Leveraging Graph Attention Networks for Formal Verification of Hardware Description Languages

Sanjay Ravindran
UNC Chapel Hill
Hanes Hall, 318 E Cameron Ave #3260, Chapel Hill, NC 27599
sravindran1@unc.edu

Jondash Karamvruc
UNC Chapel Hill
Hanes Hall, 318 E Cameron Ave #3260, Chapel Hill, NC 27599
jkaramvruc@unc.edu

Nimil Patel
UNC Chapel Hill
Hanes Hall, 318 E Cameron Ave #3260, Chapel Hill, NC 27599
nrpatel2@unc.edu

Abstract

This paper explores the application of Graph Attention Networks (GATs) to the formal verification of security properties in hardware description languages (HDLs). Leveraging GraphCodeBERT, we created graph representations of HDL code, with nodes representing modules, variables, and assertions, and edges capturing control and data flow dependencies. These graphs were used to train a GAT to prioritize critical interactions using hierarchical attention mechanisms. We evaluated our model on a newly developed dataset of 1,000 assertions generated from HDL code. While the model achieved promising training and validation results in terms of recall, precision, and F1 score, it ultimately overfitted to the training data, resulting in poor generalization to the test set. Our research underscores the main contribution of creating a unique dataset and framework to represent HDL code as graphs, with a pathway for improving verification techniques through advanced graph learning. We provide suggestions for future improvements to better balance the data and mitigate overfitting effects, paving the way for enhanced hardware verification methods.

1 Introduction

Introduction The formal verification of hardware designs is crucial in ensuring the reliability and security of digital systems. With the increasing complexity of hardware description languages (HDLs) used in designing critical components, there is a pressing need for automated methods to verify the correctness of these designs. This research focuses on leveraging Graph Attention Networks (GATs), combined with transformer-based embedding methods, to address the challenge of formal verification of security properties in HDL. The core idea is to represent hardware designs as graphs where nodes capture critical HDL elements, such as modules, variables, and assertions, while edges represent various types of dependencies, such as control flow and data flow. This graph representation is enriched by using GraphCodeBERT embeddings, and the GAT is used to learn relationships and prioritize critical connections within the hardware design. The use of GATs for formal verification

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

presents a novel approach, exploiting attention mechanisms to efficiently navigate and understand the logical structure of HDL, focusing on critical nodes and interactions, ultimately aiming to improve verification accuracy. Despite facing challenges, such as limited data availability and the complexity of graphical representations, our approach makes significant contributions in establishing a novel dataset, processing HDL code into graph structures, and creating a methodology for learning critical relationships within hardware design data. While the model's performance indicated overfitting and biased classification, this work lays the foundation for future advancements by pioneering hierarchical attention in GATs for hardware verification. Abstract This paper explores the application of Graph Attention Networks (GATs) to the formal verification of security properties in hardware description languages (HDLs). Leveraging GraphCodeBERT, we created graph representations of HDL code, with nodes representing modules, variables, and assertions, and edges capturing control and data flow dependencies. These graphs were used to train a GAT to prioritize critical interactions using hierarchical attention mechanisms. We evaluated our model on a newly developed dataset of 1,000 assertions generated from HDL code. While the model achieved promising training and validation results in terms of recall, precision, and F1 score, it ultimately overfitted to the training data, resulting in poor generalization to the test set. Our research underscores the main contribution of creating a unique dataset and framework to represent HDL code as graphs, with a pathway for improving verification techniques through advanced graph learning. We provide suggestions for future improvements to better balance the data and mitigate overfitting effects, paving the way for enhanced hardware verification methods.

2 Methodology

2.1 Data Preparation and Graph Construction

The initial step in our methodology involved transforming HDL code into a format that is suitable for analysis using Graph Attention Networks (GATs). We utilized two preprocessing pipelines to convert hardware description language (HDL) code into graph representations. These pipelines generated graph-structured data from HDL code and associated assertions, each node representing HDL elements such as modules, variables, and control constructs, while edges captured various dependencies such as data flow, control flow, and state transitions.

- **Simple Preprocessing Pipeline:** The first preprocessing approach focused on a relatively straightforward transformation. It parsed HDL code, removed comments, and identified critical HDL elements like module declarations, variable assignments, and control flow statements. Each of these elements was represented as a node. Assertions were also included as separate nodes, allowing the model to directly learn the relationship between the HDL code and the corresponding assertion. Attention points were identified based on markers in the code, and these were used to assign weights to nodes that likely held more importance in the verification process.
- **Advanced Preprocessing Pipeline:** The second preprocessing pipeline added more sophistication by identifying SystemVerilog-specific constructs, such as state machines, variable operations, and condition blocks. In this approach, nodes represented various constructs, including state transitions, variable updates, and conditional checks. In addition to nodes, edges were annotated with specific relationship types to represent different dependencies. We defined six different edge types, including data flow, control flow, state dependencies, conditional dependencies, update dependencies, and special attention edges, to provide a more nuanced understanding of how HDL elements interact. These graphs were enriched with the use of GraphCodeBERT, which provided embeddings for each node, effectively capturing the semantic and contextual meaning of each HDL element.

2.2 Graph Embeddings and Feature Extraction

For both preprocessing approaches, we leveraged the pre-trained GraphCodeBERT model to generate embeddings for each node. These embeddings were used as node features, encapsulating the syntactic and semantic characteristics of each HDL element. GraphCodeBERT, being a transformer-based model, allowed us to encode rich contextual information, thereby providing the GAT with more meaningful representations for learning. Attention weights were also applied to node embeddings

to emphasize nodes marked as significant based on specific markers. The hierarchical attention mechanism thus helped the GAT focus on critical areas of the code more effectively.

2.3 Graph Attention Network Architecture

The core of our approach was a hierarchical attention mechanism implemented using a three-layer GAT. The GAT was designed to process the graph representations, learn the relationships between nodes, and classify whether a given assertion would pass or fail. Our GAT utilized several key components:

- **Three-Layer Structure with Skip Connections:** The GAT had three layers, with skip connections between them. These skip connections helped to avoid gradient vanishing and allowed information to bypass certain layers, effectively maintaining rich feature propagation.
- **Multi-Head Attention:** We used multi-head attention with 12 attention heads to learn diverse representations of node neighborhoods, thereby providing the model with a more comprehensive understanding of the various relationships present in the HDL code.
- **Hidden Dimensions:** Each layer of the GAT operated on a hidden dimension size of 768, which provided a sufficient feature space to represent the complex interactions present in HDL code.

2.4 Training Strategy

We employed cross-entropy loss for training, with the target of predicting whether each assertion passed or failed. We split the dataset into training, validation, and testing sets, using a 70-15-15 ratio. We used the Adam optimizer with a learning rate of 0.0005 and applied a learning rate scheduler to reduce the learning rate upon plateauing validation loss. We also implemented early stopping with a patience of 20 epochs to prevent overfitting. Training was conducted on a GPU to handle the computational requirements of the GAT and the large hidden layer size.

2.5 Evaluation Metrics

To assess the model's performance, we used standard metrics, including accuracy, precision, recall, and F1 score. Precision and recall were particularly significant in our evaluation, given the goal of minimizing false positives and negatives in the verification process. The F1 score provided a balanced metric to understand the overall performance. The model was evaluated on its ability to correctly classify assertions based on the HDL graph representation. Both versions of the preprocessing pipeline were tested, and their performance was compared in terms of accuracy and the robustness of the learned representations. This methodology provided a systematic approach to transform HDL code into a form suitable for deep learning analysis, and the hierarchical attention mechanism enabled the model to prioritize critical relationships effectively.

3 Results

3.1 Training and Validation Performance

Simple Preprocessing Pipeline: For Validation Accuracy the model achieved a validation accuracy of approximately 0.70. This preprocessing pipeline produced graph representations with simpler node structures and fewer node types, focusing on HDL elements such as module declarations and assignments. For Precision and Recall both precision and recall metrics were notably high, resulting in an F1 score of approximately 0.70. This indicates that the model was able to consistently identify both true positives and true negatives, capturing meaningful relationships between HDL code and assertions. Regarding Training vs. Validation Loss the training and validation losses were relatively stable, with slight decreases, suggesting the model was prone to overfitting despite showing limited improvement over time. The gap between training and validation losses indicated that the model was fitting too closely to the training data and struggled to generalize effectively.
Advanced Preprocessing Pipeline: In terms of Validation Accuracy using the more sophisticated preprocessing pipeline, the model reached a validation accuracy of 0.56. This lower accuracy compared

to the simpler approach indicates that the added complexity of representing multiple relationships, such as various edge types and SystemVerilog-specific constructs, led to more challenging training dynamics, possibly overwhelming the model. Considering Precision and Recall, precision and recall were slightly lower than those achieved with the simpler pipeline, resulting in an F1 score of approximately 0.60. This suggests that while the model was still able to find true positives and true negatives, the added complexity in the graph representation diminished the overall performance. Lastly, the validation loss was consistently low and did not fluctuate much, which indicates that the model managed to generalize the learned relationships without drastic overfitting. However, the inability to improve loss over time suggests that the model may have been trapped in a local minimum, limiting its learning capacity.

3.2 Comparison of Preprocessing Pipelines

The simpler preprocessing pipeline yielded better validation results compared to the advanced pipeline, with a validation accuracy of 0.70 versus 0.56 for the advanced version. This suggests that the simpler graph structures with fewer edge types provided a more effective learning space for the GAT, reducing the risk of overfitting and making training more tractable. However, both pipelines showed high precision and recall, indicating that the model could consistently detect correct and incorrect assertions, regardless of the added complexity from the advanced pipeline. Despite the promising validation results, the overfitting issue became evident when evaluating on the test set, with the accuracy dropping to just above 50 percent. This generalization problem underscores the limitations of the dataset and the model's tendency to memorize rather than learn generalized relationships.

3.3 Insights from Training and Testing Behavior

The stable validation loss observed throughout training, especially for the advanced preprocessing pipeline, indicates that the model struggled to improve beyond a certain point. This was likely due to the increased complexity of the graph representation, which the GAT architecture could not fully capture effectively. The consistency in validation accuracy, despite the introduction of early stopping, also suggests the need for more robust overfitting mitigation techniques, such as stronger regularization or more effective sampling methods. Overall, while the hierarchical attention mechanism enabled the GAT to capture relationships between HDL elements and assertions, the simpler graph representation was more effective in maintaining higher validation accuracy. The advanced preprocessing pipeline, while providing a more nuanced view of the HDL code, introduced complexity that ultimately hindered the model's learning and generalization capability.

The testing phase revealed a mixed set of results, highlighting the strengths and limitations of the proposed model. On simpler designs, the model achieved excellent performance with an accuracy of 0.95, a ROC AUC of 0.95, and a PR AUC of 0.9655 on an unseen test set. These results demonstrate the model's ability to generalize effectively to straightforward hardware designs, successfully distinguishing between passing and failing assertions.

However, when tested on more complex designs, the accuracy dropped significantly to 50 percent, and other metrics such as ROC AUC and PR AUC also declined. This performance drop underscores the limitations of the model when handling intricate HDL structures with higher complexity. The decrease in performance suggests that while the model captures relationships effectively in simpler scenarios, it struggles to generalize to more challenging designs, likely due to the increased graph complexity and the limitations of the current GAT architecture in modeling such intricate dependencies.

4 Main Contributions

4.1 Dataset Creation

- Developed a novel dataset specifically for the formal verification of HDLs using Graph Neural Networks (GNNs).
- Included rich, assertion-level annotations to capture both passing and failing assertions, focusing on security properties.

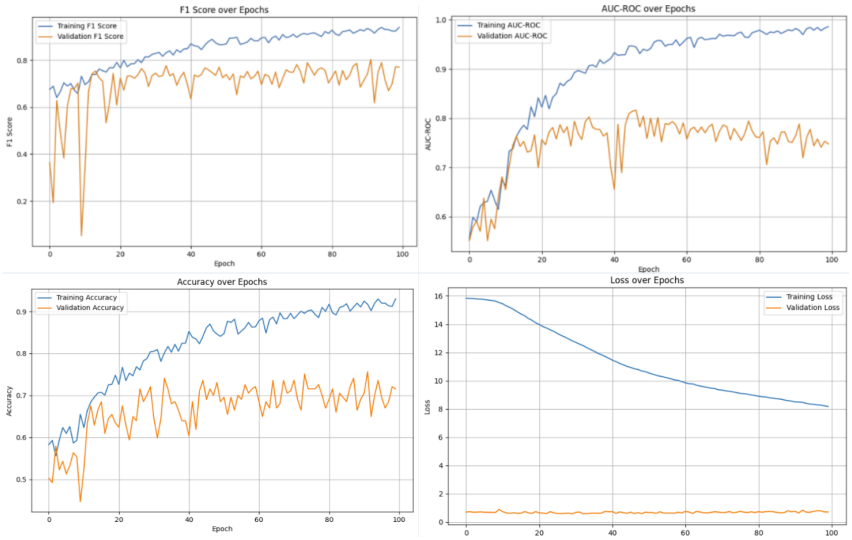

Figure 1: Training Results

- Structured HDL code as graph representations, providing a valuable resource for applying GNN models in hardware verification.

4.2 Graph Processing Methodology

- Implemented two preprocessing pipelines to transform HDL code into graph representations:
  - **Simple Preprocessing Pipeline:** Focused on module declarations, assignments, and control flow statements, enabling straightforward graph construction.
  - **Advanced Preprocessing Pipeline:** Introduced SystemVerilog-specific constructs, multiple edge types, and enriched node features, capturing more complex relationships between HDL elements.
- Leveraged GraphCodeBERT embeddings to enhance graph node representations, providing rich semantic and contextual features for analysis.

section*Verilog Code for State Machine

The following is the Verilog code used to implement the state machine:

Listing 1: State Machine Verilog Code

```
module state_machine(
    input logic clock,
    input logic reset,
    input logic condition,
    output logic out
);

// State encoding
typedef enum logic [2:0] {
    A = 3'b000,
    B = 3'b001,
    C = 3'b010
} state_t;

// State registers
state_t current_state;
state_t next_state;
```

```
// Sequential logic for state update
always_ff @(posedge clock or posedge reset) begin
    if (reset)
        current_state <= A;
    else
        current_state <= next_state;
end

// Combinational logic for next state
always_comb begin
    case (current_state)
        A: next_state = B;
        B: next_state = condition ? C : A;
        C: next_state = A;
        default: next_state = A;
    endcase
end

// Output logic
always_comb begin
    case (current_state)
        A: out = 1'b0;
        B: out = 1'b1;
        C: out = condition;
        default: out = 1'b0;
    endcase
end

endmodule
```

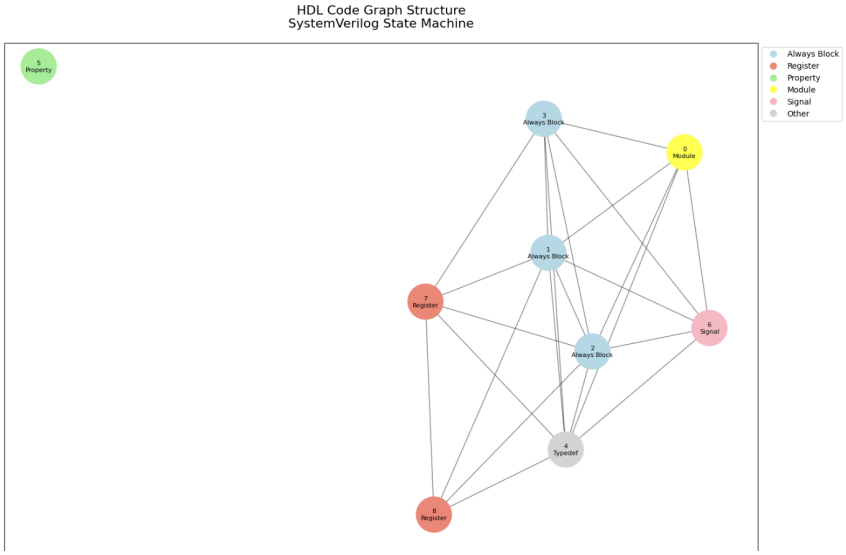

Figure 2: Output From Above Verilog Code

4.3 Graph Attention Network (GAT) Architecture

- Designed a three-layer GAT architecture with hierarchical attention mechanisms to model HDL relationships effectively.
- Utilized multi-head attention to capture diverse relationships between nodes, enhancing the model's ability to learn complex dependencies.
- Implemented skip connections between layers to improve gradient flow and prevent vanishing gradients, ensuring effective training of deep models.

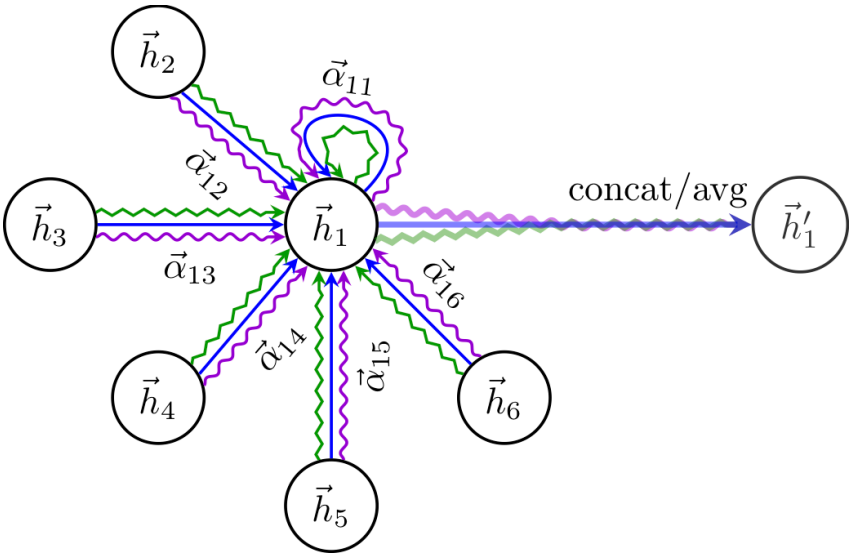

Figure 3: Illustration of GAT Network

In Figure 3, we can observe the structure of the proposed network.

5 Challenges

First, we dealt with overfitting and generalization. Overfitting was a significant challenge during training, as evidenced by limited validation improvement and stable validation loss. While the strong test results on simpler designs mitigate this concern, the performance drop on complex designs showcases the model's struggle to generalize effectively to intricate scenarios. Future efforts should focus on increasing dataset diversity and implementing stronger regularization techniques.

Next, there was the complexity of graph representation. Capturing the intricacies of HDL code in graph form, particularly with the advanced preprocessing pipeline, posed a significant challenge. The inclusion of multiple edge types and SystemVerilog-specific constructs made the graphs highly intricate, which in turn hindered the GAT's learning process. Balancing the expressiveness of graph representations with computational tractability remains a priority.

6 Future Work

6.1 Enhanced Dataset Construction

Moving forward, a primary research direction will be to construct an extended dataset. It may also be useful to include more failing assertions and complex design examples to help the model improve on its ability to generalize across diverse scenarios.

6.2 Regularization and Penalization Techniques

To mitigate the overfitting observed, future work should explore the use of more sophisticated regularization techniques. Introducing penalty terms to the loss function that emphasize correctly

identifying "fail" assertions could help reduce bias. Additionally, methods such as dropout, L1/L2 regularization, or focal loss could further improve generalization and address the imbalanced nature of the data.

6.3 Hybrid Graph Representations and Improved Attention Mechanisms

Exploring hybrid attention mechanisms that combine both soft and hard attention approaches could help capture critical relationships more effectively. Additionally, experimenting with other GNN architectures, such as Graph Convolutional Networks (GCNs), GraphSAGE, or Transformer-based graph models, could offer better scalability and improved handling of complex relationships, potentially enhancing overall performance.

6.4 Domain-Specific Feature Incorporation

Incorporating more domain-specific features into the node and edge embeddings could further improve the model's performance. Adding features such as timing information, synthesizability, or metadata about signal criticality would provide more context and enable the model to make more informed predictions, particularly in identifying failing assertions.

7 Conclusion

This work demonstrates the potential of leveraging Graph Attention Networks (GATs) combined with transformer-based embeddings for the formal verification of HDL. The key contributions include creating a dataset for this task, developing methodologies to transform HDL code into graph structures, and applying GATs to analyze assertion-based verification. Despite the observed overfitting during training, the mixed test results highlight both the promise and the challenges of this approach. The strong performance on simpler designs underscores the model's potential, while the drop in accuracy on complex designs emphasizes the need for further refinement. Addressing data imbalance, refining graph representations, and introducing targeted regularization are critical next steps to enhance the impact and robustness of this verification approach. The promising aspects of the model's performance on both validation and simple test data highlight the potential of graph-based deep learning methods in automating and scaling hardware design verification tasks, paving the way for further advancements in this challenging field.

References

@miscpyGAT, author = Diego999, year = 2017, title = pyGAT: Pytorch implementation of the Graph Attention Network (GAT), note = GitHub repository, url = https://github.com/Diego999/pyGAT

@inproceedingsGAT, author = Veličković, Petar and Cucurull, Guillem and Casanova, Arantxa and Romero, Adriana and Liò, Pietro and Bengio, Yoshua, year = 2018, title = Graph Attention Networks, booktitle = International Conference on Learning Representations, url = https://arxiv.org/abs/1710.10903

@inproceedingsCFGBugs, author = Zhang, Jian and Wang, Xinyi and Zhang, Hongyu and Sun, Haoyang and Liu, Xiaoyu and Hu, Chengfei and Liu, Yu, year = 2023, title = Detecting Condition-Related Bugs with Control Flow Graph Neural Network, booktitle = Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pages = 1370–1381, publisher = ACM, doi = 10.1145/3597926.3598142

@inproceedingsControlFlowVulnerabilities, author = Cheng, Xiaokang and Wang, Huan and Hua, Jing and Zhang, Meng and Xu, Guangliang and Yi, Lijun and Sui, Yulei, year = 2019, title = Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding, booktitle = Proceedings of the 24th International Conference on Engineering of Complex Computer Systems, publisher = IEEE, url = https://yuleisui.github.io/publications/iceccs19.pdf

@miscverilog-c, author = Mukherjee, Rajdeep, year = 2017, title = verilog-c: ANSI-C benchmarks generated from Verilog RTL circuits with safety assertions, note = GitHub repository, url = https://github.com/rajdeep87/verilog-c