Don’t DIY: Automatically transform legacy Python code to support structural pattern matching

Abstract—**As data becomes more and more complex as technology evolves, the need to support more complex data types in programming languages has grown. However, without proper storage and manipulation capabilities, handling such data can result in hard-to-read, difficult-to-maintain code. Therefore, programming languages continuously evolve to provide more and more ways to handle complex data. Python 3.10 introduced structural pattern matching, which serves this exact purpose: we can split complex data into relevant parts by examining its structure, and store them for later processing. Previously, we could only use the traditional conditional branching, which could have led to long chains of nested conditionals. Maintaining such code fragments can be cumbersome. In this paper, we present a complete framework to solve the aforementioned problem. Our software is capable of examining Python source code and transforming relevant conditionals into structural pattern matching. Moreover, it is able to handle nested conditionals and it is also easily extensible, thus the set of possible transformations can be easily increased.**

Index Terms—**Python, AST, structural pattern matching, code transformation**

I. Introduction

Python 3.10 introduced structural pattern matching to make complex data handling easier, as well as to make the corresponding code easier to read and maintain. It allows us to make conditional branching based on the structure of the data, and to destructure the data into smaller parts for later processing. Previously, we only had the traditional *if-elif-else* structure to do so, which could have resulted in deeply nested conditionals. Many programming languages support some form of structural pattern matching. Originally, the feature was first implemented in functional programming languages, like Scala [12] or Haskell [7]. Other languages like C# [3] or Java [11] partially implement structural pattern matching, however, some elements are missing from the offered feature. C++ supports structural pattern matching with the corresponding library called *Mach7* [16]. JavaScript supports object destructuring, however, it does not support structural pattern matching yet. [1]

In this paper, we introduce our framework which can transform traditional conditional branching to the newly introduced structural pattern matching in such a way that the program’s behavior remains unchanged, but the transformed source code becomes easier to read and maintain.

The paper is organized as follows. In Section II, we enumerate the related literature. The rules of structural pattern matching and how it works are described in Section III. We introduce our transformation framework in Section IV. The implemented transformations are presented in Section V, then we present our evaluation process in Section VI. Finally, we conclude the paper in Section VII.

II. Related Work

Refactoring was first defined by Martin Fowler [6]. According to his definition, the behavior of the code should not change, but the structure of the source code should be improved. Refactoring can have many purposes [10], e.g., improving readability. Regardless of the purpose, we can refactor our code manually, or by a tool, automatically. Source code is most often represented in a tree-based data structure (Abstract Syntax Tree (AST)). Most compilers and static analyzers [1], [5] use this representation for further analysis.

Automatic transformation can support compatibility between programming language versions. For example, due to backward compatibility issues, $2to3$ [14] was developed to convert source code written in Python 2. Similarly, in legacy environments, it can be useful to enable the use of newer language features, for which *Antal et al.* [2] created an automatic tool that can transform C++11 code into C++03.

Most IDEs include some kind of refactoring framework too. [2] There are, however, frameworks specifically designed for refactoring, where one can define exactly what to refactor and how to refactor. For example, the *ROSE* framework [19] for processing C, C++, and Fortran source code; *Proteus* [18] for C/C++ code; and the *Spoon* [13] framework for Java.

III. Structural pattern matching

Programming languages provide numerous ways to handle complex data types. One such way is *structural pattern matching*, which can destructure complex data into relevant parts and branch the execution depending on the structure of the data. In structural pattern matching, the two main factors are the data (the $Subject$) and the defined patterns. Pattern matching is based on the assumption that the $Subject$ follows a certain structural pattern that can be associated with a specific data processing method (which is specifically designed to handle the data of the defined structure). If there is also a specific condition that is independent of the pattern, we speak of conditional pattern matching. From the combination of several patterns and the associated data processing operations,

[1]The feature is already proposed (and in draft state): https://github.com/tc39/proposal-pattern-matching

[2]For example, the refactoring framework in IntelliJ IDEA: https://www.jetbrains.com/help/idea/refactoring-source-code.html

we are able to create a system that can ensure that the data is handled based on its structure, either conditionally or unconditionally [9]. The primary "output" of the pattern matching process is the success of the match, but a secondary outcome can be the assignment of values to variables that match the sample, i.e. the destructured data.

A. Structural pattern matching in Python

In this section, we briefly summarize the use of structural pattern matching in Python 3.10,[3] its functionality, and the rules that are very important for understanding the transformation framework later on. If the reader wishes to explore the topic of structural pattern matching in more depth, we recommend the official proposals PEP634 [4], PEP635 [8], and the paper [9] written by the feature's developers.

Prior to Python 3.10, structural pattern matching was only available in a very limited form. From sequential data, it is possible to extract data and assign them to variables (called "Iterable unpacking" [4]). However, it may not always make sense to store data in sequences; as it may be more convenient to use graph structures or classes and objects to represent the data. Handling the latter data types requires several nested *if-elif-else* structures, and the use of reflective methods like *isinstance, hasattr*, etc. Promising a more elegant and readable solution [17], complete structural pattern matching was introduced in Python 3.10. The syntax is similar to the $switch/case$ structure found in other languages, but it supports more complex patterns, as can be seen in Listing 1.

```
match obj: # obj is the Subject
  case Cat(color="Orange", weight = "a lot"):
    give_food(obj, Lasagne())
  case Cat(color="Black" | "Gray"): # Sub-patterns
    turn_around()
  case Cat(color = c) if cat_affinity > 0: # Guard
    print(f"Its a {c} cat!") # Using binding
  case _: # Wildcard branch
    raise ValueError("Object is not a cat!!")
```

Listing 1: Summary example of using Python structural pattern matching

Overview. The pattern matching process expects a pattern and a value (the $Subject$) as its "input". Perhaps the simplest way of describing the process is: "The value of the $Subject$ is fitted to the pattern". In the case of a successful match, some patterns can assign the $Subject$ (or part of it) to a new local variable ($Binding$), which can be used later (even outside the pattern matching block). In case of a failed match, the behavior of the binding process is intentionally unspecified to avoid semantic constraints that could limit the subsequent extension of $Binding$ [4]. A pattern may contain one or more sub-patterns. Sub-patterns are evaluated from left to right until the success of the pattern matching is determined.

Each branch can have an extra condition called $Guard$, which is in fact just a logical expression that is evaluated only if the pattern is successfully matched. While the pattern can only test the $Subject$, no such constraint is applied to the $Guard$.

Keywords, definitions. The *match* keyword indicates the start of the pattern matching block. The keyword must be followed by the $Subject$, the value of which will be used in the pattern matching process. This is followed by the branches (the usual Python indentation rules apply). A new branch is defined with the keyword *case*, then we need to specify the pattern which the value of the $Subject$ will be matched against. After the pattern, one can specify a $Guard$ with the *if* keyword. Finally, this is followed by the code block that is executed on a successful match (the usual Python indentation rules apply). It is possible to define a "default" branch using irrefutable patterns. Since branches are evaluated from top to bottom, only the last branch can be defined as such.

B. Patterns

Patterns have two main roles in pattern matching: they impose (structural) constraints on the $Subject$, and they define which values of the $Subject$ should be assigned to new variables. In contrast to the traditional iterable unpacking, there is no error generated in case of a failed match. For this reason, efforts have been made to keep the side effects of the patterns to a minimum. It is not allowed to assign values to attributes of objects or indexed values. Therefore, values can only be assigned to new local variables.

Patterns could be understood as declarative elements, commonly found in formal parameters of function definitions, as in they are not (and cannot contain) expressions. In this paper, we present only the patterns that are essential for understanding the transformation process. All patterns can be found in the corresponding PEPs [4], [8].

Class Pattern The class pattern has two main functions: to determine whether the $Subject$ is really an instance of the given class defined in the pattern, and to match patterns to specific attributes of the $Subject$. The attributes of the $Subject$ can be referenced by positional arguments, and by keyword arguments. The class of the $Subject$ is verified by an `isinstance` call.

Literal Pattern. The literal pattern allows you to set constraints not on the structure of the $Subject$, but on its value. The $Subject's$ value and the value defined by the pattern are compared using Python's general equality test (`x == y`). The possible values are, as the name of the pattern implies, Python literals. Expressions cannot be used in the patterns, so format strings, ranges, etc. cannot be used.

Singleton Pattern. The singleton pattern is very similar to the literal pattern, with the only difference being that it compares the value of the $Subject$ by identity (`x is y`). It is mainly used to compare the $Subject$ to the `True, False, None` singleton triple.

[3]https://docs.python.org/3/whatsnew/3.10.html
[4]PEP 3132, https://peps.python.org/pep-3132/

OR Pattern. As its name implies, the *or* pattern is similar to the `or` keyword in Python. It can be used to combine multiple sub-patterns, and if one is successfully matched, the whole pattern is successfully matched. Variable assignment within the or pattern is only possible if all sub-patterns assign values to the same set of variables. The sub-patterns will be evaluated in order, from left to right.

Assigning patterns. The job of an assigning pattern is to assign the value of the $Subject$ to an arbitrary name. Since assigning patterns does not set any constraints on the structure and/or the value of the $Subject$, they are grammatically irrefutable. One such pattern is the *Capture Pattern* and its special variant, the *Wildcard Pattern*. The form of *Capture Pattern* is a name. The pattern takes any value and assigns the value to this name, which will serve as a local variable. The same name cannot be used more than once within a pattern for assignment. The *Wildcard Pattern* accepts any value but does not create a new local variable. To use it, the '_' character must be used, which is a special character reserved for the same purpose in other programming languages that use structural pattern matching. The *AS pattern* can be used to define an OR pattern, and assign the $Subject$ to an arbitrary name using the `as` keyword.

IV. Transformation Framework

The source code transformation is controlled by the transformation framework. When designing the system, we put great emphasis on the possibility of future extensibility. Since the framework is practically a command line program, it can easily be integrated into any CI/CD system.

The system consists of two main parts. The analysis system ($Analyzer$ from now on) uses a plugin structure.[1] Plugins represent the recognisable patterns. The task of the $Analyzer$ is to recognize the AST received as input with the defined patterns, perform the necessary checks, and save which nodes of the received input are transformable with which plugins. The actual transformation of a node is executed by the transformer module (henceforth $Transformer$), the tasks of which include, among others, reading the source files, transforming them into ASTs, initializing the $Analyzer$, and merging the transformed code with the source files.

Using the transformation framework is very simple: the program expects the path to the source code to be transformed as a mandatory argument. This can be a single *.py* file or a path to a folder containing an entire Python project.

The goal of the framework is to increase readability and maintainability, which is of course a very subjective matter. Fine-tuning the refactoring process can be done in the provided $config.ini$ file. Because of the structure of the Python AST, each Python file must be treated as a separate entity, so as a first step, the tool finds all $.py$ files. The overview of the transformation process is shown in Figure 1.

Transformer. It is the job of the $Transformer$ to handle the transformation of a single file. After converting the source file into an AST, the AST is passed to the $Analyzer$. Using its results, the transformations are performed on the AST. After that, the AST gets converted back to source code. Converting the AST back to source code can lead to unnecessary changes in the file (e.g. missing comments) [15], therefore, it is the job of the $Transformer$ to merge the transformed code segments with the original source code. The $Transformer$ also checks the source and the transformed files for grammatical errors. In the case of an error, it reverts the transformation of the file and notifies the user within the log about the location and the nature of the error. Several optional features are included within the $Transformer$, all of which are explained later in this section.

Analyzer. The $Analyzer$ is responsible for examining the AST of the source code received as input. During preprocessing, it extracts the code fragments to be analyzed from the AST, which it breaks down into further, more manageable syntactic units, $Branches$. The $Transformer$ gets from the $Analyzer$ the list of branches that can be transformed, and the way to transform them. We only summarize the tasks of the $Analyzer$ here, the process of analyzing and transforming the AST will be explained in more depth in Section V-A.

A conditional statement can only be transformed if all of its $Branches$ are recognized by at least one Plugin. Therefore, it is the task of the $Analyzer$ to pass each $Branch$ to the plugins. If a conditional statement contains a $Branch$ that is not recognized by any plugin, then the statement is not transformable. Otherwise, additional checks are required.

It is also the $Analyzer's$ job to make sure, that inside the to-be-transformed conditional statement, every $Branch$ considers the same variable as their $Subject$. If each $Branch$ can be recognized by at least one plugin, but they cannot "settle" on a common $Subject$, then the conditional statement is not transformable.

Plugins. Perhaps the most important, and yet smallest, part of the framework, plugins are responsible for implementing mappings between the conditions of $Branches$ and the patterns of $Cases$. Their task is to recognize a pattern, determine its possible $Subjects$, and transform it. For each pattern, there is a set of variable names that contains the $Subjects$ that can be used to transform the pattern. Using these, the $Analyzer$ determines the $Subject$ of the entire conditional statement.

Some patterns may contain sub-patterns, so plugins can access each other. Thus, a kind of parent-child relationship can be established between the patterns, where the parent pattern may contain one or more sub-patterns, which can be used by the parent to determine the $Subject$ and perform the transformations. Furthermore, "complex" patterns can be defined, which can access and even modify their parent pattern. This is necessary in cases where several sub-patterns can be merged into a single, "complex" pattern.

Configuration options. Several features of the framework can be customized in the $config.ini$ configuration file. The $Transformer$ is capable of transforming the bodies of untransformable conditional statements. Since comments are not preserved in the AST, converting the transformed AST

[1]https://en.wikipedia.org/wiki/Plug-in_(computing)

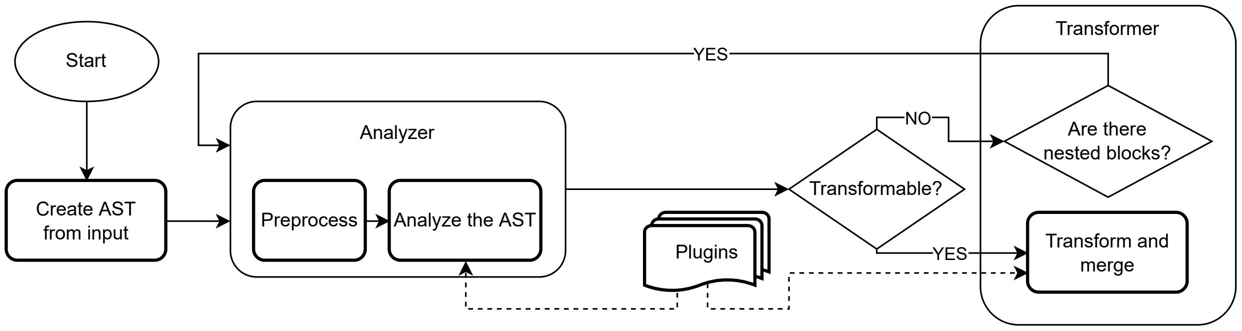

Fig. 1: The overview of our transformation process

back to source code leads to these comments being lost. The $Transformer$ can collect the comments present in the original source code if needed. We can also choose to generate a *.diff* file or edit the source code in place. The $Analyzer$ can try to "flatten" nested conditional statements. Successful flattening can greatly increase readability, which process is explained more in-depth in Section V-B. We can control the amount of code repetition in nested conditionals' flattening. We can also force the flattening, even in an unadvised situation. And of course, the minimum number of branches needed in conditional statements can be also set.

V. Implemented transformations

To understand the transformation, it is useful to highlight the differences between conditional branching and structural pattern matching. We know that conditional branching consists of $Branches$, each of which is associated with a logical expression (condition) and a code segment that will be executed if the condition is satisfied. Structural pattern matching consists of a $Subject$ and $Cases$, each of which defines a pattern, a guard, and a code segment that runs if the $Subject$ matches the pattern and the guard is evaluated to true. Hence, every conditional branch is transformable into an equivalent structural pattern by only using irrefutable patterns, and putting the original conditions into guards (see Listing 2).

```
if number == 0:
  print("It's zero!")
elif 1 == number:
  print("It's one!")
else:
  print("Default answer!")
```

⇒

```
match number:
  case _ if number == 0:
    print("It's zero!")
  case _ if 1 == number:
    print("It's one!")
  case _:
    print("Default answer!")
```

Listing 2: A seemingly unnecessary, but grammatically and logically correct transformation

Clearly, this kind of transformation looks ugly and unnecessary, as it does not improve readability. To avoid these kinds of transformations, we define a mapping, which is able to form a pattern from an input logical expression.

This mapping not only has to provide the pattern from the input logical expression, but must also be able to determine the "subject" of the condition given.

This mapping is implemented by the plugins. Their job is to guarantee that the transformed pattern will only match the determined subject if and only if the input boolean expression would evaluate to true with the same value assigned to this subject. We are presenting the implemented plugins, which, not coincidentally, correspond to the structural patterns presented in III-B. To avoid any confusion, when we talk about structural patterns we follow the *Literal pattern* notation, while for the plugins we use the *LiteralPattern* notation.

A. Recognizable patterns

Recognizable structural patterns are represented by plugins. Their task is to transform logical expressions into the structural pattern they represent, and to determine the expression's "subject".

LiteralPattern. The simplest, but perhaps the most commonly recognized pattern. It can be used to detect conditions that compare the value of the $Subject$ to a constant value. It can recognize conditions of the form `subject == literal` and `subject is singleton`. In both cases, the name of the $Subject$ and the value of the constant can easily be determined. The pattern matching process compares the value of the $Subject$ and the pattern-defined constant in the exact same way, so we can be sure that the condition and the pattern are equivalent. As previously mentioned, the singleton values *True, False* and *None* are matched similarly.

```
if a == "yes" or a is True:
    continue
elif a == "no" or a is False:
    break
```

⇒

```
match a:
    case 'yes' | True:
        continue
    case 'no' | False:
        break
```

Listing 3: An example transformation using OrPattern with its sub-patterns being Literal patterns.

OrPattern. This plugin's job is to recognize conditions that separate several expressions with the `or` keyword. Therefore it can recognize conditions of the form `E1 or E2 (or En)*`.[5] Every inner expression is considered to be a sub-pattern to be recognized by other patterns. These sub-patterns have to refer to a common $Subject$ in order for the Or pattern to be transformable. Assuming that every sub-pattern is equivalent to their respective expression, we can determine that the whole logical expression and the transformed pattern

[5]We use a hyphen to denote an arbitrary number of occurrences, as in formal grammars.

are also equivalent. Listing 3 shows an example of a successful OrPattern transformation.

GuardPattern. The GuardPattern recognizes conditions of the form `E1 and E2 (and En)*`. The expressions are considered to be sub-patterns that are recognized by other patterns. The GuardPattern can only transform the given condition if at least one of the expressions is recognizable, and has the appropriate $Subject$. If there is more than one recognizable expression, then the GuardPattern is capable of adapting to the rest of the $Branches$ in the conditional statement by selecting the expression with the appropriate $Subject$ (see Listing 4). Once an expression is selected, the rest of the expressions have to be put into the pattern's guard, unless the selected expression is considered to be a "complex" pattern, in which case that pattern gets control of the GuardPattern to potentially "merge" expression from the guard into itself. The Guard pattern can only be transformed if it is a parent pattern, since sub-patterns cannot have a guard. Assuming that the chosen pattern is equivalent to its corresponding expression, the Guard pattern is equivalent to the full condition.

```
if (a == 2) and (b == 4):
    something()
elif (b == 8 or b == 9) and c():
    something_else()
```

$\Rightarrow$

```
match b:
  case 4 if a == 2:
    something()
  case 8 | 9 if c():
    something_else()
```

Listing 4: An example of using GuardPattern, where the second branch fixes the $Subject$.

ClassPattern. The ClassPattern recognizes expressions of the form `isinstance(obj, cls)`. The Class pattern cannot only impose conditions on the $Subject$'s class, but it can also match patterns on its attributes too. Therefore, if a Class pattern is found within the sub-patterns of a Guard pattern, the ClassPattern gets the chance to merge other sub-patterns into itself. If another pattern's recognized $Subject$ is an attribute of the Class pattern's $Subject$, then that pattern can be used as a sub-pattern assigned to a keyword attribute. The second argument of the `isinstance` function can be a tuple of classes. This can be handled by the ClassPattern using the Or pattern (see Listing 5). Assuming that every argument's sub-pattern is equivalent to the relevant logical expression, we can conclude that the Class pattern as a whole is also equivalent to its original expression, since it only recognizes the `isinstance` function call, which is also used during the pattern matching process.

B. Nested conditional statements

By nesting conditional statements, we can ensure that the nested condition is only evaluated after the parent condition has been evaluated to true. It also provides the possibility to run arbitrary code between condition evaluations.

Recursive transformation. The simplest way to transform nested conditional branching. Embedded branches are transformed recursively, so code segments of any depth can be

```
if isinstance(obj, (A, B, C)):
    something()
elif isinstance(obj, (D, E)) and obj.attr == 1:
    something_else()
```

$\Downarrow$

```
match obj:
    case A() | B() | C():
        something()
    case D(attr = 1) | E(attr = 1):
        something_else()
```

Listing 5: Operation of the ClassPattern if multiple classes are given in the condition

transformed. This only happens when the parent condition is not transformable.

Flattening. Since the body of the nested branch will only execute if the parent branch's condition *and* the nested branch's condition evaluates to true, we can say that nesting the conditions is equivalent to using the `and` keyword. By exploiting this, it is possible to "flatten" nested conditional statements, i.e. to merge parent and nested branches. For each nested condition, the parent condition is added using the `and` keyword. Furthermore, if there were code segments within the parent branch before or after the nesting, they are repeated and added to the beginning and to the end of all new branches, so"flattening" may introduce code repetition. Since this transformation introduces the `and` keyword, it is clear that the transformed expression will be recognized by the GuardPattern. To avoid putting too many conditions into the guard, the transformation will only occur if the flattened branches can be further compressed via a complex pattern, otherwise the resulting flattened branches would be "ugly". See Listing 6 for an example of this process.

```
if isinstance(obj, Cat):
    if obj.color == 'black' or obj.color == 'gray':
        turn_around()
    elif obj.color == 'orange' and obj.weight == 'a lot':
        give_lasagne()
    else:
        ignore_cat()
```

$\Downarrow$

```
match obj:
    case Cat(color='black' | 'gray'):
        turn_around()
    case Cat(color='orange', weight='a lot'):
        give_lasagne()
    case Cat():
        ignore_cat()
```

Listing 6: An example of an ideal "flattening".

VI. Testing and Evaluation

We evaluated our framework from two aspects: the correctness of the transformations, and the usability in practice.

TABLE I: Details of the projects and their run times (seconds) and memory usages (MiB)

Project	SLOC	NF	NNE	NT	1 thread		2 threads		4 threads		6 threads		12 threads	
					Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
InstaPy[6]	13,405	38	1001	17	1.54	65.63	0.89	82.90	0.60	119.38	0.58	152.71	0.68	247.55
discord.py[7]	30,474	144	2325	28	2.96	65.47	1.59	86.22	0.93	120.92	0.78	157.48	0.72	255.88
pylint[8]	58,896	1276	3787	55	8.54	69.34	4.32	91.30	2.54	129.46	2.08	167.22	1.69	271.71
keras[9]	156,344	687	7342	124	13.81	78.80	6.91	107.41	3.82	141.18	2.96	188.70	2.36	308.56
django[10]	326,138	2129	9209	130	20.90	80.93	11.04	111.95	7.43	152.37	5.75	200.38	4.30	324.75
pandas[11]	358,075	1355	12,496	243	28.59	82.17	14.53	106.34	7.92	156.37	6.09	200.90	4.76	336.77

First, during development, we used regression testing in order to test the functionalities of the framework. To automate this process, we created a script that runs the tests and compares the transformed files with the reference files.

To test the usability of the framework in practice, we selected some of the most popular open-source Python repositories from GitHub that are listed in Table I. We also measured both run time [12] and memory usage on an average computer. We ran the framework with the default configuration on each project 3 times. In Table I, we can see the details of and the results of the projects: the lines of code (SLOC), the number of files (NF), the number of nodes examined (NNE), the number of transformations (NT), and the measurements data on 1, 2, 4, 6, and 12 threads. To summarize, 597 transformations were made on the 6 projects, all of which were manually validated by two of the authors. Most of them looked useful, however, some transformations seemed to be unnecessary as the transformed code was as readable as the original. We would like to note that there are several configuration options that could have been used to avoid the unnecessary transformations. We also asked 15 developers, with various Python programming skill levels (academics and industrial experts). They all agreed, that most of the transformations made the code more readable. Not surprisingly, the runtime is observed to be directly proportional to the lines of code in the transformed project.

VII. Summary

Refactoring traditional conditionals might be a very resource-intensive tasks to do by hand, and there is always room for mistakes. In this paper, we have presented a complete framework that can transform legacy Python code to support structural pattern matching. Our solution automatically analyzes and refactors the source code using Python's AST (while preserving the comments). We presented a brief introduction to structural pattern matching and described the most common patterns that exist in Python. We described how the analyzer and transformer work and discussed the architecture and capabilities of our system, as well as the configuration options. We also evaluated our framework from several aspects, and found out that it could be used in practice, either as a command line application or as a step in any CI/CD system. Moreover, as our framework is built using a plugin system, extending the framework's capabilities is easy. Both the framework and the used scripts are available in the online appendix.[13]

[12]Only the actual transformations were measured, we omitted the file copying time.

[13]https://doi.org/10.5281/zenodo.6812500

References

[1] Paul black. static analyzers in software engineering. crosstalk, the journal of defense software engineering, pages 16–17, 2009.

[2] Gábor Antal, Dávid Havas, István Siket, Árpád Beszédes, Rudolf Ferenc, and József Mihalicza. Transforming c++11 code to c++03 to support legacy compilation environments. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186, 2016.

[3] Bill Wagner and Genevieve Warren. Pattern matching overview - C guide. https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/pattern-matching, 2022. [Online; accessed 07-July-2022].

[4] Brandt Bucher and Guido van Rossum. Structural pattern matching: Specification. PEP 634, 2020. [Online; accessed 07-July-2022].

[5] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30:775–802, 06 2000.

[6] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[7] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to haskell 98, 1999.

[8] Tobias Kohn and Guido van Rossum. Structural pattern matching: Motivation and rationale. PEP 635, 2020. [Online; accessed 07-July-2022].

[9] Tobias Kohn, Guido van Rossum, Gary Brandt Bucher II, and Ivan Levkivskyi. Dynamic pattern matching with python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 85–98, 2020.

[10] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[11] Stefania Loredana Nita and Marius Iulian Mihailescu. Jdk 17: New features. In *Cryptography and Cryptanalysis in Java*, pages 9–19. Springer, 2022.

[12] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[13] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[14] 2to3 - Automated Python 2 to 3 code translation. https://docs.python.org/3/library/2to3.html. [Online; accessed 07-July-2022].

[15] Python AST module documentation. https://docs.python.org/3.10/library/ast.html). [Online; accessed 07-July-2022].

[16] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open pattern matching for c++. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 33–42, 2013.

[17] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, 2001. [Online; accessed 07-July-2022].

[18] Daniel G. Waddington and Bin Yao. High-fidelity c/c++ code transformation. *Electronic Notes in Theoretical Computer Science*, 141(4):35–56, 2005. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005).

[19] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, nov 1997.