is complete and the set $\mathcal{C}$ has been shown to be satisfiable.

In each iteration, in lines 5–6 an element $(h, \bar{v})$ is picked and removed from the queue Q. If the head of h is *false* (line 7), the edge to be added might be part of a witness of unsatisfiability of $\mathcal{C}$. In this case, it has to be checked whether the nodes $\bar{v}$ are over-approximate (line 8); this can happen when the containment

Input: A set $\mathcal{C}$ of CHCs
Output: Satisfiability of $\mathcal{C}$
Initialise: $V := \emptyset, E := \emptyset, Q := \{(h, ()) \mid h \in \mathcal{C}, h = \forall \bar{x}.\ H \leftarrow \varphi\}$

```
1  while true do
2      if Q is empty then
3          return satisfiable
4      else
5          Pick (h, v̄) ∈ Q to be considered next (guided by GNNs)
6          Q := Q \ {(h, v̄)}
7          if the head in h is false then
8              if derivation of false is genuine then
9                  return unsatisfiable
10             else
11                 Refine over-approximations
12                 Delete all affected nodes in (V, E)
13                 Regenerate elements in Q
14             end
15         else
16             Assume h = ∀x̄. p0(t̄0) ← p1(t̄1) ∧ ··· ∧ pn(t̄n) ∧ φ
17             Compute new node u = (p0, R0) for (h, v̄)
18             if u ∉ V then
19                 V := V ∪ {u}
20                 Q := Q ∪ { (d, (w1, ..., wm)) | d ∈ C is feasible for w1, ..., wm,
                                                   u ∈ {w1, ..., wm} ⊆ V }
21             end
22             E := E ∪ {(v̄, h, u)}
23         end
24     end
25 end
```

Algorithm 1: Abstract symbolic model checking algorithm for checking satisfiability of CHCs

in (4) is sometimes strict in the constructed ARG, and occurs in particular when instantiating the abstract algorithm as CEGAR (see Section 3.2). The over-approximation can then be refined in lines 11–13.

If the head of h is not *false*, a further edge is added to the graph by computing in line 17 some set R_0 satisfying (4). If the resulting node u is new in the graph, the queue Q is updated by adding possible outgoing edges for u (line 19–20).

In this paper, we apply the GNN-based guidance in line 5. Specifically, we presume that the GNN can predict the probability of h being in MUSes for each $q = (h, \bar{v}) \in Q$. We then combine this probability with certain features of q, such as the number of iterations $q \in Q$ has been waiting, to calculate a priority of q. When selecting an element q from Q, we consult this priority. We explain more details in Section 4.

3.2 CEGAR- and Symbolic Execution-Style Exploration

We now discuss two concrete instantiations of Algorithm 1. The first one, in this paper called SymEx, resembles the symbolic execution [27] of a program, and represents the relation R in an ARG node (p, R), for a k-ary relation symbol p, as a formula over free variables $z_1, \ldots, z_k$.

To obtain SymEx, in line 17 of Algorithm 1 the relation R_0 is derived from the nodes $\bar{v}$ by simple symbolic manipulation. Assuming that $v_i = (p_i, R_i)$ for $i \in \{1, \ldots, n\}$, and the relations R_i are all represented as formulas, we can define:

$$R_0[z_1, \ldots, z_k] \;=\; \exists \bar{x}.\; \bar{z} = \bar{t}_0 \wedge R_1[\bar{t}_1] \wedge \cdots \wedge R_n[\bar{t}_n] \wedge \varphi$$

where the notation $R_i[\bar{t}_i]$ expresses that the terms $\bar{t}_i$ are substituted for the free variables $\bar{z}$. In our implementation on top of the CHC solver Eldarica [28], the formula $R_0[z_1, \ldots, z_k]$ is afterwards simplified by eliminating the quantifiers, albeit only in a best-effort way by running the built-in formula simplifier of Eldarica. In SymEx, since no over-approximation is applied, the test in line 8 always succeeds, and lines 11–13 are never executed.

Our second instantiation, called CEGAR, is designed following counterexample-guided abstraction refinement [5,29] with Cartesian predicate abstraction [30]. In this version of the algorithm, we assume that a finite pre-defined set Π_p of predicates is available for every relation symbol p. If p is k-ary, then the elements of Π_p are formulas over free variables $z_1, \ldots, z_k$. The relation R in a node (p, R) is now represented as a subset of Π_p.

In line 17, the set R_0 is computed by determining the elements of Π_{p_0} that are entailed by the body of clause h:

$$R_0 \;=\; \{\phi \in \Pi_{p_0} \mid \bar{z} = \bar{t}_0 \wedge \bigwedge R_1[\bar{t}_1] \wedge \cdots \wedge \bigwedge R_n[\bar{t}_n] \wedge \varphi \models \phi\}$$

The notation $\bigwedge R_i[\bar{t}_i]$ denotes the conjunction of the elements of R_i, with $\bar{t}_i$ substituted for the free variables $\bar{z}$.

Over-approximation in CEGAR stems from the fact that a chosen set of predicates Π_p will oftentimes not be able to exactly represent a relation; the constructed ARG might then include facts $p(\bar{a})$ that are not actually entailed by $\mathcal{C}$. In line 8, the algorithm therefore has to verify that discovered derivations of *false* are genuine. This is done by collecting the clauses that were used to derive the nodes $\bar{v}$ in the ARG and constructing a counterexample tree. If the counterexample turns out to be *spurious,* further predicates are added to the sets Π_p, for instance using tree interpolation [31], in lines 11–13.

4 Guiding CHC Solvers using MUSes

In this section, we begin by defining the notion of Minimal Unsatisfiable Sets (MUSes), then we detail the process of collecting three types of training labels using the MUSes. Following that, we explain the various strategies of employing the predicted probability of a CHC being in MUSes to guide Algorithm 1.

4.1 Minimal Unsatisfiable Sets

Throughout the section, assume that $\mathcal{C}$ is an unsatisfiable set of CHCs.

Definition 6 (Minimal Unsatisfiable Set). *A subset $U \subseteq \mathcal{C}$ is a* Minimal Unsatisfiable Set *(MUS) if U is unsatisfiable and for all CHCs $h \in U$ it is the case that $U \backslash \{h\}$ is satisfiable.*

Intuitively, MUSes of a set of CHCs encoding a program correspond to minimal counterexamples (i.e., a subset of program statements) witnessing the incorrectness of the program. MUSes are therefore good candidates for guiding CHC solvers towards the critical clauses, and we aim at predicting MUSes using GNNs.

The number of MUSes can, however, be exponential in the number of CHCs in $\mathcal{C}$. We therefore consider the *union*, *intersection*, and a particular *single* MUS for $\mathcal{C}$. Denoting the set of all MUSes of $\mathcal{C}$ by $\mathrm{MUS}(\mathcal{C})$, those are:

$$\mathcal{C}^{\text{union}}_{\text{MUSes}} = \bigcup \mathrm{MUS}(\mathcal{C}), \qquad \mathcal{C}^{\text{intersection}}_{\text{MUSes}} = \bigcap \mathrm{MUS}(\mathcal{C}),$$
$$\mathcal{C}^{\text{single}}_{\text{MUSes}} = \operatorname*{argmax}_{U \in \mathrm{MUS}(\mathcal{C})} \mathrm{numAtom}(U),$$

where $numAtom(U)$ is the total number of atoms of the CHCs in U, and $\mathcal{C}^{\text{single}}_{\text{MUSes}}$ is some MUS that maximizes the total number of atoms. The three clause sets can be computed using the OptiRica extension of the Eldarica Horn solver [32].

Intuitively, $\mathcal{C}^{\text{union}}_{\text{MUSes}}$ includes all information about possible MUSes and encourages the algorithm to go through all possible error-prone areas. In contrast, $\mathcal{C}^{\text{intersection}}_{\text{MUSes}}$ only takes the intersection of all MUSes which can guide the algorithm to only focus on the most suspicious areas. $\mathcal{C}^{\text{single}}_{\text{MUSes}}$ is one of MUSes and corresponds to a long path in the ARG, given that a high number of atoms is associated with a large number of nodes. We believe a long path contains intricate information, which is challenging for human to parse, but easier for a deep-learning-based model to find.

We form three types of Boolean clause labelings by using $\mathcal{C}^{\text{union}}_{\text{MUSes}}$, $\mathcal{C}^{\text{intersection}}_{\text{MUSes}}$, and $\mathcal{C}^{\text{single}}_{\text{MUSes}}$, respectively. For instance, for $C^{\text{union}}_{\text{MUSes}}$, we obtain the labels

$$l^{\text{union}}(c) = \begin{cases} 1 & \text{if } h \in \mathcal{C}^{\text{union}}_{\text{MUSes}} \\ 0 & \text{if } h \in \mathcal{C} \setminus \mathcal{C}^{\text{union}}_{\text{MUSes}} \end{cases}$$

4.2 MUS-Based Guidance for CHC Solvers

Eldarica Heuristics The implementations of CEGAR and SymEx in the Horn solver Eldarica [28] by default use fixed, hand-written selection heuristics in line 5 of Algorithm 1. Such heuristics are defined by a ranking function $r : Q \to \mathbb{Z}$ that maps every element in the queue to an integer; in line 5, the element of Q is picked that minimizes r. The standard implementation of r used for CEGAR is defined by

$$EldCEGARRank(q) = numPredicate(q) + birthTime(q) + falseClause(q) \ ,$$

where $numPredicate((h, \bar{v})) = \sum_{(p,R)\in\bar{v}} |R|$ is the total number of predicates occurring in the considered nodes of the ARG; $birthTime(q)$ is the iteration (as an integer) in which q was added to Q;[3] and $falseClause((h, \bar{v}))$ is 0 if the head of h is not *false*, and some big integer otherwise. The rationale behind the ranking function is that nodes with few predicates tend to subsume nodes with many predicates, every clause should be picked eventually, and clauses with head *false* will trigger either termination of the algorithm or abstraction refinement.

In SymEx, for a node (p, R), we define *numConstraint(R)* to be the number of conjuncts of R. For instance, for $R = (x > 1 \wedge y < 0)$ we would get $numConstraint(R) = 2$. The default Eldarica ranking function for SymEx is defined by

$$EldSymExRank((h, \bar{v})) = \sum_{(p,R)\in\bar{v}} numConstraint(R) \ .$$

Similar to *numPredicate*, the intuition behind *EldSymExRank* is that nodes with larger formulas (more restrictions) tend to be subsumed by nodes with smaller formulas (fewer restrictions).

MUS-guided Heuristics We now introduce several new ranking functions defined with the help of MUSes. For this, suppose that $\mathcal{C}$ is the set of CHCs that a Horn solver is applied on. Under the assumption that $\mathcal{C}$ is unsatisfiable, we obtain three labeling functions l^{union}, $l^{\text{intersection}}$, l^{single} that are able to point out clauses to be prioritized in Algorithm 1.

In practice, of course, the status of $\mathcal{C}$ will initially be unknown. We therefore use three GNNs to *predict* labels $l^{\text{union}}(h)$, $l^{\text{intersection}}(h)$, $l^{\text{single}}(h)$, respectively, given just the set $\mathcal{C}$ and some clause $h \in \mathcal{C}$ as input. We interpret the prediction of a GNN as a *probability* $P(h)$ of a clause h to be in the union, intersection, or the single MUS set, and use those probabilities to define ranking functions. Table 1 lists several candidate ranking functions in terms of this membership probability P, where P stands for one of P^{union}, P^{single}, or $P^{\text{intersection}}$.

We consider two ways to convert probabilities to integers that can be used in the ranking function. In $rank_P(q)$, the elements $q = (h, \bar{v})$ of the queue Q are first sorted in descending order of $P(h)$; the number $rank_P(q)$ is the position of q in this sequence. This means that $rank_P(q)$ ranges from 0 to $|Q| - 1$, and elements with large probability $P(h)$ will be assigned small rank.

In $norm_P(q)$, we assume that $\min_P = \min\{P(h) \mid (h, \bar{v}) \in Q\}$ and $\max_P = \max\{P(h) \mid (h, \bar{v}) \in Q\}$ denote the minimum and maximum probability, respectively, among elements of Q. The normalized value $norm_P(q) \in [0, 1]$ is defined by

$$norm_P((h, \bar{v})) = \frac{P(h) - \min_P}{\max_P - \min_P}$$

In the ranking functions, we multiple $norm_P(q)$ with a negative coefficient *coef* and round the result to the nearest integer.

[3] Strictly speaking, this information cannot be computed from q, it is in practice stored as an additional field of the queue elements.

Table 1: Ranking function for queue elements $q \in Q$ in Algorithm 1, where $P \in \{P^{\text{union}}, P^{\text{single}}, P^{\text{intersection}}\}$.

Algorithm	Name	Ranking function
CEGAR	Fixed	$EldCEGARRank(q)$
	Random	$RandomRank$
	Score	$coef \cdot norm_P(q)$
	Rank	$rank_P(q)$
	R-Plus	$rank_P(q) + EldCEGARRank(q)$
	S-Plus	$coef \cdot norm_P(q) + EldCEGARRank(q)$
	R-Minus	$rank_P(q) - EldCEGARRank(q)$
	S-Minus	$coef \cdot norm_P(q) - EldCEGARRank(q)$
SymEx	Fixed	$EldSymExRank(q)$
	Random	$RandomRank$
	Score	$coef \cdot norm_P(q)$
	Rank	$rank_P(q)$
	R-Plus	$rank_P(q) + EldSymExRank(q) + birthTime(q)$
	S-Plus	$coef \cdot norm_P(q) + EldSymExRank(q) + birthTime(q)$
	R-Minus	$rank_P(q) - EldSymExRank(q) - birthTime(q)$
	S-Minus	$coef \cdot norm_P(q) - EldSymExRank(q) - birthTime(q)$
	Two-queue	$\begin{cases} \text{R-Minus}, & 80\% \textit{ probability} \\ \text{Random}, & 20\% \textit{ probability} \end{cases}$

The *RandomRank* function ensures that each CHC has an equal opportunity to be selected in each iteration. The Two-queue case denotes a setup with two queues, Q_1 and Q_2, each used with a certain probability in line 5 of Algorithm 1. We list just one such combination, which alternates between queues with the R-Minus and Random functions: there's an 80% chance we use the R-Minus queue and a 20% chance we use the Random queue.

5 Design of Model

As shown in Figure 1, within the CHC-R-HyGNN framework, we first encode a set of CHCs into a graph format, and then we build a GNN model consisting of an encoder, GNN layers, and a decoder.

5.1 Encode CHCs to Graph Representation

We apply the two (hyper-)graph representations of CHCs defined in [1]. We will briefly describe their main features here.

The first graph representation is called a *constraint graph* (CG). This graph encapsulates syntactic information by using nodes to represent each symbol and connecting them with binary edges. Each CHC and each predicate symbol is represented by a unique node. Terms and formulas are represented using their

abstract syntax trees (AST). CHC nodes are connected to the constituent atoms and constraints by binary edges. Within one CHC, common sub-expressions are represented by the same nodes. Different hyper-graph node and edges types are used to distinguish the various encoded operators (see Section 2.3).

The second graph representation is the *control- and data-flow hypergraph* (CDHG). This graph is designed to capture both control- and data-flow within CHCs using hyperedges, and therefore captures the semantics of CHCs more directly than GCs. Similar to the CG, in the CDHG, each CHC is represented by a unique node, and the atoms are rendered in the same way as in CG. Unlike the CG, the CDHG uses *control-flow hyperedges* (CFHEs) to describe the control-flow from the body to the head in each CHC, guarded by the constraint of the CHC. Furthermore, the CDHG uses *data-flow hyperedges* (DFHEs) to represent data-flow from the terms in the body to the terms in the head. These data-flows are also guarded by the constraint.

5.2 Model Structure

The R-HyGNN model consists of three sub-components: (i) encoder, (ii) R-HyGNN [1] (a GNN), and (iii) decoder:

$$\text{(i) } \mathcal{H}_0 = \text{encoder}(V, \lambda_V, L_V), \qquad \text{(ii) } \mathcal{H}_t = \text{R-HyGNN}(\mathcal{H}_0, E, L_E),$$
$$\text{(iii) } \hat{\mathcal{L}} = \text{decoder}(\mathcal{H}_t)\ .$$

The encoder in (i) first maps each node in V to an integer according to the node's type determined by λ_V and L_V. Then, it passes the encoded integers to a single-layer neural network (embedding layer) to compute initial node representations $\mathcal{H}_0$. The R-HyGNN in (ii) is a GNN with its node representation updating rule defined in (3). It takes the initial node representations ($\mathcal{H}_0$), edges (E), and edge types (L_E) as input and outputs the updated node representations $\mathcal{H}_t$. The decoder in (iii) first identifies the node that denote the CHC instead of the variables, atoms, or other elements in the CHC, then we collect the representations of these CHC nodes $\mathcal{H}_t^{\text{CHCs}}$ from all node representations $\mathcal{H}_t$. Finally, we pass $\mathcal{H}_t^{\text{CHCs}}$ to a set of fully connected neural networks to compute the probability of each CHC being in the MUSes, and $\hat{\mathcal{L}}$ is a set of probabilities.

The parameters of neural networks in (i), (ii), and (iii) are optimized together by minimizing the binary cross-entropy loss [33] between $\hat{\mathcal{L}}$ and the true labels $\mathcal{L}$, i.e.,

$$loss = -\frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i log(\hat{\mathcal{L}}_i) + (1 - \mathcal{L}_i) log(1 - \hat{\mathcal{L}}_i). \tag{5}$$

6 Evaluation

We first describe how the benchmarks are split for training and evaluation, then list some important parameters. Finally, we show and explain the experimental results. This work can be reproduced by following the instructions in [34].

Table 2: Distribution of the number of problems for both training and evaluation. T/O, N/A, and Evail. denote timeout, not available, and evaluation, respectively.

Linear LIA problems						Non-linear LIA problems					
8705						8425					
Benchmarks for training				Holdout set		Benchmarks for training				Holdout set	
7834 (90%)				871 (10%)		7579 (90%)				846 (10%)	
UNSAT		SAT	T/O	Eval.	N/A	UNSAT		SAT	T/O	Eval.	N/A
1585		4004	2245	383	488	3315		4010	254	488	358
Train	Valid	N/A				Train	Valid	N/A			
782	87	716				1617	180	1518			

6.1 Benchmark

The training and evaluation data are specified in Table 2, and available in [20]. There are 8705 linear and 8425 non-linear LIA problems, taken from CHC-COMP 2021 [19]. We first uniformly reserved 10% of the benchmarks as holdout set for the final evaluation. We ran the CEGAR in Eldarica using a 3-hour timeout to solve the remaining 90% of benchmarks, leading to three groups of benchmarks: SAT, UNSAT, and timeout. For UNSAT problems, we also computed the MUS sets $\mathcal{C}^{\text{union}}_{\text{MUSes}}$, $\mathcal{C}^{\text{intersection}}_{\text{MUSes}}$, and $\mathcal{C}^{\text{single}}_{\text{MUSes}}$. Some problems in UNSAT were eliminated in this process (N/A, for both training and evaluation) because the problems were trivial (already solved by the Eldarica preprocessor), the process of extracting MUSes timed out (3 hours), or a timeout occurred when encoding CHCs as graphs. The remaining problems are divided into training (90%) and validation (10%) datasets.

6.2 Parameters

We select the hyper-parameters for the GNN empirically, and according to the experimental results from [1]. We set the vector size of the initial node representation and the number of neurons in all intermediate neural network layers to 32; we also set the number of message-passing steps to 8 (i.e., applying (3) 8 times). The constant *coef* in Table 1 is -1000. Other parameters and the instructions to reproduce these results can be found in [20].

6.3 Experimental Results

In our experiment, we measure the number of solved problems and the average solving time for the holdout evaluation set. This included 383 and 488 problems in the linear and non-linear LIA datasets, respectively. The timeout for evaluating each problem is 1200 seconds. The additional overhead for reading and applying the GNN predicted results in each iteration is included in the solving

Table 3: Overview of the best ranking function and improvement in number of solved problems compared to the Eldarica. A ranking function marked with * (e.g., S-Plus*) denotes that there are multiple ranking functions with the same performance.

Benchmark Algorithm	MUS data set (best count)	**Best ranking function (improvement in %)**						
		Number of Solved Problems			Average Time			
		Total	SAT	UNSAT	All	Common	SAT	UNSAT
Linear CEGAR	Union (0)	R-Plus (1.4%)	R-Plus (2.4%)	R-Minus (1.0%)	R-Plus (1.3%)	S-Plus (19.1%)	S-Minus (46.5%)	Rank (31.1%)
	Single (3)	Rank (3.6%)	**R-Plus (4.0%)**	Rank (8.2%)	R-Plus (1.9%)	S-Plus (26.6%)	**R-Minus (57.9%)**	**Rank (36.3%)**
	Intersection (4)	**R-Plus (4.1%)**	S-Plus (0.8%)	**R-Plus (9.3%)**	**R-Plus (3.1%)**	**S-Plus (27.6%)**	R-Minus (45.0%)	S-Plus (0.0%)
Linear SymEx	Union (4)	**Two-Q (1.0%)**	**S-Plus* (0.0%)**	**Random (2.0%)**	Two-Q (0.9%)	R-Minus (12.7%)	**R-Minus (30.2%)**	S-Plus (26.5%)
	Single (3)	S-Minus* (0.5%)	**S-Plus* (0.0%)**	**Random (2.0%)**	Random (0.8%)	**S-Plus (12.9%)**	Random (28.4%)	S-Plus (17.6%)
	Intersection (5)	**S-Plus* (1.0%)**	**S-Plus* (0.0%)**	**S-Plus* (2.0%)**	**S-Plus (1.3%)**	Score (9.5%)	Random (28.4%)	**R-Plus (35.8%)**
Non-Linear CEGAR	Union (7)	**S-Plus (0.5%)**	**S-Plus (0.8%)**	**S-Plus* (0.0%)**	**S-Plus (7.1%)**	**R-Minus (20.8%)**	**Rank (53.5%)**	**S-Plus (19.4%)**
	Single (1)	R-Plus (0.2%)	R-Plus (0.4%)	**R-Plus* (0.0%)**	R-Plus (6.6%)	S-Plus (18.4%)	R-Minus (52.8%)	R-Minus (14.2%)
	Intersection (1)	R-Plus* (0.0%)	S-Plus (0.5%)	**S-Plus* (0.0%)**	R-Plus (5.9%)	R-Plus (20.3%)	Rank (45.8%)	S-Plus (16.8%)
Non-Linear SymEx	Union (6)	**Two-Q (6.1%)**	**S-Minus* (1.6%)**	Random (12.3%)	**Two-Q (13.3%)**	**R-Minus (7.3%)**	**Score (5.1%)**	**R-Plus (19.9%)**
	Single (3)	**Two-Q (6.1%)**	**Score (1.6%)**	**Two-Q (12.9%)**	Two-Q (12.4%)	Rank (-2.2%)	R-Minus (0.2%)	Two-Q (11.2%)
	Intersection (3)	**Two-Q (6.1%)**	**S-Plus (1.6%)**	**Two-Q (12.9%)**	Two-Q (12.7%)	S-Minus (0.6%)	Two-Q (1.7%)	S-Plus (5.4%)

time. The numerical results are shown in Tables 3 and 4. We also visualize some numerical results by scatter plots in Figure 2.

In Tables 3 and 4, under the Number of Solved Problems column, the Total, and SAT, UNSAT columns denote the number of totals solved, solved SAT, and solved UNSAT problems, respectively. Under the Average Time column, the All column denotes the average solving time for all problems, including those that timed out; the Common column means the average solving time for problems that were commonly solved using one of the ranking functions in Table 1, and the default Eldarica ranking function; the SAT and UNSAT columns are the average solving times for SAT and UNSAT problems, respectively. In certain cells, the percentage in brackets represents the improvement compared to the corresponding default ranking function. The bold text highlights the best performance in a Benchmark Algorithm block for each measurement.

Table 4: Evaluation on holdout problems using union dataset. The time unit is second.

Benchmark Algorithm	Ranking Function	Number of Solved Problems (improvement %)			Average Time (improvement %)			
		Total	SAT	UNSAT	All	Common	SAT	UNSAT
Linear CEGAR	Default	222	125	97	519.38	25.77	38.97	8.77
	Random	221 (-0.5%)	124 (-0.8%)	97 (0.0%)	523.58 (-0.8%)	27.49 (-29.5%)	37.05 (4.9%)	15.85 (-80.7%)
	R-Plus	**225 (1.4%)**	**128 (2.4%)**	97 (0.0%)	**512.41 (1.3%)**	21.65 (16.0%)	42.89 (-10.1%)	11.99 (-36.7%)
	R-Minus	220 (-0.9%)	122 (-2.4%)	**98 (1.0%)**	526.08 (-1.3%)	18.02 (-24.4%)	30.93 (20.6%)	21.60 (-146.3%)
	S-Plus	222 (0.0%)	125 (0.0%)	97 (0.0%)	517.43 (0.4%)	**20.92 (19.1%)**	34.13 (12.4%)	**7.32 (16.5%)**
	S-Minus	219 (-1.4%)	122 (-2.4%)	97 (0.0%)	522.97 (-0.7%)	12.56 (2.4%)	**20.86 (46.5%)**	9.81 (-11.9%)
	Portfolio	229 (3.2%)	130 (4.0%)	99 (2.1%)	503.16 (3.1%)	18.28 (29.1%)	45.67 (-17.2%)	19.94 (-127.4%)
Linear SymEx	Default	200	101	99	590.68	33.16	55.42	10.44
	Random	201 (0.5%)	100 (-1.0%)	101 (2.0%)	586.12 (0.8%)	30.08 (-8.5%)	39.69 (28.4%)	20.95 (-100.7%)
	R-Plus	192 (-4.0%)	**101 (0.0%)**	91 (-8.1%)	617.60 (-4.6%)	38.59 (-10.9%)	52.87 (4.6%)	21.99 (-110.6%)
	R-Minus	200 (0.0%)	100 (-1.0%)	100 (1.0%)	586.24 (0.8%)	**24.67 (12.7%)**	**38.69 (30.2%)**	10.60 (-1.5%)
	S-Plus	198 (-1.0%)	**101 (0.0%)**	97 (-2.0%)	595.02 (-0.7%)	30.22 (11.6%)	50.97 (8.0%)	**7.67 (26.5%)**
	S-Minus	201 (0.5%)	**101 (0.0%)**	100 (1.0%)	586.35 (0.7%)	30.64 (7.8%)	50.57 (8.8%)	10.65 (-2.0%)
	Two-queue	**202 (1.0%)**	**101 (0.0%)**	**101 (2.0%)**	**585.58 (0.9%)**	35.11 (-5.9%)	49.94 (9.9%)	20.14 (-92.9%)
	Portfolio	206 (3%)	101 (0.0%)	105 (6.1%)	569.1 (3.7%)	25.79 (22.2%)	44.58 (19.6%)	10.16 (2.6%)
Non Linear CEGAR	Default	432	250	182	131.12	42.05	43.34	40.28
	Random	425 (-1.6%)	243 (-2.8%)	**182 (0.0%)**	143.42 (-9.4%)	34.27 (-11.1%)	34.84 (19.6%)	38.75 (3.8%)
	R-Plus	432 (0.0%)	250 (0.0%)	**182 (0.0%)**	122.29 (6.7%)	31.74 (17.8%)	28.59 (34.0%)	37.82 (6.1%)
	R-Minus	417 (-3.5%)	240 (-4.0%)	177 (-2.7%)	154.07 (-17.5%)	**26.20 (20.8%)**	**21.46 (50.5%)**	**32.51 (19.3%)**
	S-Plus	**434 (0.5%)**	**252 (0.8%)**	**182 (0.0%)**	**121.75 (7.1%)**	34.64 (13.1%)	35.97 (17.0%)	39.10 (2.9%)
	S-Minus	421 (-2.5%)	242 (-3.2%)	179 (-1.6%)	149.02 (-13.7%)	31.76 (-2.0%)	26.33 (39.2%)	38.95 (3.3%)
	Portfolio	435 (0.7%)	253 (1.2%)	182 (0.0%)	113.49 (13.4%)	28.24 (29.1%)	30.57 (29.5%)	31.75 (21.2%)
Non Linear SymEx	Default	342	187	155	343.82	28.39	29.05	27.59
	Random	362 (5.8%)	188 (0.5%)	**174 (12.3%)**	301.90 (12.2%)	32.67 (-15.1%)	**36.24 (-24.8%)**	41.83 (-51.6%)
	R-Plus	339 (-0.9%)	**190 (1.6%)**	149 (-3.9%)	357.18 (-3.9%)	27.88 (0.3%)	47.71 (-64.2%)	**22.10 (19.9%)**
	R-Minus	361 (5.6%)	189 (1.1%)	172 (11.0%)	299.86 (12.8%)	**26.35 (7.3%)**	37.68 (-29.7%)	27.98 (-1.4%)
	S-Plus	340 (-0.6%)	189 (1.1%)	151 (-2.6%)	352.84 (-2.6%)	29.04 (-0.3%)	41.41 (-42.5%)	24.54 (11.1%)
	S-Minus	362 (5.8%)	**190 (1.6%)**	172 (11.0%)	303.65 (11.7%)	28.62 (-0.4%)	44.11 (-51.8%)	37.95 (-37.5%)
	Two-queue	**363 (6.1%)**	189 (1.1%)	**174 (12.3%)**	**297.93 (13.3%)**	30.15 (-6.2%)	41.14 (-41.6%)	32.51 (-17.8%)
	Portfolio	366 (7.0%)	191 (2.1%)	175 (12.9%)	288.85 (16.0%)	22.29 (21.4%)	42.42 (-46.0%)	26.75 (3.0%)

Table 3 displays the best ranking function and its improvement over the default Eldarica ranking function in different measurements for various combinations of benchmarks (linear and non-linear LIA), algorithms (CEGAR and SymEx), and MUS datasets (union, single, and intersection of MUSes). In the MUS dataset column, the numbers in brackets represent the count of bold text cells in the row, indicating the number of best performances achieved by that type of MUS dataset. For instance, in the last row (i.e., the Intersection row of the Non-Linear SymEx block), the number in the bracket is counted from the bold text highlighted cells in columns Total, SAT, and UNSAT under the Number of Solved Problems. This suggests that using the intersection of the MUSes dataset achieves the best performance when the evaluation set is non-linear and the algorithm is SymEx. Across the entire table, there are 17, 10, and 13 bold text counts for the union, single, and intersection MUS datasets, respectively. This indicates that the union is the most effective MUS dataset for better performance across different benchmarks and algorithms. Consequently, we provide further numerical details for the union MUS dataset in Table 4. Evaluation results for both the single and intersection MUS datasets can be found in [20].

Table 4 illustrates the evaluation results using MUS-guided ranking functions (refer to Table 1), compared to the default and random ranking functions. The portfolio rows in each Benchmark Algorithm block represent the best potential of ranking functions, as they accumulate the best performance for each problem across all ranking functions. The model is trained using the union of MUSes.

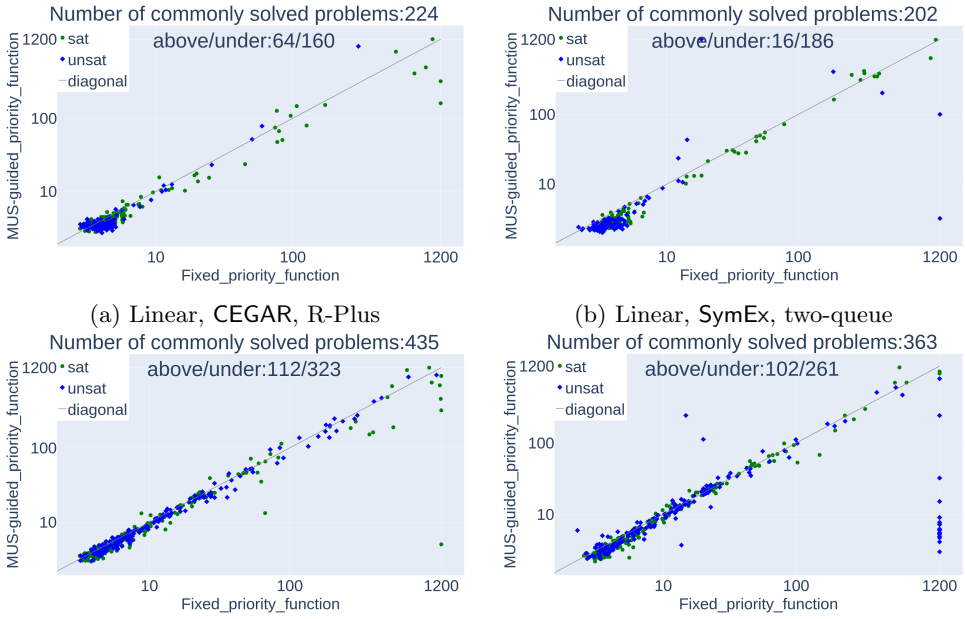
In terms of the total number of solved problems, the improvement for the Linear dataset is at most 1.4%, achieved by the CEGAR algorithm with the R-Plus ranking function. Meanwhile, for the Non-linear dataset, the improvement is 6.1%, achieved by the SymEx algorithm with the two-queue ranking function. And, this is consistent with the average solving time for all benchmarks.

The best performances in the Common column are inconsistent with the All column under Average Time. This suggests that the number of newly solved problems has a greater impact on the improvement in the average solving time for all problems than the commonly solved problems.

The improvements in the average solving time for SAT and UNSAT problems are 50.5% and 26.5%, achieved by the CEGAR with R-Minus and the SymEx with S-Plus ranking function, respectively. When combined with the corresponding numbers of total solved problems (i.e., -3.5% and -1.0%), it suggests that these ranking functions either solve the problems quickly or not at all.

Furthermore, Figure 2 shows the solving time scatter plots for the problems from the best configurations in each Benchmark Algorithm block in Table 4. Notably, a majority of the dots lie below the diagonal lines in each scatter plot, indicating the solving time is improved by the MUS-guided ranking function for more than half of the problems. This is consistent with the numerical results.

In summary, for both algorithms in different datasets, there is always at least one of the MUS-guided ranking functions that achieves the best result in terms of all aspects of measurements. Using the predicted probabilities alone (i.e., using the Rank and Score ranking function) performs weaker than other



(a) Linear, CEGAR, R-Plus

(b) Linear, SymEx, two-queue

(c) Non-Linear, CEGAR, S-Plus

(d) Non-Linear, SymEx, two-queue

Fig. 2: All benchmark average solving time scatter plots for best ranking functions in different dataset and algorithms. "above/under" means the number of dots above and under the diagonal line.

MUS-guided ranking functions that combine the predicted probability and the default heuristics. Currently, the MUS-guided ranking functions in Table 1 are designed by simply varying the relation symbols "+" and "-" between different elements (e.g., ranking functions S-Plus and S-Minus for both CEGAR and SymEx) or by setting the restart point randomly (i.e., ranking function Two-queue in SymEx). We believe MUSHyperNet has more potential if the ranking functions are designed carefully or learned from some good tasks.

7 Related Work

Machine learning techniques have been adapted in various ways to assist in formal verification. For example, the study in [35] employs Support Vector Machines (SVM) [36] and Gaussian processes[37] to select heuristics for theorem proving. Similarly, [38] introduces the use of a Recurrent Neural Network Based Language Model (RNNLM) to derive finite-state automaton-based specifications from execution traces. In the domain of selecting algorithms for program verification, [39] apply the Transformer architecture [40], while [41] uses kernel-based methods [42]. With the thriving of deep learning techniques, an increasing number of works are utilizing GNNs to learn the features from programs and logic formu-

lae. This trend is attributed to the inherent structure of these languages, which can be naturally represented as graphs and subsequently learned by GNNs. For instance, studies like [14,15], [16,43], and [44] use GNNs [24,45,46] to learn features from graph-represented logic formulas and programs, aiding in tasks such as theorem proving, SAT solving, and loop invariant reasoning.

One closed idea is NeuroSAT [17,16], which trains a GNN to predict the probability of variables appearing in unsat cores. This prediction can guide the variable branching decisions for Conflict-Driven Clause Learning (CDCL) [47]-based SAT solvers. In a similar vein, our study trains a GNN to predict the probability of a CHC appearing in MUSes. This aids in determining the processing sequence of CHCs in Algorithm 1 used for solving a set of CHCs.

8 Conclusion and Future Works

In this study, we train a GNN model to predict the probability of each CHC in a set of CHCs being in the MUSes. We then utilize these predicted probabilities to guide the abstract symbolic model-checking algorithm in selecting a CHC during each iteration. Extensive experiments demonstrate improvements in both the number of solved problems and average solving time when using the MUS-guided ranking functions, compared to the default ranking function. This was observed in two instances of the abstract symbolic model checking algorithm: CEGAR and SymEx. We believe that this approach can be extended to other algorithms, as many could benefit from understanding more about the MUSes of a set of CHCs.

There are several ways to further enhance the performance of MUSHyperNet. One of our future work is to integrate the work of manually designing the ranking functions in Table 1 to the learning process. Regarding the GNN model, we believe that incorporating an attention mechanism could bolster its performance, subsequently refining the quality of the predicted probabilities. Another avenue to explore involves integrating the GNN with the solver in a more interactive manner. Instead of predicting something at once and then using them in each iteration, we could query the GNN model to predict something in real-time based on the current context during each iteration.

Acknowledgement. We thank Zafer Esen for providing assistance in using the symbolic execution engine of Eldarica, and Marc Brockschmidt for various discussions on this work. The computations and data handling were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at UPPMAX and C3SE. The research was partially funded by the Swedish Research Council through grant agreement no. 2018-05973, by a Microsoft Research PhD grant, the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and the Wallenberg project UPDATE.

References

1. Chencheng Liang, Philipp Rümmer, and Marc Brockschmidt. Exploring representation of Horn clauses using GNNs (extended technique report). *CoRR*, abs/2206.06986, 2022.
2. Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.
3. Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi, and Maurizio Proietti. Analysis and transformation of constrained Horn clauses for program verification. *CoRR*, abs/2108.00739, 2021.
4. Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. *Horn Clause Solvers for Program Verification*, pages 24–51. Springer International Publishing, Cham, 2015.
5. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
6. Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
7. Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, Jan 2008.
8. Arie Gurfinkel. Program verification with constrained Horn clauses (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 19–29, Cham, 2022. Springer International Publishing.
9. Karel Chvalovský, Jan Jakubův, Martin Suda, and Josef Urban. ENIGMA-NG: Efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Automated Deduction – CADE 27*, pages 197–215, Cham, 2019. Springer International Publishing.
10. Jan Jakubův and Josef Urban. ENIGMA: Efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics*, pages 292–302, Cham, 2017. Springer International Publishing.
11. Jan Jakubův, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guidingmachine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 448–463, Cham, 2020. Springer International Publishing.
12. Martin Suda. Vampire with a brain is a good ITP hammer. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems*, pages 192–209, Cham, 2021. Springer International Publishing.
13. Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

14. Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2786–2796. Curran Associates, Inc., 2017.
15. Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *CoRR*, abs/1905.10006, 2019.
16. Daniel Selsam and Nikolaj Bjørner. Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. *CoRR*, abs/1903.04671, 2019.
17. Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
18. CHC-COMP benchmarks. Accessed: 2023-09-07, `https://chc-comp.github.io/`.
19. Grigory Fedyukovich and Philipp Rümmer. Competition report: CHC-COMP-21. In Hossein Hojjat and Bishoksan Kafle, editors, *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021*, volume 344 of *EPTCS*, pages 91–108, 2021.
20. Repository for the training and evaluation dataset in this work. Accessed: 2023-09-07, `https://github.com/ChenchengLiang/Horn-graph-dataset`.
21. John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
22. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.
23. Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.
24. Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.
25. Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne vanden Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, pages 593–607, Cham, 2018. Springer International Publishing.
26. Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU). *arXiv e-prints*, page arXiv:1803.08375, March 2018.
27. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, july 1976.
28. Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7, 2018.
29. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification*, pages 72–83, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
30. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April*

2-6, 2001, Proceedings, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.
31. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-Clause verification. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2013.
32. Hossein Hojjat and Philipp Rümmer. OptiRica: Towards an efficient optimizing Horn solver. In Geoffrey William Hamilton, Temesghen Kahsai, and Maurizio Proietti, editors, *Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation, HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3rd April 2022*, volume 373 of *EPTCS*, pages 35–43, 2022.
33. David R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):215–242, 1958.
34. Code repository for reproduce this work. Accessed: 2023-09-07, `https://github.com/ChenchengLiang/Relational-Hypergraph-Neural-Network-PyG`.
35. James Bridge, Sean Holden, and Lawrence Paulson. Machine learning for first-order theorem proving. *Journal of Automated Reasoning*, 53, 08 2014.
36. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
37. Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.
38. Tien-Duy B. Le and David Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 106–117, New York, NY, USA, 2018. Association for Computing Machinery.
39. Cedric Richter and Heike Wehrheim. Attend and represent: A novel view on algorithm selection for software verification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1016–1028, 2020.
40. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
41. Cedric Richter, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Algorithm selection for software validation based on graph kernels. *Automated Software Engineering*, 27(1):153–186, June 2020.
42. Bernhard Scholkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.
43. Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.
44. Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

45. Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.
46. Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.
47. Joao P. Marques-Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.