CONCORD: A DSL for Generating Simplified and Scalable Graph-Based Code Representations

Mootez Saad and Tushar Sharma
Dalhousie University
Halifax, Canada
{mootez,tushar}@dal.ca

***Abstract*—Graph-based representations have gained attention for their ability to model structural and semantic information capturing relevant characteristics and features of source code for training deep learning models. However, existing methods face limitations: they lack flexibility in constructing cross-language graphs, produce non-interoperable outputs, and generate excessively large graphs, hindering their adoption and raising scalability and efficiency issues in graph-based neural network training.**

In this work, we introduce CONCORD, a domain-specific language (DSL), to address these challenges. We aim to 1) enable customizable graph-based code representations across programming languages, 2) reduce graph size complexity through simplification heuristics, and 3) improve scalability and reproducibility in software engineering tasks.

CONCORD provides a configurable DSL to automate graph construction and implements heuristics to reduce graph size while preserving critical information. We evaluate its effectiveness on two tasks: *code smell detection* and *vulnerability detection*. For each, we compare performance and graph size against baseline representations without simplification heuristics.

On code smell detection, CONCORD preserved 95.1% of baseline performance, and exceeded it by 5% in one setting, while reducing, on average, the number of nodes and edges by 13.11% and 13.67%, respectively. For vulnerability detection, it improved performance by 3.65% over the baseline while reducing the number of nodes and edges by 3.63% and 3.62%, respectively. This demonstrates that CONCORD's heuristics maintain or enhance performance while improving scalability.

CONCORD represents a step towards advancing graph-based code analysis by offering a flexible, language-agnostic approach to generate streamlined code representations. Its simplification heuristics balance performance and scalability, enabling efficient training of neural models without sacrificing accuracy. In addition, it reduces development overhead, promotes reproducibility through standardized representations, and broadens accessibility to graph-based methods for software engineering tasks.

***Index Terms*—component, formatting, style, styling, insert**

I. INTRODUCTION

Deep learning (DL)-based methods for source code analysis have gained significant momentum [1], [2]. Such methods have been applied in various software engineering tasks, such as defect prediction [3], [4], [5], code smell detection [6], [7], [8], code summarization [9], [10], [11], and program synthesis [1], [2]. Generating code embeddings is a common step in these approaches, where code is converted from its raw textual form to a numeric representation, using models such as CodeBERT [12], that can be later used to solve such tasks [6], [12], [13]. However, treating source code as a stream of tokens ignores its rich structure and semantics [14], [15], [16]. Hence, efforts have been made to incorporate the code's structural and semantic features into its vector representation [14], [15], [17].

Many works experimented with graph-based neural network architectures (GNN) [18], [19] that can better capture different semantic aspects (*e.g.,* control and data flow) of source code. Often, based on the downstream task, it is required to combine multiple code representations such as Abstract Syntax Tree (AST) and Program Dependence Graph (PDG) into a unified graph structure. For example, Allamanis *et al.* [20] augmented AST with custom *data flow edges* and *control flow edges* to detect variable misuse in a code snippet.

However, the process of combining code representations and pre-processing them is often done manually and from scratch which wastes significant research time and resources. Some works were proposed to improve such a process. Bieber *et al.* [21] provide *python_graphs* to construct graph representations of Python programs suitable for training machine learning models by employing static analysis. It is capable of constructing control-flow graphs, data-flow graphs, and composite *program graphs* that fuses control-flow, data-flow, syntactic, and lexical information of a program. Rice *et al.* [22] developed a *javac* plugin *feature-javac* that creates feature graphs for Java-based source code repositories. These graphs incorporate a variety of syntactic and semantic features. It bears similarities with the *python_graphs* library and is designed to facilitate machine learning on source code research.

Despite such progress, the current mechanisms suffer from limitations that hinder their adoption. Existing works often lack flexibility, are restricted to a single programming language, and produce non-interoperable outputs. Furthermore, they frequently generate excessively large graphs, which raises scalability and efficiency issues for training graph-based neural networks. Moreover, the lack of standardized generation pipelines makes results difficult to reproduce and integrate across tools and studies.

More recently, tools like COMEX [23] have advanced this area by generating combined code views (*e.g.,* AST, CFG) from source code. While offering valuable customizations, these frameworks still lack a way for fine-grained configuration and focus on structural simplification. Such method of simplification may not adequately address the graph complexity that arises from sections of code that, while syntactically correct, are semantically less relevant for a given analysis task. To address these challenges, we introduce CONfigurable COde Representation (CONCORD), a Domain Specific Language (DSL) and a unifying framework designed to automate the generation of custom code representations, and implements a set of heuristics for graph size reduction. The DSL automatically generates code representations from source code, and as per the specified configuration.

The primary goals of CONCORD are to *reduce the computational costs* of applying GNN to source code by simplifying their graph representations, and to streamline such generation process. While improving model performance is not the *main* objective of this simplification, we use performance metrics to validate that the process does not discard information critical to the downstream task.. Our empirical evaluation on *code smell* and *vulnerability detection* tasks demonstrates CONCORD's effectiveness in this regard. For the code smell detection task, CONCORD reduced the average graph size by 12.35%. While this efficiency gain could result in a performance drop of up to 4.9% when detecting certain smells, we surprisingly observed an improvement of up to 5.0% in others, relative to the baseline model. A 2.63% graph size reduction was accompanied by a performance improvement of 3.65% for vulnerability detection. These results confirm that CONCORD successfully lowers computational requirements by creating smaller, more efficient graph representations while maintaining or even slightly enhancing model effectiveness. The key contributions of this study are:

- Provide a one-stop solution in the form of a DSL that unifies multiple edge augmentation techniques to produce rich and expressive graph representations in a *composable*, *configurable*, and *cross-linguistic* manner.
- Propose and implement two pruning techniques resulting in reduced size of code graphs that make training more tractable and less computationally expensive.
- Present a case study to demonstrate the capabilities of CONCORD to apply on code smell detection and vulnerability detection tasks.

II. BACKGROUND AND MOTIVATION

This section establishes the motivation for CONCORD by first detailing the challenges posed by current graph-based code representations. We then analyze the specific limitations of existing methods and build the rationale for our approach.

A. Graph-Based Representations for Code Analysis

Graph-based representations have become central to modern code analysis methods [19], [24]–[26] because they preserve the rich structural and semantic information that is lost when source code is treated as a simple sequence of tokens.

The challenges in generating useful code graphs stem from concrete limitations in the current tooling and methodologies. We identify the primary issues that hinder progress and scalability in graph-based code analysis.

First, existing methods are often tightly coupled to a single programming language, restricting their applicability. For example, popular tools like *python_graphs* [21] and *feature-javac* [22] are designed exclusively for Python or Java, respectively, limiting their use in cross-language research and for projects built with other common languages.

Second, they tend to produce fixed, monolithic representations that include all possible types of edge augmentations at once. This "*all-or-nothing*" approach prevents users from tailoring the graph structure to the specific needs of a downstream task. For instance, a user cannot easily specify a configuration to include data-flow edges while excluding certain lexical edges, even if the latter are irrelevant to their model.

Moreover, the set of features and augmentations offered by existing solutions is disjoint, with different operations available in different libraries. This fragmentation makes it difficult to compare results across studies or combine artifacts from different tools without considerable upfront effort [27].

Finally, these methods do not consider the size of the generated graphs. We elaborate on such an aspect in the following subsection.

B. The Issue of Graph Size

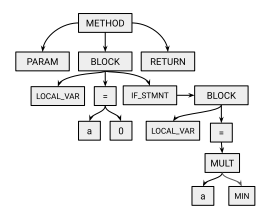
Graphs derived from source code grow large due to their exhaustive structural granularity, hierarchical nesting, and metadata. While this explicitness supports semantic precision, it severely impacts the scalability and training efficiency of graph neural networks: large graphs strain memory, increase computational complexity, and introduce noise from redundant substructures.

To demonstrate this issue, let us consider a small code snippet and its corresponding AST:

Listing 1: Example of a C-like code snippet

```c
void foo(int x) {
    int a = 0;
    if (a < MIN) { int b =
      a*MIN; }
}
```

Fig. 1: AST of Listing 1



Despite being only four lines long, the resulting AST contains 19 nodes as shown in Figure 1. Moreover, an AST can only capture the structural aspects of code, which necessitates including other attributes to represent control and data flow. However, this augmentation further aggravates such an issue, resulting in a larger graph.

During training, a GNN must store and manipulate large adjacency matrices representing the graph, which entails the need for large memory and substantial computational resources. One way to solve this issue is by reducing the number of parameters of these GNNs. However, this can harm their representation learning capacity. An alternative direction would be performing the simplification at the *data* level.

C. Noisy Statements in Source Code

Many codebases contain *noisy* statements that do not contribute much to their core logic. A prime example is print statement; while common, it is often used for purposes auxiliary to the program's primary function, such as debugging and error tracing. Studies by Beller *et al.* [28] and Liu *et al.* [29] have shown that professional software developers rely on print statements such as `printf` as a debugging method. Similarly, temporary variables introduced to improve readability can also add unnecessary length and complexity to the code's structure.

We hypothesize that pruning such statements can *reduce* source code and allow the model to learn better its salient features to capture its intended behavior. This approach is analogous to natural language processing, where some elements, such as stop-words, punctuation, and special characters, are removed to reduce the noise present in the text.

III. CONCORD LANGUAGE DESIGN

CONCORD is a DSL and framework that we propose in this paper to construct customized graph representations of source code. The constructed graph representation can be employed for training DL models. Figure 2 provides an overview of a generic pipeline for DL model training using CONCORD.

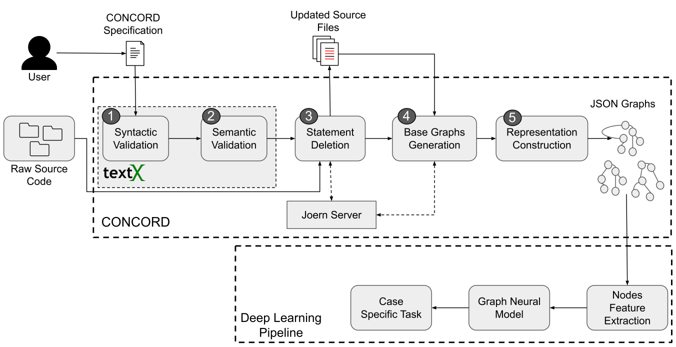


Fig. 2: Overview of the steps involved to generate graph representations of source using CONCORD.

To train a GNN model, a user first generates a graph representation of the source code. They provide a CONCORD configuration which is interpreted by the language's backend to generate the concrete graphs. These graphs can later be used for training and inference. The language grammar is defined and interpreted using *textX* [30]. It is formalized by a Parsing Expression Grammar (PEG) [31] as shown in Listing 2. PEG is a formalism used to describe language syntax and facilitate parsing. Unlike Context-Free Grammar (CFG), PEG ensures deterministic parsing and disallows ambiguity. It employs recursive descent parsing, prioritizes alternatives in order, handles left recursion gracefully, and provides better support for nested structures.

```
<Main> → Tasks { <Task>* } Representations { <Representation>+ }
<Representation> → ID { STRING STRING <BaseGraphType>+ [<Task>]+ }
<BaseGraphType> → AST | CFG | PDG
<Task> → ID { <Operation>+ conditions { <Condition>+ } }
<Operation> → <GraphElement> <OperationType> <GraphElementType>
<GraphElement> → Node | Edge
<OperationType> → add | remove
<GraphElementType> → <EdgeType> | <NodeType>
<EdgeType> → next_token | next_sibling | for_cfg | while_cfg | last_read_write |
        guarded_by | returns_to | computed_from | last_lexical_use
<NodeType> → print | logging | sys_exit | simple_assignment
<Condition> → <CodeCondition>
<CodeCondition> → <CodeConditionAction> <CodeBlock>
<CodeConditionAction> → exclude | include
<CodeBlock> → catch | for | while | if | else
```

Listing 2: PEG-style representation of CONCORD's grammar

A. *Key Entities of* CONCORD *Language*

A *Representation* specifies the intended graph to be constructed; it is defined by a set of base graph structures, as well as a set of *Tasks*. The base graphs are: AST, CFG, and PDG. If multiple base graphs are specified, they are merged into a single graph structure.

A *Task* comprises a set of *Operations* that are applied to the base graphs. An *Operation* represents an atomic action that can be applied to the graph, and there are two types—*node removal* and *edge addition*.

A *Code Condition* specifies the code structures that are exempt from node removal. Currently, supported code structures include control statements such as *if* statements and *for* loops (rule *CodeBlock* in Listing 2). In other words, nodes (*i.e.,* statements) that are within these blocks will not be removed if specified. This feature provide greater flexibility to the user, where instead of blindly pruning statements through all code regions, they may want to keep some blocks intact depending on their use case.

B. *Graph Generation Process*

In this part, we describe the different stages involved in generating the graphs based on the user-provided specification.

1) Syntactic validation: In this stage, *textX* [30] constructs a model from the CONCORD specification file provided by the user and checks whether it conforms to the grammar rules and the metamodel of the language.

2) Semantic Validation: This step ensures the correctness and consistency of the CONCORD specification. While syntactic validation checks if the specification adheres to the grammar rules, semantic validation verifies that the specified operations, conditions, and combinations make sense in the context of the DSL. For instance, combining `Node` with `EdgeType` (*e.g.,* `Node add next_token`) is invalid because edge-related operations should be applied to edges, not nodes.

3) Statement Deletion: In this step, we prune the statements specified by the user at the file level. Later in Section III-C, we illustrate how this process is implemented and the design decisions behind it.

4) Base Graphs Generation: Once the specification passes the validation steps and statements are pruned, we generate the base graph representations using *joern* [32], a platform for code analysis that supports C/C++, Java, JavaScript, Python, and PHP. The generated representations are stored in JSON format.

5) Representation Construction: Finally, we import the base graphs and construct the representation using the *NetworkX* library [33]. The framework generates code representations as per the specification file. We store code representations in JSON format, which can be parsed by NetworkX, allowing further processing, if needed.

C. Node Removal

When source code is transformed into graphs, the resulting structure can be significantly complex in terms of size (*i.e.,* the number of vertices and edges). To mitigate this issue, one approach is to discard graphs from the dataset that exceed a certain threshold. For example, Zhou *et al.* [19] do not consider graphs with more than 500 nodes. However, such a strategy can lead to information loss, which may adversely impact the performance of DL models.

Gu *et al.* [34] introduced *Simplified Syntax Tree* (SST), which is an approximation that simplifies the AST by pruning nodes such as type declarations and modifier keywords. By using this simplified representation, they were able to improve the performance of neural models in code search tasks using the CodeSearchNet benchmark. Inspired by this heuristic, we introduce a similar way of simplification in CONCORD, however, at a wider scope (*i.e.,* statement-level removal). The types of statements that can be removed are *simple assignment*, *print*, and *system exit* statements, as a means to reduce the complexity of the graph representations (Step 3, Figure 2).

Simple assignment statements initialize a variable with a literal value, such as an integer, float, double, or boolean value. Print statements are functions or methods provided by the programming language to output information on the console, such as the `println()` statement in Java. Similarly, exit statements are methods used to transfer control of the program, such as the `System.exit()`. In our approach, we perform statement removal at the file level *i.e.,* we identify the line numbers corresponding to these statements and *then* perform the removal in each source code file in a repository. We chose to perform statement removal at the file level rather than removing them after generating the base graphs to leverage the data flow engine of the *joern* tool for generating the PDG. This is especially relevant when the user specifies more complex base representations, such as PDG or CFG, as it ensures more reliable results by using a well-implemented and maintained engine. The overall process is illustrated in Figure 3, where we first collect the statements specified by the user, and then perform the deletion, taking into consideration the code blocks specified in the specification.

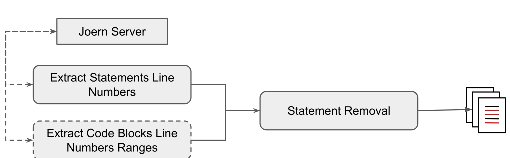


Fig. 3: Statement removal process using CONCORD

Algorithm 1 presents the procedure for identifying simple assignment statements. We traverse the AST of each method, and identify the subtrees that represent simple assignment statements. A statement is considered a simple assignment statement if it satisfies the following four conditions: 1) The node representing the statement is of type **ASSIGNMENT**. 2) Its left subtree of the node has only one node of type **IDENTIFIER**. 3) The non-leaf nodes of its right subtree represent one of the allowed operators (arithmetic, logical, relational, and bitwise). 4) All leaf nodes of its right subtree are literals.

Algorithm 1 Check if an AST subtree is a simple assignment.

```
function ISSIMPLEASSIGNMENT(root)
    ▷ Condition 1: Root must be an ASSIGNMENT node with exactly two children
    if root.type ≠ ASSIGNMENT or root.children.size() ≠ 2 then
        return false
    end if
    lhs_subtree ← root.children[0]
    rhs_subtree ← root.children[1]
    ▷ Condition 2: The left subtree must be a single IDENTIFIER node
    if lhs_subtree.type ≠ IDENTIFIER or lhs_subtree.children.size() ≠ 0 then
        return false
    end if
    ▷ Condition 3: All non-leaf nodes in the right subtree must be allowed operators
    nonLeafNodes ← GETNONLEAFNODES(rhs_subtree)
    if not nonLeafNodes.ALL(isAllowedOperation) then
        return false
    end if
    ▷ Condition 4: All leaf nodes in the right subtree must be literals
    leafNodes ← GETLEAFNODES(rhs_subtree)
    if not leafNodes.ALL(isLiteral) then
        return false
    end if
    return true
end function

function ISALLOWEDOPERATION(node) ▷ Returns whether a node represents an allowed operation
    arithmeticOperation ← { add, sub, mult, div, exp, mod }
    bitwiseOperation ← { not, and, xor, or, shiftLeft, shiftRight }
    logicalOperation ← { not, and, or }
    relationalOperation ← {equals, notEquals, greaterThan, lessThan, ...}
    allowedOperations ← arithmeticOperation + bitwiseOperation + logicalOperation + relationalOperation
    return node.type ∈ allowedOperations
end function
```

We design the filtering procedure in this way to ensure that even complex statements involving operations between literals, such as `int a = 1*7+(1-7)`, are also considered. Though such statements are rarely found in practice, we did so for the sake of completeness.

To remove a simple assignment, the statement is replaced with an empty string rather than deleting the source code line To explain why we follow such an approach, let us consider the example given in Listing 3.

```
public static void foo(String[] args) {
   for (int i = 1; i <= n; ++i) { System.out.println
     ("Printer"); }
 }
```

Listing 3: Simple assignment statement to be removed

If we were to remove the entire line, the resulting code would be completely erroneous, as seen in Listing 4, with mismatched opening and closing brackets.

```
public static void foo(String[] args) {
        System.out.println("Printer"); }}
```

Listing 4: Incorrect removal of the statement if the entire line is deleted

However, replacing the statement with an empty string instead preserves the rest of the code that has not matched the condition, which is relevant for the code representation process.

```
public static void foo(String[] args) {
   for (; i <= n; ++i) { System.out.println("Printer
    "); } }
```

Listing 5: How the statement is removed by CONCORD

Moreover, *joern* employs fuzzy parsing [35] to handle incomplete source code with errors, but the accuracy of the output depends on the extent of the error in the input. This approach ensures a careful deletion process that retains code fragments without a significant loss of information.

D. Edge Addition

The next major operation provided by CONCORD is edge addition. Edges generated from base graphs, such as ASTs and PDGs, capture certain aspects of the source code, but neglect others. For instance, while ASTs effectively reflect the structure of a code snippet, they do not capture its control flow. As an alternative approach, several studies [19], [20], [36] have proposed additional sets of edges that can augment the base representations for various tasks. However, these augmentations are fragmented across different, non-interoperable tools and are not offered in a configurable way. CONCORD unifies these proven techniques into a single, flexible method. Below, we provide a detailed description of each type of edge that can be added.

1) NEXTSIBLING: Used by Wenhan *et al.* [36], this type of edge connects an AST node to its sibling node. The reason for such augmentation is primarily due to the behaviour of GNNs that does not consider the order of nodes. Hence, it is helpful for the neural model to provide the order of children.

2) NEXTTOKEN: This edge connects sibling nodes of an AST with each other. Allamanis *et al.* [20] introduced this edge type in their study; which was also used by Wenhan *et al.* [36]. The rationale for introducing the edge is that an AST does not reflect the sequential nature of the source code. Let us consider Figure 4 that represents the syntax tree of this C-like statement `int i = 1+1`

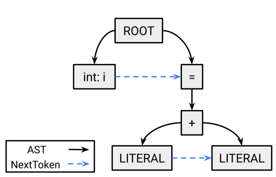


Fig. 4: Example of a NEXTTOKEN edges in an AST

The figure shows that the NEXTTOKEN edges in blue reflect, to an extent, the sequential property of the code snippet.

3) LASTREAD: Let `U` is to the set of tokens where a variable `var` could have been last used. This set can consist of multiple nodes, such as when the variable is used in both paths of a conditional statement, or even tokens that come after in the code, as in the case of loops [20]. The LASTREAD edges create a connection between these nodes in the set `U` and `var`.

4) LASTWRITE [20]: It is similar to LASTREAD, except the set `U` would contain the tokens where `var` could have been last written to.

5) LASTLEXICALUSE [20]: It is also similar to both of the aforementioned edges, however, it is flow independent. In other words, all occurrences of variable `var` in a source code snippet are chained or linked together by this type of edge.

6) COMPUTEDFROM [20]: This concerns assignment statements. It connects tokens that are on the right-hand side of an assignment statement to the variable on the left-hand side. For example, in Figure 4 both of the literal leaf nodes will be connected to the variable `i` by this edge.

7) RETURNSTO [20]: Connects a method declaration node with the node that refers to the method's `return` statement.

8) GUARDEDBY and GUARDEDBYNEGATION [20]: A GUARDEDBY edge connects a conditional node to the usage of the variable when the guard condition is true. GUARDEDBYNEGATION performs a similar action but when the conditional is false. Figure 5 presents the AST of the snippet given below (note that we pruned some parts of it for presentation purposes). `if (a '!=' b) {a '=' 5;} else {b'='5;}`

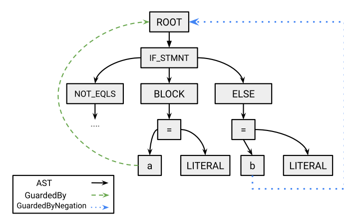


Fig. 5: Example of GUARDEDBY and GUARDEDBYNEGATION AST edges

The green edge refers to a GUARDEDBY relation, whereas GUARDEDBYNEGATION is represented by the blue edge.

9) WHILECFG and FORCFG: Both of these relations were introduced by Wenhan *et al.* [36] to model the control flow behaviour of while and for loops. Concretely, each relation is represented by two edges: `WhileExec` and `WhileNext` for WHILECFG and the same logic applies for FORCFG. The `WhileExec` connects the conditional node of the while statement to its block node, whereas `WhileNext` does the inverse. Again, the same mechanism is implemented for the for-loop blocks.

IV. EXPERIMENTS

To validate CONCORD, we conduct an empirical study to solve two software engineering tasks: code smell and vulnerability detection. We choose these tasks given their importance in software engineering, specifically in software maintenance. Second, the data used in these tasks covers two different programming languages (Java and C/C++). This would demonstrate how CONCORD can be used seamlessly irrespective of the programming language without engineering overhead.

We first present the CONCORD specifications used to generate the different representations and the rationale behind each one. Then, we describe the two aforementioned tasks and the dataset used. Finally, we provide the training details of the graph-based neural network model.

A. CONCORD *Specifications*

To evaluate and demonstrate CONCORD's pruning strategies mentioned in Sections III-D and III-C, we define three CONCORD configurations: R_1, R_2, and R_3 specified in Table I.

TABLE I: Specification of each representation. Nodes: ✓means that the statement type (*i.e.,* set of nodes) is kept, whereas, ✗ signifies that it is pruned. Edges: NEXTTOKEN (NT); WHILECFG (WCFG); FORCFG (FCG); GUARDEDBY (GB); COMPUTEDFROM (CF);

Representation	Base Representation	Print Statements	Simple Assignment Statements	Edges
R_1	AST	✓	✓	NT, WCFG, FCFG, GB, CF
R_2	AST	✓	✗	NT, WCFG, FCFG, GB, CF
R_3	AST	✗	✓	NT, WCFG, FCFG, GB, CF

We select these configurations to show how the DSL can be used to capture different aspects of source code. We utilize AST as the base representation to model the structural nature of code Similarly, we specify the same edge configuration for the three representations. Specifically, the NEXTTOKEN edge is used to model its sequential nature, while the WHILECFG, FORCFG, GUARDEDBY, and COMPUTEDFROM edges are added to capture the flow of control and data through the program. The representations differ in their pruning strategies, where R_2 prunes the simple assignment statements and R_3 discards the print statements. Figure 6 illustrates, as an example, specification of R_1 using the CONCORD syntax. Given space limitations, the remaining two specifications can be found in the replication package [37]. In the `Tasks` block, we define the list of tasks that will be executed.

```
Tasks {
    task1 {
        Edge add next_token
        Edge add for_cfg
        Edge add while_cfg
        Edge add computed_from
        Edge add guarded_by

        Conditions {
            exclude while
            exclude if
}}}
Representations {
    r1 {
        "/dir/repos_list.csv"
        "output_dir"
        AST
        task1
}}
```

Fig. 6: The R_1 specification using CONCORD's grammar.

Then, in the `Representations` block, the list of graph representations that will be generated is specified. Each representation includes the list of source code repositories that will be used as input, one or multiple base representations, and the tasks of that will be applied. Note that, for space constraints, we are only specifying one task in the `Tasks` block and the same in the `Representations` block.

B. *Code Smell Detection*

1) Task Description: Code smells are symptoms of sub-optimal design and implementation choices manifested in the form of poorly written code that requires refactoring [38]. Their presence in a given software system hinders its quality, specifically maintainability [39], [40] and reliability [41]–[44]. Hence, many efforts have been made to produce tools and methods to detect them [45]. However, an important aspect of code smells is that they are subjective in nature [45], [46]. A code snippet can be smelly in one context, and benign in another. This means that classifying such anomalies deterministically using rule-based tools may not produce accurate results. To this end, we conduct a set of experiments showcasing how the features of CONCORD can be leveraged. In this study, we consider four types of code smells that were initially investigated by Sharma *et al.* [6]: *Complex method*, *Complex conditional*, *Feature envy*, and *Multifaceted abstraction*. We formulate the problem as a *one-vs-all binary classification problem*, where each code smell represents a dataset where each sample can be categorized as smelly or non-smelly, which is commonly adopted in related literature.

2) Dataset: We randomly selected and cloned a set of 100 projects from the original set of 922 Java repositories that were mined by Sharma *et al.* [6] in their study that aimed to explore the feasibility of detecting code smells using deep learning.

To ensure reliability, we selected the snapshot of each project before their replication package was published. Specifically, we checked out the commit before January 26, 2019, to reduce the possibility of refactoring, which could have removed these smells and corrupted the labeling.

After executing CONCORD using each of the three specifications, we generate an initial collection of $\approx 1.5M$ methods. To construct the datasets for each code smell, we randomly select a subset of the methods, while maintaining the positive-to-negative ratio mentioned in the work of Sharma *et al.* [6]. We did so to keep the training time tractable, given the resources that we had access to, similar to what was done in related research [47], [48].

TABLE II: Code smell detection datasets statistics

Smell	Train		Validation		Test	
	N	P	N	P	N	P
Complex Method	60,000	3,222	7,500	403	7,500	403
Complex Conditional	60,000	2,478	7,500	310	7,500	310
Feature Envy	30,905	326	3,864	41	3,864	41
Multifaceted Abstraction	5,184	180	648	23	648	23

Finally, we perform an 80:10:10 split corresponding to training, validation, and testing using stratified sampling to ensure fair evaluation for each classifier. Table II presents a summary of the training dataset with the class distribution for each dataset at each split.

TABLE III: Graph size statistics for code smell datasets.

Smell	R_1		R_2		R_3	
	Avg. Nodes	Avg. Edges	Avg. Nodes	Avg. Edges	Avg. Nodes	Avg. Edges
Complex Method	40.91	65.68	33.1	52.63	33.59	53.59
Complex Conditional	40.82	65.48	32.77	52.02	33.35	53.16
Feature Envy	274.68	431.18	267.3	418.04	270.28	423.99
Multifaceted Abstraction	487.28	770.42	415.0	649.12	434.6	683.49

Table III tabulates the average number of nodes and edges in each dataset after running CONCORD across all configurations. We observe differences in the number of nodes and edges between the implementation and design smells. This is because we represent a design smell instance (*i.e.,* a class) by combining the graphs of its methods into one graph. Consequently, on average, design smells exhibit a higher number of nodes compared to their implementation counterparts.

C. Vulnerability Detection

1) Task Description: Software vulnerabilities are flaws or weaknesses in a system's design or implementation that can compromise its security or functionality. They can lead to severe consequences, including data breaches and unauthorized access. Earlier approaches included static analysis methods and symbolic execution [49]–[51] to detect them. Deep learning approaches have emerged as complementary approaches. Specifically, approaches that leverage the graph structures of source code that can model its properties appropriately [24].

TABLE IV: Average number of nodes and edges in the Devign dataset under each representation.

Representation	Avg. Nodes	Avg. Edges
R_1	192.22	314.73
R_2	182.88	300.91
R_3	187.61	305.75

2) Dataset Used: We use the dataset released by Zhou *et al.* [52]. It includes vulnerable and benign C/C++ functions that were extracted from four large-scale open-source repositories and manually labeled. The dataset is composed of $27,318$ samples, where $21,854$ are used for training, $2,732$ for validation, and $2,732$ for testing. We generate the graph representations using the CONCORD specifications mentioned in Section IV-A.

D. Model description

We employ a Gated Graph Neural Network (GGNN)-based [53] classifier to solve the downstream tasks. We select this architecture over other variants such as Graph Convolutional Networks (GCN) [54] or Graph Attention Networks (GAT) [55], due to its superior performance demonstrated in other software engineering tasks [27] including clone detection, code classification, and vulnerability detection.

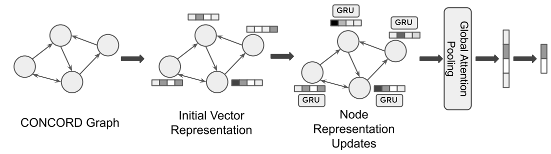


Fig. 7: Gated Graph Neural Network-based classifier

Figure 7 illustrates the end-to-end training process of the classifier. We trained the model with the following hyperparameters, inspired by the experiments conducted by Siow *et al.* [27] and Chakraborty *et al.* [24]: a learning rate of 1×10^{-3}, a GGNN layer input size of 768, and a GGNN layer hidden size of 128. We used a batch size of 128 and trained the model for 100 epochs, with an early stopping patience of 25 epochs.

E. Research Questions

We train a classifier for each dataset-representation combination, giving a total of 15 trained models. Given it is a classification problem, we compute precision (P), recall (R), and (F_1) measure. Furthermore, we compute FLOPs (number of floating points operation) to measure the model's complexity following similar work [56]. The lower the FLOPs value is, the lower the computational complexity.

Through the conducted experiments, we would like to answer the following research questions.

RQ1 *Does* CONCORD *generate graphs that allow graph-based models to learn features to solve the downstream tasks?*
RQ2 To what extent do CONCORD simplification heuristics reduce the size of graph-based code representations?
RQ3 *What is the impact of each pruning heuristic on downstream performance?*
RQ4 *How do* CONCORD*'s simplification heuristics balance model performance against computational efficiency?*

V. RESULTS

Table V shows the performance of the models trained on datasets generated from the three specifications. We report the average over five runs with different initializations.

A. RQ1: Validity of the Baseline Representation

The goal of this research question is to validate whether the graphs allow GNN models to learn features that can be used to solve the downstream tasks. On the code smell detection datasets, we obtain an average precision, recall, and F1 scores of 52.46, 47.2, and 49.20, respectively when training the models using the R_1 representation. Although the primary focus of the study is not to showcase the performance of GNNs on code smell detection tasks, a possible explanation for obtaining such values could be attributed to the extreme class imbalance of each dataset. Table II shows that the smelly samples account for only $\sim 4\%$ of the total sample size on average. Given that we retained the real-life distribution of smelly and benign sample ratios, this adds robustness to our evaluation. Indeed, experimenting with datasets that do reflect

TABLE V: Classification results for each specification. P: Precision, R: Recall, FLOPs: Number of Floating Point Operations. The highest F1 scores across the three configurations in each dataset are in **bold and underlined**. The ★ symbol indicates the highest performing pruning strategy. The ↑ sign indicates that the higher the value is the better, whereas, ↓ indicates the lower the value the better. The reported number of FLOPs was measured using the test set of each dataset.

	R1				R2				R3			
Software Engineering Task	P ↑	R ↑	F1 ↑	FLOPs ($\times 10^9$) ↓	P ↑	R ↑	F1 ↑	FLOPs ($\times 10^9$) ↓	P ↑	R ↑	F1 ↑	FLOPs ($\times 10^9$) ↓
Code smell detection												
Complex Method	55.31	47.22	**<u>50.95</u>**	0.171	52.95	47.02	49.81	0.162	55.63	46.34	50.56★	0.165
Complex Conditional	48.03	43.458	45.62	0.175	43.448	41.10	42.24	0.168	47.024	45.31	**<u>46.15</u>**★	0.170
Feature Envy	53.40	63.11	57.85	0.300	51.15	59.31	54.93	0.252	82.025	48.17	**<u>60.69</u>**★	0.260
Multifaceted Abstraction	53.11	35.02	**<u>42.21</u>**	0.232	52.09	24.13	32.98	0.205	52.09	29.10	37.34★	0.210
Vulnerability detection												
Devign	54.14	68.12	60.33	0.181	55.05	65.13	59.66	0.172	52.54	77.21	**<u>62.53</u>**★	0.175

the actual distribution of anomalies can lead to misleading results. Chakraborty *et al.* [24] showed that the performance of the state-of-the-art GNN-based models used for vulnerability detection dropped by more than 50% when trained and evaluated on representative data. For the vulnerability detection task, the obtained score F1 is 60.33% which is close to what was measured and reported in similar works that used graph-based representations and graph neural networks on the same dataset [52], [57].

Summary: GNN models yield a reasonable performance on data that maintains real-world code smell distribution where smelly snippets are rare. In addition, the results on the Devign dataset are consistent with previous works. This indicates that our proposed approach generates representations that capture the desired code features for code analysis tasks.

B. RQ2: Impact of Heuristics on Graph Size

A primary goal of CONCORD is to generate more scalable and computationally efficient graphs for deep learning models.

TABLE VI: Reduction rates of nodes and edges for R_2 and R_3 compared to R_1

	R2		R3	
Task	Nodes %	Edges %	Nodes %	Edges %
Smell detection				
Complex Method	19.09%	19.86%	17.89%	18.40%
Complex Conditional	19.72%	20.56%	18.30%	18.82%
Feature Envy	2.69%	3.05%	1.60%	1.67%
Multifaceted Abstraction	14.83%	15.75%	10.81%	11.28%
Vulnerability detection				
Devign	4.86%	4.39%	2.40%	2.85%

As detailed in Table VI, the pruning heuristics lead to a consistent decrease in the average number of nodes and edges across all code smell datasets. For the *Complex Method* and *Complex Conditional* smells, the reduction is particularly significant, with node and edge counts decreasing by approximately 18-20% for both the R_2 and R_3 representations. The effect is less pronounced but still present for *Feature Envy* and *Multifaceted Abstraction*, where the graphs are inherently larger as they represent entire classes. Similarly, for the vulnerability detection task on the Devign dataset, there is a reduction in graph size for both pruned representations compared to the baseline. R_2 reduced the average node count by 4.86% and the edge count by 4.39%. R_3 resulted in a 2.40% reduction in nodes and a 2.85% reduction in edges.

Summary: Results confirm that CONCORD's heuristics generate more compact graph representations, directly addressing the goal of improving the efficiency of training GNN models by reducing the complexity of their input data.

C. RQ3: Impact of Heuristics on Performance

Although the simplification is not targeted to improve performance, but to improve efficiency and convenience, the changes in the representation influence the performance that we study in this research question.

On the *Complex Method* task, the baseline representation R_1 achieved the highest F1 score of 50.95%. However, R_3, which prunes print statements, performed nearly as well with an F1 score of 50.56%. This suggests that for *Complex Method* detection, the full AST representation is slightly more informative, but removing print statements minimally impacts performance.

Regarding the *Complex Conditional* task, we observe a different pattern. Here, R_3 outperformed both R_1 and R_2, achieving the highest F1 score of 46.15%. This improvement, albeit modest, indicates that removing print statements may help in focusing on the structural aspects relevant to identifying complex conditionals. Interestingly, R_2, which removes simple assignments, performed worst with an F1 score of 42.24%, suggesting that these assignments may carry valuable information for this particular code smell.

The *Feature Envy* detection task showed the most dramatic improvement with pruning. R_3 significantly outperformed both R_1 and R_2, achieving an F1 score of 60.69% compared to 57.85% for R_1 and 54.93% for R_2. This substantial improvement indicates that removing print statements is particularly effective for detecting *Feature Envy*, possibly by emphasizing the structural relationships between methods and attributes that are characteristic of this smell.

Conversely, *Multifaceted Abstraction* detection benefited most from the full AST representation (R_1), which achieved an F1 score of 42.21%. Both pruning strategies led to decreased performance, with R_3 (37.34%) performing better than R_2 (32.98%). This suggests that for identifying such a smell, retaining all nodes in the AST preserves more information, which would be lost through pruning.

For the vulnerability detection task, we observe that R_3 outperformed both R_1 and R_2, achieving the highest F1 score of 62.53%. This improvement over R_1 (60.33%) and R_2 (59.66%) indicates that removing print statements is beneficial for vulnerability detection, possibly by allowing the model to focus more on the structural and data flow aspects of the code that are relevant to identifying vulnerabilities.

Across all tasks, R_2 (simple assignment removal) consistently underperformed compared to R_1 and R_3; this suggests that simple assignments, despite their apparent simplicity, may contain valuable information for both code smell and vulnerability detection tasks. On the other hand, R_3 (print statement removal) generally performed well, achieving the highest F1 scores in 3 out of 5 tasks.

Summary: While the primary goal of simplification is efficiency, removing print statements R_3 often had the secondary benefit of enhancing or maintaining performance. It surpassed the full representation R_1 on the Complex Conditional, Feature Envy, and vulnerability detection tasks, which indicates that pruning these less informative statements helps the model focus on more structurally significant code aspects.

D. RQ4: Tradeoff between Performance and Number of computations

On the *Complex Method*, R_3 outperforms R_2 with an F1 score of 50.56% compared to 49.81%, while using slightly more FLOPs (0.165×10^9 vs. 0.162×10^9). The 1.5% improvement in F1 score comes at a cost of only 1.85% more FLOPs, making R_3 the better choice for this task. The marginal increase in computational cost can be justified by the performance gain.

In *Complex Conditional* R_3 shows a more significant advantage. It achieves an F1 score of 46.15%, substantially higher than R_2's 42.24%. This 9.3% improvement in F1 score is achieved with only a 1.2% increase in FLOPs (0.170×10^9 for R_3 vs. 0.168×10^9 for R_2).

The most drastic difference in this task is observed in the *Feature Envy* dataset. R_3 dramatically outperforms R_2 with an F1 score of 60.69% compared to 54.93%. This substantial 10.5% improvement in F1 score comes with a modest 3.2% increase in the number of FLOPs (0.260×10^9 for R_3 vs. 0.252×10^9 for R_2). Similar to what was observed in the *Complex Method* dataset, the significant performance gain can justify the small computational overhead, making R_3 the superior choice for *Feature Envy*. As for *Multifaceted Abstraction* detection, R_3 again proves to be more effective. It achieves an F1 score of 37.34% compared to R_2's 32.98%, with a 13.2% improvement. This comes at a cost of 2.4% more FLOPs (0.210×10^9 for R_3 vs. 0.205×10^9 for R_2).

In the Devign vulnerability detection task, R_3 continues to show advantages. It achieves an F1 score of 62.53%, outperforming R_2's 59.66%. This 4.8% improvement in F1 score is achieved with only a 1.7% increase in FLOPs (0.175×10^9 for R_3 vs. 0.172×10^9 for R_2).

Summary: While removing simple assignments (R_2) is the most computationally efficient (lower FLOPs), removing print statements (R_3) consistently achieves higher F1 scores across all tasks. This suggests simple assignments contain more valuable information, making R_3 a better compromise between performance and computational cost.

VI. Discussion

A. More Data does not Entail Higher Performance

As we have demonstrated in Section V, more data does not necessarily translate to higher performance. We have seen how with R_3 representation that had less information, managed to out perform R_1 in 3/5 tasks. To further investigate the impact of removing print statements, we select samples from the test set of the *Complex Conditional* dataset where the models trained on R_1 and R_3 exhibit disagreement in their classifications. Listing 6 presents a snippet where such a disagreement occurred[1]. The model trained on R_1 classifies the code snippet as smelly, whereas the model trained on R_3 classifies it correctly. Misclassification can be attributed to the *noise* that print statements introduce, as they lead the model to learn implicit patterns that potentially hinder the learning of an effective representation.

```
public TwoPassDataIndexer(...PARAMS) {
  TObjectIntHashMap predicateIndex;
  List eventsToCompare;
  predicateIndex = new TObjectIntHashMap();
  System.out.println(STRING);
  System.out.print(STRING);
  try {
    File tmp = File.createTempFile("events",
                                    null);
    tmp.deleteOnExit();
    int numEvents = ...;
  System.out.println(STRING);
  System.out.print(STRING);
    eventsToCompare = ...;
    // done with predicates
    predicateIndex = null;
    tmp.delete();
  System.out.println(STRING);
  System.out.print(STRING);
    sortAndMerge(eventsToCompare);
  System.out.println(STRING);
  }
  catch(IOException e) {
  System.out.println(e);
  }
```

Listing 6: An example of noisy print statements affecting the predictive performance.

[1]Original code taken from here http://bit.ly/446v0n7

Of course, the simplification strategies offered by DSL should not be considered as a silver bullet. For instance, it may be unwise to perform print statement removal if we would like to generate representations for code search that retrieves snippets that perform I/O operations. Continuing with the analogy with NLP, the same can be applied to stop words. If an NLP system is used for stylistic analysis (*e.g.,* authorship attribution) removing stop words may remove important stylistic markers, such as sentence structure and word choice.

B. CONCORD *and Large Language Models*

While Transformer-based models are becoming the dominant architecture, they process source code as a simple sequence of tokens, overlooking its rich structural and semantic nature. CONCORD positions itself not as a replacement, but as a complementary tool that can be integrated with LLM-powered systems to address this limitation. First, it can be used to create hybrid, graph-based inputs for advanced neural architectures. In this approach, CONCORD generates the graph structure, while a pre-trained LLM like can provide the rich vector embeddings for each individual node in the graph. This combines the explicit structural representation from CONCORD with the deep semantic understanding of a language model, creating a more expressive input for graph neural networks. Second, CONCORD can function as a pre-processing and simplification tool to optimize the data fed into LLMs. Through its heuristics to prune noisy and less informative statements, CONCORD creates a more concise and semantically focused version of the source code. This simplified representation can then be fed to an LLM, reducing input token length, which in turn lowers computational costs and can improve model performance by removing noise that might otherwise hinder the learning process. Through these integration pathways, CONCORD enhances LLM-based systems by providing them with more structured, efficient, and semantically rich data.

C. Other Applications of CONCORD

CONCORD's features can be leveraged for various downstream GNN-related and non-NN tasks. For machine learning applications like vulnerability detection or Type-4 code clone detection, users can generate tailored graphs by combining base representations (*e.g.,*, CFG, AST) with specific edge types (NEXTSIBLING, NEXTTOKEN) and node removal operations to filter noise. Beyond machine learning, the representations generated by the DSL are valuable for other engineering tasks. They can be used for code visualization to aid in comprehension and debugging, or for static analysis to identify optimization opportunities such as dead code or unused variables (*e.g.,*, using LASTUSE edges).

VII. THREATS TO VALIDITY

Internal validity: Internal validity threats are related to experiment errors. When we cloned the repositories, we took into account the snapshots that were before the publication date of the replication package released by Sharma *et al.* [6]. We did this to minimize the risk of code smells being refactored. We also kept the same positive-to-negative class ratio when constructing the datasets to avoid erroneous results as discovered by Chakraborty *et al.* [24]. Furthermore, to ensure fairness in evaluation, all models were trained and evaluated on the same splits. Finally, we set the same random seed across all models during the training and inference phases.

External validity: This type of threat is related to the generalizability of the method. Although we provided a use case for code smell detection and vulnerability detection, CONCORD can be integrated into many other tasks, such as clone detection and code search. Moreover, it supports multiple programming languages, unlike the aforementioned tools, which makes it flexible and can be used on multi-linguistic datasets and benchmarks.

Construct validity: We carefully constructed the R_1, R_2, and R_3 configurations to measure the effect of removing simple assignment and print statements across code snippets in isolation. As for metrics, we used adequate measures to evaluate classifiers and calculate their computational costs similar to related work [6], [58], [59].

VIII. RELATED WORK

The *python_graphs* [21] library constructs Python program graphs (control/data-flow, syntactic, lexical) with edge augmentations akin to CONCORD, while *feature-javac* [22] generates Java feature graphs for DL tasks. Both tools are language-specific, unlike CONCORD, which leverages *joern* for multi-language support and includes graph reduction to ease GNN training costs. Further, CONCORD enables configurability (selecting relationship types), unlike prior tools that fix included relationships. This flexibility supports diverse modeling objectives. Recent efforts such as COMEX [23] have simplified the generation of structural code representations by automating the extraction of multiple code-views (*e.g.,*, AST and CFG,). However, COMEX remains limited in several ways: it supports only predefined code-view combinations for Java and C#, lacks a mechanism to define new edge types or control structural relationships, and produces large, dense graphs without built-in simplification strategies. In contrast, CONCORD generalizes this process through a DSL that unifies graph construction across languages, supports fine-grained configurability, and incorporates graph simplification heuristics to enhance scalability while preserving essential semantics. In addition, COMEX is limited to Java and C#, whereas CONCORD offers a language-agnostic approach and supports multiple programming languages off the shelf.

IX. CONCLUSIONS AND FUTURE WORK

In this study, we propose CONCORD, a DSL for graph code representation construction. It is a method to build representations in a configurable and customizable way, and importantly, provides a set of simplification heuristics to reduce the size of the generated graphs. We demonstrate how it can be used in a real example to detect code smells and vulnerabilities. Our experiments show that it can effectively generate graph representations of source code and reduce computational costs without affecting the overall performance. In the future, we will investigate how we can apply CONCORD in other tasks in source code analysis, extend it further to accommodate other modeling techniques, and investigate its integration with other deep learning architectures such as language models of code.

X. Data Availability

We have made the **replication package** available online [37] for other researchers to use and extend the proposed DSL and framework.

References

[1] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A survey on machine learning techniques applied to source code," *Journal of Systems and Software*, vol. 209, p. 111934, 2024.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, July 2018.

[3] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[4] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," CODASPY '16, 2016.

[5] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[6] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, 2021.

[7] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering*, 2019.

[8] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning.," in *ENASE*, pp. 137–146, 2018.

[9] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," MSR '20, Association for Computing Machinery, 2020.

[10] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Aug. 2016.

[11] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 253–264, IEEE, 2021.

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong (YIMING), L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of EMNLP 2020*, September 2020.

[13] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping api elements for code migration with vector representations," ICSE '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

[15] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic code repair using neuro-symbolic transformation networks," *ArXiv*, vol. abs/1710.11054, 2017.

[16] L. Buch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (Los Alamitos, CA, USA), pp. 95–104, IEEE Computer Society, feb 2019.

[17] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to error-handling specification mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 423–433, Association for Computing Machinery, 2018.

[18] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794, 2019.

[19] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[20] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[21] D. Bieber, K. Shi, P. Maniatis, C. Sutton, V. J. Hellendoorn, D. D. Johnson, and D. Tarlow, "A library for representing python programs as graphs for machine learning," *ArXiv*, vol. abs/2208.07461, 2022.

[22] A. Rice, H. Mercer, K. Chandra, K. Chandra, and Y. Yu, "features-javac," 2018.

[23] D. Das, N. S. Mathews, A. Mathai, S. Tamilselvam, K. Sedamaki, S. Chimalakonda, and A. Kumar, "Comex: A tool for generating customized source code representations," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, 2024.

[24] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Transactions on Software Engineering*, sep 2022.

[25] D. Cui, S. Wang, Y. Luo, X. Li, J. Dai, L. Wang, and Q. Li, "Rmove: Recommending move method refactoring opportunities using structural and semantic representations of code," *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022.

[26] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 261–271, 2020.

[27] J. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 554–565, 2022.

[28] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," ICSE '18, 2018.

[29] A. Liu and M. Coblenz, "Debugging Techniques in Professional Programming," 3 2023.

[30] I. Dejanović, R. Vaderna, G. Milosavljević, and v. Vuković, "TextX: A Python tool for Domain-Specific Languages implementation," *Knowledge-Based Systems*, vol. 115, pp. 1–4, 2017.

[31] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," *SIGPLAN Not.*, vol. 39, p. 111–122, jan 2004.

[32] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, IEEE, May 2014.

[33] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[34] J. Gu, Z. Chen, and M. Monperrus, "Multimodal representation for neural code search," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 483–494, 2021.

[35] R. Koppler, "A systematic approach to fuzzy parsing," *Softw. Pract. Exper.*, vol. 27, p. 637–649, jun 1997.

[36] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering* (K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M.-E. Fokaefs, and M. Zhou, eds.), pp. 261–271, 2020.

[37] Anonymous, "Replication package for Concord DSL and framework." https://github.com/concord-dsl/CONCORD, 2025.

[38] M. Fowler, *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 ed., 1999.

[39] M. Perepletchikov and C. Ryan, "A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software," *IEEE Transactions on Software Engineering*, vol. 37, pp. 449–465, Aug. 2011.

[40] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their

impact on software maintenance," in *IEEE International Conference on Software Maintenance, ICSM*, pp. 56–65, Universita di Salerno, Salerno, Italy, IEEE, Dec. 2012.

[41] A. Ikama, V. Du, P. Belias, B. A. Muse, F. Khomh, and M. Hamdaqa, "Revisiting the impact of anti-patterns on fault-proneness: A differentiated replication," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 56–65, 2022.

[42] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, pp. 243–275, June 2012.

[43] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some Code Smells Have a Significant but Small Effect on Faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, pp. 33–39, Sept. 2014.

[44] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 351–360, Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, 2013.

[45] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[46] H. Nandani, M. Saad, and T. Sharma, "Dacos—a manually annotated dataset of code smells," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 446–450, 2023.

[47] J. Shi, Z. Yang, B. Xu, H. J. Kang, and D. Lo, "Compressing pre-trained models of code into 3 mb," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, (New York, NY, USA), Association for Computing Machinery, 2023.

[48] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), 2022.

[49] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: Security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 462–472, 2017.

[50] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[51] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, 2019.

[52] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, *Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks*. 2019.

[53] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2016.

[54] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[55] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[56] M. Saad, J. A. H. López, B. Chen, D. Varró, and T. Sharma, "An adaptive language-agnostic pruning method for greener language models for code," *Proc. ACM Softw. Eng.*, June.

[57] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, "Regvd: revisiting graph neural networks for vulnerability detection," ICSE '22, 2022.

[58] Z. Zhang, H. Zhang, B. Shen, and X. Gu, "Diet code is healthy: simplifying programs for pre-trained models of code," *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[59] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, "Understanding neural code intelligence through program simplification," ESEC/FSE 2021, (New York, NY, USA), p. 441–452, Association for Computing Machinery, 2021.