.

Pyttern: a Python-Based Program Query Language

Julien Liénard[1], Kim Mens[1] and Siegfried Nijssen[1]

[1]*Université catholique de Louvain*

Abstract

Despite an abundance of tools available for expressing and detecting structural patterns in program source code, their steep learning curve often creates a barrier for non-expert developers. To address this issue, we present Pyttern, a program query language for Python that is easy to learn and use while maintaining sufficient expressiveness for defining and detecting small structural coding idioms. Pyttern leverages a combination of Python syntax and regex-like wildcards to identify coding patterns. To evaluate the strengths and limitations of our language prototype, we conducted a study involving 30 junior developers. Participants were asked to compare our language with established program query languages by expressing and detecting small coding idioms or flaws. Based on the feedback gathered from this study, we highlight Pyttern's strengths and areas for improvement.

Keywords

Programming Language, Query Language, Pattern Matching, Python, Static Program Analysis, Software Tools

1. Introduction

Detecting structural patterns in program source code can be useful, to check whether developers produce high-quality code that adheres to best practices [1]. In an educational context, it can help raise students' awareness of such practices and help educators assess their students' understanding of programming concepts and techniques. By identifying common misconceptions in student code, personalised feedback and instruction could be provided to overcome such mistakes.

Teachers of introductory programming courses (e.g., secondary school teachers), however, are not always expert developers. While program query languages (PQL) may help providing insights into common coding patterns or flaws observed in students' programs, teachers' lack of extensive software engineering experience may pose challenges when adopting such languages that often have a steep learning curve.

To address this issue, we propose a program query language that balances usability and expressiveness, providing a tool that is intuitive and easy to learn while still sufficiently powerful to write and detect a wide range of structural code patterns. This paper presents an initial prototype of that language, which was presented to a group of junior developers with the aim of identifying the language's strengths and limitations in practical usage.

Our proposed program query language is called Pyttern, a deliberate blend of the words 'Python' and 'pattern'. Pyttern is a Python-based query language that combines regular expression elements with Python syntax while performing its search on abstract syntax trees. Its main purpose is to help non-experts write expressive structural patterns to detect in programs, in a language that is easy to read and write by remaining close to the Python language it reasons about. In effect, the patterns to detect resemble Python programs with holes represented by expressive wildcards.

The feedback obtained on the usage of this language during our preliminary evaluation has identified several areas of improvement that will influence Pyttern's future development. By iteratively refining the language based on user feedback, we hope to improve its usability and efficacy, eventually making it a helpful resource for teachers looking for ways to analyze student coding patterns.

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium

julien.lienard@uclouvain.be (J. Liénard); kim.mens@uclouvain.be (K. Mens); siegfried.nijssen@uclouvain.be (S. Nijssen)

2. Related Work

In our evaluation (Section 4) we compare the strengths and weaknesses of Pyttern to some other program query languages. Although many different PQLs have been proposed over the years, we limited ourselves to three of them because of the limited number of participants in our study and the time they had available. We opted for *logic meta programming*, *AST traversal* and *regular expressions*, because of their prevalence and popularity in the field. They also represent vastly different approaches to program querying.

Regular expressions (RegEx) are a widely used instrument for finding patterns in character sequences. Codegex [2] and other program query languages or tools [3, 4, 5] often rely on regular expressions, among other techniques, to analyze and query code. A majority of *lint* [6] and many other program analysis tools work by parsing programs into an *abstract syntax tree* (AST) and *traversing* that tree to analyse the structure of a program. Finally, quite some research has been conducted to explore the power of *logic meta programming* to declare programming patterns as logic rules on top of object-oriented programs, reified as a logic knowledge base [7, 8, 9, 10, 11]. Each of these different approaches will be compared to Pyttern in Section 4.

In other program query languages, like Pyttern, queries are written in a syntax close to that of the host programming language. Such queries typically take the form of a code snippet, potentially incorporating additional syntax that goes beyond the capabilities of the underlying programming language. Such queries are relatively straightforward to express, even for non-expert users, assuming that they are sufficiently familiar with programming in the host language. The level of expressiveness and precision achievable with such code queries varies, depending on the user intent, language definition and the particular search engine in use.

Plain code represents the simplest form of code query, involving code snippets defined within the syntax of the host programming language itself [12, 13, 14].

Code with holes takes a different stance. Instead of relying on a search engine to determine to what similar code fragments a given code snippet matches, some search engines support queries that explicitly define one or more 'holes' within the code [15, 16, 17]. Such queries allow the user to specify holes as placeholders for part of the code. Some search engines go further and accept queries in custom languages that significantly extend an existing programming language [18, 19].

In particular, a main source of inspiration for Pyttern was SCRUPLE [15]. SCRUPLE is a tool to describe high-level patterns for querying over C source code. It contains a set of symbols that can be used as substitutes ('holes') for syntactic entities in C. The SCRUPLE pattern language derives its expressive capability from four primary categories of pattern symbols: wildcards for single syntactic entities, wildcards for collections of syntactic entities, named wildcards and some additional features.

SCRUPLE uses an extended nondeterministic finite state automaton to match patterns in the code. This Code Pattern Automaton (or CPA) uses 3 types of transitions: move to the left child, move to the right neighbour and move to the next parent. It ensures that the CPA will always terminate as every node can only be checked once with those transitions. If the CPA finishes on a terminal state, there is a match. This technique served as inspiration for the matching algorithm used in Pyttern which uses the same set of transitions when matching patterns in code. But Pyttern does not use a CPA but relies on Tree Matching between the pattern and the code AST.

3. The Pyttern language

Pyttern is a new pattern description language with a Python-like syntax that blends readability and flexibility, enabling developers of any skill level to define and detect various coding patterns and flaws. By incorporating explicit wildcard elements, Pyttern allows for clear and expressive definition of complex code patterns.

```
1 def count(lst, predicate):
2     tot = 0
3     for val in lst:
4         if predicate(val):
5             tot += 1
6     return tot
```

Listing 1: Example of an accumulator pattern in Python.

```
1 def ?(?*):
2     ?accumulator = 0
3     ?:*
4         for ? in ?:
5             ?:*
6                 ?accumulator += ?
7     return ?accumulator
```

Listing 2: Accumulator pattern in Pyttern.

```
1 def approx_pi(n):
2     var = 0
3     if n > 0:
4         for i in range(n):
5             var = (-1)**i / (2*i + 1)
6         var *= 4
7     return var
```

Listing 3: Example of a flawed instance of the accumulator pattern in Python due to the use of '=' instead of '+=' at line 5.

Motivating example To illustrate this, let's assume a teacher wants to check that their students are using an accumulator pattern as instructed to solve a specific exercise. An accumulator pattern, as illustrated in code listing 1, refers to a technique where you first initialize an accumulator variable (line 2), and then iterate over a sequence (line 3), updating the accumulator variable with the result of some operation during the iteration (line 5). In the end, the accumulator variable which retains the cumulative result of all these operations is returned as result (line 6).

A possible Pyttern pattern to check whether a Python function implements an accumulator, is shown in Listing 2. The key elements of the accumulator pattern are clearly visible on lines 2 (initialisation), 4 (iteration), 6 (accumulation) and 7 (returning) of the pattern. The remainder of the pattern consists mainly of wildcards that serve as placeholders or "holes" in the program. For example, we are not interested in the exact function name or argument list on line 1, nor the exact iteration variable or range on line 4. Other wildcards such as those on lines 3 and 5 indicate that the for-loop and accumulation, respectively, can occur at any indentation level. The code shown earlier in listing 1 matches this Pyttern (accumulator) pattern. The colours indicate what wildcards match what part of the code. The program shown in listing 3 does *not* match this pattern. It does not correctly increment the accumulation variable on line 5 since it uses a mere assignment "=" instead of the "+=" as expected.

AST matching Even though Pyttern users are not exposed to it, the core idea underlying Pyttern is to match an Abstract Syntax Tree (AST) representing a pattern, to the AST of a program in which to detect the pattern. When matching a Pyttern pattern to a program's AST, Pyttern aligns each node in the pattern's AST with corresponding nodes in the program's AST. Pyttern matches nodes using their labels starting from the root node.

If the pattern would contain no wildcards, the pattern would match the program code only if it is identical to the program. However, Pyttern offers more flexibility through wildcards, which allows a pattern to match a program by skipping some nodes in the program's AST. A one-to-one correspondence between nodes in the pattern and program ASTs is hence not rigidly enforced any more when these wildcards are employed, enabling Pyttern to identify more complex code patterns where certain nodes may be omitted or repeated in the program's structure.

Pyttern's wildcards Pyttern's syntax and semantics thus revolve around the specification and definition of matches of patterns that capture structural code characteristics within a program. They are defined using a combination of Python and regular expression-like syntax. Table 1 summarises some of the wildcards Pyttern currently supports to represent various code patterns. To make the understanding of the language easier, each of its wildcards start with the '**?**' character. A more in depth description and examples for each of the Pyttern wildcards can be found on GitHub[1].

Wildcard	Meaning
?	Matches a single node in the AST
?*	Matches zero or more nodes in an AST list node
?{n, m}	Match between n and m nodes
?***name***	Matches a single node and names it for later reference
?[***Type***, ...]	Match an AST node of any of the given types
?:	Matches the body of this wildcard at one nesting level
?:*	Matches the body of this wildcard at any nesting level

Table 1
Wildcards supported by the Pyttern language.

Some implementation details For wildcard matching, Pyttern employs a recursive approach. It relies on backtracking to take into account all potential instances of matching characters and wildcards, which is necessary to process the named wildcards correctly, as they lead to dependencies between matched parts of code that are far away from each other. The pattern matching begins by taking into account the program's top-level structures and then proceeds recursively through nested parts.

To construct ASTs for the Python programs on which the pattern will be matched, we use the native AST class provided by Python.[2] We cannot use the same AST class to produce ASTs for patterns expressed in Pyttern, as it features special wildcard constructs that don't exist in Python. Instead, we use an ANTLR [20] parser library for Python, along with a Python grammar extension that complies with Pyttern's wildcard syntactic constructs. Our main motivation for choosing ANTLR is that it is relatively straightforward to extend it with additional constructs for the wildcards.

Pyttern's matching algorithm then employs a modified Depth-First Search (DFS) strategy to align and match both ASTs. This DFS traversal is adapted to account for situations where nodes may be skipped based on the wildcards used in the pattern. The DFS traversal navigates through both ASTs at the same time to match corresponding nodes. If there are no children in both trees, the algorithm moves on to the nodes' right neighbour. Otherwise, it moves to the current nodes' first child. The method goes to the parent nodes' neighbour if there are no right neighbours. When matching a wildcard, the wildcard can change this sequence by directly going to the nodes' right neighbour or the parents' neighbour, skipping some steps in the process. We use the same principle as explained in section 2 for SCRUPLE [15]. This ensures that we always terminate the matching process.

4. Comparing Program Query Languages

Experiment To assess possible strengths and limitations of Pyttern w.r.t. other program query languages (PQL), we conducted a study involving 30 junior developers. The participants were master-

[1]https://github.com/JulienLie/pyttern/tree/main
[2]https://docs.python.org/3.12/library/ast.html

Short name	Description
`if` instead of `while`	Code fragments without a `while` loop containing an `if` where instead there should have been a `while` statement.
Erase argument value	Code fragments that accidentally or purposefully erase an argument variable by reassigning it to a constant.
Loop variable not used	Code containing a loop with a loop variable that is not used inside the loop.
Misplaced `return`	In Python, nested blocks are defined using indentation. This sometimes causes a `return` statement to be at the wrong indentation level.
Too much indentation	When writing programs, less experienced students sometimes create complex solutions with lots of different indentation levels.
`print` instead of `return`	Some students use a `print` statement instead of a `return` statement at the end of their Python function.

Table 2
Coding idioms and flaws to be detected with different PQLs.

level university students in computer science or computer science engineering who followed a course on programming paradigms. As part of this course they were asked to compare different PQLs by expressing and detecting the same coding idioms or flaws in a dataset of student code that was provided to them. The requested coding idioms and flaws can be found in Table 2. The four PQLs selected for this study were: regular expressions with Python's *re* module, AST traversal with Python's *ast* module, our Pyttern language prototype, and the use of Prolog with a set of predicates reasoning over an AST reified as logic facts. Before the study, the participants completed anonymous surveys on their background as well as on their knowledge of the two PQLs assigned to them.

The survey results are summarised in Figure 1. Our participants were primarily intermediate to advanced Python developers with little experience in program query languages and intermediate knowledge of Prolog (the basic concepts of Prolog were introduced earlier in a course they all attended).

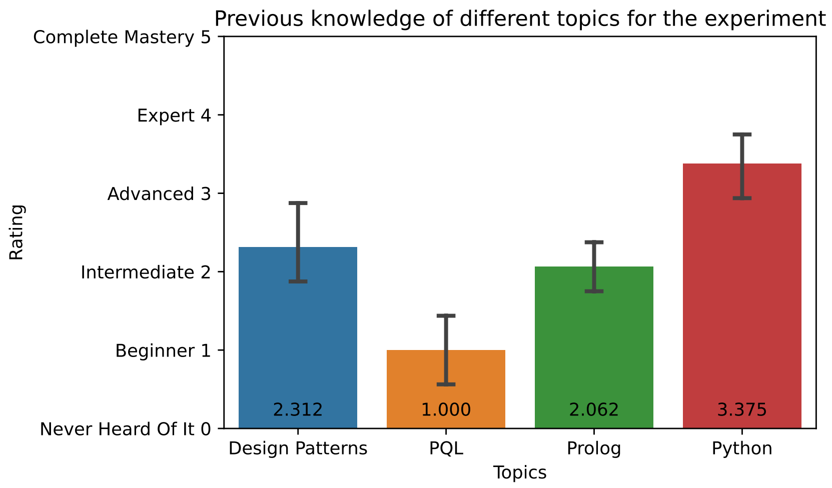

Figure 1: Background knowledge of the study participants

Next, during a two-hour session per language, they had to write program queries in two PQLs they were assigned to. After the study, they completed another anonymous survey on how they appreciated the two PQLs they experimented with. Finally, they were asked to experiment on their own with a third PQL of their choice, and write a final report of about 5 pages, to compare the different PQLs they had tried from different points of view. Table 3 shows how the languages were distributed over the different participants, in such a way that we obtained sufficient coverage of the different PQLs, but with higher coverage for Pyttern as that was the main language we wanted to compare against.

Language	Pyttern	AST Traversal	RegEx	Prolog
first	6	6	4	
second	8			8
third	3	6	5	2
Total	17	12	9	10

Table 3
Distribution of query languages over participants

Results Since in this study we observed little or no difference due to the order in which the PQLs were assigned to the particpants, in what follows we decided to group all results, regardless of the order in which participants experimented with their first or second PQL.

Several participants did confirm that Pyttern strikes a *right balance between expressiveness and ease of use*, although they also confirmed that this is the case for AST traversal. In the case of Pyttern, with its syntax close to Python, its *ease of use* comes at the cost of a *lack of expressiveness for more complex queries*. In the case of AST traversal, its *expressiveness* comes at the cost of a *steeper learning curve*.

Figure 2 indeed shows a tendency of Pyttern being appreciated as more *easy to use* than other PQLs, and in particular than using AST traversal. Figure 3 shows that Pyttern scores less than the other languages on its ability to *express more complex patterns*.

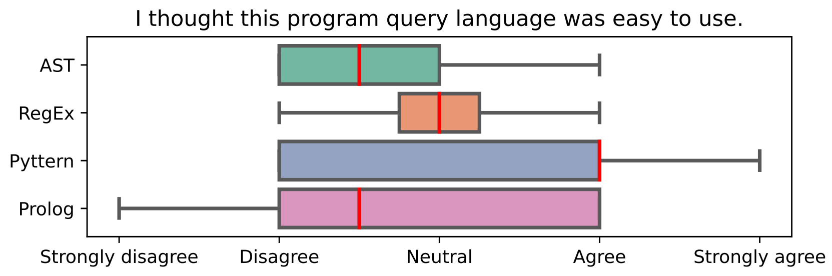

Figure 2: Box plot illustrating participants' ratings on the ease of use of different program query languages.

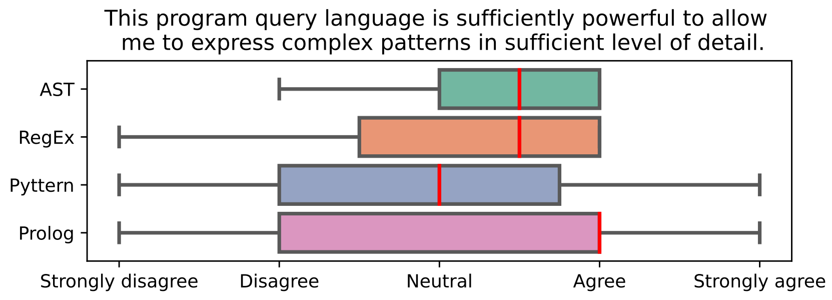

Figure 3: Participants' ratings on the expressiveness of different PQLs to capture complex code patterns.

To improve Pyttern's expressiveness, missing language features highlighted by participants were having a negation operator or the ability to combine simpler patterns into more complex ones.

As opposed to expressiveness, Pyttern scores better than logic meta programming on several criteria. As emphasized in Figure 4, it was considered (perhaps surprisingly) more **declarative** than the other languages, including Prolog. This is due to the explicit AST representation (or complex regular expressions) that queries need to manipulate in the other languages, whereas in Pyttern this remains hidden under the hood.

A frequent remark was that Pyttern still lacks extensive documentation. Because of that, occasionally students had to request help from the Pyttern developer. This is confirmed in Figure 5 and is also the case for logic metaprogramming, which was also a prototype. Due to its current lack of maturity Pyttern

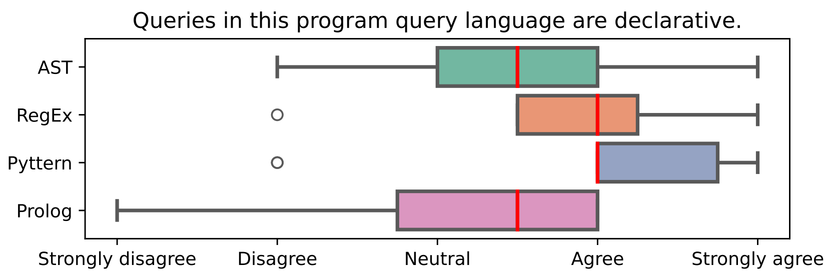

Figure 4: Box plot illustrating participants' ratings on how declarative different query languages are.

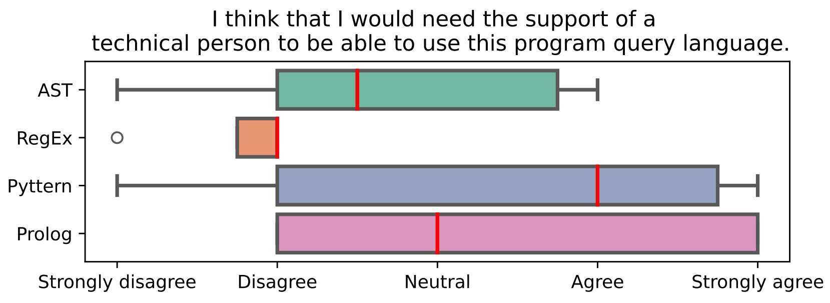

Figure 5: Participants' perceptions of their need for technical support to use the various query languages.

was also considered weak w.r.t. understanding how to debug queries. A tool similar to Regex101 that visually illustrates what parts of the code a query matches would be ideal.

Nevertheless, despite those limitations, Pyttern was considered the **easiest to learn and use** without needing a lot of background other than Pyttern.

5. Conclusion and Future Work

Conclusion Encouraged by the results of our user study, we feel that Pyttern could become an effective tool for finding symptoms of simple programming misconceptions [21] in small programs. The purpose of the first prototype of Pyttern presented in this paper was to collect initial feedback on its strengths and weaknesses in order to further improve it in the future. The user study indeed highlighted a number of issues that we had not considered before. We will improve upon Pyttern based on that feedback and conduct a more thorough validation on a more mature prototype in the future.

Limitations One limitation comes from the fact that Pyttern's parser compares AST structures in a quite literal manner. As a result, we sometimes have to define multiple alternative patterns. For example, consider the pattern defined in Listing 2. It will detect all accumulators that use the += statement, but not those using `?accumulator = ?accumulator + ?`. To overcome this, we have to define an alternative pattern using the second statement. Adding a disjuntion operator (OR) to the language could overcome this but may come at the cost of less readable patterns. Note that Pyttern's syntactic construct **?[*Type*, ...]** already provides some enhanced expressiveness. If in Listing 2 we would replace line 4 by

$$\rightarrow\rightarrow ?[For, While]:$$

this would allow the looping construct in the accumulator pattern to be either a for-loop OR a while-loop.

Another issue is that Pyttern expects methods and expressions to appear in exactly the order in which they were specified in the pattern. This is not always desired. For method definitions in a class the order should play no role, for example.

Future work Some feedback received from the study participants has already been included in Pyttern. We are increasing the speed of matching and improving the documentation, but more can be done still. Performance could be enhanced further, especially regarding backtracking, by implementing smart memoisation or tabling techniques. We also envision improving the output and providing debugging and visualisation tools to improve the self-efficacy of Pyttern users so that they need not rely on a technical person (cf. Figure 5).

We also intend to extend the expressiveness of Pyttern to include negation, the ability to express alternatives, or the ability to define more complex patterns in terms of previously defined 'subpatterns'.

Since we are building Pyttern incrementally, further validation of these new language features will be required. We already have a second experiment in preparation, where Pyttern will be validated by students following a software maintenance and evolution course. A further survey and validation with Pyttern's intended users (non-expert teachers that quickly want to express interesting structural patterns regarding their students' potential programming misconceptions) also needs to be envisaged.

Threats to validity Several factors can affect the validity of our current findings. The limited sample size of participants in the experiment may not accurately reflect the intended user base. In particular, some participants in our study may have more development experience than our target users. We observed that despite the higher learning curve, some of our participants were more in favour of using complex regular expressions or writing parse tree visitors for Python ASTs, because it would allow them to write more complex patterns.

Given that the experiment was carried out during a Master's course, there may be bias in the way the students view the tool that the assistant and course teacher developed. We kept the surveys anonymous in an effort to mitigate this bias.

For organisational reasons and to gather more data points specifically for the Pyttern language, the assignment of languages to participants was distributed semi-randomly. The distribution method and partitioning can be seen in Figure 6. This semi-random distribution could have influenced participants' perceptions and appreciation of the different languages.

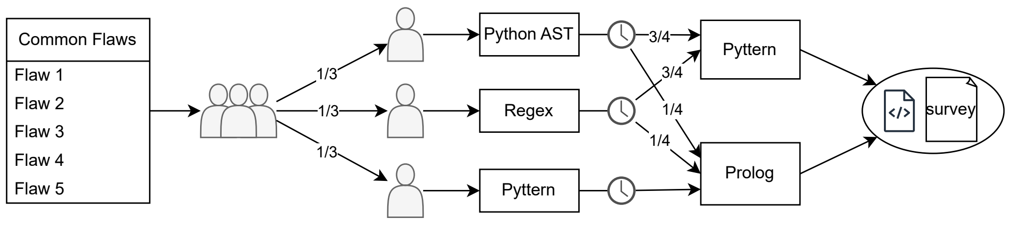

Figure 6: Assignation method and repartition for the experiment

During the study, participants who were assigned Pyttern as their first language identified some small problems and concerns with the documentation. Because the study's goal was targeted at enhancing our early Pyttern prototype, we addressed these concerns right away. As a result, individuals who were assigned Pyttern as second language received a slightly enhanced version of the prototype, which may have resulted in a different evaluation than those who used it first.

The box plots were generated based on evaluations conducted after participants had used their first two languages. After experimenting with a third language, participants were required to write a report comparing all three languages. These reports may therefore provide a more comprehensive and nuanced understanding of participants' final assessments than the box plots alone.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: Abstraction and reuse of object-oriented design, in: ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiser-

slautern, Germany, July 26–30, 1993 Proceedings 7, Springer, 1993, pp. 406–431.
[2] X. Zhang, Y. Zhou, S. H. Tan, Efficient pattern-based static analysis approach via regular-expression rules, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2023, pp. 132–143. doi:10.1109/SANER56733.2023.00022.
[3] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix, Using static analysis to find bugs, IEEE Software 25 (2008) 22–29. doi:10.1109/MS.2008.130.
[4] Spotbugs website, 2006. URL: https://spotbugs.github.io/.
[5] Spotbugs rules, 2006. URL: https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html.
[6] S. C. Johnson, Lint, a C program checker, Bell Telephone Laboratories Murray Hill, 1977.
[7] R. Wuyts, Declarative reasoning about the structure of object-oriented systems, in: Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176), IEEE, 1998, pp. 112–124.
[8] K. Mens, I. Michiels, R. Wuyts, Supporting software development through declaratively codified programming patterns, Expert Systems with Applications 23 (2002) 405–413. URL: https://www.sciencedirect.com/science/article/pii/S0957417402000763. doi:https://doi.org/10.1016/S0957-4174(02)00076-3.
[9] C. De Roover, T. D'Hondt, J. Brichau, C. Noguera, L. Duchien, Behavioral similarity matching using concrete source code templates in logic queries, in: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, 2007, pp. 92–101.
[10] E. Hajiyev, M. Verbaere, O. De Moor, Codequest: Scalable source code queries with datalog, in: ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20, Springer, 2006, pp. 2–27.
[11] K. De Volder, Jquery: A generic code browser with a declarative configuration language, in: Practical Aspects of Declarative Languages: 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006. Proceedings 8, Springer, 2006, pp. 88–102.
[12] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, Y. L. Traon, Facoy: a code-to-code search engine, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 946–957.
[13] S. Zhou, H. Zhong, B. Shen, Slampa: Recommending code snippets with statistical language model, in: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2018, pp. 79–88.
[14] S. Zhou, B. Shen, H. Zhong, Lancer: Your code tell me what you need, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 1202–1205.
[15] S. Paul, A. Prakash, A framework for source code search using program patterns, IEEE Transactions on Software Engineering 20 (1994) 463–475.
[16] A. Mishne, S. Shoham, E. Yahav, Typestate-based semantic code search over partial programs, in: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2012, pp. 997–1016.
[17] R. Mukherjee, S. Chaudhuri, C. Jermaine, Searching a database of source codes using contextualized code search, arXiv preprint arXiv:2001.03277 (2020).
[18] K. Inoue, Y. Miyamoto, D. M. German, T. Ishio, Code clone matching: A practical and effective approach to find code snippets, arXiv preprint arXiv:2003.05615 (2020).
[19] J. Lawall, G. Muller, Coccinelle: 10 years of automated evolution in the linux kernel, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018, pp. 601–614.
[20] T. Parr, The Definitive ANTLR 4 Reference, 2nd ed., Pragmatic Bookshelf, 2013.
[21] L. Chiodini, I. Moreno Santos, A. Gallidabino, A. Tafliovich, A. L. Santos, M. Hauswirth, A curated inventory of programming language misconceptions, in: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1, 2021, pp. 380–386.