ABSTRACT

VIZSLICE: AN APPROACH FOR UNDERSTANDING SLICING DATA VIA VISUALIZATION

by Rachel Ania Kaczka Jennings

Several approaches have been suggested for computing program slices based on different perspectives, including forward slicing, backward slicing, static slicing, and dynamic slicing. The applications of slicing are numerous, including testing, effort estimation, and impact analysis. Surprisingly, given the maturity of slicing, few approaches exist for visualizing slices. Here we present our research for visualizing large systems based on program slicing. In particular, we use treemaps to facilitate hierarchical, slicing-based navigation, we use bipartite graphs to facilitate visual impact analysis over a given variable or line of code, parallel coordinates to facilitate visual impact analysis over code blocks or variable groupings, and a text-based code browser to provide detailed context for the relevant visualizations. We believe our tools support various software maintenance tasks, including providing analysts an interactive visualization of the impact of potential changes, thus allowing developers to plan maintenance accordingly. We evaluate the research by assessing usability through a think aloud protocol and a heuristic evaluation. Our results indicate users could effectively complete the evaluation tasks we provided, and the visual idioms utilized in vizSlice were effective at communicating the underlying data to them. However, controls for these visualizations need improvement in both affordance and visibility. Regardless of any difficulties users experienced with vizSlice, users consistently rated the system positively on the measured heuristics. We provide insights on these results, future plans for improving vizSlice, and provide guidance for future research on visualizing program slices.
VIZSLICE: AN APPROACH FOR UNDERSTANDING SLICING DATA VIA VISUALIZATION

Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Science in Computer Science
by
Rachel Ania Kaczka Jennings
Miami University
Oxford, Ohio
2017

Advisor: Dr. Gerald Gannod
Reader: Dr. Hakam Alomari
Reader: Dr. Matthew Stephan
Reader: Dr. James Kiper

©2017 Rachel Ania Kaczka Jennings
This thesis titled

VIZSLICE: AN APPROACH FOR UNDERSTANDING SLICING DATA VIA VISUALIZATION

by

Rachel Ania Kaczka Jennings

has been approved for publication by

College of Engineering and Computing

and

Department of Computer Science and Software Engineering

____________________________

Dr. Gerald Gannod

____________________________

Dr. Hakam Alomari

____________________________

Dr. Matthew Stephan

____________________________

Dr. James Kiper
# Contents

1 Introduction 1

2 Background and Related Work 4

2.1 Slicing 4

2.1.1 Analysis Type 4

2.1.2 Direction 6

2.2 Program Visualization 7

2.3 Slicing Visualization 8

2.4 D3 9

2.5 Evaluation Techniques 10

3 Approach 11

3.1 Research Questions 11

3.2 Implementation 11

3.3 Evaluation 13

4 Visualizations 14

4.1 Treemap 15

4.2 Bipartite Graph 18

4.3 Parallel Coordinates 21

4.4 Code View 25
List of Tables

6.1 Raw data from the student think aloud protocols .......................... 41
6.2 Raw data from the professional think aloud protocols .................. 41
6.3 Averages from the think aloud protocol by task number .............. 42
6.4 Averages from the think aloud protocol by task groupings .......... 42
6.5 Raw data from the student heuristic evaluations ....................... 45
6.6 Raw data from the professional heuristic evaluations ................. 45
6.7 The calculated medians by question number ............................. 45
6.8 The calculated medians by question groupings ......................... 45
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The flow of data in <strong>vizSlice</strong></td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>An example of a SeeSoft style visualization</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of the Constellation Visualization</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Screen capture of SeeSlice being used to visualize a Slice.</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>The <strong>vizSlice</strong> pipeline</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Example source data from srcSlice</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>A treemap representation of the Radius system</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>An example bipartite graph - full graph</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>An example bipartite graph with the variable <code>greq</code> selected</td>
<td>21</td>
</tr>
<tr>
<td>4.5</td>
<td>An example bipartite graph with the line number 65923 selected</td>
<td>21</td>
</tr>
<tr>
<td>4.6</td>
<td>A parallel coordinates visualization of car data [40]</td>
<td>22</td>
</tr>
<tr>
<td>4.7</td>
<td>An parallel coordinates visualization using brushing [40]</td>
<td>23</td>
</tr>
<tr>
<td>4.8</td>
<td>Parallel coordinates with brushing in <strong>vizSlice</strong></td>
<td>24</td>
</tr>
<tr>
<td>4.9</td>
<td>An example of the code view in <strong>vizSlice</strong> with highlighting after a selection in the bipartite graph</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>A function being displayed in <strong>vizSlice</strong></td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>The results of rubric evaluations on the student groups</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>The results of rubric evaluations on the professionals</td>
<td>42</td>
</tr>
<tr>
<td>6.3</td>
<td>The average results of rubric evaluations by task number</td>
<td>43</td>
</tr>
</tbody>
</table>
6.4 The average results of rubric evaluations by task groupings .......................... 43

6.5 The medians from heuristic evaluation by question number ..................... 46

6.6 The medians from heuristic evaluation by question groupings ............... 47
List of Listings

4.1 Example input to srcML .............................................. 25
4.2 Example output of srcML ............................................. 27
Dedication

To my sisters,
April, Robbie, Shelby, and Jazzy.

My life is immeasurably better because of each of you.
Acknowledgements

Thank you to my adviser Dr. Gerald Gannod for his encouragement and support during the completion of this thesis and throughout my time at Miami University. Without his guidance, I certainly would not be where I am now. Additionally, I would like to acknowledge my committee members Dr. Matthew Stephan, Dr. Hakam Alomari, and Dr. James Kiper for their support and input.

A special thanks to Luke Pitstick and Paulo Caetano Virote de Souza for aiding me on my work directly. Luke implemented a series of scripts to connect the various portions of the vizSlice pipeline’s data inputs and outputs. Paulo primarily implemented a UI to house the visualizations created, but also aided in the integration of the code view to the function level visualizations.

Thank you to my friends, for being an encouragement and reminding me that being social is a good thing.

Thank you to my parents, especially my mother, for providing me with emotional support and encouragement. Thank you to my sisters, for believing that I am much cooler than I actually am. And thank you to my brother for being supportive of my work.

Last, though definitely not least, thank you to my loving husband for his unending encouragement and for being the sounding boarding for all of my terrible ideas.

Thank you all for believing in me, even when I may not have believed in myself. I couldn’t have done it without you.
Chapter 1

Introduction

Software maintenance and comprehension of large systems is difficult due to increasing complexities that stem from dependencies in code. As systems become larger, it becomes more difficult to identify these dependencies[1]. There have been attempts at automating static analyses of larger systems[2, 3]. But static analysis is not without its flaws. For example, Wheeler claims that the Heartbleed vulnerability could have been discovered with a more thorough human review[4] and not solely automated processes. Similarly, bugs such as Apple’s ‘goto fail’ bug could have been detected with a more thorough human analysis[5]. Program slicing allows users to decompose programs into smaller pieces, allowing them to filter out unrelated information to more easily determine the factors that impact risk and failure[6]. Tools utilizing slicing can aid human comprehension and analysis, providing for a more thorough and efficient human analysis.

Program Slicing, as defined by Weiser, is a method of automatically decomposing programs by analyzing a program’s data and control flow, and is similar to mental abstractions made by users when performing maintenance and debugging tasks[6]. Program Slicing is a mature field and is used for various software maintenance tasks such as impact analysis, debugging, and comprehension[1, 7–11]. Slicing is characterized using two aspects: direction and analysis type. The direction of a slice can either be forward or backward: a forward slice determines which components the slicing point effects, while a backward slice determines what components effect the slicing point. A slicing analysis can be static or dynamic: static slicing analyzes only the source
code, while *dynamic slicing* analyzes the source code in the context of a specific input, ignoring code that does not impact the slicing point on that input. While program slicing can be useful, it generates a large amount of complex data, which makes analysis difficult\[12\]. While computing slices on large systems can be computationally expensive, there have been some advancements in static, forward slicing\[12, 13\]. However as programs grow larger, slicing data becomes more difficult to comprehend.

There are several visualizations designed to represent different types of program data. While many program visualization techniques exist, few visualizations have been developed to aid in the comprehension of program slicing. Additionally, these tools focus primarily on dynamic slicing. As an example, SeeSlice presents forward and backward dynamic slicing data in a visualization where each line is color coded based on the distance, in a dependency graph, from the slicing point\[14\]. This type of visualization preserves the code structure, allowing users to navigate code in a manner that they are accustomed to. However SeeSlice's visualization fails to present potentially useful information, such as system and subsystem complexity. Additionally, this style of visualization has limitations in the amount of code that can be visualized, preventing the entire system from being visualized at once. Furthermore, only one slicing point can be seen at a time. Slices for multiple variables are not displayed simultaneously.

Our research involves a system for analyzing and visualizing static forward program slicing in an interactive format. In this work, we utilize srcML\[15\] and srcSlice\[13\] to collect slicing data from software systems. We then visualize the collected slicing data using a variety of visualization idioms. This process can be seen in Figure 1.1. We use a treemap to represent system complexity with subsystems, directories, files, and functions represented in each successive level of abstraction. We then use slicing data at each level to provide visual cues into the complexity of the subject software system. Additionally, each function's slicing data is represented in our system in a bipartite graph and parallel coordinates visualization. In these visualizations, both size and color are used to encode slicing metrics. While using the bipartite graph or parallel coordinates visualization, the original code is visible to the user in a style similar to editors such as Sublime Text\[16\] which displays a files contents in a primary viewing pane but also in a secondary minimized view along
the right side of the application window. These allow the user to see both the general complexity of the system and the details of each function, facilitating easier impact analysis utilizing program slicing and interactive visualization techniques.

![Diagram of data flow](image)

Figure 1.1: The flow of data in **vizSlice**

**Thesis Statement**

Using **vizSlice**, software maintainers can quickly and efficiently evaluate slicing data for use in impact analysis and system comprehension through visualization. **vizSlice** is web-based, scalable, fast, and visually appealing. This will allow for an efficient and thorough human analysis of software.
Chapter 2

Background and Related Work

This chapter describes background material for the research and relevant related work. Section 2.1 covers the concept of program slicing. Section 2.2 presents relevant work on general program visualization techniques and Section 2.3 presents work on program visualization techniques specific to visualizing program slices. Finally, Section 2.4 presents a common technology used in web-based visualizations.

2.1 Slicing

Slicing is the process of evaluating a set of program statements to determine their potential impact. Program slicing is used to aid debugging, optimization, analysis, and various other software maintenance tasks. Slicing can be primarily categorized in two aspects: analysis type and direction. Here we formally define both aspects.

2.1.1 Analysis Type

A static slicing criterion is defined as a tuple $C = \langle i, V \rangle$ where $i$ is a statement and $V$ is a subset of variables in program $P$. The slicing criterion is used to determine the projection function $\text{Proj}_C$, which discards the state trajectory for all ordered pairs except those starting with $i$ and with values of variables from $V$. A state trajectory is finite list of ordered pairs $(n_1, s_1)(n_2, s_2)\cdots(n_k, s_k)$ such
that n is in the set of nodes from program P and s is a function mapping variables in V to their values. Each pair in a state trajectory gives the value of V immediately prior to the execution of statement n. A state trajectory, defined as \( T = (t_1, t_2, ..., t_n) \), allows us to say

\[
\text{Proj}_{C}((n, s)) = \begin{cases} 
\lambda & \text{if } n \neq i \\
(n, s \mid V) & \text{if } n = i
\end{cases}
\]

where \( s \mid V \) is s restricted to domain V and \( \lambda \) is the empty string. Proj can then be extended to the entire trajectories:

\[
\text{Proj}_{C}(T) = \text{Proj}_{C}(t_1) \cdots \text{Proj}_{C}(t_n).
\]

A static slice is then defined as the subset of a program that preserves the projection of its behavior. Formally, a slice S is of program P on a slicing criterion, \( C = \langle i, V \rangle \), is any executable program that holds two properties. Firstly, S can be obtained by deleting zero or more statements from P. Secondly, whenever P halts on input I with state trajectory T, S also halts on I with state trajectory \( T' \), where

\[
\text{Proj}_{C}(T) = \text{Proj}_{C}(T_i)
\]

and where \( C_i = \langle \text{succ}(i), V \rangle \) and \( \text{succ}(i) \) is the nearest successor to i, or i itself, in the slice \( [6] \).

A dynamic slice evaluates with respect to a given trajectory on a specific input. In contrast, a static slice evaluates based on all possible trajectories through the flowgraph. Before defining a dynamic slice, we must first define a few functions. \( F(T, q) \) gives the sublist of T from position 1 up to and including position q. \( B(T, q) \) gives the sublist of T from, but not including, position q to the end of the list. \( \text{DEL}(T, r) \), where r is a predicate on the set of instructions in T, is the subtrajectory obtained from T by deleting all elements from it that satisfy r.

A dynamic slice evaluates based on some input x and therefore a dynamic slicing criterion is defined as the triple \( C = \langle x, I^q, V \rangle \), where \( I^q \) is statement I at position q on T, and V is a subset of variables in P. A dynamic slice of program P on C is then any executable, \( P_r \), that can be obtained from P by deleting zero or more statements from P and, when executed on input x, produces
trajectory $T\tau$ that has position $q\tau$ such that:

$$F(T\tau, q\tau) = DEL(F(T, q), T(i) \notin N\tau \text{ and } 1 \leq i \leq q),$$

for all $v \in V$, the value of $v$ before the execution of $T(q)$

is equal to the value of $v$ before the execution of $T\tau(q\tau)$ and

$$T\tau(q\tau) - T(q) - I$$

where $N\tau$ is a set of instructions in $Pr[10]$.

In summary, a static approach analyzes a program without regard to potential input, thus analyzing all potential branching paths within the program. Meanwhile, a dynamic approach analyzes the program with regards to a given input to the program, thus analyzing the execution path taken by the program on this input. A static analysis is useful for a variety of tasks including program comprehension and impact analysis[1,17,18], and a dynamic analysis is primarily useful for debugging tasks[7,10,11].

### 2.1.2 Direction

A slice can either be forward or backward in direction. The direction of a slice is determined by the method used to delete statements. A forward slice is constructed by deleting the statements that cannot be affected by the slicing criterion and a backwards slice is constructed by deleting that have no effect on upon the slicing criterion[19]. Forward slicing is useful for impact analysis[17,18], while backward slicing is useful in debugging tasks to identify the portions of code that impact the current point of interest within the system[6]. In summary, forward slicing evaluates which statements a slicing criterion effects and backward slicing focuses on which statements effect the slicing criterion[20]. srcSlice currently supports forward, static slicing on large systems[13]. Therefore we are interested in visualizing forward, static slicing and its use for impact analysis and system comprehension.
2.2 Program Visualization

Several visualizations have been developed for use in program visualization. Treemaps have been used to display hierarchical aspects of programs, such as folder and file structures. Treemaps utilize nested boxes to show hierarchical ordering. Additionally, color and size data encoding allows treemaps to visualize problem specific metadata [21]. *Spider SENSE* is an example of a toolkit designed for software visualization that utilizes treemaps [22].

*SeeSoft* is a scalable visualization tool that presents code as single pixel tall lines. Each line is color coded based on metadata. The metadata used varies by each use case. SeeSoft also allows users to view the actual code by selecting a portion from the visualization to view. This allows users to see a high level view of the data and view the details of the code [23]. Several tools utilize SeeSoft style visualizations for visualization of data. The *GAMMATELLA* tool utilizes a SeeSoft style visualization, as well as a treemap to display program execution data [24]. *Tarantula* is a tool developed for fault localization that utilizes a SeeSoft style visualization to display the likelihood of code to be a fault [25].

A *Constellation* visualization has been developed as an alternative to SeeSoft style visualizations. A Constellation visualization visualizes programs as a dependency graph. The edges of the graph are not shown and each node in the graph is a single point. The visualization then colors each point by various metadata, similar to a SeeSoft visualization. The visualization then adjusts nodes’ positions based on edge strengths. The result is a visualization that tends to cluster similarly color encoded nodes, which can reduce analysis time [26].
2.3 Slicing Visualization

*SeeSlice* is a tool developed to visualize program slices utilizing a SeeSoft style visualization [14]. SeeSlice is limited to visualizing 900 lines of code per column and the number of columns is limited by the screen size and caps at 25 columns. Maximizing SeeSlice’s display limits a user to at most around 25k lines of code. This means very large programs cannot be visualized in their entirety. SeeSlice does allow users to select a section of code for more detailed viewing, similar to SeeSoft. When selected, the original code is displayed and relevant lines are highlighted and metrics are color encoded [14][23].

Alternatively, the *Constellation* visualization mentioned in Section 2.2 can be used to display slicing data. This visualization takes a standard dependency graph and represents each node as a single, color coded pixel, and hides the links in the graph. This results in related nodes being grouped together while maintaining visual clarity for systems with many links between nodes. The Constellation visualization can be an improvement over the line highlighting used by SeeSlice [26].

Additionally, 3D visualizations have been used to display slicing data. Metaballs are a 3D modeling visualization based on technique developed by Blinn [27] and have shown success in modeling organic shapes and structural relationships in other fields. They have since been applied to visualize program metadata, such as slicing [28]. However these visualizations have difficulties...
While several visualization paradigms exist for program slicing, several of them can be difficult to comprehend quickly, causing design decisions to take longer and errors to be more likely. Data visualizations should allow for information to be digested quickly and accurately while providing sufficient detail to the users to make informed decisions. The more complex the system, the more likely a visualization will be required to comprehend the available data. The options explored here have difficulty scaling to large systems, limiting the usability of such visualizations.

2.4 D3

Data Driven Documents [29], commonly referred to as D3, is a visualization ‘kernel’ through which visualizations and visualization frameworks can be designed. D3 provides the ability to manipulate DOM elements through data driven operations. It works similarly to other document transformers such as CSS and JQuery. D3’s design is influenced by graphical libraries such as Protovis[29] [30].
D3 makes use of selections, operators, and data joins. Selections allow DOM elements to be queried from the document. Selections in D3 work similarly to selections used by JQuery and CSS. Operators act upon selections, allowing users to modify content. In D3, these operators are primarily data based, meaning that DOM elements are changed based on the data they represent or are bound to. Data joins bind data to DOM elements, enabling data based operators. D3 uses special operators that allow users to animate, update, and transition between new or updated data sets. These tools allows for the creation of highly interactive visualizations in D3.

D3 is also more scalable than previous tools used for web-based data visualizations, showing significant performance improvements over Flash and Protovis. Additionally, D3 provides increased functionality and allows users to create documents not previously possible in earlier technologies.

We use D3 as the visualization framework. Slicing generates large amounts of data. D3’s scalability allows us to display a significant portion of this data. Additionally, the interactive capabilities D3 provides allows us to create visualizations that users can explore and analyze.

2.5 Evaluation Techniques

We use a think aloud protocol and heuristic evaluation to evaluate vizSlice’s usability. A think aloud protocol allows us to understand a user’s thought process as they complete tasks using vizSlice. This is done by having users articulate their thought process and by asking questions while users interact with the system. However evaluators should seek to minimize the number of interruptions these questions can cause as they may bias the user’s interactions. We used a think aloud protocol to evaluate vizSlice tasks that we believe to be common software maintenance issues that can be solved with the provided visualizations.

Heuristic evaluations allow us to utilize heuristics to evaluate vizSlice from the user’s perspective. Only a few users are needed to identify the majority of problems within the system, due to diminishing returns. We conducted heuristic evaluations via surveys where users respond to a series of questions using a Likert scale. For this survey we used 8 of the 10 suggested heuristics.
Chapter 3

Approach

This chapter presents our approach to slicing visualization implemented in our framework, entitled \textit{vizSlice}. \textit{Section 3.1} covers the research questions for the evaluation of \textit{vizSlice}. \textit{Section 3.2} covers the implementation details of the \textit{vizSlice} system and \textit{Section 3.3} covers the methods of evaluation.

3.1 Research Questions

We developed \textit{vizSlice} to support a number of slicing based software maintenance tasks. To evaluate \textit{vizSlice}'s usability, we seek to answer the following questions:

1. How easy is it for users to navigate \textit{vizSlice}?

2. How well can users answer slicing based software maintenance questions correctly and efficiently with \textit{vizSlice}?

3.2 Implementation

The system automates the data collection, transformation, filtering, and visualization process for program slices. Data collection is handled by srcSlice\cite{13}, a scalable slicing tool that performs well on large systems. While other slicing tools have limits on the number of lines being processed, this approach can slice extremely large systems, such as the Linux Operating System. This provides the
ability to analyze large systems utilizing program slicing. A data source, such as a github repository, is linked to a server through a web interface to run srcSlice over the source code. srcSlice outputs slicing data into two data files: sliceML and srcML. sliceML contains the slicing data and srcML contains the entirety of the source compiled into a single xml document. Slicing data in sliceML is stored as slicing profiles. Slicing profiles are calculated for every variable in the system. A slicing profile lists a full variable path including folder structure, file, and function name. Additionally, the slicing lines for each variable are listed in its corresponding slicing profile. A **slicing line** is defined as a line in the program where the variable is being directly used. Other data that srcSlice collects includes **called functions**, **dependent variables**, and **variable aliases**. For the purposes of the **vizSlice**, we are interested only in the slicing lines.

Once the data is collected from the source code using srcSlice, it is transformed into usable JSON for interactive web visualizations. JSON is a compact key-value representation typically used in web applications. A Python parser is used to transform the data from a sliceML file into a JSON file storing a tree-like data structure of the slicing data. This makes the data usable in JavaScript for further data processing. The data from the JSON file is then formatted according to the visualization being used and the user’s subsequent interactions. The pipeline can be seen in Figure 3.1.

![Figure 3.1: The vizSlice pipeline](image)

Because these visualizations are highly dependent on user interactions, filtering and formatting vary based on which visualization is desired. In the **treemap** visualization, data filtering allows the desired level of abstraction of the system to be viewed. The visualization handles data filtering based upon user interactions. Once a function has been selected for more detailed viewing, a **bipartite graph** or **parallel coordinates graph** may be viewed. The desired data is then filtered out of the JSON tree, formatted according to the visualization desired, and displayed. Further user interactions can then filter the data further. In the **bipartite graph** visualization, data specific to a
variable or line can be displayed while hiding the remaining data. In a parallel coordinates graph, interaction allows for multiple consecutive lines of code to be selected for filtering, instead of a single line, and for multiple variables to be selected simultaneously. This allows a user to look at how a particular block of code is affected by various variables. In addition to either the bipartite graph or parallel coordinates graph, a text-based code view is also displayed. The code view allows for users to see the original source code for the selected function. Lines selected via the bipartite graph or parallel coordinates graph are highlighted in the code view as well, allowing for quick identification of the related source code.

3.3 Evaluation

We have evaluated the system through user studies focused on usability. We validated the system and visualizations for common use cases, such as impact analysis. We evaluated usability through a think aloud protocol and heuristic evaluation through user surveys. Two user groups, computer science and software engineering undergraduate students and professional developers, completed these evaluations over a 1 month period in November 2016.
Chapter 4

Visualizations

Software visualization provides a means for comprehending complexity without having to directly browse source code. Program slicing can be similarly utilized to improve software comprehension in various software maintenance tasks. Therefore, we have developed a series of visualizations in order to improve software comprehension through the use of srcML, srcSlice, and D3.

Output data from srcSlice contains multiple pieces of information. The output contains multiple slicing profiles, one for each unique variable within the system. An example slicing profile can be seen in Figure 4.1. Each slicing profile begins with the full path to the file in which a particular variable is contained. It is then followed the function name and variable name. After the variable name, each profile contains 5 lists of data. The first, def, includes all line numbers where the variable is set to a new value using an assignment operator. For instance, def{115} in the example shown in Figure 4.1 means that the variable itr was assigned a value on line 115 in the function grad_iterator_create. Similarly, use includes all the lines on which the variable is utilized in some fashion. dvars lists variables that are data dependent upon the selected variable. For example dvars{NULLitr} indicates that itr provides data to the variable NULLitr. The third data list pointers is a list of variables to which the selected variable is a pointer. In the example Figure 4.1, pointers{subst, env, prim, input, vlist, sess_list, var, fp, list, stmt, config} indicates that the variable itr is a pointer to the listed variables. The final data list cfuncs lists the functions that are called using the given variable. From the example, cfuncs{ grad_iterator_attach[1], grad_emalloc[2]}
indicates that the functions `grad_iterator_attach` and `grad_emalloc` are called with the variable `itr` as a parameter. Additionally the value given in brackets with each function indicates which parameter the variable was used for. A value of 1 indicates that it was the first parameter, while a value of 2 indicates that it was the second parameter [13].

```
radius-1.6/lib/list.c,grad_iterator_create,itr,
  def{115},use{83,86,87,88,89,119,120,121,247,249,250,254},
  dvars{NULLitr},pointers{subst,env,prim,input,vlist,sec,sess_list,
  var,fp,list,stmt,config},
  cfuncs{grad_iterator_attach{1},grad_emalloc{2}}
```

Figure 4.1: Example source data from srcSlice

We developed a Python parser to convert the slicing profiles from srcSlice to a JSON tree. The parser converts the full file and variable path into a tree structure. Each node in the tree is one level of abstraction in the path, named for that abstraction. The leaves of the tree are the variables in the profiles. Each leaf node stores additional information from the slicing profile, such as the `def` and `use` lines and the total count of `def` and `use` lines. Each level of abstraction above the variable also stores the average number of slicing lines (the union of `def` and `use`) for the levels of abstraction below it in the JSON tree. Additional information from the slicing profile, such as `dvars` and `cfuncs`, is discarded. The JSON object also includes some statistical computations for the system as a whole. This information is stored as an array in JSON. The array stores the minimum, the lower quartile, the median, the upper quartile, and the maximum of the average number of slicing lines, starting at the function level. The JSON tree object and statistical array allows us to easily create hierarchical visualizations using D3 [29].

4.1 Treemap

This section is adapted from previously published work on vizSlice[36].
Treemaps are a common visualization for hierarchical data and were initially developed as a file system visualization [21, 37]. Treemaps are used to map hierarchical information into 2D spaces and use 100% of the display area, maximizing the display areas usage. This is done by partitioning the available display space into rectangles of varying sizes based on some weight. Treemaps, as defined by Johnson [21], have four main properties: 1) the bounding box of an ancestor node must be equal to or greater than the bounding box of its descendants; 2) the bounding boxes of two nodes intersect if and only if one node is an ancestor to the other; 3) the display area of a node is proportional to its weight; and 4) the weight of an ancestor node is equal to or greater than the sum of the weights of its children. For example, this type of visualization translates well to a directory tree structure. Directories are ancestors that contain either more directories or files and file size can be used a weight. This allows a user to quickly determine the files and directories that utilize the most amount of disk space.

In our implementation, the treemap represents the various levels of abstraction present in the code. In particular, each node within the treemap represents a directory, file, function, or variable. The weight of each node represents the number of slicing lines contained in that portion of the system. Each node is also color encoded to represent how that portion of the system compares to the statistical average number of slicing lines for the entire system. This allows us to quickly view which portions are larger, in terms of slicing, and which portions are relatively more complex. Child elements are hidden until the user interacts with the visualization. A mouseover on a node within the treemap in a current view reveals the children of the given node. Clicking on the given node will “zoom” the visualization to the next level of abstraction. If a variable is selected, a new function level visualization opens. Figure 4.2 shows an example treemap of the Radius system[38] at the top level with a hover effect on the radiusd directory. The name of the system is listed in the top bar that spans the entire width of the visualization. The large bounding boxes are child elements of the parent node within the system. Though files are not prohibited from appearing at this level, only directories can be seen at this level in this example. The name of each child element is labeled
with its name. When a user hovers over a child element, the bounding boxes for the next level of elements from the tree is temporarily revealed. If a user clicks on a child element, that bounding box expands to encompass the full width of the visualization. Additionally, the top bar’s text will change to include all ancestor nodes’ text and the child element’s text. 

Figure 4.2: A treemap representation of the Radius system

The color scale is calculated relative to the system’s statistical slicing sizes. Red, the highest end of the scale, is the greatest average number of slices present in the system at the function level. Yellow, the middle of the scale, represents the median of the average number of slicing lines across the entire system. Green and orange represent the system’s lower and upper quartiles, respectively. The lowest end of the scale, blue, is the minimum. Each entity’s color is determined through an interpolation function, provided by D3, on this scale based on the entity’s average slicing size.

This visualization allows users to gain a broad understanding of the system’s complexity. Users
are able to quickly identify the most complex portions of the system using the color scale and size. Without such a visualization, users would have inspect each file individually for each section of a system that they are interested in understanding. For example, if a user wanted to determine which portion of the Radius system would be the most complex to edit so they could estimate effort and time, then the user would need to investigate each subsystem and their associated files to get an understanding of their relative complexities. This likely involves some amount of estimation or guesswork on the part of the user, especially for large systems. However with the treemap visualization presented here, users quickly gain an understanding each subsystem at a glance. The number of slicing lines determines the size of each bounding box, allowing users to better understand the scale of that subsystem. The color allows users to understand the relative complexity of the subsystem in terms of how frequently variables are used. These aspects of the visualization allow users to quickly understand a system’s complexity with less guesswork and estimation.

4.2 Bipartite Graph

This section is taken verbatim from previously published work on vizSlice\cite{36}.

The goal of the function level visualization is to show the detailed relationships present within the function of interest. The visualization shows the exact lines that a variable is used on and the original code for the function. To do this, we use a bipartite graph. In particular, given a function f within a program, we create a bipartite graph G = (U, V, E) for f, with U equal to the set of all variables in f, V equal to the set of the union of all slicing lines for those variables, and E equal to the set of all relationships between the variables in U and their corresponding slicing lines. More
formally:

\[ U = \{ \text{variables in } f \}, \]
\[ V = \{ \text{Slines}(u) \mid u \in U \}, \text{ and} \]
\[ E = \{ (u, v) \mid u \in U \land v \in \text{Slines}(u) \} \]

where Slines(u) is the set of slicing lines for variable u. Since U and V represent two different kinds of entities, namely variables and source lines, the graph contains no odd cycles. It is important that no odd cycles are present in the data, otherwise the visualization would become cluttered and difficult to read with connections arching between two lines or two variables[36].

Figure 4.3 contains a visualization of a function found in the Radius open source system. The bipartite graph was generated using D3 and depicts variables on the left-hand column and Slines in the right-hand column. Since the bipartite graph is constructed at the function level, all of the variables, including parameters, of a given function are shown on the left, while the slicing lines of a function are shown on the right. The relative sizes of the “bars” for the variables provide a sense of the size of the slice for that variable. Similarly, the multi-colored bar size for a slicing line indicates the number of variables used by a given line of code[36].

As shown in Figure 4.3, we provide a number of annotations relevant to each slice. Specifically, in addition to the labels for the variable names and the slicing line, a count is displayed that shows

![Figure 4.3: An example bipartite graph - full graph](image)
the number of slicing lines related to a variable, or conversely, the number of variables associated to a given slicing line. For instance, the variable labeled \textit{greq} has a count value of 8 indicating that it appears on 8 separate slicing lines within the source for the function. Likewise, for source line 65923, the count value of 3 indicates that 3 variables appear in that line of code[36].

In addition to labeling the graph, we use color to capture the relationship between a given variable and the slicing line in the right-hand column. When all relationships are depicted in the bipartite graph representation, the coloring provides the user a means for locating the neighbor for a given variable or slicing line[36].

In order to reduce complexity of the bipartite graph representation and to support impact analysis, our visualization allows a user to focus their attention on specific variables or lines of code. In the case of specific variables, clicking on the bar adjacent to a variable isolates the slicing lines for that variable only, as shown in Figure 4.4. This allows the maintainer to get a visual feel for how drastic a change will be. For example, if the slice selected highlights a large portion of the slicing lines, the maintainer will know that the change is quite drastic. This may perhaps lead the maintainer to make a decision that the new feature should be refactored into an independent module rather than being a part of the current function. As can be seen in the visualization, selection of the bar adjacent to \textit{greq} reveals that the corresponding slicing lines are 65925, 65934, 65938, 65941, 65942, 65943, 65947, and 65948[36].

A similar complexity-reducing visualization is shown in Figure 4.5. In this case, the focus of the visualization is on a particular line number. When maintainers select the bar adjacent to a slicing line, the visualization focuses attention on those variables that are referenced on that particular line. In Figure 4.5 the selected line 65923 references three variables: \textit{input}, \textit{inputsize}, and \textit{output}. Note that the bar uses three colors, with each color corresponding to the colors used for the variables[36].
4.3 Parallel Coordinates

The parallel coordinates idiom is used to display multidimensional data along multiple axes\cite{39}. A parallel coordinates visualization is visually similar to a bipartite graph when only two axes are used, however it differs in that it facilitates more complex interactions that allow users to investigate blocks of code and variable groupings. Each axis is displayed as a single line and is placed parallel to the previous axis. Each data point then traverses across these axes according to its properties. In Figure 4.6 we can see an example of cars being represented in a parallel coordinates visualization. Each car is a line and each axis represents one aspect of the car. The axes, from left to right, are mpg, cylinders, displacement, horsepower, weight, the time it takes to go from 0 to 60, and the year it was made. This type of visualization allows users to identify patterns and relationships quickly in a data set \cite{39,40}.

There exist two primary types of interactions for these visualizations in a digital space. The first, brushing, allows for users to select ranges on each axis that the user is interested in exploring. If a user selects a particular range in the data, the data points that are excluded are greyed out across all axes. Users can do this across multiple axes to filter data to their interests. An example of this can be seen in Figure 4.7 when a grouping of elements are selected in the weight column. The second, is axes reordering. This allows users to reposition axis to find meaningful relationships between axes that are not initially placed adjacent to each other.
Brushing interactions allow us to investigate the data in a way not currently possible with the bipartite graph. A common software maintenance task involves editing code blocks or continuous sections of code. While this is a common task, the previous visualizations do not address this need in the context of the available data. In vizSlice we are interested in displaying only two axes, one for variables and one for line numbers. We use this idiom to display the same data used in the bipartite graph. Since we are only using two axis in our implementation, we are not particularly interested in utilizing axes reordering and instead focus on brushing. With parallel coordinates in vizSlice, users can utilize the brushing interaction to filter out code blocks that interest them. The brushing interaction can be performed on the right axis to select multiple consecutive lines of code. Figure 4.8 shows an example of this interaction where lines 455 to 464 has been selected. This allows the user to quickly identify the variables that impact this particular section of code. In the example shown in Figure 4.8, users can quickly identify all seven variables that impact the selected lines. Additionally, the brushing interaction can be performed on the left axis to select
Figure 4.7: An parallel coordinates visualization using brushing\textsuperscript{[40]}

a grouping of consecutive variables. Brushing can be used on both axis simultaneously, as well. This allows users to investigate both a particular section of code and a grouping of variables. The primary reason we included this visualization in addition to the bipartite graph is how they differ in usage. The brushing interaction provided by the parallel coordinates visualization allows users to investigate blocks of code such as conditional statements or loops. This is not possible with the current implementation of the bipartite graph.
Figure 4.8: Parallel coordinates with brushing in vizSlice
4.4 Code View

When performing software maintenance tasks, it is natural to want to view the source code of the system. Several paradigms currently exist for source code viewing and editing. In the context of slicing, SeeSlice uses a code view as the primary visualization idiom in its visualization framework. To incorporate the level of detail that such a visualization includes, we have included a simple code viewing framework. This framework utilizes the srcML file output from the srcML tool. The srcML file format utilizes XML to create a fully queryable document that contains the entire source code for the system. This file format preserves comments, whitespace, and preprocessing statements. An example input to srcML and its associated output can be seen in Listings 4.1 and 4.2 respectively. Listing 4.1 shows original function select_matching_attr from the Radius system. The output of srcML records the programming language and file name within the third tag in the output shown in Listing 4.2. Every file within the system receives similarly structured unit tag. The function tag surrounds all information for a particular function within the defined unit tag for a particular file. This information includes the return type of the function, the function name, the parameter list, and all of the associated source code. Comments from the source code are also included, though not shown in this example. This information is formatted to follow XML syntax to allow for queries to be run against the document.

```c
int select_matching_attr(void *data, char const *name, 
    grad_dict_attr_t const *dict_entry ARG_UNUSED)
{
    struct dict_match *dm = data;
    if (strlen(name) >= dm->len && strncmp(name, dm->text, dm->len) == 0)
        obstack_grow(&dm->stk, name, strlen(name)+1);
    return 0;
}
```

Listing 4.1: Example input to srcML
Utilizing standard querying features of XML and the full function path, we pull the code from this document into vizSlice and remove all XML tags. Then, utilizing Highlight.js [41], we provide syntax highlighting on the code. This provides a view like what users are already familiar within IDEs. Additionally, we include an overlay that provides a minimized view of the code that allows users to view the larger context of the function. Finally, to fully integrate the view with the visualizations, when a selection is made in the bipartite graph or parallel coordinates visualization, the associated lines of code are highlighted in the code view. This allow users to view the slicing data in the context of their own code. An example of this can be seen in Figure 4.9. In the corresponding bipartite graph representation of the function, the variable data was selected. The lines of code that data is used on were then highlighted in the code view. The integration of the code view with the function level visualizations was completed by Paulo Caetano Virote de Souza.

![Figure 4.9: An example of the code view in vizSlice with highlighting after a selection in the bipartite graph](image)
Listing 4.2: Example output of srcML
Chapter 5

vizSlice Environment

This chapter presents an overview of the vizSlice environment. Section 5.1 presents architecture of vizSlice, Section 5.2 presents the integration of the visualizations into the environment, Section 5.3 presents the implementation of the data pipeline from source code to vizSlice, and Section 5.4 describes the typical usage of vizSlice.

5.1 Architecture

vizSlice was built utilizing an Ubuntu server on a standard Linux, Apache, MySQL, PHP (LAMP) configuration and an install of srcML and srcSlice. We use a combination of HTML, CSS, JavaScript, PHP, and Python in the structure of vizSlice. PHP, HTML, and CSS form the core of the vizSlice’s design. Meanwhile, JavaScript primarily handles the implementation of the visualizations and some minor data wrangling. Python is used to generate a JSON file for each system being visualized within vizSlice. When a repository is added to vizSlice to be visualized, PHP is used to run srcML, srcSlice, and the Python parser in the appropriate order. This creates the two main files needed for vizSlice to visualize the selected system. When the user then selects a system to visualize, the treemap will use the JSON file generate by the Python parser. After
navigating to a function and choosing to view a more detailed visualization, **vizSlice** will then use the function’s name to create a temporary JSON adjacency list of that function for use in the bipartite graph and parallel coordinates visualizations. The function name is also used to query the srcML document to pull in the original source code into the code view.

### 5.2 Visualizations

Visualizations in **vizSlice** are primarily implemented in JavaScript utilizing the D3 JavaScript library[29]. Data stored in the JSON object is used to generate the necessary data format for each visualization. The code view utilizes Highlight.js[41] for syntax highlighting. Each visualization is explained in additional detail in chapter 4.

From the homepage of **vizSlice**, a user can choose to download the data files for previously sliced systems generated by srcML, srcSlice, and the python parser. Additionally, they can choose to view the treemap for the selected system. After navigating to the treemap, users must then navigate to a function of interest within that system. This is done through clicking through the various levels of the treemap. After a function is selected, users must click on any variable within the treemap to navigate to the detailed function level visualizations. The default view for the detailed function level visualization is the bipartite graph. A parallel coordinates view can be selected from a drop down menu. The code view is present with both of these visualizations. A users can interact with either the bipartite graph or the parallel coordinates visualization if they desire to highlight or bring to focus a specific relationship or set of relationships. In Figure 5.1 an example of the function level visualization with the code view can be seen. The full function path can be seen at the top of the page. Users can use the drop-down menu in the top-right to change the visualization to parallel coordinates. The visualization appears on the left and the code view on the right. When a selection is made in the visualization, the code view highlights the appropriate lines of source code. Additional controls for the system are on the left of the screen.
5.3 Pipeline Backend

The pipeline for *vizSlice* can be seen in Figure 3.1. The source in this pipeline is a tarball of a git repository or a public git repository link. When a link is provided, *vizSlice* clones the repository. If a tarball is uploaded, *vizSlice* uses this instead of cloning a repository. After the repository is cloned, srcML is run against the repository, producing an XML document. The format of this XML document can be seen in Listing 4.2. Once srcML has completed this, srcSlice is run using the XML output. This generates the list of slicing profiles needed for the visualizations. However, this data is still not in the correct format, therefore the Python parser generates the JSON object that the visualizations require. Finally, when all data files are generated they are moved to appropriate locations on the server for storage. As srcML currently only supports C, C#, C++, and Java, so does *vizSlice*. If a repository contains files of an unsupported programming language, those files will be ignored by srcML and, subsequently, the rest of the pipeline. The timing of these events is controlled by PHP scripts written by Luke Pitstick that run the appropriate bash commands.

5.4 Typical Usage

Typical usage of *vizSlice* would include visual impact analysis and software comprehension tasks. Software maintainers often need to evaluate the source code complexity of a system in order to estimate maintenance effort. *vizSlice* allows users to do this by analyzing slicing data visually.
Should a user desire to modify an entire subsystem within a system they are evaluating, they can do so using the treemap visualization. The size of the system, in terms of the number of slicing lines, is represented by the size of the bounding box for that subsystem. Additionally, the color represents the average number of slicing lines that portion of the system contains at the function level. Each bounding box is sorted by its size. This allows users to quickly identify the more complex areas of the subsystem quickly. This view is also provided for each function. This allows users to quickly identify which variables have the highest usage within the function. If a user desires to view the function in detail, perhaps to view how different variables impact each other, this is possible by moving to the bipartite graph or parallel coordinates visualizations. Within the bipartite graph, users can either select a variable to view which lines it impacts or select a line to see which variables impact it. If a user is interested in a more complex selection, such as the interactions between variables within a block of code, such as a conditional statement, they can use the parallel coordinates visualization. Here users can select a continuous set of line numbers, such as those representing a code block. This will allow users to quickly see which variable impacts that portion of the function without having to read source code, giving users a high-level understanding of the complexity.

Suppose a user is evaluating the divide function within the vasnprintf.c file from the gnu directory of the Radius system. Using the treemap, the user can quickly identify that the r_ptr variable has the highest usage in this function at 21 slicing lines. A user can then move to the bipartite graph to view what these slicing lines are and how r_ptr interacts with other variables. Users can identify patterns using the bipartite graph by selecting the variable. Patterns in usage, such as how many other variables impact the same lines or how distant each slicing line is from another slicing line, can be identified quickly without needing to read the source code in detail. Perhaps a user knows of a particularly complex code block they need to maintain. They can quickly filter out which variables impact this code block using the parallel coordinates visualization. From the divide function, if a user selects line numbers 455 to 464, they can quickly identify the seven
variables within this selection of code with little effort. This can be seen in Figure 4.8. Additionally, users can utilize the parallel coordinates visualization to determine if multiple variables overlap in usage. Users can select a grouping of variables that they may be interested in and determine quickly if these variables directly interact with each other. Finally, users can view the source code for a function with both the bipartite graph and parallel coordinates visualization if they need to the amount of detail that the source code provides.
Chapter 6

Evaluation

As previously mentioned in Section 3.1, vizSlice was evaluated to measure usability with respect to scenarios relevant to software maintenance. During these evaluations we sought to answer the following questions:

1. How easy is it for users to navigate vizSlice?

2. How well can users answer slicing-based software maintenance questions correctly and efficiently with vizSlice?

In order to evaluate these questions we utilized a think aloud protocol\[31\] and heuristic evaluation \[33\].

6.1 Design

We evaluated the system through user studies for usability. Validation of the system and of the chosen visualizations was evaluated for common use cases such as impact analysis. A think aloud protocol\[31, 32\] was used to evaluate a user’s thought process while performing a task within the system by an evaluator asking a series of evaluation questions at each step in the task. A think aloud protocol allowed us to investigate a user’s ability to efficiently utilize vizSlice for slicing based
tasks. Users also completed a heuristic evaluation by answering several survey questions using a Likert scale. A heuristic evaluation allowed us to evaluate how user’s view vizSlice. In particular, we were able to investigate vizSlice’s performance and visual appeal, as well as additional heuristic measures.

### 6.1.1 Think Aloud Protocol Design

A think aloud protocol was used by two user groups, students and professionals, to actively evaluate the system through the use of scenarios. Before beginning a think aloud protocol, users were asked to develop a strategy that they would expect to be able to use to achieve the goal they are given to complete and then explain that strategy to the evaluator. The evaluator then asked three questions before any action was taken by the user on each screen. The three questions were:

1. What is your immediate goal?
2. What are you looking for in order to accomplish that task?
3. What action are you going to take?

Users may respond to the first question with the task they were given, but are more likely to respond with an intermediate goal. This could include responses such as “find the treemap” or “find a count”. When users respond to the second question, they respond with the type of control they are looking for, such as a button or a menu selection. The third question is answered when a user has decided what they intend to interact with, such as clicking a button or typing in input.

After these questions were answered, the user then completed the action and the evaluator asked the following two questions:

1. What happened on the screen?
2. Is your goal complete?
Users respond the the first of these questions with what they saw happen on the screen. Sometimes this response is “nothing”, such as when they attempt an invalid action. However a typical response is more likely to be “the screen changed” or “an error message appeared”. Users usually answer the second question with usually a “yes” or “no” response.

These steps were repeated until the user completed the primary tasks given for evaluation. The evaluator also made observations of usability problems during the think aloud process. Additionally, screen recordings captured both audio and on screen actions to allow for more detailed analysis and comparisons after completion of each think aloud. Users were given the following software maintenance slicing based tasks to be completed during the think aloud protocols:

- Navigate to the treemap to find vasnprintf.c in the gnu directory of the radius system.

- Navigate to the divide function.

- Identify the number of variables used in the system.

- Identify the top 3 variables with the highest impact (in terms of line usage) in the system.

- Identify the 3 variables with the least impact (in terms of line usage) in the system.

- From the treemap go to the detailed function visualization.

- Using the bipartite graph,
  - How many lines is the variable b_msd used on?
  - Which line numbers is it used on?
  - How many and which variables impact line number 619?

- Using the parallel coordinates visualization,
  - How many and which variables impact lines 455 to 464?
  - Are the variables num and remainder used within the same code block at any point?
To the first, third, and fourth questions presented in this list, users are asked to respond with some number to represent a count. The third and fourth questions also requires users to respond with variables names, such as "num" or "remainder". The second question requires users to respond with line numbers such as “400” or “120”. For the last question asked, users respond with either a “yes” or “no”. No explanations of their answers need to be given, however users may explain anyway due to the design of a think aloud protocol.

6.1.2 Heuristic Evaluation Design

Users were asked to perform a heuristic evaluation [33], after completing the think aloud protocol.

The heuristic evaluation survey asked users to answer a series of questions regarding usability of vizSlice using a Likert scale. The survey was distributed digitally using Qualtrics. Usability questions focused on three primary areas: visibility, affordance, and feedback. However, the survey also included secondary questions regarding areas such as navigation, language, errors, and user support. Many of these questions overlapped between primary question and secondary questions.

The following questions were asked during the heuristic evaluation survey:

Visibility of system status

1. It is always clear what is happening on the site?

2. Whenever something is loading a progress bar or indicator is visible?

3. It is easy to identify what the controls are used for?

Match between system and the real world

4. The system uses plain English?

5. The website follows a logical order?

User control and freedom

6. The user is able to return to the main page from every page?
7. The user is able to undo and redo any actions they may take?

**Consistency and standards**

8. The same words are used consistently for any actions the user may make?

9. The system follows usual website standards?

**Error prevention**

10. There are no broken links or images on the site?

11. The system is error free?

12. Errors are handled correctly if they occur?

**Flexibility and efficiency of use**

13. Users may tailor their experience so that they can see information relevant to them easily?

**Help users recognize, diagnose, and recover from errors**

14. Errors are in plaintext?

15. The problem that caused the error is given to the user?

16. Suggestions for how to deal with an error are provided?

**Help and documentation**

17. Help documentation is provided to the user?

18. Live support is available to the user?

19. User can email for assistance?

These questions were then answered using a Likert scale from consisting of five answers from strongly disagree to strongly agree [34]. Values from 1 to 5 were assigned to each response, where strongly agree was 1 and strongly disagree was 5. This allowed for further numerical analysis of the data.
6.2 Data

This section describes the results from the evaluation. Section 6.2.1 describes the results from the think aloud protocol and Section 6.2.2 describes the results from the heuristic evaluations.

6.2.1 Think Aloud Protocol Results

A total of 9 student groups and 4 professionals completed think aloud protocols. Each student group consisted of approximately 4 software engineering and computer science undergraduate students enrolled in CSE 212 - Software Engineering for UI/UX. Students were primarily third year students, although some second and fourth year students were included. Each professional held development positions in industry and held computing degrees.

Each think aloud protocol was screen captured. Upon review of the screen captures two student groups had bad recordings due to technical problems and were excluded from our analysis. One student group only partially completed the given tasks and failed to finish the think aloud protocol. It is possible that this student group failed to complete the given tasks because they found the task difficult or were no longer interested in completing the exercise. Regardless, this think aloud protocol was also excluded from the following analysis. This resulted in 10 usable evaluations.

Each think aloud protocol was evaluated on a rubric. The rubric broke the given tasks down to smaller actions that would need to be taken to complete the larger given tasks. This resulted in a total of 13 actions for evaluation. Each action was evaluated on a scale of 0, 1, or 2. Users received a 0 mark if they experienced extreme difficulty, failed to complete the task, or answered incorrectly, a 1 mark if they experienced moderate difficulty, and a 2 mark if they finished the task correctly with relatively little difficulty. The explicit tasks given for evaluation can be seen in Section 6.1.1. The following 13 tasks were included in the rubric, and encompassed the explicit tasks given to the users.

1. Navigate to the treemap visualization.
2. Navigate through the treemap visualization to the divide function.

3. Identify the number of variables in the divide function.

4. Identify the top 3 and bottom 3 variables, in terms of line usage, in the system.

5. Navigate to the bipartite graph visualization for the divide function.

6. Identify how many lines $b_{msd}$.

7. Identify which lines $b_{msd}$ is used on.

8. Identify how many variables affect line number 619.

9. Identify which variables affect line number 619.

10. Navigate to the parallel coordinates visualization.

11. Identify how many variables impact lines 455 to 464.

12. Identify which variables impact lines 455 to 464.

13. Identify whether or not the variables $num$ and $remainder$ are used within the same code block at any point within the function.

These tasks were broken down into categories based on what portion of the system is being evaluated. The first grouping includes only Task 1. This task allows for the evaluation of the navigation of the system upon initialization. The second group includes Tasks 2 through 4. These tasks are used to evaluate the use of the treemap, including its navigation and comprehension. The third grouping includes Tasks 5 through 9. These tasks are used for the evaluation of bipartite graph. The fourth grouping includes the remaining tasks, Tasks 10 through 13. These tasks are used for the evaluation of the parallel coordinates visualization.

The raw data for the student evaluations is presented in Table 6.1. From the 6 student groups that completed the think aloud protocols, it is apparent that students struggled significantly with Tasks 1, 5, 11, and 13. It is also apparent that student users excelled on Tasks 3, 6, 8, and 10.
Figure 6.1 displays these results as a stacked bar chart, making the struggles and successes of the students’ experiences clearer. The remaining Tasks appear to have mixed results among student users. Table 6.2 presents the raw data for the professional evaluations. From the 4 professionals evaluated, it is apparent that these users strongly struggled with Tasks 1, 3, 4, 7, 11, 12, and 13. Professional users performed well on Tasks 2, 6, and 10. The remaining Tasks had mixed results in this user group. Figure 6.2 shows this data as a stacked bar chart.

Table 6.3 shows the averages of the results of the evaluation of each user group and of the combined users scores. Figure 6.3 shows these values as a bar chart, allowing us to compare normalized values across user groups. Through this we can see that the two user groups’ performances differed significantly (by at least a difference of 1) on Tasks 3, 4, 7, and 12. Professionals only out-performed students on one question, question number 5. Both user groups had agreement on question 10.

From Table 6.4 and Figure 6.4 we can see that student users had a better experience than professional users when the averages are aggregated by the tasks’ inherent groupings. The radar chart displays the various groupings on separate axes in a radial layout. Each value from the data series is then plotted on along these axes. The middle value of the axes is highlighted in red for reference.

The bipartite graph and treemap visualizations show fairly positive results. While there are issues with a few of the controls in these visualizations, users that struggled likely did so due to inexperience with these type of visualizations. Users from both groups learned how to utilize the controls quickly. This is particularly evident for the bipartite graph, where users from both groups showed improvement on Task 9 - Identify which variables affect line number 619 when compared to Task 7 - Identify which lines b_msd is used on, as seen in Figure 6.3 which both have similar controls. Additionally, through Figure 6.4 we can see that professional users had a poorer experience with the system as a whole than student users. The portion of the system with the worst usability is the home page and parallel coordinates.
Table 6.1: Raw data from the student think aloud protocols

<table>
<thead>
<tr>
<th>Group</th>
<th>Task Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 2 2 2 0 2 2 2 1 1 2 1</td>
</tr>
<tr>
<td>2a</td>
<td>0 2 2 2 1 2 0 1 0 2 0 0 0</td>
</tr>
<tr>
<td>2b</td>
<td>0 2 2 2 1 2 1 2 1 2 1 2 2</td>
</tr>
<tr>
<td>3</td>
<td>2 2 1 1 0 2 2 2 2 2 1 1 1</td>
</tr>
<tr>
<td>6</td>
<td>2 1 1 1 0 2 2 2 2 2 2 0</td>
</tr>
<tr>
<td>7</td>
<td>0 0 2 0 0 2 1 2 2 0 1 0 0</td>
</tr>
</tbody>
</table>

Table 6.2: Raw data from the professional think aloud protocols

<table>
<thead>
<tr>
<th>Group</th>
<th>Task Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 0 0 1 2 0 2 2 2 1 1 0</td>
</tr>
<tr>
<td>2</td>
<td>0 2 0 0 1 1 1 0 0 2 1 0 0</td>
</tr>
<tr>
<td>3</td>
<td>1 2 0 0 1 1 0 1 1 2 0 0 0</td>
</tr>
<tr>
<td>4</td>
<td>0 0 0 0 0 1 1 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Figure 6.1: The results of rubric evaluations on the student groups
Figure 6.2: The results of rubric evaluations on the professionals

<table>
<thead>
<tr>
<th>Group</th>
<th>Home Page</th>
<th>Treemap</th>
<th>Bipartite Graph</th>
<th>Parallel Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students</td>
<td>1</td>
<td>1.50</td>
<td>1.67</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>0.50</td>
<td>2</td>
<td>1.33</td>
<td>1.83</td>
</tr>
<tr>
<td></td>
<td>1.50</td>
<td>1.50</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>1.67</td>
<td>0</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>0.67</td>
<td>0.75</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>0.80</td>
<td>1.40</td>
<td>0.70</td>
<td>1.40</td>
</tr>
<tr>
<td></td>
<td>1.70</td>
<td>1.20</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>1.40</td>
<td>0.70</td>
<td>1.50</td>
<td>0.80</td>
</tr>
<tr>
<td>Professionals</td>
<td>0.80</td>
<td>1.03</td>
<td>1.20</td>
<td>0.88</td>
</tr>
<tr>
<td>Combined</td>
<td>1.50</td>
<td>1.50</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>1.25</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>1.25</td>
<td>0.50</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.40</td>
</tr>
</tbody>
</table>

Table 6.3: Averages from the think aloud protocol by task number

<table>
<thead>
<tr>
<th>Group</th>
<th>Home Page</th>
<th>Treemap</th>
<th>Bipartite Graph</th>
<th>Parallel Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students</td>
<td>1</td>
<td>1.44</td>
<td>1.43</td>
<td>1.08</td>
</tr>
<tr>
<td>Professionals</td>
<td>0.50</td>
<td>0.42</td>
<td>0.85</td>
<td>0.56</td>
</tr>
<tr>
<td>Combined</td>
<td>0.80</td>
<td>1.03</td>
<td>1.20</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Table 6.4: Averages from the think aloud protocol by task groupings
Figure 6.3: The average results of rubric evaluations by task number

Figure 6.4: The average results of rubric evaluations by task groupings
6.2.2 Heuristic Evaluation Results

Students were asked to complete the heuristic evaluation survey individually. Of the approximately 32 students, only 15 students chose to begin the survey. Of the 15 students that began the survey, 2 chose not to complete it. Those 2 respondents were dropped from the analysis. This left 13 student responses. The 4 professionals also completed surveys. Of the 4 professional respondents, 1 respondent did not answer all questions. The questions for the heuristic evaluation are presented in Section 6.1.2. The raw data for the students and the professionals can be seen in Table 6.5 and Table 6.6 respectively. Users could respond with strongly agree, somewhat agree, neutral/neither, somewhat disagree, strongly disagree. These responses have been represented by integer values from 5 to 1. For example, strongly agree is represented by a 5, neutral is represented by a 3, and strongly disagree is represented by a 1. Therefore if a user responds with a lower value, that user holds a lower opinion of that particular statement.

Central tendency of the responses were calculated using medians\cite{42}. These calculations can be seen in Table 6.7. Through this we can see that both user groups agreed on Questions 3, 5, 10, 12, 14, 15, and 16. The system was generally rated favorably, with only 4 questions’ combined scores rated lower than 3 or neutral. These questions are 1, 3, 17, and 18. There are also 8 questions rated above 3, or neutral. This includes questions 5 through 11 and question 19. The remaining 7 questions, 2, 4, and 12 through 16, were rated as neutral. This can be seen easily in Figure 6.5 The red line represents a neutral response. The central tendencies for the heuristic evaluation by the questions’ groupings are seen in Table 6.8. These values can be seen in a spider chart in Figure 6.6. The radar chart displays the various groupings separate axes in a radial layout. Each value from the data series is then plotted on along these axes. The neutral value 3 in both Figures 6.5 and 6.6 is highlighted in red. Figure 6.6 allows us to quickly identify which groupings are better rated by users, and which user groups rated the particular portions of the system more highly.
Table 6.5: Raw data from the student heuristic evaluations

<table>
<thead>
<tr>
<th>id</th>
<th>Visibility of system status</th>
<th>Match between system and the real world</th>
<th>User control and freedom</th>
<th>Consistency and standards</th>
<th>Error prevention</th>
<th>Flexibility and efficiency of use</th>
<th>Help users recognize, diagnose, and recover from errors</th>
<th>Help and documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 4 3</td>
<td>3 4</td>
<td>5 4</td>
<td>4 4</td>
<td>4 3 3</td>
<td>3</td>
<td>3 3 3</td>
<td>4 2 4</td>
</tr>
<tr>
<td>2</td>
<td>1 3 2</td>
<td>3 2</td>
<td>4 4</td>
<td>3 4</td>
<td>5 4 5</td>
<td>4</td>
<td>3 3 3</td>
<td>1 1 3</td>
</tr>
<tr>
<td>3</td>
<td>2 4 2</td>
<td>2 4</td>
<td>5 2</td>
<td>5 2</td>
<td>5 5 3</td>
<td>4</td>
<td>3 3 3</td>
<td>3 1 3</td>
</tr>
<tr>
<td>4</td>
<td>2 3 2</td>
<td>2 4</td>
<td>5 4</td>
<td>4 5</td>
<td>5 5 3</td>
<td>4</td>
<td>3 3 3</td>
<td>4 2 5</td>
</tr>
<tr>
<td>5</td>
<td>1 4 1</td>
<td>1 1</td>
<td>1 4</td>
<td>2 2</td>
<td>5 5 3</td>
<td>1</td>
<td>3 3 3</td>
<td>1 1 1</td>
</tr>
<tr>
<td>6</td>
<td>2 3 2</td>
<td>4 4</td>
<td>4 2</td>
<td>3 3</td>
<td>1 2 2</td>
<td>2</td>
<td>3 2 3</td>
<td>3 2 5</td>
</tr>
<tr>
<td>7</td>
<td>3 2 3</td>
<td>4 5</td>
<td>1 3</td>
<td>3 4</td>
<td>5 4 4</td>
<td>3</td>
<td>4 4 3</td>
<td>1 2 3</td>
</tr>
<tr>
<td>8</td>
<td>4 5 2</td>
<td>5 5</td>
<td>5 5</td>
<td>5 5</td>
<td>5 5 5</td>
<td>2</td>
<td>3 3 3</td>
<td>3 2 1</td>
</tr>
<tr>
<td>9</td>
<td>3 2 2</td>
<td>4 3</td>
<td>4 2</td>
<td>4 3</td>
<td>4 4 3</td>
<td>4</td>
<td>3 3 3</td>
<td>3 2 4</td>
</tr>
<tr>
<td>10</td>
<td>3 4 3</td>
<td>3 2</td>
<td>5 4</td>
<td>5 5</td>
<td>5 5 3</td>
<td>2</td>
<td>3 3 3</td>
<td>3 2 4</td>
</tr>
<tr>
<td>11</td>
<td>2 3 1</td>
<td>4 2</td>
<td>5 5</td>
<td>4 4</td>
<td>4 4 4</td>
<td>4</td>
<td>4 4 4</td>
<td>1 1 4</td>
</tr>
<tr>
<td>12</td>
<td>4 4 3</td>
<td>2 4</td>
<td>2 2</td>
<td>4 2</td>
<td>5 2 1</td>
<td>2</td>
<td>4 2 2</td>
<td>2 2 5</td>
</tr>
</tbody>
</table>

Table 6.6: Raw data from the professional heuristic evaluations

<table>
<thead>
<tr>
<th>id</th>
<th>Visibility of system status</th>
<th>Match between system and the real world</th>
<th>User control and freedom</th>
<th>Consistency and standards</th>
<th>Error prevention</th>
<th>Flexibility and efficiency of use</th>
<th>Help users recognize, diagnose, and recover from errors</th>
<th>Help and documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 1 2</td>
<td>3 4</td>
<td>5 3</td>
<td>3 4</td>
<td>5 5 3</td>
<td>4</td>
<td>3 3 3</td>
<td>1 5 5</td>
</tr>
<tr>
<td>2</td>
<td>4 3 2</td>
<td>2 4</td>
<td>3 5</td>
<td>4 4</td>
<td>5 5 3</td>
<td>2</td>
<td>3 3 3</td>
<td>3 3 3</td>
</tr>
<tr>
<td>3</td>
<td>5 5 5</td>
<td>5 5</td>
<td>5 5</td>
<td>5 4 5</td>
<td>5 5 4</td>
<td>5</td>
<td>3 3 3</td>
<td>3 3 3</td>
</tr>
<tr>
<td>4</td>
<td>4 2 1</td>
<td>2 3</td>
<td>2 5</td>
<td>5 5</td>
<td>2 3 3</td>
<td>-</td>
<td>4 - -</td>
<td>1 2 -</td>
</tr>
</tbody>
</table>

Table 6.7: The calculated medians by question number

<table>
<thead>
<tr>
<th>Group</th>
<th>Visibility of system status</th>
<th>Match between system and the real world</th>
<th>User control and freedom</th>
<th>Consistency and standards</th>
<th>Error prevention</th>
<th>Flexibility and efficiency of use</th>
<th>Help users recognize, diagnose, and recover from errors</th>
<th>Help and documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students</td>
<td>2 4 2</td>
<td>4 4</td>
<td>5 4</td>
<td>4 4</td>
<td>5 4 3</td>
<td>3</td>
<td>3 3 3</td>
<td>3 2 4</td>
</tr>
<tr>
<td>Professionals</td>
<td>4 3 2</td>
<td>3 4</td>
<td>4 4</td>
<td>5 5</td>
<td>5 5 3</td>
<td>4</td>
<td>3 3 3</td>
<td>3 2 4</td>
</tr>
<tr>
<td>Combined</td>
<td>3 3 2</td>
<td>3 4</td>
<td>4 4</td>
<td>5 4</td>
<td>5 4 3</td>
<td>3</td>
<td>3 3 3</td>
<td>3 2 4</td>
</tr>
</tbody>
</table>

Table 6.8: The calculated medians by question groupings
Figure 6.5: The medians from heuristic evaluation by question number
Figure 6.6: The medians from heuristic evaluation by question groupings
6.3 Analysis

This section analyzes the evaluation results presented in Section 6.2. Section 6.3.1 analyzes the results from the think aloud protocol presented in Section 6.2.1 and Section 6.3.2 analyzes the results from the heuristic evaluation presented in Section 6.2.2.

6.3.1 Think Aloud Protocol Analysis

In Figure 6.3 we can see that students generally outperformed professionals on each task, meaning that student users understood and utilized vizSlice more effectively than professionals. Professionals only outperformed students on one task, Task 5- Navigate to the bipartite graph visualization for the divide function. From Figure 6.4 we can also see that students had better performance across the system as whole, when compared to professionals. These contrast in performance may be due to differences in how the user groups approached the system. Students were more likely to explore the site as a whole and look for clearly marked labels and controls, but professionals tended to try interacting with the current state of the system rather than exploring. However, it’s important to note that neither group did well on this task overall.

Tasks where user groups generally performed well include tasks 2, 6, 8, and 10. These tasks spanned across all 4 task groupings. Specifically, these tasks included navigating through the treemap, using the bipartite graph to identify the number of lines a variable impacts, using the bipartite graph to identify the number of variables that impact a particular line, and navigating to the parallel coordinates graph. This indicates that controls for these tasks have good visibility, affordance, and feedback.

It is apparent that several groups, both students and professionals, had significant difficulty on many of the tasks. Both user groups struggled with tasks 1, 4, 5, 11, 12, and 13. These tasks also spanned across all 4 task groupings, and included both navigation and usability tasks. Users likely struggled in these areas due to a lack of visibility and affordance with the system’s controls. Mixed
results in Tasks 3 and 7 indicate a need for improvement as well. Task 3 required users to identify the number of variables being used within the selected function from the treemap view. This is a task we would like users to be able to complete easily in the system but seems to have mixed results. While users were easily able to see each variable in the selected function, they had to count the variables by hand to be able to answer this question.

Users’ actions while using vizSlice indicate that not all controls are visible to the user. Specifically, users failed to realize interactions with the visualizations were possible, resulting in longer response times, incorrect answers, and confusion about the system as a whole. Users quickly learned that interaction by clicking or hovering was possible with the treemap visualization, but often failed to realize interaction with the bipartite and parallel coordinate graphs was possible. Users consistently reverted to using the textual representation of the code or to tracing the visualization with their cursor or finger. Additionally, some controls did not afford the action required to utilize them. Users also expected controls to exist that did not, such as sorting in the bipartite graph. Some users first tried to sort the bipartite graph in order to obtain information more quickly. However sorting the bipartite graph is currently not a control available for the visualization.

These issues can be easily remedied by implementing a text-based tutorial feature for the system in order to provide users with the required information. Providing information regarding the visualization’s content and controls is expected to improve users’ understanding and performance when using vizSlice. Additional controls could also be implemented in order to meet users’ expectations, such as a control to sort the information presented by the visualizations.

Users also encountered some difficulty when navigating the system. In particular, users struggled when progressing from a treemap visualization to a function level visualization, and with switching the type of visualization displayed at the function level. This represents a lack of visibility and feedback around the controls used to make these transitions. To resolve the visibility issues, clearer controls can be implemented by adding explicitly marked buttons placed nearer the visualizations. Additionally, adding clear headings to pages to indicate where users are within the
system would resolve the feedback issue.

In summary, 4 tasks’ controls have good visibility, affordance, and feedback. The remaining tasks have some issues in regards to visibility and affordance. However, these issues are easily addressed by either tutorials, additional visual cues, or controls visually separate from the visualizations.

6.3.2 Heuristic Evaluation Analysis

Responses from the heuristic evaluation survey are ordinal, making the results a little more difficult to analyze. Responses indicate the level of agreement a user has with a particular statement. A 5 represents a response of strongly agree, a 3 represents neutral, and a 1 represents strongly disagree. Due to the nature of these responses, means were not calculated. Instead we used medians to determine central tendencies [42].

No central tendency measurement for any grouping fell below the neutral option. However using Figure 6.6 we identify which question groupings were rated most poorly. The combined scores for flexibility and efficiency of use, help users recognize, diagnose, and recover from errors, help and documentation, and visibility of system status were all poorly rated when compared to the remaining groupings. We can also see that professional users rated the system more positively than student users in most areas.

From the Figure 6.5 we can see that the two user groups evaluated the site differently. The most noticeable differences are in questions 1, 13, and 18, where professional users rated the question higher than the student users. Three groupings of questions had consistently positive scores. These groupings are user control and freedom, consistency and standards, and error prevention. The most poorly rated questions are questions 1, 3, 17 and 18. These questions are included in two groupings, visibility of system status and help and documentation. Improvement to questions 1 and 3 would likely be made with improvements to control visibility on the visualizations and with the addition of text based tutorials regarding the system’s intended usage, as previously mentioned in
Section 6.3.1 with regards to the think aloud protocols’ results. Questions 17 and 18 would likely be improved by these same measures.

The groupings flexibility and efficiency of use and help users recognize, diagnose, and recover from errors were rated poorly overall, when compared with other portions of the system. The grouping flexibility and efficiency of use included a single question: Users may tailor their experience so that they can see information relevant to them easily?. It was poorly rated by students, yet rated well by professionals. This could represent differences in the interpretation of the question or differences in the levels of exploration users completed prior to answering the survey. Regardless, improvement to this grouping could easily be made by implementing a login system that allows users to keep information relevant to them in a readied state. The grouping help users recognize, diagnose, and recover from errors was rated as neutral by both user groups across all 3 questions. This is likely because few users encountered issues that were interpreted as errors. Most responses to questions 14, 15, and 16 by both user groups were neutral. Out of 16 responses, questions 14, 15, and 16 had 12, 11, and 13 neutral responses respectively. While users likely experienced few errors, questions 15 and 16 refer to some form of error documentation being provided to the user. Therefore, this portion of the system could be improved through the addition of FAQ pages and system tutorials to reduce error rates and to provide information on what to do when errors are encountered.

Overall the system was rated well by users. No grouping was rated negatively by users. Every question grouping was rated neutral or higher, and issues found through the heuristic evaluation can be easily addressed through tutorials, additional visual cues, documentation, or controls visually separate from the visualizations.
6.4 Threats to Validity

There are two primary threats to validity. The first being the structure of the think aloud protocols. The two user groups had varied experiences. Student users performed think aloud protocols as groups of 4, and conducted the protocol independent of a researcher. This was done largely as a matter of convenience. Students were required to complete a think aloud protocol in groups as part of an in-class exercise. Professional users performed think aloud protocols as individuals with a researcher present and asking questions throughout. This may have impacted the results. Student users may have been subjected to group think, while professional users were not. This may explain the performance differences apparent between user groups in Figure 6.4. Additionally, this same problem may have impacted the differences in how the user groups responded in the heuristic evaluation to question 18, regarding the availability of live support. Professional users rated this statement as a 4 - somewhat agree while students rated it as a 2 - somewhat disagree. This may be the result of whether or not an ‘expert’, or researcher, was present, regardless of whether or not the researcher actively aided the user.

The second threat comes from the limitations of the questions from the heuristic evaluation. It is possible that the heuristic evaluation does not cover every relevant topic of design to properly evaluate the site. Additionally, there is some debate over whether a 5-point Likert scale is sufficient to determine individuals’ attitudes toward statements and alternatives, such as fuzzy logic based responses, have been suggested[43].
Chapter 7

Conclusions and Future Work

This chapter discusses the conclusions we have drawn from the vizSlice approach and its evaluation. Additionally, we include some discussion of potential future work on vizSlice. Section 7.1 discusses our conclusions and Section 7.2 discusses future work.

7.1 Conclusions

Results show that users were able to utilize the vizSlice tool efficiently to accomplish the tasks posed to them in the experiments. However, there is a significant difference between both user bases in how they approached and utilized the system. We discovered some areas for improvement in visibility and navigation for the current design. Specifically, users often failed to recognize all of the available controls and visualizations, resulting in some incorrect results and confused participants. Also of note, some struggled switching from one visualization to another, resulting in some affordance, visibility, and feedback issues.

Overall, we are quite pleased with vizSlice’s ability to assist users in program comprehension tasks. While our study identified areas of improvement, these are easily solvable. To conclude, vizSlice successfully allows users to quickly and efficiently evaluate slicing data for use in impact
analysis and system comprehension, and that several opportunities for future work exist.

7.2 Future Work

Future work includes adding text-based tutorials to improve user understanding of the visualizations and evaluating these for their usefulness. Work can also be done to improve vizSlice using specific user feedback regarding system visibility and the types of controls available to users, such as sorting variables in the function level visualizations. Additionally, a comparison of vizSlice with IDEs and other existing tools using the same slicing-based maintenance and comprehension tasks is necessary.

Additions to vizSlice can be made to utilize other aspects of the data provided by srcSlice. This may require modifying existing visualizations or adding new ones. vizSlice can also be expanded to include options for backwards slicing[44].

vizSlice’s approach can also be incorporated existing systems. There is potential use for vizSlice in the context of debuggers for code analysis. Additionally, vizSlice could be incorporated into git repository systems such as GitLab for static in-browser analysis of source code.

Additionally, we seek to distribute the results of the vizSlice evaluation to the community[45].
Bibliography


national Workshop on Visualizing Software for Understanding and Analysis, pp. 1–8, IEEE, sep 2011.


