



Data-driven Optimization of Inductive Generalization

Nham Le 
 University of Waterloo
 nv3le@uwaterloo.ca

Xujie Si 
 McGill University
 CIFAR AI Chair, Mila
 xsi@cs.mcgill.ca

Arie Gurfinkel 
 University of Waterloo
 arie.gurfinkel@uwaterloo.ca

Abstract—Inductive generalization (IG) is the key to the efficiency of modern Symbolic Model Checkers (SMCs). In this paper, we introduce a *data-driven* method for inductive generalization, whose performance can be automatically improved through historical runs over similar instances. Our method is inspired by recent advances for the part-of-speech (PoS) tagging problem in natural language processing (NLP). Specifically, we use a hierarchical recurrent neural network augmented with syntactic and semantic information to predict essential parts of a proof obligation that could be generalized, instead of checking each part one by one. We develop a prototype called ROPEY by incorporating our method into SPACER – a state-of-the-art SMC, and perform evaluations on the KIND2’s simulation benchmarks. ROPEY is evaluated in two settings: *online learning* – for a given instance, we run SPACER for a number of iterations and collect its trace on which ROPEY is trained, and then use ROPEY to guide SPACER to finish the remaining solving process; and *transfer learning* – ROPEY is trained over historical runs of SPACER in advance, and for future instances, ROPEY is used directly to guide SPACER from the very beginning. For non-trivial benchmarks, ROPEY perfectly answers 72% and 77% of the queries in the online and transfer learning settings, respectively. While the speed improvement is not the focus of the paper, our preliminary results are promising: for non-trivial instances, ROPEY’s end-to-end running time is 25% faster.

I. INTRODUCTION

Model checking has been widely used in various important areas such as robustness analysis of deep neural networks [27], verification of hardware designs [16], software verification [3], analysis [20] and testing [41], parameter synthesis in biology [5], and many others. The central challenge of model checking is to find a concise and sound approximation of all possible states a given system may reach, which does not cover any undesired states (i.e. violating given specifications). Tremendous progress has been made by innovations in efficient data representations [10], scalable SAT solvers [43], [35], [18], and effective heuristics [14], [13], [32]. Modern model checkers share a common basis, namely, IC3 [7], of which the key insight is *inductive generalization (IG)*. This idea has been generalized to support rich theories [26] that are crucial for many verification tasks [30], [22] beyond hardware verification. The generalized IC3 with rich theories, also known as satisfiability checking for Constrained Horn

Clauses modulo Theory (CHC) [6], becomes the core part of a broad range of verification tasks.

Existing IG techniques follow either an enumerative search process [7], [8] or ad-hoc heuristics [21], [31]. These heuristics are effective but demand non-trivial domain-specific (or even problem-specific) expertise. In this work, we aim to learn such heuristics automatically from the past successful IGs. We observe that verification problems as well as associated IGs are not isolated from each other. Taking software verification as an example, verifying different properties of the same program involves similar or same IGs; different versions of programs have a similar code base; and different software may use the same conventions, idioms, libraries and frameworks, resulting in similar structures.

Our approach is inspired by recent advances in deep learning, especially in NLP where non-trivial semantic correlations between words are learned automatically using Neural Networks (NNs) [33]. However, IG raises many new challenges for deep learning. First, the input and the output of IG are symbolic expressions, which are *highly structured* with *rich semantics*. Slight syntactic variations can lead to dramatic changes in semantics. Second, more importantly, given that neural networks hardly provide any reliable guarantees, how to design a data-driven system based on deep neural networks, which exhibits *learnability* from past experiences but still preserves *soundness*? All these challenges have to be properly addressed in building a data-driven reasoning framework. In this work, we share our design choices and empirical findings in building a data-driven inductive generalization engine ROPEY, which introduces a neural component into SMC. Specifically, we make the following contributions:

- we adapt standard deep learning models to effectively represent symbolic expressions by incorporating both syntactic and semantic information;
- we design a simple but effective learning objective so that training data can be collected with nearly no changes of existing model checkers;
- our integration algorithm achieves soundness by design, and in the worst case, the learning component may only hurt the running time performance;
- we implement ROPEY on top of SPACER, a state-of-the-art CHC-solver. Our empirical evaluations indicate that ROPEY can effectively predict perfect answers for IG

This work was supported, in part, by an Individual Discovery Grant from the Natural Sciences and Engineering Research Council of Canada, and the Canada CIFAR AI Chair Program.

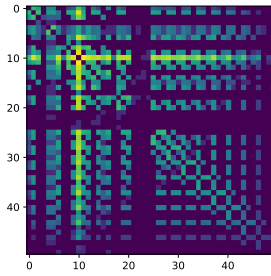


Fig. 1: Literal co-occurrences in solving PRODUCER_CONSUMER_luke_2_e7_1068_e8_1019.

queries, and this predictive power directly translates to an improvement in end-to-end running time.

The utility of our current solution is modest since its applications are restricted to two use-cases: verification of *multiple* properties of a *single* system (transfer learning), and guiding verification of a hard property using its partial run (online learning). This, however, is already useful in the context of multi-property verification that is common both in hardware and software verification domain [12]. More importantly, we demonstrate that NN-based heuristics can be effective in IC3-style algorithms. We believe this will lead to many further improvements, including heuristics that will eventually transfer between systems.

The rest of the paper is structured as follows. Sec. II shows a motivating example. Sec. III gives an overview of our approach. Sec. IV describes two novel embedding methods for converting symbolic expressions into numerical vectors. Sec. V formalizes the learning problem and describes our neural network architecture. Sec. VI presents our empirical evaluation and ablation study. Finally, Sec. VII discusses closely related work, and Sec. VIII concludes the paper.

II. A MOTIVATING EXAMPLE

In this section, we motivate our approach by illustrating the solving process of a particular CHC problem – the variant e7_1068_e8_1019 of the problem PRODUCER_CONSUMER_luke_2 from KIND2 [11] benchmarks. We identify a bottle neck in IG, observe a pattern in the solving process, and explain how this leads to our intuition. While we use a specific instance for illustration, the results generalize to others in our benchmarks. We assume familiarity with SMC [15] and inductive generalization of IC3 [7]. These are also summarized in Sec. III.

SPACER cannot solve this variant in less than 930s. SPACER proves that the instance is safe up to depth 29 in 883s, in which 545s (61%) is spent on IG – so this is the bottleneck.

During inductive generalization process, SPACER takes a candidate lemma L , and uses an SMT solver to check whether each literal of L can be dropped. Each call to the SMT solver is potentially very costly. Thus, it is desirable to drop or skip multiple literals together.

We conjecture that there is a pattern between literals: some groups of literals may always be dropped or kept together. If this correlation is known, it can be used to speed up IG.

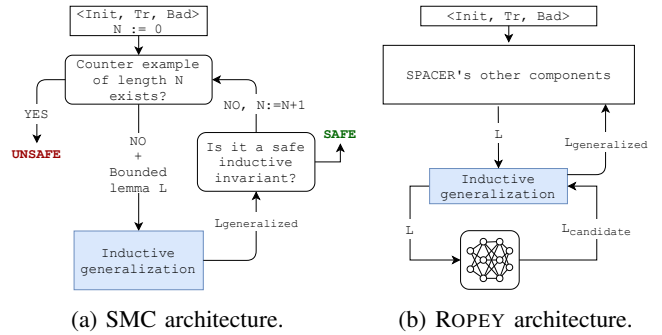


Fig. 2: Overview of Symbolic Model Checking and ROPEY.

To verify our hypothesis, in Fig. 1 we visualize the co-occurrences of kept literals in the instance. Literals are ordered by the time they are learned. Each cell X_{ij} in the grid is the number of times the literals ℓ_i and ℓ_j appear together in some generalized lemma (normalized by the largest value). In the figure, brighter cells indicate larger values.

The figure shows a strong geometric pattern, with literals clustered into unusual groups. However, we are not able to tell the exact heuristics describing those patterns. In this paper, we turn this observation into a practical inductive generalization method with the help of data-driven approach.

III. OVERVIEW

In this section, we give an overview of our technique, outline the challenges involved, and our key insights to address them. The context is symbolic SMT-based Model Checking (SMC) [7], [26], [29], also known as satisfiability checking for Constrained Horn Clauses modulo Theory (CHC) [6]. In Model Checking, the high-level goal is to show that an infinite state transition system (Tr) does not have an execution/path that reaches a set of bad states (Bad) by finding a formula Inv that is an inductive invariant of Tr and does not intersect with Bad . The goal of CHC solving is to show that a set of First Order Logic formulas Φ that satisfy the Horn restriction [6] is satisfiable by exhibiting a symbolic formula $Model$ that defines an FOL model that satisfies Φ . The two problems are closely related. Model Checking is often reduced to CHC solving. Both problems are in general undecidable.

Fig. 2a shows the basic structure of an SMC algorithm based on IC3 architecture. In the paper, we use SMC SPACER [29], but the architecture is common to many engines. SMC iteratively unrolls Tr , uses an SMT solver to find a bounded counterexample (which is usually decidable), and, if no counterexample is found, attempts to create an inductive invariant. The invariant is constructed as a set of so called *lemmas*, where each lemma blocks a predecessor of Bad (a *proof obligation*), and is a disjunction of atomic formulas. An example lemma is $x \leq 0 \vee y$, which is often written as a set for convenience, i.e. $\{x \leq 0, y\}$. Many of the details of the algorithm are not important, and we omit them here. The step we focus on in this paper is *inductive generalization* (IG) (highlighted in blue in Fig. 2a), that is responsible for generalizing learned lemmas. In practice, IG is crucial for the performance of SMC.

Input: the original F-inductive lemma $L = \{\ell_1, \ell_2, \dots, \ell_n\}$
Output: a generalized F-inductive lemma $K \subseteq L$

```

1  $K \leftarrow \emptyset$  // kept literals
2  $C \leftarrow L$  // literals to check
3 while  $C \neq \emptyset$  do
4    $K, C \leftarrow \text{dropOne}(K, C)$ 
5 return  $K$ 
6 function  $\text{dropOne}(K, C)$ 
7    $lit \leftarrow \text{pick}(C)$ 
8   if  $\text{isInductive}(K \cup C \setminus \{lit\})$  then
9      $C \leftarrow C \setminus \{lit\}$ 
10  else
11     $K \leftarrow K \cup \{lit\}$ 
12     $C \leftarrow C \setminus \{lit\}$ 
13  return  $K, C$ 

```

Fig. 3: ITERDROP algorithm.

Conceptually, inductive generalization is a simple process, usually done with an algorithm similar to the one we call ITERDROP¹, shown in Fig. 3. ITERDROP starts with a valid lemma $L = \{\ell_1, \dots, \ell_n\}$, and proceeds to generalize L by removing an arbitrary chosen literal from L , and using an SMT solver to check whether the lemma is still valid (by calling `isInductive`). The details of `isInductive` are not important – but it can be quite expensive. If the call succeeds, the literal is removed, otherwise it is kept. The goal is to generalize to a valid lemma with a minimal number of literals. From now on, when the context is clear, we use *generalization* instead of inductive generalization.

We illustrate ITERDROP with a sample run, shown in Fig. 4a. Start from the given lemma $L = \{x_3, x_1, x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$, ITERDROP proceeds as follows:

- 1) it tries to drop the first literal, x_3 , by checking whether $L'_1 = \{x_1, x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is valid;
- 2) assume that L'_1 is valid, then $L \leftarrow L'_1$, x_1 is chosen next;
- 3) now, assume that $L'_2 = \{x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is not valid. L remains as is and $x_6 = 1$ is chosen next;
- 4) assume that $L'_3 = \{x_1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is valid, then $L \leftarrow L'_3$, and $x_9 - x_{10} \geq 41$ is chosen next;
- 5) assume that $L'_4 = \{x_1, x_5 = 1\}$ is not valid, then L is unchanged, and $x_5 = 1$ is chosen next;
- 6) assume that $L'_5 = \{x_1, x_9 - x_{10} \geq 41\}$ is valid, then L'_5 is the final generalized lemma.

The example highlights the difficulty of inductive generalization. First, each call to `isInductive` is potentially very expensive. Thus, reducing the number of the calls is highly desirable. Second, many of the calls, like steps 3 and 5 are “useless” – no new lemma is learned from them. Thus, reducing such “useless” calls is also highly desirable. Finally, a solver makes many (up to thousands) such inductive generalization calls per run.

Our *key insight* is that since generalization happens frequently, and, while the lemmas are different, the literals are similar, *it is possible to learn the co-occurrence between*

¹While there are more advanced IG techniques, such as [23], we choose ITERDROP since it is used in SPACER – a state-of-the-art CHC solver.

literals that do and do not occur in the same lemma together. This co-occurrence, if learned, could then be used to improve inductive generalization!

Crucially, SPACER learns new literals all the time, and literals between different instances of the same problem are often similar, for instance, $x_1 - 2x_3 \geq 20$ and $x_1 - 2x_3 \geq 25$. Thus, an ML-based solution is useful to transfer knowledge between different sets of literals. Our method is inspired by the PoS-tagging problem in NLP, in which NNs automatically learn co-occurrence patterns between words and their tags. We elaborate more on this inspiration in Sec. V. We have also tried creating our own hand-crafted heuristics for directly calculating co-occurrence (for example, by using Boolean abstraction of literals), but none worked well in practice.

Concretely, we propose a novel neural network architecture, denoted by M , that learns from past IG queries, and is then used to predict answers for new IG queries. As shown in Fig. 4c, M outputs a binary mask (a list of zeros and ones) corresponding to literals that should be dropped or kept in the lemma. To evaluate M in the context of an SMC, we devise a new neural-based IG algorithm called XDROP, that has M at its core (Fig. 6). We have developed ROPEY, a prototype SMC that uses XDROP to guide SPACER. (Fig. 2b).

In Fig. 4b, we illustrate a run of XDROP on our example: (1) it runs M on the input L ; (2) it creates a mask $\{0, 1, 0, 1, 0\}$, corresponding to a candidate $L_{cand} = \{x_1, x_9 - x_{10} \geq 41\}$; (3) it checks the inductiveness of L_{cand} ; (4) it accepts L_{cand} , and runs ITERDROP starting from L_{cand} . Note that XDROP runs only 3 inductiveness checks, compared to 5 used by ITERDROP.

Challenges. To make ROPEY a practical verification engine, we have to address challenges in both the machine learning and the logical soundness aspect. For machine learning, the challenge is in representing symbolic expressions as vectors, while still maintaining their rich semantic structure. For logical soundness, the challenge is in setting up the learning objective and using the neural net in a way that guarantees the soundness of a verification engine.

Representation learning of symbolic formulas. Literals are symbolic formulas, which are structured and have meaning sensitive to small changes. Simply viewing a literal as a sequence of tokens fails to capture the subtle semantic differences between structurally similar formulas.

We incorporate both syntactic and semantic information of a literal into its representation. Our approach views a literal as a directed acyclic graph (DAG), which is post-processed from its abstract syntax tree (AST), and then adapts TREELSTM [44] to embed such a DAG structure. Our approach also takes semantic information into consideration so that specific properties of values are respected: embedding of numbers and variables should preserve their relative order and equality.

Learning for inductive generalization. Directly using ML to address the generalization problem is a non-trivial structure prediction problem. It takes in a set of symbolic formulas and outputs another set of symbolic formulas that are more general and more concise. Rather than having an

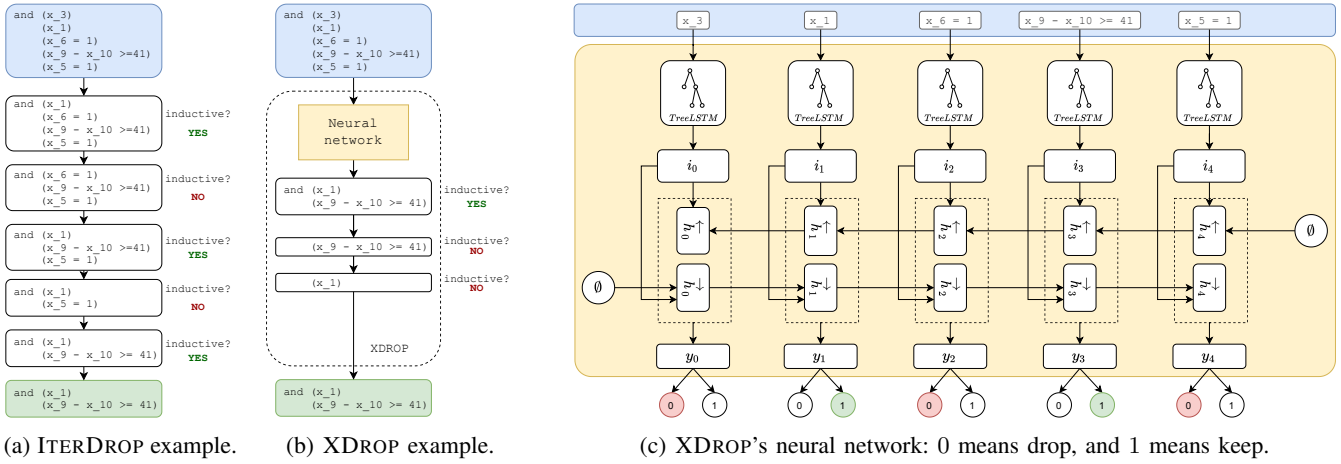


Fig. 4: Examples of how ITERDROP and XDROP do inductive generalization on the same query.

end-to-end ML solution, we embed a learning component in a classic symbolic approach of generalization. Specifically, the learning component captures the co-occurrence between literals appearing in past runs and predicts the likelihood of keeping or dropping a literal in the current run. Furthermore, uncertainties introduced by the learning component have to be carefully controlled, which otherwise could lead to unsound conclusion. ROPEY is designed to make sound progress no matter what predictions the learning component provides. Bad predictions may be harmful to the performance, but not to soundness!

IV. REPRESENTATION LEARNING

Machine learning frameworks [36] and algorithms [44], [38] operate over fixed-length numerical vectors. One challenge for applying machine learning for IG is converting discrete structures with rich semantic meanings into such numerical representations. In this section, we describe how we embed the basic unit of our inputs – symbolic formulas – into fixed-length vectors, while still maintaining their syntactic and semantic meaning to a certain extent.

A. Representing and normalizing symbolic formulas

Abstract Syntax Trees (ASTs) are natural representations of formulas that are traditionally used in parsing and compilers. They preserve the key structure of the formula, while hiding (or abstracting) unnecessary details such as white space, commas and parentheses. Alternative representations such as sequences of tokens abstract too much of the structure of the formula, while highlighting unnecessary differences. Thus, we represent logical formulas using their ASTs: operators label nodes of the tree, operands are children, constants (boolean and numeric) and variables are leaves. An example of an AST is shown in Fig. 5b.

Ideally, we would like to represent semantically equivalent formulas with the same AST. However, this is not guaranteed if one naively parses a formula into an AST. For example, $x + 0 > y$ and $x > y$ are semantically equivalent, yet differ in the concrete syntax, and have different ASTs. To address this, we rewrite each formula in a “normal” form by simplifying as

well as ordering commutative operators. Specifically, we use a simplification engine of Z3 [17]. Our normalizer cannot handle sophisticated semantic equivalences, such as normalizing $2/7 \cdot x_9 - 4/7 \cdot x_{10} \geq 6$ into $1/7 \cdot x_9 - 2/7 \cdot x_{10} \geq 3$. Improving the normalization process to handle such cases would be an interesting future work.

Note that semantically equivalent rewriting and normalization make our representations of symbolic formulas essentially *directed acyclic graphs (DAGs) modulo semantic equivalence*, because semantically equivalent subtrees share the exact same embedding. Indeed, representations of symbolic formulas in our implementation are DAGs, although they are viewed as if they were trees by the embedding model. Without further notice, when we refer to a node in a tree, we actually mean its corresponding node in the DAG.

We use TREELSTM [44] to embed a symbolic formula, or more concretely its AST representation, into a fixed-length vector. TREELSTM is essentially a recursive process, where the embedding of a (sub-)tree is an aggregation of the embedding of the root node and embeddings of its subtrees. The basic requirement of using TREELSTM is to have an embedding for each node. In the rest of this section, we describe the features used to embed each AST node into a fixed-length vector.

B. Embedding features of an AST node

A common technique to map a node N to a vector is to first map the infinite (or simply large) set Σ of all possible nodes into a finite set T of tokens (a.k.a. *encoding*), and then *embed* each token into a vector using an embedding matrix of size $|T| \times d_{\text{emb}}$.

a) *Encoding*: Under the standard encoding scheme, many nodes have to be mapped into the same token. For example, in NLP, all out-of-vocabulary words are mapped into a token $\langle \text{UNK} \rangle$. Similarly, variable names, and numerical constants over an expression can be mapped into two tokens: $\langle \text{VAR} \rangle$ and $\langle \text{NUM} \rangle$, respectively.

Unfortunately, this encoding scheme is inadequate in our setting. We believe that both the variable names and the values

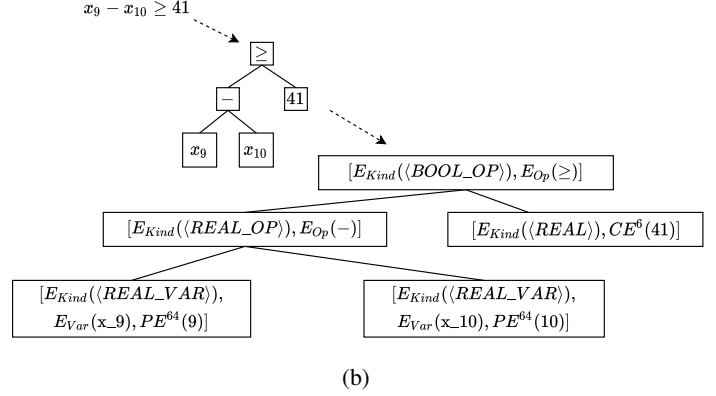
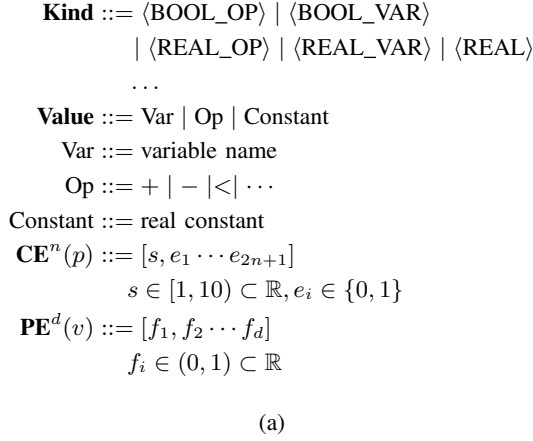


Fig. 5: (a) The grammar for AST node features, and (b) an example AST and its semantic features.

of the numeric constants are highly relevant for successful generalizations! For example, consider two pairs of formulas:

$$\begin{aligned}
 x_1 - 2x_3 + 7x_5 &\geq 10 & x_1 - 2x_3 + 7x_5 &\geq 14 & (1) \\
 x_1 - 2x_3 + 7x_5 &\geq 10 & x_1 + x_3 - x_5 &\geq 0 & (2)
 \end{aligned}$$

Pair (1) represents two parallel hyperplanes, with the first subsuming the second. Pair (2) represents two intersecting hyperplanes and cannot be simplified any further. The difference between the two pairs disappears when all numeric constants are mapped to a small finite set of tokens. Yet, this difference is crucial for successful learning in our context!

Instead of abstracting variables (or constants) into a single token, we propose a finer granularity abstraction as follows. Each node is abstracted as a pair of $\langle \mathbf{Kind}, \mathbf{Value} \rangle$, whose grammar is shown in Fig. 5a. **Kind** captures the type (or sort) of the expression of an AST node. The encoding is one of the pre-defined symbols, such as $\langle \text{BOOL_OP} \rangle$ for a Boolean operator, etc. **Value** captures the content of an AST node. It could be a *Variable Name*, an *Operator*, or a *Constant*. Operators are encoded as their string representation. Constants are encoded as their string representations. Variable Names are encoded using the form x_i , where x is some fixed string, and i a numeric id of the variable.

Next, we describe how we embed the pair $\langle \mathbf{Kind}, \mathbf{Value} \rangle$ into a fixed-length vector.

b) Embedding: **Kind** is embedded into a fixed-length vector of length d_{Kind} using a standard embedding matrix [34] E_{Kind} of the size $|\mathbf{Kind}| \times d_{\text{Kind}}$. **Value** could be embedded in the same manner. However, given **Value** is quite diverse, we propose different embedding methods for different kinds of values. When **Value** is an Op, we introduce the second embedding matrix E_{Op} of the size $|\text{Op}| \times d_{\text{Op}}$.

When **Value** is a Variable Name, we combine two embedding methods. The first method, which we call *Naive Embedding*, is the same as above, in which we use another embedding matrix E_{Var} of the size $|\text{Var}| \times d_{\text{Var}}$. The second method, which we call *Positional Embedding*, based on the method introduced in [46]. It embeds the id t of the normalized variable name x_t as follows: The embedding of the position

t is a vector $\text{PE}^d(t)$ of length d . The value for the i^{th} entry in the vector $\text{PE}^d(t)$ is defined as follows:

$$\text{PE}^d(t)_i = \begin{cases} \sin(\omega_k \cdot t) & \text{if } i = 2k \\ \cos(\omega_k \cdot t) & \text{if } i = 2k + 1 \end{cases}$$

where $\omega_k = 10000^{-2k/d}$. This embedding satisfies many nice properties: each position is mapped to a unique value, all entries in the vector are between 0 and 1 (which makes learning easier), and, lastly, for every fixed offset k , there exists a transformation matrix $T \in \mathbb{R}^{d \times d}$ s.t. $T \cdot \text{PE}^d(t)_i = \text{PE}^d(t+k)_i$ holds for any position t and index i [46]. This last property allows the model to learn relative positions easily. In practice, we combine the two methods by concatenating their vectors.

When **Value** is a Constant, we want to embed it in a way that allows the network to quickly extract magnitudes of constants along with their values. We propose the following *Constant Embedding* method: Given a numerical value p , its embedding is a vector $\text{CE}^n(p)$ of length $2(n+1)$. To embed it, we first write p in its scientific notation: $p = s \times 10^e$. The entries in $\text{CE}^n(p)$ are then calculated as follows:

$$\begin{aligned}
 \text{CE}^n(p)_1 &= s \\
 \text{CE}^n(p)_{i \neq 1} &= \begin{cases} 1 & \text{if } i = 2 + n + e \\ 0 & \text{if } i \neq 2 + n + e \end{cases}
 \end{aligned}$$

Simply put, we embed the significant s as the first entry in the vector, and the rest is the one-hot encoding of e in the range $[-n, n]$. For example, with $n = 2$, $p = 42 = 4.2 \times 10^1$, its embedding is $\text{CE}^2(42) = [4.2 \ 0 \ 0 \ 1 \ 0]$. Similarly, $\text{CE}^3(0.42) = [4.2 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0]$.

The final feature vector for a node is then the concatenation of the embedding of **Kind** and **Value**. In our experiments, we set $d_{\text{Kind}} = d_{\text{Op}} = d_{\text{Var}} = d = 64$, and $n = 6$. We conclude this section with an example. Fig. 5b shows an AST for $x_9 - x_{10} \geq 41$ and its transformation into a tree of feature vectors, with $n = 6$ and $d = 64$.

V. LEARNING TO GENERALIZE

In this section, we elaborate on our insight first mentioned in Sec. III, then we describe the details of our model.

Word	Tag	Literal	Tag
Travelers	noun	x_3	drop
love	verb	x_1	keep
to	preposition	$x_6 = 1$	drop
park	verb	$x_9 - x_{10} \geq 41$	keep
here	adverb	$x_5 = 1$	drop

TABLE I: Two examples for PoS-tagging (left) and IG (right).

A. Lemma Labeling Problem

In Natural Language Processing, part-of-speech tagging (PoS-tagging) is the process of labeling each word in a text (corpus) a particular part of speech, based on *its definition and its context*. Table I (left) shows an example of tagging a sentence. To correctly tag each word, a tagger needs to know that “park” in this context is a verb, not a noun. State-of-the-art PoS-tagger tackles this problem purely from the probabilistic view [45]: in the dataset, how many times “park” is tagged as a NOUN, how many times “park” is tagged as a VERB given that the following word is tagged as an ADVERB, etc.

Our insight is that the inductive generalization could be viewed as a special case of PoS-tagging in which there are only two tags: drop and keep. Table I (right) shows one such example. We also view the problem in the same probabilistic way: in the dataset, how many times x_3 is kept, how many times x_3 is dropped given that x_1 is kept, etc. It is reasonable to expect there are shared patterns between different properties of the same system, or between different points in time of the same solving process. However, it is not expected that the learned pattern is transferable between different systems (x_3 in one system is completely different from x_3 in the others, just like “park” in English and Korean are completely different).

Formally, we define our problem as an instance of the *sequence labeling problems*:

Problem 1 (Lemma labeling problem) \mathcal{L} is the set of all possible literals. Given a list of literals L of length n and a vector \mathcal{M} of zeros and ones, $|\mathcal{M}| = n$, train a tagger $M : \mathcal{L}^n \mapsto \{0, 1\}^n$ s.t. $M(L) \approx \mathcal{M}$.

Note that in the problem definition we keep the lemma as a list instead of a set of literals. This means that given a different ordering from the same set of literals, we might end up with a different result. However, this is also the behavior of SPACER, because SPACER maintains the lemma as a list of literals, and *pick*(C) in Fig. 3 simply returns the first element in C .

B. Model

To handle inputs of different lengths, we use two variants of the Long Short-Term Memory (LSTM) [25] network. At the high level, the information (hidden state) at each timestep t in a vanilla LSTM is $\vec{h}_t = LSTM(i_t, \vec{h}_{t-1})$, where i_t is the input at timestep t , and a vector of zeros is used for the initial \vec{h}_0 . Intuitively, the formula says that the hidden state at timestep t captures information from every *prior* timestep.

The first variant, Bidirectional-LSTM [38], has been shown to improve the labeling performance in NLP tasks [47]. It extends LSTM by including information from *later* timesteps as

Input: the original F-inductive lemma $L = \{\ell_1, \ell_2, \dots, \ell_n\}$
Output: a generalized F-inductive lemma
1 $L_{Cand} \leftarrow \{\ell_i \mid \ell \in L, M(L)[i] = 1\}$
2 **if** isInductive(L_{Cand}) **then**
3 | **return** iterDrop(L_{Cand})
4 **else**
5 | **return** iterDrop(L)

Fig. 6: XDROP algorithm.

well, thus, allowing the network to use better context information. Concretely, it adds the backward $\overleftarrow{h}_t = LSTM(i_t, \overleftarrow{h}_{t+1})$. Then, the hidden state h_t is the concatenation $[\overleftarrow{h}_t, \vec{h}_t]$.

The second variant, TREELSTM [44], has been shown to be suitable for tree-like inputs, such as ASTs. It extends LSTM by considering the linear chain of timesteps as a special case of a tree, in which each node has exactly one child. Given a node i_j in a tree, with $H(i_j)$ is the set of hidden states corresponding to each child node of i_j , TREELSTM extends the equations with $h_j = TreeLSTM(i_j, H(i_j))$. Intuitively, TREELSTM passes information from all children to their parent, allowing better topology information to be learned. In this work, we use the information at the root node as the summary of the whole tree.²

Fig. 4c shows our full model with a Bidirectional LSTM layer on top of a TREELSTM layer in a hierarchical manner. From top to bottom in Fig. 4c, at a literal ℓ_t corresponding to an AST with root $Root_t$, we calculate the following:

$$\begin{aligned}
i_t &= TreeLSTM(Root_t, H(Root_t)) \\
\overleftarrow{h}_t &= LSTM(i_t, \overleftarrow{h}_{t+1}) & \overrightarrow{h}_t &= LSTM(i_t, \overrightarrow{h}_{t-1}) \\
h_t &= [\overleftarrow{h}_t, \overrightarrow{h}_t] & y_t &= W \cdot h_t + b
\end{aligned}$$

where $W \in \mathbb{R}^{|h_t| \times 2}$ and $b \in \mathbb{R}^2$ are the weight matrix and bias that transforms h_t to a vector of size 2. Each equation above corresponds to a layer in Fig. 4c. Finally, the predicted label for ℓ_t is the index of the max value of y_t .

Fig. 6 describes how we use the learned model in our neural-based IG algorithm XDROP. Given that deep learning models could make arbitrary predictions, special care need to be taken in order to preserve soundness. In the worst case, XDROP should be effectively the same as ITERDROP. More formally, we have the following important yet straightforward theorem.

Theorem 1 XDROP is sound and terminating.

XDROP is implemented in Python using PyTorch [36], while SPACER is implemented in C++. We implement a client-server architecture in which XDROP is wrapped in a gRPC server which connects to a gRPC client inside SPACER.

C. Discussion

Using NNs to guide generalization might seem arbitrary at first. Perhaps a simpler heuristic based on counting frequency is sufficient. In fact, we have tried many different handcrafted heuristics first. However, two common problems arose: (a) the

²It is also possible to use the sum of every node in the tree as the summary, as mentioned in [44].

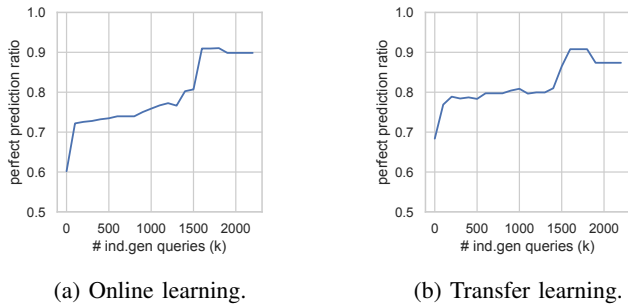


Fig. 7: M 's predictive power for benchmarks with at least k IG queries.

heuristics do not work consistently across different benchmarks; (b) even if a heuristic works, it does not transfer to different properties since different literals are learned for different properties and systems.

There are many alternative ways to guide generalization using a neural component than the one we chose. Perhaps most desirable is to have an end-to-end solution in which the neural component takes an original lemma as input and produces a generalized lemma as output. However, the symbolic reasoning required for this is so complex that we believe that such a solution is much harder to train and scale up. Another alternative is to learn an approximation of the inductive check, i.e., the function $isInductive(Context, L) \mapsto \{true, false\}$ that determines whether a candidate lemma L is inductive in the current context. We have tried such an approach, but could not make it effective. The difficulty is that the $Context$ that is used by the inductive checker is a large symbolic formula. This makes training the network difficult. We suspect it is as hard as *learning a neural SMT-solver* [40], [39].

VI. EMPIRICAL EVALUATION

A. Benchmarks and environment setup

We evaluate ROPEY on a set of simulation benchmarks publicly available³ for the KIND2 model checker [11] (simply called KIND2 from now on). This benchmark suite corresponds to verification of systems that are known to be challenging for IG, for which SPACER behaves poorly. Furthermore, KIND2 benchmarks can be easily grouped into training set (i.e. a set of original benchmarks) and testing set (i.e. a set of corresponding variants). In total, KIND2 consists of 324 benchmarks.

We train ROPEY's neural network M using Adam optimizer [28] with dropout rate 0.5. We set the hidden size of TreeLSTM to be 64, and use embedding dimensions mentioned in Sec. IV.⁴ We stop training when either the performance has not been improved over the last 250 epochs or the number of epochs reaches a predefined threshold (i.e. 1500). Naive Embedding, Positional Embedding and Constant Embedding are always used. Ablation study for those embeddings is

³<https://github.com/kind2-mc/kind2-benchmarks>.

⁴These dimensions could be further fine-tuned, which we leave as interesting future work.

discussed in Sec. VI-E. All experiments are performed on a Linux desktop equipped with an Intel® Xeon E5-2680 v2, an NVIDIA 1080 Ti GPU, and 64GBs of memory. The artifacts including code and data are available on the project website at <https://nhamlv-55.github.io/Ropey>.

Given that evaluating benchmarks with a short running time (i.e. less than one second) is susceptible to noise, for all experiments we report both the numbers for all benchmarks and the numbers for non-trivial benchmarks. We define a non-trivial benchmark as the one that takes at least 5 seconds to solve, or has at least 100 IG queries (depending on whether we are measuring running time or predictive power, respectively).

B. Predictive power

We evaluate the model M in two settings, namely, *online learning* and *transfer learning*. Given a lemma in the form of a list of literals, M predicts a likely inductively generalized lemma, which is a sub-list of the given lemma. We define a prediction returned by M as a *perfect prediction* iff given the same input, vanilla SPACER produces the same exact answer. Note that this is a conservative criterion because there might be multiple valid inductive generalizations.

Online learning In this setting, we collect 144 benchmarks from KIND2 that have at least 2 IG queries in their solving trace. For each of them, we use SPACER to solve it until completion or until a time limit of 930 seconds is reached. Each solving trace is then split in half, and M is trained on the first half to predict the answers to queries seen in the second half of the trace (tail queries). We measure how many percent of the tail queries are perfectly predicted by M . The average length of queries is 9.75 literals.

M achieves 60.19% perfect prediction ratio for all benchmarks and 72.18% for non-trivial benchmarks. The trend of perfect prediction ratio along with the corresponding number of queries are plotted in Fig. 7a, where Y-axis is the perfect prediction ratio and X-axis is benchmarks ordered according to their total number of IG queries. The plot shows that M generally works better for larger benchmarks. For instance, M returns perfect predictions for more than 90% of the queries in benchmarks with 1600 or more IG queries.

Transfer learning In this setting, we use 123 benchmarks (i.e., 30 seed benchmarks and 93 variant benchmarks) from KIND2 based on their naming convention. For example, `metros_2_e1_1116.smt2` is one variant of `metros_2.smt2`. Note that we have fewer benchmarks in this task since some seed benchmarks can be solved without any IG queries, while its variants cannot. Those seeds and variants are all excluded from the task. The average length of the queries for this task is 8.43 literals.

We train M on traces generated by solving the seed benchmarks to completion or until timeout. The models are then used to predict queries asked during the solving process of the corresponding variants.

M achieves 68.36% and 76.89% perfect prediction ratio for all benchmarks and non-trivial benchmarks, respectively. We also plot the trend of perfect prediction ratio in Fig. 7b.

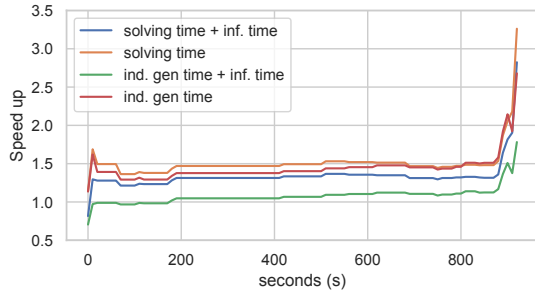


Fig. 8: ROPEY’s speedups for benchmarks taking more than s seconds to solve.

	All	Non-trivial
solving + inf. time	0.81560	1.25385
solving time	1.14085	1.69792
ind. gen time	1.13570	1.63041
ind. gen + inf. time	0.70519	0.91891

TABLE II: ROPEY’s speedups compared with SPACER.

Similar to the online learning setting, M generally works better for larger benchmarks. It is a little surprising that the perfect prediction ratio of transfer learning setting is slightly better than the ratio of online learning. This might indicate that in our benchmarks, queries in the beginning and at the end of the same benchmark are more different than queries between seeds and variants. Quantifying this observation is an interesting direction for future work.

C. Running time

ROPEY’s running time can be broken down into few components: SPACER’s time (in which IG time is a subcomponent), communication time over gRPC, data parsing time, and model running time. We group the later three components into *inferencing time*. On average, inferencing takes 48.1% and 24% of the total running time for all and non-trivial benchmarks, respectively. For future work, we state that there are opportunities for engineering improvement to reduce the inferencing time.

We measure the speedup in IG time and SPACER’s solving time with and without the inferencing time. If ROPEY times out, we measure the running time that ROPEY needs to verify to the same depth as SPACER. The timeout is set to be 930 seconds, and in cases where ROPEY times out, we rerun it with the timeout set to 2790 seconds to allow it to verify to the same depth as SPACER. The results are in Table II. We also plot in Fig. 8 the speedups achieved at different running time threshold s , e.g for benchmarks that takes more than 50 seconds to solve, 100 seconds to solve, etc.

For unsolved benchmarks, notice the spikes at the tail of Fig. 8: ROPEY takes much less time to reach to the same depth as SPACER, up to $2.8\times$ faster (inferencing time included).

D. Training time

In this paper, we specifically consider realistic applications where training time is not a bottleneck – train once on one instance and apply to many similar instances (offline), or train during a very long run (days or weeks) and apply to the rest of

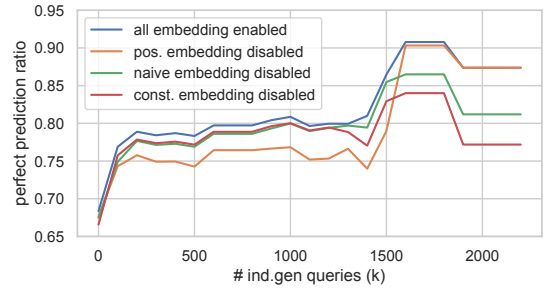


Fig. 9: Effects of using different embeddings for benchmarks with at least k IG queries.

the run (online). For that reason, we do not optimize training code, nor do we run training in an isolated environment where time measurements are meaningful. Nonetheless, we share some statistics of the training time – the minimum, median and maximum training time are 17, 1027 (17 minutes), and 165811 seconds (46 hours), respectively. More details are hosted on our project webpage https://nhamlv-55.github.io/Ropey/training_time. Training any individual model (i.e., when GPU is used to train only a single model) is faster, but training all models sequentially is too slow. Since we do not consider training time itself to be of significant interest, we train as many models in parallel as possible.

E. Ablation study

Embedding variables and constants is crucial for our tasks. In this ablation study, we evaluate three embeddings we proposed in Sec. IV-B for handling variables and constants. Fig. 9 shows four plots of ROPEY with four different embedding configurations. ROPEY achieves the best performance when all embeddings are enabled. ROPEY’s performance drops dramatically when the positional embedding is disabled, indicating leveraging variable’s position information helps for capturing co-occurrence patterns. Disabling Naive Embedding or Constant Embedding does not affect the performance much for benchmarks with relatively small number (i.e. < 1000) of IG queries, however, the performance drops dramatically when the number of IG queries becomes large.

VII. RELATED WORK

There has been a number of work studying neural learning for symbolic reasoning. Some studied the capability of deep learning models on handling relatively simple symbolic reasoning tasks, such as symbolic expression equivalence [1] or logical entailment [19], which can be easily performed by a symbolic engine like SMT solver. [2] and [37] focus on learning embeddings of programs using paths over abstract syntax trees or control flows, and the learned embeddings are helpful for suggesting function or variable names. Our focus is on improving state-of-the-art symbolic engines on non-trivial symbolic reasoning tasks like symbolic model checking. The most relevant work is [4], which predicts a high-level strategy (or configuration) of an SMT solver based on *static* statistics of a verification instance. In contrast, our approach learns from

dynamic runs and provides guidance for decisions in a finer granularity. Two other related work are [24] and [42]. The former also uses deep learning to guide numerical analysis, where the soundness is not a concern as imperfect prediction results in less precise (but still acceptable) numerical approximations. Like our problem, the latter also faces the soundness issue and proposes an end-to-end reinforcement learning based approach, which however suffers from scalability issues.

VIII. CONCLUSION

In this paper, we explore how deep neural networks can be used in IC3. We chose inductive generalization because it is (a) a common bottleneck; and (b) seemed suitable to optimize with NNs. We view this as a first step in using data-driven NNs to guide IC3. Specifically, we propose a data-driven approach to improving inductive generalization, which effectively embeds symbolic formulas in fixed-length vectors and uses a hierarchical recurrent neural network to guide inductive generalization (i.e., predict which literals of a lemma should be kept or dropped). We build a prototype, ROPEY, and evaluate it on KIND2 benchmark suite. We observe promising predictive power of neural networks in inductive generalization and modest improvement in terms of absolute running time over the state-of-the-art SMC engine, SPACER, which boosts the solving time for non-trivial instances by 25%.

Our work shows that it is possible for NNs to learn complex symbolic patterns in IC3, and such learned patterns can be used to improve IC3. ROPEY's pure performance does not show a strong gain yet, but is still encouraging. We envision the performance gain would be much more significant by improving ROPEY with better engineering effort or leveraging advanced hardware acceleration for deep learning models in the future (like TPUs). Another orthogonal improvement is to explore more advanced transformer-based language models like GPT-3 [9] to further improve the prediction accuracy.

REFERENCES

- [1] M. Allamanis, P. Chanthirasegaran, P. Kohli, and C. A. Sutton, "Learning continuous semantic representations of symbolic expressions," in *ICML 2017*, vol. 70, 2017.
- [2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *POPL*, vol. 3, 2019.
- [3] T. Ball, "Secrets of software model checking," in *AGP 2002*, 2002.
- [4] M. Balunovic, P. Bielik, and M. T. Vechev, "Learning to Solve SMT Formulas," in *NeurIPS*, 2018.
- [5] J. Barnat, L. Brim, A. Krejci, A. Streck, D. Safranek, M. Vejnar, and T. Vejpustek, "On parameter synthesis by parallel model checking," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 3, 2012.
- [6] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Gurevich 75*, 2015.
- [7] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011.
- [8] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *FMCAD*, 2011.
- [9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS*, vol. 33, 2020, pp. 1877–1901.

- [10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, 1986.
- [11] A. Champion, A. Mebsout, C. Stickels, and C. Tinelli, "The Kind 2 Model Checker," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, 2016, pp. 510–517. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_29
- [12] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, p. 135–143.
- [13] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods Syst. Des.*, vol. 19, no. 1, 2001.
- [14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000.
- [15] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*, 1st ed. Springer Publishing Company, Incorporated, 2018.
- [16] E. M. Clarke, K. L. McMillan, S. V. A. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *CAV*, 1996.
- [17] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008.
- [18] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003.
- [19] R. Evans, D. Saxton, D. Amos, P. Kohli, and E. Grefenstette, "Can neural networks understand logical entailment?" in *ICLR*, 2018.
- [20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *PLDI*, 2002.
- [21] A. Griggio and M. Roveri, "Comparing different variants of the ic3 algorithm for hardware model checking," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 6, 2016.
- [22] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *CAV*, 2015.
- [23] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 157–164.
- [24] J. He, G. Singh, M. Püschel, and M. T. Vechev, "Learning fast and precise numerical analysis," in *PLDI*, 2020.
- [25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997.
- [26] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, vol. 7317, 2012.
- [27] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett, "The marabou framework for verification and analysis of deep neural networks," in *CAV*, 2019.
- [28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization." *CoRR*, vol. abs/1412.6980, 2014.
- [29] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," in *CAV*, 2014.
- [30] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, "Automatic abstraction in smt-based unbounded software model checking," in *CAV*, 2013.
- [31] H. G. V. Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, "Global guidance for local generalization in model checking," in *CAV*, 2020.
- [32] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, 2006.
- [33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NeurIPS*, 2013.
- [34] —, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, 2013.
- [35] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC*, 2001.
- [36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NeurIPS*, 2019.

- [37] V. K. S. R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, “Ir2vec: A flow analysis based scalable infrastructure for program encodings,” *CoRR*, vol. abs/1909.06228, 2019.
- [38] M. Schuster and K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
- [39] D. Selsam and N. Bjørner, “Guiding High-Performance SAT Solvers with Unsat-Core Predictions,” in *SAT*, 2019.
- [40] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, “Learning a SAT Solver from Single-Bit Supervision,” in *ICLR*, 2019.
- [41] O. Sheyner, J. W. Haines, S. Jha, R. Lippmann, and J. M. Wing, “Automated generation and analysis of attack graphs,” in *SSP*, 2002.
- [42] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, “Learning loop invariants for program verification,” in *NeurIPS*, 2018.
- [43] J. P. M. Silva and K. A. Sakallah, “GRASP – a new search algorithm for satisfiability,” in *ICCAD*, 1996.
- [44] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” in *ACL*, 2015.
- [45] H. Tsai, J. Riesa, M. Johnson, N. Arivazhagan, X. Li, and A. Archer, “Small and practical BERT models for sequence labeling,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3632–3636. [Online]. Available: <https://www.aclweb.org/anthology/D19-1374>
- [46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.
- [47] X. Zhang and H. Wang, “A joint model of intent determination and slot filling for spoken language understanding,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI’16. AAAI Press, 2016, p. 2993–2999.