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

2 Background

—

2.1 Abstraction Based Model Checking

—

2.2 CEGAR

—

2.3 Abstract Interpolation

—

3 Problem Overview

—

*All contributions are considered equal.

4 Data Collection

For a single instance of data extracting, the inputs are a list of templates $T_0, T_1, ..., T_n$ and a program P. The output is a list of templates with 0 or 1 labels $T_0^l, T_1^l, ..., T_n^l, l = 0$ or 1. A concrete example for an output is $T_0^1 = (\text{"}inv\_main8 : VerifHintInitPred(((\_0 + -1 * \_2) >= 0))\text{"}, 1)$

Algorithm 1: Data extracting process

Result: $T_0^l, T_1^l, ..., T_n^l$
initialization: CurrentTemplateList = $\{T_0, T_1, ..., T_n\}$;
Solvalbility=CEGAR(CurrentTemplateList,HornClauses);
if *Solvalbility == True* **then**
  while *CurrentTemplateList is not empty* **do**
    CurrentTemplateList = CurrentTemplateList $-\{T_k\}(0 \leqslant k \leqslant n)$;
    Solvalbility=CEGAR(CurrentTemplateList,HornClauses);
    if *Solvalbility == True* **then**
      RedunantTemplatesList=RedunantTemplatesList $\cup\ T_k^0$;
    else
      CriticalTemplatesList=CriticalTemplatesList $\cup\ T_k^1$;
    end
  end
else
  No need for templates;
end

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.

Benchmarks	File type	Total file	Solved with abstract (abstract:manual)	Unsolved with abstract (abstract:manual)	Training program (templates)	Testing program (templates)	Solved by read templates
chc-comp	.smt2	1216	-	-	-	-	-
sv-comp smt	.sm2	5911	-	-	-	-	-
sv-comp C	.c	555	-	-	-	-	-
In total	.c,.smt2	-	-	-	-	-	-

5 Training Model

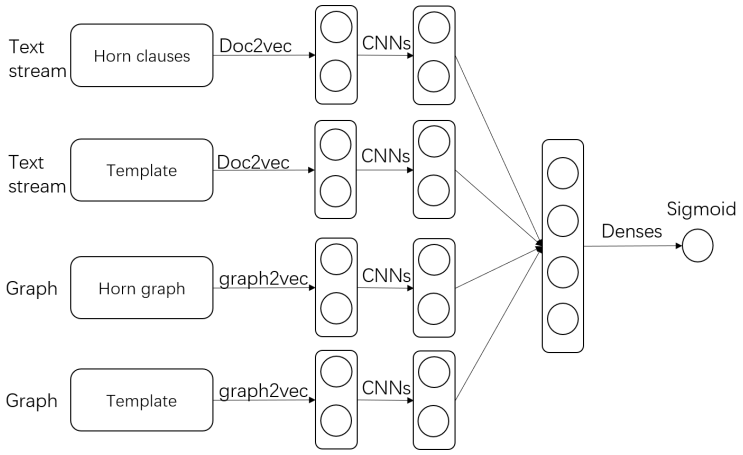


Figure 1: Neural network structure

5.1 Program Embedding

5.1.1 Text Level

Doc2Vec [1] embed different length of sentences to fixed vector. Program text streams are embedded into 100 dimensions. Template text streams are embedded to 20 dimensions.

5.1.2 Graph Level

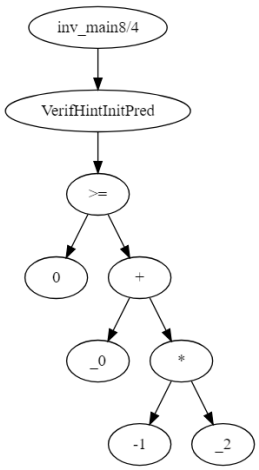
Graph2Vec[2]. WL relabeling process [3] to generate random subgraphs (can be seen as sentences). Then use Doc2Vec embed subgraphs to fixed vector. Program graphs (horn graphs) are embedded into 100 dimensions. Templates are embedded to 20 dimensions.

5.1.3 Graph 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

```
digraph dag {
0 [label="inv_main8/4"];
1 [label="
   VerifHintInitPred"];
2 [label=">="];
3 [label="0"];
4 [label="+"];
5 [label="_0"];
6 [label="*"];
7 [label="-1"];
8 [label="_2"];
0->1
1->2
2->4
2->3
4->6
4->5
6->8
6->7
}
```

(a) Template graph example in graphviz format

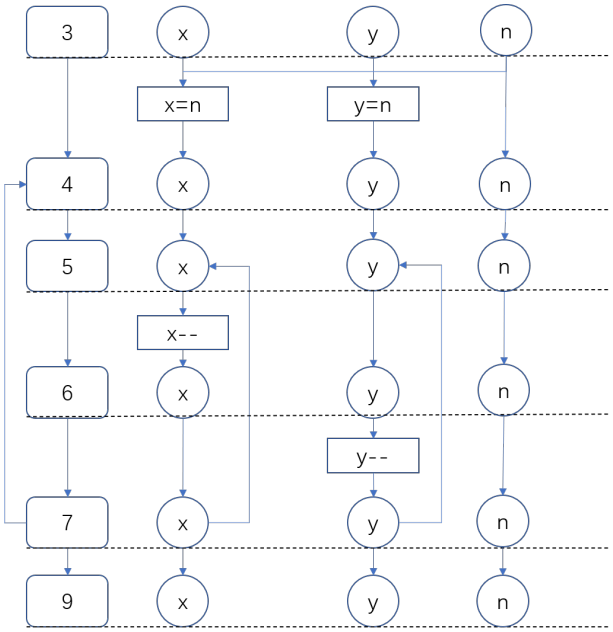


(b) Template graph example

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

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

(a) C program for horn graph example



(b) Horn graph example

Figure 3

6 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.

Solvability:

Benchmarks	File type	Total file	with template graph	with horn graph	with control flow graph	with control flow and template graphs	with horn and template graphs
chc-comp	.smt2	1216	-	-			
sv-comp smt	.sm2	5911	-	-			
sv-comp C	.c	555	-	-			

Time consumption:

Benchmarks	File type	Total file	with template graph	with horn graph	with control flow graph	with control flow and template graphs	with horn and template graphs
chc-comp	.smt2	1216	-	-			
sv-comp smt	.sm2	5911	-	-			
sv-comp C	.c	555	-	-			

6.1 Work flow

7 Related work

Program synthesis:

Logical reasoning: NEO, BLAZE, CEGIS.

Machine learning methods: Program synthesis from examples: Encode inputs to fixed vector, then decode this vector to target language.

Program induction: Directly give output. The learned program is induced latently within the weights and activations of a neural network.

Guide formal method's search process by Training a statistical model to rank some options existed in search space and try them first.

8 Discussion and Future work

References

[1] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

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

[3] 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.