International Conference on Formal Methods
and Foundations of Artificial Intelligence
Eszterházy Károly Catholic University
Eger, Hungary, June 5–7, 2025

Using GNN for Refactoring P4 Programs

Benedek Szabolcs Csüllög[a], **Máté Tejfel**[b],

[a]Eötvös Loránd University Budapest, ELTE
benedek.csullog@gmail.com

[b]Eötvös Loránd University Budapest, ELTE
matej@caesar.elte.hu

Abstract

P4 [2] is a domain-specific programming language mainly used for programmable switches running inside next-generation computer networks. The language is designed to use the software-defined networking (SDN) paradigm which separates the data plane and control plane layers of the program.

P4Query [3] is a static analysis framework for P4. The framework is centered around an extensible internal graph representation where the results of the different static analysis methods are also stored as part of the graph. The information in the knowledge graph is accessed using graph queries written in the Gremlin [4] query language. In this way, the framework guarantees a unique standard representation both for the stored data and for the data access mechanism. P4 programs can naturally be represented as abstract syntax trees (ASTs), which are directed graphs generated by the framework. The framework can also be used to implement P4 specific refactoring steps [5].

The paper focuses on two primary research directions. The first investigates a training strategy for Graph Neural Networks (GNN) [6], where the model progressively learns the AST representation over multiple training epochs. In each epoch, specific portions of the graph are selectively removed to enhance the model's ability to generalize its understanding of the structure. Initially, the training may involve the deletion of leaf nodes, which represent the terminal elements of the syntax tree. By starting with these simpler nodes, the model is encouraged to focus on the relationships and interactions among the remaining nodes, fostering a deeper comprehension of the overall tree structure.

As the training advances, the strategy can adapt to remove more complex nodes or entire subtrees, gradually increasing the complexity of the learning task. This

progressive removal not only helps the model to learn to reconstruct the graph from partial information but also aids in developing resilience against noise and variability in the data. By forcing the GNN to infer the missing parts of the AST, the model becomes adept at understanding the underlying patterns and semantics of the code.

Additionally, the individual nodes of the AST are encoded using autoencoders, with the embedding process being supported by the Code2vec model. Code2vec [1] is a framework that learns continuous feature representations for nodes in a graph. In the context of abstract syntax trees, the authors transformed the nodes of the syntax tree into vector representations. This allows machine learning models to capture the structural and semantic properties of the programs represented by the trees. By leveraging these vector representations, the model can effectively recognize and classify specific programs based on their syntax trees.

Code2Vec accomplished this by training models capable of recognizing well-known algorithms—such as sorting, searching, or counting—across multiple programming languages, including Java, C++, and Python. This approach involves generating a syntax tree from the source code, which the trained model then analyzes to identify the underlying algorithm. After the analysis, the model also provides a probability score for its prediction.

The above dual approach allows the GNN to capture both the local features of individual nodes and the global structure of the tree. The combination of selective node removal and sophisticated embedding techniques ultimately enhances the model's capability to perform tasks such as code classification and refactoring.

The researchers in [6] utilized Graph Neural Networks to investigate their expressive power, with a particular focus on distinguishing between different graph structures. Their study specifically examined tasks such as node classification, graph classification, and graph isomorphism testing.

The second research direction explores training multiple neural networks, each specializing in different aspects of the AST to perform automated refactoring tasks. The models learn to identify and modify labeled components within the AST, such as variable names, function declarations, and control structures, based on learned transformation patterns. Our findings demonstrate that deep learning techniques can effectively interpret and manipulate P4 programs by leveraging the structural information embedded in abstract syntax trees.

In summary, graph-based models provide a practical way to perform syntax-aware refactoring based on the structural properties of code. By learning directly from abstract syntax trees, these models can support transformations that follow the formal rules of the language and preserve program correctness. This makes them suitable for tasks such as parameter reordering, renaming, and code cleanup, where the syntactic structure plays a central role. Applied to P4, this approach can contribute to maintaining consistent and readable code, while reducing the need for manual intervention in repetitive refactoring steps.

References

[1] U. Alon, M. Zilberstein, O. Levy, E. Yahav: *code2vec: learning distributed representations of code*, Proc. ACM Program. Lang. 3.POPL (Jan. 2019), doi: 10.1145/3290353, url: https://doi.org/10.1145/3290353.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker: *P4: programming protocol-independent packet processors*, SIGCOMM Comput. Commun. Rev. 44.3 (July 2014), pp. 87–95, issn: 0146-4833, doi: 10.1145/2656877.2656890, url: https://doi.org/10.1145/2656877.2656890.

[3] D. Lukács, G. Tóth, M. Tejfel: *P4Query: Static analyser framework for P4*, Annales Mathematicae et Informaticae 57 (2023), pp. 49–64, doi: 10.33039/ami.2023.03.002.

[4] M. A. Rodriguez: *The Gremlin graph traversal machine and language (invited talk)*, in: Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015, Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 1–10, isbn: 9781450339025, doi: 10.1145/2815072.2815073, url: https://doi.org/10.1145/2815072.2815073.

[5] M. Tejfel, D. Lukács, P. Hegyi: *P4 Specific Refactoring Steps*, Acta Cybernetica 27.1 (Mar. 2025), pp. 53–65, doi: 10.14232/actacyb.308085, url: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4465.

[6] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun: *Graph neural networks: A review of methods and applications*, AI Open 1 (2020), pp. 57–81, issn: 2666-6510, doi: https://doi.org/10.1016/j.aiopen.2021.01.001, url: https://www.sciencedirect.com/science/article/pii/S2666651021000012.