
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

Keywords First keyword · Second keyword · More

1 Introduction

Systematic Predicate Abstraction using Deep Learning

Reduce CEGAR iterations.

Summarize the pipeline:

2 Background

—

2.1 Abstraction Based Model Checking

—

2.2 CEGAR

—

2.3 Abstract Interpolation

—

3 Problem Overview

—

*All contributions are considered equal.

4 Program representation

Eldarica can take SMT and C format programs as inputs and transform the programs to intermediate horn clause format [cite]. In this format, the semantics are transformed to horn clauses. By using horn clauses as inputs, our method can adapt to many verification problems because many verification problems can be transformed to SMT format and eventually to horn clauses. We have an example in Figure 1a and 1b to explain how the horn clauses capture the semantics of the program.

In Figure 1a, 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 can be found in Figure 1b. 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 and there are two variables x and y inside. *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 $assume(x == n \& \& y == n \& \& n \geq 0)$ in the C program. Line 0 means in control location $while(x! = 0)$, there are two variables x , and y , which satisfy the *constraint*. 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 $y! = 0$.

```

0  extern int n;
1  void main() {
2      int x, y;
3      assume (x==n&&y==n&&n>=0);
4      while (x!=0) {
5          x--;
6          y--;
7      }
8      assert (y==0);
9  }
```

(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 1

How to transform program semantics to horn clauses and why it can be transformed in this way can be found in [cite]. Our main purpose is to use this horn clauses format as input of deep learning structure to select predicates to build valid abstract transition systems for model checkers.

Text streamed Horn clauses as inputs are not enough to make neural networks to understand the semantics because text-level embedding cannot capture the structural information included in the horn clauses. Therefore, we represent horn clauses by a graph which contains both control flow and data flow information, then we embed the graph to represent the original program.

The graph to represent horn clauses in Figure 1b is in Figure .

Formally, the graph consists of five categories of nodes, two categories of hyperedges, and six categories of edges. The graph is defined as $G = (V_{CL}, V_{Arg}, V_C, V_{Op}, V_{FV}, HE_{CF}, HE_{DF}, E_{CFI}, E_{CFO}, E_{DFI}, E_{DFO}, E_{CD}, E_{Arg})$, in which $V_{CL} = \{cl_{Initial}, cl_{Main_k}, cl_{false}\}$ is control location node set, $V_{Arg} = \{Arg_k\}$ is argument set, V_C is constant value set, $V_{Op} = \{+, -, ==, \dots\}$ is operator set, $V_{FV} = \{v_k\}$ is free variables set. k is a positive integer. HE_{CF} is a set of Guarded control flow hyper edges. it is used between two control locations. it takes two inputs, a control location $cl_{Initial}$ or cl_{Main_k} and a boolean value from operator. It has a control flow out edge e_{CFO} (one element from E_{CFO}) connected to next control location node. Similarly, HE_{DF} is a set of Guarded data flow hyper edges. It guards the data flow from *Body* to *Head*. It takes two inputs, a value from argument Arg_k or free variable v_k and a boolean value and has a data flow out edge e_{DFO} (one element from E_{DFO}) connected to next operator or argument node. E_{CFI}, E_{CFO} are control flow in and control flow out edges, they are only used when there are connections between nodes and guarded control flow hyperedges. similarly, E_{DFI}, E_{DFO} are data flow in and out edges, they are only corresponding to guarded data flow hyperedges. E_{CD} is condition edge. It connects boolean operator to the hyperedges to send boolean values to the hyperedges. E_{Arg} is argument edge. It connects arguments to corresponding control location node. We use lower case to represent one elements in the set. For example v_{CL} means one control location node from V_{CL} .

Table 1: Graph elements description

Graph elements	Name	Elements	Inputs	Outputs
V_{CL}	Control location node	$CL_{Initial}, CL_{Main_k}, CL_{false}$	v_{CL}	he_{CF}
V_{Arg}	Argument node	Arg_k	v_{OP}, v_{Arg}, v_{FV} , or v_C	
V_C	Constant node	Constants	-	
V_{Op}	Operator node	Arithmetic and boolean operators	-	
V_{FV}	Free Variable node	Free variables	-	variable
HE_{CF}	Guarded control flow hyperedge	-	v_{OP} or v_C, v_{CL}	v_{CL}
HE_{DF}	Guarded data flow hyperedge	-	v_{OP} or v_C, v_{Arg} or v_{FV}	v_{Arg}
E_{CFI}	Control flow in edge	-	v_{CL}	v_{CL}
E_{CFO}	Control flow out edge	-		
E_{DFI}	Data flow in edge	-		
E_{DFO}	Data flow out edge	-		
E_{CD}	Condition edge	-		
E_{Arg}	Argument edge	-	v_{Arg}	v_{CL}

The summary of G 's elements is shown in Table 1

Graph construction. The graph is constructed by parsing the horn clauses line by line. The *Head* contains a control location and arguments. Argument nodes (represented as circles) and the control location nodes (represented as rectangular) are connected by the *Argumented edges* (dotted edges). *Body* has same structure with *Head*. If *Head* and *Body* have the same control location, then they share the control location and arguments.

The *Constraint* contains the constraints to make the transition from *Body* to *Head* satisfiable. We can parse it to two information, data flow AST and constraint AST. Data flow AST takes *Body'* arguments, constant value, or free variables in *constraint* as inputs, and go through some arithmetic operator to the root as output (a value) eventually to an *Head'* argument. The constraint AST takes *Body'* arguments, free variables in *constraint* as the inputs. The root of the constraint AST tree is a boolean operator which can output a boolean value to all guarded control and data flow hyperedges from *Body* to *Head*. If there is no *constraint*, the boolean value inputs to the hyperedges are set to true.

We can represent the transition by the definition of graph:

$Head(CL_{Main_m}, Arg_{head}) : \neg Body(CL_{Main_n}, Arg_{body}), [Constraint(Arg_{head}, Arg_{body}, V_C, V_{Op}, V_{FV})]$, where m and n are positive integer numbers. They can be equal or not equal. Arg_{head} and Arg_{body} are the set of arguments in head and body respectively. If $m == n$, $Arg_{head} == Arg_{body}$.

To associate *Head* and *Body*, we represent control flow information first by connecting the control location from *Body* to *Head*, and between the control locations, there is a guarded control flow hyperedge which takes *Body'* control location and the boolean value from AST tree constructed from *constraint* as inputs and output the control location information to *Head's* control location. Then, we build data flow between *Head* and *Body*. Every argument in *Head* is guarded by a guarded data flow hyper edge which takes *Body'* arguments, data flow AST tree root, or free variables and the boolean value from *constraint's* AST tree as inputs. In summary, all the control and data flows from *Body* to *Head* are guarded by a hyperedge, and one of the inputs of the hyperedges is the boolean value from the root of constraint AST tree constructed by the predicates in *constraint*.

4.1 Templates Representation

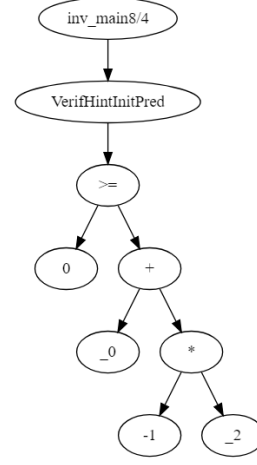
Each template contain location, category, and predicate information. One example template is "inv_main8:VerifHintInitPred(((0 + -1 * _2) >= 0))" in which "inv_main8" is its control location, "VerifHintInitPred" is its category (The meaning of the categories can be found in appendix), and "(((0 + -1 * _2) >= 0))" is the predicate. $_0$ and $_2$ are canonical encoding of variable names in source code. For example, $_0$ is x and $_2$ is y . We can combine the three components into one graph. The root node contains the location information ("inv_main8") as its node attribute. The only child of the root contains the category information ("VerifHintInitPred"). The predicate ("(((0 + -1 * _2) >= 0)))" can be represent as a binary tree. We connect the binary tree's root to the category node to connect all parts together. One example is shown in Figure 2b. We use graphviz to manipulate the graph. The corresponding graphviz text representation of the template is shown in 2a

```

0  digraph dag {
1  0 [label="inv_main8/4 "];
2  1 [label="
   VerifHintInitPred "];
3  2 [label=">="];
4  3 [label="0 "];
5  4 [label="+ "];
6  5 [label="_0 "];
7  6 [label="* "];
8  7 [label="-1 "];
9  8 [label="_2 "];
10 0->1
11 1->2
12 2->4
13 2->3
14 4->6
15 4->5
16 6->8
17 6->7

```

(a) Template graph example in graphviz format



(b) Template graph example

Figure 2: Graph representation for an example template "inv_main8:VerifHintInitPred(((_0 + -1 * _2) >= 0))"

5 Data Collection

For a single instance of data extracting from manual[1] or simple heuristics[2], the inputs are a list of templates T_0, T_1, \dots, T_n and a program P . The output is a list of templates with 0 or 1 labels $T_0^l, T_1^l, \dots, T_n^l, l = 0$ or 1. A concrete example for an output is $T_0^1 = ("inv_main8 : VerifHintInitPred((_0 + -1 * _2) >= 0))", 1)$

Algorithm 1: Templates extracting process

Result: $T_0^l, T_1^l, \dots, T_n^l$
 initialization: $\text{CurrentTemplateList} = \{T_0, T_1, \dots, T_n\};$
 $\text{Solvability} = \text{CEGAR}(\text{CurrentTemplateList}, \text{HornClauses});$
if $\text{Solvability} == \text{True}$ **then**
 while $\text{CurrentTemplateList}$ is not empty **do**
 $\text{CurrentTemplateList} = \text{CurrentTemplateList} - \{T_k\} (0 \leq k \leq n);$
 $\text{Solvability} = \text{CEGAR}(\text{CurrentTemplateList}, \text{HornClauses});$
 if $\text{Solvability} == \text{True}$ **then**
 $\text{RedunantTemplateList} = \text{RedunantTemplateList} \cup T_k^0;$
 else
 $\text{CriticalTemplateList} = \text{CriticalTemplateList} \cup T_k^1;$
 end
 $\text{TemplateList} = \text{RedunantTemplateList} \cup \text{CriticalTemplateList};$
 end
else
 Cannot solve within timeout, no template extracted;
end

Algorithm 2: Predicates extracting process

Result: $T_0^l, T_1^l, \dots, T_n^l$
 initialization: $\text{CurrentTemplateList} = \{T_0, T_1, \dots, T_n\};$
 $\text{Solvability}, \text{CurrentPredicateList} = \text{CEGAR}(\text{CurrentTemplateList}, \text{HornClauses});$
if $\text{Solvability} == \text{True}$ **then**
 while $\text{CurrentPredicateList}$ is not empty **do**
 $\text{CurrentPredicateList} = \text{CurrentPredicateList} - \{P_k\} (0 \leq k \leq n);$
 $\text{CurrentTemplateList} = \text{transform}(\text{CurrentPredicateList})$
 $\text{Solvability} = \text{CEGAR}(\text{CurrentTemplateList}, \text{HornClauses});$
 if $\text{Solvability} == \text{True}$ **then**
 $\text{RedunantPredicateList} = \text{RedunantPredicateList} \cup P_k^0;$
 else
 $\text{CriticalPredicateList} = \text{CriticalPredicateList} \cup P_k^1;$
 end
 end
 $\text{TemplateList} = \text{transform}(\text{RedunantPredicateList} \cup \text{CriticalPredicateList});$
else
 Cannot solve within timeout, no template extracted;
end

The final goal is to have proper predicates that can represent the abstract transition system. The templates are heuristics to generate the predicates in each CEGAR iteration. There are additional predicates added to solve the program in every CEGAR iteration but the templates are given and fixed before the CEGAR iteration.

The strategy *A* (fixed time out) to extract the training data (a list of templates with labels) is that for one program, we run Eldarica with an abstraction heuristic (e.g. use option -abstract:manual for .c files and -abstract for .smt2 files). If Eldarica can solve the program within 60 seconds. We mark this program as solvable. In Eldarica, we have an initial list of templates. We delete these templates one by one to see if the program can be solved with the remaining templates. If the program is still solvable within the timeout, then that deleted template is critical, we mark it as useful and label it as 1. By doing so iteratively, we can label the initial list of templates. This labelled list of templates will be used in the training process.

1. chc-comp benchmarks: 38/1216 .smt2 files (programs) need templates to solve the program within in 60 seconds.
2. sv-comp smt benchmarks: 7/6814 .smt2 files need templates.
3. sv-comp c benchmarks: 31/545 .c files need templates.

Only 76 training programs in total.

A alternative strategy *B* (variable timeout): First, confirm the solvability (i.e. the program can be solved within 60 seconds with the abstraction heuristic). Second, record the solving time with and without the abstraction heuristic. If Eldarica takes less solving time with abstraction heuristic, pass this solving time as timeout to Eldarica to label the templates used in the program.

1. chc-comp benchmarks: 40/1216. 32 programs (204 templates)for training and 8 (47 templates) for testing. 8/8 solved by read templates. Read templates time consumption/original templates consumption = 40.25/40.86 (in seconds)
2. sv-comp smt benchmarks: 15/5911. 12 programs (254 templates)for training and 3 (74 templates) for testing. 3/3 solved by read templates. Read templates time consumption/original templates consumption = 28.07/27.28 (in seconds)
3. sv-comp c benchmarks: 41/555. 32 programs (4635 templates)for training and 9 (350 templates) for testing. 8/9 solved by read templates. Read templates time consumption/original templates consumption = 34.04/53.46 (in seconds)

96 training programs. 76 programs for training, 20 programs for testing. 18/20 solved by read templates. Read templates time consumption/original templates consumption = 140.67/126.84 (in seconds)

Strategy *C*: keep using Strategy *B*'s on larger benchmarks, and use these three benchmarks only for testing.

We try to use balanced and imbalanced data set to train the neural network.

6 Training Model

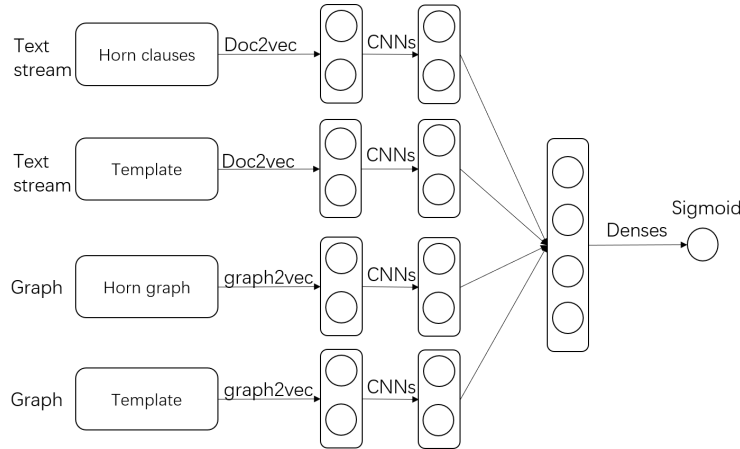


Figure 3: Neural network structure

6.1 Program Embedding

6.1.1 Text Level

Doc2Vec [3] embed different length of sentences to fixed vector. Program text streams are embedded into 100 dimensions. Template text streams are embedded to 20 dimensions.

Table 2: Benchmarks

Benchmarks	File type	Total file	Solved	sat	unsat	Unsolved
chc-comp-smt	.smt2	1216	421	339	82	795
sv-comp-smt	.smt2	5911	4730	4516	214	1181
sv-comp-c	.c	555	323	262	61	231
In total	-	-	-	-	-	-

6.1.2 Graph Level

Graph2Vec[4]. WL relabeling process [5] to generate random subgraphs (can be seen as sentences). Then use Doc2Vec embed subgraphs to fixed vector. Program graphs (horn graphs) are embedded into 100 dimensions. Templates are embedded to 20 dimensions.

7 Experiments

The predicted result is a score for a template. We add two rank modes to decide which templates will be used for solving a particular program. For example, we have 10 templates, their scores ranged from 0 to 1. The first rank mode can specify a threshold for the scores. If the threshold is 0.4, all templates that have scores larger than 0.4 will be used to solve the program, and the left templates are discarded. The second rank mode can specify a hierarchical number (N) which decides to use top N ranked templates by score. If we specify the hierarchical number (N) be 5, then the top 5 ranked templates by score will be used to solve the program, and the left 5 templates will be discarded.

7.1 Ablation studies

Table 3: Imbalanced Predicates with control flow and template graph

Benchmarks	Training program	Testing program	Training predicates	Testing predicates	Solved programs	Total time consumption (predicted:abstract)
chc-comp-smt	130	33	969	225	33/33	63.92 : 63.72 (s)
sv-comp-smt	784	196	5907	1406	196/196	514.60 : 511.186
sv-comp-c	116	29	4148	847	20/29	578.33 : 56.74
In total	1030	258	10588	2914	250/258	1119.73 : 629.82

Table 4: Imbalanced Templates with control flow and template graph

Benchmarks	Training program	Testing program	Training predicates	Testing predicates	Solved programs	Time consumption (predicted/abstract)
chc-comp-smt	-	-	-	-	-	- (s)
sv-comp-smt	-	-	-	-	-	-
sv-comp-c	-	-	-	-	-	-
In total	-	-	-	-	-	-

CEGAR iterations:

Solvability:

Benchmarks	Total file	with tem- plate graph	with horn graph	with control flow graph	with control flow and tem- plate graphs	with horn and tem- plate graphs
chc-comp-smt	1216	-	-			
sv-comp-smt	5911	-	-			
sv-comp-c	555	-	-			

Time consumption:

Benchmarks	Total file	with tem- plate graph	with horn graph	with control flow graph	with control flow and tem- plate graphs	with horn and tem- plate graphs
chc-comp-smt	1216	-	-			
sv-comp-smt	5911	-	-			
sv-comp-c	555	-	-			

8 Related work

Text level learning for improving ATP (formal verification?) [6]

Guiding formal method’s search process.

Program graph representation.

end-to-end graph embedding. Message passing, convolutional graph neural network. Define a the first state of nodes. Update node representations by edges type and the connected node. Iteration is intuitively the filters of CNNs. The edges, hyper-edges, messages should capture the characteristics or relations between nodes. To deal with arbitrary length of information (different number of nodes), aggregation or abstraction then aggregation.

AST embedding: code2vec [7]. TBCNN [8].

FormulaNet [9]. GAT [10]. Gated Graph Neural Networks (GGNN) [11]

fake task graph embedding.

9 Discussion and Future work

References

- [1] 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.
- [2] Jérôme Leroux, Philipp Rümmer, and Pavle Subotić. Guiding craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, Jun 2016.
- [3] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.
- [4] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.
- [5] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12, 2011.

- [6] 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.
- [7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.