Systematic Predicate Abstraction using Deep Learning

A Preprint

Name [*]
Department of
University
Address
xxx@xxx

Name [*]
Department of
University
Address
xxx@xxx

Name [*]
Department of
University
Address
xxx@xxx

Abstract

Systematic Predicate Abstraction using Deep Learning

***K*eywords** First keyword · Second keyword · More

1 Introduction

Systematic Predicate Abstraction using Deep Learning

1.1 Problem overview and working pipeline

Our purpose is to find better predicates that can represent correct abstract transition system by a learned heuristic. In CEGAR iteration, the predicates come from two sources. One is initial predicates that can be generated by some heuristics before the first iteration [cite]. Another is the predicates generated from counter-examples after the first iteration. We can add an heuristic to generate better predicates in both situations. The heuristic is the importance of arguments. It can be represented by counting the number of occurrence of arguments in predicates.

In the case of verifiable programs, we expect the learned heuristic can reduce CEGAR iterations or total time consumption for verifying the program. In the case of unverifiable programs (cannot verify the program within a fixed time), we expect the learned heuristic can help Eldarica to verify it.

The working pipeline is:

1. Collect training inputs: Feed verifiable programs to Eldarica. Eldarica transforms the program to horn clauses and horn graphs. Eldarica uses current heuristic to solve the horn clauses. During the CEGAR process, some predicates are generated.
2. Collect training labels: In in generated predicates, count the number of occurrence of arguments appeared in horn clauses to be the training labels.
3. Train a model: A GNN + multilayer perceptrons model is built and trained. The input is horn graph, the output is the number of occurrence of arguments.
4. Apply trained model: For one program, apply Eldarica to transform the program to horn graph and feed it to the trained model. The output is the number of occurrence (i.e. importance) of arguments.
5. Apply predicted results: Use the output as a heuristic to start a CEGAR process to solve the horn clauses transformed from the program by Eldarica.

[*]All contributions are considered equal.

1.2 Contribution

- We define horn graph which captures control flow and data flow in horn clauses. The semantics and syntaxes of the program is preserved as much as possible. Horn graph bridges horn clauses and graph related analysis and learning methods.
- We provides a way to build a interface that take horn clauses as inputs and node features or graph representation as output for deep learning tasks based on horn clauses.
- We extend GNN to handle hyperedges
- Experiment results give confidence that

2 Background

2.1 Abstraction Based Model Checking

Model checking is a technique for program verification. It builds a transition system based on variables' states in all control locations and to check if there is a path to the error states. The error states are built by specifications given by users. But, programs with infinite states are ubiquitous. Therefore, we need to find a finite transition system (abstract transition system $\widehat{M}$) to represent the infinite transition system M, such that if there is no path to the error state in $\widehat{M}$, M has no error as well, and if there is a path to the error state in $\widehat{M}$, we can use this concrete error path to check if there is a error in M.

The major challenge in this idea is to find a appropriate $\widehat{M}$. CEGAR [1] is a state-of-art framework to find $\widehat{M}$ iteratively. CEGAR framework is described in Figure 1. It starts with a initial abstract transition system $\widehat{M}$. The model checker can check if there is a path to the error state. If there is no such path, the the system M is safe, otherwise give the path as a counterexample to check the feasibility. Checking feasibility means to use this counterexample as input to M If M return a error, this counterexample is feasible and M is not safe. If there is no error returned, this counterexample is infeasible which means $\widehat{M}$ does not represent M. Therefore, we need a new abstract transition system $\widehat{M'}$ obtained by a refinement process.

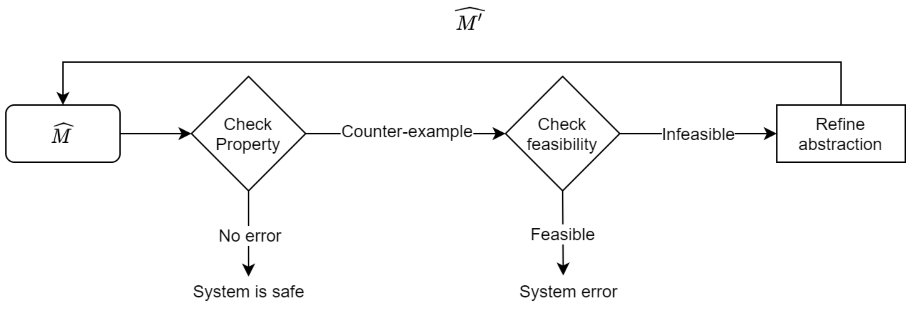

Figure 1: CEGAR framework

$\widehat{M}$ is built by a set of predicates P. To update $\widehat{M}$ to $\widehat{M'}$ means to update P to a new predicate set P' by adding new predicates. The new predicates are constructed by splitting the counterexample path to two parts A and B and find the Interpolants I of A and B, such that I consists of common elements in A and B, $A \Rightarrow I$, and $I \Rightarrow \neg B$. I is the new predicates to be added to P.

2.1.1 Heuristics for predicate generating

— Abstract Interpolation [2].

Manually constructed templates [2]

Learned argument importance as heuristic.

2.2 GNN

The program is transformed to horn graph which preserves the structural information of the graph by catching control flow and data flow. We apply tf2_gnn framework to learn the node features from graph inputs. tf2_gnn is a implementation of general GNN framework which provides standard inputs and outputs for graph learning. Except implementing several popular GNN algorithms, such as [3, 4], it also provides great flexibility to implement customized GNNs. The inputs of tf2_gnn is node vector and edges which represent the relations between node vectors. The output can be node features or a graph feature. Since we want to learn the importance of arguments. We take node features as outputs. We gather these learned argument node features and apply multilayer perceptrons to learn the importance of them.

2.2.1 Message passing and hyperedge extension

—

3 Program representation

Our raw inputs are programs in text format and could be written in different languages. We first transform programs to horn clauses which can preserve the semantics. We perform learning process in Horn clauses level. In this way, we can eliminate the affects from different program languages.

But, Horn clauses are also in text format and text-level embedding methods [5] are not enough to capture the structural information. The downstream multilayer perceptron cannot learn well from the text-level embedding. Therefore, we transform Horn clauses to graph format which contains both control flow and data flow information that capture the structure information from Horn clauses, then we use GNN to learn the representation of the horn graph to be the representation of the original program.

3.1 Program to Horn clauses

Eldarica can transform SMT or C format programs to intermediate horn clause format and the semantics are preserved [cite]. By using horn clauses as the learning inputs, we don't need to care about the input languages but focus on solving the logic problems. We have an example in Figure 2a and 2b to explain how to transform c program to horn clauses and preserving semantics.

In Figure 2a, we have a simple while loop C program which assumes variable x and y equal to external input Integer n and $n \geq 0$. The assertion is $y == 0$. The corresponding Horn clauses are described in Figure 2b. Every line is consists of $Head : -Body, [constraint]$. For example, in line 0, $Head$ is $inv\_main4(x, y)$ which means in control location $while(x! = 0)$ in the main function, there are two variables x and y. $Body$ is empty here because this is the initial state of the program. The $constraint$ is $n \geq 0 \wedge x = n \wedge n = 0 \wedge y = n$ which comes from the $assume(x == n\&\&y == n\&\&n >= 0)$ in the C program. Line 0 means in control location $while(x! = 0)$, there are two variables x, and y, which satisfy the $constraint : n \geq 0 \wedge x = n \wedge n = 0 \wedge y = n$. In line 1, The body is not empty, this means that there is a transition from body to head and this transition satisfy the $constraint$. Line 1 means from control location $while(x! = 0)$ to next iteration, it must satisfy that, $x! = 0$ and in the new iteration, the variables $arg1$ and $arg2$ in the control location $while(x! = 0)$ must equal to $x - 1$ and $y - 1$, (values of x, y come from last iteration). Line 2 transforms the semantic of assertion $y == 0$ to horn clauses. Line 2 means that from control location $while(x! = 0)$ to false state, in the control location $while(x! = 0)$, the variable x is 0 and y satisfy $y! = 0$.

```
extern int n;
void main(){
    int x,y;
    assume(x==n&&y==n&&n>=0);
    while(x!=0){
        x--;
        y--;
    }
    assert(y==0);
}
```

(a) An input example: C program

```
0  inv_main4(x, y) :- ,[n >= 0 & x = n & 0 = n & y =
       n].
1  inv_main4(arg1, arg2) :- inv_main4(x, y), [x != 0
       & x + -1 = arg1 & y + -1 = arg2 & n = 0].
2  false :- inv_main4(0, y), [y != 0].
```

(b) Horn clauses for C program

Figure 2

How to transform program in c or smt2 format to horn clauses and why the sematic can be preserved can be found in [cite]. Our main purpose is to use this horn clauses format as input of deep learning structure to perform downstream tasks.

3.2 Horn clauses to Horn graph

As we mentioned before, horn clauses in text format is hard to learn the structure information. Therefore, we need transform horn clauses to horn graph. We defined a way to construct Horn graph from Horn clauses, which preserves syntactical and sematic information as much as possible. This definition is not unique, one can define their own horn graph.

We first provided an example that transform horn clauses in Figure 2b to horn graph in Figure 6. Then, we give formal definition of our horn graph construction process.

3.2.1 Horn graph construction example

The graph is constructed by parsing the horn clauses line by line. In Figure 2b, every line is a horn clause. In , Line 0 "inv_main4(x, y) :- ,[n >= 0 & x = n & 0 = n & y = n]", the left hand side of :- is $Head$ and the right hand side before ",[]"is $Body$, and content in "[]" is $Constrains$. In line 0, $Heaed$ is "inv_main4(x, y), $Body$ is empty, and $Constrain$ is "[n >= 0 & x = n & 0 = n & y = n]. The $Constraint$ are constraints to make the transition from $Body$ to $Head$ satisfiable.

Figure 3 shows how to draw Line 0. In $Head$ "inv_main4(x, y)", argument nodes V_{arg}(x and y) are drawn as inv_main4_argument_0 and inv_main4_argument_1. They point to control location node CL_{normal} "inv_main4" by argument edges E_{Arg} (dotted lines). $Body$ is empty, we represent it as initial control location node $CL_{initial}$. In $Constrain$ "[n >= 0 & x = n & 0 = n & y = n]", $n >= 0$ and $0 = n$ are $Guard$ because they don't have assignment to arguments. n is a free variable V_{free}, 0 is a constant V_C. They can be drawn as a syntax tree and connected by operator node V_{OP} &, $>=$, and $=$. $x = n$ and $y = n$ are $DataFlow$ because they contain argument assignment operation. This means argument x and y are assigned with free variable n. We connect the free variable n to argument nodes x and y through guarded data flow hyperedge HE_{DF}. Control location node in $Head$ and $Body$ is connected through control flow hyperedge HE_{CF}. All guarded hyperedges are connected to the root node of syntax tree drawn by $Guard$. In this case, the root of $Constraint$' syntax tree is operator $\&$. The intuition is all control flow from $Body$ to $Head$ and data flow from elements in $Constrain$ to arguments are guarded by $Guard$. If there is no $Guard$ in $Constrain$, a constant true boolean node points to the hyperedges build in this transition.

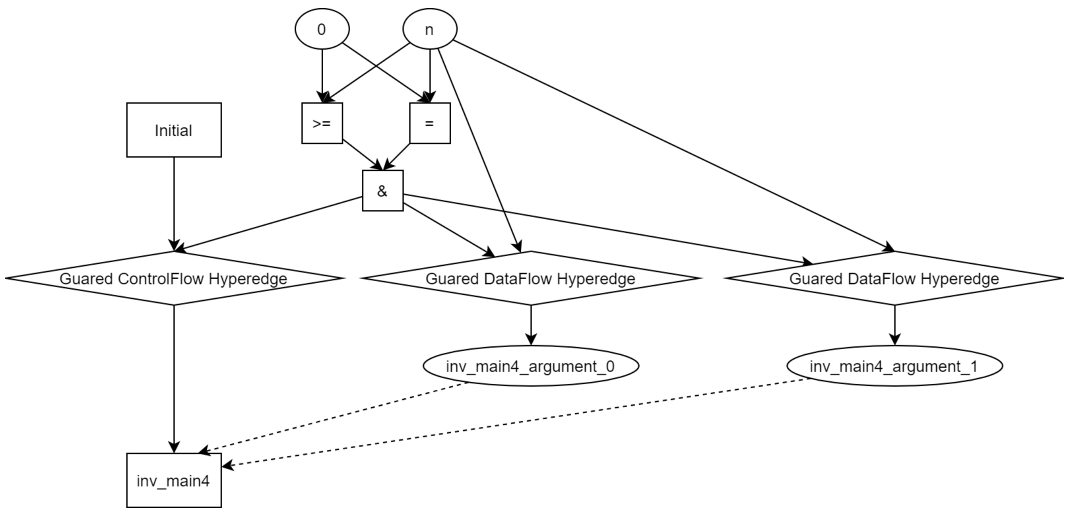
0
n
Initial
>=
=
&
Guarded ControlFlow Hyperedge
Guarded DataFlow Hyperedge
Guarded DataFlow Hyperedge
inv_main4_argument_0
inv_main4_argument_1
inv_main4

Figure 3: Horn graph for horn clause inv_main4(x, y) :- ,[n >= 0 & x = n & 0 = n & y = n]

In the similar way, Figure 4 illustrates how to draw Line 1 "inv_main4(arg1, arg2) :- inv_main4(x, y), [x != 0 & x + -1 = arg1 & y + -1 = arg2 & n = 0]". Since $Head$ and $Body$ have the same control location node and arguments, control location node "inv_main4" points to itself and guarded by a control flow hyperedge. arg1 is x and arg2 is y. They are drawn as inv_main4_argument_0 and inv_main4_argument_1 and connected to the control location node "inv_main4". $Constrain$ [x != 0 & x + -1 = arg1 & y + -1 = arg2 & n = 0] includes two $Guard$ n = 0, x != 0 and two $dataFlow$ x + -1 = arg1, y + -1 = arg2.

Figure 5 shows how to construct Line 2 "false :- inv_main4(0, y), [y != 0]". We draw a connection from $Body$'s control location node "inv_main4" to $Head$' control location node False CL_{false}. $Body$'s first argument (i.e. inv_main4_argument_0) imply a data flow (inv_main4_argument_0 $\leftarrow$ 0). This data flow is not from $Constrain$,

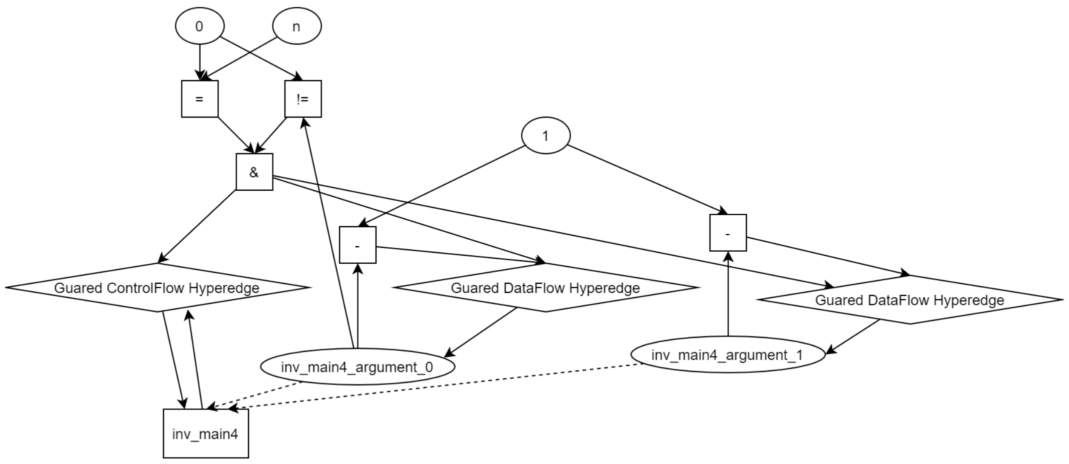
0
n
=
!=
&
1
-
-
Guarded ControlFlow Hyperedge
Guarded DataFlow Hyperedge
Guarded DataFlow Hyperedge
inv_main4_argument_0
inv_main4_argument_1
inv_main4

Figure 4: Horn graph for horn clause inv_main4(arg1, arg2) :- inv_main4(x, y), [x != 0 & x + -1 = arg1 & y + -1 = arg2 & n = 0]

so there is no guard for it. $Constrain$ [y != 0] is a $Guard$ because it is not a assignment operation. We know inv_main4_argument_0 = 0 and inv_main4_argument_1 = y, so we connect inv_main4_argument_0 and inv_main4_argument_1 with operator node != . Then we connect the root of $Guard$ (i.e. !=) to guarded hyperedges.

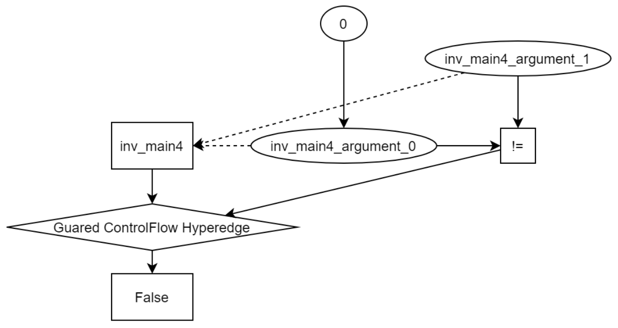

Figure 5: Horn graph for horn clause false :- inv_main4(0, y), [y != 0]

A overview of the example horn graph construction is shown in Figure 6

3.2.2 Horn graph Definition

We first define all nodes and edges in the graph. Formally, the graph consists of four categories of nodes, two categories of hyperedges, and five categories of binary edges. The graph is defined as $G = (V_{CL}, V_{Vri}, V_C, V_{Op}, E_{CF}, E_{DF}, E_{CD}, E_{Arg}, E_{ST}, HE_{CF}, HE_{DF})$, in which V_{CL} has three variance nodes $\{CL_{Initial}, CL_{normal}, CL_{false}\}$. They are control location nodes. V_{Vri} is variable node, The variances are argument node V_{arg} and free variable nodeV_{free}. V_C is constant node. V_{Op} is operator node which represents unary or binary operators. E_{CF} is control flow edge. It connects V_{CL}. E_{DF} is data flow edge. It connect data flows to arguments. E_{CD} is condition edge. It connects the root of $Guard$ through hyperedges to V_{CL} and V_{arg}. The intuition is that all control flow and data flow are guarded by $Guard$ through hyperedges. E_{ARG} is argument edge. It connect arguments to the corresponding control locations. The intuition is that the arguments belong to these control locations. E_{ST} is syntax tree edge. All elements in syntax tree built by $Guard$ are connected by E_{ST}. HE_{CF} is guarded control flow hyperedge. It connects V_{CL} and guarded by the root of $Guard$ syntax tree. HE_{DF} is guarded fata flow hyperedge. It connect all data flow elements from V_{Vri} and V_C to V_{arg} guarded by the root of $Guard$ syntax tree as well.

The summary of G's nodes and edges is shown in Table 1 and 2 respectively.

Horn graph construction.
We can represent the transition by $Head(V_{CL}^{Head}, V_{arg}^{Head}) : -Body(V_{CL}^{Body}, V_{arg}^{Body}), [Constraint(Guard, DataFlow)]$. We can construct the graph by following steps:

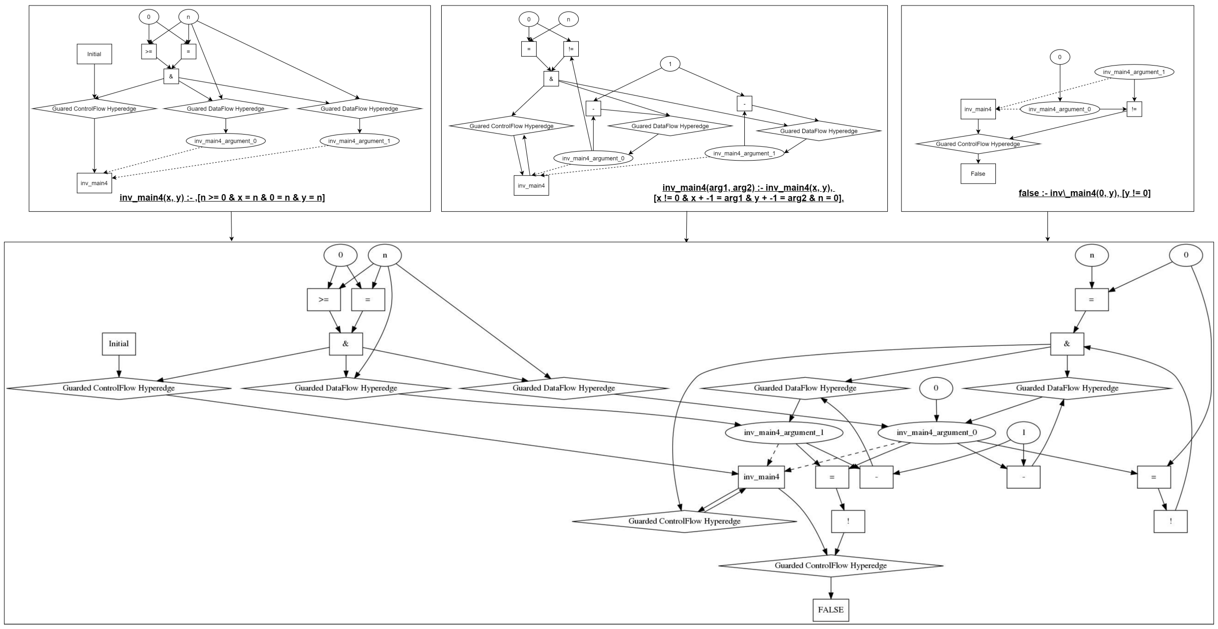
inv_main4(x, y) :- ,[n >= 0 & x = n & 0 = n & y = n]
inv_main4(arg1, arg2) :- inv_main4(x, y),
[x != 0 & x + -1 = arg1 & y + -1 = arg2 & n = 0],
false :- inv_main4(0, y), [y != 0]
Initial
Guarded ControlFlow Hyperedge
Guarded DataFlow Hyperedge
inv_main4_argument_0
inv_main4_argument_1
inv_main4
False
FALSE

Figure 6: Overview of Horn graph construction for Figure 2b

Table 1: Nodes in graph

Graph Nodes	Name	Sub-elements	Connected edge types
V_{CL}	Control location node	$CL_{Initial}$, CL_{normal}, CL_{false}	E_{CF}, HE_{CF}
V_{Vri}	Variable node	V_{arg}, V_{free}	E_{Arg}, HE_{DF}, E_{ST}
V_C	Constant node	-	$E_{CF}, E_{DF}, E_{CD}, E_{ST}, HE_{CF}, HE_{DF}$
V_{Op}	Operator node	-	$E_{DF}, E_{CD}, E_{ST}, HE_{CF}, HE_{DF}$

1. We connect arguments to corresponding control locations in $Head$ and $Body$. V_{arg}^{Head} connects to V_{CL}^{Head} and V_{arg}^{Body} connects to V_{CL}^{Body} by E_{ARG}. If $V_{CL}^{Head} == V_{CL}^{Body}$ which implies $V_{arg}^{Head} == V_{arg}^{Body}$, only one set of arguments and corresponding control location nodes are drawn.
2. We build the control flow by connecting control location node V_{CL}^{Body} in $Body$ to control location node V_{CL}^{Head} in $Head$ through guarded control flow hyperedge HE_{CF}. If $V_{CL}^{Head} == V_{CL}^{Body}$, it points to itself and also through HE_{CF}.
3. In $Constraint$, we have two types of expressions (i.e. $Dataflow$ and $Guard$). If the expression is a assignment, and there are arguments involved, then it is a $Dataflow$, otherwise, it is a $Guard$. For $Dataflow$. We first draw a syntax tree(connected by E_{ST}) for the right hand side of the assignment expression . The root of the syntax tree connects to the left hand side argument through data flow hyperedge HE_{DF}. For $Guard$, we build a syntax tree (connected by E_{ST}) for it and connect the root to all HE_{CF} and HE_{DF} in this transition by E_{CD}. This means all control flow and data flow are guarded by $Guard$. If there is no $Guard$ in $Constraint$, a constant node with boolean value $True$ will be the root of $Guard$.
4. $Head$ and $Body$ may imply some data flows. We can directly draw these data flows using E_{DF}. They are not guarded by $Guard$. The example can be found in previous Section 3.2.1

4 Data Collection

Our raw inputs are programs in form of .smt2. The customized Eldarica can transform the program to horn clause, then draw horn graph and save the graph to dot format (.gv file). We transform horn graphs to standard GNN inputs (nodes vectors and edges). GNN can learn the node representation. We gather argument node representations from all node representations to be the input of the multilayer perceptron. The output (label) of the multilayer perceptron for each

Table 2: Edges in graph

Graph Eedges	Name	Start from	End at
E_{CF}	Control flow edge	V_{CL}	V_{CL}
E_{DF}	Data flow edge	V_{Vri}, V_C, V_{Op}	V_{arg}
E_{CD}	Condition edge	V_C, V_{OP}	V_{CL}, V_{arg}
E_{ARG}	Argument edge	V_{Arg}	V_{CL}
E_{ST}	Syntax tree edge	V_{OP}, V_C, V_{Vri}	V_{OP}
HE_{CF}	Guarded control flow hyperedge	V_{OP}, V_C, V_{CL}	V_{CL}
HE_{DF}	Guarded data flow hyperedge	V_{OP}, V_C, V_{Vri}	V_{Arg}

argument representation input is the number of occurrence of that argument node appeared in refined predicates. The multilayer perceptron is trained with GNN together.

The refined predicates are obtained from Algorithm 1 and described in next subsection.

4.1 label collection

All programs in benchmarks can be solved by Eldarica in a limited time. We feed these programs to Eldarica, then the CEGAR process will produce a set of predicates to build the abstract transition system (abstract model). But these predicates are redundant. We use Algorithm 1 to further refine the predicates. Then, we count the number of occurrence of arguments in these refined predicates to be our training labels. If there is no refined predicate found, then there is no label to be collected.

Algorithm 1 describes a single instance of predicates refining process, the inputs are a program in form of .smt2 and a list of initial templates $TemplateList = \{T_0, T_1, ..., T_n\}$ crafted manually [2] or generated by simple heuristics[6] . The program will be transformed to horn clauses ($HornClauses$). The templates are heuristics for predicates generating and generated before the CEGAR iteration. $TemplateList$ could be a empty list which means no heuristic for predicate generating. The output is a set of refined predicates $CriticalPredicateList$.

With these two inputs, we start the CEGAR process. We can get a list of predicates ($ExtractedPredicateList$) derived from T and counter-examples in CEGAR iterations. If $ExtractedPredicateList$ is not empty, the predicates refining process begins. If $ExtractedPredicateList$ is empty, we don't need any predicate to verify the program. No label is extracted in this case.

We suppose there is no redundancy in $ExtractedPredicateList$ and assign $ExtractedPredicateList$ to $CriticalPredicateList$ before the refining process. Then, we delete predicates one by one from $CriticalPredicateList$ and see if we can verify the program with remaining predicates. To do so, we need a customized CEGAR $\widehat{CEGAR}$ which does not generate new predicates from counter-examples. In other words, $\widehat{CEGAR}$ will only use $CriticalPredicateList$ as initial predicates to build the abstract model.

If $\widehat{CEGAR}(HornClauses, CriticalPredicateList)$ returns False, which means $\widehat{CEGAR}$ cannot find the correct abstract model with remaining predicates within a fixed time, then we determine that the deleted one predicate is critical, and we add this back predicate to $CriticalPredicateList$. In the contrast, if $\widehat{CEGAR}(HornClauses, CurrentTemplateList)$ returns True, we don't add the predicate back to $CriticalPredicateList$.

By repetitively deleting predicate one by one from $CriticalPredicateList$, and check the solvability by $\widehat{CEGAR}$. We finally get a set of non-redundant predicates ($CriticalPredicateList$) that represents correct abstract model.

Notice that $ExtractedPredicateList$ may contain multiple $CriticalPredicateList$ that can represent correct abstract model in the fixed time. For example, $ExtractedPredicateList = \{P_0, P_1, P_2\}$. Both $\{P_0, P_1\}$ and $\{P_1, P_2\}$ can represent the abstract model correctly. This method only find one set of them. One can also first construct the permutation of $ExtractedPredicateList$, then use $\widehat{CEGAR}$ find all $CriticalPredicateList$. But, in the case of some complicated programs, there may be a large number of predicates in $ExtractedPredicateList$. Therefore, finding all $CriticalPredicateList$ is hard.

Algorithm 1: Predicates refining process

```
Input: TemplateList = $\{T_0, T_1, ..., T_n\}$, HornClauses;
Output: CriticalPredicateList ;
ExtractedPredicateList=CEGAR(TemplateList,HornClauses);
if ExtractedPredicateList is empty then
    No CriticalPredicateList extracted
else
    CriticalPredicateList=ExtractedPredicateList;
    for $P_k$ in ExtractedPredicateList do
        CriticalPredicateList = CriticalPredicateList $-\{P_i\}(i \in [0, n])$;
        Solvalbility=$\widehat{CEGAR}$(CurrentPredicateList,HornClauses);
        if Solvalbility == False then
            CriticalPredicateList = CriticalPredicateList $\cup P_i$;
        end
    end
end
```

5 Learning Model

The learning model is shown in Figure 7. The input horn graph is transformed from a program by Eldarica. For a horn graph, we transform the node to vectors by embedding layer. The edges include relation between node vectors. GNN takes node features and their relations (edges) as inputs. It outputs node embeddings. Then, we gather argument node embeddings $v'_{argument\ node}$ to be the inputs of multilayer perceptron. The final outputs are the number occurrences of arguments appeared in refined predicates. GNN and the multilayer perceptrons are trained together.

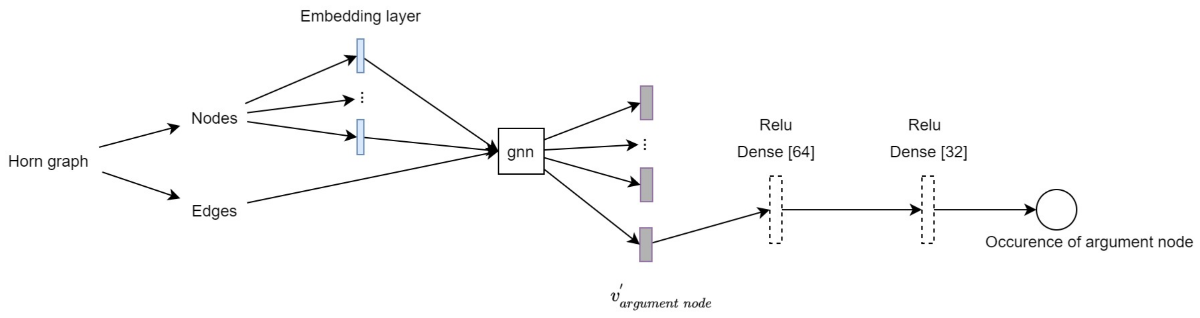

Figure 7: Training structure

5.1 Training results and analysis

We collected the number of arguments' occurrence to be our training label. Since we care about the importance of the arguments, we can transform the occurrence label to ranking and learn the ranking of arguments. So we have two variances of our training. One is to predict the occurrence using occurrence label directly, then rank the predicted number or normalize them to represent the importance and feed back to Eldarica. Another variance is to predict the ranking using ranking label.

The numerical results of training using different labels are shown in 3

Parameters: max_epochs=1000,patience=50

Table 3: Training results

Benchmarks	Label	Train loss	Valid loss	Test loss	Mean loss	Best valid epoch
LIA-lin	occurrence	-	-	-	-	-
LIA-lin	rank	-	-	-	-	-
LIA-nonlin	occurrence	-	-	-	-	-
LIA-nonlin	rank	-	-	-	-	-
In total	occurrence	-	-	-	-	-
In total	rank	-	-	-	-	-

6 Feed learned heuristic back to Eldarica

7 Evaluation

7.1 Benchmarks

We separate our benchmarks to linear or nonlinear group. All programs in the benchmarks can be verified by Eldarica in a limited time. In these verifiable programs, we use Algorithm 1 to select programs that need initial predicates to be verified. Then we transform these selected programs to horn graphs and divide them to train, valid, and test set randomly. The numerical details can be found in Table 4

Table 4: Benchmarks

Benchmarks	Total program	Graphs with labels	Train	Valid	Test
LIA-lin	7741 (7533)	413	247	82	84
LIA-nonlin	7653	1212	727	242	243
In total	15394(15186)	1625	974	324	327

7.2 Setup

Eldarica settings: Timeout for verifying a program.

learning model settings: R-GCN.

All code regarding the experiment can be found in Git repository https://github.com/ChenchengLiang/Systematic-Predicate-Abstraction-using-Machine-Learning

7.3 Experimental results

Benchmarks	Total program	No heuristic	Templates as heuristic	Learned heuristic
LIA-lin	-	-	-	-
LIA-nonlin	-	-	-	-
Total	-	-	-	-

Table 5: The number of verified programs in fixed time

8 Related work

DeepMath [7] is the first trial that perform text-level learning to Guide formal method's search process. They use neural sequence models to select premises for automated theorem prover (ATP). Their input pair for the learning model is character-level or word-level conjecture and axiom. Output is the relevance of these two inputs. With the predicted relevances (range [0,1]), a importance rank of a set of axioms to one conjecture is obtained. DeepMath treats conjecture and axiom as text-level information and builds character-level and word-level models to embed them.

Benchmarks	Average CEGAR iterations (No heuristic)	Average time consumption (No heuristic)	Average CEGAR iterations (learned heuristic)	Average time consumption (learned heuristic)
LIA-lin	-	-	-	-
LIA-nonlin	-	-	-	-
Total	-	-	-	-

Table 6: Average CEGAR iterations and time consumption

FormulaNet [8] performs the same task (premise selection for theorem proving) like DeepMath. But, FormulaNet represent represent the input (a higher-order logic formula) as a graph and provide a embedding method to catch the formula's representation.

Our input is program. Graph is a natural way to represent programs. There are different ways to learn the program graphs.

(1) End-to-end learning: TBCNN [9] takes AST as input and propose a tree-based convolutional neural network to learn program representation. *Learning to represent programs with graphs* [10] define a program graph based on AST to catch syntactic and semantic structure. Then, it uses GNN to learn the program representation.

(2) Semi-supervised or unsupervised graph embedding aims at using various methods to represent the graph to low-dimensional vector for down stream applications (e.g. classification, recommendation, etc.). The graph embedding and downstream networks are not trained jointly. Before combining with deep learning, graph embedding works in the context of dimensionality reduction. The typical techniques are principle component analysis (PCA), multidimensional scaling (MDS),Isomap [11], Locally Linear Embedding (LLE) [12], and Laplacian Eigenmap [13]. Their complexities are quadratic corresponding to the number of vertices. DeepWalk [14] maps the graph's nodes to sentence's words and uses word embedding technique skip-gram [15] to perform the node embedding. And, it is extended to Graph2Vec [16] to learn the graph representation.

9 Discussion and Future work

References

[1] 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.

[2] 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.

[3] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *arXiv e-prints*, page arXiv:1710.10903, Oct 2017.

[4] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[6] Jérôme Leroux, Philipp Rümmer, and Pavle Subotić. Guiding craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, Jun 2016.

[7] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, Francois Chollet, and Josef Urban. Deepmath - deep sequence models for premise selection. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2235–2243. Curran Associates, Inc., 2016.

[8] 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.

[9] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. TBCNN: A tree-based convolutional neural network for programming language processing. *CoRR*, abs/1409.5718, 2014.

[10] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[11] Joshua Tenenbaum, Vin Silva, and John Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 01 2000.

[12] Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[13] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 585–591. MIT Press, 2002.

[14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[15] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[16] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.