DETECTive: Machine Learning-driven Automatic Test Pattern Prediction for Faults in Digital Circuits

DETECTive: Machine Learning-driven Automatic Test Pattern Prediction for Faults in Digital Circuits

Vincenzo Petrolo
University of Illinois Chicago, USA
Politecnico di Torino, IT
vpetro4@uic.edu

Sourav Medya
University of Illinois Chicago, USA
medya@uic.edu

Mariagrazia Graziano
Politecnico di Torino, IT
mariagrazia.graziano@polito.it

Debjit Pal
University of Illinois Chicago, USA
dpal2@uic.edu

Abstract

Due to the continuous technology scaling and the ever-increasing complexity and size of the hardware designs, manufacturing defects have become a key obstacle in meeting end-user demand. Despite decades of research, traditional test-generation techniques often struggle to scale to massive and complex designs. Such scalability issues stem from the numerous backtracking the traditional test generation techniques perform before converging to a test pattern. In this work, we present DETECTive that leverages deep learning on graphs to learn fault characteristics and predict test pattern(s) to expose faults without requiring backtracking. DETECTive is trained on small circuits, and its learned knowledge is transferable to predict test patterns for circuits that contain up to 29× more gates than the training circuits. Since DETECTive avoids backtracking completely, it can predict test patterns up to 15× faster than academic tools and up to 2× faster than commercial tools. DETECTive achieves up to 100% pattern accuracy on synthetic designs and up to 95% test pattern accuracy on realistic designs. To our knowledge, DETECTive is the first to leverage deep learning to predict test patterns for digital hardware designs that can complement the traditional test generation techniques for faster design closure.

1 Introduction

The progressive technology miniaturization and the ever-increasing complexity of modern designs contribute to post-manufacturing defects. The widely used *stuck-at fault* model [9] represents interconnections fixed at specific logic values, potentially causing significant errors, especially in arithmetic circuits. To detect circuit faults, effective test patterns are required. These patterns must activate faulty behavior(s) in faulty circuits compared to defect-free ones and propagate this behavior to the circuit's output. However, there are 2^n possible test patterns in circuits with n inputs making exhaustive enumeration computationally inefficient.

To accelerate test pattern generation, D-Alg [16], PODEM [11], and FAN [10] assign Boolean values to inputs using heuristics. However, reconvergent fanout often triggers conflicts, causing the

algorithm to backtrack and revisit prior input assignments. The cumulative backtrack count can surpass hundreds of thousands for large circuits with hard-to-detect faults, impacting test pattern generation time. **Consequently, it is crucial to develop scalable and computationally efficient techniques for test generation**.

The success of Deep Learning (DL) in diverse scientific domains have led to investigations into its potential application in Automatic Test Pattern Generation (ATPG). Early attempts, such as in [7] for combinational circuits and later in [5] for sequential circuits, employed neural networks to model faulty gate-level netlists. However, these approaches face scalability issues for larger circuits. More recent techniques [17, 18] utilize Artificial Neural Networks (ANNs) to learn heuristics that guide the PODEM algorithm and reduce the frequency of backtracking. Nonetheless, these techniques still grapple with scalability challenges due to its coupling with the PODEM algorithm. Recently, Graph Neural Networks (GNNs) [13] have shown significant success in different tasks across various domains [6]. Due to the graph's natural alignment with circuit representations [8], GNNs have produced state-of-the-art results in multiple circuit-related design automation problems [20, 14]. In this work, we develop a scalable and computationally efficient DL model, DETECTive, using GNNs to *predict* test patterns for large digital circuits. DETECTive is designed to learn the activation and propagation of a stuck-at fault even if the circuit contains reconvergent fanout. It uses the learned knowledge to predict Boolean values for the circuit's inputs. The adoption of a predictive model for test pattern generation has two significant advantages – i) it eliminates the need for backtracking in the process and ii) it enhances runtime efficiency during test pattern prediction.

We summarize our main contributions as follows – (i) To our knowledge, we are the first to introduce the concept of *Automatic Test Pattern Prediction* (ATPP). We have also developed DETECTive, a fully DL-based proof-of-concept model to predict test patterns; (ii) We have developed a systematic technique to encode and learn fault type and its relation to fault's activation and propagation paths. To that extent, we identify design features that are highly relevant to test pattern prediction and applicable across hardware designs; (iii) We establish with empirical evidence that DETECTive's learned knowledge of test pattern prediction is transferable and can be applied to realistic designs that are 29× bigger than training designs. DETECTive can predict test patterns with up to 95% accuracy and runs 15× faster than academic tools and 2× faster than state-of-the-art commercial ATPG tools.

2 Preliminaries

Automatic Test Pattern Generation (ATPG). Errors may be induced in a chip during manufacturing for various reasons, including imperfection at the nanoscale fabrication. Among many such models that capture various errors, we consider those captured using stuck-at-fault models. A stuck-at-fault is an error where a wire between a pair of logic gates is always stuck at a specific logic value, either **0** or **1**. ATPG finds one or more test patterns, *i.e.*, assigns Boolean values to inputs that detect as many faults as possible on a chip. A test pattern needs to activate a fault followed by propagating the fault's effect to one or more visible outputs. For an n input design, the possible number of patterns are 2^n, making it infeasible to perform an exhaustive search. Researchers have proposed multiple ATPG algorithms and heuristics such as D-Alg [16], PODEM [11], and FAN [10] to accelerate the test pattern generation. Even after decades of research, finding a test pattern with high-fault coverage for industrial-scale designs is extremely difficult.

Despite incorporating heuristics to enhance its efficiency, ATPG encounters challenges when it produces an incorrect test pattern. In such instances, ATPG resorts to time-consuming backtracking, which involves undoing specific Boolean assignments and replacing them with new ones. This issue becomes particularly problematic when dealing with industrial-scale circuits, as the large search space can cause the backtracking process to become the limiting factor in the overall performance of the algorithm. This work aims to avoid backtracking during test pattern generation using DL techniques.

Graph Neural Networks. Graph neural networks (GNNs) use deep learning architectures, incorporate both graph topology and node attribute information to solve many tasks in various applica-

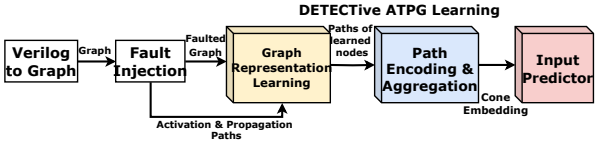
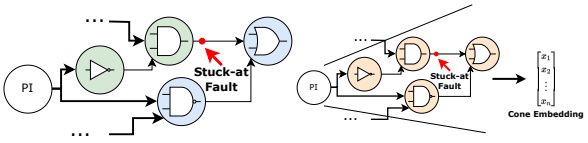


Figure 1: DETECTive workflow.



(a) (b)

Figure 2: (a): The Primary Input (PI) activates the stuck-at fault through the green path and propagates it through the blue path. (b): The Cone embedding contains both the paths.

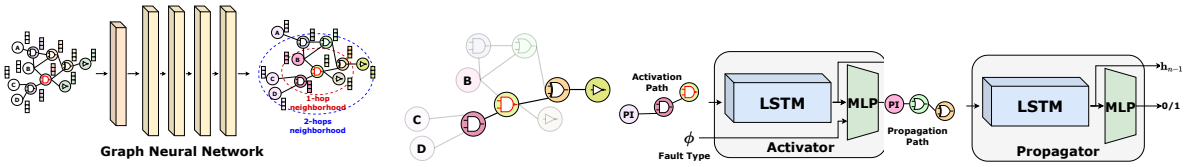
tion domains [20, 14]. Intuitively, GNNs map similar nodes nearby in a low-dimensional embedding space. GNNs operate by message passing between neighboring nodes and update the representation of a node by aggregating information from its neighbors. There exists various GNN models [13, 19] with two broad categories – *transductive* and *inductive*. While transductive models require the entire graph structure for training to produce node embedding vectors, inductive models can be applied to unseen graphs without retraining. In this work, we use an inductive GNN to predict test patterns.

Long-Short Term Memory. The Long-Short Term Memory (LSTM) model [12] is widely used in the context of Natural Language Processing to encode variable-length sentences. A LSTM is an improved version of a Recurrent Neural Network (RNN) designed to address the vanishing gradient problem in traditional RNNs. LSTMs are well-suited at capturing long-range dependencies in sequential data for task like time series forecasting. They use memory cells to store and retrieve information over extended time periods, facilitating the learning of complex sequential patterns. In our setting, each path is a variable-length gate sequence, and therefore we use LSTM for the *Path Encoding*.

3 ATPP as a Learning Problem

We propose to automatically predict test pattern(s) for a gate-level combinational circuit assuming the *stuck-at-fault* model. Formally, we define Automatic Test Pattern Prediction (ATPP) as ***Given a gate-level netlist C with I inputs, O outputs, and a single stuck-at fault f, DETECTive predicts test pattern(s) that can both excite and propagate f to at least one of the outputs of C.***

Workflow and insights: Figure 1 shows the workflow of DETECTive that has three different modules. Given a gate-level netlist, we construct a directed graph where each node of the graph represents a gate or a primary input (PI), and each directed edge represents an interconnect between a pair of gates or a gate and PIs. Then, we inject a stuck-at fault on a randomly selected wire. We annotate the appropriate graph edge with the fault and call that graph a *faulty graph*. The *Graph Representation Learning* module learns new representations (features) for each node in the faulty graph. As DETECTive predicts a Boolean value for each PI individually, it becomes crucial for the model to capture the impact of each PI on the fault detection. To this end, we extract the activation and propagation paths (c.f., Figure 2a) using the depth-first search (DFS) algorithm on the learned graph representations. We also provide an embedding (or representation) of the Cone of Influence (COI) shown in Figure 2b to the *Input Predictor*. The representation of COI is obtained via the *Path Encoding and Aggregation* module by using the representations of the important paths. This process re-iterates for all the primary inputs until the entire test pattern is built.



(a) (b) (c) (d)

Figure 3: (a): The final output from the GNN with one GAT [19] layer followed by GCN [13] layers is the learned representations from the initial graph structure and the node features.(b): A path of learned nodes carries information regarding the neighboring topology as well. (c): Activator module architecture. (d): Propagator module architecture.

4 Proposed Methodology: DETECTive

4.1 Overview of DETECTive architecture

Our goal is to learn how to automatically predict test patterns for a gate-level design. DETECTive consists of three modules – 1) Graph Representation Learning, 2) Path Encoding and Aggregation, and 3) Input Predictor. The primary objective of DETECTive is to process the input netlist and learn important features that involve domain-specific insights such as encoding of the activation and propagation paths. Figure 1 shows our proposed ATPP framework DETECTive.

4.2 Graph Representation Learning

In this module, we aim to learn features automatically to solve the ATPP problem. For a circuit containing a stuck-at fault, we focus on two critical aspects for our DL model – a) the circuit topology,and b) the fault activation and propagation paths that each PI leverages for fault identification.

Circuit as a graph. We model circuits as graphs where each gate is considered as a node, and the corresponding interconnections are the directed edges [8]. Further, each node (gate) has a feature vector that comprises three key components: i) the gate type, ii) a categorical variable indicating if the gate output is faulty, and iii) fan-out, *i.e.*, the number of connections to other gates at the output. This graph-based modeling helps us to obtain expressive representation of the circuit netlist by leveraging the power of the recent DL methods on graph data.

Learning circuit representation using GNNs. DETECTive uses a Graph Neural Network (GNN)-based architecture to create powerful and expressive representations using the circuit topology and initial node features. The goals of the GNN are to automatically learn a new set of feature vectors tailored to the ATPP problem, and to incorporate the neighboring circuit topology for each node in the feature vector (c.f., Figure 3a). To achieve the above-mentioned goal, we design a GNN-based architecture composed of a Graph Attention Network (GAT) [19] layer followed by Graph Convolutional Network (GCN) [13] layers with following objectives. The GAT layer computes a new representation for each node by aggregating the information from the embeddings of its neighbors with different attention weights as follows:

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j$$

where α is the learnable attention weight, $\mathbf{\Theta}$ is a set of learnable parameters, $\mathbf{x}$ are the feature vectors, and $\mathcal{N}(i)$ is the 1-hop neighborhood of the i^{th} node. Due to the attention mechanism, the representation of individual nodes focus on important neighbors (*e.g.*, the faulty ones in our case). After incorporating the importance of neighbors, we focus on the circuit topology in the representation learning process. To accomplish this, we take the enhanced representations obtained from the prior step and further feed them through multiple GCN layers. The objective here is to aggregate both the features from the neighborhood and the circuit topology to achieve an informative representation of each nodes:

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j\in\mathcal{N}(i)\cup\{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j\hat{d}_i}}\mathbf{x}_j$$

where $\mathbf{\Theta}$ is a set of learnable parameters, $e_{j,i}$ is the weight associated to the edge $i \to j$ and $\hat{d}_i$ (as well as $\hat{d}_j$) is the sum of all the weights of the edges. Thus, the new feature vector at node i is

computed as the sum of normalized neighboring feature vectors times a set of learnable parameters. The GNN layers allow to capture the complex feature relationship and the information over longer paths.

4.3 Path encoding and aggregation

As ATPG algorithms [16, 11, 10] heavily depend on activation and propagation paths to justify Boolean value assignments to PIs, we consider these paths as essential features for our model. However, the path lengths are variable. Therefore, one cannot use a multi-layer perceptron (MLP) for final prediction of Boolean values to each primary input. Thus, we first need a fixed size representation for all paths. We encode a path as a sequence of different number of nodes into a fixed size n-dimensional embedding in an Euclidean space. The LSTM [12] is highly effective to make predictions for sequential input data. Hence, we leverage LSTM to generate path embeddings. LSTMs possess three key attributes: i) the capacity to retain information from previous inputs, ii) the capability to process sequences of varying lengths, and iii) the ability to produce a unified encoding (or representation) for the entire input sequence. Therefore, given a path of length n, we formulate the path sequence as follows: $\mathbf{P} = [\mathbf{p}_0, \mathbf{p}_1, ...\mathbf{p}_{n-1}]$; where each $\mathbf{p}_i$ is the embedding of each learned node of the GNN module (c.f., Figure 3b). For each $\mathbf{p}_i$, the LSTM computes a hidden state (*i.e.*, memory) $\mathbf{h}_i$ which is used for computing the next hidden state $\mathbf{h}_{i+1}$ of the element next in the sequence with $\mathbf{h}_{i+1} = f(\mathbf{p}_i, \mathbf{h}_i)$, where f is the learnable function by LSTM. Subsequently, f computes the final state encoding $\mathbf{h}_{n-1}$ as a representation of the entire path $\mathbf{P}$.

Activator and Propagator. The activation and propagation of a fault are critical in detecting the fault (c.f., Section 2). Consequently, in DETECTive, we have designed dedicated DL components aimed at learning these two tasks with two separate modules each utilizing a different LSTM. These modules are referred to as the Activator and the Propagator shown in Figure 3c and 3d, respectively.

The Activator aims to predict a Boolean value for a PI such that the fault is activated. It takes an activation path and the fault type, and performs a binary classification task as follows:

$$\mathbf{y}_{act} = \text{Softmax}(\text{MLP}(\text{LSTM}(\mathbf{P})||\phi))$$

where the LSTM computes an encoding for the given activation path $\mathbf{P}$. This is further concatenated with the fault-type ϕ to generate a final vector which is fed through a MLP to predict the Boolean value assignment to the PI. The Softmax is a non-linear function that transforms the MLP output in a probability $\mathbf{y}_{act}$ of either being **0** or **1**. The Propagator module is similar to the Activator. The module only takes the propagation path as input as the fault type is irrelevant for propagation. We train DETECTive end-to-end with the Cross-Entropy loss function. Consequently, the two MLPs learn their respective tasks and generate the representation of the paths.

4.4 Input Predictor

In our model, we construct the final Input Predictor module to predict a Boolean value for a PI. To achieve this, we aggregate all the information regarding the activation and propagation paths from the previous modules. We refer to this as the cone embedding ($\mathbf{C}$) and compute it as follows:

$$\mathbf{C} = (\|_{i=0}^{p-1} (\mathbf{x}_i^{act} \parallel (y_i^{act})) \parallel (\|_{i=0}^{p-1} (\mathbf{x}_i^{prop} \parallel y_i^{prop}))$$

where $\parallel$ is the concatenation operator, $\mathbf{x}_i^{act}$ is the encoding of the i^{th} activation path and y_i^{act} is the Activator's prediction to be assigned to the PI. Note that, if there are less than p activation paths, the i^{th} vectors are replaced with zeros (for experiments, we have chosen $p = 10$). The cone embedding $\mathbf{C}$ contains all the path encodings as well as the respective predictions made from the Activator and Propagator. Subsequently, the cone embedding is fed into a deep neural network (*i.e.*, an MLP) which assigns a Boolean value to a PI using a Sigmoid function, $y = \text{Sigmoid}(\text{MLP}(\mathbf{C}))$, where the Sigmoid adapts the output of the MLP in a range between 0 and 1.

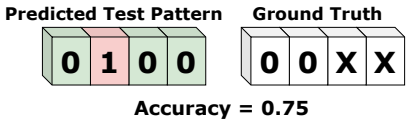


Figure 4: An example of our accuracy metric.

4.5 Our learning task

Predicting a test pattern is an iterative procedure where DETECTive independently assigns a Boolean value to each PI of the circuit under analysis. Ideally, we want the predicted pattern to be as close as possible to the ground truth. Therefore, we formulate the learning task as a minimization problem where we minimize the Binary Cross Entropy loss function which quantifies the difference between the predicted pattern and the ground truth. The loss function is:

$$\mathcal{L} = \sum_{i=0}^{N-1} \text{BCE}(T_i, f_\theta(G, \phi, A_i, P_i))$$

where f_θ is our model DETECTive, G is the graph for the circuit under analysis, ϕ represents the fault type, A_i is a set of activation paths, and P_i is a set of propagation paths. DETECTive predicts the value to be assigned to the i^{th} input and computes the loss $\mathcal{L}$ using the corresponding bit in the ground truth test pattern T_i. Since the task requires a binary prediction (*i.e.*, an input could be either **0** or **1**), we use the BCE loss function. Our end goal is to minimize the loss $\mathcal{L}$ by optimizing the model parameters θ.

5 Experimental Setup

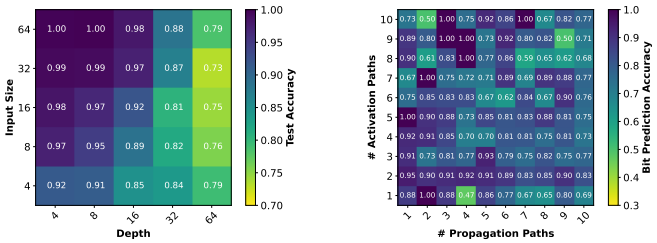
Dataset generation: To train DETECTive we need substantial amount of data involving fault location, fault activation, and fault propagation paths. To this end, we develop a parameterized *Random Circuit Graph Generator* (RCGG) that accepts *number of inputs*, *number of outputs*, *maximum fan-in and fan-out*, *maximum path depth*, and *types of gates as input* and outputs a combinational circuit as a directed graph object and a Verilog netlist. Next, we synthesize the generated netlist using Yosys [4] and use ABC [1] for the technology mapping. Without loss of generality, we restrict ABC to use only NOT and 2-input NAND gates during technology mapping. Our dataset contains 2,000 combinational circuits where each circuit has four inputs, one output, and a depth between 4 and 30. We replicate each circuit multiple times and randomly inject exactly one fault at different location. This produces training circuits with diverse activation and propagation paths.

Ground truth extraction: In ML parlance, ground truth is used to quantify the quality of the trained model and the predicted output. In the context of ATPP, the task of ground truth extraction is harder as there may exist multiple test patterns (ground truths) that can activate and propagate the same fault. To avoid model bias, we extract all the test patterns for a given fault using ATALANTA [2].

Training parameters: We train DETECTive on the 80% of the dataset and validate on the remaining 20% dataset. We use 1000 epochs and a learning rate of 0.001, and select the model parameters with the highest validation accuracy. Each module (*e.g.*, GNN, LSTM, MLP) has the hidden dimension of 32.

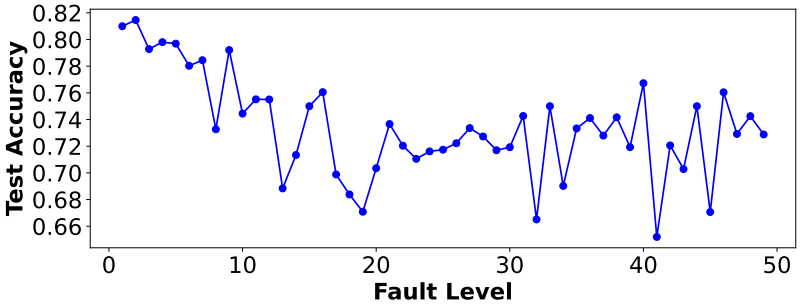
Evaluation platform: We train DETECTive on NVIDIA Tesla P100 GPUs and Intel Xeon E5-2650 @2.20 GHz.

Evaluation metrics: We train DETECTive to generate a single test pattern that exposes the designated stuck-at fault, prioritizing it rather than aiming for broader fault coverage. Considering the possibility of having multiple test patterns for a single stuck-at fault, we compute our model's accuracy by selecting the test pattern that exhibits the closest similarity to the predicted test pattern. The accuracy metric is determined through a bit-wise comparison of the two test patterns. In Figure 4, all bits in predicted test pattern (on left) except the third bit from right matches with ground truth (on right). Hence, the prediction accuracy is $3/4 = 0.75$.

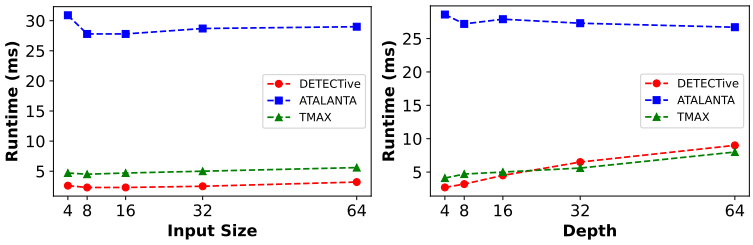


(a) (b)

Figure 5: (a) Test accuracy on faulty circuits with different dimensions. (b) Bit-level prediction accuracy with increasing reconvergency for activation and propagation tasks.



(a)



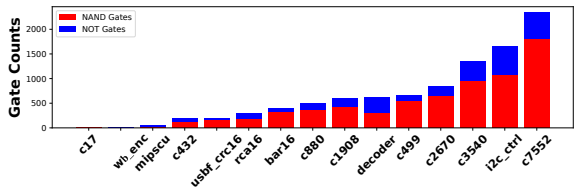
(b) (c)

Figure 6: (a) Test accuracy for deeper faults. Runtime comparison between DETECTive, ATALANTA, and Synopsys TMAX with (b) increasing inputs and (c) increasing depth.

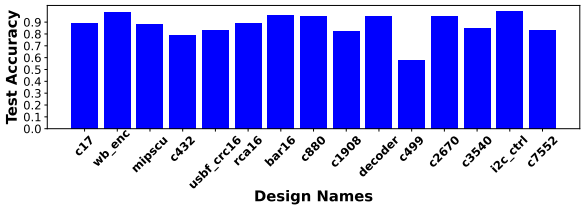
6 Experimental Results

6.1 Accuracy with varying circuit parameters

In this experiment, we investigate effect of the number of inputs and circuit depth on DETECTive's test pattern prediction accuracy. We use circuit datasets with number of inputs and depth varying as 2^N with $N = 2, 3, \ldots, 6$ and inject a stuck-at-fault randomly per circuit. Figure 5a shows average prediction accuracy per dataset. We observe that as the number of inputs increases (Y-axis), DETECTive's test pattern prediction accuracy progressively increases. DETECTive achieves pattern accuracy of up to 100% (on average greater than 90%) on all datasets. This is due to the fact that a significant fraction of the test patterns assign *don't care* values (*i.e.*, **x**) to most of the inputs and only sets either **0** or **1** to only a few inputs for fault detection, thus simplifying the prediction process. On the other hand, as circuit depth increases (X-axis), DETECTive's accuracy progressively decreases. This is due to the LSTM in the Activator and Propagator modules which lose precision in the path embedding task after a specific path length. Such loss in precision in path embedding makes the pattern prediction task harder for the input predictor module, which heavily



(a)



(b)

Figure 7: (a) Number of gates for each Verilog design used for testing. (b) Test accuracy for realistic Verilog benchmarks.

relies on those embedding to make the final prediction. We also observe a similar loss in accuracy as we scale circuit by increasing the input size and the circuit depth. **This experiment shows that DETECTive is a promising approach to predict test patterns for circuits.**

6.2 Accuracy in presence of reconvergent fanout

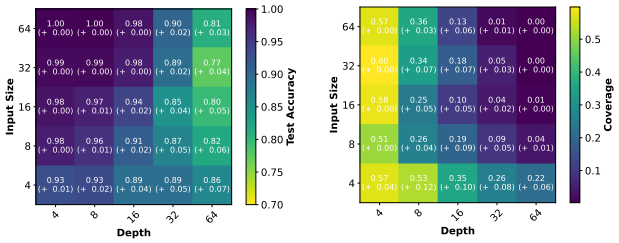
In this experiment, we investigate DETECTive's prediction accuracy in the presence of multiple activation and propagation paths. Such paths constitute a reconvergent fanout, often leading to conflicting input assignments resulting in backtracking. We generate a dataset containing 10,000 circuits of fixed depth and fixed input size. We consider up to 10 activation paths and 10 propagation paths when measuring the accuracy of the test patterns predicted by DETECTive. Since our prediction is at bit-level, instead of considering word-level accuracy, we consider bit-level accuracy of test patterns. Figure 5b shows test pattern accuracy as a function of number of activation and propagation paths. We observe that with a progressive increase in the number of activation paths and propagation paths, the test pattern prediction accuracy progressively decreases from 88% to 68%. With more paths, the task of Boolean value assignment to inputs becomes increasingly difficult for the input predictor. **This experiment shows that DETECTive is capable of predicting highly accurate test patterns in the presence of reconvergent fanout**.

6.3 Accuracy with varying fault location

In this experiment, we investigate how the fault location affects DETECTive's test pattern prediction accuracy. To this end, we generate circuit datasets containing circuits of an input size of four and a depth of 64 and inject faults at levels between 1 and 50. The results in Figure 6a shows that the test pattern accuracy progressively decreases as the fault location gets closer to the output. This is because a deeper fault location requires a longer activation path which increases the difficulty of the path embedding task. **This experiment shows that DETECTive is reasonably effective in predicting test patterns for both shallow and deep faults**.

6.4 Test pattern prediction runtime comparison

In this experiment, we compare the inference runtime of DETECTive with ATALANTA [2] and Synopsys TMAX [3]. It's worth noting that tools like ATALANTA and TMAX typically do not encounter runtime challenges with circuits of this scale. Nevertheless, the comparison provides valuable insights into the relative efficiency of DETECTive's DL model in this context, offering a perspective on potential runtime improvements compared to traditional tools. First, we compare



(a) (b)

Figure 8: (a): Ensemble model improvement compared to Figure 5a. (b): Average test patterns coverage of ensemble model.

runtime for a fixed-depth circuit with increasing number of inputs. We fix the circuit depth at 4 and show runtime comparison in Figure 6b. Since DETECTive iteratively predicts test patterns, we observe a progressive increase in DETECTive's runtime starting at 2.6 ms for a 4-bit test pattern to 3.2 ms for a 64-bit test pattern. When compared with ATALANTA and TMAX, we achieve a runtime improvement of up to 15× and 2×, respectively.

In the second experiment, we compare runtime for fixed-size input circuit with increasing depth. We fix the input size to 4. Figure 6c shows the runtime comparison for fixed-size input circuit. We observe that DETECTive is 10× faster than ATALANTA and comparable to TMAX. The runtime increase with progressive depth increase is due to the longer time taken by LSTM to compute embedding of longer paths. Moreover, note that ATALANTA and TMAX are compiled tools whereas DETECTive is implemented in Python and thus incurs overhead introduced by the Python interpreter. **This experiment shows that DETECTive is computationally efficient when predicting test patterns and often faster or comparable to the state-of-the-art ATPG tools**.

6.5 Knowledge transferability to real designs

In this experiment, we evaluate DETECTive's effectiveness on realistic designs in terms of pattern prediction accuracy. Figure 7a shows gate counts of our realistic designs. Figure 7b shows prediction accuracy of DETECTive for the realistic designs. We observe a slight decrease in the accuracy as the circuit size progressively increases. It is worth noting that in spite being trained on synthetic designs, DETECTive predicts test patterns with high accuracy for realistic designs that contain up to 29× more gates than synthetic designs. **This experiment shows that DETECTive learns to predict test patterns and its learned knowledge is transferable, making it a promising approach for real-world adoption**.

6.6 Accuracy and coverage via ensemble models

For a stuck-at fault, traditional ATPG might find multiple test patterns to detect it. However, DETECTive can predict only one test pattern at once. Since we *predict* as compared to *generation* of traditional ATPG, there exists a chance that the test pattern may be unable to detect a fault. Correspondingly, we create an ensemble $\mathbb{E}$ of DETECTive models trained with different seeds and ground truth test patterns to increases the chances of detecting a fault. We evaluate $\mathbb{E}$ on the same test dataset of Section 6.1 in terms of test pattern accuracy and coverage. In Figure 8a we observe improved test pattern accuracy for almost all circuit sizes. This is due to the presence of multiple DETECTive instances in $\mathbb{E}$ which together predict more accurate set of test patterns. When compared to Figure 5a, the average test pattern accuracy improves by up to 7%. Next, we evaluate if $\mathbb{E}$ produces any additional test patterns compared to the single-instance DETECTive of Section 6.1. We measure the coverage increase as the number of newly predicted test patterns by $\mathbb{E}$ expressed as a fraction of ground truth test patterns. Figure 8b shows the results. The ensemble $\mathbb{E}$ improves the test pattern coverage by up to 12%. **This experiment shows that ensembled DETECTive model can substantially increase the likelihood of predicting a correct test pattern**.

7 Limitations

Further exploration of enhanced path encoding techniques is required to broaden DETECTive's applicability. Despite achieving high accuracy in test pattern prediction for many circuit configurations, DETECTive exhibits certain limitations when dealing with deeper fault locations and reconvergent fanout scenarios, which are well-known challenges for traditional ATPG tools. This behavior is related to the creation of path embeddings through LSTM networks. While ensemble models can improve accuracy, they are unable to address this underlying issue. Despite these limitations, DETECTive is promising for rapid prediction of test patterns.

8 Related work

Recently, DL has been applied to various challenging design automation problem including ATPG. Chakradhar et al. pose ATPG as an energy minimization problem and employ ANN for test pattern generation [7].Later, they extended it to sequential circuits [5]. Roy et al. [17, 18] propose a neural network-enhanced PODEM algorithm which improves decision-making and avoids global energy minima computation. Pan et al. [15] use Reinforcement Learning to generate test vectors for delay-based Trojan detection. In contrast, DETECTive is a standalone framework that uniquely combines graph learning and LSTM-based sequence learning to predict test patterns. It's important to highlight that we intentionally refrain from comparing DETECTive to above-mentioned works as they are tailored for test pattern generation whereas DETECTive focuses on a predictive modeling a distinction crucial for a fair evaluation.

9 Conclusion

We present DETECTive, a fast, scalable, and accurate DL model to predict test patterns. These patterns can be used to detect faults and as seed patterns to traditional ATPG tools.

References

[1] Accessed: September 4, 2024. ABC. `https://people.eecs.berkeley.edu/~alanmi/abc/`.

[2] Accessed: September 4, 2024. Atalanta. `https://github.com/hsluoyz/Atalanta`.

[3] Accessed: September 4, 2024. Synopsys TestMAX ATPG. `https://www.synopsys.com/implementation-and-signoff/test-automation/testmax-atpg.html`.

[4] Accessed: September 4, 2024. Yosys Open SYnthesis Suite. `https://yosyshq.net/yosys/`.

[5] J. Bannino, J.-F. Santucci, and D. Floutier. 1996. Hybrid neural model for automatic test pattern generation. *Int'l Conf. on Electronics, Circuits, and Systems (ICECS)* 1 (1996), 259–262 vol.1.

[6] H. Cai, V. W. Zheng, and K. Chang. 2018. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Trans. on Knowledge and Data Engineering (TKDE)* 30, 09 (2018), 1616–1637.

[7] S.T. Chakradhar, M.L. Bushnell, and V.D. Agrawal. 1988. Automatic test generation using neural networks. *Int'l Conf. on Computer-Aided Design (ICCAD)* (1988), 416–419.

[8] Pal D., Ma S., and Vasudevan S. 2020. Emphasizing Functional Relevance Over State Restoration in Post-Silicon Signal Tracing. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39, 2 (2020), 533–546.

[9] Richard D. Eldred. 1959. Test Routines Based on Symbolic Logical Statements. *J. of ACM (JACM)* 6, 1 (1959), 33–37.

[10] Hideo Fujiwara and Takeshi Shimono. 1983. On the Acceleration of Test Generation Algorithms. *IEEE Trans. on Computers (TC)* C-32, 12 (1983), 1137–1144.

[11] Prabhakar Goel. 1981. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Trans. on Computers (TC)* C-30 (1981), 215–222.

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation (NC)* 9, 8 (1997), 1735–1780.

[13] Thomas N Kipf and Max Welling. 2016. Semi-supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016).

[14] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu. 2019. High Performance Graph Convolutional Networks with Applications in Testability Analysis. *Design Automation Conf. (DAC)* (2019), 1–6.

[15] Z. Pan, J. Sheldon, and P. Mishra. 2020. Test Generation using Reinforcement Learning for Delay-based Side-Channel Analysis. *Int'l Conf. on Computer-Aided Design (ICCAD)* (2020), 1–7.

[16] T. Paul Roth, Willard G. Bouricius, and Peter R. Schneider. 1967. Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits. *IEEE Trans. on Computers (TC)* EC-16, 5 (1967), 567–580.

[17] Soham Roy, Spencer K. Millican, and Vishwani D. Agrawal. 2021. Training Neural Network for Machine Intelligence in Automatic Test Pattern Generator. *Int'l Conf. on VLSI Design (ICVD)* (2021), 316–321.

[18] Soham Roy, Spencer K. Millican, and Vishwani D. Agrawal. 2022. Multi-Heuristic Machine Intelligence Guidance in Automatic Test Pattern Generation. *Microelectronics Design and Test Symp. (MDTS)* (2022), 1–6.

[19] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph Attention Networks. *Int'l Conf. on Learning Representations (ICLR)* (2018).

[20] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H. S. Lee, and S. Han. 2020. GCN-RL Circuit Designer: Transferable Transistor Sizing with Graph Neural Networks and Reinforcement Learning. *arXiv preprint arXiv:2005.00406* (2020), 1–6.