Exploring Representation of Horn Clauses using GNNs

Chencheng Liang\textsuperscript{1}, Philipp Rümmer\textsuperscript{1,2} and Marc Brockschmidt\textsuperscript{3}

\textsuperscript{1}Uppsala University, Department of Information Technology, Uppsala, Sweden
\textsuperscript{2}University of Regensburg, Regensburg, Germany
\textsuperscript{3}Microsoft Research

Abstract
In recent years, the application of machine learning in program verification, and the embedding of programs to capture semantic information, has been recognised as an important tool by many research groups. Learning program semantics from raw source code is challenging due to the complexity of real-world programming language syntax and due to the difficulty of reconstructing long-distance relational information implicitly represented in programs using identifiers. Addressing the first point, we consider Constrained Horn Clauses (CHCs) as a standard representation of program verification problems, providing a simple and programming language-independent syntax. For the second challenge, we explore graph representations of CHCs, and propose a new Relational Hypergraph Neural Network (R-HyGNN) architecture to learn program features.

We introduce two different graph representations of CHCs. One is called constraint graph (CG), and emphasizes syntactic information of CHCs by translating the symbols and their relations in CHCs as typed nodes and binary edges, respectively, and constructing the constraints as abstract syntax trees. The second one is called control- and data-flow hypergraph (CDHG), and emphasizes semantic information of CHCs by representing the control and data flow through ternary hyperedges.

We then propose a new GNN architecture, R-HyGNN, extending Relational Graph Convolutional Networks, to handle hypergraphs. To evaluate the ability of R-HyGNN to extract semantic information from programs, we use R-HyGNNs to train models on the two graph representations, and on five proxy tasks with increasing difficulty, using benchmarks from CHC-COMP 2021 as training data. The most difficult proxy task requires the model to predict the occurrence of clauses in counter-examples, which subsumes satisfiability of CHCs. CDHG achieves 90.59\% accuracy in this task. Furthermore, R-HyGNN has perfect predictions on one of the graphs consisting of more than 290 clauses. Overall, our experiments indicate that R-HyGNN can capture intricate program features for guiding verification problems.

Keywords
Constraint Horn clauses, Graph Neural Networks, Automatic program verification

1. Introduction

Automatic program verification is challenging because of the complexity of industrially relevant programs. In practice, constructing domain-specific heuristics from program features (e.g., information from loops, control flow, or data flow) is essential for solving verification problems.
For instance, [1] and [2] extract semantic information by performing systematical static analysis to refine abstractions for the counterexample-guided abstraction refinement (CEGAR) [3] based system. However, manually designed heuristics usually aim at a specific domain and are hard to transfer to other problems. Along with the rapid development of deep learning in recent years, learning-based methods have evolved quickly and attracted more attention. For example, the program features are explicitly given in [4, 5] to decide which algorithm is potentially the best for verifying the programs. Later in [6, 7], program features are learned in the end-to-end pipeline. Moreover, some generative models [8, 9] are also introduced to produce essential information for solving verification problems. For instance, Code2inv [10] embeds the programs by graph neural networks (GNNs) [11] and learns to construct loop invariants by a deep neural reinforcement framework.

For deep learning-based methods, no matter how the learning pipeline is designed and the neural network structure is constructed, learning to represent semantic program features is essential and challenging (a) because the syntax of the source code varies depending on the programming languages, conventions, regulations, and even syntax sugar and (b) because it requires capturing intricate semantics from long-distance relational information based on re-occurring identifiers. For the first challenge, since the source code is not the only way to represent a program, learning from other formats is a promising direction. For example, inst2vec [12] learns control and data flow from LLVM intermediate representation [13] by recursive neural networks (RNNs) [14]. Constrained Horn Clauses (CHCs) [15], as an intermediate verification language, consist of logic implications and constraints and can alleviate the difficulty since they can naturally encode program logic with simple syntax. For the second challenge, we use graphs to represent CHCs and learn the program features by GNNs since they can learn from the structural information within the node’s N-hop neighbourhood by recursive neighbourhood aggregation (i.e., neural message passing) procedure.

In this work, we explore how to learn program features from CHCs by answering two questions: (1) What kind of graph representation is suitable for CHCs? (2) Which kind of GNN is suitable to learn from the graph representation?

For the first point, we introduce two graph representations for CHCs: the constraint graph (CG) and control- and data-flow hypergraph (CDHG). The constraint graph encodes the CHCs into three abstract layers (predicate, clause, and constraint layers) to preserve as much structural information as possible (i.e., it emphasizes program syntax). On the other hand, the Control- and data-flow hypergraph uses ternary hyperedges to capture the flow of control and data in CHCs to emphasize program semantics. To better express control and data flow in CDHG, we construct it from normalized CHCs. The normalization changes the format of the original CHC but retains logical meaning. We assume that different graph representations of CHCs capture different aspects of semantics. The two graph representations can be used as a baseline to construct new graph representations of CHC to represent different semantics. In addition, similar to the idea in [16], our graph representations are invariant to the concrete symbol names in the CHCs since we map them to typed nodes.

For the second point, we propose a Relational Hypergraph Neural Network (R-HyGNN), an extension of Relational Graph Convolutional Networks (R-GCN) [17]. Similar to the GNNs used in LambdaNet [18], R-HyGNN can handle hypergraphs by concatenating the node representations involved in a hyperedge and passing the messages to all nodes connected by the
Table 1
Proxy tasks used to evaluate suitability of different graph representations.

<table>
<thead>
<tr>
<th>Task</th>
<th>Task type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Argument identification</td>
<td>Node binary classification</td>
<td>For each element in CHCs, predict if it is an argument of relation symbols.</td>
</tr>
<tr>
<td>2. Count occurrence of relation symbols in all clauses</td>
<td>Regression task on node</td>
<td>For each relation symbol, predict how many times it occurs in all clauses.</td>
</tr>
<tr>
<td>3. Relation symbol occurrence in SCCs</td>
<td>Node binary classification</td>
<td>For each relation symbol, predict if a cycle exists from the node to itself (membership in strongly connected component, SCC).</td>
</tr>
<tr>
<td>4. Existence of argument bounds</td>
<td>Node binary classification</td>
<td>For each argument of a relation symbol, predict if it has a lower or upper bound.</td>
</tr>
<tr>
<td>5. Clause occurrence in counter-examples</td>
<td>Node binary classification</td>
<td>For each CHC, predict if it appears in counter-examples.</td>
</tr>
</tbody>
</table>

Finally, we evaluate our framework (two graph representations of CHCs and R-HyGNN) by five proxy tasks (see details in Table 1) with increasing difficulties. Task 1 requires the framework to learn to classify syntactic information of CHCs, which is explicitly encoded in the two graph representations. Task 2 requires the R-HyGNN to predict a syntactic counting task. Task 3 needs the R-HyGNN to approximate the Tarjan’s algorithm [19], which solves a general graph theoretical problem. Task 4 is much harder than the last three tasks since the existence of argument bounds is undecidable. Task 5 is harder than solving CHCs since it predicts the trace of counter-examples (CEs). Note that Task 1 to 3 can be easily solved by specific, dedicated standard algorithms. We include them to systematically study the representational power of graph neural networks applied to different graph construction methods. However, we speculate that using these tasks as pre-training objectives for neural networks that are later fine-tuned to specific (data-poor) tasks may be a successful strategy which we plan to study in future work.

Our benchmark data is extracted from the 8705 linear and 8425 non-linear Linear Integer Arithmetic (LIA) problems in the CHC-COMP repository^1^ (see Table 1 in the competition report [20]). The verification problems come from various sources (e.g., higher-order program verification benchmark^2^ and benchmarks generated with JayHorn^3^), therefore cover programs with different size and complexity. We collect and form the train, valid, and test data using the predicate abstraction-based model checker Eldarica [21]. We implement R-HyGNNs^4^ based on the framework tf2_gnn^5^\(^6^\). Our code is available in a Github repository. For both graph representations, even if the predicted accuracy decreases along with the increasing difficulty of tasks, for undecidable problems in Task 4, R-HyGNN still achieves high accuracy, i.e., 91% and 94% for constraint graph and CDHG, respectively. Moreover, in Task 5, despite the high

---

^1^https://chc-comp.github.io/
^2^https://github.com/chc-comp/hopv
^3^https://github.com/chc-comp/jayhorn-benchmarks
^4^https://github.com/ChenchengLiang/tf2-gnn
^5^https://github.com/microsoft/tf2-gnn
accuracy (96%) achieved by CDHG, R-HyGNN has a perfect prediction on one of the graphs consisting of more than 290 clauses, which is impossible to achieve by learning simple patterns (e.g., predict the clause including \texttt{false} as positive). Overall, our experiments show that our framework learns not only the explicit syntax but also intricate semantics.

\textbf{Contributions of the paper.} (i) We encode CHCs into two graph representations, emphasizing abstract program syntactic and semantic information, respectively. (ii) We extend a message passing-based GNN, R-GCN, to R-HyGNN to handle hypergraphs. (iii) We introduce five proxy supervised learning tasks to explore the capacity of R-HyGNN to learn semantic information from the two graph representations. (iv) We evaluate our framework on the CHC-COMP benchmark and show that this framework can learn intricate semantic information from CHCs and has the potential to produce good heuristics for program verification.

\section{Background}

\subsection{From Program Verification to Horn clauses}

Constrained Horn Clauses are logical implications involving unknown predicates. They can be used to encode many formalisms, such as transition systems, concurrent systems, and interactive systems. The connections between program logic and CHCs can be bridged by Floyd-Hoare logic \cite{22, 23}, allowing to encode program verification problems into the CHC satisfiability problems \cite{24}. In this setting, a program is guaranteed to satisfy a specification if the encoded CHCs are satisfiable, and vice versa.

We write CHCs in the form \( H \leftarrow B_1 \land \cdots \land B_n \land \varphi \), where (i) \( B_i \) is an application \( q_i(\bar{t}_i) \) of the relation symbol \( q_i \) to a list of first-order terms \( \bar{t}_i \); (ii) \( H \) is either an application \( q(\bar{t}) \), or \texttt{false}; (iii) \( \varphi \) is a first-order constraint. Here, \( H \) and \( B_1 \land \cdots \land B_n \land \varphi \) in the left and right hand side of implication \( \leftarrow \) are called “head” and “body”, respectively.

An example in Figure 1 explains how to encode a verification problem into CHCs. In Figure 1a, we have a verification problem, i.e., a C program with specifications. It has an external input \( n \), and we can initially assume that \( x == n, y == n, \) and, \( n >= 0 \). While \( x \) is not equal to \( 0, x \) and \( y \) are decreased by \( 1 \). The assertion is that finally, \( y == 0 \). This program can be encoded to the CHC shown in Figure 1b. The variables \( x \) and \( y \) are quantified universally. We can further simplify the CHCs in Figure 1b to the CHCs shown in Figure 1c without changing the satisfiability by some preprocessing steps (e.g., inlining and slicing) \cite{25}. For example, the first CHC encodes line 3, i.e., the assume statement, the second clause encodes lines 4-7, i.e., the while loop, and the third clause encodes line 8, i.e., the assert statement in Figure 1a. Solving the CHCs is equivalent to answering the verification problem. In this example, with a given \( n \), if the CHCs are satisfiable for all \( x \) and \( y \), then the program is guaranteed to satisfy the specifications.

\subsection{Graph Neural Networks}

Let \( G = (V, R, E, X, \ell) \) denote a graph in which \( v \in V \) is a set of nodes, \( r \in R \) is a set of edge types, \( E \in V \times V \times R \) is a set of typed edges, \( x \in X \) is a set of node types, and \( \ell : v \rightarrow x \) is a
labelling map from nodes to their type. A tuple $e = (u, v, r) \in E$ denotes an edge from node $u$ to $v$ with edge type $r$.

Message passing-based GNNs use a neighbourhood aggregation strategy, where at timestep $t$, each node updates its representation $h_v^t$ by aggregating representations of its neighbours and then combining its own representation. The initial node representation $h_v^0$ is usually derived from its type or label $\ell(u)$. The common assumption of this architecture is that after $T$ iterations, the node representation $h_v^T$ captures local information within $t$-hop neighbourhoods. Most GNN architectures [26, 27] can be characterized by their used “aggregation” function $\rho$ and “update” function $\phi$. The node representation of the $t$-th layer of such a GNN is then computed by $h_v^t = \phi(\rho(\{h_u^{t-1} \mid u \in N_v^t, r \in R\}), h_v^{t-1})$, where $R$ is a set of edge type and $N_v^t$ is the set of nodes that are the neighbors of $v$ in edge type $r$.

A closed GNN architecture to the R-HyGNN is R-GCN [17]. They update the node representation by

$$h_v^t = \sigma(\sum_{r \in R} \sum_{u \in N_v^t} \frac{1}{c_{v,r}} W_r h_u^{t-1} + W_0 h_v^{t-1}),$$

where $W_r$ and $W_0$ are edge-type-dependent trainable weights, $c_{v,r}$ is a learnable or fixed normalisation factor, and $\sigma$ is an activation function.
3. Graph Representations for CHCs

Graphs as a representation format support arbitrary relational structure and thus can naturally encode information with rich structures like CHCs. We define two graph representations for CHCs that emphasize the program syntax and semantics, respectively. We map all symbols in CHCs to typed nodes and use typed edges to represent their relations. In this section, we give concrete examples to illustrate how to construct the two graph representations from a single CHC modified from Figure 1c. In the examples, we first give the intuition of the graph design and then describe how to construct the graph step-wise. To better visualize how to construct the two graph representations in Figures 2 and 3, the concrete symbol names for the typed nodes are shown in the blue boxes. R-HyGNN is not using these names (which, as a result of repeated transformations, usually do not carry any meaning anyway) and only consumes the node types. The formal definitions of the graph representations and the algorithms to construct them from multiple CHCs are in the full version of this paper [28]. Note that the two graph representations in this study are designed empirically. They can be used as a baseline to create variations of the graphs to fit different purposes.

3.1. Constraint Graph (CG)

Our Constraint graph is a directed graph with binary edges designed to emphasize syntactic information in CHCs. One concrete example of constructing the constraint graph for a single CHC \( L(x, y, n) \leftarrow L(x', y', n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1 \) modified from Figure 1c is shown in Figure 2. The corresponding node and edge types are described in Tables 2 and 3 in the full version of this paper [28].

We construct the constraint graph by parsing the CHCs in three different aspects (relation symbol, clause structure, and constraint) and building relations for them. In other words, a constraint graph consists of three layers: the predicate layer depicts the relation between relation symbols and their arguments; the clause layer describes the abstract syntax of head and body items in the CHC; the constraint layer represents the constraint by abstract syntax trees (ASTs).

**Constructing a constraint graph.** Now we give a concrete example to describe how to construct a constraint graph for a single CHC \( L(x, y, n) \leftarrow L(x', y', n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1 \) step-wise. All steps correspond to the steps in Figure 2. In the first step, we draw relation symbols and their arguments as typed nodes and build the connection between them. In the second step, we construct the clause layer by drawing clauses, the relation symbols in the head and body, and their arguments as typed nodes and build the relation between them. In the third step, we construct the constraint layer by drawing ASTs for the sub-expressions of the constraint. In the fourth step, we add connections between three layers. The predicate and clause layer are connected by the relation symbol instance (\( \text{RSI} \)) and argument instance (\( \text{AI} \)) edges, which means the elements in the predicate layer are instances of the clause layer. The clause and constraint layers are connected by the \( \text{GUARD} \) and \( \text{DATA} \) edges since the constraint is the guard of the clause implication, and the constraint and clause layer share some elements.
Figure 2: Construct constraint graph from the CHC $L(x, y, n) \leftarrow L(x', y', n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1$. Note that some nodes have multiple concrete symbol names (e.g., node $rs_{a1}$ has two concrete names, $x$ and $x'$) since one relation symbol may bind with different arguments.

### 3.2. Control- and Data-flow Hypergraph (CDHG)

In contrast to the constraint graph, the CDHG representation emphasizes the semantic information (control and data flow) in the CHCs by hyperedges which can join any number of vertices. To represent control and data flow in CDHG, first, we preprocess every CHC and then split the constraint into control and data flow sub-formulas.

**Normalization.** We normalize every CHC by applying the following rewriting steps: (i) We ensure that every relation symbol occurs at most once in every clause. For instance, the CHC $q(a) \leftarrow q(b) \land q(c)$ has multiple occurrences of the relation symbol $q$, and we normalize it to equi-satisfiable CHCs $q(a) \leftarrow q'(b) \land q''(c)$, $q'(b) \leftarrow q(b') \land b = b'$ and $q''(c) \leftarrow q(c') \land c = c'$. (ii) We associate each relation symbol $q$ with a unique vector of pair-wise distinct argument variables $\bar{x}_q$, and rewrite every occurrence of $q$ to the form $q(\bar{x}_q)$. In addition, all the argument vectors $\bar{x}_q$ are kept disjoint. The normalized CHCs from Figure 1c are shown in Table 2.

**Splitting constraints into control- and data-flow formulas.** We can rewrite the constraint $\varphi$ to a conjunction $\varphi = \varphi_1 \land \cdots \land \varphi_k$, $k \in \mathbb{N}$. The sub-formula $\varphi_i$ is called a “data-flow sub-formula” if and only if it can be rewritten to the form $x = t(\bar{y})$ such that (i) $x$ is one of the arguments in head $q(\bar{x}_q)$; (ii) $t(\bar{y})$ is a term over variables $\bar{y}$, where each element of $\bar{y}$ is an argument of some body literal $q'(\bar{x}_q')$. We call all other $\varphi_j$ “control-flow sub-formulas”. A constraint $\varphi$ can then be represented by $\varphi = g_1 \land \cdots \land g_m \land d_1 \land \cdots \land d_n$, where $m, n \in \mathbb{N}$ and $g_i$ and $d_j$ are the control- and data-flow sub-formulas, respectively. The control and data flow sub-formulas for the normalized CHCs of our running example are shown in Table 2.
Table 2
Control- and data-flow sub-formula in constraints for the normalized CHCs from Figure 1c

<table>
<thead>
<tr>
<th>Normalized CHCs</th>
<th>Control-flow sub-formula</th>
<th>Data-flow sub-formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(x, y, n) )</td>
<td>( n \geq 0 \land x = n \land y = n )</td>
<td>( n \geq 0 )</td>
</tr>
<tr>
<td>( L(x, y, n) )</td>
<td>( n \geq 0 )</td>
<td>( x = n, y = n )</td>
</tr>
<tr>
<td>( L(x', y', n') )</td>
<td>( \land x \neq 0 )</td>
<td>( x = x' - 1, y = y' - 1 )</td>
</tr>
<tr>
<td>( y' = y \land n' = n )</td>
<td>( \land empty )</td>
<td>( x' = x, y' = y, n' = n )</td>
</tr>
<tr>
<td>false</td>
<td>( \land x = 0 \land y \neq 0 )</td>
<td>( y \neq 0 )</td>
</tr>
</tbody>
</table>

Constructing a CDHG. The CDHG represents program control- and data-flow by guarded control-flow hyperedges \( CFHE \) and data-flow hyperedges \( DFHE \). A \( CFHE \) edge denotes the flow of control from the body to head literals of the CHC. A \( DFHE \) edge denotes how data flows from the body to the head. Both control- and data-flow are guarded by the control flow sub-formula.

Constructing the CDHG for a normalized CHC \( L(x, y, n) \) is shown in Figure 3. The corresponding node and edge types are described in Tables 5 and 6 in the full version of this paper [28].

In the first step, we draw relation symbols and their arguments and build the relation between them. In the second step, we add a \( guard \) node and draw ASTs for the control flow sub-formulas. In the third step, we construct guarded control-flow edges by connecting the relation symbols in the head and body and the \( guard \) node, which connects the root of control flow sub-formulas. In the fourth step, we construct the ASTs for the right-hand side of every data flow sub-formula. In the fifth step, we construct the guarded data-flow edges by connecting the left- and right-hand sides of the data flow sub-formulas and the \( guard \) node. Note that the diamond shapes in Figure 3 are not nodes in the graph but are used to visualize our (ternary) hyperedges of types \( CFHE \) and \( DFHE \). We show it in this way to visualize CDHG better.

4. Relational Hypergraph Neural Network

Different from regular graphs, which connect nodes by binary edges, CDHG includes hyperedges which connect arbitrary number of nodes. Therefore, we extend R-GCN to R-HyGNN to handle hypergraphs. The updating rule for node representation at time step \( t \) in R-HyGNN is

\[
h_v^t = \text{ReLU}\left( \sum_{r \in R} \sum_{p \in P_r} \sum_{e \in \mathcal{E}_{r,p}} W_{r,p}^t \cdot ||h_{u}^{t-1} | u \in e|| \right),
\]

where \( ||\cdot|| \) denotes concatenation of all elements in a set, \( r \in R = \{ r_i | i \in \mathbb{N} \} \) is the set of edge types (relations), \( p \in P_r = \{ p_j | j \in \mathbb{N} \} \) 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 \( e \in \mathcal{E}_{r,p} \) 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 updating process for the representation of node \( v \) at time step 1 is illustrated in Figure 4.

Note that different edge types may have the same number of connected nodes. For instance, in CDHG, \( CFHE \) and \( DFHE \) are both ternary edges.
Horn clause: \(L(x, y, n) \leftarrow L'(x', y', n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1\)

Step 1: \(L(x, y, n), L'(x', y', n')\)

Step 2: \(x \neq 0\)

Step 3: \(L(x, y, n) \leftarrow L(x', y', n') \land x \neq 0\)

Step 4: \(x = x' - 1 \land y = y' - 1\)

Step 5: \(L(x, y, n) \leftarrow L'(x', y', n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1\)

Figure 3: Construct the CDHG from the CHC \(L(x, y, n) \leftarrow L'(x', y', n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1\).

Overall, our definition of R-HyGNN is a generalization of R-GCN. Concretely, it can directly be applied to the special-case of binary graphs, and in that setting is slightly more powerful as each message between nodes is computed using the representations of both source and target nodes, whereas R-GCN only uses the source node representation.

4.1. Training Model

The end-to-end model consists of three components: the first component is an embedding layer, which learns to map the node’s type (encoded by integers) to the initial feature vectors; the second component is R-HyGNN, which learns program features; the third component is a set of fully connected neural networks, which learns to solve a specific task by using gathered node representations from R-HyGNNs. All parameters in these three components are trained together. Note that different tasks may gather different node representations. For example, Task 1 gathers all node representations, while Task 2 only gathers node representations with type \(r_s\).

We set all vector lengths to 64, i.e., the embedding layer output size, the middle layers’ neuron size in R-HyGNNs, and the layer sizes in fully connected neural networks are all 64. The maximum training epoch is 500, and the patient is 100. The number of message passing steps is 8 (i.e., (2) is applied eight times). For the rest of the parameters (e.g., learning rate, optimizer, dropout rate, etc.), we use the default setting in the tf2_gnn framework. We set these parameters
Figure 4: An example to illustrate how to update node representation for R-HyGNN using (2). At the right-hand side, there are three types of edges connected with node 1. We compute the updated representation $h_1^1$ for node 1 at the time step 1. $\parallel$ means concatenation. $x_i$ is the initial feature vector of node $i$. The red blocks are the trace of the updating for node 1. The edge type 1 is a unary edge and is a self-loop. It has one set of learnable parameters as the update function i.e., $W_{r_0,p_1}$. The edge type 2 is binary edge, it has two update functions i.e., $W_{r_1,p_1}$ and $W_{r_2,p_1}$. Node 1 is in the first position in edge [1,2], [1,3], [1,4], and [1,5], so the concatenated node representation will be updated by $W_{r_1,p_1}$. On the other hand, for the other two edges [6,1] and [7,1], node 1 is in the second position, so the concatenated node representation will be updated by $W_{r_2,p_1}$. For edge type 3, the same rule applies, i.e., depending on node 1’s position in the edge, the concatenated node representation will go through different parameter sets. Since there is no edge that node 1 is in the second position, we use a dashed box and arrow to represent it. The aggregation is to add all updated representations from different edge types.

empirically according to the graph size and the structure. We apply these fixed parameter settings for all tasks and two graph representations without fine-tuning.
5. Proxy Tasks

We propose five proxy tasks with increasing difficulty to systematically evaluate the R-HyGNN on the two graph representations. Tasks 1 to 3 evaluate if R-HyGNN can solve general problems in graphs. In contrast, Tasks 4 and 5 evaluate if combining our graph representations and R-HyGNN can learn program features to solve the encoded program verification problems. We first describe the learning target for every task and then explain how to produce training labels and discuss the learning difficulty.

Task 1: Argument identification. For both graph representations, the R-HyGNN model performs binary classification on all nodes to predict if the node type is a relation symbol argument (rs) and the metric is accuracy. The binary training label is obtained by reading explicit node types. This task evaluates if R-HyGNN can differentiate explicit node types. This task is easy because the graph explicitly includes the node type information in both typed nodes and edges.

Task 2: Count occurrence of relation symbols in all clauses. For both graph representations, the R-HyGNN model performs regression on nodes with type rs to predict how many times the relation symbols occur in all clauses. The metric is mean square error. The training label is obtained by counting the occurrence of every relation symbol in all clauses. This task is designed to see if R-HyGNN can correctly perform a counting task. For example, the relation symbol $L$ occurs four times in all CHCs in Figure 2, so the training label for node $L$ is value 4. This task is harder than Task 1 since it needs to count the connected binary edges or hyperedges for a particular node.

Task 3: Relation symbol occurrence in SCCs. For both graph representations, the R-HyGNN model performs binary classification on nodes with type rs to predict if this node is an SCC (i.e., in a cycle) and the metric is accuracy. The binary training label is obtained using Tarjan’s algorithm [19]. For example, in Figure 2, $L$ is an SCC because $L$ and $L'$ construct a cycle by $L \leftarrow L'$ and $L' \leftarrow L$. This task is designed to evaluate if R-HyGNN can recognize general graph structures such as cycles. This task requires the model to classify a graph-theoretic object (SCC), which is harder than the previous two tasks since it needs to approximate a concrete algorithm rather than classifying or counting explicit graphical elements.

Task 4: Existence of argument bounds. For both graph representations, we train two independent R-HyGNN models which perform binary classification on nodes with type rsa to predict if individual arguments have (a) lower and (b) upper bounds in the least solution of a set of CHCs, and the metric is accuracy. To obtain the training label, we apply the Horn solver Eldarica to check the correctness of guessed (and successively increased) lower and upper arguments bounds; arguments for which no bounds can be shown are assumed to be unbounded. We use a timeout of 3 s for the lower and upper bound of a single argument, respectively. The overall timeout for extracting labels from one program is 3 hours. For example, consider the CHCs in Fig. 1c. The CHCs contain a single relation symbol $L$; all three arguments of $L$ are
bounded from below but not from above. This task is (significantly) harder than the previous three tasks, as boundedness of arguments is an undecidable property that can, in practice, be approximated using static analysis methods.

**Task 5: Clause occurrence in counter-examples**  This task consists of two binary classification tasks on nodes with type `guard` (for CDHG), and with type `clause` (for constraint graph) to predict if a clause occurs in the counter-examples. Those kinds of nodes are unique representatives of the individual clauses of a problem. The task focuses on unsatisfiable sets of CHCs. Every unsatisfiable clause set gives rise to a set of minimal unsatisfiable subsets (MUSes); MUSes correspond to the minimal CEs of the clause set. Two models are trained independently to predict whether a clause belongs to (a) the intersection or (b) the union of the MUSes of a clause set. The metric for this task is accuracy. We obtain training data by applying the Horn solver Eldarica [25], in combination with an optimization library that provides an algorithm to compute MUSes\(^7\). This task is hard, as it attempts the prediction of an uncomputable binary labelling of the graph.

6. Evaluation

We first describe the dataset we use for the training and evaluation and then analyse the experiment results for the five proxy tasks.

6.1. Benchmarks and Dataset

Table 3 shows the number of labelled graph representations from a collection of CHC-COMP benchmarks [20]. All graphs were constructed by first running the preprocessor of Eldarica [25] on the clauses, then building the graphs as described in Section 3, and computing training data. For instance, in the first four tasks we constructed 2337 constraint graphs with labels from 8705 benchmarks in the CHC-COMP LIA-Lin track (linear Horn clauses over linear integer arithmetic). The remaining 6368 benchmarks are not included in the learning dataset because when we construct the graphs, (1) the data generation process timed out, or (2) the graphs were too big (more than 10,000 nodes), or (3) there was no clause left after simplification. In Task 5, since the label is mined from CEs, we first need to identify unsat benchmarks using a Horn solver (1-hour timeout), and then construct graph representations. We obtain 881 and 857 constraint graphs when we form the labels for Task 5 (a) and (b), respectively, in LIA-Lin.

Finally, to compare the performance of the two graph representations, we align the dataset for both two graph representations to have 5602 labelled graphs for the first four tasks. For Task 5 (a) and (b), we have 1927 and 1914 labelled graphs, respectively. We divide them to train, valid, and test sets with ratios of 60%, 20%, and 20%. We include all corresponding files for the dataset in a Github repository \(^8\).

\(^7\)https://github.com/uuverifiers/lattice-optimiser/
\(^8\)https://github.com/ChenchengLiang/Horn-graph-dataset
Table 3
The number of labeled graph representations extracted from a collection of CHC-COMP benchmark [20]. For each SMT-LIB file, the graph representations for Task 1,2,3,4 are extracted together using the timeout of 3 hours, and for task 5 is extracted using 20 minutes timeout. Here, T. denotes Task.

<table>
<thead>
<tr>
<th>SMT-LIB files</th>
<th>Constraint graphs</th>
<th>CDHGs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Unsat</td>
</tr>
<tr>
<td>Linear LIA</td>
<td>8705</td>
<td>1659</td>
</tr>
<tr>
<td>Non-linear LIA</td>
<td>8425</td>
<td>3601</td>
</tr>
<tr>
<td>Aligned</td>
<td>17130</td>
<td>5260</td>
</tr>
</tbody>
</table>

6.2. Experimental Results for Five Proxy Tasks
From Table 4, we can see that for all binary classification tasks, the accuracy for both graph representations is higher than the ratios of the dominant labels. For the regression task, the scattered points are close to the diagonal line. These results show that R-HyGNN can learn the syntactic and semantic information for the tasks rather than performing simple strategies (e.g., fill all likelihood by 0 or 1). Next, we analyse the experimental results for every task.

Task 1: Argument identification. When the task is performed in the constraint graph, the accuracy of prediction is 100%, which means R-HyGNN can perfectly differentiate if a node is a relation symbol argument $r_{sa}$ node. When the task is performed in CDHG, the accuracy is close to 100% because, unlike in the constraint graph, the number of incoming and outgoing edges are fixed (i.e., $RSA$ and $AI$), the $r_{sa}$ nodes in CDHG may connect with a various number of edges (including $RSA$, $AST_1$, $AST_2$, and $DFHE$) which makes R-HyGNN hard to predict the label precisely.

Besides, the data distribution looks very different between the two graph representations because the normalization of CHCs introduces new clauses and arguments. For example, in the simplified CHCs in Figure 1c, there are three arguments for the relation symbol $L$, while in the normalized clauses in Figure 2, there are six arguments for two relation symbols $L$ and $L'$. If the relation symbols have a large number of arguments, the difference in data distribution between the two graph representations becomes larger. Even though the predicted label in this task cannot directly help solve the CHC-encoded problem, it is important to study the message flows in the R-HyGNNs.

Task 2: Count occurrence of relation symbols in all clauses. In the scattered plots in Figure 5, the x- and y-axis denote true and the predicted values in the logarithm scale, respectively. The closer scattered points are to the diagonal line, the better performance of predicting the number of relation symbol occurrences in CHCs. Both CDHG and constraint graph show good performance (i.e., most of the scattered points are near the diagonal lines). This syntactic information can be obtained by counting the $CFHE$ and $RSI$ edges for CDHG and constraint graph, respectively. When the number of nodes is large, the predicted values are less accurate. We believe this is because graphs with a large number of nodes have a more complex structure, and there is less training data. Moreover, the mean square error for the CDHG is larger than the constraint graph because normalization increases the number of nodes and the
maximum counting of relation symbols for CDHG, and the larger range of the value is, the more difficult for regression task. Notice that the number of test data (1115) for this task is less than the data in the test set (1121) shown in Table 3 because the remaining six graphs do not have a $r_s$ node.

**Task 3: Relation symbol occurrence in SCCs.** The predicted high accuracy for both graph representations shows that our framework can approximate Tarjan’s algorithm [19]. In contrast to Task 2, even if the CDHG has more nodes than the constraint graph on average, the CDHG has better performance than the constraint graph, which means the control and data flow in CDHG can help R-HyGNN to learn graph structures better. For the same reason as task 2, the number of test data (1115) for this task is less than the data in the test set (1121).

**Task 4: Existence of argument bounds.** For both graph representations, the accuracy is much higher than the ratio of the dominant label. Our framework can predict the answer for undecidable problems with high accuracy, which shows the potential for guiding CHC solvers. The CDHG has better performance than the constraint graph, which might be because predicting argument bounds relies on semantic information. The number of test data (1028) for this task is less than the data in the test set (1121) because the remaining 93 graphs do not have a $rsa$ node.

**Task 5: Clause occurrence in counter-examples.** For Task (a) and (b), the overall accuracy for two graph representations is high. We manually analysed some predicted results by visualizing the (small) graphs9. We identify some simple patterns that are learned by R-HyGNNs. For instance, the predicted likelihoods are always high for the $r_s$ nodes connected to the $false$ nodes.

9https://github.com/ChenchengLiang/Horn-graph-dataset/tree/main/example-analysis/task5-small-graphs
Experiment results for Tasks 1, 3, 4, 5. Both the fourth and fifth tasks consist of two independent binary classification tasks. Here, + and − stands for the positive and negative label. The T and P represent the true and predicted labels. The Acc. is the accuracy of binary classifications. The Dom. is dominant label ratio. Notice that even if the two graph representations originate from the same original CHCs, the label distributions are different since the CDHG is constructed from normalized CHCs.

<table>
<thead>
<tr>
<th>Task</th>
<th>Files</th>
<th>T</th>
<th>P</th>
<th>+</th>
<th>-</th>
<th>Accuracy</th>
<th>Dom.</th>
<th>+</th>
<th>-</th>
<th>Accuracy</th>
<th>Dom.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1121</td>
<td></td>
<td></td>
<td>93863</td>
<td>0</td>
<td>100%</td>
<td>95.1%</td>
<td>142598</td>
<td>0</td>
<td>99.9%</td>
<td>72.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1835971</td>
<td></td>
<td></td>
<td>10</td>
<td>381445</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1115</td>
<td></td>
<td></td>
<td>3204</td>
<td>133</td>
<td>96.1%</td>
<td>70.1%</td>
<td>8262</td>
<td>58</td>
<td>99.6%</td>
<td>50.7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>301</td>
<td>7493</td>
<td></td>
<td></td>
<td>15</td>
<td>8523</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (a)</td>
<td>1028</td>
<td></td>
<td></td>
<td>13685</td>
<td>3264</td>
<td>91.2%</td>
<td>79.7%</td>
<td>30845</td>
<td>4557</td>
<td>94.3%</td>
<td>75.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2928</td>
<td>71986</td>
<td></td>
<td></td>
<td>3566</td>
<td>103630</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (b)</td>
<td></td>
<td></td>
<td></td>
<td>18888</td>
<td>4792</td>
<td>91.4%</td>
<td>74.8%</td>
<td>41539</td>
<td>4360</td>
<td>94.3%</td>
<td>67.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3291</td>
<td>66892</td>
<td></td>
<td></td>
<td>3715</td>
<td>92984</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 (a)</td>
<td>386</td>
<td></td>
<td></td>
<td>1048</td>
<td>281</td>
<td>95.0%</td>
<td>84.7%</td>
<td>1230</td>
<td>206</td>
<td>96.9%</td>
<td>86.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>154</td>
<td>7163</td>
<td></td>
<td></td>
<td>121</td>
<td>9036</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 (b)</td>
<td>383</td>
<td></td>
<td></td>
<td>3030</td>
<td>558</td>
<td>84.6%</td>
<td>53.1%</td>
<td>3383</td>
<td>481</td>
<td>90.6%</td>
<td>54.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>622</td>
<td>3428</td>
<td></td>
<td></td>
<td>323</td>
<td>4361</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One promising result is that the model can predict all labels perfectly for some big graphs\(^{10}\), which contain more than 290 clauses, which confirms that the R-HyGNN is learning certain intricate patterns rather than simple patterns. In addition, the CDHG has better performance than the constraint graph, possibly because semantic information is more important for solving this task.

7. Related Work

Learning to represent programs. Contextual embedding methods (e.g., transformer [29], BERT [30], GPT [31], etc.) achieved impressive results in understanding natural languages. Some methods are adapted to explore source code understanding in text format (e.g., CodeBERT [32], cuBERT [33], etc.). But, the programming language usually contains rich, explicit, and complicated structural information, and the problem sets (learning targets) of it [34, 35] are different from natural languages. Therefore, the way of representing the programs and learning models should adapt to the programs’ characteristics. Recently, the frameworks consisting of structural program representations (graph or AST) and graph or tree-based deep learning models made good progress in solving program language-related problems. For example, [36] represents the program by a sequence of code subtokens and predicts source code snippets summarization by a novel convolutional attention network. Code2vec [37] learns the program from the paths in its AST and predicts semantic properties for the program using a path-based attention model. [38] use AST to represent the program and classify programs according to functionality using[10]https://github.com/ChenchengLiang/Horn-graph-dataset/tree/main/example-analysis/task5-big-graphs
the tree-based convolutional neural network (TBCNN). Some studies focus on defining efficient program representations, others focus on introducing novel learning structures, while we do both of them (i.e. represent the CHC-encoded programs by two graph representations and propose a novel GNN structure to learn the graph representations).

### Deep learning for logic formulas.
Since deep learning is introduced to learn the features from logic formulas, an increasing number of studies have begun to explore graph representations for logic formulas and corresponding learning frameworks because logic formulas are also highly structured like program languages. For instance, DeepMath [39] had an early attempt to use text-level learning on logic formulas to guide the formal method’s search process, in which neural sequence models are used to select premises for automated theorem prover (ATP). Later on, FormulaNet [40] used an edge-order-preserving embedding method to capture the structural information of higher-order logic (HOL) formulas represented in a graph format. As an extension of FormulaNet, [41] construct syntax trees of HOL formulas as structural inputs and use message-passing GNNs to learn features of HOL to guide theorem proving by predicting tactics and tactic arguments at every step of the proof. LERNA [42] uses convolutional neural networks (CNNs) [43] to learn previous proof search attempts (logic formulas) represented by graphs to guide the current proof search for ATP. NeuroSAT [44, 45] reads SAT queries (logic formulas) as graphs and learns the features using different graph embedding strategies (e.g. message passing GNNs) [46, 47, 26]) to directly predict the satisfiability or guide the SAT solver. Following this trend, we introduce R-HyGNN to learn the program features from the graph representation of CHCs.

### Graph neural networks.
Message passing GNNs [26], such as graph convolutional network (GCN) [17], graph attention network (GAT) [48], and gated graph neural network (GGNN) [47] have been applied in several domains ranging from predicting molecule properties to learning logic formula representations. However, these frameworks only apply to graphs with binary edges. Some spectral methods have been proposed to deal with the hypergraph [49, 50]. For instance, the hypergraph neural network (HGNN) [51] extends GCN proposed by [52] to handle hyperedges. The authors in [53] integrate graph attention mechanism [48] to hypergraph convolution [52] to further improve the performance. But, they cannot be directly applied to the spatial domain. Similar to the fixed arity predicates strategy described in LambdaNet [18], R-HyGNN concatenates node representations connected by the hyperedge and updates the representation depending on the node’s position in the hyperedge.

## 8. Conclusion and Future Work

In this work, we systematically explore learning program features from CHCs using R-HyGNN, using two tailor-made graph representations of CHCs. We use five proxy tasks to evaluate the framework. The experimental results indicate that our framework has the potential to guide CHC solvers analysing Horn clauses. In future work, among others we plan to use this framework to filter predicates in Horn solvers applying the CEGAR model checking algorithm.
References


