# Systematic Predicate Abstraction using Graph Neural Networks (Technical Report)

Chencheng Liang[1], Philipp Rümmer[1,2], Marc Brockschmidt[3], and Parosh Aziz Abdulla[1]

[1] Uppsala University, Uppsala, Sweden
chencheng.liang@it.uu.se
[2] University of Regensburg, Regensburg, Germany
philipp.ruemmer@it.uu.se
[3] Microsoft Research
marc@marcbrockschmidt.de

## 1 Introduction

In recent decades, automated program verification has become standard industry practice. The theme of encoding verification problems to Constrained Horn clauses (CHCs) [1] and solving them by symbolic model checking have been actively pursued in research [2]. However, model checkers, that exhaustively check whether the encoded model meets given specifications, suffers from state space explosion problem. Counterexample-guided abstraction refinement (CEGAR) [3] is one of the promising techniques to deal with models with infinite states by iteratively refining and checking the model's abstraction. Refining model's abstraction in each CEGAR' iteration is essential to improve the solving time and requires domain-specific knowledge.

In the predicate abstraction-based model checking [4,5], the tools, such as Eldarica [6], refine the abstract model by the means of Craig interpolation [7]. They use a set of predicates to represent the abstract model, and refining it by adding new predicates to the set. The new predicates in each CEGAR iteration are obtained by searching suitable interpolations in the infinite interpolation lattice derived from the counter-examples. [8] systematically explores the interpolant lattice by searching good maximal feasible *interpolation abstraction* in abstraction lattices constructed by powerset lattice over *templates*. In the context of program verification, templates are variables and terms in the programs. Nevertheless, the templates are chosen manually [9] depending on domain-specific knowledge (program features) extracted from static analysis (e.g., control- and data-flow analyses). Extracting program features usually need careful engineering and considerable domain expertise.

Therefore, we want to combine a powerful data-driving feature extracting technique, deep learning, to learn the program features automatically [10]. In our previous work (CHC-R-HyGNN) [11], we built a Graph Neural Network (GNN) [12] -based framework to describe and extract program features in the form of CHCs. Learning program features from the form of CHC can eliminate the syntactic affects from source code in different programming languages. From

the best of our knowledge, we didn't see any research that apply GNNs to guide the template selection which is the essential to systematically explore predicate abstraction in infinite interpolant lattice for solving CHC-encoded program verification problems.

In this study, we apply the CHC-R-HyGNN framework, which learns the program features from CHCs, to guide the template selection. To train a relevance filter for the templates, we label the training data by a predicate miner.

Furthermore, we explore how to combine the GNN-selected templates with the templates generated from static analysis in [8]' experiments to further improve the solving time for program verification problems. We evaluate our methods in the model checker Eldarica [6]. Notice that the abstraction interpolation is a solver-independent technique. Therefore, the trained template relevance filter can be applied to other model checkers as well.

Our dataset is extracted from a collection of problems from CHC-COMP[4]. The problems come from various sources (e.g., benchmarks generated with JayHorn[5] and differernt synthesis tools[6]) and in total, there are 8705 linear and 8425 non-linear Linear Integer Arithmetic (LIA) problems. More details about the benchmarks are in the Table 1 of the competition report [13].

*Contributions and the organisation of the study:*

(i) We build a predicate miner to label the templates and apply CHC-R-HyGNN to train a GNN model as the template relevance filter. It works as a data driven-based heuristic to guide the searching in infinite interpolant lattice (Section 3).
(ii) We build a end-to-end pipeline that can combine the templates generated from various static analysis strategies with GNN-selected templates. This can leverage both methods and obtain the best performance in program verification problems (Section 4).
(iii) We evaluate our methods on CHC-COMP benchmark and show that they assist the predicate abstraction-based model checking to reduce solving times and solve more problems (Section 5).

## 2 Background

In this section, we first introduce CHCs and explain how program verification problems are encoded to CHCs. Secondly, we introduce one of the main streams to solve CHCs by using CEGAR and predicate abstraction. Next, we explain how the abstraction interpolation guides the predicate abstraction for program verification problems [8]. Finally we introduce two mian components (i.e., representing CHCs by graphs and R-HyGNN) in CHC-R-HyGNN framework.

[4] `https://chc-comp.github.io/`
[5] `https://github.com/chc-comp/jayhorn-benchmarks`
[6] `https://github.com/chc-comp/synthesis`

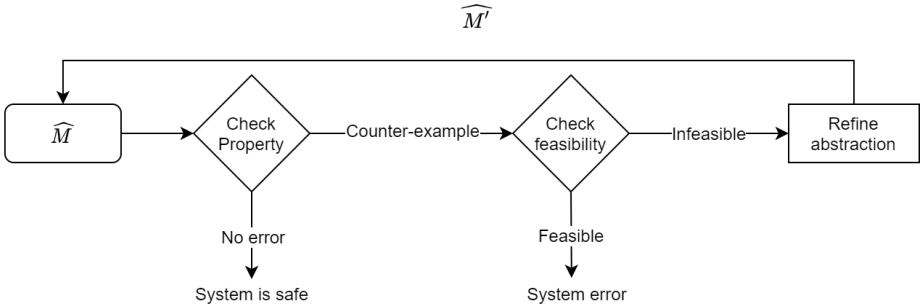


**Fig. 1.** CEGAR framework.

### 2.1 Encode Program Verification to Horn Clauses

A CHC can be written in the form $H \leftarrow B_1 \wedge \cdots \wedge B_n \wedge \varphi$, where $B_i$ is an application $p_i(t_1^i \cdots, t_k^i)$ of an uninterpreted fixed-arity relation symbol $p$ to a list of first-order terms; $H$ is either an application $p(t_1 \cdots, t_k)$, or *false*; $\varphi$ is a constraint over some background theory. Here, $H$ and $B_1 \wedge \cdots \wedge B_n \wedge \varphi$ in the left and right hand side of implication $\leftarrow$ are called "head" and "body", respectively. The first-order variables in a CHC are implicitly universally quantified.

CHCs can naturally encode the program verification problem since a CHC can express (i) an assumption by $p_0(\bar{t}) \leftarrow \mathit{Assume}(\bar{t})$, where $\mathit{Assume}(\bar{t})$ is a first-order formula encoding the assumption condition at control location 0 over a vector of variables $\bar{t} = (t_1^0, \cdots, t_k^0)$; (ii) A Floyd-style inductiveness condition by $p_i(\bar{t}) \leftarrow p_{i'}(\bar{t'}) \wedge T(\bar{t}, \bar{t'})$, where $T(\bar{t}, \bar{t'})$ is a first order formula encoding the transition condition between the control location $i$ and $i'$; or (iii) an assertion by *false* $\leftarrow p_i(\bar{t}) \wedge \neg \mathit{Assertion}(\bar{t})$, where $\mathit{Assertion}(\bar{t})$ is a first-order formula encoding an assertion condition at the control location $i$. For instance, the loop body of $\mathit{while}(x \neq 0)\{x = x+1;\}$ can be encoded to $L(x) \leftarrow L'(x') \wedge x' \neq 0 \wedge x = x' - 1$. The program is safe if and only if the encoded CHCs are satisfiable.

### 2.2 Solving Horn Clauses with Predicate Abstraction and CEGAR

CEGAR [3] is one of the state-of-art frameworks for solving model checking problems by iteratively refining the abstract model. The process is shown in Figure 1, where it begins from a model's abstraction $\widehat{M}$. If the abstraction is satisfy the property, then the concrete model satisfy the property as well (i.e., the system is safe), otherwise, a counter-example is generated and the feasibility (spuriousness) is checked. If the counter-example is feasible, the model is unsatisfiable to the property (i.e., system error). If the counter-example is infeasible, the abstraction is over-approximated and needs to be refined. The property is checked again with the refined new abstracted model $\widehat{M'}$.

Similar to the CEGAR process, predicate abstraction based model checkers, such as Eldarica [6] construct solutions for CHCs by building an *abstract*

*reachability graph* (ARG) over a set of given predicates. The solver attempts to construct a closed ARG (check property) from a initial predicate set (abstract model). If a closed ARG is constructed successfully (no error), the CHCs are solvable (system is safe). If it fails to construct a closed ARG, an counter-example (i.e., a path from entry clauses to a violated assertion clauses) is extracted. The feasibility is checked by a theorem prover. If the counter-example is feasible, then the CHCs are unsatisfiable (system error). If the counter-example is infeasible, A new abstract model is generated by adding new predicates which can eliminate the infeasible counter-example (refine the abstract model). The new predicates are generated by means of Craig interpolation.

### 2.3 Abstract Interpolation with Templates

An interpolation problem is to finding an interpolant $I$ such that for a conjunction $A \wedge B$, $I \rightarrow A$ and $B \rightarrow \neg I$, where $I$ contains variables that occur in both $A$ and $B$. Craig interpolation can be used to systematically refine the abstract model. The intuition is that between the most abstract model and the concrete model, there are infinite abstract models (interpolants) Searching suitable abstract model from infinite abstract models can be encoded to searching suitable interpolants from infinite interpolant lattice.

Abstract interpolation [8] is a semantic and solver-independent framework, which modifies the interpolation query for the theorem provers to compute possibly user-specified or quantified interpolants that might be hard to find for the prover alone. Instead of calling theorem provers to search suitable interpolants in infinite interpolant lattice, [8] first build an abstract interpolant lattice (a powerset of lattice over templates). Then, apply an algorithm to search maximal feasible interpolation abstractions in the abstract interpolant lattice. The maximal feasible interpolation abstractions are used to form a better interpolation query for the theorem provers to obtain better interpolants.

Constructing the abstraction lattice is essential for systematically explore the infinite interpolant lattice. However, in [8,9], the templates for generating abstraction lattices are constructed through manually static analyses for the programs. In this study, we select the templates by a GNN-based model.

### 2.4 CHC-R-HyGNN framework [11]

**Graph Encoding for Constrained Horn Clauses.** Graph is a suitable format to represent CHC-encoded problems since they contain rich structural information. We represent the CHCs by two graphs with typed nodes and edges (i.e., *control- and data-flow hypergraph* (CDHG) and *constraint graph* (CG)) to emphasis syntactic and semantic information [14]. In the Figure 2, we list concrete examples which represent the CHC $L(x) \leftarrow L(x') \wedge x' \neq 0 \wedge x = x' - 1$ by CDHG and CG. Intuitively, The CG parse CHCs in three different aspects, relation symbol, clause structure, and constraint, to describe the relations between relation symbols and their arguments, abstract syntax of head and body, and

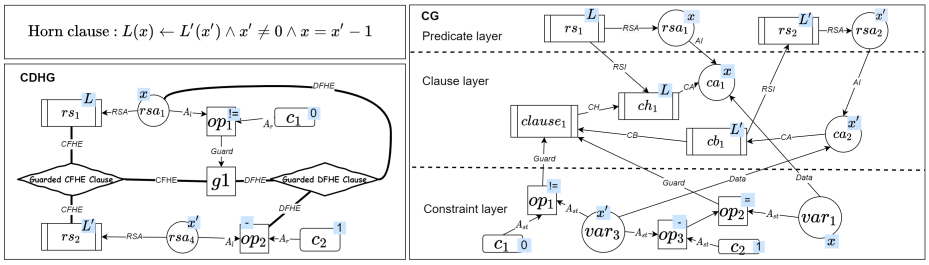


**Fig. 2.** CDHG and CG

constraint. The CDHG draw all symbols in CHCs as typed nodes and use control and data flow hyperedge to describe the control and data flow in verification problems.

Formally, A graph with typed node and edges, $G = (V, E, R, X, \ell)$, consists of a set of nodes $v \in V$, a set of typed edges $E \in V^* \times R$ where $V^*$ is a list of node from $V$, a set of node types $x \in X$, a set of edge types $r \in R$, and a map $\ell : v \to x$. The detailed constructing process from CHCs to CG and CDHG is in [14].

**Relational Hypergraph Neural Network (R-HyGNN).** Since CDHG is hypergraph (a graph with edges which connect arbitrary number of nodes), we choose a more generalized MP-GNN architecture R-HyGNN to learn the features. The node representation updating rule for R-HyGNN at time step $t$ is

$$h_v^t = \text{ReLU}(\sum_{r \in R} \sum_{p \in P_r} \sum_{e \in E_v^{r,p}} W_{r,p}^t \cdot \|[h_u^{t-1} \mid u \in e]), \tag{1}$$

where $\|\{\cdot\}$ means concatenation of all elements in a set, $r \in R = \{r_i \mid i \in \mathbb{N}\}$ is the set of edge types (relations), $p \in P_r = \{p_j \mid j \in \mathbb{N}\}$ is the set of node positions under edge type $r$, $W_{r,p}^t$ denotes learnable parameters when the node is in the $p$th position with edge type $r$, and $e \in E_v^{r,p}$ is the set of hyperedges of type $r$ in the graph in which node $v$ appears in position $p$, where $e$ is a list of nodes. The initial node representation $h_v^0$ is derived from the node types. Note that R-HyGNN can handle multiple edge types with the same number of connoted nodes. For instance, It can handle CDHG which consists of five binary and two ternary edges.

## 3 Template Selection

### 3.1 Training model

The input of the model is a graph representation (CDHG or CG) of the CHCs attached with unlabeled templates. The unlabeled template set consists of (i)

single boolean variables, (ii) single positive and negative integer variables, (iii) combination of all pair of integer variables in the form of $v_1 + v_2 = 0$ and $v_1 - v_2 = 0$, and (iv) combination of all pair of integer variables in the form of $v_1 + v_2 >= 0$, $v_1 - v_2 >= 0$, $-(v_1 + v_2) >= 0$, and $-(v_1 - v_2) >= 0$. The unlabeled templates are attached to the graph representations by connecting the roots of template syntax trees to the corresponding relation symbol nodes.

The output of the model is a set of selected templates from the unlabeled templates. We train two models to predict the usefulness of boolean and integer templates independently. For the boolean and integer templates, the model perform binary and multi- classification, respectively. Even if the usefulness of the templates are predicted separately by two models, we merge them when we construct the abstract interpolant lattice.

The neural network consists of three components: (i) a embedding layer which map the integer-encoded nodes to the real-value feature vectors according to the node types, (ii) the R-HyGNN, and (iii) a set of fully connected neural networks which receive gathered *template* node representations from R-HyGNNs.

### 3.2 Training data

We label the the usefulness of the templates by a template miner.

...

## 4 Apply Selected Templates

[8] proposed four strategies to construct the templates by performing static analysis in the programs. We call them T2, 3, 4, and 5. And, every template has a cost value (an integer) to guide the search algorithm to find "good" interpolants in the abstract interpolant lattice constructed by the templates.

In this study, we combine the GNN-selected templates (T1) with (T2-5) by two ways. The cost of the T1 is decided by three ways.

The two ways to combine templates in each CEGAR iteration are:

1. union: using either T1 or one of T2-5.
2. random: merging the new predicate set generated from T1 and one of T2-5.

The three ways to assign cost values to the GNN-predicted templates are:

1. C1: all templates use the same cost value.
2. C2: the cost values are inversely proportional to the sigmoid output from GNNs.
3. C3: the cost values are inversely proportional to the number of variables in the tempaltes.

## 5 Evaluation

We first describe some important training parameters, then describe the benchmark and the dataset for training, and finally we analyse the the experimental results.

**Table 1.** The data distribution of a collection of CHC-COMP benchmark [13]. Only the sat problems are used to generate labeled data. Some problems are sat but are not labeled because there is no mined templates, or the graph representation is too big for the GNN.

| | | Solvable | | | Unsolvable |
|---|---|---|---|---|---|
| | Total | Unsat | Sat | Labeled | |
| Linear LIA | 8705 | 2160 | 2944 | 441 | 1514 |
| Non-linear LIA | 8425 | 2826 | 3668 | 1323 | 234 |

### 5.1 Model parameters

The implementation is based on the framework tf2_gnn [7]. We set the all middle layer sizes in the framework to 64, the number of message passing steps to 8 (i.e., apply Eq. 1 8 times), the maximum training epoch to 500, and the patient to 100. For the rest of parameters, we use the default settings in the framework tf2_gnn.

### 5.2 benchmarks and dataset

The dataset we used for training and evaluation is specified in Table 1. The models are trained by the labeled dataset, and it is divided to train, valid, and test set by 60%, 20%, and 20%. The dataset is available in a Github repository[8]

### 5.3 Experimental results

The measurements are number of solved problems and the solving time for solvable problems.

## 6 Related work

### 6.1 Machine Learning for Program Verification

Machine learning techniques has been adapted in different ways to aid formal verification. For instance, [15] propose a Recurrent Neural Network Based Language Model (RNNLM) to mine the finite-state automaton-based specification from the execution trace. [16,17] apply Transformer architecture [18] and kernel-based methods [19] respectively to select algorithms for program verification. [20] uses Support Vector Machines (SVM) [21] and Gaussian process [22] to select the heuristics for theorem proving. Along With the thriving of deep learning techniques, more and more works tend to use GNNs to learn the features of programs and logic formulae since they are highly structured language and can be naturally represented by graph and learned by GNNs. For instance, [23,24], [25,26],

[7] `https://github.com/microsoft/tf2-gnn`
[8] `https://github.com/ChenchengLiang/Learning-abstract-predicate-dataset`

**Table 2.** 100+ benchmark solvability

| Options | Solvable_number | Unique_solved_number | unique_solvable_list |
|---|---|---|---|
| Term_splitClauses_1_cost_same | 51 | 0 | [] |
| Octagon_splitClauses_1_cost_same | 53 | 0 | [] |
| RelationalEqs_splitClauses_1_cost_same | 55 | 0 | [] |
| RelationalIneqs_splitClauses_1_cost_same | 55 | 0 | [] |
| Empty_splitClauses_1_cost_same | 46 | 0 | [] |
| Random_splitClauses_1_cost_same | 50 | 0 | [] |
| Unlabeled_splitClauses_1_cost_same | 53 | 1 | [chc-LIA-Lin_1387.smt2 |
| PredictedCG_splitClauses_1_cost_shape | 48 | 0 | [] |
| PredictedCG_splitClauses_1_cost_logit | 48 | 0 | [] |
| PredictedCG_splitClauses_1_cost_same | 48 | 0 | [] |
| PredictedCDHG_splitClauses_1_cost_shape | 52 | 0 | [] |
| PredictedCDHG_splitClauses_1_cost_logit | 52 | 0 | [] |
| PredictedCDHG_splitClauses_1_cost_same | 52 | 0 | [] |
| Term_monoDirectionLayerGraph_union_0.0_splitCl... | 52 | 0 | [] |
| Term_monoDirectionLayerGraph_union_0.0_splitCl... | 52 | 0 | [] |
| Term_monoDirectionLayerGraph_union_0.0_splitCl... | 53 | 1 | [chc-LIA-Lin_3850.smt2 |
| Term_hyperEdgeGraph_union_0.0_splitClauses_1_c... | 49 | 0 | [] |
| Term_hyperEdgeGraph_union_0.0_splitClauses_1_c... | 49 | 0 | [] |
| Term_hyperEdgeGraph_union_0.0_splitClauses_1_c... | 49 | 0 | [] |
| Octagon_monoDirectionLayerGraph_union_0.0_spli... | 53 | 0 | [] |
| Octagon_monoDirectionLayerGraph_union_0.0_spli... | 53 | 0 | [] |
| Octagon_monoDirectionLayerGraph_union_0.0_spli... | 53 | 0 | [] |
| Octagon_hyperEdgeGraph_union_0.0_splitClauses_... | 49 | 0 | [] |
| Octagon_hyperEdgeGraph_union_0.0_splitClauses_... | 49 | 0 | [] |
| Octagon_hyperEdgeGraph_union_0.0_splitClauses_... | 49 | 0 | [] |
| RelationalEqs_monoDirectionLayerGraph_union_0.... | 55 | 0 | [] |
| RelationalEqs_monoDirectionLayerGraph_union_0.... | 55 | 0 | [] |
| RelationalEqs_monoDirectionLayerGraph_union_0.... | 55 | 0 | [] |
| RelationalEqs_hyperEdgeGraph_union_0.0_splitCl... | 49 | 0 | [] |
| RelationalEqs_hyperEdgeGraph_union_0.0_splitCl... | 49 | 0 | [] |
| RelationalEqs_hyperEdgeGraph_union_0.0_splitCl... | 49 | 0 | [] |
| RelationalIneqs_monoDirectionLayerGraph_union_... | 55 | 0 | [] |
| RelationalIneqs_monoDirectionLayerGraph_union_... | 55 | 0 | [] |
| RelationalIneqs_monoDirectionLayerGraph_union_... | 56 | 1 | [chc-LIA-Lin_8361.smt2 |
| RelationalIneqs_hyperEdgeGraph_union_0.0_split... | 49 | 0 | [] |
| RelationalIneqs_hyperEdgeGraph_union_0.0_split... | 49 | 0 | [] |
| RelationalIneqs_hyperEdgeGraph_union_0.0_split... | 49 | 0 | [] |
| Term_monoDirectionLayerGraph_random_0.5_splitC... | 48 | 0 | [] |
| Term_monoDirectionLayerGraph_random_0.5_splitC... | 48 | 0 | [] |
| Term_monoDirectionLayerGraph_random_0.5_splitC... | 48 | 0 | [] |
| Term_hyperEdgeGraph_random_0.5_splitClauses_1_... | 46 | 0 | [] |
| Term_hyperEdgeGraph_random_0.5_splitClauses_1_... | 46 | 0 | [] |
| Term_hyperEdgeGraph_random_0.5_splitClauses_1_... | 46 | 0 | [] |
| Octagon_monoDirectionLayerGraph_random_0.5_spl... | 48 | 0 | [] |
| Octagon_monoDirectionLayerGraph_random_0.5_spl... | 48 | 0 | [] |
| Octagon_monoDirectionLayerGraph_random_0.5_spl... | 48 | 0 | [] |
| Octagon_hyperEdgeGraph_random_0.5_splitClauses... | 46 | 0 | [] |
| Octagon_hyperEdgeGraph_random_0.5_splitClauses... | 46 | 0 | [] |
| Octagon_hyperEdgeGraph_random_0.5_splitClauses... | 46 | 0 | [] |
| RelationalEqs_monoDirectionLayerGraph_random_0... | 48 | 0 | [] |
| RelationalEqs_monoDirectionLayerGraph_random_0... | 48 | 0 | [] |
| RelationalEqs_monoDirectionLayerGraph_random_0... | 48 | 0 | [] |
| RelationalEqs_hyperEdgeGraph_random_0.5_splitC... | 46 | 0 | [] |
| RelationalEqs_hyperEdgeGraph_random_0.5_splitC... | 46 | 0 | [] |
| RelationalEqs_hyperEdgeGraph_random_0.5_splitC... | 46 | 0 | [] |
| RelationalIneqs_monoDirectionLayerGraph_random... | 48 | 0 | [] |
| RelationalIneqs_monoDirectionLayerGraph_random... | 48 | 0 | [] |
| RelationalIneqs_monoDirectionLayerGraph_random... | 48 | 0 | [] |
| RelationalIneqs_hyperEdgeGraph_random_0.5_spli... | 46 | 0 | [] |

| | | Linear | | Non-linear | |
|---|---|---|---|---|---|
| | | Solved problems | Solving times | Solved problems | Solving times |
| T0 | | | | | |
| T2 | | | | | |
| T3 | | | | | |
| T4 | | | | | |
| T5 | | | | | |
| T1 | C1 | | | | |
| | C2 | | | | |
| | C3 | | | | |
| Union, C3 | T1+T2 | | | | |
| | T1+T3 | | | | |
| | T1+T4 | | | | |
| | T1+T5 | | | | |
| Random, C3 | T1+T2 | | | | |
| | T1+T3 | | | | |
| | T1+T4 | | | | |
| | T1+T5 | | | | |

**Fig. 3.** T0 means no template is used, T1 is GNN-selected templates, T2-5 are templates from [8], and C1-3 are ways to assign cost value to GNN-selected templates. Union means combine the generated new predicates when use the two template sets. Random means use either the two template sets to generate the templates.

and [27] learn features from graph represented logic formulae and programs by GNNs [28,29,30] to aid theorem proving, SAT solving, and loop invariant reasoning, respectively. Every study has their unique graph representation of program logic, GNNs, and working pipeline, therefore, is not extensible to other tasks. CHC-R-HyGNN [11] proposes a general framework, which designs syntactic- and semantic-based graph representations for CHCs to generalize the graph representation for program logic, a hyperedge-compatible GNN (R-HyGNN), and two working pipelines for regression and classification tasks, to aid decision and value predication problems in program verification. We apply a modified CHC-R-HyGNN framework to build a relevance filter for the templates to guide the predicate abstraction.

### 6.2 Explore Interpolation Lattice for Predicate Abstraction

## 7 Conclusion and future works

Relevance filter can be used for each CEGAR iteration

## References


1. Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.
2. Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. *Horn clause solvers for program verification*, pages 24–51. 09 2015.
3. 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.
4. 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.
5. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 232–244, New York, NY, USA, 2004. Association for Computing Machinery.
6. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification (extended technical report), 2013.
7. William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, 1957.
8. Jérôme Leroux, Philipp Rümmer, and Pavle Subotić. Guiding Craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, Jun 2016.
9. Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods*, pages 265–281, Cham, 2017. Springer International Publishing.
10. Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.
11. Chencheng Liang, Philipp Rümmer, and Marc Brockschmidt. Exploring representation of horn clauses using gnns, 2022.
12. 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.
13. Grigory Fedyukovich and Philipp Rümmer. Competition report: CHC-COMP-21, 2021.
14. Chencheng Liang, Philipp Rümmer, and Marc Brockschmidt. Exploring representation of horn clauses using gnns, 2022.

15. 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, page 106–117, New York, NY, USA, 2018. Association for Computing Machinery.
16. 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.
17. 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.
18. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.
19. Bernhard Schölkopf and A.J. Smola. *Smola, A.: Learning with Kernels - Support Vector Machines, Regularization, Optimization and Beyond. MIT Press, Cambridge, MA*, volume 98. 01 2001.
20. James Bridge, Sean Holden, and Lawrence Paulson. Machine learning for first-order theorem proving. *Journal of Automated Reasoning*, 53, 08 2014.
21. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
22. Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.
23. 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.
24. 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.
25. Daniel Selsam and Nikolaj Bjørner. Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. *CoRR*, abs/1903.04671, 2019.
26. 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.
27. Xujie Si, Hanjun Dai, Mukund Raghothaman, M. Naik, and Le Song. Learning loop invariants for program verification. In *NeurIPS*, 2018.
28. 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.
29. F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.
30. Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks, 2015.