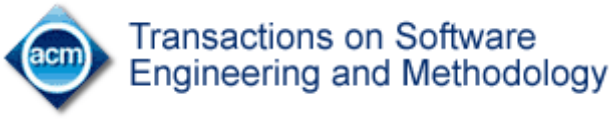

CONCORD: Towards a DSL for Configurable Graph Code Representation

Journal:	*Transactions on Software Engineering and Methodology*
Manuscript ID	TOSEM-2023-0377
Manuscript Type:	Journal-First Paper
Date Submitted by the Author:	18-Oct-2023
Complete List of Authors:	Saad, Mootez; Dalhousie University Faculty of Computer Science, Computer Science Sharma, Tushar; Dalhousie University,
Computing Classification Systems:	Software and its engineering, Artificial intelligence, Computing methodologies

CONCORD: Towards a DSL for Configurable Graph Code Representation

MOOTEZ SAAD, Dalhousie University, Canada
TUSHAR SHARMA, Dalhousie University, Canada

Deep learning is widely used to uncover hidden patterns in large code corpora. To achieve this, constructing a format that captures the relevant characteristics and features of source code is essential. Graph-based representations have gained attention for their ability to model structural and semantic information. However, existing tools lack flexibility in constructing graphs across different programming languages, limiting their use. Additionally, the output of these tools often lacks interoperability and results in excessively large graphs, making graph-based neural networks training slower and less scalable.

We introduce CONCORD, a domain-specific language to build customizable graph representations. It implements reduction heuristics to reduce graphs' size complexity. We demonstrate its effectiveness in code smell detection as an illustrative use case and show that: first, CONCORD can produce code representations automatically per the specified configuration, and second, our heuristics can achieve comparable performance with significantly reduced size. CONCORD will help researchers a) create and experiment with customizable graph-based code representations for different software engineering tasks involving DL, b) reduce the engineering work to generate graph representations, c) address the issue of scalability in GNN models, and d) enhance the reproducibility of experiments in research through a standardized approach to code representation and analysis.

Additional Key Words and Phrases: Code representation, domain-specific language, code smell detection, source code analysis

ACM Reference Format:
Mootez Saad and Tushar Sharma. 2023. CONCORD: Towards a DSL for Configurable Graph Code Representation. 1, 1 (October 2023), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, deep learning (DL)-based methods for source code analysis have gained significant momentum [1, 56]. Such methods have been applied in various software engineering tasks, such as defect prediction [39], [22], [70], code smell detection [55], [41], [24], code summarization [27], [29], [6], and program synthesis [1, 56]. Generating code embeddings is a common step in these approaches, where code is converted from its raw textual form to a numeric representation that can be processed and fed to DL-models [17, 31, 49, 55]. However, treating source code as a stream of tokens ignores its rich structure and semantics [3], [14], [16], [10]. Hence, efforts have been made to incorporate the code's structural and semantic features into its vector representation [3, 10, 14, 16].

To further leverage the power of semantics, researchers are experimenting with graph-based neural network architectures [67, 69] that can better capture different semantic aspects (*e.g.,* control

Authors' addresses: Mootez Saad, Dalhousie University, Canada, mootez@dal.ca; Tushar Sharma, Dalhousie University, Canada, tushar@dal.ca.

XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.* [2] augmented AST with custom *data flow edges* and *control flow edges* to detect variable misuse in a code snippet. The process of combining code representations, pre-processing them, and training a DL model is done manually and from scratch which wastes significant research time and resources.

Current tools for constructing graph representations of source code face challenges due to different parsers used, resulting in varied ASTs representation and hindering effective combinations. Lack of interoperability arises from generating graphs in different formats (JSON, GraphML), and functionality fragmentation occurs due to unique edge augmentation operations introduced by each tool. Combining these representations or artifacts requires considerable effort, as noted by Siow *et al.* [58].

Another significant problem in generating graphs from source code is the high size complexity such structures can attain. To demonstrate this issue, let us consider a small code snippet and its corresponding AST:

Listing 1 Example of a C-like code snippet

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

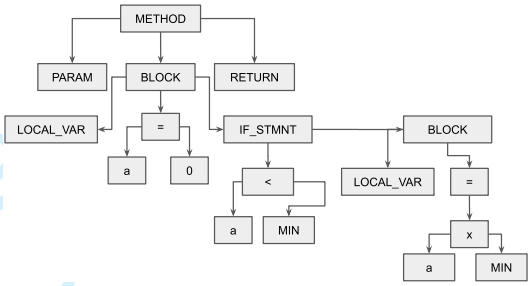

Fig. 1. AST of Listing 1

Despite being only four lines long, this code results in an AST containing 19 nodes as shown in Figure 1.

An AST can only capture the structural aspects of code, which is why it might be required to include other relevant attributes to represent control and data flow. Therefore, in order to effectively model these properties, additional structures such as Control Flow Graphs (CFGs) need to be incorporated. However, this augmentation further aggravates the issue of complexity in terms of graph size.

Graph Neural Networks (GNNs) require large memory and computational resources during training, which renders them computationally expensive to train and difficult to scale. This is because GNNs operate on graph data, which is often more complex and larger than other types of data as we have shown in the previous example. During training, a GNN must store and manipulate large adjacency matrices representing the graph. Additionally, these models typically perform multiple rounds of message passing or graph convolution operations adding further to memory requirements. This entails that computational requirements are directly proportional to the size of the graph and the number of layers in a GNN. Hence, finding reduction techniques is critical to make the training tractable.

A codebase may contain *noisy* statements that do not contribute to its core logic. Finding unrelated print statements is natural: often, developers use print statements for debugging and tracing errors. Studies have shown [8, 40] that professional software developers rely on print statements such as `printf` as a debugging method. These statements do not relate to the overall semantics because

debugging is not the primary goal of the said snippet. We can claim the same about temporary variables. We introduce temporary variables to make source code more readable for others, however, this also comes at a cost by making the program longer. One might conjecture that the removal of such statements can *reduce* source code and allow the model to learn better its implicit patterns and salient features to capture its intended behaviour. Moreover, such modifications to the code while generating its representation highly depend on the downstream task. The existing tools and technologies do not offer any customizable way to generate code representation automatically.

This paper introduces CONfigurable Code Representation (CONCORD), a Domain Specific Language (DSL) and a unifying framework designed to automate the generation of custom code representations. The DSL automatically generates code representations from source code and as per the specified configuration. By leveraging CONCORD researchers can easily obtain customized code representations for their downstream tasks without the need to handle intricate code representation details. This streamlined approach not only enhances overall efficiency but also reduces the risk of errors. Experimental results demonstrate that our proposed method *achieves improved model performance at a lower computational cost.* We applied CONCORD to solve the code smell detection task, and managed to maintain up to 100% performance and reduce computations by 10.15%.
We summarize the key contributions of this work below.

- We provide a one-stop solution in the form of a DSL that unifies multiple edge augmentation techniques from previous works to produce rich and expressive graph representations in a *composable*, *configurable*, and *cross-linguistic* manner.
- We propose and implement two pruning techniques to reduce the size of code graphs to make training more tractable.
- We present a case study to demonstrate the capabilities of CONCORD to apply on code smell detection task.
- We show how the proposed pruning techniques can reduce size and computational time for GNN training while preserving performance.
- We have made the **replication package** available online [5] for other researchers to use and extend the proposed DSL and framework.

2 BACKGROUND AND RELATED WORK

2.1 Graph Neural Networks

Graph Neural Networks [54] operate on graph-structured data, using a message-passing mechanism to learn node and edge representations. The process involves computing messages, aggregating them, and updating node representations iteratively. They have been used in the context of Software Engineering to solve different tasks, specifically those that focus on source code analysis. Such works include but are not limited to defect prediction [11, 69], [13], [44], [37], clone detection [62], [45], code search [43], [66], [61], and code summarization [36], [42]. The reason behind the prevalence of using such architecture is that source code is often modelled using graphs like in the case of syntax trees, control flow graphs and call graphs. Such structures encompass properties that cannot be reflected if the source code was represented as a flat sequence of tokens.

2.2 Domain Specific Languages

A domain-specific language (DSL) is a specialized language that uses concepts and rules of a domain [20]. DSLs are divided into two major categories—*external* and *internal*. External DSLs like CSS and SQL, have their own syntax and parser that interprets the language or translates it into another language. Internal DSLs are written on top of an existing general-purpose language such as libraries or APIs. The proposed DSL CONCORD falls into the first category. There have been many DSLs

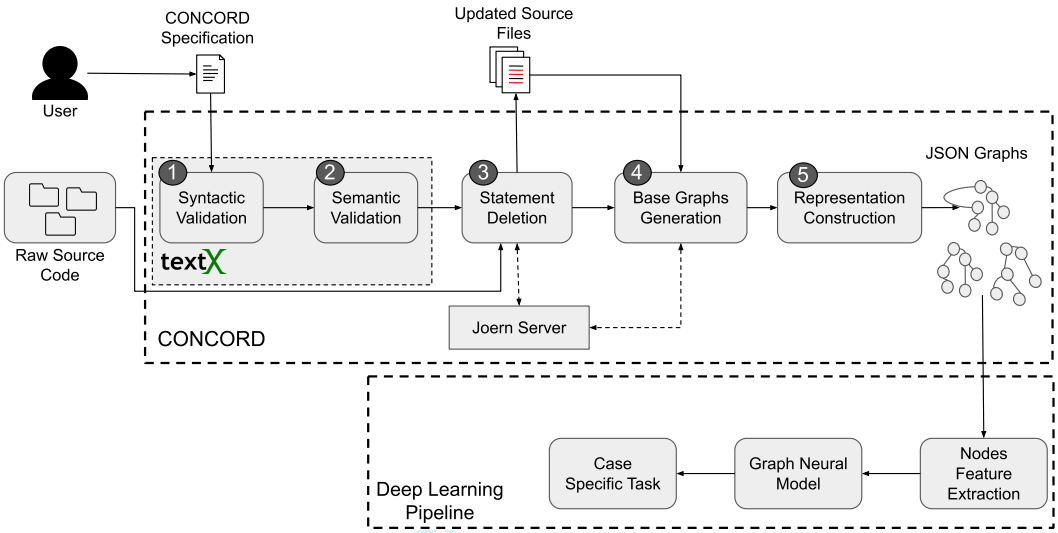

Fig. 2. Overview of CONCORD

proposed in software engineering literature [59]. For example, Pig Latin [50] is a textual DSL that abstracts the MapReduce implementation. The DSL code is translated by the compiler into MapReduce jobs that are executed by the infrastructure provided by Apache Hadoop. Andrzejak *et al.* proposed NLDSL [4] that is used to create chain and pipeline operations for data processing. Giner-Miguelez *et al.* [21] presented a DSL to precisely describe machine learning datasets, aligning with the data-centric cultural shift in the ML community. The DSL aids in dataset selection and result reproducibility, with a Visual Studio Code plugin for usability. Case studies and experiments validate its expressiveness and potential impact on achieving higher-quality ML models, particularly in terms of fairness, diversity, and bias mitigation. Morales *et al.* [46] have proposed a DSL to model AI engineering processes, tailored for multidisciplinary teams developing AI-embedded software. It provides a structured framework and common ground to address communication issues and promote best practices. Its goal is to facilitate the formalization of AI processes within organizations and seamlessly integrate with existing model-driven tools, advancing the adoption of AI engineering practices.

2.3 Tools for code graphs construction

Bieber *et al.* [9] provide an open-source Python library called *python_graphs* to construct graph representations of Python programs suitable for training machine learning models by employing static analysis. The library is capable of constructing control-flow graphs, data-flow graphs, and composite "program graphs" that combine control-flow, data-flow, syntactic, and lexical information about a program. Rice *et al.* [53] 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 has been designed to facilitate machine learning on source code research.

Limitations of existing work: The first limitation is that each tool supports one programming language, either Python or Java, thereby limiting their applicability to those languages only. Second is the lack of *composability* and *configurability*. These tools produce representations that include all types of edge augmentations at once. In other words, users cannot specify configurations that

specify what augmentations to be included or excluded. Finally, the set of augmentations in these tools is disjoint, *i.e.,* a list of operations is found in one tool but missing in another. This fragmented nature of functionalities makes it difficult to inter-operate the structures yielded by these methods.

2.4 Source code reduction

Zhang *et al.* [68] introduce an approach referred to as *DietCode* that leverages large pre-trained models such as CodeBERT [17] for source code. Through an empirical analysis of CodeBERT's attention mechanism, they discovered that the model attends more to certain types of tokens and statements such as keywords and data-relevant statements. Based on this finding, they proposed three strategies for simplifying CodeBERT's input program: *word dropout*, *frequency filtering*, and an attention-based approach that *selects statements and tokens* with the highest attention weights. They tested the efficacy of *DietCode* on code search and code summarization tasks and found that it performs comparably to CodeBERT while requiring 40% less computational cost during fine-tuning and testing. Rabin *et al.* [52] propose *sivand*, which employs delta debugging to identify the most critical code segments for a deep learning model. This technique involves a deep learning model receiving a code snippet as input, which is subsequently segmented. The neural network model then processes each segment for a downstream task, such as method name prediction. If a segment achieves a desirable score, it is further divided. This iterative process continues until the subset's performance fails to meet the targeted score. Ultimately, the algorithm produces the smallest code snippet that satisfies the model's objective.

Limitations of existing work: The limitation of these works is that they require multiple rounds of expensive computation. Specifically, *sivand* requires the execution and evaluation of a deep learning model at each stage of program reduction, which can be inefficient if the data scales upwards. Similarly, at least one forward and backward pass (*i.e.,* fine-tuning) is needed by *DietCode* to compute the attention scores of each token. This overhead operation is computationally demanding given that the number of parameters (125M and 220M) in the considered Language Models.

As we have discussed, graph neural networks have the capacity to better model source code. However, the current state of tooling that addresses the challenges such models bring is limited. The aim is to propose a new solution in the form of a domain-specific language that enables flexible and configurable graph code modeling to address such limitations.

3 CONCORD DSL 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. To train a GNN model, a user first generates a graph representation of the source code. They provide a Concord configuration which is interpreted, and the language's backend generates the concrete samples. These samples can later be used for training and inference. The language grammar is defined and interpreted using *textX* [15]. A DSL is typically characterized by a metamodel, which serves as a representation of its domain-specific entities and their interconnections. Figure 3 illustrates Concord's metamodel and the relationship between the DSL's elements.

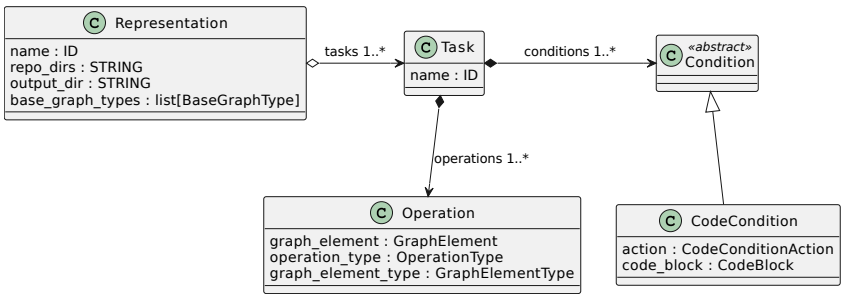

Fig. 3. CONCORD's metamodel excerpt

Concord's grammar is formalized by a Parsing Expression Grammar (PEG) [18]. 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. In Table 1 we present the different grammar rules and their body. The full grammar specification can be found in the replication package.[1]

Table 1. Grammar Rules for CONCORD

Rule	Rule body
BaseGraphType	AST|CFG|PDG
CodeBlock	catch|for|while|if|else
CodeConditionAction	exclude|include
Comment	\/*(.|\n)*?*\/|\/\/.*?$
EdgeType	next_token|next_sibling|for_cfg|while_cfg|last_read_write| guarded_by|returns_to|computed_from|last_lexical_use
GraphElement	Node|Edge
NodeType	print|logging|sys_exit|simple_assignment
OperationType	add|remove

3.1 DSL key elements

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

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 Table 1). 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.

[1]https://bit.ly/3Kb5u84

3.2 Graph generation process

Syntactic validation: In the syntactic evaluation, *textX* [15] constructs a model from the CONCORD configuration file and checks whether it conforms to the grammar rules and the metamodel of the language.

Semantic validation: We ensure the correctness of representation semantics. For example, `Node remove next_token` is syntactically correct, but it is semantically invalid since *next_token* is an edge addition operation. The framework warns the user when it detects such issues.

Statement deletion: In this step, we prune the statements specified by the user at the file level.

Base graphs generation: Once the configuration passes the validation steps and statements are pruned, we generate the base graph representations by communicating with *joern* [64] (a platform for code analysis that supports C/C++, Java, Javascript, Python and PHP), through its HTTP server. The generated representations are stored in JSON format.

Representation construction: Finally, we import the base graphs and construct the representation using the *NetworkX* library [25]. The framework generates code representations as per the specification in the configuration file. We store code representations in JSON format, which can be parsed by NetworkX, allowing further processing, if needed.

3.3 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.* [69] 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.* [23] 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 (*i.e.,* statement-level removal) scale. The three 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 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 for 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 in order 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 4, 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.

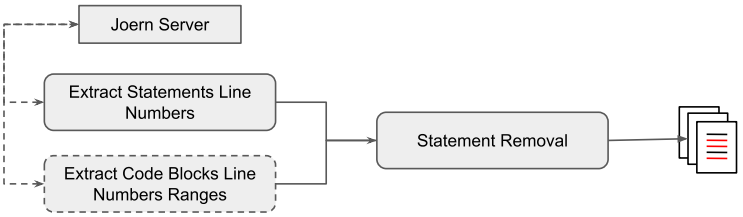

Fig. 4. Statement removal process using CONCORD

Algorithm 1 presents the procedure for identifying simple assignment statements. After loading the source code into *joern*, we extract the AST of each method. Then, for each tree, we iterate over its nodes and return those 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 have only one node of type `IDENTIFIER`.
(3) The non-leaf nodes of its right subtree represents 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 node is simple assignment

```
function isSimpleAssignment(root)                    ▷ Returns whether an assignment node is simple
    condition1 ← root.type == "ASSIGNMENT"
    condition2 ← root.children.Count(isIdentifier) == 1
    nonLeafNodes ← GetNonLeafNodes(root)
    condition3 ← nonLeafNodes.Count(!isAllowedOperation) == 0
    leafNodes ← GetLeafNodes(root)
    condition4 ← leafNodes.Counter(isLiteral) == leafNodes.size()
    return condition1 && condition2 && condition3 && condition4
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, greaterEqualsThan, lessEqualsThan}
    allowedOperations ← arithmeticOperation + bitwiseOperation + logicalOperation + relationalOpera-
tion
    return node.type ∈ allowedOperations
end function
```

Although it is sufficient to check whether the right subtree is composed of only one node of type `IDENTIFIER`, we implemented it 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 type of statements are rarely found in practice, we did so for the sake of completeness.

When removing simple assignment statements we do not remove the entire source code line, rather, we replace the statement at that specific line with an empty string. To explain why we follow such an approach, let us consider the example given in Listing 2.

Listing 2 Simple assignment statement to be removed

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

If we were to remove the entire line, the resulting code would be completely erroneous, as seen in Listing 3, with mismatched opening and closing brackets.

Listing 3 Incorrect removal of the statement if the entire line is deleted

```
public static void foo(String[] args) {

     System.out.println("Printer");
   }
 }
```

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.

Listing 4 How the statement is removed by CONCORD

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

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.

3.4 Edge addition

The next major operation provided by CONCORD is edge addition. Edges generated from base representations, such as ASTs and PDGs, capture certain aspects of the source code. 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 [2, 63, 69] have proposed additional sets of edges that can augment the base representations for various tasks. In the rest of the sections, we will provide a detailed description of each type of edge that can be added.

3.4.1 NEXTSIBLING. Used by Wenhan *et al.* [63], 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.

3.4.2 NEXTTOKEN. This edge connects terminal nodes of an AST with each other. Allamanis *et al.* [2] introduced this edge type in their study; which was also used by Wenhan *et al.* [63]. The rationale for introducing the edge is that an AST does not reflect the sequential nature of the source code. Let us consider Figure 5 that represents the syntax tree of this C-like statement `int i = 1+1`

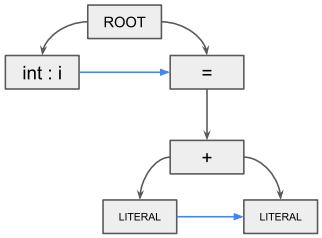

Fig. 5. Example of a NextToken edges in an AST

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

3.4.3 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 [2]. The LastRead edges create a connection between these nodes in the set U and `var`.

3.4.4 LastWrite [2]. It is similar to LastRead, except the set U would contain the tokens where `var` could have been last written to.

3.4.5 LastLexicalUse [2]. 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.

3.4.6 ComputedFrom [2]. 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 5 both of the literal leaf nodes will be connected to the variable `i` by this edge.

3.4.7 ReturnsTo [2]. Connects a method declaration node in the AST with the node that refers to the method's `return` statement.

3.4.8 GuardedBy *and* GuardedByNegation [2]. 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 6 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;}`
The green edge refers to a GuardedBy relation, whereas GuardedByNegation is represented by the blue edge.

3.4.9 WhileCFG *and* ForCFG. Both of these relations were introduced by Wenhan *et al.* [63] 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 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.

Through the implementation of all of these operations, Concord provides a unifying tool bench for code graph constructions to deal with the address the current fragmented nature of tools for building such structures.

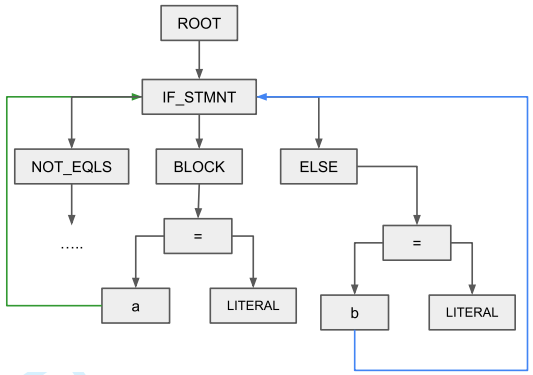

Fig. 6. Example of GuardedBy and GuardedByNegation edges in an AST

4 CASE STUDY: CODE SMELL DETECTION

To demonstrate how Concord can be used effectively, we conducted a study to prepare code representations to classify code smells. Code smells are symptoms of sub-optimal design and implementation choices manifested in the form of poorly written code that requires refactoring [19]. Their presence in a given software system hinders its quality, specifically maintainability [7, 51] and reliability [26, 28, 30, 32]. Hence, many efforts have been made to produce tools and methods to detect them [57]. However, an important aspect of code smells is that they are subjective in nature [48, 57]. 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 where we showcase how the features of Concord can be put into use in the task of code smell detection.

In this study, we consider four types of code smells that were initially investigated by Sharma *et al.* [55]: *Complex method*, *Complex conditional*, *Feature envy*, and *Multifaceted abstraction*. We formulate the problem as a one-vs-all binary classification problem, meaning that each code smell has its own dataset where each sample can be either categorized as smelly or non-smelly, which is the commonly adopted way in related literature.

4.1 Data preparation

Sharma *et al.* [55] carried out an empirical study to explore the feasibility to detect code smells using deep learning techniques without feature engineering. They identified a set of 922 Java repositories after applying a filtering criteria using RepoRepears' [47] eight repository characteristics. We randomly selected and cloned a set of 100 projects from the repositories used by Sharma *et al.* [55]. To ensure greater reliability, we selected the snapshot of each project before the replication package by Sharma *et al.* [55] was published. Specifically, we checked out the commit prior to January 26, 2019, to reduce the possibility of refactoring that could have led to the removal of code smells.

To demonstrate the capabilities of Concord and evaluate the pruning strategies mentioned in Sections 3.4 and 3.3, we define three Concord configurations: R_1, R_2, and R_3 specified in Table 2. In Listing 5 we show how R_2 is expressed using the Concord syntax. All of the specification files for these representations can be found in the replication package[2].

We select these configurations to capture different aspects of the source code using the DSL. Specifically, we utilize AST as the base representation to capture the structural properties of the code for the three representations. The NextToken edge is used to model its sequential nature,

[2]https://bit.ly/3Q9LpTo

Listing 5 R_2 specification using CONCORD's syntax.

```
Tasks {
    task2 {
        Edge add next_token
        Edge add for_cfg
        Edge add while_cfg
        Edge add computed_from
        Edge add guarded_by
        Edge remove simple_assignment
        Conditions {
            exclude while_block
            exclude if_block
}}}
Representations {
    r2 {
        "/dir/repos_list.csv"
        "output_dir"
        AST
        task2
}}
```

while the WHILECFG, FORCFG, GUARDEDBY, and COMPUTEDFROM edges are specified to capture the flow of control and data through the program.

Table 2. Specification of each representation.

Nodes: ✓means that such statement type (*i.e.*, set of nodes) is kept, whereas, ✗ signifies that it is removed.
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

The representations R_2 and R_3 differ from R_1 as they specify a pruning strategy. After executing CONCORD using each of the three configurations, we generate an initial collection of $\approx 1.5M$ methods, with $\approx 500K$ samples for each configuration. We then filter out methods generated by R_1 that did not contain at least one print and a simple assignment statement. 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.* [55]. Finally, we perform an 80:10:10 split corresponding to training, validation, and test using stratified sampling to ensure fair evaluation for each classifier. Table 3 presents a summary of the training dataset with the class distribution for each dataset at each split.

Table 3. Statistics of datasets

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

Table 4 tabulates the average number of nodes and edges in each dataset after running CONCORD using the three mentioned configurations.

Table 4. Average number of nodes and edges for each dataset under each representation

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

One may observe the difference in the number of nodes and edges between the implementation smells (*Complex Conditional/Method*) and design smells (*Multifaceted Abstraction* and *Feature Envy*). This arises because we represent the graph of a design smell instance (*i.e.,* a class) by combining the graphs of its methods. Consequently, on average, design smells have a higher number of nodes compared to its implementation counterpart.

During the construction of the datasets, we ensured the presence of a mapping with the aid of the original dataset provided by Sharma *et al.* [55]. In particular, for each code smell dataset, a CSV file was used to represent the mapping between each code snippet (*i.e.,* method) and its corresponding graph representations generated using R_1, R_2, and R_3.

Table 5. Example of how datasets are stored

concord_id	project	baseline_file	r1_file	r2_file	r3_file	label	split
23731	ganttproject	hasCdata.code	hasCdata_21.json	hasCdata_28.json	hasCdata_24.json	0	train
421	greenmail	clear.code	clear_42.json	clear_58.json	clear_23.json	0	test
…	…	…	…	…	…	…	…
10806	c5	clear.code	clear_108.json	clear_107.json	clear_104.json	0	val

Furthermore, the CSV file includes additional columns to indicate the type of split (train, validation, or test). This approach was adopted to ensure that the models are trained and evaluated on the same data, thereby ensuring a fair evaluation. In Table 5 we show a snapshot on how these CSV files are stored.

4.2 Model description

We employ a Gated Graph Neural Network (GGNN)-based [38] classifier to solve the code smell classification task. We select this architecture over other variants such as Graph Convolutional Networks (GCN) [34] or Graph Attention Networks (GAT) [60], due to its superior performance demonstrated in other software engineering tasks [58] including clone detection, code classification, and vulnerability detection.

Fig. 7. Gated Graph Neural Network-based classifier

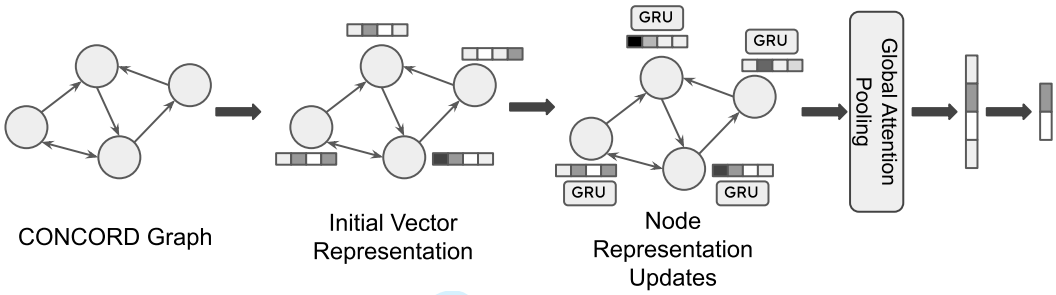

Figure 7 illustrates the end-to-end training process of the classifier. We take the graph yielded by CONCORD and generate an initial node feature matrix V using the pre-trained embedding layer of CodeBERT [17]. Once created, we feed V and the adjacency matrix E to the GGNN layer. In this layer, a Gated Recurrent Unit (GRU) [12] is created for each vertex in V. The GGNN updates the hidden states of the nodes in an iterative manner, using the following equations:

$$a_v^t = \sum_{u \in \mathcal{N}(v)} W_{\text{edge}} \cdot h_u^{t-1}$$

$$z_v^t = \text{GRU}_{\text{update}}(a_v^t, h_v^{t-1})$$

$$h_v^t = (1 - z_v^t) \odot h_v^{t-1} + z_v^t \odot z_v^t$$

where:

- a_v^t is the sum of edge weights for node v at time step t,
- W_{edge} is a learnable weight matrix for the edges,
- $\mathcal{N}(v)$ represents the neighbourhood of node v in the graph,
- $\text{GRU}_{\text{update}}(\cdot)$ is the update function of the GRU,
- z_v^t is the update gate for node v at time step t,
- h_v^t is the hidden state of node v at time step t, and
- $\odot$ represents element-wise multiplication.

After t time steps, the hidden representations of all the nodes for all the graphs are updated. We map the feature matrix V^t to a single vector since we are solving a graph-level classification problem. We employ a *Global Attention Pooling* layer [38] that computes an attention score for each node in the graph.

Finally, we pass the resulting pooled vector into an MLP layer to perform the final classification. Table 6 lists all of the hyper-parameters that were used to train the classifier. Our choice of

hyperparameters is inspired by the experiments conducted by Siow *et al.* [58] and Chakraborty *et al.* [11].

Table 6. Hyper-parameters and values

Hyper-parameter	Value
Learning rate	1e-3
GGNN Layer Input Size	768
GGNN Layer Hidden Size	128
MLP Layer Input Size	128
MLP Output Size	2
Batch size	128
Number of Epochs	100
Patience	15

4.3 Research question

We train a classifier for each code smell-representation combination, giving a total of 12 trained models. Given it is a classification problem, we compute precision (P), recall (R), and (F_1) measure. The F_1 measure metric is considered insufficient for such classification tasks [65]; hence, we also include Matthews Correlation Coefficient (MCC) metric. Furthermore, we compute FLOPs (floating operation points) to measure the model's complexity. The lower the FLOPs value, the lower the complexity, in turn, the lower the processing. Through the conducted experiments, we would like to answer the following research question.

RQ. *Does reducing the size complexity of code representation affect the performance and the needed computational operations?*

Answering this research question will highlight the size reduction achieved by CONCORD framework and its effect on aspects such as computational cost and performance.

4.4 Results

Table 7 shows the performance of the models trained on datasets generated from the three representation configurations. We report the average over five runs with different initialization.

Table 7. Classification results for each representation. P: Precision, R: Recall, MCC: Matthews Correlation Coefficient, FLOPs: Floating Point Operations.

	R1					R2					R3				
	P	R	F1	MCC	FLOPs ($\times 10^9$)	P	R	F1	MCC	FLOPs ($\times 10^9$)	P	R	F1	MCC	FLOPs ($\times 10^9$)
Complex Method	0.53	0.47	0.47	0.47	0.171	0.42	0.35	0.36	0.36	0.162	0.47	0.41	0.41	0.41	0.165
Complex Conditional	0.48	0.43	0.44	0.44	0.175	0.43	0.40	0.40	0.40	0.168	0.47	0.45	0.45	0.45	0.170
Feature Envy	0.53	0.63	0.55	0.47	0.300	0.51	0.67	0.53	0.46	0.252	0.82	0.48	0.56	0.54	0.260
Multifaceted Abstraction	0.52	0.16	0.24	0.25	0.232	0.52	0.13	0.21	0.21	0.205	0.52	0.13	0.21	0.23	0.210

We obtain the average Precision, Recall, F1 measure and MCC as 0.515, 0.423, 0.425, 0.408 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 task, one possible explanation for obtaining such values could be the extreme class imbalance of each dataset. Table 3 shows that the smelly samples accounts for only ~ 4% of the total sample size on average. Given

that we retained the real-life distribution of smelly and benign sample ratio, this adds robustness to our evaluation. Indeed, experimenting with datasets that do reflect the actual distribution of anomalies can lead to misleading results. Chakraborty *et al.* [11] showed how state-of-the-art GNN-based models used for vulnerability detection failed to yield the same results when trained and evaluated on representative data, and their performance dropped by more than 50%.

Key observation 1: Graph Neural Network-based models seem to yield a reasonable performance on data that maintains real-world code smell distribution where the presence of smelly snippets is rare. The results indicate that our proposed approach generates code representation that can effectively capture the desired code features for code analysis tasks.

Models trained on the R_2 representation show on average 0.47, 0.38, 0.37, and 0.357 in terms of the classification metrics. The models attain around 88% (in terms of MCC) of the original performance compared to the models that were trained using R_1. We performed a paired t-test to measure whether such a difference is significant. We defined the null hypothesis as H_0: *The mean difference is zero (i.e., there is no significant change in* MCC *after removing simple assignment statements)* and the alternative hypothesis as H_1: *the mean difference is not zero (i.e., there is a significant change in MCC after removing such statements).* We found that the *p-value* = 0.099 > 0.05, hence, we conclude that there is no significant difference in the compared MCCs. In addition, these models exhibit such performance with lower size complexity (*i.e.,* nodes and edges) graph samples and less computation. Table 4 shows that graphs under the R_2 representation have 11.3% fewer nodes and 12% fewer edges on average. Consequently, the computational load is reduced by approximately 2.27×10^7 FLOPS.

Key observation 2: The elimination of basic assignment statements have resulted in a decrease in the size and complexity of the samples and the associated computational expenses needed for model training. Nonetheless, the models maintained roughly 88% of their predictive performance, though such a difference is not statistically significant.

Interestingly, upon removing print statements as specified in R_3, the trained models maintain their performance on par with the models trained using R_1 representation (in terms of MCC). The removal of print statements also resulted in an increase of MCC scores for the *Complex Conditional* and *Feature Envy* code smells by 2.2% and 14.9%, respectively. Additionally, the models achieved comparable results for the *Complex Method* and *Multifaceted Abstraction* code smells, with MCC scores of 0.41 and 0.47, and 0.23 and 0.25, respectively. Similar to the R_2 representation, the R_3 graphs are smaller in size, having 8% fewer nodes and 9% fewer edges compared to R_1. Therefore, graphs generated from R_3 require less computation, with the reduction of floating point operations by approximately 1.87×10^7 operations.

Key observation 3: Adopting `print` statement removal as a pruning technique has enabled the models to retain their predictive performance at a rate of 100%. In addition to maintaining this high-performance rate, the approach has also resulted in reduced graph sample sizes, leading to lower computational costs.

Both strategies *i.e.,* simple assignment removal (in R_2) and print statement removal (in R_3) reduce computational costs by decreasing the number of FLOPS performed. The R_2 representation requires approximately $\times 1.02$ fewer operations than R_3. However, in terms of performance, R_3 is superior as illustrated in the results. It manages to maintain the same level of performance as the models trained on the original data, whereas R_2 did not fully attain the performance of R_1. The choice between these strategies depends on a specific goal. If the objective is to reduce computation, adopting

the simple assignment strategy could be a suitable option, as it can, to some extent, maintain the original performance. However, if the goal is to balance both performance and computation, removing print statements might be a better option. This strategy reduces computation while fully preserving performance.

Key observation 4: Both pruning strategies have led to a decrease in computational costs given they have reduced the input size of the GNN-based models. However, removing print statements seems to be more a efficient strategy in this context because it succeeded in fully retaining the performance and reducing the complexity of graphs at the same time.

In order 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 by providing different classifications. Listing 6 presents a snippet where such a disagreement occurred. For presentation purposes, the code has been simplified, whereas the full version can be found here[3].

Listing 6 An example showing noisy print statements affecting the predictive performance. It is a benign sample from *Complex conditional* smell ground truth. Models trained on R_1 representation detects the smell whereas models trained on R_3 classifies it as benign.

```
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);
    }
```

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

[3]http://bit.ly/446v0n7

representation. Analogous to stop words in the Natural Language Processing (NLP) domain, print statements, in this context, introduce noise and do not help the model to classify correctly. Moreover, their high frequency increases computational costs.

Concord dsl and framework automates the customized code representation. Of course, the automation and simplification strategies offered by dsl must be used according to the requirements of the downstream task. For example, it may be unwise to perform print statement removal if we would like to generate code representation for code search as the downstream task that retrieves snippets that print to the I/O. Continuing with the analogy we drew from NLP, the same can be applied to stop words. For instance, 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.

5 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 prior to the publication date of the replication package released by Sharma *et al.* [55]. This was done to minimize the risk of code smells being refactored. Moreover, we excluded code snippets that did not have simple assignment and print statements to avoid generating misleading results. Furthermore, during the sampling process for the construction of the data set, we ensured the preservation of the same positive-to-negative class ratio to avoid erroneous results as discovered by Chakraborty *et al.* [11]. Furthermore, to ensure fairness in evaluation, all models were trained and evaluated on the exact same splits.

External validity: This type of threat is related to the generalizability of the method. Although we provided a use case for code smell detection, Concord can be integrated into many other tasks, such as vulnerability detection or 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: Construct validity threats are related to measurements and randomness. First, we carefully constructed the R_1, R_2, and R_3 configurations to measure the effect of removing simple assignment and print statements across the code snippets in isolation. Moreover, we set the same random seed across all models during the training and inference phases. Regarding metrics, we used adequate measures to evaluate classifiers and calculate their computational costs as done in similar work [33, 52, 55, 68].

6 CONCLUSIONS, IMPLICATIONS, 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. We demonstrate how it can be used in a real example to detect code smells. Our experiments show that it can effectively produce graph representation of source code and reduce computational costs while maintaining up to 100% performance.

Implications for Research: Using this dsl, researchers no longer need to manually implement disparate edge augmentations and modelling configurations, saving time and effort. Moreover, by introducing a unified tooling support, researchers can adopt a standardized approach to code representation and analysis, enhancing the reproducibility of experiments and fostering transparency in research findings. Having a tool like Concord that unifies the fragmented functionality of graph-based code representations and allows composition can accelerate research progress in the field. Researchers can quickly experiment with different configurations, facilitating faster

iteration and hypothesis testing. The ease of use and automation provided by the tooling support can lead to more efficient exploration of new analysis techniques and advancing the state-of-the-art in software engineering research. We also hope that by open-sourcing our work, the DSL can be further expanded by the research community to accommodate new emerging ideas.

Implications for Industry: In the industry context, CONCORD addresses the challenge of scalability in training GNNs. The reduction heuristics can contribute to the creation of scalable solutions for bug detection, code refactoring, and security analysis in industries. In addition, the reduced engineering overhead of using CONCORD facilitates quicker adoption of graph-based source code analysis in real-world software development pipelines, potentially improving code quality and software maintenance practices.

In the future, we will investigate how we can apply CONCORD in other tasks in source code analysis, and extend it further to accommodate other modelling techniques that can emerge in the literature. We hope that this DSL helps the research community explore new frontiers by providing new tooling support.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (jan 2019), 29 pages. https://doi.org/10.1145/3290353

[4] Artur Andrzejak, Kevin Kiefer, Diego Elias Costa, and Oliver Wenz. 2019. Agile Construction of Data Science DSLs (Tool Demo). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Athens, Greece) *(GPCE 2019)*. Association for Computing Machinery, New York, NY, USA, 27–33. https://doi.org/10.1145/3357765.3359516

[5] Anonymous Authors. 2023. Replication package for Concord DSL and framework. https://github.com/concord-dsl/CONCORD

[6] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-level encoding for neural source code summarization of subroutines. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 253–264.

[7] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *IEEE International Conference on Software Maintenance, ICSM*. Universita di Salerno, Salerno, Italy, IEEE, 56–65.

[8] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior among Programmers *(ICSE '18)*.

[9] David Bieber, Kensen Shi, Petros Maniatis, Charles Sutton, Vincent J. Hellendoorn, Daniel D. Johnson, and Daniel Tarlow. 2022. A Library for Representing Python Programs as Graphs for Machine Learning. *ArXiv* abs/2208.07461 (2022).

[10] L. Buch and A. Andrzejak. 2019. 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 95–104. https://doi.org/10.1109/SANER.2019.8668039

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* (sep 2022).

[12] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[13] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. 2022. RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code. *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2022).

[14] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 423–433. https://doi.org/10.1145/3236024.3236059

[15] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. 2017. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems* 115 (2017), 1–4. https://doi.org/10.1016/j.knosys.2016.10.023

[16] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. 2017. Semantic Code Repair using Neuro-Symbolic Transformation Networks. *ArXiv* abs/1710.11054 (2017).

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong (YIMING), Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of EMNLP 2020*.

[18] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Not.* 39, 1 (jan 2004), 111–122. https://doi.org/10.1145/982962.964011

[19] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Programs* (1 ed.). Addison-Wesley Professional.

[20] Martin Fowler and Rebecca Parsons. 2022. *Domain-specific Languages*.

[21] Joan Giner-Miguelez, Abel Gómez, and Jordi Cabot. 2023. A domain-specific language for describing machine learning datasets. *Journal of Computer Languages* 76 (2023), 101209.

[22] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning *(CODASPY '16)*.

[23] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal Representation for Neural Code Search. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 483–494. https://doi.org/10.1109/ICSME52107.2021.00049

[24] Mouna Hadj-Kacem and Nadia Bouassida. 2018. A Hybrid Approach To Detect Code Smells using Deep Learning.. In *ENASE*. 137–146.

[25] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.

[26] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. 2014. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (Sept. 2014), 33–39.

[27] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved Automatic Summarization of Subroutines via Attention to File Context *(MSR '20)*. Association for Computing Machinery.

[28] Aurel Ikama, Vincent Du, Philippe Belias, Biruk Asmare Muse, Foutse Khomh, and Mohammad Hamdaqa. 2022. 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)*. 56–65. https://doi.org/10.1109/SCAM55253.2022.00012

[29] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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)*.

[30] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. 2013. Mining the relationship between anti-patterns dependencies and fault-proneness. In *Proceedings - Working Conference on Reverse Engineering, WCRE*. Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, 351–360.

[31] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 474, 12 pages.

[32] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (June 2012), 243–275.

[33] Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Woosuk Kwon, Joseph Hassoun, and Kurt Keutzer. 2022. Learned Token Pruning for Transformers *(KDD '22)*.

[34] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJU4ayYgl

[35] Rainer Koppler. 1997. A Systematic Approach to Fuzzy Parsing. *Softw. Pract. Exper.* 27, 6 (jun 1997), 637–649. https://doi.org/10.1002/(SICI)1097-024X(199706)27:6%3C637::AID-SPE99%3E3.0.CO;2-3

[36] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. *Improved Code Summarization via a Graph Neural Network*. Association for Computing Machinery, New York, NY, USA, 184–195.

[37] Rongfan Li, Bihuan Chen, Fengyi Zhang, Chao Sun, and Xin Peng. 2022. Detecting Runtime Exceptions by Deep Code Representation Learning with Attention-Based Graph Neural Networks. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[38] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. 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*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.05493

[39] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2022).

[40] Amanda Liu and Michael Coblenz. 2023. Debugging Techniques in Professional Programming. (3 2023).

[41] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* (2019).

[42] Shangqing Liu, Yu Chen, Xiaofei Xie, J. Siow, and Yang Liu. 2020. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *International Conference on Learning Representations*.

[43] Shangqing Liu, Xiaofei Xie, L. Ma, J. Siow, and Yang Liu. 2021. GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search. *IEEE Transactions on Software Engineering* (2021).

[44] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2023. Combining Graph Neural Networks With Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[45] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2021. Modeling functional similarity in source code with graph-based siamese networks. *IEEE Transactions on Software Engineering* (2021).

[46] Sergio Morales, Robert Clarisó, and Jordi Cabot. 2022. Towards a DSL for AI Engineering Process Modeling. In *Product-Focused Software Process Improvement*, Davide Taibi, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder, and Pekka Abrahamsson (Eds.). Springer International Publishing, Cham, 53–60.

[47] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for Engineered Software Projects. *Empirical Softw. Engg.* (2017).

[48] Himesh Nandani, Mootez Saad, and Tushar Sharma. 2023. DACOS—A Manually Annotated Dataset of Code Smells. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 446–450. https://doi.org/10.1109/MSR59073.2023.00067

[49] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API Elements for Code Migration with Vector Representations. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 756–758. https://doi.org/10.1145/2889160.2892661

[50] Christopher Olston, Benjamin C. Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*.

[51] Mikhail Perepletchikov and Caspar Ryan. 2011. A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *IEEE Transactions on Software Engineering* 37, 4 (Aug. 2011), 449–465.

[52] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding Neural Code Intelligence through Program Simplification *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3468264.3468539

[53] Andrew Rice, Henry Mercer, Karthik Chandra, Karthik Chandra, and Yijun Yu. 2018. *features-javac*. https://github.com/acr31/features-javac

[54] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[55] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. 2021. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software* 176 (2021), 110936. https://doi.org/10.1016/j.jss.2021.110936

[56] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. 2021. A Survey on Machine Learning Techniques for Source Code Analysis. https://doi.org/10.48550/ARXIV.2110.09610

[57] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158–173. https://doi.org/10.1016/j.jss.2017.12.034

[58] J. Siow, Shangqing Liu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2022. Learning Program Semantics with Code Representations: An Empirical Study. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2022), 554–565.

[59] Jürgen Thanhofer-Pilisch, Alexander Lang, Michael Vierhauser, and Rick Rabiser. 2017. A Systematic Mapping Study on DSL Evolution. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 149–156. https://doi.org/10.1109/SEAA.2017.25

[60] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[61] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2020. Multi-Modal Attention Network Learning for Semantic Source Code Retrieval *(ASE '19)*.

[62] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. 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)* (2020), 261–271.

[63] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. 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*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). 261–271.

[64] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/sp.2014.44

[65] Jingxiu Yao and Martin Shepperd. 2020. Assessing Software Defection Prediction Performance: Why Using the Matthews Correlation Coefficient Matters. In *Proceedings of the Evaluation and Assessment in Software Engineering* (Trondheim, Norway) *(EASE '20)*. Association for Computing Machinery, New York, NY, USA, 120–129. https://doi.org/10.1145/3383219.3383232

[66] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2023. DeGraphCS: Embedding Variable-Based Flow Graph for Neural Code Search. *ACM Trans. Softw. Eng. Methodol.* (mar 2023).

[67] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[68] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. 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).

[69] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. 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.

[70] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* (2021).