The Journal of Systems & Software 176 (2021) 110945

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Facilitating program comprehension with call graph multilevel hierarchical abstractions☆

Rakan Alanazi [a,b,*], Gharib Gharibi [a], Yugyung Lee [a]

[a] School of Computing and Engineering, University of Missouri-Kansas City, MO, USA
[b] Faculty of Computing and Information Technology, Northern Border University, Rafha, Saudi Arabia

ARTICLE INFO

Article history:
Received 17 December 2019
Received in revised form 6 February 2021
Accepted 4 March 2021
Available online 12 March 2021

Keywords:
Program comprehension
Static analysis
Static call graphs
Machine learning
Hierarchical clustering

ABSTRACT

Program comprehension is a fundamental prerequisite for software maintenance and evolution. In order to understand a software structure, developers often read its codebase or documentation—if available and not outdated. Both approaches are tedious, time-consuming, and inefficient. Recent methods and tools have emerged to facilitate program comprehension, such as static call graphs, which depict the structure of the software system as a directed graph. However, the usage of call graphs still faces two main challenges: (1) large call graphs can be difficult to understand, and (2) they are limited to a single level of granularity, such as function calls. In this paper, we introduce a coarsening technique to create multi-level, hierarchical representations of the call graph. Specifically, we propose a hierarchical clustering approach of the execution paths to visualize the call graph at different granularity levels and for different software units, including packages, classes, and functions. Our overarching goal is to assist software developers in understanding the software system from a high-level of abstraction to the low-level of implementation with the ability to focus on particular parts of the system individually. To validate our approach and tool support, we conducted a user study of 18 software engineers from more than 11 industries who carried out several tasks using our system and then answered a survey. The results demonstrate that our approach is feasible to automatically construct multi-level abstractions of the call graph and hierarchically cluster them into meaningful abstractions. A video demo of the tool is available at https://rakanalanazi.github.io/CodEx/.

1. Introduction

Program comprehension is an imperative prerequisite for software reuse, debugging, testing, maintenance, and evolution (Cornelissen et al., 2009). In order to facilitate the task of understanding the software and its implementation, developers often read the system's documentation, i.e., a high-level description of the software system, and then manually map their understanding of the system to its low-level implementation. However, software documentation is often outdated, i.e., it does not match the current software implementation (Mkaouer et al., 2016). As a software system evolves and increases in size and complexity, understanding its implementation and structure becomes an even more challenging and time-consuming task. Manually mapping the high-level functionality to its low-level implementation is expensive, time-consuming, and error-prone. Therefore, software developers use analysis tools to understand and gain more knowledge about the system's implementation. One of the techniques used by software developers to understand the functionality and structure of a system is call graphs. A call graph depicts the system and its structure in terms of function calls or operational invocations (Naeimian, 2019; Hoogendorp, 2010; Gharibi et al., 2018b). Call graphs (Ryder, 1979; Murphy et al., 1998) aim to facilitate software comprehension and operational analysis tasks and could simplify software-related activities, such as debugging and maintenance (Tunali and Tüysüz, 2020).

A software call graph is a directed graph in which each node represents a software unit, e.g., package, class, function, and each edge represents a direct relationship, such as a function call. Using call graphs, a software developer can learn the structure and inner connections of the software system by examining an organized call graph rather than reading the implementation line-by-line. For example, Fig. 1 illustrates the call graph for one of the systems we studied in this paper. The call graph was automatically constructed and visualized using our tool, named CodEx.

☆ Editor: Raffaela Mirandola.

* Corresponding author at: School of Computing and Engineering, University of Missouri-Kansas City, MO, USA.
E-mail addresses: rna9r3@mail.umkc.edu (R. Alanazi), ggk89@mail.umkc.edu (G. Gharibi), LeeYu@umkc.edu (Y. Lee).

https://doi.org/10.1016/j.jss.2021.110945

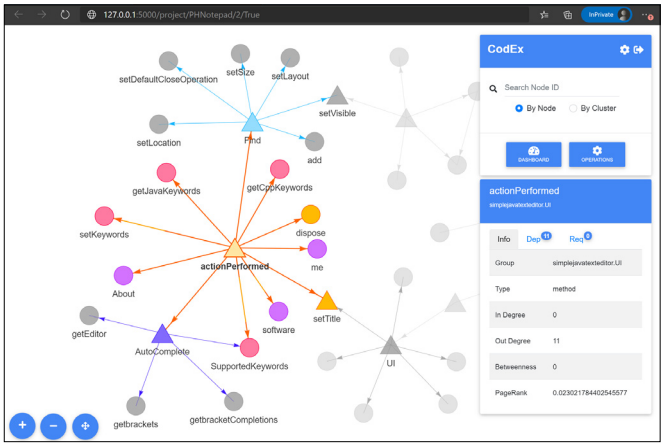


Fig. 1. An instance of a call graph example using CodEx. In this example, a node is selected, and correspondingly all of its neighbors are highlighted. Its metadata is also visualized in the side panel.

As a software system evolves over time and its complexity increases, using call graphs for system comprehension becomes more challenging. Specifically, the usage of call graphs for program comprehension still faces two main challenges:

- As the software system evolves over time, its call graph exponentially grows in size and connections. This leads to making call graphs much harder to visualize and interpret by a developer and thus increases the overhead in program comprehension.
- Execution paths can quickly grow in-depth and numbers and subsequently their encoding size, which introduces efficiency and scalability challenges during the analysis tasks, e.g., path clustering.

In order to overcome the challenges of understanding large call graphs, we present a data-driven approach, equipped with an automated tool to facilitate visualizing and understanding large call graphs of software systems. Our approach aims at creating multi-level hierarchical abstractions of the call graph with different levels of granularity for different system views (e.g., functions, classes, and packages). Our overarching goal is to facilitate the task of program comprehension.

In particular, to address the aforementioned challenges, we adapt and extend two main techniques for visualizing call graphs: multi-level graph abstraction and hierarchical clustering. The multi-level graph abstraction presents the most abstract representation of the system first (i.e., package call graph), then class call graphs, and finally reaching function call graphs. It utilizes a coarsening approach that hierarchically aggregates the functions and classes based on their dependencies to ultimately generate the package call graph (a directional call graph of the system's packages). Using the generated package call graph, the user can explore the system from a high-level of abstraction to the implementation level for simplified system comprehension. The hierarchical clustering approach aims to reduce the size of the execution paths and their complexity so that we can efficiently extract meaningful and useful information that aids program comprehension.

In the literature (Hamou-Lhadj et al., 2005; Cornelissen et al., 2007, 2009; Chan et al., 2003; Reiss and Renieris, 2001; Feng et al., 2018), various techniques have been proposed to summarize and reduce the size of the execution paths, including sampling (Chan et al., 2003), filtering (Hamou-Lhadj et al., 2005), and compression (Reiss and Renieris, 2001). These approaches are promising but are mainly limited to a single level of granularity and might result in information loss or omitting important execution paths. Recent abstract techniques, such as Feng et al. (2018), Gharibi et al. (2018b,a) introduced multiple levels of granularity for execution traces. However, these approaches lack hierarchical *visualization* of the abstraction levels and the linkage between different levels of abstractions. Moreover, these approaches are limited to call graphs representing function calls while ignoring other system units, such as packages and class hierarchies. Alternatively, our approach addresses this challenge by visualizing the call graphs at different levels of abstractions, enabling hierarchical analysis and navigation mechanisms, which in return, facilitate exploring and investigating the execution paths of a system.

The main contributions of this paper include:

- A coarsening technique to construct multi-level static call graph representations.
- A mechanism to link the hierarchical abstraction clusters to their corresponding call graphs.
- A refinement technique to project a coarse graph to finer-level graph.
- An interactive, visualization tool that enables a top-down and bottom-up analysis of the system and its execution paths for an enhanced program comprehension experience.
- A user study to evaluate the usability and usefulness of our tool.

In a previous work (Gharibi et al., 2018b), we focused on constructing and visualizing static call graphs in a single level of granularity. The current work, however, supports the automatic construction of the call graph in multi-levels of granularity. In addition, the current system design makes it much easier to be extended to other programming languages. A basic approach for clustering the execution traces was also introduced in our previous work (Gharibi et al., 2018a). We extended the approach by automatically mapping clustered paths to the system's call graph and providing new features that allow the user to navigate between different abstraction levels of the clustered graph to aid the comprehension process from a high level of abstraction to the low-level implementation. An interactive visualization tool is also developed to support the comprehension process at all abstraction levels.

The rest of this paper is organized as follows: Section 2 presents a brief background on the topic of call graphs. Section 3 presents our approach. Section 4 highlights the visualization features of our visualization tool. Section 5 explains in detail the usage of CodEx and its benefits using two case studies. We evaluate our work in Section 6 using a user study. Section 7 briefly discusses the related work. Finally, Section 8 concludes the paper and summarizes our future work.

2. Background

In this section, we present a brief background on the topic of call graphs and define some of the terms repeatedly used in the rest of this paper.

Call Graphs have been widely used to facilitate understanding the structure, evolution, and execution flow of software systems (Grove et al., 1997; Ryder, 1979; Bogar et al., 2018; Walunj et al., 2019). A call graph can be dynamic (Graham et al., 1982), constructed at runtime, or static (Murphy et al., 1998), constructed at compile time. A dynamic call graph represents a single execution path of the system. In contrast, a static call graph represents all possible execution paths of the system. Our research focuses on static call graphs, and hereafter we use the

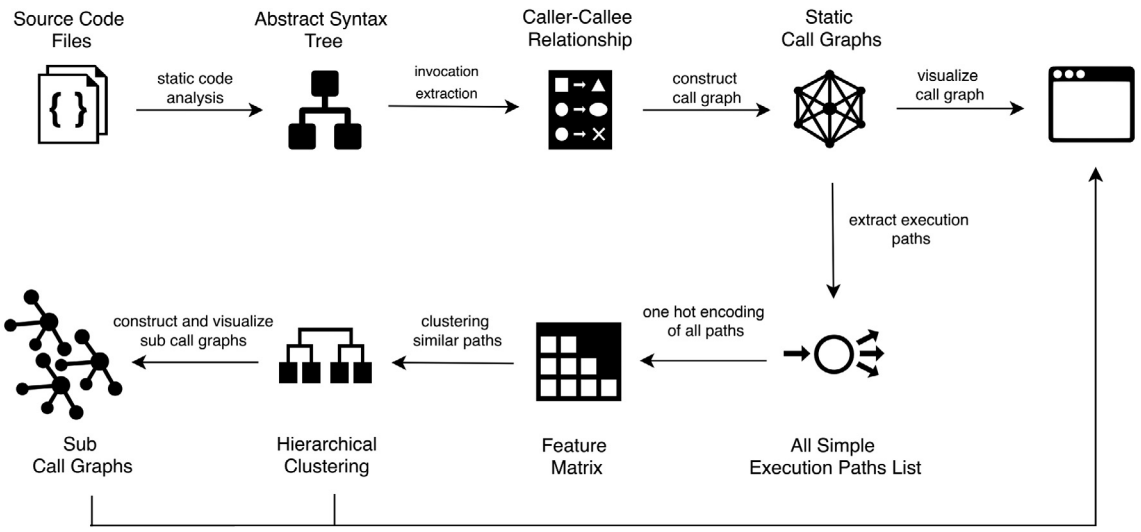


Fig. 2. Approach overview.

term "call graph" to refer to "static call graph" for simplicity unless otherwise distinguished.

In a software system, a static call graph (Murphy et al., 1998) is defined as a directed graph $\vec{G}$ = (V, E) where V is the set of v entities, and E is the set of edges where each edge, $\vec{e} \in E$, represents an entity call (u_{caller}, v_{callee}). In-degree of node v, denoted by $deg^{-}(v)$, is the total number of edges incoming to node v. The out-degree of a node v, denoted by $deg^{+}(v)$, is the total number of edges outgoing from node v.

In this paper, we suggest three different levels of call graphs: Function Call Graph (FCG), Class Call Graph (CCG), and Package Call Graph (PCG). Each graph represents a different view of the system. They are described as the following:

- Function Call Graph: Represents the low level of the system by capturing the function calls. It can be constructed from the caller–callee relationships. For example, if function v calls function u, they are represented in a function call graph as two nodes with a directed edge from v to u.
- Class Call Graph: This graph captures communication between classes, and it represents a coarse-grained function call graph. For example, given a function in class A that calls a function in class B, then the class call graph will contain node A and node B with a directed edge from A to B.
- Package Call Graph: It represents a coarse-grained class call graph. Classes belonging to the same package will be represented with a single node with the package name, and parallel edges will be removed.

3. Approach

For a large system, the developer tends to explore the system in a top-down manner (Burkhardt et al., 2002; Maalej et al., 2014). In particular, the developer first analyzes packages and selects a goal package for further investigation. Then, he/she investigates deeper into class levels and then function levels. Our approach follows the same top-down manner for exploring the execution paths. If the size of the function call graph is small and easy to understand, the developer can apply our clustering approach directly to the function call graph. For a more complicated system, the developer can use our coarsening technique to abstract the function call graph. This technique produces multi-level call graphs, including *Package Call Graph* and *Class Call Graph*. After the coarsening, the user can decide the abstraction level and apply clustering at any level according to their needs, and then these clusters can be projected to lower levels. Thus, enhancing the user's understanding of the functionality and organization of the overall system. Before explaining our approach in more detail, we first discuss our unified representation in the following subsection. Fig. 2 illustrates an overview of our approach.

3.1. Unified representation of caller–callee

The first step towards constructing a call graph is extracting the code structure and the entity relationships, i.e., calls. Call dependencies between entities can be obtained using existing static analysis tools such as PyCG (Salis et al., 2020) for Python or java-callgraph (Gousios, 2019) for Java. We extend the Java tool to output the caller–callee list in a unified representation. Thus, each edge in the caller–callee list is represented using the following format:

```
Flavor:Namespace:Identifier(Parameters)
```

Flavor represents the entity type, such as method, function, or object. **Namespace** is the fully-qualified name of the entity, which consists of the package name followed by the class name. It is used to define the scope of the identifier. **Identifier** represents the name of the entity. **Parameters** are the function arguments.

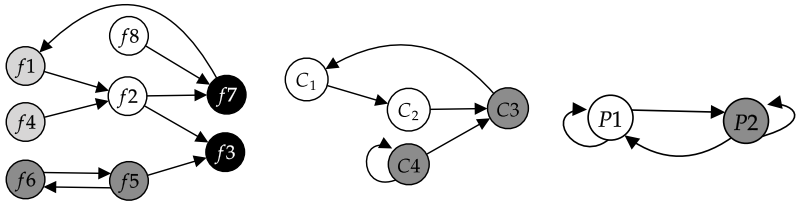
3.2. Graph construction

This phase consists of two main steps: (1) parsing the edges list to construct a function call graph, then (2) simplifying the function call graph to different levels of abstraction using the coarsening technique.

3.2.1. Graph schema

While parsing the caller–callee list, we also extract nodes' properties and their relationships using a key–value data structure. To avoid the entity name conflict problem, we maintain the fully qualified name of each entity, which consists of the namespace followed by its name and parameters. This is important for the following steps since modules, classes, and functions may have identical names across different namespaces and packages. Each entity has the following attributes:

```
M:P1.C1:F1  M:P1.C1:F2
M:P1.C1:F4  M:P1.C1:F2
M:P1.C2:F2  M:P2.C3:F7
M:P2.C3:F7  M:12.C1:F1
M:P1.C2:F2  M:P2.C3:F3
M:P1.C2:F8  M:P2.C3:F7
M:P2.C4:F6  M:P2.C4:F5
M:P2.C4:F5  M:P2.C4:F6
M:P2.C4:F5  M:P2.C3:F3
```



(a) Caller Callee List (b) Function Call Graph (c) Class Call graph (d) Package Call graph

Fig. 3. A toy example showing the coarsening technique. Reducing the number of nodes and edges by merging them based on their namespace. The node's color represents the namespace to which the node belongs. For example, functions *f1* and *f4* in (b) both belong to the same class, i.e., *c1*, and therefore they are merged into a single class node in (c).

```
{
"idx": "i"
"id": "fully qualified name of the entity"
"namespace" : "PackageName.ClassName",
"label" : "entity name",
"flavor" : "function/object/class/package"
"successors" : []
"predecessors" : []
}
```

More attributes will be added to the node after constructing the call graph, identifying the articulation nodes in the graph (Hopcroft and Tarjan, 1973). An articulation node is a node that disconnects the graph if removed.

3.2.2. Graph coarsening

Coarsening is a popular type of graph reduction, which can be defined as an aggregation process of graph nodes to identify nodes of the next coarser graph (Chevalier and Safro, 2009). An advantage of a multi-level coarsening technique is to facilitate graph data analysis by merging nodes and edges using specific criteria. In this paper, we define our own criteria to simplify the function call graph into multi-level graphs (Package and Class call graphs). Fig. 3 illustrates the coarsening technique. For example, to construct the class call graph, our graph coarsening technique takes the function call graph as an input and then collapses nodes that have the same namespace into a single node. Similarly, we construct the package call graph by merging class nodes with the same namespace, from the class call graph, into a single package node.

3.3. Execution paths extraction

In this step, we need to extract all possible execution paths from each source node s to all target nodes t. Each source node represents an entry point, and each target node represents an exit point in a system. Based on the number of in-degrees and out-degrees of a node, we can identify the type of the node: A node with $deg^{-}(u) = 0$ is called an entry point, and a node with $deg^{+}(u) = 0$ is called an exit point.

A simple execution path of a graph is a path that does not include any cycles. We use *all_simple_paths* function built-in NetworkX library (Hagberg et al., 2008) to extract all simple execution paths. This function uses a depth-first search to generate the paths in the graph between the given entry and exit points. However, before that, we extended the function to first break the back edges in the graph to remove the cycles and create more entry points and exit points. Our criteria for breaking cycles rely on the order of back edge discovery. For example, in Fig. 3.a, the edge $(F6, F5)$ comes before the edge $(F5, F6)$. Thus, the back edge, in this case, is $(F5, F6)$. And it is removed when extracting all simple execution paths.

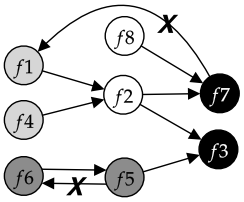


Fig. 4. Breaking graph cycles by removing back edges.

Table 1
An example of execution paths (P:path, f:function).

P_1	f_1	f_2	f_3
P_2	f_1	f_2	f_7
P_3	f_4	f_2	f_3
P_4	f_4	f_2	f_7
P_5	f_6	f_5	f_3
P_6	f_8	f_7	–

A simple execution path is a list of edges connecting a list of vertices $v_1, v_2, v_3, \ldots, v_n$ with the restrictions that all edges have the same direction. In addition, none of the edges or vertices can be repeated. This process results in a list of size P, where P is the total number of paths. Each path has a list of functions $Pi = [f1, f2, \ldots, fn]$. All paths are exported to a *CSV* file where each row represents a path. Table 1 shows an example of generated simple paths from the function call graph in Fig. 4.

3.4. Feature matrix

Before clustering the execution paths using machine learning techniques, we first preprocess our data and put it in a consumable format. Then, a feature matrix of size N ×M is generated, where N is the total number of entities, and M is the total number of features. We encode the paths in a one-hot encoding manner. Table 2 illustrates the entities (i.e., the execution paths). Each entity in the feature matrix has a feature vector, $f_i = [f_1, f_2, \ldots, f_n]$. These features represent the presence or absence of a function in a given path. Table 2 shows an example of feature matrix, which contains 6 entities (P_1–P_6) and eight binary features (f_1–f_8). We notice that f_1 is present in entities P_1 and P_2, while absent in the rest of entities (P_3-P_6).

3.5. Hierarchical clustering

Hierarchical clustering is a distance-based algorithm that uses a similarity function to measure the distance between two clusters, i.e., how close they are. It allows the developer to explore

Table 2
An example of feature matrix (*P*:path, *f*:function).

Entity/Feature	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
P_1	1	1	1	0	0	0	0	0
P_2	1	1	0	0	0	0	1	0
P_3	0	1	1	1	0	0	0	0
P_4	0	1	0	1	0	0	1	0
P_5	0	0	1	0	1	1	0	0
P_6	0	0	0	0	0	0	1	1

the data in different levels of granularity. There are two hierarchical clustering approaches, Divisive clustering (top-down) and agglomerative clustering (bottom-up). Our approach uses agglomerative hierarchical clustering (AHC) to cluster the execution paths into hierarchical abstractions.

As shown in Algorithm 1, each path is a singleton cluster, and then the two most similar clusters are joined at each step until it forms a single large cluster, which contains all the paths. There are several advantages of using AHC for our approach. AHC process is more similar to the reverse engineering approach, where the architecture of a software system is recovered in a bottom-up fashion (Wiggerts, 1997). Moreover, AHC provides different levels of abstraction and can be useful for developers to select the desired number of clusters when the results are valid and meaningful.

Algorithm 1 Agglomerative Clustering Algorithm

```
1: procedure CLUSTERING( SimMatrix, linkage )
2:     cluster ← {}
3:     for each p in paths do
4:         cluster ← cluster ∪ p
5:     while cluster ≠ 1 do
6:         Join the two closest clusters
7:         Update the distance matrix
```

Hierarchical clustering does not require specifying the number of clusters. However, performing this clustering requires two things: (1) similarity measures of execution paths and (2) a linkage type. Several researchers conducted experiments on a set of systems to compare various similarity measures, and linkage types (Maqbool and Babri, 2007; Davey and Burd, 2000; Anquetil and Lethbridge, 1999). They concluded that Jaccard similarity produces more reliable measurements as well as the complete linkage algorithm. Thus, we present our results using the Jaccard Similarity as a similarity measure and the complete link as the linkage type.

3.5.1. Similarity measures

To measure the similarity between a pair of entities, we used Jaccard similarity (Maqbool and Babri, 2004), which measures the dissimilarity between two sets. It is widely used in clustering problems, such as text clustering. It can be calculated by computing the size of the intersections divided by the size of the union of two sets and then subtracting the result from one, as shown in the following Equation.

$$d(P_i, P_j) = 1 - \frac{|P_i \cap P_j|}{|P_i \cup P_j|}$$

3.5.2. The linkage type

Hierarchical Agglomerative clustering comes with different variants to measure the distance between two clusters, known as linkages. There are three main types of linkages used in many software architecture recovery techniques (Maqbool and Babri, 2004, 2007). The types include Single Linkage (SL), Complete Linkage (CL), and Average Linkage (AL). We use the complete linkage method in our work to measure the distance between two clusters, which is identified by taking the farthest point i in cluster C_1 from the most distant point j in cluster C_2 as shown in the following Equation.

$$d(C_i, C_j) = \max(d(C_1[i], C_2[j]))$$

Table 3
An example of a clusters table (C: cluster, P: path).

C_1	P_1					
C_2	P_2					
C_3	P_3					
C_4	P_4					
C_5	P_5					
C_6	P_6					
C_7	P_1	P_2				
C_8	P_3	P_4				
C_9	P_1	P_2	P_3	P_4		
C_{10}	P_1	P_2	P_3	P_4	P_5	
C_{11}	P_1	P_2	P_3	P_4	P_5	P_6

3.6. Converting cluster to graph

Hierarchical clustering produces a dendrogram, which illustrates how the AHC is performed in a bottom-up approach. AHC starts with the low-level execution paths up to the root, where the linkage algorithm is completed. We have different levels of abstractions for a given system in the form of a tree. To get a multi-level granularity of the system call graph, we need first to convert these clusters to graphs. We first extract cluster data from dendrograms, such as paths that belong to each cluster. This process results in a list of size C, where C is the total number of clusters. Each cluster has a list of paths $Ci = [P1, P2, \ldots, Pn]$. All clusters are exported to a *CSV* file, where each row represents a cluster, and each column represents a path ID. Table 3 lists an example of extracted clusters from a given dendrogram example.

Algorithm 2 takes *cluster id* as input to retrieve all the paths that are part of the cluster. Then each path in the selected cluster will be extracted from the path file. This can be done by pointing *path id* to its corresponding line in the path file. Finally, we pass these paths into *G* to build a call graph of the cluster. Later, the call graph of the cluster will be mapped to the original call graph using CodEx (see Section 4).

Algorithm 2 Cluster to Call Graph Conversion

```
1: procedure CLUSTER_TO_GRAPH( cluster_id)
2:     pathsList ← GetClusterPaths(cluster_id)
3:     G ← DiGraph()
4:     for path_id ∈ pathList do
5:         path ← GetPath(path_id)
6:         G.add_path([path])
7:     return G
```

3.7. Mapping from high-level to low-level call graph

Our approach constructs different levels of the call graph automatically. First, it constructs a function call graph and then coarsens it into different levels of abstraction (i.e., class and package call graphs). The developer often can start from the most abstract level (i.e., package call graph) and then selects a specific cluster to investigate it further. Then, a refinement technique is applied to project the clustered graph back to a more detailed call

graph (e.g., class or function call graph). We automate this process by mapping each node in the clustered graph to its corresponding node at the finer-level graph using the attribute namespace. It is important to note that we use the attribute namespace to coarsen call graphs, and therefore, the same attributes are used to expand the node back to its original representation. The implementation details are presented in Algorithm 3.

Algorithm 3 Mapping High-Level to Low-level Call Graph

```
1:  procedure MAPPING_HI_TO_LOW( G_cluster , G_org )
2:      R ← DiGraph()
3:      for each edge(u, v) ∈ E_org do
4:          i ← u[namespace]
5:          j ← v[namespace]
6:          if i, j ∈ V_cluster then
7:              if (i, j) ∈ E_cluster or i == j then
8:                  R.add_node(u)
9:                  R.add_node(v)
10:                 R.add_edge((u, v))
11:     return R
```

The algorithm takes two graphs $G_{cluster}$ and G_{org} as inputs, assuming that $G_{cluster}$ is the selected cluster at the coarsest level, and G_{org} is the fine-grained of the coarsest graph. For example, if a cluster is selected from the package call graph, the $G_{cluster}$ represents the cluster, while G_{org} is the class call graph. The algorithm first constructs a new graph, named R, to obtain the corresponding nodes and edges in both graphs. Each node in both graphs has several attributes (see graph schema in Section 3.2.1). For each edge (u, v) of G_{org}, we use the attribute namespace as a label for both nodes, $u_{namespace}$ and $v_{namespace}$, and check if their labels exist in $G_{cluster}$. This condition confirms that both nodes u and v are in the scope of $G_{cluster}$. The second condition is to make sure that the corresponding edges are only obtained at each iteration. The same algorithm can be applied to project graph R to the next finer-level. This case is regarded as the function call graph.

4. Software visualization

Our previous work (Gharibi et al., 2018a) relied on Horner (2021) to generate the call graph of a given system. However, similar to many existing call graph visualization systems, the generated figures are often presented using a static format such as SVG, PNG, and DOT formats. The graphs of large systems can become very complex and challenging to analyze and understand. Thus, graph entities such as nodes, edges, and associated attributes will likely be visually overlapping due to limited space on the screen and the canvas size, which makes understanding such static graphs tedious and challenging. To overcome these issues, we develop our own visualization tool, which provides a dynamic, browser-based, user-friendly, intuitive interface. CodEx presents a new set of features that allow the user to interactively explore the call graph in different views, with a side panel to facilitate exploring the metadata of the graph and its nodes. The tool displays different graph levels and views in separate tabs in a browser to help developers studying each level separately. We describe these features and demonstrate them with examples in the following subsections.

- Nodes Shapes and Colors: Node-link technique is commonly used to visualize and explore relationships between a software system entities (Merino et al., 2018). Thus, we used the node-link technique to represent the call graph of a system. Graph nodes are colored based on the file or module to which they belong to. This can help the developer see how functions/classes of different modules/packages interact with each other. Another node feature is its shape. We use a circle to represent an entity and arrows to represent relationships between the entities. Triangle nodes represent articulation points. Articulation points (also known as cut vertices) can disconnect the graph into two or more sub-graphs if removed.
- Highlighting Nodes: When the user selects a node by clicking it, the tool highlights the selected node and its neighbors, while coloring the rest of the graph with a transparent Gray color. This feature helps developers to focus on particular functions and their relationships. Moreover, selecting a node opens a side panel with its metadata.
  The user can browse the metadata, including node's type, in-degree, out-degree, and graph matrices. Also, clicking on the Dependent tab will show all the node neighbors while clicking on the Required tab will show all nodes that call the current node.
- Search and Filtering: When the user clicks on the dashboard button, a modal will show up, providing valuable information about the system call graph, such as the number of nodes, edges, entry points, and exit points. In addition, the user can view, search, and filter nodes and their metadata using a provided table.
- Mapping Clusters to a Call Graph: We integrated the mapping method into the tool to map the hierarchical clusters and the original call graph visual and automated. When the developer clicks on a cluster of interest in the hierarchical view, the tool will convert the cluster to a call graph using Algorithm 2. The developer will then navigate to a graph view in the browser tab and insert the cluster number in the search box. If a particular cluster is selected, its nodes and paths will be colored with a bright color and the rest of the graph recedes into Gray. For example, in Fig. 5, Cluster 985 is selected in the search box, and its relevant nodes and paths are highlighted.
- Multi Hierarchical Clustering View: The dendrogram is one of the popular formats for presenting hierarchical clusters. However, when the size of the graph increases, it becomes difficult to read and identify clusters, especially when horizontal lines connecting clusters overlap (Sander et al., 2003). To overcome this issue, we extend CodEx to view the hierarchical clustering results in two different layouts: force-directed tree and hierarchical design.
- Saving and Loading: CodEx can save the results and state of each process in a different format (e.g., GML, JSON, CSV). When a call graph is rendered, the layout of the nodes and edges in the graph will be saved and then could be reloaded for further analysis.

4.1. Implementation

We implemented the proposed methods using Python programming language. We have also developed a Web application for the visualization using Flask (Grinberg, 2018) for the back-end and Bootstrap (Mark Otto, 2019) for the front-end.

Defining Nodes and Edges: The first step towards constructing the call graph is to define and extract entities and their dependencies. We define functions and objects as entities because considering only functions will ignore the object-oriented structure of the system (Naseem et al., 2013). They are also regarded as essential components of traditional systems and represent the functionality of a system more clearly than other components. Relationships between these entities are function calls and class instantiating. To extract the objects, functions, and their dependencies, we integrated an existing static analysis tool, (Gousios,

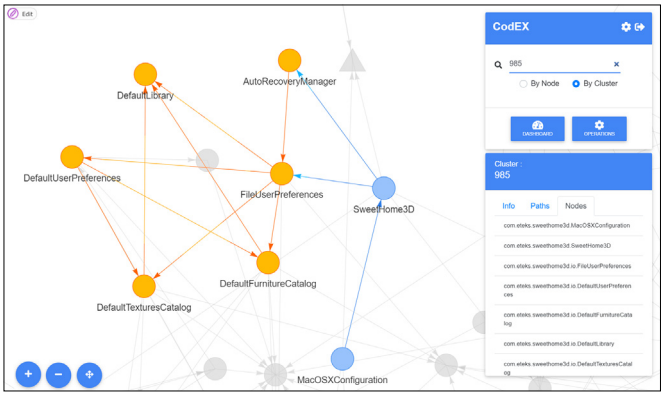


Fig. 5. Highlighting a cluster by ID search. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 4
Subject systems.

System	Version	Description	LOC
PHNotepad	3.0	Code editor	962
SweetHome 3D	6.2	Interior 2D design	109,970

2019) to generate the caller–callee list as a text file. The analysis tool uses the Apache Byte Code Engineering Library (BCEL) to analyze a given *.jar* file as input.

Constructing and Visualizing the Call Graph: We parse the caller–callee relationships file to construct a graph using a generic graph data structure, NetworkX (Hagberg et al., 2008). This library was used to manipulate and render the structure of the call graph. Then, we created a JSON schema to obtain node properties, see Section 3.2.1. After constructing the call graph, the results are saved in different file formats, including GML and JSON. JSON schema is used for visualization of the call graph.

Hierarchical Clustering of the Execution Paths: To construct multiple levels of abstractions for the call graph, we first extract execution paths of the system using a modified version of the depth-first search (DFS) algorithm that is built-in NetworkX. Second, we cluster the execution paths using the AHC algorithm with *Numpy* and *Scipy* libraries (Oliphant, 2007).

To fit the machine learning model onto the generated paths, we built a feature matrix, see Section 3.4, which is fed into the hierarchical clustering algorithm. We used the same parameter settings as defined in our original approach, where the similarity matrix is Jaccard with the complete linkage type.

5. Case study

This section presents a case study using two illustrative examples to illustrate the tool's applicability and usefulness. In the first case study, we examined if we can generate meaningful hierarchical clusters at different granularity levels. In the second case study, we illustrate how we alleviate the challenges of understating complex call graph by adopting a multi-level graph abstraction technique. Table 4 summarizes the subject systems implemented in Java. We present the results with higher resolution and readable images for all case studies at the tools' website (Alanazi, 2021).

5.1. Example I: PHNotepad

In this example, we applied our approach to PHNotepad, a Java code editor written in Java. The tool is equipped with operational features such as search, auto-completion, and intuitive,

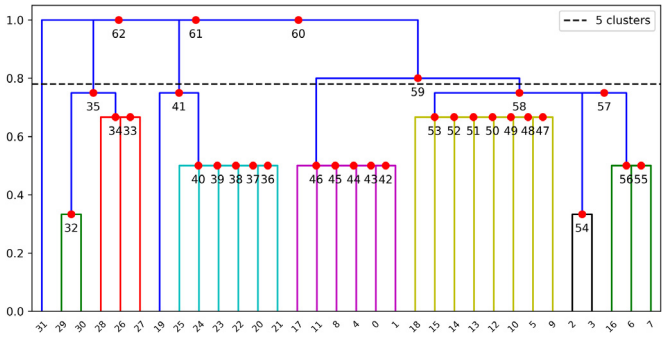


Fig. 6. Dendrogram of PHNotepad. Number of clusters is determined by using Calinski–Harabasz score.

Table 5
Call graph of PHNotepad in class level and function level.

Entity	Class call graph	Function call graph
Nodes	6	36
Edges	7	36
Entry point	1	4
Exit point	3	28
Paths	7	32

Table 6
Call graphs of SweetHome3D in package, class and function levels.

Entity	*PCG*	*CCG*	*FCG*
Nodes	9	196	5148
Edges	28	901	9501
Entry point	2	10	1517
Exit point	2	39	2821

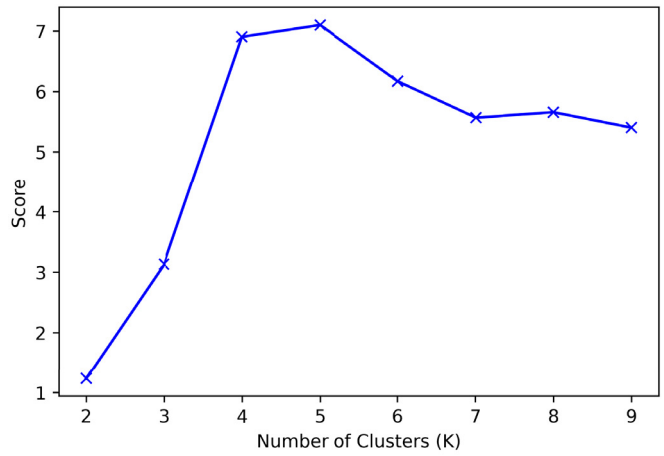
easy-to-use GUI. Although the program system has 962 lines of code, we selected this system due to its outstanding design and manageable size for manual analysis and verification.

One of the challenges in unsupervised learning algorithms is to evaluate whether the used clustering algorithm produces meaningful results. Thus, our first example focused on a manageable size subject that enabled us to inspect the results manually and evaluate their significance. Also, we wanted to examine if our approach can identify the functionalities of the system.

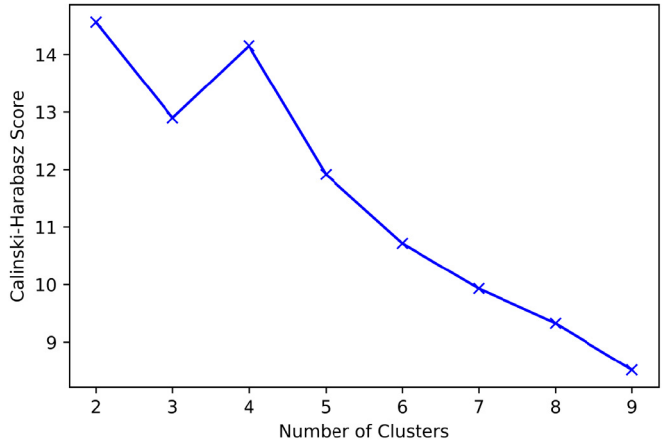
The tool automatically generates different levels of the call graph for a given software system using the coarsening technique presented in Section 3. Since PHNotepad has only one package, the tool automatically ignores the package call graph. Table 5 shows the analysis results for the class and function call graphs. We notice that the system has a single entry point in the class level graph. Using the search feature in CodEx, it is noted that the entry point is a class called '*SimpleJavaTextEditor*' that contains the main method calling all the other methods required to run the application. Due to the small size of the class call graph, we will ignore class call graphs and apply our approach to the function call graph.

The results of the clustered function call graph are represented as a dendrogram in Fig. 6. In order to determine the number of clusters, we use Calinski–Harabasz to compute the optimal number of clusters, *K*. Fig. 7(a) presents the scores of different values of *K* ranging from 2 to 10. The highest Calinski–Harabasz refers to the optimal value of *K*. Based on Calinski–Harabasz, the *K* value of 5 yielded the highest score, while *k=2* yielded the lowest score. Thus, the functional call graph data were clustered into five groups in this example.

In Fig. 6, the Agglomerative hierarchical clustering is reasonably close to the software structure of the system. We notice that Cluster 31 does not group with any other cluster except the



(a) PHNotepad FCG



(b) SweetHome3D PCG

Fig. 7. Evaluating the result of AHC using the Calinski–Harabasz score.

last cluster due to the nature of the algorithm, where all clusters eventually will group together. We used CodEx to visualize the function call graph of Cluster 31 (Fig. 8.a). We found that Cluster 31 represents the auto-complete feature for matching brackets. Also, we observed that this cluster represents a disconnected graph, which is shown as a singleton cluster in the dendrogram. With more investigation with the function call graph, we found that these two nodes depend on functions from external libraries, which was not considered during the call graph construction. For Cluster 35, when we mapped it to the call graph (Fig. 8.b), we found that it represents the search feature in PHNotepad. Cluster 19 was initially a singleton cluster, but was merged with Cluster 40 due to sharing a node, i.e., the main method (Fig. 8.c). Cluster 40 (Fig. 8.d) handles the initialization of the main graphical interface of the system. Our results represent that there are strong relationships between some clusters. Cluster 46 (Fig. 8.e) handles the graphical interface of the search feature in the system. Cluster 53 (Fig. 8.f) handles all button actions in main interface. Cluster 57 (Fig. 8.g) handles the auto-complete feature. Cluster 62 (Fig. 8.h), which is the root, represents the complete call graph without missing any node or edge.

Our example confirms that the call graph results generated correct execution paths of the PHNotepad system. Thus, it is demonstrated that the proposed methods can be used for building abstractions automatically by extracting call graphs of a software system and clustering them. In addition, the visualization tool is very useful for further exploration and comprehension of the automatically generated call graphs.

5.2. Example II: SweetHome3D

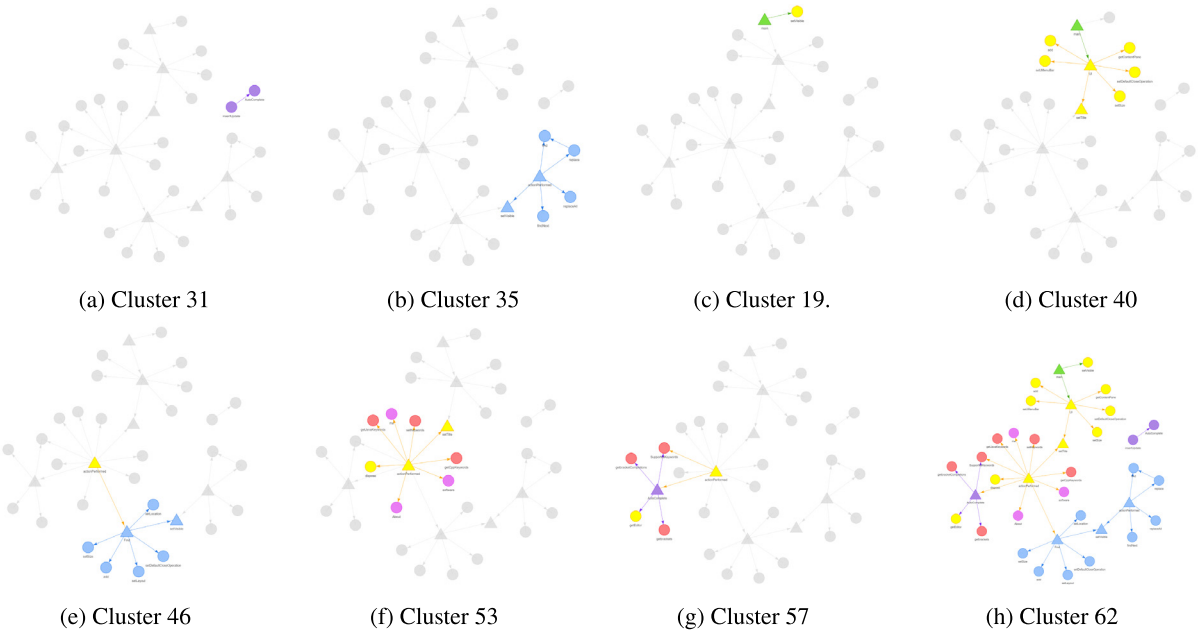
In this example, we applied our approach to SweetHome3D, an interior design application written in Java. The user can design a house in 2D by drawing the plan of the house, adding furniture and home appliance, and then preview it in 3D. It comes with many easy-to-use features, including import texture and furniture, recording videos, and allowing user-customized preferences, such as setting the system language, fonts, and units of measurement.

The system includes two versions: desktop and applet. Our goal here is to validate the feasibility of our system in helping the developer differentiate between the source code of the two different versions and then further explore the desktop version. This will illustrate that our approach is useful for bridging the cognitive gap and facilitating comprehension tasks.

This example showcases the task of understanding a system of multiple packages. For a medium to a large-sized system, the developer tends to explore in a top-down manner. The developer starts the program comprehension process by analyzing the packages of the system, then he/she dives deeper and deeper into the classes, functions, and, finally, the source code. Similarly, our approach provides a top-down approach to analyze the system. We start by constructing and analyzing the package call graph, and then particular clusters are selected for more in-depth analysis. The user can navigate from the package graph down to the class graph for further investigation. Finally, we repeat the same process on the class graph to go deeper into a function graph.

In the following, we analyze the results for each level as follows:

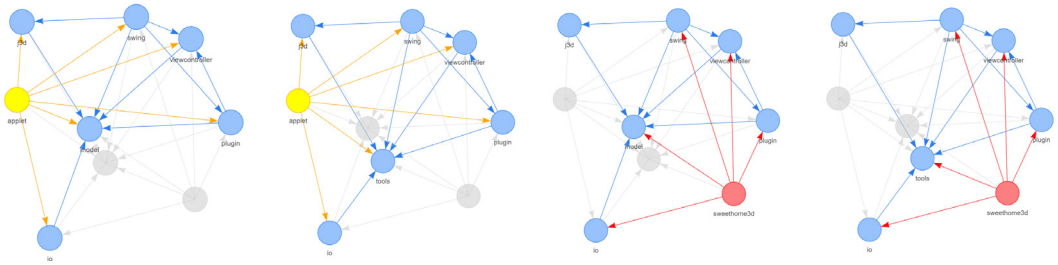
- **Package Call Graph**: Table 6 depicts SweetHome3D package call graph, which includes 9 packages (nodes), 28 edges, and 2 entry points of two main methods. The first main method runs the desktop version, and the second main method runs the applet version. The clustering of the package call graph (PCG) generated from the proposed methods is shown in the form of a dendrogram. The results of the hierarchical clustering are shown in Fig. 10. The dendrogram can be divided into two or four groups.
  As mentioned previously, we used the Calinski–Harabasz score for finding an optimal number of clusters, as illustrated in Fig. 7(b). The first-best and the second-best optimal numbers of the clusters are 2 and 4, respectively. We obtained similar results from the AHC. We further explored the meaning of the four clusters, namely Clusters 53, 63, 74, and 84. From Fig. 9, we observed that our approach successfully differentiates between the two versions of SweetHome3D—desktop and applet. As seen from the dendrogram in Fig. 10, Clusters 53 and 63 were merged into Cluster 64, representing the applet version. Similarly, Clusters 74 and 84 were merged into Cluster 85, which represents the desktop version.
- **Class Call Graph**: The next step is to explore the class level. First, we will choose one of the clusters that belong to the desktop version, Clusters 74 and 84. In this scenario, we will investigate Cluster 84 by selecting one of its sub-clusters (i.e., Cluster 83). Then, convert it to a call graph using the conversion method (see Fig. 11.B). After that, uncoarsen the cluster graph from the package level to the class level (see Fig. 11.C) using our mapping method, as discussed in Section 3.7.
  Fig. 11.C shows the class graph. It has five different colors which represent the following packages: The green nodes



(a) Cluster 31 (b) Cluster 35 (c) Cluster 19. (d) Cluster 40

(e) Cluster 46 (f) Cluster 53 (g) Cluster 57 (h) Cluster 62

Fig. 8. Converting PHNotepad clusters from the dendrogram to call graphs. Node's color represents the class to which the function belongs. (a) Auto-completed matching brackets. (b) Representing the search feature action events. (c) Setting GUI components visible. (d) Providing the Main GUI (e) Handling GUI of the search feature. (f) Handling all buttons' actions in the main interface. (g) Handling the auto-complete programming language keywords feature. (h) Presenting the function call graph of system. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) Cluster 53 (b) Cluster 63 (c) Cluster 74 (d) Cluster 84

Fig. 9. Call graphs of Cluster 53, 63, 74, and 84. The red node represents the package having a main class for the desktop version, while the yellow node represents the package having a main method for the applet version.

are in the '*io*' package, the yellow nodes are in the '*tool*' package, the blue nodes are in the '*swing*' package, the red represents the '*j3d*' package, and the pink nodes belong to the '*sweethome3d*' package.

Table 7 depicts the uncoarsened class call graph (CCG'). Comparing the (CCG') to the original class graph (CCG), the number of nodes was decreased by 54%, from 196 to 107, while about 27% of the edges were decreased, from 901 to 247. In addition, *CCG' is more manageable and more specific to a certain domain*. To further investigate CCG', we apply our approach to this call graph. Then, we randomly select a cluster, 985 (see Fig. 11.D). This cluster has eight classes from two different packages (*sweethome3d*, and *io*). The classes from the '*io*' package are related to processes of reading and setting the default user preferences in the system.

- **Function Call Graph**: To investigate the function level of the selected cluster, we uncoarsen CCG'_{985} to produce function call graph (FCG_{985}). Fig. 11.E shows the result of uncoarsening the graph. The produced graph has eight different colors that represent classes, to which functions belong. We notice that the graph has three big star networks. We apply our approach to see if this call graph can be partitioned further. Our approach can successfully partition the call graph. For example, Cluster 474 represents one of the main star networks. Using CodEx, we found that the cluster handles the user preferences functionality, such as setting the system's language or changing the measurement units.

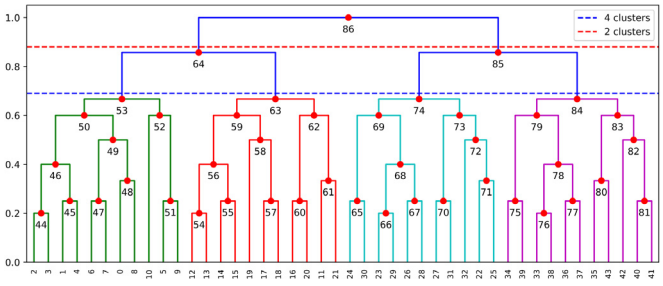


Fig. 10. PCG Dendrogram of SweetHome3D. Number of clusters is determined by using Calinski–Harabasz score. Cluster 64 represents the applet version while cluster 85 represents desktop version.

Another interesting observation is that the blue nodes represent the "*set*" methods that belong to class 'FileUserPreferences,' while the yellow nodes represent the "*get*" methods that belong to another class named 'DefaultUserPreferences'.

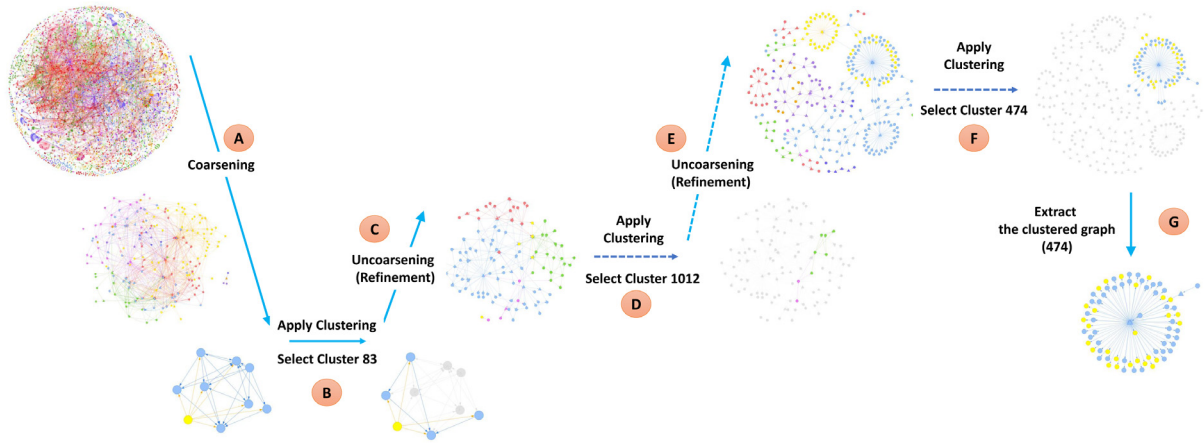


Fig. 11. Resulted call graphs during our comprehension process on SweetHome3D. (A) Coarsening the function call graph to multi-levels of abstraction. (B) Applying our clustering approach and select Cluster 83. (C) Uncoarsening Cluster 83 to class-level. (D) Applying our approach to the clustered graph and select Cluster 985. (E) Uncoarsening the call graph of Cluster 985 to the function-level. (F) Applying our approach and select Cluster 474. (G) Extract the call graph of Cluster 474. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 7
Structural characteristics of call graphs at different levels in SweetHome3D.

Entity	PCG_{org}	$PCG_{cluster83}$	CCG'	$CCG_{cluster985}$	FCG'	$FCG_{cluster474}$
Nodes	9	5	107	8	301	74
Edges	28	8	247	12	331	73
Entry point	2	1	4	1	65	1
Exit point	2	1	30	1	205	71
Paths	44	5	622	10	383	71

Table 8
Time cost for each step in second.

Software	PHNotepad	SweetHome3D		
Process/Level	*FCG*	*PCG*	*CCG'*	*FCG'*
Call graph	0.026	0.024	0.170	0.291
Execution paths	0.002	0.002	0.342	0.854
One hot encoded	0.001	0.012	0.474	0.340
Hierarchical clustering	0.686	2.171	10.412	6.119
Generate subgraph	0.189	0.036	0.009	0.077
Total	0.904	2.245	11.407	7.681

5.3. Execution time overhead

During the previous tasks, we measured the time cost for each process that we applied. Our main goal here is to measure whether the amount of time needed to run CodEx is affordable given the outputs it produces. As shown in Table 8, "Hierarchical Clustering" represents the vast majority of the processing overhead at each level. This is due to the high computation complexity of AHC. By examining the results of PHNotepad shown in Table 8, we observe that the entire processing time is around 0.9 s. The longest sub-process time, 0.686 s, representing 75.8% of the entire processing time, was spent on the hierarchical clustering process. As for the second most costly process, generate sub-graphs, we generate 63 clusters. However, the cost would be much lower if a specific cluster is selected to investigate, similar to what we did in the second example.

For the second example, the time cost of the overall process is around 21 s. Up to 54% of all levels belong to CCG'. This is due to the large number of execution paths, which led to an increase in the hierarchical clustering process overhead. Nevertheless, overall, the time required to run the tool analysis processes, end-to-end, can be considered short for practical tasks.

It is important to mention here that using a machine learning platform that supports GPU computations (e.g., PyTorch) could drastically reduce the execution time overhead.

5.4. Discussion

In this section, we discuss our findings of the aforementioned two examples. Specifically, we examined if our approach generates meaningful hierarchical clusters with different levels of granularity. As shown in both examples, CodEx was able to construct meaningful hierarchical structures and present the comprehensible functionality units at multiple levels of granularity.

In Example I, we observed that the call graph of the software system could be partitioned into five graphs, i.e., Cluster 31, 35, 41, 46, and 58, using CH score. These graphs represent the high-level functionality of the software, refer to Fig. 8. Moreover, Cluster 35 of Example I presents the logic of the search feature in the software. Further investigations show that its sub-clusters describe all of its functionalities, including *find, findNext, replace, and replaceAll.*

In Example II, we start by analyzing and clustering execution paths of the most abstract level, Package Call Graph (PCG). Then, we focus on an interesting high-level cluster representing the desktop version of the software and map it to lower levels (i.e., Class Call Graph (CCG), Function Call Graph (FCG)). In this way, we were able to: (1) remove irrelevant information, (2) stay focused on a certain domain, and (3) reduce computation time due to the smaller call graph. This will reduce the cognitive effort to understand the call graph of a software system.

Table 9
Description of the tasks for the user study.

ID	Task description
T1	Name two packages that have a high fan-out (outgoing calls) with no fan-in (no incoming calls)
T2	Name two packages that have a high fan-in (incoming calls) with no fan-out (no outgoing calls). **Rationale.** Analysis of dependencies between entities and assessing the quality of system design are essential to conduct visualization and to assist in software comprehension.
T3	Find any interesting path in the package level and project it to class level **Rationale.** Investigating the internal structure of an artifact/s is a typical comprehension task.
T4	Removing the class 'Transformation' in the 'model' package, - How many classes will be affected? - Name all affected class and which package they belong to **Rationale.** Impact analysis allows us to estimate how much of an impact such a code change would have on the system. It can also help estimate the effort that needs to be made to make such changes.
T5	The source code of SweetHome3D comes with two versions (applet and desktop version), - Find all functions that are used in applet versions - Find all functions that are shared between these two versions **Rationale.** Investigating the functionality of (a part of) the system and understanding its role on the software is one of the main and useful activities in software comprehension for engineers and researchers.

Table 10
User study questions and their types.

#	Question	Type
Q1	The tool's interface is intuitive and user-friendly	Likert scale
Q2	I would recommend software developers to use this tool	Likert scale
Q3	I would prefer to manually inspect the codebase rather than using this tool	Likert scale
Q4	I believe that using this tool can save time and efforts in inspecting the codebase of a given system	Likert scale
Q5	The task of identifying a feature that is implemented over multiple functions in the codebase is faster to achieve using this tool rather than manually inspecting the code	Likert scale
Q6	Filtering by type, searching, retrieving the information, code coloring, and shapes of the nodes were helpful during program comprehension tasks	Likert scale
Q7	The generated clustered graphs are beneficial to identify the functionality and structure of a given system – visually – without having to inspect the source code	Likert scale
Q8	The multi-level call graphs visualization is beneficial to understand the overall system structure from different views (functions, classes, and packages).	Likert scale
Q9	The tool is useful for clustering and visually exploration the execution paths of a system in both views (i.e., call graph and hierarchical view)	Likert scale
Q10	Overall, the visualization tool is useful to understand the software structure	Likert scale
Q11	Please list any reasons for your answer to the previous question	Open-Ended
Q12	Would you like to add other comments? Limitations? Suggestions?	Open-Ended
Q13	I think that I would like to use this system frequently	Likert scale
Q14	I found the system unnecessarily complex	Likert scale
Q15	I thought the system was easy to use	Likert scale
Q16	I think that I would need the support of a technical person to be able to use this system	Likert scale
Q17	I found the various functions in this system were well integrated	Likert scale
Q18	I thought there was too much inconsistency in this system	Likert scale
Q19	I would imagine that most people would learn to use this system very quickly	Likert scale
Q20	I found the system very cumbersome/awkward to use	Likert scale
Q21	I felt very confident using the system	Likert scale
Q22	I needed to learn a lot of things before I could get going with this system	Likert scale

6. User study

A basic objective of carrying out user studies is to seek insight into how a specific technique is useful (Kosara et al., 2003). Similarly, our user study aims to gain insight into how useful our approach and tool are based on user experience feedback. Moreover, it can help us assess the strengths and weaknesses of our visualization tool and can guide future efforts to improve existing techniques.

In this section, we evaluate the usefulness and usability of our approach and tool using a user study. The user study involved 18 participants who carried out a set of tasks and then answered a questionnaire. The questionnaire included 12 questions to assess the usefulness of the tool and 10 questions to assess its usability using the System Usability Scale (SUS) (Brooke, 1996).

6.1. Participants

The user study was conducted by inviting software engineers with at least three years of experience to test our tool and then answer the survey. We reached out to more than 25 software engineers. Only 18 opted to participate in the study. 13 out of 18 participants mentioned their industries. The participants belong to more than 11 international industries worldwide, including Google, Apple, and others. Our participants were experienced (39% more than 5 years; 61% 3 to 5 years).

6.2. Procedure

To make the tool accessible to all participants, we deployed it on a Linux virtual machine (VM) running Ubuntu 18.04 LTS on Azure.

Before the participants started the study, they had to complete several small tasks. First, they had to sign a consent form. Second, they were asked to answer some questions to gather demographic information, such as work and level of experience, and assess their views on the research area. Third, the participants were asked to watch a short video clip that provides a brief overview of our visualization tool.

To evaluate the tool in this study, participants were provided with a set of software comprehension tasks and a questionnaire.

The tasks were intended to collect data about the usefulness and the usability of the tool. The given tasks are listed in Table 9. The subject study was the SweetHome3D application (refer to Section 5.2 for more information on this system). The participants then completed the questionnaire while using the tool. The responses of the participants were collected using Google Forms.

6.3. Tasks

Each participant was asked to perform a set of tasks using our tool. When we designed our tasks, we kept two main goals at the core of the study: (1) The tasks should be representative software comprehension tasks, and (2) they should exercise all of the tool's features. To achieve the first goal, we designed our comprehension tasks based on a common comprehension framework from Pacione et al. (2004). They studied several sets of tasks used in comprehension evaluation literature and software visualization. Our tasks cover most comprehension activities in Pacione's framework. The uncovered activities are mainly concerned with dynamic aspects that are not within the scope of this paper. For the second goal, our tasks covered the major features of the tool, including searching, investigating, projecting to lower-level, and extracting a cluster as a call graph.

Table 9 shows the user study tasks with rationales. The first two tasks asked the participants to determine the entry and exit points of the current graph using the tool's filtering and search features. In the third task, the participants were asked to analyze and investigate the resulted clusters and project the path or cluster of interest using the projection feature. In the fourth task, the participants were asked to explore the information associated with a specific node. In the fifth task, the participants were asked to explore part of the system using a top-down approach supported by the tool.

6.4. Questionnaire

Our questionnaire consists of three types of questions including: (1) Likert scale questions to evaluate the usefulness of the tool's features at different visualization views, (2) open ended questions to gain feedback on the design of the tool, and (3) System Usability Score (SUS) based questions, to evaluate the usability of the tool. The list of questions and their types are listed in Table 10.

6.5. Results and discussion

We first report the results of usefulness questions, Q1 to Q10, respectively, and then discuss the strengths and weaknesses of the tool from user feedback from Q11 and Q12. Finally, we report and discuss the results of the usability score.

6.5.1. Usefulness

To evaluate the usefulness of the tool, the participants were asked to perform a set of tasks and answer a survey with the following ten questions and to express their opinions accordingly. The results are illustrated using a Likert scale in Fig. 12.

Q1: This question asks participants about their opinion on the tool's interface. More than half of the participants, 12 out of 18, agree and strongly agree that the CodEx interface is intuitive and user-friendly.

Q2: We asked participants whether they would recommend CodEx to other developers. The responses were positive, and the vast majority agree with this statement, which represents 94.4% of the participants.

Q3: The goal of this question was to assess whether experienced developers (all of our participants have more than three years of experience using Java) would prefer manual inspection of the code over an automated tool that visualizes the code structure using call graphs. The results show that only one participant strongly agrees with this statement, 4 participants agree, while five disagree and one strongly disagrees.

While five participants illustrated that they would prefer to inspect the code base manually rather than using our tool, upon further investigation and discussions with the participants, we discovered a discrepancy between our question and what participants understood from the question. The participants referred to the fact that they would prefer to inspect the source code directly for debugging and maintenance tasks, which does not contradict with our tool. Our goal of the question was targeted towards the cognitive tasks of understanding the system's structure, but not the actual debugging task. Overall, low-level maintenance tools and high-level analysis tools (ours) are complementary in nature for supporting the understanding of software systems (Wettel, 2010). The answers from Q4 actually support this claim, as we explain in the following.

Q4: In this question, we aimed to assess the execution overhead of using CodEx. 16 out of 18 participants agree and strongly agree that using CodEx could save them time and effort while carrying out the task of program comprehension. These findings strongly align with our execution time analysis that was discussed with the illustrative examples above.

Q5: This question focused on the specific task of identifying the functionality of a system under investigation. We asked the developers to compare carrying out this task using CodEx versus manual inspection. The results show that more than half, 61.1% prefer to inspect the system using CodEx. Thus, CodEx is useful to identify a feature in the system and all the underlying functions that implement it.

Q6: After asking the participants to explore different features of the system, we asked them whether or not these features were helpful for program comprehension tasks. We noticed that the majority, 14 out of 18, agree with this statement.

Q7: This specific question reflects the participants' opinion on the usability of generating and visualizing clusters without looking at the source code. Specifically, 13 out of 18 participants agree and strongly agree with this statement. Only two participants disagree with the statement.

Q8: In this question, we asked the participants if the multi-level call graphs are helpful to visually understand the system structure from the views of packages, classes, and functions. As expected, multi-level visualizing of the call graphs seems to be one of the best features that facilitate overall program comprehension. 10 participants agree, 6 participants strongly agree, and only 2 participants are neutral about this statement. None of the responses digressed with this statement.

Q9: In this question, we aimed to assess the overall satisfaction of the two main contributions of our tool, i.e., multi-level and hierarchical visualizations of the call graphs. The responses to this statement included the following: 1 disagree, 2 neutral, 11 agree, and 4 strongly agree. Overall, about 80% of the participants agree and strongly agree that multi-level and hierarchical visualizations of the call graphs are useful to understand the overall structure of a software system.

Q10: In this question, the participants were asked to rate the overall usefulness of CodEx. The resulting scores show that 88.8% of the participants agreed, while 11% have neutral responses.

6.5.2. Participants' feedback

The questionnaire contains two open-end questions. The participants were asked to evaluate their overall opinion regarding the tool, provide us feedback that could help improve the tool and share valuable insights into the difficulties encountered during

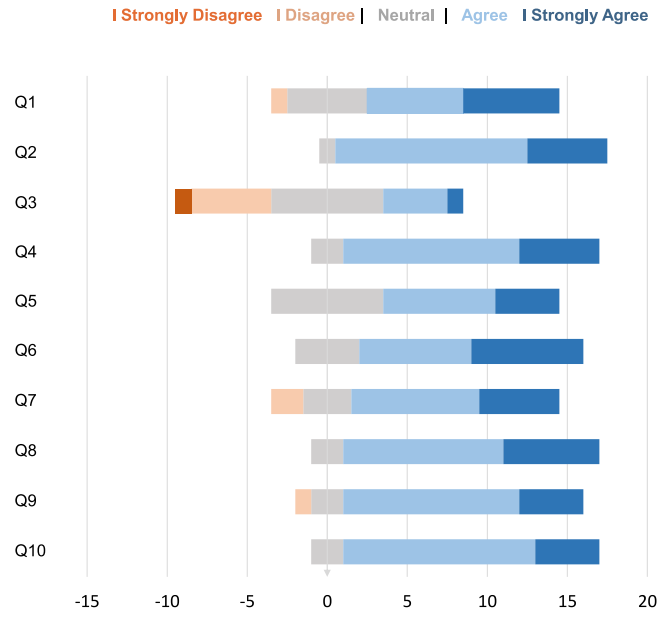


Fig. 12. Likert scale representation of the survey responses. Number of questions: 10. Number of participants: 18.

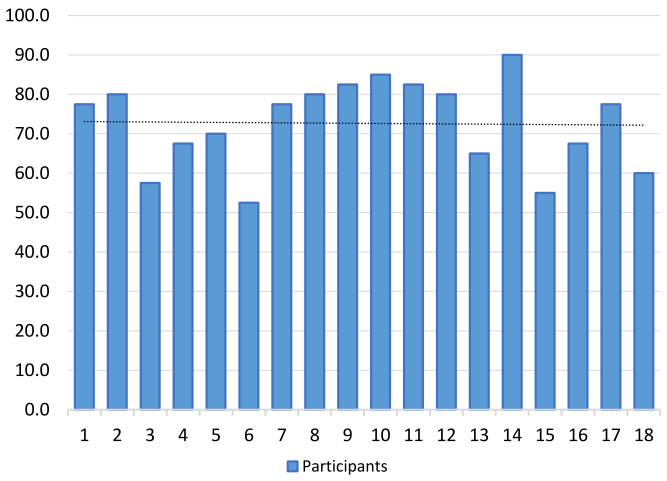


Fig. 13. The System Usability Score (SUS).

the analysis. We summarize and highlight participants' feedback as follows:

Graph Filtering. Some participants thought that it was challenging to explore the function call graph. One participant commented: "The visualization seems slow and hangs when there are too many nodes and edges". It is one of the major drawbacks of node-link representation. It tends to become highly cluttered when large numbers of nodes and edges are visualized. Several participants suggested an improvement by providing features to filter/hide irrelevant nodes and edges, which allow the graph to be more manageable and easy to explore. Some participants suggest that the tool may be more selectively showing only execution paths containing a particular node.

Lack of Information and Customization. We used some basic software metrics such as in-degree, out-degree to help identify components that, for instance, need to be refactored. However, two participants suggested including more structural data. One participant commented: "It would be good to know how many classes there are in a package". Some participants suggested "providing more software metrics data, such as LOC, WMC, and there should be ways to query and sort, such as by color or size". Another suggestion was made regarding the ability to customize colors or styles of graphs. One participant suggested adding a brief description that appeared when hovering over an icon. Another participant suggested an enrichment tool available as a plugin for the popular IDEs, including IntelliJ and VS.Code.

6.5.3. Usability

To analyze the usability of the tool, we considered The System Usability Score (SUS). Fig. 13 shows the average value for the System Usability Scores, which is 72.6. According to a previous study (Lewis and Sauro, 2018), this score is above the average 68, which can be interpreted as Good. Although the tool has a 'Good' average, the minimum score is 52 among other similar studies.

6.5.4. Threats to validity

Although we justify the rationale of each program comprehension task, the choice of tasks may bias the results in favor or against our proposed approach. To minimize this threat we choose the tasks inspired by Pacione et al. (2004) work. They studied several sets of tasks used in comprehension evaluation literature and software visualization. We designed our tasks to cover most comprehension activities based on Pacione's framework. The uncovered activities are mainly concerned with dynamic aspects that we do not consider in our study.

Another possible threat to validity is that the number of subjects in the study is less than usual—only one subject system. However, we argue that we used a system that is well-curated and well-documented of manageable size, which we believe is a good representative of real-world software systems. Moreover, a primary concern was to validate the usefulness of the tool. Therefore, we needed to be able to check the subject system manually. With our tool implemented and ready to be used, one can conduct several other use cases, and it is made our future work.

7. Related work

Call Graph Visualization: Several researchers have proposed different visualization tools to generate and visualize call graph of a system (Alnabhan et al., 2018; Jin et al., 2019; Gharibi et al., 2018b; Walunj et al., 2019). However, these tools use Gansner and Ellson (2017) for drawing graphs. For example, Alnabhan et al. (2018) proposed a 2D software visualization approach. They used geometric forms to represent different entities of the source code. For example, classes and methods are described as rectangles and circles, respectively, and arrows represent relationships between the methods.

Other interesting works exist in this area, including interactive visualization for the call graph. Shah and Guyer (2016) propose an interactive call-graph visualization tool for programs written in Java or C++. Their tool represents the call graph in a grid of pixels where each pixel or cell represents a function.

REACHER (LaToza and Myers, 2011) is interactive visualization tool implemented as eclipse plugin. It helps developers explore static call graph paths of the specific method instead of manually traversing calls to understand the control flow.

Another tool that is very similar to ours is introduced by Lemos et al. (2013). They proposed SysGraph4AJ (MultiLevel System Graphs for AspectJ). The tool supports visualization of the system's structure and structural testing at the unit level. Although SysGraph4AJ allows user to explore the system in multi-level views, only dependency among the methods is supported.

Bohnet and Döllner (2006) combine the static structure and dynamic analysis properties in a 3D landscape view. The tool

extracts dynamic call graph information and allows the developer to navigate and gain insight into how features are implemented. The dynamic call graph would require the developer to know what input data to provide and execute it. Moreover, obtaining all possible execution paths of the system is very hard. In contrast, Our work focuses on the static call graph of the software system. The static call graph considers all possible execution paths of the system.

These tools are promising but limited in several aspects. (1) Limited to a single level of abstraction (i.e., function level) and thus may not well support different maintenance tasks requiring understanding either fine or coarse grain, or both levels. (2) Visualize portion of the call graph and does not depict the overall structure of the system. (3) Display all system entities, including classes, methods, and attributes, is likely to lead to an information overload and fail to understand the software. (4) Generated call graph represented in a static format such PNG and DOT format, which requires more effort to understand.

Execution Paths Reduction Techniques: The massive size of the execution paths causes scalability issues. To improve the comprehensibility of execution paths, researchers have proposed many reductions and compression techniques to reduce the size. Hamou-Lhadj et al. (2005) used filtering techniques to reduce the number of execution paths, which resulted in smaller call graphs and higher abstraction levels of the system. The filtering approach filters out the utility components from the execution paths. Similarly, Cornelissen et al. (2007, 2009) focused on reducing the graph size by reducing the stack depth of the execution path. While both approaches aim at reducing the call graph size, they may omit essential execution paths from the call graph. In contrast, our approach considers every execution path in the system and focuses on clustering them into several hierarchies to reduce the overall graph size.

Another research direction is to compress and reduce the size of the execution paths. For example, Chan et al. (2003) reduced the size of the execution paths using sampling techniques and provided a tool to sample, visualize, and animate the dynamic traces of the system. Reiss and Renieris (2001) on the other hand, focused on reducing the size of the execution paths by encoding the repeated ones. The authors used a comparative approach to find similar paths in the system and encode them together, which resulted in reducing the overall size of the graph. While these approaches help reduce the overall size of the execution paths and ultimately the graph, they still provide a single level of granularity only. In contrast, we focus on clustering the execution paths over multiple levels of abstractions. Not only does this approach reduce the size of the graph at each hierarchical level, but it can also help developers to better understand the software system with multiple hierarchical levels of granularity.

Another work that is very similar to our approach is the Sage tool (Feng et al., 2018). Sage is a dynamic analysis approach that focuses on building hierarchical abstractions from function calls and can label each hierarchical abstraction with a proper functionality name. Our work focuses on the static analysis of the software system. The dynamic analysis can represent a single execution of the software system based on the input, while the static analysis considers all possible execution paths of the system. Moreover, we focus on building hierarchical abstractions from different abstraction levels of the system. Sage visualizes the identified clusters in vertically stacked layers, where users may lose the position of the visible layer due to the lack of a global overview. CodEx provides a hierarchical view as well as a mechanism to link clusters between different levels of the system call graph to facilitate exploring and investigating the execution paths of a system.

Hierarchical Clustering: Software clustering is one of the commonly used techniques for program comprehension. Several syntactic-based clustering techniques have been proposed Mitchell and Mancoridis (2006), Islam et al. (2014). These clustering researches are based on static structural dependencies (e.g., inheritance, method calls, references). Other clustering approaches rely on semantic clustering (Sun et al., 2017; Kuhn et al., 2007; Santos et al., 2014), which group the source code in terms of identifier names and comments. Unlike previous software clustering techniques, our approach uses execution paths to guide the clustering of source code entities that perform similar functionality. Execution paths can also reflect the overall system workflow (Jin et al., 2018).

8. Conclusions and future work

In this paper, we presented an automated data-driven approach to support the comprehension tasks of software systems, equipped with an interactive visualization tool. Our primary research contribution is twofold: First, we designed the abstraction method to bridge the cognitive gap between the high-level functionality of a software system and its low-level implementation. Specifically, the abstraction method based on the call graph of a software system was developed by creating a multi-level, hierarchical three-tier abstraction of software with different granularity levels (e.g., functions, classes, and packages). Second, we developed an interactive visualization tool that allows developers to understand large software systems through the call graphs' abstraction. The tool will enable visual inspection, tracing, and exploration of system functionality at different granularity levels.

We presented a case study to exemplify the benefits of automatic abstraction and visualization for software comprehension. Our initial evaluation allowed us to assess the tool's usefulness and usability using a user study assessment. However, there is still room to enhance our visualization tool.

For large systems, the visualization could be further improved by using the package hierarchy to coarsen the graph to more than the three levels. Moreover, extracting and calculating more software metrics and utilizing the size, shape, and color of a node or edge to reflect these matrices to give the developer more insight into the complexity of the software. Also, a more rigorous validation will be performed as part of our future work. We also plan to use GPU platforms to further reduce the execution time spent on the clustering task.

CRediT authorship contribution statement

Rakan Alanazi: Methodology, Conceptualization, Visualization, Software, Writing - reviewing. **Gharib Gharibi:** Conceptualization, Writing - reviewing. **Yugyung Lee:** Supervision, Writing - reviewing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the industry participants who helped in conducting the user study and evaluate the software system. We also thank the anonymous reviewers for their time and effort in reviewing this work. The first author would like to thank Northern Border University for the scholarship. The coauthor, Yugyung Lee, would like to acknowledge the partial support of the NSF, USA Grant No. 1747751, 1935076, and 1951971.

References

Alanazi, R., 2021. CodEx visual clustering-based exploration tool. https://rakanalanazi.github.io/CodEx/.

Alnabhan, M., Hammouri, A., Hammad, M., Atoum, M., Al-Thnebat, O., 2018. 2D visualization for object-oriented software systems. In: 2018 International Conference on Intelligent Systems and Computer Vision (ISCV). IEEE, pp. 1–6.

Anquetil, N., Lethbridge, T.C., 1999. Experiments with clustering as a software remodularization method. In: Sixth Working Conference on Reverse Engineering (Cat. No. PR00303). IEEE, pp. 235–255.

Bogar, A.M., Lyons, D.M., Baird, D., 2018. Lightweight call-graph construction for multilingual software analysis. arXiv preprint arXiv:1808.01213.

Bohnet, J., Döllner, J., 2006. Visual exploration of function call graphs for feature location in complex software systems. In: Proceedings of the 2006 ACM Symposium on Software Visualization, pp. 95–104.

Brooke, J., 1996. SUS: a "quick and dirty'usability. Usability Eval. Ind. 189.

Burkhardt, J.-M., Détienne, F., Wiedenbeck, S., 2002. Object-oriented program comprehension: Effect of expertise, task and phase. Empir. Softw. Eng. 7 (2), 115–156.

Chan, A., Holmes, R., Murphy, G.C., Ying, A.T., 2003. Scaling an object-oriented system execution visualizer through sampling. In: 11th IEEE International Workshop on Program Comprehension, 2003. IEEE, pp. 237–244.

Chevalier, C., Safro, I., 2009. Comparison of coarsening schemes for multilevel graph partitioning. In: International Conference on Learning and Intelligent Optimization. Springer, pp. 191–205.

Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., Van Wijk, J.J., Van Deursen, A., 2007. Understanding execution traces using massive sequence and circular bundle views. In: 15th IEEE International Conference on Program Comprehension (ICPC'07). IEEE, pp. 49–58.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. Softw. Eng. 35 (5), 684–702.

Davey, J., Burd, E., 2000. Evaluating the suitability of data clustering for software remodularisation. In: Proceedings Seventh Working Conference on Reverse Engineering. IEEE, pp. 268–276.

Feng, Y., Dreef, K., Jones, J.A., van Deursen, A., 2018. Hierarchical abstraction of execution traces for program comprehension. In: Proceedings of the 26th Conference on Program Comprehension. ACM, pp. 86–96.

Gansner, E., Ellson, J., 2017. Graphviz-graph visualization software. URL http://www.graphviz.org/. (Cited on pages 69, 81, and 82).

Gharibi, G., Alanazi, R., Lee, Y., 2018a. Automatic hierarchical clustering of static call graphs for program comprehension. In: 2018 IEEE International Conference on Big Data (Big Data). IEEE, pp. 4016–4025.

Gharibi, G., Tripathi, R., Lee, Y., 2018b. Code2graph: automatic generation of static call graphs for Python source code. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 880–883.

Gousios, G., 2019. java-callgraph: Java Call Graph Utilities. https://github.com/gousiosg/java-callgraph,

Graham, S.L., Kessler, P.B., Mckusick, M.K., 1982. Gprof: A call graph execution profiler. In: ACM Sigplan Notices. ACM, pp. 120–126.

Grinberg, M., 2018. Flask web development: developing web applications with python. " O'Reilly Media, Inc.".

Grove, D., DeFouw, G., Dean, J., Chambers, C., 1997. Call graph construction in object-oriented languages. ACM SIGPLAN Notices 32 (10), 108–124.

Hagberg, A., Swart, P., S. Chult, D., 2008. Exploring Network Structure, Dynamics, and Function Using NetworkX. Technical Report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Hamou-Lhadj, A., Braun, E., Amyot, D., Lethbridge, T., 2005. Recovering behavioral design models from execution traces. In: Ninth European Conference on Software Maintenance and Reengineering. IEEE, pp. 112–121.

Hoogendorp, H., 2010. Extraction and Visual Exploration of Call Graphs for Large Software Systems (Ph.D. thesis). Faculty of Science and Engineering.

Hopcroft, J., Tarjan, R., 1973. Algorithm 447: efficient algorithms for graph manipulation. Commun. ACM 16 (6), 372–378.

Horner, E., 2021. Constructing Call Graphs. URL https://github.com/davidfraser/pyan.

Islam, S., Krinke, J., Binkley, D., Harman, M., 2014. Coherent clusters in source code. J. Syst. Softw. 88, 1–24.

Jin, W., Cai, Y., Kazman, R., Zheng, Q., Cui, D., Liu, T., 2019. ENRE: a tool framework for extensible entity relation extraction. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings. IEEE Press, pp. 67–70.

Jin, W., Liu, T., Zheng, Q., Cui, D., Cai, Y., 2018. Functionality-oriented microservice extraction based on execution trace clustering. In: 2018 IEEE International Conference on Web Services (ICWS). IEEE, pp. 211–218.

Kosara, R., Healey, C.G., Interrante, V., Laidlaw, D.H., Ware, C., 2003. Thoughts on user studies: Why, how, and when. IEEE Comput. Graph. Appl. 23 (4), 20–25.

Kuhn, A., Ducasse, S., Gîrba, T., 2007. Semantic clustering: Identifying topics in source code. Inf. Softw. Technol. 49 (3), 230–243.

LaToza, T.D., Myers, B.A., 2011. Visualizing call graphs. In: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, pp. 117–124.

Lemos, O.A.L., Zanichelli, F.C., Rigatto, R., Ferrari, F., Ghosh, S., 2013. Visualization, analysis, and testing of Java and Aspectj programs with multi-level system graphs. In: 2013 27th Brazilian Symposium on Software Engineering. Ieee, pp. 49–58.

Lewis, J.R., Sauro, J., 2018. Item benchmarks for the system usability scale. J. Usability Stud. 13 (3).

Maalej, W., Tiarks, R., Roehm, T., Koschke, R., 2014. On the comprehension of program comprehension. ACM Trans. Softw. Eng. Methodol. (TOSEM) 23 (4), 31.

Maqbool, O., Babri, H.A., 2004. The weighted combined algorithm: A linkage algorithm for software clustering. In: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.. IEEE, pp. 15–24.

Maqbool, O., Babri, H., 2007. Hierarchical clustering for software architecture recovery. IEEE Trans. Softw. Eng. 33 (11), 759–780.

Mark Otto, J.T., 2019. Bootstrap. https://getbootstrap.com/,

Merino, L., Ghafari, M., Nierstrasz, O., 2018. Towards actionable visualization for software developers. J. Softw.: Evol. Process 30 (2), e1923.

Mitchell, B.S., Mancoridis, S., 2006. On the automatic modularization of software systems using the bunch tool. IEEE Trans. Softw. Eng. 32 (3), 193–208.

Mkaouer, M.W., Kessentini, M., Bechikh, S., Cinnéide, M.Ó., Deb, K., 2016. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empir. Softw. Eng. 21 (6), 2503–2545.

Murphy, G.C., Notkin, D., Griswold, W.G., Lan, E.S., 1998. An empirical study of static call graph extractors. ACM Trans. Softw. Eng. Methodol. (TOSEM) 7 (2), 158–191.

Naeimian, A., 2019. Parallel Paths Analysis Using Function Call Graphs (Master's thesis). University of Waterloo.

Naseem, R., Maqbool, O., Muhammad, S., 2013. Cooperative clustering for software modularization. J. Syst. Softw. 86 (8), 2045–2062.

Oliphant, T.E., 2007. Python for scientific computing. Comput. Sci. Eng. 90 (9), 10–20.

Pacione, M.J., Roper, M., Wood, M., 2004. A novel software visualisation model to support software comprehension. In: 11th Working Conference on Reverse Engineering. IEEE, pp. 70–79.

Reiss, S.P., Renieris, M., 2001. Encoding program executions. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001. IEEE, pp. 221–230.

Ryder, B.G., 1979. Constructing the call graph of a program. IEEE Trans. Softw. Eng. (3), 216–226.

Salis, V., Sotiropoulos, T., Louridas, P., Spinellis, D., Mitropoulos, D., 2020. PyCG: Practical call graph generation in Python. Nature 15, 16.

Sander, J., Qin, X., Lu, Z., Niu, N., Kovarsky, A., 2003. Automatic extraction of clusters from hierarchical clustering representations. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer, pp. 75–87.

Santos, G., Valente, M.T., Anquetil, N., 2014. Remodularization analysis using semantic clustering. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). IEEE, pp. 224–233.

Shah, M.D., Guyer, S.Z., 2016. An interactive microarray call-graph visualization. In: 2016 IEEE Working Conference on Software Visualization (VISSOFT). IEEE, pp. 86–90.

Sun, X., Liu, X., Li, B., Li, B., Lo, D., Liao, L., 2017. Clustering classes in packages for program comprehension. Sci. Program. 2017.

Tunali, V., Tüysüz, M.A.A., 2020. Analysis of function-call graphs of open-source software systems using complex network analysis. Pamukkale Üniversitesi Mühendislik Bilimleri Dergisi 26 (2), 352–358.

Walunj, V., Gharibi, G., Ho, D.H., Lee, Y., 2019. GraphEvo: Characterizing and understanding software evolution using call graphs. In: 2019 IEEE International Conference on Big Data (Big Data). IEEE, pp. 4799–4807.

Wettel, R., 2010. Software Systems as Cities (Ph.D. thesis). Università della Svizzera italiana.

Wiggerts, T.A., 1997. Using clustering algorithms in legacy systems remodularization. In: Proceedings of the Fourth Working Conference on Reverse Engineering. IEEE, pp. 33–43.