Learned Formal Proof Strengthening for Efficient Hardware Verification

Minwoo Kang
Azade Nova
Eshan Singh
Geetheeka Sharron Bathini
Yuriy Viktorov
John Wawrzynek, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-260
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-260.html

December 1, 2023

Acknowledgement

I would like to express my deep gratitude to my advisor, Prof. John Wawrzynek, for the continued support and guidance. I would also like to thank Prof. Sophia Shao for serving as my second reader. Finally, I thank the collaboration, mentoring, and support by the Google CI2 team members who co-authored this work.

Learned Formal Proof Strengthening for Efficient Hardware Verification

by Minwoo Kang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John Wawrzynek
Research Advisor

[illegible]

(Date)

* * * * * * *

DocuSigned by:
Sophia Shao
BD2654D7DCA8400...

Professor Sophia Shao
Second Reader

8/9/2023

(Date)

Learned Formal Proof Strengthening for Efficient Hardware Verification

by

Minwoo Kang

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor Sophia Shao

Summer 2023

Abstract

Learned Formal Proof Strengthening for Efficient Hardware Verification

by

Minwoo Kang

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

Proof decomposition via assume-guarantee with helper properties, i.e. *helpers*, is one of the most promising approaches to address the complexity of hardware formal verification (FV). While helpers can be hand-crafted by human experts, state-of-the-art methods replace this labor-intensive process with circuit analysis, from which a large set of helper candidates can be quickly synthesized. However, these candidates are often of lower-quality, and distilling the full auto-generated set down to a subset of high-quality, appropriate helpers still requires significant computational or manual effort. In this work, we propose a novel approach to automate helper quality assessment: Learned Formal Proof Strengthening (LFPS), a neural model that can accurately predict helper effectiveness without having to run actual formal proofs. The LFPS model jointly learns representations of property specifications expressed in SystemVerilog and the graph representation of the circuit under verification. When evaluated against three RTL designs from industrial SoCs, our model is shown to achieve significant predictive performance, with an average of 98.2% accuracy and 98.3% F-1 score. Once trained, the model can also serve as a fast predictor for search algorithms: we show that LFPS neural-guided search can outperform random sampling in finding sets of helper candidates that achieve inductive proof strengthening.

Contents

Contents **i**

1 Introduction **1**

2 Preliminaries **4**

3 Learned Formal Proof Strengthening **6**
- 3.1 Problem Definition 6
- 3.2 Model Architecture 7

4 Datasets and Data Augmentation **10**
- 4.1 Dataset Generation 10
- 4.2 Data Augmentation 12

5 Experiments **15**
- 5.1 Next Helper Effectiveness Prediction 15
- 5.2 Ablation Study: Data Augmentation 16
- 5.3 Searching for Effective Helper Subsets 17

6 Related Work **19**

7 Conclusion **20**

Bibliography **21**

Chapter 1

Introduction

Hardware verification is a crucial and growing stage in the digital circuit design process, necessary to make sure critical bugs do not escape detection [14]. Unlike simulation-based verification, formal verification (FV) can mathematically prove that a design meets a given specification for all possible input stimuli [23], finding all difficult to detect, corner-case bugs. Such formal specifications of the expected behaviors for a design are referred to as *assertions*, and it is the goal of FV to obtain guarantees that the assertions hold in all design states, which we refer to as formal *proofs* [9]. However, FV suffers greatly from the increasing complexity of the design due to the exponential growth of the number of design states [11]. As a result, FV often fails to reach a formal proof or find a counterexample for an assertion, despite running for days on multi-core CPU servers.

Proof decomposition via assume-guarantee with helper properties provides a promising approach to addressing this problem, by using simpler, proven properties (i.e. *helpers*) to reduce the complexity of the assertion [16]. Practical adoption of this approach, however, remains hindered by the cost of producing meaningful helper properties. Aside from the manual development by a human expert, the state-of-the-art in helper generation is based on assertion mining methods that produce a large set of candidates using circuit analysis [38, 17, 7, 26, 37]. However, these auto-generated assertions are often of lower-quality [32] such that the full set of candidates needs to be reduced to the most reasonable subset. Furthermore, the same given helper that achieves proof strengthening for one assertion might not be effective for another assertion we intend to prove. To the best of our knowledge, there are no known heuristics to reliably estimate such effectiveness of individual helper assertions, let alone of arbitrary sets of helpers. The remaining option is to run actual formal proofs, where each iteration takes at least 30 minutes. Therefore, a brute-force search over all combinations of helpers quickly becomes impractical, and random or naive heuristic-based search is the only available method to search for appropriate helper subsets.

This work presents a first approach towards *learning to reason about formal properties of hardware designs*. We suggest that in the absence of established heuristics, a data-driven approach can provide breakthroughs in automating the reasoning about helper effectiveness. In particular, we base our proposal on the fact that hardware designs are typically described at the register-transfer level (RTL) using a hardware description language (HDL) such as SystemVerilog [14]. Formal properties are expressed in the corresponding SystemVerilog Assertion (SVA) syntax which

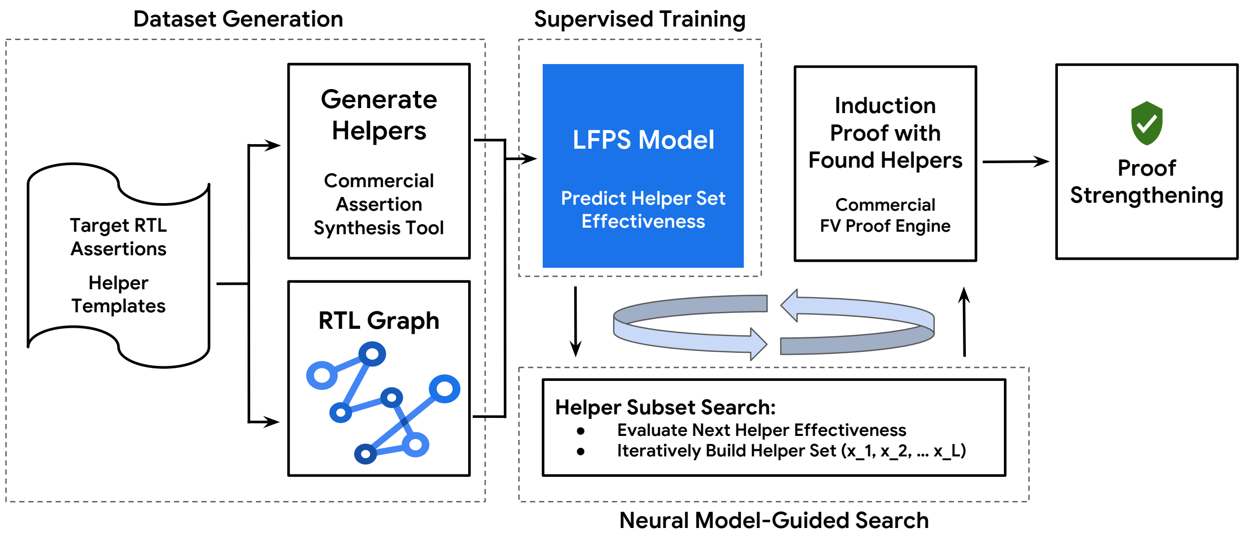

Dataset Generation
Target RTL Assertions
Helper Templates
Generate Helpers
Commercial Assertion Synthesis Tool
RTL Graph
Supervised Training
LFPS Model
Predict Helper Set Effectiveness
Induction Proof with Found Helpers
Commercial FV Proof Engine
Proof Strengthening
Helper Subset Search:
Evaluate Next Helper Effectiveness
Iteratively Build Helper Set (x_1, x_2, ... x_L)
Neural Model-Guided Search

Figure 1.1: Overall workflow of auto-generating helper assertions, rapidly evaluating helper effectiveness via the LFPS model, and iteratively searching for the best subset out of a large pool of auto-generated candidates. Set of helpers found by the neural model-guided search are then used as assumptions to strengthen the inductive proof of the target assertion.

consists of: (1) Boolean, arithmetic, or temporal logic operators, and (2) natural language text describing the user-defined variable names. Motivated by the recent success of machine learning models in understanding natural languages [13, 20, 33, 6, 28] and software programs [31, 8, 3, 4], we investigate the viability of learning representations of formal properties through their textual descriptions.

To this end, we present *Learned Formal Proof Strengthening (LFPS)*—an end-to-end neural approach of learning to predict helper effectiveness. As illustrated in Fig. 1.1, we first auto-generate candidate helpers using a commercial assertion mining tool, and also extract a high-level graph representation of the RTL that capture syntax-level dependencies between SystemVerilog signal variables. Then, we take random subsets of helpers and measure their effectiveness by running formal proofs to produce supervised datasets. We train our proposed LFPS model on this data towards autoregressively predicting the effectiveness of new helpers, and we demonstrate how the trained model could be used as a fast predictor to search for appropriate helper subsets. The following summarize our key contributions:

- Proposal and formalization of a new learning task with significant implications in industrial hardware verification. In Section 3.1 and 4.1, we detail the problem of learning to predict the effectiveness of helper properties and describe methods of data generation for this task.
- A novel neural architecture to capture the semantics of formal assertions and context of the underlying digital circuit. In Section 5.1, we show that the model is capable of accurately

predicting helper effectiveness for real-world, industrial designs with significant complexity—we achieve on average, a 98.2% accuracy and a 98.3% F-1 score.

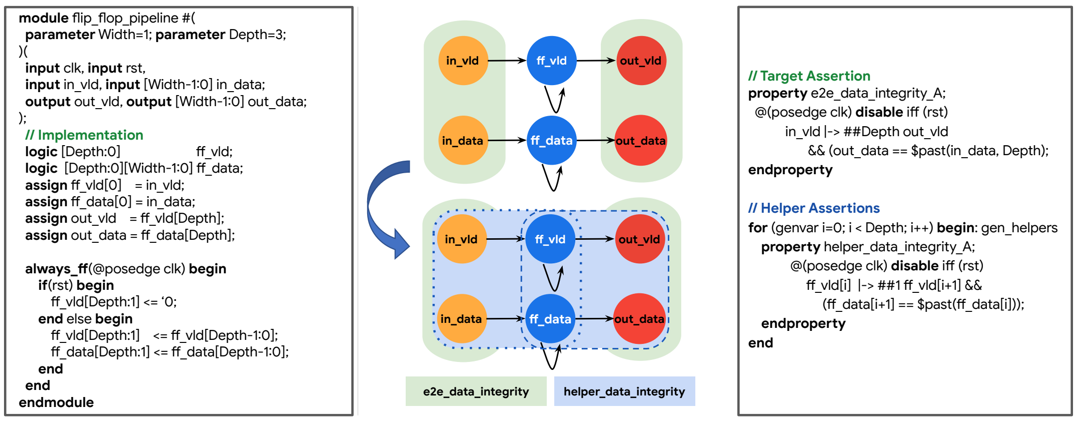
- A data augmentation methodology to resolve fundamental challenges in supervised dataset generation for the proposed task: size limitations and label imbalance. We show in 5.2 that our proposed data augmentation significantly improves model accuracy, up to a 23.2 % increase in accuracy.
- A proof-of-concept neural-guided search algorithm based on the train LFPS model. In Section 5.3, we show that a naive implementation of model-guided beamsearch outperforms random search for two out of three RTL designs, improving `pass@1` by up to 0.45 and `pass@5` by up to 0.25.

Chapter 2

Preliminaries

Hardware FV aims to determine if a digital circuit implementation satisfies a given specification. This specification consists of a list of properties written as Boolean formulae. As *assumptions*, they constrain the input stimuli applied by the FV tool; and as *assertions*, they define the expected behavior to be checked by the tool.

An example of a RTL design containing a flip-flop (FF) pipeline is shown in Fig. 2.1. A target assertion concerning the given example is also shown on the right of the same figure. This assertion describes an end-to-end property of the design regarding data integrity—input data to the module should be correctly observed as outputs after a set number of cycles. Formally verifying



```
module flip_flop_pipeline #(
 parameter Width=1; parameter Depth=3;
)(
 input clk, input rst,
 input in_vld, input [Width-1:0] in_data;
 output out_vld, output [Width-1:0] out_data;
);
 // Implementation
 logic [Depth:0]                ff_vld;
 logic  [Depth:0][Width-1:0] ff_data;
 assign ff_vld[0]    = in_vld;
 assign ff_data[0] = in_data;
 assign out_vld    = ff_vld[Depth];
 assign out_data = ff_data[Depth];

 always_ff(@posedge clk) begin
   if(rst) begin
     ff_vld[Depth:1] <= '0;
   end else begin
     ff_vld[Depth:1]    <= ff_vld[Depth-1:0];
     ff_data[Depth:1] <= ff_data[Depth-1:0];
   end
 end
endmodule
```

```
// Target Assertion
property e2e_data_integrity_A;
  @(posedge clk) disable iff (rst)
        in_vld |-> ##Depth out_vld
            && (out_data == $past(in_data, Depth);
endproperty

// Helper Assertions
for (genvar i=0; i < Depth; i++) begin: gen_helpers
  property helper_data_integrity_A;
       @(posedge clk) disable iff (rst)
          ff_vld[i]  |-> ##1 ff_vld[i+1] &&
             (ff_data[i+1] == $past(ff_data[i]));
  endproperty
end
```

Figure 2.1: Left: An example RTL module expressed in SystemVerilog. Right: Data integrity assertion and related helpers for the RTL module, expressed in SVA syntax. Center: RTL graph representing the given example, with nodes for clock and reset omitted. Orange nodes indicate module input ports; solid blue indicate internal FFs; and red indicate output ports. Nodes highlighted by a light-green background are nodes to which the target assertion maps. The connection between nodes highlighted in light-blue represent the helpers used as assumptions to strengthen the proof of the target.

any assertion typically refers to performing mathematical *k*-induction [36]. *k*-induction proofs have a reasonable time-complexity for small numbers of *k*, but are generally NP-hard [30]. As a result, FV with *k*-induction remains limited by design complexity.

To address this proof complexity challenge, we consider proof decomposition, which aims to strengthen the inductive proof of the target assertion by supplying appropriate sets of helper properties [16]. Example of such helpers are shown on the right of Fig. 2.1. Without the use of helpers, proving the target assertion requires checking the design behavior end-to-end. This is visually represented in the figure by the discontinuity between nodes that map to the target assertion, highlighted in light-green.

On the other hand, the shown helpers concern relationships between the internal FF states and the module input/output ports. Compared to proving the full, end-to-end behavior of the design, proving the helpers is a more manageable task. And once proven, these helpers can be used as assumptions about the design, reducing the state space that the proof engine needs to examine in order to reach a formal proof for the target assertion. In Fig. 2.1, the effects of using helpers as assumptions is represented by the connection between nodes annotated in light-blue. Note how the use of helpers as assumptions bridges the previously discontinued set of nodes highlighted in light-green.

Chapter 3

Learned Formal Proof Strengthening

3.1 Problem Definition

A RTL design can be represented as a dataflow graph of $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{A})$, where nodes $\mathcal{V}$ are ports or signals in the design, edges $\mathcal{E}$ represent assignments connecting signals at the RTL, feature matrix $\mathcal{A}$ represents node attributes including name, type, fan-in, fan-out, etc. Let the set of undetermined target assertions be $\mathcal{T} = \{a_{\text{target}}\}$ and the set of all synthesized helper assertions be $\mathcal{X} = \{x\}$.

We are interested in learning to predict the effectiveness of sets of helpers. A direct approach would be to model the effectiveness of arbitrary sets of helper candidates $\{x_i\} \subseteq \mathcal{X}$ up to some maximum size $|\mathcal{X}_i| \leq L$. However, for a large pool of helper candidates $|\mathcal{X}| >> 1$ and multiple target assertions $|\mathcal{T}|$, the total complexity of learning to predict all possible combinations of helper subsets and target assertions becomes quickly intractable, since the number of combinations grows exponentially $\sim |\mathcal{T}| \cdot |\mathcal{X}|^L$.

Autoregressive Classification of Next Helper Effectiveness. A promising alternative approach is to learn the conditional probability of whether the addition of a new helper assertion further improves complexity reduction of the proof, on top of the already assumed subset of helpers. Modeling this conditional probability enables an autoregressive approach to incrementally predict effectiveness of candidate helper given existing helpers.

Formally, suppose we have a dataset $\mathcal{D}$, where each data point $(y, x, \{x_i\}, a_{\text{target}}, \mathcal{G})$ consists of a previously assumed set of helpers $\{x_i\} \subseteq \mathcal{X}$, the target assertion a_{target}, RTL graph $\mathcal{G}$, the next helper candidate x, and the corresponding binary label, y, describing whether the new helper further improves proof complexity reduction. In other words, a binary label is assigned for an unexplored helper $x \in \mathcal{X} \setminus \{x_i\}$ as:

$$\begin{aligned} y = \mathbb{1} \iff & \texttt{proof\_bound}(x \cup \{x_i\}, a_{\text{target}}, \mathcal{G}) \\ & - \texttt{proof\_bound}(\{x_i\}, a_{\text{target}}, \mathcal{G}) > 0. \end{aligned}$$

We can then express the conditional probability as:

$$Pr(y; x \,|\, \{x_i\}, a_{\text{target}}, \mathcal{G}).$$

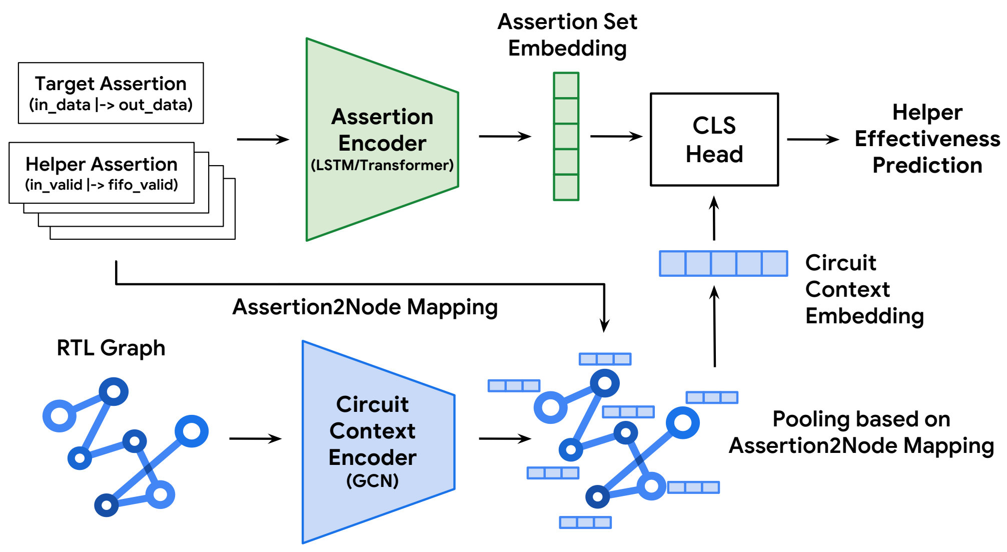

Assertion Set Embedding
Target Assertion (in_data |-> out_data)
Helper Assertion (in_valid |-> fifo_valid)
Assertion Encoder (LSTM/Transformer)
CLS Head
Helper Effectiveness Prediction
Circuit Context Embedding
Assertion2Node Mapping
RTL Graph
Circuit Context Encoder (GCN)
Pooling based on Assertion2Node Mapping

Figure 3.1: Model architecture for classifying helper assertions by their effectiveness in reducing inductive proof complexity. The full model comprises of an assertion specification encoder and a circuit graph context encoder.

3.2 Model Architecture

We propose a model architecture that jointly learns and leverages representations of formal assertion specifications and the underlying circuit context. We hypothesize that capturing both representations leads to more accurate predictions and better model generalization to subsets of helper assumptions unseen during training. The overall architecture is described in Fig. 3.1, including the two parallel encoders for the assertion specifications and circuit context, and a final classifier head that combines the two embeddings.

Assertion Encoder (AE). This encoder takes the formal property specification expressed in SVA syntax as inputs and outputs an embedding that summarizes the target assertion a_{target}, previously assumed helpers $\{x_i\}$, and the next helper candidate x. To do so, the textual representations of each assertion must first be tokenized into vectors of integers. Consider the following example, a part of the example helper property from Fig. 2.1:

```
ff_vld[i] |-> ##1 ff_vld[i+1]
```

The raw specification text contains a mix of natural language tokens describing the signal or port names (e.g. `ff_vld`) and SVA operators (`|->, ##, +`). We tokenize the given example into a sequence of tokens as:

```
[ ff, vld, [], i, |->, ##, 1, ff, vld, [], (, i, +, 1, )]
```

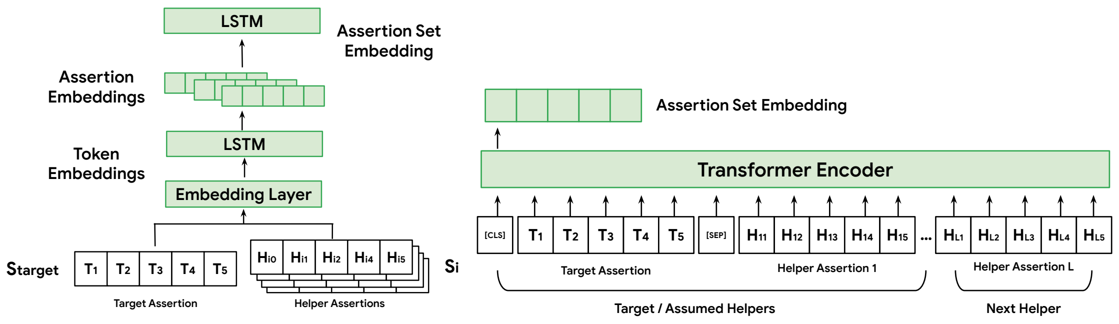


Figure 3.2: Left: LSTM-based Assertion Encoder—individual assertions are first encoded into separate assertion embeddings, and then passed to a second LSTM layer to produce an embedding for the full set of assertions. For brevity, token sequences are represented for the case of $n = 5$. Right: Transformer-based Assertion Encoder architecture—all token sequences of assertions are concatenated with separation tokens and passed to the Transformer encoder.

Note that we use a simple, custom tokenization scheme. In this work, we do not consider existing tokenizers for natural language (e.g. WordPiece [34] or Byte-Pair Encoding [35]) or that for software programs [20], since they cannot natively process SystemVerilog syntax. Future work may explore improvements to the simple tokenizer we are using or find ways to extend existing tokenizers.

Formally, for each helper in the subset $\{x_i\}$, we refer to its tokenized helper specification as $\mathcal{S}_i = [H_{i,0}, H_{i,1}, \ldots, H_{i,n}]$. Similarly, we refer to the tokenization of the target assertion as $\mathcal{S}_{\text{target}} = [T_0, T_1, \ldots, T_n]$.

LSTM-based AE. Formal property specifications, similar to natural language sentences, are inherently sequential. Based on this intuition, we employ a recurrent neural network (RNN) such as Long-Short Term Memory (LSTM) [18] to model each property specification based on its sequence of token embeddings. The left figure of Fig. 3.2 visualizes this process of passing a sequence of tokens $\mathcal{S}_i$ to the embedding layer, and the output sequence of token embeddings then fed into a LSTM layer. In short, the latent representation of each assertion is formulated as:

$$\text{AssertEmbed}(x_i) = \text{LSTM}\Big(\text{Embed}(\mathcal{S}_i)\Big).$$

The equation above shows the case for helpers, but the same formulation applies to target assertions as well.

From this, multiple assertion embeddings are produced: one for the target assertion and one for each of the helpers. A second layer of LSTM then aggregates the sequence of assertion embeddings to output the final embedding summarizing the entire set of assertions.

Transformer-based AE. However, recall that in the problem formulation of next helper effectiveness prediction (Section 3.1), we explicitly differentiate the *context* component (already assumed helpers

$\mathcal{X}_i$ and target assertion a_{target}) from the *question* component (next helper candidate x) of the input list of assertion specifications. This partitioning of the input token sequence is similar to how generative Question Answering (QA) tasks are formulated for language models (LMs).

To enable this explicit conditioning, and to capture the interrelations between potentially larger sets of helpers, we replace the LSTM-based encoder with a Transformer Encoder backbone with multi-head self-attention [40], as shown on the right of Fig. 3.2. Similar to BERT [13], this Assertion Encoder uses segment embeddings to differentiate between the context and question token sequences. We also include the special classification token (`[CLS]`) and separation token (`[SEP]`) to mark the beginning and end of each token sequence. The output hidden vector for the `[CLS]` token is read out from the Transformer encoder as the assertion set embedding, summarizing the entire set of helpers and target assertion:

$$\begin{aligned}\text{AssertSetEmbed}(a_{\text{target}}, \{x_i\}, x) = \\ \text{Transformer}\Big(\text{Concat}\left(\mathcal{S}_{\text{target}}, \mathcal{S}_1, \ldots, \mathcal{S}_L\right)\Big)\end{aligned}$$

where the previously assumed set $\{x_i\}$ contains $L-1$ helpers. The token sequence for the next candidate helper x is represented as the final, L-th token sequence $\mathcal{S}_L$.

Circuit Context Encoder (CCE). This module takes the RTL Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{A})$ as input and passes through a message-passing GNN. Also given as input are mappings between each assertion x and a subset of nodes in the RTL graph. , $\mathcal{V}_x \subseteq \mathcal{V}$, that correspond to signals that constitute the assertion specification. This mapping between each assertion and graph nodes are pre-computed during dataset generation. Node embeddings corresponding to each assertion are then mean-pooled to generate a circuit context embedding per assertion. With a l-layer GNN backbone, the circuit context embedding for each assertion x_i can thus be derived as:

$$\text{CircuitContextEmbed}(x_i) = \text{Pool}\Big(\Big\{h_v^{(l)} | v \in \mathcal{V}_{x_i}\Big\}\Big).$$

where $h_v^{(l)}$ is th final, l-th GNN layer node embedding of node v in the RTL Graph. And finally, the circuit embedding concerning all assertions in the data point are computed by mean-aggregating all circuit context embeddings. In choosing the particular GNN architecture, we follow [39] and use the graph convolutional networks (GCN)[21] as the encoder model capturing latent representations of the high-level RTL graph.

Classifier Head. Embeddings produced by both encoders are concatenated and fed into a final linear layer with sigmoid activation. Final outputs are binary predictions on the effectiveness of the new helper candidate x.

Chapter 4

Datasets and Data Augmentation

4.1 Dataset Generation

To the best of our knowledge, this is the first work to apply deep learning to the problem of estimating helper property effectiveness in reducing proof complexity, or more broadly, to the problem of reasoning about formal properties of digital circuits. Furthermore, there are no established datasets or benchmarks for hardware verification that: (1) concern industry-scale designs and (2) include formal testbenches with appropriate formal properties. Therefore, as part of this work, we generate new datasets for the next helper effectiveness classification.

We consider a total of three RTL designs, each a component of an industrial ML accelerator SoC design. 1R1W FIFO is a classical single-stream FIFO queue implementation; OoO-Read buffer is a buffer implementation with in-order writes but out-of-order read operations; the Direct Memory Access (DMA) controller is an implementation of a typical controller module that tracks the validity of data transfers between hosts and end points of a DMA interface. High-level design details about the complexity of RTL used in this work are summarized in Table 4.1.

As target assertions, we take five data integrity assertions from the formal testbenches of each design. Data integrity refers to a commonly used suite of checks to ensure the correctness of data flow through the design that are highly relevant in formal verification. The considered target assertions do not converge (reach a full proof) within six hours of running induction proof engines using Cadence Jasper FPV, an industry-standard formal property verification tool.

For each RTL design, we run an industry-grade formal property mining tool, Jasper Behavioral Property Synthesis (BPS), to generate a set of auto-generated helpers. Inputs to BPS are the full set of SystemVerilog files describing the RTL design and five RTL simulation traces generated from running random test programs about each design. We also supply a list of templates shown in Table 4.2, based on which BPS matches its internal extracted list of signal candidates to synthesize assertions. Once generated, we run formal engines through Jasper FPV to prove all synthesized assertions and mark those that are successfully proven within two hours of wall-clock runtime. Only the assertions that are proven within this timeout are considered as helper assertion candidates and used for dataset generation. The exact number of final helper candidates for each RTL are listed in

Table 4.1: High-level details about the RTL designs used in this study. TB stands for testbench.

Parameter Count	1R1W FIFO	OoO-Read Buffer	DMA Controller
Helpers Generated	1617	1443	2332
Assumptions	0	17	33
Covers	80	109	471
Embedded Assertions	46	224	216
Flip-Flops (FFs)	29	164	161
Gates	1295	8386	8348
Lines of Code (LOC)	3996	10232	17184

Table 4.2: BPS templates used in this study. All templates are base on assertions that check for data integrity.

Template Name	Signal List	Expression	Signal Constraints
Data Integrity 1	`X,Y`	`X |-> Y;`	`X.width=1, Y.width=1`
Data Integrity 2	`X,Y`	`X |-> (Y == 0);`	`X.width=1`
Data Integrity 3	`X,Y,Z`	`(X && (Y == 0)) == Z;`	`X.width=1, Z.width=1`
Data Integrity 4	`X,Y,Z`	`(X && (Y == 0)) == Z;`	`X.width=1, Z.width=1`
Data Integrity 5	`X,Y`	`X == Y;`	`X.name, Y.name=~".*value"`
Data Integrity 6	`X,Y,Z`	`X |-> (Y >= Z);`	`X.width=1`
Data Integrity 7	`X,Y,Z`	`X |-> (Y == Z);`	`X.width=1`
Data Integrity 8	`X,Y,Z`	`!X |-> (Y == Z);`	`X.width=1`
Data Integrity 9	`X`	`X <= N;`	`X.width=1, number N.value>0`
Data Integrity 10	`X,Y,Z`	`(X == 0) && !Y |-> !Z ;`	`Y.width=1, Z.width=1`

the first row of Table 4.1.

For each design, we randomly sample a total of 2000 sequences of helpers, each of length 5. Since we consider five target assertions, there are 400 sequences of five helpers sampled for each target assertion. For each sequence of helpers, e.g. $[x_1, x_{20}, x_{31}, x_{1500}, x_{100}]$, we attempt an induction proof of the target with Jasper FPV assuming the assertions $\{x_1\}$, $\{x_1, x_{20}\}$, ..., $\{x_1, x_{20}, x_{31}, x_{1500}, x_{100}\}$. In other words, a total of five separate induction proof attempts are performed per sequence. All proof bounds are measured by setting the verification tool to time out at 30 minutes. The ground-truth label, y, is assigned by comparing the bounds reached by assuming two incrementally large sets of helpers. For example for the data point with ssumed helper set $\{x_i\} = \{x_1, x_{20}\}$ and next helper candidate $x = x_{31}$, the label is computed by comparing the bound reached by assuming $\{x_1, x_{20}\}$ and

Table 4.3: Binary label ratio in the training datasets, broken down by target assertion. Before data augmentation, significant label imbalance is observed; after data augmentation, label imbalance is resolved for most cases.

Target Assert ID	1R1W FIFO		OoO-Read Buffer		DMA Controller	
	Before	*After*	Before	*After*	Before	*After*
0	0.178	0.515	0.296	0.669	0.024	0.097
1	0.299	0.666	0.335	0.705	0.008	0.040
2	0.263	0.626	0.272	0.645	0.148	0.446
3	0.150	0.458	0.316	0.684	0.053	0.208
4	0.18	0.516	0.355	0.724	0.125	0.403

by assuming $\{x_1, x_{20}, x_{31}\}$. Thus, a total of 10K data points are collected per RTL design with 2K points per each target assertion. We perform a standard 60%-20%-20% split of train, validation, and test data in all experiments.

4.2 Data Augmentation

Two distinctive challenges arise when applying supervised learning to our problem. The first challenge is the *cost of obtaining supervised labels*. Labeling each data point requires an actual attempt of running formal proof with the helpers assumed. And the time to resolve whether the helpers are actually effective typically requires at least half an hour, even for relatively small RTL designs. Concretely, for this work, we collect 10K data points with a 30 minute timeout on Jasper FPV, all sessions running on cloud VMs with 8 vCPU cores from Intel Xeon Processors. The total compute cost of producing each dataset is 5K CPU-hours, and we repeat the data collection for three separate designs. We parallelize data collection with 200 CPU instances, bringing down the nominal wall-clock time of data collection to 25 hours per dataset. Nonetheless, the total computational cost remains the same, and this parallelization is only possible with access to such a large amount of compute resources as well as licenses for the commercial FV tool. Overall, the time and computational burden of obtaining ground-truth labels fundamentally limits the size supervised datasets that can be created for the learning problem at hand.

Second, due to the nature of auto-generated helper candidates, the likelihood of finding effective helper subsets is relatively low. This implies that a dataset collected from evaluating a list of randomly selected helper subsets can have significant *label imbalance* with a very low ratio of positive labels. For example, Table 4.3 summarizes the ratio of positive labels per target assertion in each dataset we generated for this work. We observe a noticeable degree of label imbalance, particularly for the DMA Controller design. While this is a common challenge in many real-world applications of machine learning, label imbalance combined with limitations to supervised dataset size can significantly hinder model training.

Assertion-Level Augmentation. In response to the challenges listed above, we introduce a data augmentation methodology that leverages inherent symmetries found in Boolean formulae describing property specifications. Consider the following example, adapted from the helper assertion in Fig. 2.1:

```
in_vld |-> ff_data[i+1] == $past(ff_data[i]).
```

Out of the two logical operators in the assertion, the second operator, `'=='`, indicating a logical comparison is associative. Therefore, a logically equivalent assertion can be constructed by swapping the left- and right-hand sides around the operator:

```
in_vld |-> $past(ff_data[i]) == ff_data[i+1].
```

We refer to this idea as *assertion-level* data augmentation.

Set-Level Augmentation. On a different-level, augmented data can also be obtained from re-ordering the sequence of helpers that appear in the context helper set $\{x_i\}$, which we refer to as *set-level* augmentation. This is based on the observation that the relative ordering of already assumed helpers does not impact whether the newly added helper will further improve or hurt proof complexity reduction. For example, consider a data point in the dataset $(x, \{x_i\}, a_{\text{target}}, y)) \in \mathcal{D}$ where $\{x_i\} = [x_1, x_{20}, x_{31}, x_{500}] \subseteq \mathcal{X}$. An equivalent augmentation can be obtained from a random permutation of the previously assumed list of helpers $\{x_i\}$, such as $\widetilde{\{x_i\}} = [x_{20}, x_{31}, x_1, x_{500}]$.

Augmented Training Data. We address the imbalance in the original dataset with our proposed data augmentation methods. Specifically, we augment positive label data in the training set only, leaving the validation and test sets unchanged to reflect the original distribution of labels. We use both levels of augmentation, including up to four assertion-level augmentations and one set-level augmentation for each positive data point. The resulting change in label ratios is summarized in Table 4.3. After data augmentation, we achieve closer to 50%-50% balance in labels for most designs and target assertions.

Table 4.4: Hyperparameter search space. Different sets of hyperparamters are considered depending on the Assertion Encoder architecture. GNN-related parameters are only relevant for the full LFPS model architecture which includes the Circuit Context Encoder module.

Hyperparameter	Values Considered
All Models	
Optimizer	AdamW
Batch Size	{64}
Learning Rate	{1e-5 ~ 1e-3}
Loss Function	{Cross Entropy, Focal Loss}
Classifier Head Layers	{1}
Classifier Dropout Rate	{0.0, 0.1}
Dropout	{0.0, 0.1}
MLP and LSTM-based Assertion Encoders	
MLP Layers	{2, 4}
Embedding Dim (d_{model})	{64 , 128, 256 }
Transformer-based Assertion Encoders	
Transformer Configs (d_{model}, Num Layers)	BERT-“Nano” = (64,1) BERT-Tiny (128, 2) BERT-Mini (256, 4)
GNN-based Circuit Context Encoders	
GCN Layers	{2, 4}
GCN Embedding Dim	{64 , 128, 256}

Chapter 5

Experiments

5.1 Next Helper Effectiveness Prediction

Baselines. We compare the full LPFS model performance against three baseline neural approaches. All three baselines do not contain the GCN-based CCE module for circuit graph encoding, and each use a different model architecture for the assertion encoder All models are implemented in Python using the Tensorflow library [1]. As discussed in Section 3.2, the Transformer AE operates on the entire token sequence, a concatenation of token sequences for the target assertion, assumed helpers, and the next to-be-added helper. The first token is fixed to be the classification token (`[CLS]`), from which the full assertion set embedding is read out. We use a maximum sequence length of 512, identical to BERT [13], with zero-padding. For the MLP and LSTM based AE models, each assertion is first encoded into a separate assertion embedding. As a result, a zero-padded token sequence of individual target and helper assertion are given to the MLP or a single-layer LSTM. For the MLP encoder, the outputs of the MLP encoder are mean-aggregated to produce the final assertion set embedding; for the LSTM encoder, each assertion embeddings are passed as a sequence to the next level LSTM as inputs. The LSTM output for the final assertion embedding, corresponding to the next helper to be added, is used as the assertion set embedding.

Training Setup. For each neural model, we perform an independent hyperparameter search—details about each hyperparameter space are listed in Table 4.4. In all approaches, we use the largest augmented dataset, where up to four assertion-level augmentations and one set-level augmentation are included for data points with positive labels. We also train all of the models to minimize either the binary cross entropy loss or focal loss [27] for up to 150 epochs with mini-batch size 64 and the AdamW [29] optimizer. For Transformer-based models, we use the Noam learning rate schedule [40] with one tenth of the total training steps as warm-up and a decay factor of 0.5; in all other models, a fixed learning rate schedule is used. All models are trained on a single Nvidia V100 GPU.

We evaluate the models and report their performances in three metrics—binary accuracy, precision, and recall—as shown in Table 5.1. Note that precision and recall are particularly important metrics in our evaluation as the test datasets can be highly imbalanced in labels. We make

Table 5.1: Accuracy of the LFPS model compared against baseline architectures on the task of autoregressive next helper effectiveness classification. The full model including both the GCN-based CCE and Transformer-based AE achieves the highest or close-to-highest accuracy, precision, and recall for all RTL designs.

Model	1R1W FIFO			OoO-Read Buffer			DMA Controller		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
MLP AE	0.698	0.711	0.804	0.769	0.764	0.967	0.925	0.667	0.053
LSTM AE	0.941	0.929	0.971	**0.982**	0.987	0.987	0.978	0.747	0.966
Transformer AE	0.946	0.957	0.948	**0.982**	0.984	**0.991**	0.987	0.969	**0.983**
Full LFPS Model (GCN CCE + Transformer AE)	**0.967**	**0.985**	**0.977**	**0.982**	**0.986**	0.990	**0.988**	**0.978**	0.981

three observations from the results. (1) As expected, accurately summarizing the sequential nature of token sequences representing each elaborated assertion text is critical to model performance. This is shown by comparing the performances of the MLP-based assertion encoders and the LSTM-based encoders. (2) Differentiating between the *context* segment, consisting of the target assertion and the already assumed helpers, and the *question* segment, referring to the next helper to be considered, further improves predictive performance. This is shown by the performance difference between the LSTM model and the Transformer encoder model, which incorporates segment embeddings as in BERT. (3) Finally, comparing the Transformer-based models with and without the circuit context encoder, we observe that providing information learned from the structural characteristics of the RTL graph can help improve predictions about helper effectiveness.

5.2 Ablation Study: Data Augmentation

We next perform an ablation study on how the predictive performance of LFPS model varies by the amount of augmented data used during training. Recall that we only augment the training dataset, not the test dataset, and only include augmentations for original data points with positive labels, as an effort to reduce label imbalance. The full LFPS model with both the GCN-based CCE and Transformer Encoder AE are trained, first with no augmentations, and then with varying sizes of augmented training datasets. Specifically, we allow up to two, three and four assertion-level augmentations per positive data point.

Figure 5.1 summarizes the results. For all RTL designs, we find that having even the smallest amount of augmentation significantly improves both accuracy and F1-score. Further improvements are observed as we increase the amount of augmented training data but at a slower rate. Note that baseline of no data augmentation is produced using other known techniques to improve model training over imbalanced data, e.g. focal loss [27] as the training loss function. Overall, we observe that addressing label imbalance directly with oversampling additionally contributes to improved performance. Increasing the number of augmentations further provides more training data leading to better training of larger model configurations considered in our hyperparameter space.

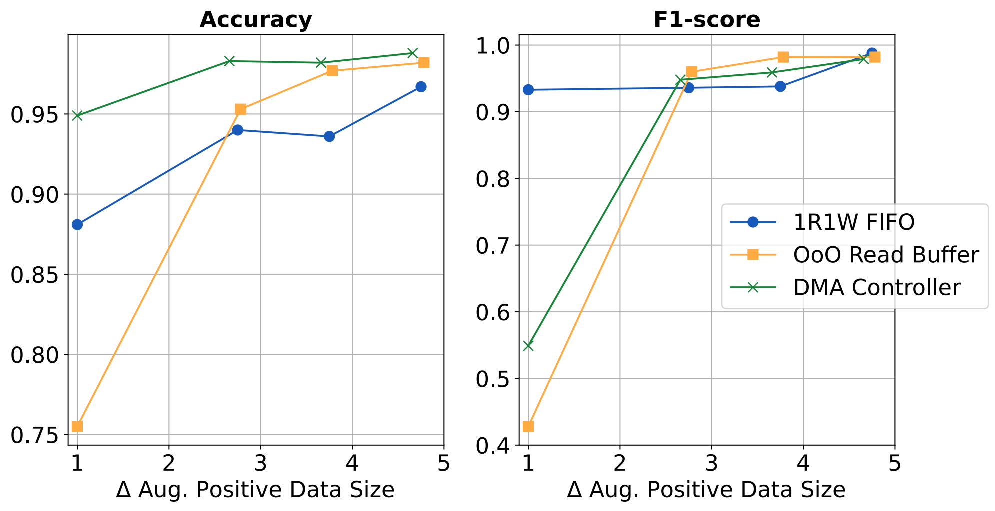


Figure 5.1: Binary accuracy and F1-score of the LFPS model trained with datasets of varying amounts of data augmentation. The horizontal axis refers to the normalized count of positive data points in the training set; the smallest value of 1 refers to the case without any augmentations and largest value close to 5× is the case with up to four assertion-level augmentations and one set-level augmentation per data point. Vertical axis represents the accuracy metrics; higher is better.

5.3 Searching for Effective Helper Subsets

In this section, we showcase a proof-of-concept implementation of neural model-guided search based on the trained LFPS model. In particular, we consider a simple beam search with the model used to guide the ranking of candidate subsets at each step. For a beam width of B, this beam search is initialized with an empty set of helpers $\mathcal{X}_{t=0} = \emptyset$ and a full set of candidates $\mathcal{X}$ for a given target $a_{\text{target}} \in \mathcal{T}$. At each time step t, we evaluate the conditional probability of all remaining helpers $Pr(y = 1; x \,|\, \mathcal{X}_t, a_{\text{target}}, \mathcal{G})$, $\forall x \in \mathcal{X} \setminus \mathcal{X}_t$ through model inference and choose the top-B candidates. Incrementally, each beam builds up the set of helpers $\mathcal{X}_t$ until: (1) the set contains a total of five helpers, or (2) the model predicts that none of the remaining helpers further improve helper set effectiveness.

We evaluate the performance of LFPS-guided beam search against vanilla random search that samples sets of five helpers without replacement. We take two cases of the beam search, using (1) the Transformer-based AE-only model and (2) the full LFPS model, including the GCN-based CCE, as the next helper effectiveness predictor. For both cases we use a beam width of $B = 5$. For a fair comparison, we allow random search to consider a total of five randomly sampled helper subsets (same number as beam width used) and take the best subset. As the metric of evaluation, we consider `pass@k`. Here, `pass@k` measures if the search retrieves a set of appropriate helpers that induces proof bound improvements when used as assumptions. As there are five target assertions, each a separate test case, `pass@k` is calculated the number of passing test cases out of five. For random sampling, we capture the variance in search performance by repeating the search over 20

Table 5.2: Evaluation of LFPS model-based beam search in finding effective helper subsets, compared against random sampling. The evaluation metric `pass@k` is calculated out of five test cases, corresponding to each of the five target assertions. LFPS-guided beam search outperforms random sampling for all values of *k* for 1R1W FIFO and OoO-Read Buffer, but underperforms for the DMA Controller due to limitations of the naive beam serach implementation.

Search Method		`pass@k`		
	k	1R1W FIFO	OoO-Read Buffer	DMA Controller
Random Search	1	0.24 ± 0.21	0.35 ± 0.21	0.24 ± 0.16
	3	0.47 ± 0.23	0.64 ± 0.15	0.39 ± 0.17
	5	0.55 ± 0.23	0.79 ± 0.15	**0.41** ± 0.15
Transformer AE-Only LFPS-Guided Beam Search	1	0.60	0.60	0.00
	3	0.60	0.60	0.00
	5	0.60	0.60	0.00
GCN CCE + Transformer AE LFPS-Guided Beam Search	1	0.60	**0.80**	0.20
	3	0.60	**0.80**	0.20
	5	**0.80**	**0.80**	0.20

iterations and reporting the average and standard deviation of measured `pass@k`. LFPS model-based beam search, on the other hand, is deterministic by implementation and retrieves the same top-*k* subsets every time.

Results on search method performance are shown in Table 5.2. For the first two designs, we observe that model-based beam search achieves a higher `pass@k` for all values of *k* considered compared to random. Our approach, however, does not perform well on the last case of DMA Controller. This result can be attributed to: (1) limitations of the naive beam search algorithm that is unable to explore candidates beyond the fixed top-*k* model predictions and (2) the scarcity of helper candidates that induce proof strengthening, as shown in the relatively low ratio of positive labels in the dataset collect for this RTL designs (See Table 4.3). Future work based on more advanced search algorithms, e.g. stochastic beam search [22], should be able to quickly overcome on this limitation.

Chapter 6

Related Work

Formal Techniques for Verification Complexity Reduction. Prior work proposed various approaches to reduce FV complexity via proof decomposition. Abstraction is one example [2, 10, 12] where the goal is to remove design details that are irrelevant to the property of interest. While some abstraction-based methods are simple to implement, e.g. reducing the size of a data bus or memory, more complex abstractions are prone introducing false errors and thus require significant domain expertise. More recently, automated helper mining has been actively explored [26, 17]. However, these works are usually restricted to a specific set of target properties and highly regular design structures [7], or rely on the compositional structure of the problem and need to run formal proofs, facing scalability issues [15]. Others works are able to produce a set of helper candidates that may be useful in FV [38] and rank them [32], but the ranking criteria is limited to the context of simulation-based verification and not applicable to FV. It remains a critical problem to quickly discern which helpers are appropriate for each target assertion, since using an inappropriate helper can fail to reduce, if not worsen, proof complexity.

Applications of ML to Formal Methods. Prior work that apply statistical or machine learning to improve formal methods have mostly focused on directly improving SAT/SMT solvers and automatic theorem provers [19, 24, 5, 41]. Our work is related to some of these works in that we also attempt to replace or augment parts of formal methods that have historically relied on heuristics. For example, [25] learns better heuristics to solve Quantified Boolean Formulas (QBFs) more efficiently with deep reinforcement learning (RL). However, there are substantial differences between this work and prior applications of ML: (1) in terms of scale, the hardware designs we discuss in this work are much larger and complex, by orders-of-magnitude, compared to the formulae and theorems considered in prior work; (2) this work proposes to automate proof strengthening through assume-guarantee decomposition, which is orthogonal to ideas for improving solvers and algorithms for theorem proving.

Chapter 7

Conclusion

In this work, we introduce the new task of learning to predict helper effectiveness, a major step towards fully automated formal proof decomposition. Our proposed neural architecture, LFPS, jointly learns representations of formal property specifications and circuit context to accurately predict the effectiveness of each additional helper. We further demonstrate that the trained model can be incorporated into a search algorithm and effectively serve as a fast evaluator of candidate helper subsets.

We envision several promising directions for future work. First, this work presents the potentials of learning about formal properties from natural language-like descriptions of their specification. A potential next step would be to assess if pre-trained language models [13, 20, 31, 8, 33] could be applied in our problem domain. Also, a neural approach that combines both steps of helper candidate auto-generation and evaluating helpers by their effectiveness in proof strengthening would be an interesting research direction, similar to prior work on program synthesis and neuro-symbolic methods [4, 3, 42].

Bibliography

[1] Martıén Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. Savannah, GA, USA. 2016, pp. 265–283.

[2] Zaher S. Andraus and Karem A. Sakallah. "Automatic Abstraction and Verification of Verilog Models". In: *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. San Diego, CA, USA: Association for Computing Machinery, 2004, pp. 218–223. ISBN: 1581138288. DOI: 10.1145/996566.996629. URL: https://doi.org/10.1145/996566.996629.

[3] Jacob Austin et al. "Program synthesis with large language models". In: *arXiv preprint arXiv:2108.07732* (2021).

[4] Matej Balog et al. "Deepcoder: Learning to write programs". In: *arXiv preprint arXiv:1611.01989* (2016).

[5] Kshitij Bansal et al. "Learning to reason in large theories without imitation". In: *arXiv preprint arXiv:1905.10501* (2019).

[6] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[7] Satrajit Chatterjee and Michael Kishinevsky. "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics". In: *Formal Methods in System Design* 40 (2012), pp. 147–169.

[8] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[9] Edmund M. Clarke et al. "Bounded model checking using satisfiability solving". In: *Formal methods in system design* 19 (2001), pp. 7–34.

[10] Edmund M. Clarke et al. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". In: *J. ACM* 50.5 (Sept. 2003), pp. 752–794. ISSN: 0004-5411. DOI: 10.1145/876638.876643. URL: https://doi.org/10.1145/876638.876643.

[11] Edmund M. Clarke et al. "Model checking and the state explosion problem". In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures* (2012), pp. 1–30.

[12] Dennis Dams and Orna Grumberg. “Abstraction and Abstraction Refinement”. In: *Handbook of Model Checking*. Cham: Springer International Publishing, 2018, pp. 385–419. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_13`. URL: `https://doi.org/10.1007/978-3-319-10575-8_13`.

[13] Jacob Devlin et al. “BERT: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[14] Harry Foster. “The 2022 Wilson Research Group IC/ASIC Functional Verification Treads,””. In: *White Paper. Wilson Research Group and Mentor, A Siemens Business* (2022).

[15] Mihaela Gheorghiu Bobaru, Corina S Păsăreanu, and Dimitra Giannakopoulou. “Automated assume-guarantee reasoning by abstraction refinement”. In: *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20*. Springer. 2008, pp. 135–148.

[16] Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. “You assume, we guarantee: Methodology and case studies”. In: *Computer Aided Verification: 10th International Conference, CAV’98 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*. Springer. 1998, pp. 440–451.

[17] Samuel Hertz, David Sheridan, and Shobha Vasudevan. “Mining hardware assertions with guidance from static analysis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.6 (2013), pp. 952–965.

[18] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[19] Sean B Holden et al. “Machine learning for automated theorem proving: Learning to solve SAT and QSAT”. In: *Foundations and Trends® in Machine Learning* 14.6 (2021), pp. 807–989.

[20] Aditya Kanade et al. “Learning and evaluating contextual embedding of source code”. In: *International conference on machine learning*. PMLR. 2020, pp. 5110–5121.

[21] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[22] Wouter Kool, Herke Van Hoof, and Max Welling. “Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 3499–3508.

[23] Thomas Kropf. *Introduction to formal hardware verification*. Springer Science & Business Media, 1999.

[24] Vitaly Kurin et al. “Can Q-learning with graph networks learn a generalizable branching heuristic for a SAT solver?” In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9608–9621.

[25] Gil Lederman et al. “Learning heuristics for quantified boolean formulas through deep reinforcement learning”. In: *arXiv preprint arXiv:1807.08058* (2018).

[26] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. “General LTL specification mining”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 81–92.

[27] Tsung-Yi Lin et al. “Focal loss for dense object detection”. In: *ICCV*. 2017, pp. 2980–2988.

[28] Yinhan Liu et al. “Roberta: A robustly optimized BERT pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[29] Ilya Loshchilov and Frank Hutter. “Decoupled weight decay regularization”. In: *arXiv preprint arXiv:1711.05101* (2017).

[30] Florent R. Madelaine and Barnaby Martin. “On the complexity of the model checking problem”. In: *CoRR* abs/1210.6893 (2012). arXiv: `1210.6893`. URL: `http://arxiv.org/abs/1210.6893`.

[31] Arvind Neelakantan et al. “Text and code embeddings by contrastive pre-training”. In: *arXiv preprint arXiv:2201.10005* (2022).

[32] Debjit Pal, Spencer Offenberger, and Shobha Vasudevan. “Assertion Ranking Using RTL Source Code Analysis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (2020), pp. 1711–1724. DOI: `10.1109/TCAD.2019.2921374`.

[33] Colin Raffel et al. “Exploring the limits of transfer learning with a unified text-to-text transformer”. In: *The Journal of Machine Learning Research* 21.1 (2020), pp. 5485–5551.

[34] Mike Schuster and Kaisuke Nakajima. “Japanese and korean voice search”. In: *2012 ICASSP*. IEEE. 2012, pp. 5149–5152.

[35] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural machine translation of rare words with subword units”. In: *arXiv preprint arXiv:1508.07909* (2015).

[36] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. “Checking safety properties using induction and a SAT-solver”. In: *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000 Austin, TX, USA, November 1–3, 2000 Proceedings 3*. Springer. 2000, pp. 127–144.

[37] Pashootan Vaezipoor et al. “Learning branching heuristics for propositional model counting”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 2021, pp. 12427–12435.

[38] Shobha Vasudevan et al. “GoldMine: Automatic assertion generation using data mining and static analysis”. In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. 2010, pp. 626–629. DOI: `10.1109/DATE.2010.5457129`.

[39] Shobha Vasudevan et al. “Learning semantic representations to verify hardware designs”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 23491–23504.

[40] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[41] Minchao Wu et al. “Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 9330–9342.

[42] Lisa Zhang et al. “Neural guided constraint logic programming for program synthesis”. In: *Advances in Neural Information Processing Systems* 31 (2018).