Mining Code Change Patterns in Ada Projects

Robin van Straeten
Radboud University
Nijmegen, The Netherlands

Bin Lin†
Hangzhou Dianzi University
Hangzhou, China

***Abstract*—The Ada programming language, while not deemed as a mainstream choice for daily software applications, still plays a vital role in security-critical systems. Understanding how the source code of Ada projects changes over time can provide valuable insights into the maintenance and evolution of these software systems, which can be further leveraged for various software engineering tasks including refactoring recommendation and bug fixing. In this study, we employ fine-grained program dependence graphs to mine code change patterns of Ada-based systems. By analyzing 50 open-source Ada projects, we identify the typical modifications developers perform in Ada codebases and the underlying development activities that led to these changes. In comparison to Java code change patterns, our results indicate more diverse and specialized change practices in Ada software development.**

***Index Terms*—Code Change Patterns, Ada Programming Language, Mining Software Repositories**

I. Introduction

As early as over 20 years ago, there was already a claim that "*Ada is perceived as a dying language by most software engineers*" [1]. Many years have passed ever since, and in fact, the Ada language is not really dead. In 2023, the new international standard was published for Ada 2022 [2]. A quick search on GitHub[3] shows that there are 300-600 new Ada projects every year since 2015, indicating that there is still an active Ada community, although not large compared to other programming languages such as Java and Python.

While Ada is not considered as a mainstream programming language for daily software applications, today, Ada code still widely exists in embedded real-time systems, many of which are security-critical, such as defense, civil aviation and rail [4]. For example, the Air Data Inertial Reference Unit of Airbus A350 XWB and International Space Station Communication Subsystem both contain Ada code [5]. Given the importance of these domains, understanding how Ada projects evolves offers several benefits: First, the information can provide valuable insights on how to better maintain Ada legacy software; Second, the relevant knowledge can facilitate the research and development of automated tools for code modification/migration. For example, previous studies [6] have leveraged code change patterns to improve IDE code completion suggestions, recommend bug fixes, and suggest refactoring operations.

While code change patterns have been widely studied for other popular programming languages like Java [6], [7], [8] and Python [9], [10], no insights have been provided for Ada, which is important for applications requiring high security, correctness and safety. This paper aims to fill this gap by conducting an empirical analysis of code changes in Ada projects, replicating the study of Nguyen *et al.* [8]. More specifically, we employ fine-grained Program Dependence Graphs (fgPDGs) to capture the relationships within programs and identify recurring code change patterns in Ada projects.

Our main goal is to understand how developers perform code changes in Ada projects and how the changes differ from those in Java projects. We pose the following RQs:

RQ_1: What are the common code change patterns in Ada projects? We identified 1,093 code change patterns across 50 Ada repositories. These patterns not only vary in their frequency of occurrence but also in their distribution across repositories. We also highlighted the most frequent change patterns within and across repositories.

RQ_2: What types of activities do the change patterns belong to? We manually classified all mined code change patterns into distinct development activities and observed that the activities associated with these patterns span various software maintenance types (adaptive (8.78%), perfective (32.30%), corrective (25.34%) and preventive (33.58%)), with some patterns being repeated over extended periods.

RQ_3: To what extent do individual developers or teams share the change patterns? We investigated the distribution of identified change patterns across the Ada developer community and found that the discovered change patterns were much less ubiquitous for Ada projects than those in Java projects. For example, 85% of Java projects had at least 10% common patterns, while this is only true for 56% of Ada projects.

RQ_4: What is the temporal distribution of the discovered change patterns? We examined whether developers tend to repeat their code changes over a long period. The results confirm this assumption and indicate the promises of creating a centralized code change pattern repository.

II. Related Work

Numerous studies have attempted to detect code changes from software repositories. One of the most fundamental steps of these techniques is code differencing, namely detecting the differences between two versions of source code. Based on code differencing, researchers can further understand how source code changes. The code change detection techniques can either have pre-defined patterns (one of the most prominent types is refactoring detection), or identify/extract patterns through mining the code changes across software repositories. This section presents a brief overview of these techniques.

† Corresponding Author.

A. Code Differencing

Canfora *et al.* [11] proposed LDIFF, an enhanced line differencing tool. LDIFF is based on the information produced by Unix diff. More specifically, LDIFF addresses the limitation of Unix diff being unable to distinguish modified lines from added and removed lines, and also provides the possibility of tracking line moving.

Fluri *et al.* [12] presented CHANGEDISTILLER, which extracts code changes based on abstract syntax trees (ASTs). CHANGEDISTILLER first matches the nodes of two ASTs and then calculates an edit script of basic tree edit operations which transform one AST to the other. The ASTs used are simplified, with code statements being used as leaf nodes.

JSYNC, a tool designed for code clone management and proposed by Nguyen *et al.* [13], also provides the functionality of tracking code changes. Likewise, JSYNC is also based on AST, in which each node represents class, method, statement, or expression. JSYNC adopts tree edit operations such as addition, deletion, modification, and update of tree nodes.

Falleri *et al.* [14] developed GUMTREE, aiming to capture more fine-grained code changes. For example, a return statement will be converted to a subtree with the node "ReturnStatement" as the parent and the returned value as the child. GUMTREE addressed two major challenges of previous work on AST-based edit script computation, namely the difficulty of handling move actions and the scalability issue of handling fine-grained ASTs with thousands of nodes. At the time of our study, GUMTREE supports 16 programming languages, and Ada is not one of them.

Other code differencing algorithms have used more complex representations of source code. For example, Raghavan *et al.* [15] developed DEX, in which each version of source code is represented by abstract semantic graphs (ASGs). An ASG is an AST with extra edges indicating type information. Xing and Stroulia [16] presented UMLDIFF, which automatically detecting structural changes based on the class models of a Java project reverse engineered from source code. The tool JDIFF, developed by Apiwattanapong *et al.* [17], specifically looks into the matching methods in an object-oriented language. The representations for two Java methods are compared to extract code differences.

B. Refactoring Detection

Refactoring, the action of improving internal code structure of a software system without changing its external behaviors [18], is one of the most popular code change types researchers have showed enormous interest in.

As early as 2006, Xing and Stroulia [19] developed JDEVAN (Java Design Evolution Analysis), which can detect refactoring operations by applying a set of predefined queries on the code changes produced by UMLDIFF [16].

REFDIFF by Silva *et al.* [20] is a multi-language refactoring detection tool supporting 13 common refactoring types for Java, C, and JavaScript projects. REFDIFF converts different versions source code to high level models representing code elements and TF-IDF is used to calculate code similarity.

REFACTORINGMINER developed by Tsantalis *et al.* [21] is by far the most comprehensive and reliable refactoring detection tool for Java projects. REFACTORINGMINER currently supports around 100 refactoring types. Instead of using similarity thresholds, the tool detects refactoring operations using pre-defined heuristic rules. Based on the core algorithm of REFACTORINGMINER, Atwi *et al.* [22] developed PYREF to detect refactoring operations in Python projects. PYREF mainly focuses on method-level refactoring operations and currently supports nine different types of refactoring.

C. Code Change Pattern Mining

Several studies have focused on detecting repetitive code change patterns from software revision history. Negara *et al.* [23] is among the first to identify unknown frequent code change patterns from a fine-grained sequence of code changes produced by AST-based code differencing. Their approach can identify the set of instances of the same kind of code changes.

Janke and Mäder [24] presented a novel method for automatically extracting code change patterns from Git repositories. Their approach transforms AST-based edit scripts into a graph dataset and applies graph mining techniques to extract frequent subgraphs. They found that those mined change patterns are largely persistent across projects.

Nguyen *et al.* [8] presented a graph-based mining tool CPATMINER to mine fine-grained semantic code change patterns from a large number of Java repositories. Their approach aims to address the issue of previous studies that semantic closeness of the changes is not always well captured. Especially, they observed that syntactically unchanged program statements could also carry important semantic change elements and the atomic changes might not be on the contiguous lines of source code. They proposed a representation of code named fine-grained program dependence graph (fgPDG) which represents the data/control dependencies among fine-grained program elements at the expression granularity. Based on fgPDG, CPATMINER is able to extract meaningful semantic code change patterns, as confirmed by developers.

Golubev *et al.* [10] presented PYTHONCHANGEMINER, which implements the core algorithm of CPatMiner to discover semantic change patterns in Python. They used PythonChangeMiner to analyze 120 projects from four different software engineering domains and the patterns were manually categorized based on their structure and content. While PYTHONCHANGEMINER is based on CPATMINER, there are some critical differences in their implementation. First, PYTHONCHANGEMINER utilizes GumTree [14], while CPATMINER employs JSync's algorithm [13] to identify the corresponding and changed nodes to construct change graphs from fgPDGs.

Based on CPATMINER, Dilhara *et al.* [9] developed R-CPATMINER to detect repetitive code changes in Python machine learning systems. R-CPATMINER also integrates RefactoringMiner to make the process refactoring-aware.

While relevant, these studies mainly focus on more popular programming languages, with little attention paid to Ada.

III. ADA-CPATMINER

In this section, we present ADA-CPATMINER, the tool for mining code change patterns from Ada projects. The main algorithm of ADA-CPATMINER tightly follows CPATMINER [8], which is also the foundation for PYTHONCHANGEMINER [10]. In this section, we detail how ADA-CPATMINER works and explain the necessary deviations from the original CPATMINER approach.

A. Overview

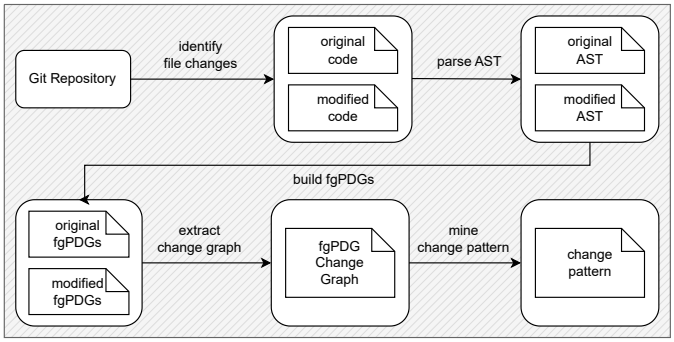

Fig. 1: Overview of ADA-CPATMINER

ADA-CPATMINER follows the following steps to extract code change patterns from Ada projects. Given a set of Ada software repositories, ADA-CPATMINER first iterates the commits of each repository to identify changed Ada files. The original files and the changed files are then parsed into ASTs and method pairs with identical names are extracted. For these method pairs, ADA-CPATMINER generates fine-grained Program Dependence Graphs (fgPDGs). The fgPDGs of method pairs are further compared to extract the change graphs. Finally, ADA-CPATMINER mines code changes patterns in all of the obtained change graphs.

B. File Change Identification

For each Ada repository in our dataset, we first iterate the commits and identify the changed Ada files. To do so, we use the PYDRILLER framework [25]. We only focus on the modified files with the "`.adb`" extension, which is typically used for the Ada implementation files (or Ada body). These adb files are akin to the cpp files of the C++ programming language.

C. AST Parsing

The ADA-CPATMINER parses ASTs using *Libadalang* [1]. *Libadalang* is a library for parsing and conducting semantic analysis of Ada code, which allows ADA-CPATMINER to perform semantic queries on ASTs, such as reference resolution (what a reference corresponds to) and type resolution (which type an expression belongs to). These queries can be helpful in this case, because it can be hard to determine whether an expression is a variable reference or a function call based on the AST alone.

[1]*Libadalang*: https://github.com/AdaCore/libadalang/

D. Fine-Grained Program Dependence Graphs Building

Program Dependence Graphs (PDGs) are a representation of a program that captures both data and control dependencies [26]. These dependencies are crucial for various software engineering tasks, including program slicing, optimization, and, as in our context, code change pattern mining. Traditional PDGs can be coarse, and often represent dependencies at the level of statements or blocks. On the other hand, another widely used program representation Abstract Syntax Tree (AST) only focuses on the syntactic structure of the program, which, while valuable, can sometimes overlook the semantic nuances of code changes. To bridge the gap between the high-level view of PDGs and the syntactic granularity of AST, Nguyen *et al.* [8] proposed fine-grained Program Dependence Graphs (fgPDGs), which not only captures data and control dependencies, but deals with source code at a more fine-grained level of expressions [8].

1) Node Types in fgPDGs: The architecture of fgPDGs is structured based on three primary node types:

1) **Data nodes** encapsulate program elements like variables, literals, and constants, highlighting the data components of a program.
2) **Operation nodes** symbolize the active elements of a program, these nodes represent operations ranging from arithmetic to function calls.
3) **Control nodes** illustrate the program's control flow, representing sequences such as 'if', 'while', and 'for'.

2) Edge Types in fgPDGs: In fgPDGs, the relationships between nodes are articulated through edges, which can be categorized into two types:

1) **Control edges** serve as the bridge between control nodes and their controlled counterparts, *i.e.,* these edges define the control flow.
2) **Data edges** illustrate the program's data flow.

E. Change Graph Extraction

A *change graph* connects the fgPDGs of the code fragments before and after changes. The connection is established through so-called map edges, which link nodes from the fgPDG of the pre-change version to those in the fgPDG of the post-change version if they represent corresponding unchanged elements. Nodes deleted in the older version or introduced in the new version are also annotated, providing a clear view of the changed elements.

To derive the fine-grained changes within each modified method, an initial step involves computing tree-based differences using an AST differencing algorithm. This algorithm maps nodes between two ASTs, based on node types and structural similarities between subtrees rooted at those nodes. Nodes that remain unmapped are deemed deleted from the old tree or added to the new tree. Leveraging this information, corresponding nodes in the two fgPDGs–before and after the changes–are linked.

1) AST Differencing Algorithm: We initially re-implemented JSYNC for Ada, which was the AST differencing algorithm used in CPATMINER [8]. However, during our experimentation, we found the performance was not optimal. For example, "move" operations were often misclassified as "delete and add" operations because it was unable to find the correct mappings. Therefore, we decided to opt for GUMTREE [14]. *GumTree* was also used in the work of Golubev *et al.* [10] to mine Python code change patterns. In their case, they were able to use GUMTREE directly due to its Python support. Such support does not exist for Ada, therefore, we ported GUMTREE for the Ada language.

2) An Example of Change Graph Extraction: To have a better understanding on how change graph extraction works, we present a code change from the SDLAda repository[2] (Listing 1) and its corresponding change graph (Figure 2). In the example, the if statement is inverted and an early return is added to the "not" condition. In the change graph, the left dotted rectangle contains the fgPDG of the code snippet before change, while the right one contains the fgPDG after code change. All the unchanged nodes are connected through a '*map*' edge.

```
if Initialized (W) then
   W.Resource := Invalid_Resource;
end if;
```
(a) Before changes.

```
if not Initialized (W) then
   return;
end if;
W.Resource := Invalid_Resource;
```
(b) After changes.

Listing 1: A code change from SDLAda repository: the inverted if statement and early return.

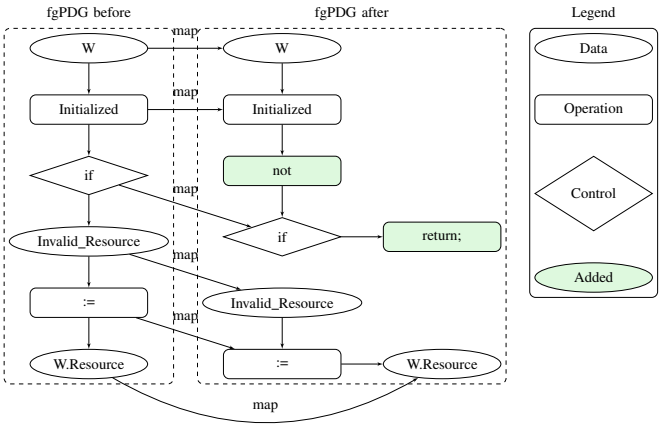

Fig. 2: fgPDG Change Graph

[2]https://github.com/Lucretia/sdlada

F. Change Pattern Mining

Change patterns provide a systematic way to comprehend recurring code alterations. These patterns are not just repetitive syntactic modifications but are semantic in nature, indicating a deeper behavioral or structural change in the software.

1) Semantic Change Patterns: A change is identified as a semantic change pattern if its corresponding fgPDG is a connected graph and is repeated (found isomorphic) at least σ times in the change graph dataset, with σ being user-configurable [8]. The notion of isomorphism implies that the graphs have a one-to-one correspondence between their nodes and edges, preserving the relational structure.

2) Change Pattern Mining Algorithm: While a more thorough description of the change pattern mining algorithm is available in the original paper of CPATMINER [8], here we briefly introduce the main idea of this algorithm. The algorithm commences by identifying nodes that appear frequently as foundational building blocks, and subsequently, these nodes are recursively extended to craft more expansive patterns. The algorithm adheres to the following strategy:

1) **Pattern Initialization:** Starting with frequently occurring nodes, a set of size-of-1 patterns is constructed.
2) **Pattern Expansion:** Each size-of-1 pattern is incrementally expanded to establish larger patterns. To do so, the algorithm
   - identifies adjacent nodes for extension, placing significance on their type and their relationship with existing sub-graph,
   - differentiates node types, ensuring certain criteria are met for expansion. For instance, method-call nodes are always eligible for expansion, while other operation nodes require both incoming and outgoing connections to the sub-graph, and
   - adds nodes and their corresponding map edges (if existent) to the sub-graph to encapsulate the change data.
3) **Isomorphic Clustering:** With potential expansions in place, these patterns are clustered based on isomorphism, grouping structurally identical graphs. To enhance efficiency, a heuristic (Exas) combining graph vectorization and hashing is employed [27].
4) **Pattern Selection:** Among the clusters, those not meeting the minimum frequency σ or already discovered are discarded. From the remaining clusters, the most frequent one is chosen, and its extensions are sought recursively. This greedy approach obviates exhaustive search complexities, ensuring scalability.

G. Validation

We manually evaluated a collection of patterns to validate the performance of ADA-CPATMINER. In this validation, we followed the approach adopted by Golubev *et al.* [10], in which we consider a pattern correct if it actually represents a repeated change and incorrect if it does not.

To obtain these patterns, we used the GraphQL API [28] to collect projects from GitHub which fulfill the following conditions: 1) the main language is Ada, 2) the project has between 500 and 1,000 commits, and 3) the project is not a fork of another project. We selected the top 10 repositories based on the stargazer count.

TABLE I: The summary of projects used to validate the *Ada Change Pattern Miner*. The dataset was collected in June 2023.

Project	Stars	Commits	Contributors
AdaDoom3/AdaDoom3	246	831	7
jrmarino/synth	243	734	12
tofgarion/spark-by-example	146	778	5
Lucretia/sdlada	99	520	11
zertovitch/hac	86	833	1
wookey-project/ewok-kernel	71	520	5
Blunk-electronic/M-1	31	647	2
jrmarino/AdaBase	30	586	2
Componolit/libsparkcrypto	27	801	3
Componolit/gneiss	22	643	3
yet-another-static-site-generator/yass	22	781	5

Since ADA-CPATMINER identifies changes of any size, we needed to filter out changes too small to be meaningful patterns. We decided to use the same thresholds as Nguyen *et al.* [8] for CPATMINER. That is, the minimum number of nodes in a change graph is three, and the minimum frequency of repeated graphs to form a pattern is also three (following the "Rule of Three" [18]). In total, 1,836 change graphs were extracted and processed, resulting in 66 patterns containing 228 samples. We manually reviewed all patterns for verification. We determined that 58 (87.9%) of the inspected patterns were straightforward and actually represented change patterns. Eight (12.1%) patterns were deemed incorrect. In one pattern, the *GumTree* algorithm incorrectly mapped two methods with similar names, which led to an incorrect pattern. In another pattern, the changes had similar AST structures but were unrelated. For the remaining six patterns, the same object had moved and had nothing in common except for this object. Given these results, we believe our tool is overall reliable.

IV. STUDY DESIGN

A. Research Questions

Our primary objective is to understand how developers perform code changes within Ada programs and identify recurrent change patterns that can shed light on everyday programming tasks and highlight potential opportunities for automation. To reach this goal, we replicate the study of CPATMINER [8] and formulate the following research questions. The same nature of questions asked not only helps us to look into Ada projects from a structured perspective, but also makes results comparison possible.

RQ1: What are the common code change patterns in Ada projects? This question seeks to uncover the prevalent change patterns in Ada programs by leveraging the capabilities of fine-grained program dependence graphs. Identifying these patterns helps us to understand the typical modifications developers perform in Ada codebases and offers insights into the common challenges or requirements Ada developers face, potentially leading to the development of tools or practices that can assist in these recurrent scenarios.

RQ2: What types of activities do the change patterns belong to? This question aims to categorize the patterns based on the development activities prompting these changes. By examining activities associated with these patterns, we can also understand whether the change patterns are diverse or if they tend to cluster around specific types of activities.

RQ3: To what extent do individual developers or teams share the change patterns? This question seeks to gauge the generality and ubiquity of the identified change patterns across the Ada programming community. By determining how widespread these patterns are among various developers, we can understand whether these patterns are unique to specific developers or teams or if they are common practices shared across the community. The broader the adoption of these patterns, the more indicative it is of their utility for a wide population of developers.

RQ4: What is the temporal distribution of the discovered change patterns? This research question aims to understand whether most instances of change patterns only occur in a short time or recur over long durations. The latter can underscore the value of creating a repository of such patterns, which would enable developers to tap into prior knowledge about code modifications and serve as a base for change automation.

B. Data Collection

The foundation of this research is built upon a robust dataset that accurately represents the Ada development community's practices concerning change patterns. In this study, two distinct corpora were utilized: a "*validation corpus*" for examining the reliability of ADA-CPATMINER (Section III-G) and a "*depth corpus*" specifically tailored to answer our research questions. In this subsection, we will introduce how we collected these repositories for the "*depth corpus*".

1) Repository Selection: We collected open-source software (OSS) repositories from GitHub, given its dominance in OSS development. We aimed to focus on high-quality, team-driven Ada projects that reflect collaborative development practices and, by extension, a broader range of change patterns. Therefore, we applied the following criteria to filter projects:

- Projects should have at least 1,000 commits, ensuring a rich history of development and changes.
- The projects should not be forks of other repositories. This criterion is essential to avoid the potential duplicates in change graphs.
- We select the top 50 Ada repositories based on the stargazer count, to ensure these projects are popular and have sufficient community engagement.

Given our project selection criteria, we first used the GraphQL API of GitHub to identify all repositories where Ada was the primary programming language. This initial sweep yielded a total of 5,932 Ada projects. By removing repositories with less than 1,000 commits, only 82 repositories were left.

Given the possibility of misclassifications inherent in automated data collection, we manually inspected these 82 repositories and removed repositories that 1) were incorrectly classified as Ada repositories, 2) were forks of other projects, or had inflated commit counts due to automated commits. Then we picked the top 50 projects based on their stargazer count from the remaining repositories. The list of adopted projects in this study can be found in our replication package.

Figure 3 illustrates in which year these repositories were created. The creation dates of these repositories span from 2010 to 2021. Notably, a majority of these projects (41 out of 50) were updated in the current year, indicating that these repositories are still actively being maintained. Only a handful of repositories had their last update in previous years, with 7 in 2022, 1 in 2019, and 1 in 2018. It is also worthwhile to mention that AdaCore maintains a significant portion of the repositories (20 out of 50). AdaCore is the leading provider of commercial software solutions for Ada.

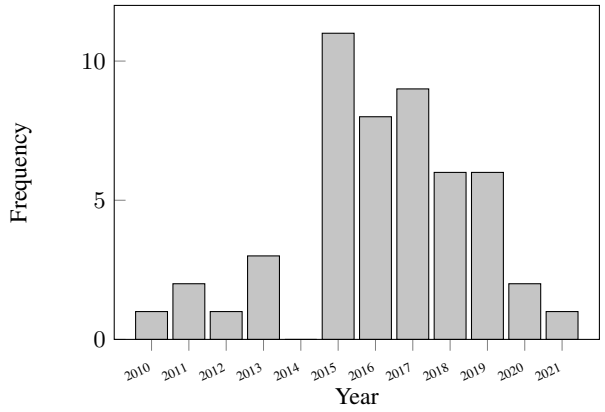

Fig. 3: Repository creation year counts.

2) Code Change Pattern Extraction: Cumulatively, the selected 50 projects encompass 52K Ada source files, amounting to 6.8M SLOC in their latest snapshot. Given the diverse range of projects and their varying commit counts, it was essential to normalize the data for a balanced analysis. Therefore, we extracted the most recent 2,000 commits involving Ada code changes from each selected project. We applied ADA-CPATMINER on these commits.

Change Pattern Size Constraints. Given the capability of ADA-CPATMINER to detect changes of any magnitude, we need to introduce constraints to filter out minuscule changes that might not represent meaningful patterns. To this end, we adopted the thresholds proposed by Nguyen *et al.* [8] in their work on CPATMINER:

- *Minimum Node Count*: The smallest permissible change graph was set to comprise at least 3 nodes. This threshold ensures that the identified changes are sufficient to be considered patterns.
- *Maximum Node Count*: To ensure computational efficiency and curtail the change pattern mining algorithm's running time, we discard those change graphs which has more than 100 nodes.

Pattern Frequency Threshold. We set the minimum frequency threshold of repeated graphs σ to be 3. This decision follows *et al.* [8]. It aligns with the well-established "Rule of Three", which posits that a pattern or behavior observed three times can be considered a consistent trend [18].

In total, we collected 86K change graphs. However, only 81K were deemed suitable for change pattern mining given the change pattern size constraints. The statistical information of these projects can be found in Table II, which also includes the information of the validation corpus for comparison purposes.

TABLE II: Collected Repositories. The dataset was collected in August 2023

	Val.-Corpus	Depth-Corpus
# projects	10	50
# total source files	1.5K	52K
# total SLOCs at last snapshot	186K	6.8M
# developers with analyzed commits	31	196
# analyzed commits	1.8K	20K
# total changed methods (graphs)	5.8K	81K
# total AST nodes of changed methods	1.7M	33M
# Total changed graph nodes	90K	1.4M

3) Change Activity Identification: We followed Nguyen *et al.* [8] and manually classified mined change patterns into four types of development activities based on ISO/IEC 14764:2006 [29]: adaptive, perfective, corrective, and preventive.

- *Adaptive* changes refer to those which modify software to keep them usable in a changed/changing environment.
- *Perfective* changes refer to those which provide enhancements for users such as performance optimization.
- *Corrective* changes refer to those which correct discovered problems such as bugs.
- *Preventive* changes refer to those which improve software products before they become faults, such as refactoring.

4) Pattern Share Identification: To understand whether the discovered code change patterns represent widespread practices or are unique to specific developers or teams, we calculate the percentage of patterns shared with others for unique developers and projects.

5) Pattern Temporal Distribution Extraction: We first isolated patterns of individual developers and created a timeline for each pattern based on commit timestamps from GitHub. We then calculated the time intervals between successive commits. To maintain our focus on the temporal distribution, instances from the same commit were excluded.

6) Running Time: The depth-corpus experiment was conducted on a Windows 10 Server, equipped with 8th Gen Intel(R) Core(TM) i7-8700K Hexa-Core 3.7 GHz CPU and 64GB of RAM. Change graph extractions were run with 12 threads, while change pattern mining was run on only one thread. In total, the execution of CPATMINER on the depth-corpus took 29 hours and 45 minutes. However, it is noteworthy that 5 out of 50 analyzed repositories were responsible for over 71% of the total running time.

V. Results

This section answers the four research questions regarding common code change patterns, activities associated with these patterns, pattern ubiquity among developers and teams, and the temporal distribution of patterns.

A. RQ1: What are the common code change patterns in Ada projects?

In this research question, we aim to shed light on typical modifications developers frequently perform in Ada projects. While all the extracted patterns can be found in our replication package, here we will present the most frequent patterns based on occurrence count and the most frequent patterns among repositories. We also present the temporal distribution of patterns.

1) Most Frequent Patterns Based on Occurrences: When only looking into the number of occurrences of code change patterns, we found that the most frequent patterns are typically refactoring operations applied to a wide range of methods. Interestingly, these patterns are also often confined to a single repository and are predominantly applied in a single commit. Here we illustrate the top-3 frequent patterns extracted with Ada-CPatMiner.

The most frequent pattern originated from the RecordFlux repository[3], which was identified 155 times. An example can be found in Listing 2. This pattern pinpoints a specific commit where the "Extract Subprogram" refactoring technique was employed across 155 analogous subprograms. In Ada, subprogram is akin to functions or methods in C++ and Java. The subprograms which return a value are referred to as "functions", while those which do not return any value are called "procedures".

```
procedure Set_Value (Ctx : in out Context;
                     Val : RFLX.Universal.Value) is
   Value : constant RFLX_Types.U64 := To_U64 (Val);
   Buffer_First, Buffer_Last : RFLX_Types.Index;
   Offset : RFLX_Types.Offset;
begin
   Set (Ctx, F_Value, Value, 8, True,
        Buffer_First, Buffer_Last, Offset);
   RFLX_Types.Insert (Value, Ctx.Buffer, Buffer_First,
                      Buffer_Last, Offset, 8,
                      RFLX_Types.High_Order_First);
end Set_Value;
```

(a) Before changes.

```
procedure Set_Value (Ctx : in out Context;
                     Val : RFLX.Universal.Value) is
begin
   Set_Scalar (Ctx, F_Value, To_U64 (Val));
end Set_Value;
```

(b) After changes.

Listing 2: An example of the most frequent change pattern. A new procedure `Set_Scala` was extracted from the original procedure.

The second most frequent pattern, also from the RecordFlux repository, occurred 55 times. This pattern highlights a commit where a function was removed as part of if conditions involving a Context variable. An example can be found in Listing 3.

```
procedure Verify (Ctx : in out Context; Fld : Field)
is
begin
   if
      Has_Buffer (Ctx)
      and then Invalid (Ctx.Cursors (Fld))
      and then Valid_Predecessor (Ctx, Fld)
      and then Path_Condition (Ctx, Fld)
   then
```

(a) Before changes.

```
procedure Verify (Ctx : in out Context; Fld : Field)
is
begin
   if
      Invalid (Ctx.Cursors (Fld))
      and then Valid_Predecessor (Ctx, Fld)
      and then Path_Condition (Ctx, Fld)
   then
```

(b) After changes.

Listing 3: An example of the second most frequent change pattern. The "`Has_Buffer`" function with the Context variable `ctx` was removed.

The third pattern was found in the ocarina repository[4]. This change shows another instance of the "Extract Subprogram" refactoring, applied to 35 subprograms. Listing 4 shows an example.

```
procedure Visit_Process_Instance (E : Node_Id) is
     begin
if not AINU.Is_Empty (Subcomponents (E)) then
   S := First_Node (Subcomponents (E));
   while Present (S) loop
      Visit (Corresponding_Instance (S));
      S := Next_Node (S);
   end loop;
end if;
```

(a) Before changes.

```
procedure Visit_Process_Instance (E : Node_Id) is
begin
   Visit_Subcomponents_Of (E);
```

(b) After changes.

Listing 4: An example of the third most frequent change pattern. A new procedure `Visit_Process_Instance` was extracted from the original procedure.

2) Patterns Most Frequent Among Repositories: Given that most patterns reside in a single repository, we also specifically looked into the patterns which recur across multiple repositories. Here we also list the top-3 patterns.

The leading pattern was identified across four repositories with 12 occurrences. In this pattern, an extra condition is added to an existing if condition. An example can be found in Listing 5.

[3]https://github.com/AdaCore/RecordFlux

[4]https://github.com/OpenAADL/ocarina

The second most frequent pattern across projects was observed in 4 different repositories, occurring five times. This pattern underscores a common practice when converting an unbounded string to a bounded one: adding an if statement. Listing 6 illustrates an example.

The third most frequent pattern, identified in 3 different repositories and occurring four times, indicates that when invoking the Close procedure, an if statement is frequently added, as illustrated in Listing 7.

```
if Kind (F) = K_Port_Spec_Instance then
```

(a) Before changes.

```
if Kind (F) = K_Port_Spec_Instance and then
   Is_Data (F) then
```

(b) After changes.

Listing 5: An example of the most frequent change pattern across repositories. The "`Kind`" function is part and a new if condition is added.

```
Ada.Text_IO.Put_Line
  (File, "# "
    & Ada.Strings.Unbounded.To_String (Document.Title));
Ada.Text_IO.New_Line (File);
```

(a) Before changes.

```
if Document.Print_Footer then
  Ada.Text_IO.Put_Line
    (File, "# "
      & Ada.Strings.Unbounded.To_String (Document.Title));
  Ada.Text_IO.New_Line (File);
end if;
```

(b) After changes.

Listing 6: An example of the second most frequent change pattern across repositories. An 'if' statement was added when converting an Unbounded_String to a String type.

```
overriding procedure Finalize (Object : in out Library) is
begin
    Close (Object.Handle, Raise_On_Error => False);
end Finalize;
```

(a) Before changes.

```
overriding procedure Finalize (Object : in out Library) is
begin
    if Object.Handle /= null then
        Close (Object.Handle, Raise_On_Error => False);
    end if;
end Finalize;
```

(b) After changes.

Listing 7: An example of the third most frequent change pattern across repositories. An 'if' statement was added when calling the `Close` procedure.

Lesson 1: Most patterns reside in a single repository or a single commit, and the patterns across projects have much fewer occurrences. This fact reminds the researchers to be especially careful when applying learned changed patterns to other projects, as they might not be applicable in other projects.

B. RQ2: What types of activities do the change patterns belong to?

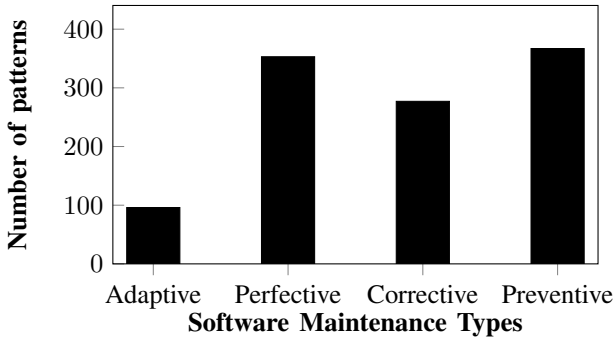

Fig. 4: The distribution of patterns by their Software Maintenance Type.

We manually classified all mined change patterns into distinct development activities: adaptive (8.78%), perfective (32.30%), corrective (25.34%) and preventive (33.58%). Notably, within the preventive changes, 12% were identified as refactoring.

For comparison, in the original study of Java change pattern mining, Nguyen *et al.* [8] found that 9% were adaptive, 20% were perfective, 35% were corrective, and 36% were preventive. Intriguingly, a significant 35% of their preventive changes were identified as refactoring.

These findings highlight profound differences in development activities between Ada and Java projects, especially within the perfective and corrective software maintenance categories. Additionally, the disparity in refactoring within preventive changes between the two languages is evident. We posit that these variations stem from Ada's foundational design philosophy centered on safety and reliability. Ada was designed with an emphasis on software engineering principles, including strong type-checking, array bound checking, initialization of variables, all geared towards early error detection during development rather than post-deployment.

Lesson 2: Corrective activities account for much smaller portion of code changes in Ada projects compared to Java. Such activity classification might be applied elsewhere to help developers understand the stability and reliability of their projects.

C. RQ3: To what extent do individual developers or teams share the change patterns?

By examining the extent to which the discovered patterns are shared or diverge among developers, we aim to uncover whether they represent widespread practices or are unique to specific developers or teams. A shared collection of patterns across the community can signify universally faced challenges or universally adopted best practices, while unique patterns might indicate team-specific challenges or solutions.

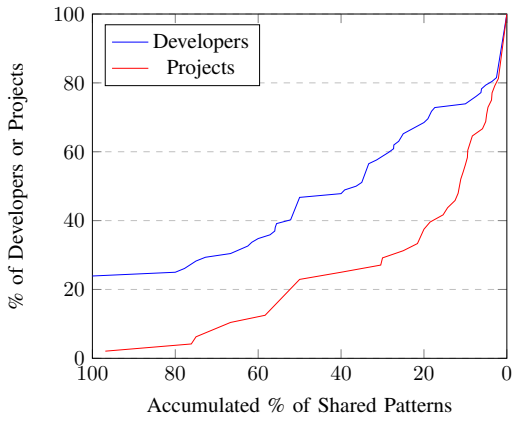

Fig. 5: Accumulated % of shared patterns over developers and projects.

1) Pattern Ubiquity Among Developers: The ubiquity of change patterns among developers offers a lens into the collective practices and challenges faced within the Ada developer community. As illustrated in Figure 5, a significant portion of developers, 23%, share all their change patterns with other developers; this suggests a commonality in challenges or solutions they encounter or adopt. However, the shared patterns decrease as we move toward a broader spectrum. Only 36% of developers have at least half of their patterns shared by their peers. Furthermore, a majority, 74%, have a minimal overlap, with just 10% of their patterns being common. This data indicates a blend of shared practices and unique challenges or solutions individual developers might face.

2) Pattern Ubiquity Among Projects: Diving deeper into the projects, we explore the extent to which change patterns are shared across different Ada projects. Figure 5 visually represents the accumulated percentage of shared change patterns among these projects. A striking observation is that only a single project, representing 2% of the total, has a vast majority (97%) of its patterns shared with other projects. However, only 23% of projects have at least half their patterns in common with others. A more significant percentage, 56%, have a modest overlap, sharing at least 10% of their patterns with other projects. This data underscores the diversity of practices, challenges, and solutions across different Ada projects, with some patterns being universally adopted while others remain project-specific.

3) Comparison with Java Code Change Patterns: The exploration of pattern ubiquity in Ada projects presents a contrasting landscape when compared with the findings of Nguyen *et al.* [8] for Java projects.

From the perspective of developers, 43% of individuals shared all their change patterns with others, while only 23% of Ada developers exhibited the same level of pattern sharing. Furthermore, 75% of Java developers had at least half of their patterns mirrored by their peers, compared to 36% of Ada developers. A broader view reveals that 88% of Java developers shared at least 10% common patterns, whereas this figure stands at 74% for Ada developers.

From the perspective of projects, the disparity is even more pronounced. A significant 31% of Java projects had all their patterns in common with other projects, while this was true for only 2% of Ada projects. Moreover, 55% of Java projects shared at least half of their patterns with others, in contrast to 23% of Ada projects. Furthermore, when considering projects with at least 10% common patterns, Java stands at 85% compared to Ada's 56%.

One plausible explanation for these differences could be Ada's relatively fewer available open-source libraries. With fewer libraries to rely on, Ada developers might often resort to crafting custom solutions, leading to a more diverse set of change patterns. Additionally, the emphasis on backward compatibility in common Ada libraries could mean that once a solution is implemented, there would be less need for drastic changes, leading to fewer shared patterns over time. Java, with its vast ecosystem of open-source libraries, might encourage developers to converge on common solutions and patterns, leading to a higher degree of shared practices.

These findings emphasize the importance of understanding the nuances of each programming community and the factors that influence their shared and unique development practices. It also underscores the role that available resources, like libraries, play in shaping the development landscape of a language.

> **Lesson 3:** While many Ada projects share code change patterns, overall these patterns are much more unique compared to those in Java projects. This fact might to a certain extent limit the ability of mining-based code change automation techniques for Ada projects, while still possible.

D. RQ4: What is the temporal distribution of the discovered change patterns?

We plot the durations of two consecutive instances of same change patterns applied by same developers in Figure 6. The figure reveals that a majority (54%) of the pattern instances recur after at least a month, with a significant 28% resurfacing after more than a year. This distribution suggests that change patterns in Ada development persist over time, namely a developer would repeat similar code changes for a while. This result also reinforces the idea that developers might benefit from a centralized repository of change patterns, such that you could easily reuse.

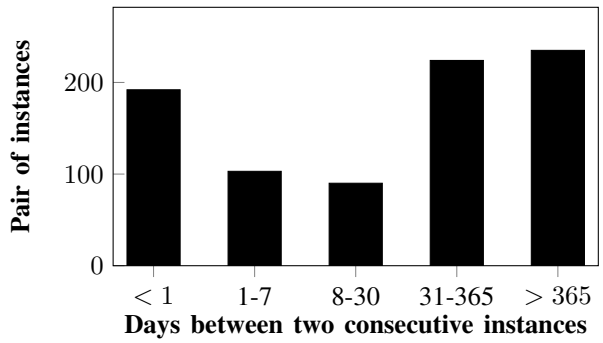

Fig. 6: Histogram of Change Patterns over Time

Compared to the temporal distribution of Java code change patterns [8], similar trend was found (Slightly less than half of the Java change pattern instances are repeated by the same developer after at least one month). While a higher percent of Ada change patterns recur over one year, the differences are not that huge.

Lesson 4: Developers do repeat their code changes over a long period, and the trend holds for both Ada and Java projects. A repository of change patterns might help developers easily reapply code changes.

E. Replication

To facilitate the replication and extension, the source code of ADA-CPATMINER and the detected code change patterns are available at https://github.com/codechangepattern/Ada-CPatMiner.git.

VI. THREATS TO VALIDITY

Threats to internal validity concern the extent in which the evidence supports our claim about cause and effect. While our repository selection criteria were designed to ensure a comprehensive and representative dataset, they inherently introduce a potential bias. By focusing on repositories with a high stargazer count and a minimum number of commits, we inadvertently excluded newer or smaller projects that could offer unique insights into Ada development practices.

Threats to construct validity concern the relationship between theory and observation. The ADA-CPATMINER, though tailored for Ada, is based on principles derived from tools like CPATMINER [8] and PYTHONCHANGEMINER [10]. While these program languages have very different syntax, we believe this would not lead to significant imprecision of change pattern extraction, which was demonstrated through our manual validation. However, there might still be specific Ada-centric nuances or constructs that the tool might not capture optimally.

While necessary for computational efficiency, the constraints imposed on the size of change graphs might exclude certain intricate or more extensive change patterns that could be of significance.

Threats to external validity concern the generalizability of our results. While our findings provide valuable insights into code change patterns in Ada projects, caution should be exercised when attempting to generalize these findings to other programming languages or development ecosystems.

Besides, our study is grounded in the analysis of open-source Ada projects on GitHub. This focus might not capture the practices and patterns prevalent in proprietary or enterprise Ada projects, which could differ significantly from open-source development.

Moreover, our dataset was collected in August 2023, which offers a snapshot of Ada development practices up to that point. As the Ada community evolves and adopts new practices or tools, some of our findings might become less relevant or require re-evaluation in the future.

Threats to conclusion validity concern the degree to which the conclusion presented is reliable. One of the potential threats lies in the quality and relevance of the change patterns we extracted. While our methodology was rigorous in identifying and categorizing these patterns, the absence of qualitative assessment means that we cannot definitively comment on the meaningfulness or utility of these patterns in real-world Ada development scenarios. In change pattern mining, the quality of patterns is not solely determined by their frequency or distribution but also by their relevance and applicability to developers.

VII. CONCLUSIONS

During this research, we embarked on a comprehensive exploration of change pattern mining in Ada projects by replicating the study of Nguyen *et al.* [8] with necessary customization for the Ada programming language. We developed ADA-CPATMINER to analyze a wide selection of Ada projects. Our analysis pinpoints a series of recurring change patterns within Ada projects, discloses how universally these patterns are adopted among individual developers and across diverse projects, and indicates the promises of building a centralized change pattern repository for easy change reapplication.

Given the insights and the challenges encountered, several avenues for future exploration and enhancement emerge:

- **Refinement of the ADA-CPATMINER**: The current version of the ADA-CPATMINER, though effective, has room for enhancement. Future work could focus on optimizing its algorithms, expanding upon its Ada AST support, and refining its accuracy.
- **Dataset Expansion**: Our research explored open-source Ada projects. A more comprehensive view might emerge if future research could incorporate proprietary Ada projects.
- **Qualitative Research on Pattern Usefulness**: We would like to assess the meaningfulness and usefulness of the mined patterns. Engaging with Ada developers and stakeholders can provide a deeper understanding of the real-world applications and applicability of the identified patterns.

References

[1] I. Gilchrist, "Attitudes to ada—a market survey," in *Proceedings of the 1999 annual ACM SIGAda international conference on Ada*, 1999, pp. 229–242.

[2] I. IEC, "Iso/iec 8652:2023 information technology — programming languages — ada," *International Standards Organization, Geneva, Switzerland*, 2023.

[3] "GitHub." [Online]. Available: https://github.com/

[4] R. Amiard and G. A. Hoffmann, *Introduction to Ada*. AdaCore, 2023.

[5] AdaCore, "Ada: Meeting tomorrow's software challenges today," AdaCore, Tech. Rep., 02 2019.

[6] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[7] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, Aug. 2009.

[8] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 819–830.

[9] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering repetitive code changes in python ml systems," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 736–748.

[10] Y. Golubev, J. Li, V. Bushev, T. Bryksin, and I. Ahmed, "Changes from the Trenches: Should We Automate Them?" Aug. 2021, arXiv:2105.10157 [cs]. [Online]. Available: http://arxiv.org/abs/2105.10157

[11] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An enhanced line differencing tool," in *2009 IEEE 31st International Conference on Software Engineering*. Vancouver, BC, Canada: IEEE, 2009, pp. 595–598. [Online]. Available: http://ieeexplore.ieee.org/document/5070564/

[12] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov. 2007. [Online]. Available: https://ieeexplore.ieee.org/document/4339230/

[13] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone Management for Evolving Software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, Sep. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6007141/

[14] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324.

[15] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: a semantic-graph differencing tool for studying changes in large code bases," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* Chicago, IL, USA: IEEE, 2004, pp. 188–197. [Online]. Available: http://ieeexplore.ieee.org/document/1357803/

[16] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Long Beach CA USA: ACM, Nov. 2005, pp. 54–65. [Online]. Available: https://dl.acm.org/doi/10.1145/1101908.1101919

[17] T. Apiwattanapong, A. Orso, and M. J. Harrold, "JDiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, no. 1, pp. 3–36, Mar. 2007. [Online]. Available: https://doi.org/10.1007/s10515-006-0002-0

[18] M. Fowler, *Refactoring: improving the design of existing code (2nd Edition)*. Addison-Wesley Professional, 2018.

[19] Z. Xing and E. Stroulia, "Refactoring Detection based on UMLDiff Change-Facts Queries," in *2006 13th Working Conference on Reverse Engineering*. Benevento, Italy: IEEE, 2006, pp. 263–274. [Online]. Available: http://ieeexplore.ieee.org/document/4023996/

[20] D. Silva, J. P. Da Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A Multi-Language Refactoring Detection Tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, Dec. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8966516/

[21] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, Mar. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9136878/

[22] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza, "PYREF: Refactoring Detection in Python Projects," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Luxembourg: IEEE, Sep. 2021, pp. 136–141. [Online]. Available: https://ieeexplore.ieee.org/document/9610674/

[23] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, May 2014, pp. 803–813. [Online]. Available: https://dl.acm.org/doi/10.1145/2568225.2568317

[24] M. Janke and P. Mader, "Graph Based Mining of Code Change Patterns from Version Control Commits," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9129812/

[25] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Oct. 2018.

[26] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987.

[27] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and efficient structural characteristic feature extraction for clone detection," in *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2009, pp. 440–455.

[28] "GitHub GraphQL API documentation." [Online]. Available: https://docs-internal.github.com/en/graphql

[29] I. IEC, "Iso/iec 14764:2006 software engineering - software life cycle processes - maintenance," *International Standards Organization, Geneva, Switzerland*, 2006.