COSPEX: A Program Comprehension Tool for Novice Programmers

Nakshatra Gupta*
Ashutosh Rajput*
Sridhar Chimalakonda
Research in Intelligent Software & Human Analytics (RISHA) Lab
Department of Computer Science & Engineering
Indian Institute of Technology Tirupati, India
{cs17b007,cs17b020,ch}@iittp.ac.in

ABSTRACT

Developers often encounter unfamiliar code during software maintenance which consumes a significant amount of time for comprehension, especially for novice programmers. Researchers have come up with automated techniques that can provide effective code comprehension and summaries by analyzing a source code and present key information to the developers. Existing debuggers represent the execution states of the program but they do not show the complete execution at a single point. Studies have revealed that the effort required for program comprehension can be reduced if novice programmers are provided with worked examples. Hence, we propose COSPEX (Comprehension using Summarization via Program Execution) - an *Atom plugin* that dynamically extracts key information for every line of code executed and presents it to the developers in the form of an interactive example-like dynamic information instance. As a preliminary evaluation, we presented 14 undergraduates having *Python* programming experience up to 1 year with a code comprehension task in a user survey. We observed that *COSPEX* helped novice programmers in program comprehension and improved their understanding of the code execution. The source code and tool are available at: https://github.com/rishalab/COSPEX, and the demo on *Youtube* is available at: https://youtu.be/QQY-8KuDaEM.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Documentation**; **Maintaining software**.

KEYWORDS

code summarization, dynamic summarization, program comprehension, software maintenance

*Equal contribution

1 INTRODUCTION

Program comprehension is an important activity that helps developers during software maintenance and evolution [2]. The developers often perform the essential and considerably complex task of comprehending the functionality of an unfamiliar software [9]. They primarily rely on documentation of the source code while comprehending how a particular *method* or code snippet works [15]. A natural language summary of source code facilitates the task of program comprehension by reducing developers' efforts significantly [12]. However, manually-written documentation is often effort-intensive, prone to errors and thus hinders the task of comprehension [7]. An automated code summarizer reduces developers' efforts, helps in improving quality of documentation and also improves code comprehension [7].

Information Retrieval (IR) based approaches [7, 12] and Neural Network based approaches [3, 5, 18] are widely explored in the literature [10] to address the challenge of program comprehension through summarization. These approaches do not focus on interactions between various program components and miss out on the important dynamic information associated with the code during its execution [1, 4, 5, 15]. Dynamic analysis, i.e., the analysis of a source code during run-time, does not statically approximate the behaviour of code by reasoning about a running program [1]. Researchers have leveraged dynamic analysis to facilitate the task of program comprehension [8, 14]. However, these approaches capture higher level abstraction of dynamic information that comprises only *function calls* and *return events* [13]. Sulir *et al.* [13] aimed towards extracting lower level dynamic information by collecting some of the sample values of each variable during runtime and showing value of the variable being read or written in the line. However, it is a cumbersome task for developers to keep track of the changes in variable values, if the source code file is large [16]. The approach may not be useful for program comprehension in situations where the variable values change quickly such as iterations in a loop. Though an existing built-in debugger can also display the value of any variable at a program point, it does not present any synopsis of the variable values over time [16]. Hence a developer

has to manually record changes in variable values and thus results in an additional burden on the developer [16].

Learning to program involves understanding of how a machine turns a static program written in an editor into a dynamic entity where the causal relationships between statements are important for understanding and describing the working of the program [11]. Thus an automated tool that can aid novice programmers in achieving understanding of the code and develop a better mental model construction can be of great help [17]. In a recent study conducted in a novice programming environment, Zhi *et al.* [19] emphasized that the effort required and difficulty involved in accomplishing a learning task for novice programmers can be reduced if worked examples are provided to the learners. A successful interpretation of a program involves complete understanding of the flow of the program and updates to the variable values [6]

To this end, we propose **COSPEX (Code Summarizer via Program Execution)** - a tool that extracts key information from the code dynamically and presents it in the form of worked examples to aid the program comprehension. These examples include the *arguments* passed to each individual *method*, the *values* returned by the *method* and information about changes in the values of each variable during the execution of the program. We present the information about data-flow in the generated worked example by monitoring these changes in variable values. All this data is presented in a user-friendly format with interactive features to provide key insights to the important parts. To the best of our knowledge, *COSPEX* is one of the initial dynamic analysis tools aimed at thorough analysis of *Python* source code snippets for helping programmers (especially beginners) to understand the functionality of the code and learn all the key insights from runtime through various interactive features.

2 COSPEX: DESIGN AND IMPLEMENTATION

COSPEX is implemented as a *plugin* for *Atom IDE* and aims at comprehension of *Python* code snippets by executing the snippet and extracting the required information for comprehension in the form of dynamic information instances. The tool accepts a code snippet containing test case as input. The snippet is then executed for the particular test case and key information is extracted from the program stack at each stage of execution and presented in the form of interactive dynamic information instances. Figure 1 explains the structure of the generated output for a *Quick Sort* program. Some of the key features of the generated output are:

- The dynamic summary is presented in the form of collapsible blocks, each of which represents a *function* call (Figure 1[C]).
- Each collapsible block, when expanded reveals further lower-level information inside the *function call* (Figure 1[A]).
- For each function call, the upper level information includes the parameters and return values for the call.
- Upon exploring more, the key changes to the dynamic information inside the function is presented (Figure 1[G]).
- Natural language explanation for key code components.
- For every iteration of a loop, a sliding window interface is provided to facilitate the programmer to understand the flow of data inside the loop (Figure 1[B]).
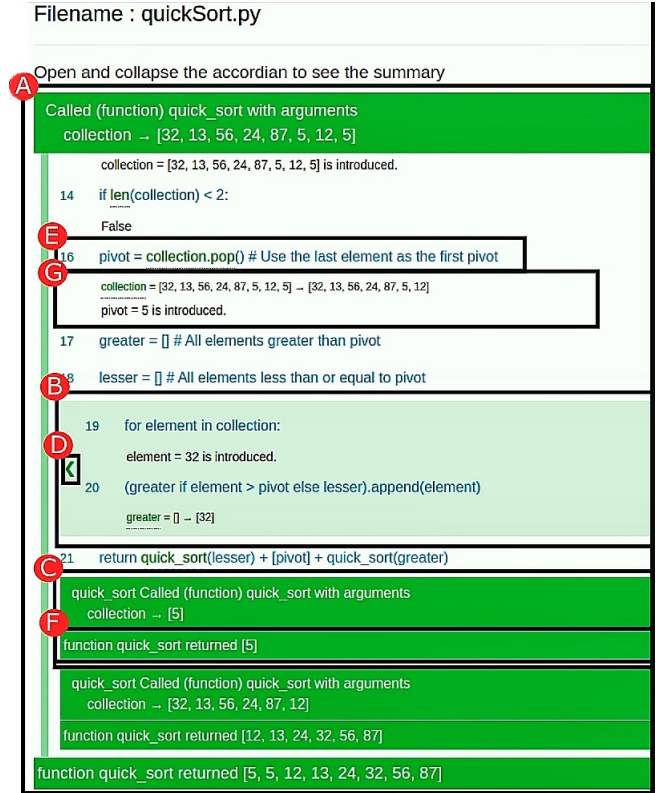

Figure 1: Part of the generated output for *Quick Sort* program. [A] is the expanded view of the first collapsible block. [B] is the sliding-window interface for loops. [C] represents the high level information for the first recursive call in a collapsed form. [F] shows the value returned by the function shown in [C]. [D] shows the arrows used to navigate through iterations of the loop.[E] highlights a line of code containing the line number, the code and comment. [G] highlights the natural language description of [E].

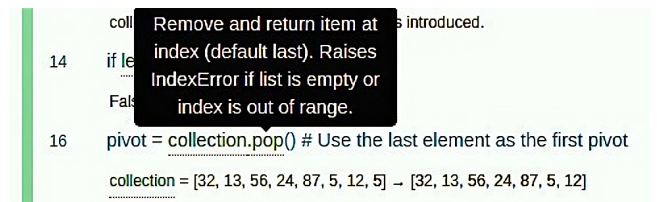**Figure 2: A pop-up appears upon hovering over in-built *Python* functions such as *pop*() which contains information about the function.**

- The tool also provides information about standard python functions upon hovering over the function name (Figure 2).
- The history of a variable till that instance is shown upon hovering the cursor over the variable (Figure 4).

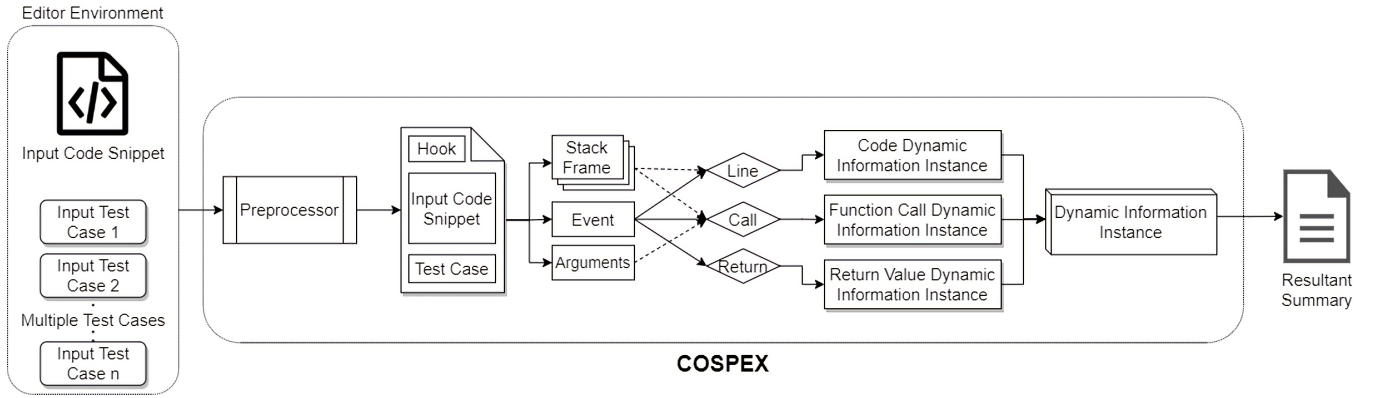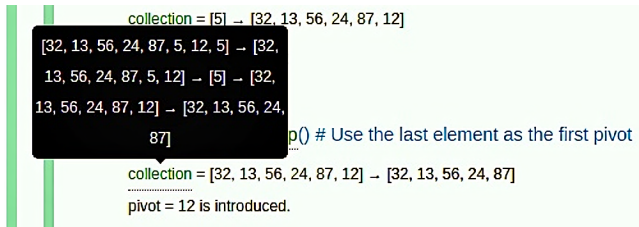*COSPEX* is implemented in 6 steps as shown in Figure 3.

Figure 3: Architecture of *COSPEX*

Figure 4: A pop-up appears upon hovering over key variables such as the *collection* list. The pop-up contains all the changes that have occurred on the variable till that instance.

Step 1 - The developer opens the code snippet in *Atom* IDE and calls the *functions* which need to be tested using appropriate test cases before invoking the tool.

Step 2 - The preprocessor combines a *hook* with input code and the test cases into a separate program which is then executed. The *hook* intercepts any *events* passed between any software components and uses callback *functions* to run different *functions* based upon its state.

Step 3 - The execution of the program is traced using the *hook*. The trace function takes 3 arguments- stack frame, *event* and arguments for the *event*. A stack frame maintains a record of all the data on the stack corresponding to a subprogram call. The value of the *event* is detected by the *hook* and it can take 3 values- *Line*, *Call* and *Return*.

Step 4 - Each *event* represents a different state of the program and provides a set of information unique to that state. We propose to call it the dynamic information instance of the state. The *events* are defined as follows:

- **Call**: When a *function* is called, a *call event* is generated. The tool extracts the arguments and the *name* of the *function* using the arguments of the *event* and the caller of the *function* from the stack frame for the *function call* dynamic information instance.
- **Line**: Before a line of code is executed, a *line event* is generated. Any change to the local variables is stored in the code dynamic information instance corresponding to the

line of code. The tool also extracts the line number and the corresponding code using the arguments of the *line event*.
- **Return**: When a *function* is returned, a *return event* is generated. The tool extracts the value returned by it and stores it in the return value dynamic information instance.

Step 5 - This information is extracted for every line of code in the program and compiled into a dynamic information instance of the complete program. The dynamic information instance stores the type of event along with the corresponding information associated with it.

Step 6 - The extracted dynamic information instance of the complete program is compiled into a dynamic summary using predefined templates. For example, if the event is "Line" and if there is a change in value of a variable then its summary is shown as "[variable name] = [old value] -> [new value]", if a new variable is introduced then its summary is shown as "[variable name] = [value] is introduced."

3 PRELIMINARY EVALUATION

We performed a preliminary user-study similar to McBurney et al. [7] to evaluate *COSPEX*. Fourteen university students of age group 18 to 21 years with varying experiences of upto 1 year in *Python* programming took part in the study. All the participants were undergraduate students of Computer Science studying in their second or fourth semester. All the participants were presented with six code snippets and the corresponding outputs generated by *COSPEX* which were chosen based on the number of *function calls*, *function interactions*, presence of loops. The snippets were of *Quick Sort*, *Fibonacci Series*, *Maximum Profit Calculation*, *Rod Cutting* (Greedy Approach), *Longest Common Subsequence* (Dynamic Programming), *Subset Generation*.

3.1 Evaluation Criteria

Based on their interpretation of *COSPEX*, the participants were requested to navigate to a user-survey form and answer 8 questions. 6 of the questions were answerable on a linear scale of 1 (Strongly Disagree) to 5 (Strongly Agree). We chose the mean and median scores of the responses for these questions as evaluation metrics. The last two questions were open-ended where the participants

No.	Question
Q1	Does the dynamic information instance help in understanding the functionality of the code snippet?
Q2	Does the dynamic information instance miss important information and that hinders the comprehension of the code snippet?
Q3	Does the information about data-flow inside a *method* help understand the working of the *method*?
Q4	Does the sliding window interface for loops help you get insights about the execution inside the loop at each step?
Q5	Does the hovering interface for inbuilt python functions that shows information about the function aid your program comprehension?
Q6	Does the hovering interface for important variables that shows all the value changes to that variable upto that particular instance aid your program comprehension?
Q7	Write a brief review of our approach.
Q8	How can we further improve the generated summary?

Table 1: Questions for the user-study and Results for Q1-Q6

were asked to review the existing tool and provide suggestions to improve the tool. Table 1 shows the questionnaire and results for Q1-Q6. The questionnaire and detailed results of the user-study are presented here[1].

3.2 Results
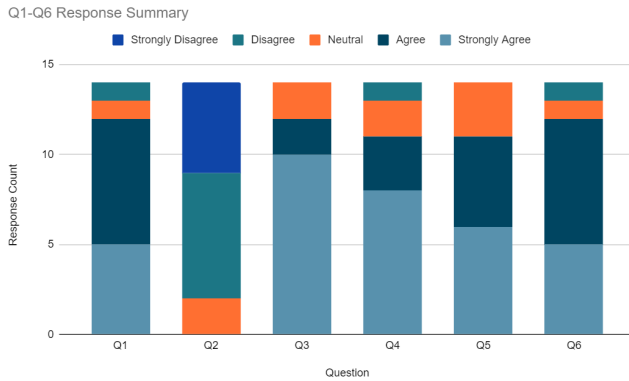
Figure 5: Summary of responses received for Q1-Q6

Firstly, the participants agreed that COSPEX helped them understand the code snippet satisfactorily with a mean score of 4.14 (median - 4). Further, they agreed that the provided dynamic instance information supports in comprehending code snippets (mean - 1.79, median - 2). The participants also agreed that the information about the data-flow inside a method proved to be useful for comprehension (mean - 4.78, median - 5). The sliding window interface for loop iterations was welcomed by the participants with a mean score of 4.55 (median - 4.5). The hovering window interface for inbuilt python function aided the program comprehension (mean

[1]https://bit.ly/3eWNn5i

- 4.21, median - 4.5). The participants also acknowledged that the hovering window interface for important variables which showed the changes to the particular variable upto that instance helped in understanding of the execution (mean - 4.14, median - 4). We recorded a standard deviation < 1 for all the answers. The summary of the responses we received for Q1-Q6 can be seen in Figure 5.

We also used two open-ended questions to understand the views of participants on *COSPEX* and a few interesting responses include:

- "*The approach summarizes the working step-wise to give a better understanding of the source code*"
- "*It explains the code in a step-by-step debugging manner which makes it easier to understand*"
- "*The approach helps people understand the complete flow of the program (step-by-step)*"
- "*Interactive interface helps understanding the summary better*"
- "*The step by step explanation helps us understand the code execution better and will especially help beginners at early stages of programming*"
- "*The interface provides insights about the working of code*"

These responses suggest that the participants validated the idea of step-by-step representation of data-flow within the program and also welcomed the interface provided for better understanding of the summary. The participants also pointed out the usefulness of the tool for novice programmers. When asked to provide suggestions for further improvement, two of the participants suggested to further add more "verbal form" to the presented summary.

4 DISCUSSION AND LIMITATIONS

COSPEX requires input test cases from developers to execute the corresponding code and extracts the required information for generating the dynamic information instance. It is an effort-intensive task for a developer to provide test cases for every snippet that needs to be analyzed. An inherent limitation to dynamic analysis is that the test cases used for information extraction may not cover all the branches of the source code. Hence, the quality of the generated dynamic information instance relies on the code-coverage provided by the test-cases. In addition, the dynamic visualization might become complex for large code snippets, and test cases. In a future version, we plan to mitigate this by allowing developers to select appropriate statements for comprehension. Based on the feedback from participants, one future direction is to improve the tool by adding more natural language phrases to the output along with pre-defined templates in the tool. Our user-study is currently limited to 14 participants and six different code snippets, but we plan to conduct a controlled experiment for complex code snippets and comprehension tasks. Also, there is an inherent bias in selecting the users, who in this case are CS students but novices to programming. As a future work, we plan to do evaluation with novice developers in industry as well.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented an approach and a tool to extract key information from *Python* code snippets to present it in a form that aids the comprehension of the snippets. Our approach is to execute the code snippets and extract information such as *function calls, return events, variable values, comments*(if present in the snippet) during

runtime. We compile this information into a dynamic information instance with interactive features that help the programmers gain key insights into the program execution. These features include a sliding window interface for loop iterations, pop-ups which appear with key information upon hovering over variable instances or standard python functions. In a user-study conducted with 14 participants, we evaluated our approach based on the manual interpretation of the participants. Each participant answered 6 questions to review the effectiveness and usefulness of our approach. Based on the results of the user-study, we observe that our approach to present the dynamically extracted information in a step-by-step manner proved to be useful in representing the meaning and functionality of the program.

As a part of future work, we intend to mitigate some of the limitations of *COSPEX*. We plan to automate the process of test-case generation for code execution. This will make the process less effort intensive and also cover all the branches of the code for better comprehension. We intend to add information about the file being analyzed. We also plan to leverage static analysis along with dynamic analysis to improve the readability of the summary by adding more natural language phrases as suggested by the participants.

REFERENCES

[1] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–36.

[2] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[3] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[4] Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. *arXiv preprint arXiv:2004.02843* (2020).

[5] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE,

[6] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. Students Struggle to Explain Their Own Program Code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 206–212.

[7] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

[8] David Rothlisberger, Marcel Harry, Walter Binder, Philippe Moret, Danilo Ansaloni, Alex Villazon, and Oscar Nierstrasz. 2012. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Transactions on Software Engineering* 38, 3 (2012), 579–591.

[9] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 308–311.

[10] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2021. Neural Code Summarization: How Far Are We? *arXiv preprint arXiv:2107.07112* (2021).

[11] Elliot Soloway. 1986. Learning to program= learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858.

[12] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.

[13] Matúš Sulír. 2018. Integrating Runtime Values with Source Code to Facilitate Program Comprehension. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 743–748.

[14] Matúš Sulír, Milan Nosál', and Jaroslav Porubän. 2016. Recording concerns in source code using annotations. *Computer Languages, Systems & Structures* 46 (2016), 44–65.

[15] Matúš Sulír and Jaroslav Porubän. 2017. Generating method documentation using concrete values from executions. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[16] Matúš Sulír and Jaroslav Porubän. 2017. Labeling source code with metadata: A survey and taxonomy. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 721–729.

[17] Lasang Jimba Tamang, Zeyad Alshaikh, Nisrine Ait Khayi, Priti Oli, and Vasile Rus. 2021. A Comparative Study of Free Self-Explanations and Socratic Tutoring Explanations for Source Code Comprehension. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 219–225.

[18] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[19] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. 2019. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 98–104.