vizSlice: Visualizing Large Scale Software Slices

Hakam W. Alomari, Rachel A. Jennings, Paulo Virote de Souza, and Matthew Stephan
Department of Computer Science and Software Engineering
Miami University
Oxford, OH, USA
Email: {alomarhw, kaczkara, virotep, stephamd}@miamioh.edu

Gerald C. Gannod
Department of Computer Science
Tennessee Technological University
Cookeville, TN, USA
Email: jgannod@tntech.edu

Abstract—Program slicing has long been used to facilitate program understanding. Several approaches have been suggested for computing 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. In this paper, we present our tool for visualizing large systems based on program slicing and through two visualization idioms: treemaps and bipartite graphs. In particular, we use treemaps to facilitate slicing-based navigation, and we use bipartite graphs to facilitate visual impact analysis by displaying relationships among system decomposition slices showing the relevant computations involving a given slicing variable. We believe our tool will support various software maintenance tasks, including providing analysts an interactive visualization of the impact of potential changes, thus allowing them to plan maintenance accordingly. Finally, we show that, through the use of both existing scalable slicing and scalable visualization approaches, our tool can facilitate analysis of large software systems.

I. INTRODUCTION

Software maintenance and comprehension are difficult tasks especially with large software systems. Program slicing is an attractive option for understanding the behavior of a software system by reducing the amount of code to be examined at any point in the maintenance process [1], [2]. The idea is fairly simple; given a variable and the location of that variable in a program, slicing identifies other parts of the program that are affected by this variable. The idea was first proposed by Weiser [3] as an aid to debugging. However, since that time, program slicing has been applied to almost all aspects of software engineering including testing, maintenance, debugging, reverse engineering, comprehension, automatic parallelization, refactoring, measurements, and impact analysis. These various applications of program slicing require different properties; thus, a number of different slicing definitions have been proposed. These definitions are covered in detail in various surveys of the slicing literature [4], [5].

Unfortunately, program slicing, in spite of being a very mature research area, suffers from performance and scalability issues when analyzing large software systems. As systems grow larger, the slicing data becomes more difficult to comprehend [1], [6], [5]. In addition to these performance and scalability issues, visualization of program slices is an open research problem [7], [8], [9]. While there are several visualizations techniques designed to represent various types of program data, few visualizations have been developed to aid in the comprehension of program slices.

To this end, we introduce a visual impact analysis approach and a tool, vizSlice, based directly on source-code slices. vizSlice uses a treemap to represent a slice-based system complexity in each level of abstraction. Additionally, it uses a bipartite graph representation at the function level to represent functions’ slices. vizSlice uses both size and color in these visualizations to encode slicing metrics. These visualizations enable system maintainers to assess the impact of proposed changes and better plan their maintenance efforts. It entails computing a slice for all variables in a system efficiently and modeling the slice as it pertains to the system. Specifically, our tool facilitates a fully automated visualization framework for change impact analysis based on our previous lightweight static forward program slicing approach [6]. Our motivation for developing this is the need to better visualize the large amount of data generated by slicing tools and to facilitate visual impact analysis.

The remainder of this paper is organized as follows. Section II discusses background and related work. Section III describes how we designed our slicing visualizations. This is followed in Section IV with details on using our tool and the insights it provides, including impact analysis. Finally, Section V summarizes the paper and provides our current research plans and future work.

II. BACKGROUND AND RELATED WORK

A. Slicing Approaches and srcSlice

The calculation of a program slice is, with few exceptions, based on the notion of a Program Dependence Graph (PDG) or one of its variants, for example, a System Dependence Graph (SDG). Unfortunately, building the PDG/SDG is quite computationally costly and space intensive. As such, slicing approaches generally do not scale well, and while there are some costly workarounds, generating slices for a very large system can often take days of computing time.

The slicing technique we use in our proposed vizSlice tool, srcSlice [2], [6], [10], is distinguished from existing work in multiple ways. The method we use is non-PDG based; there is no graph to traverse nor data-flow equations to be solved. srcSlice retrieves on-the-fly information only as needed. It is this on-the-fly approach that allows us to provide visualizations for impact analysis that scale well and are efficient. In addition,
unlike the majority of approaches in the literature, we compute a slice for every variable in the program including those that are local and global. New variables are added to the slice profile as they are encountered.

B. Slice Visualization

Slice visualization is a relatively unexplored research area, however, there is some notable work. Ball and Eick [7] provide an interface for exploring program slices that is browser based, called SeeSlice. It requires exploration through multiple columns at once, has rudimentary graphical features given its age, and requires many context switches during exploration. Additionally, its purpose is quite general, allowing for related procedure and file identification. Deng et al. [11] also developed a browser-based program slice viewer, which suffers from the same shortcomings as SeeSlice.

An approach that depicts dependency graphs based on PDG slices was developed by Krinke [12]. It provides a textual visualization of program slices for the purpose of fine-grained analysis. They argue that graphical visualizations are better suited to understanding in-the-large, whereas textual ones allow for deeper analysis. While this may be true for general understanding and purposes, the framework we develop is tailored to tasks such as impact analysis, and the data we use is at the low level, that is, local and global variables, and their respective forward slices. This low level allows us to facilitate deep analysis as it pertains to complexity and impact assessments. The same differences between our work and others apply to similar textual visualizations [13], [14].

There are some tools that visualize slices as graphs. PROVIS [15] displays PDGs that stem from slices. This falls victim to usability, as software analysts are required to navigate large graphs that have limited edges. Rilling and Mudar [16] develop 3D graphs intended for program comprehension. While this work is very interesting and provides powerful three dimensional visualizations, it focuses only on relationships and not impact analysis. We may extend our work in the future to see if 3D visualizations could aid in impact analysis.

The work most related to ours is that of Gallagher [8]. They develop a CASE tool called Surgeons Assistant. It displays decomposition slices textually, and also displays graphs. In contrast to our framework and tool, this approach involves much manual work as impact analysis involves careful inspections. In addition, the usability and intuitive nature of the graphs themselves are quite dated as they are built upon the VCG tool [17], which was developed over twenty years ago. Through VCG, Surgeons Assistant displays very simple graphs that require cumbersome exploration as these graphs are folded over one another. In addition, the visualization of these basic graphs do not scale well, especially when applied to current systems that are orders of magnitude bigger than those originally analyzed by Surgeons Assistant twenty years ago. Surgeons Assistant works only within the CASE environment, whereas our approach works on a variety of languages and environments, and is being implemented as a cloud solution on the web.

While slices are typically smaller than the original program, they may still be large and complicated, especially for large systems. Current slicing tools, which are based on code visualization via reduced textual representations, are inadequate to the task of exploring slices, and therefore limit a maintainers understanding of a program. In these tools, the statements in the slice are highlighted and colored using a single browser. If the slice crosses procedural or file boundaries, additional browsers must be opened to view other entities. Such tools burden the programmers with the task of examining many different slices of a program at once with many context switches because so much effort is required just to see and comprehend one single slice. Moreover, existing visualization solutions visualize program slices based on slices of program statements rather than decomposition slices of variables [7]. Our tool features forward decomposition slicing, capturing all computations on a given variable. This enables a maintainer to analyze all effects of proposed changes to a given variable and to isolate the effect of those changes.

There has been little-to-no research on slicing-based change impact analysis using modern visualization approaches applied on large and evolving systems. Visual impact analysis is a software visualization technique that allows software maintainers to judge the impact of proposed changes and to plan their maintenance accordingly. However, many current visualization techniques suffer from a scalability problem when dealing with a large amount of information, such as the information retrieved by slicing. The problem of scale has motivated work by many, including Hutchins and Gallagher [18], to reduce the amount of slicing information used to assess impact analysis.

III. TOOL DESIGN

Using slicing profile data, we have explored the use of different visualizations to help maintainers sort through the complexity of software systems. In this section we describe two different visualizations we employ in our tool for navigating source code based on slicing: (1) Navigation of source code with a treemap using complexity-based on slicing metrics, and (2) viewing dependencies using a bipartite graph representation on the slice to support visual impact analysis.

The vizSlice tool is a web-based application1 that implements a pipeline of tools including (1) srcML, an XML format for source code [19], [20], (2) srcSlice [2], [6], [10], (3) a newly created vizSlice parser used to extract information to enable visualization of slicing, and (4) D3 visualization [21]. vizSlice starts by checking out a system version from a GIT repository. It then converts source code to srcML format using srcSlice to build the system dictionary with slice profiles for all variables in the system. Finally, vizSlice parses the slicing profiles as input to D3 visualization. The parser converts the variable path from each slice profile into a tree structure, such as having each node in the tree represent one level of abstraction in its path, that is, subsystems, directories, files, and functions. Leaf nodes are variables in slice profiles.

1http://vizslice.csi.miamioh.edu
A. srcSlice Output

srcSlice produces a system dictionary of all the slice profiles of all variables. The dictionary is three-tiered and consists of three maps: On the first level is a map from files to functions. The second level is a map from functions to variable names. The third and final level is a map from variable names to slice profiles. Figure 2 is an example of srcSlice’s output showcasing its easy-to-parse format.

Due to the non-nested nature of the output format, file names are repeated for every function and variable the files contain. Similarly, function names will be repeated for every variable they contain. A typical use case of srcSlice is to see what areas a modification to a variable may impact, for example, a name change, type change, value change, et cetera.

B. Coarse-grained Visualization of Complexity

Variables with a large number of associated slicing lines have great impact on their associated functions and procedures. When these variables are localized within a function or procedure, the cost of changing those functions is likely higher. In particular, a function with several variables that have a large number of slicing lines may be expensive to modify. In order to help users identify complexity through the use of slicing, vizSlice employs a treemap visualization that uses size to represent slicing lines and color to depict an average slicing metric for the appropriate aggregate composite, for example, directory-level or file-level aggregation.

A treemap is a hierarchically composed visualization. A given view on a treemap shows sibling nodes of a tree as rectangles. A mouseover on an entity in a current view reveals the children of that given entity. In our visualizations, the top-level of the visualization depicts directories. As the mouse is used to hover over a directory in a treemap, such as the radius system and radiusd directory in Figure 1, the visualization reveals information for the entities contained in the composite. Clicking on a given section will zoom the visualization to the next hierarchical level of abstraction, in this case the file level. Selection of a file and subsequent click will reveal the next level of abstraction until the user reaches a function. This final level will show the variables of the function, each sized based on the relative size of the slices and colored based on the average color scale.

The color scale is calculated relative to the systems average 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 average slicing size across the entire system. Green and orange represent the systems lower and upper standard deviations, respectively. The lowest end of the scale, blue, is always 0.

https://www.gnu.org/software/radius/ - The size of the radius-1.6 system is ~100K LoC, the number of variables found is ~10K, and the execution time for the srcSlice tool is 1.115 seconds.
formally, the variables in $U$ and their corresponding slicing lines. More variables, and $E$ equal to the set of all relationships between $V$ equal to the set of the union of all slicing lines for those $(U, V, E)$ for $f$, with $U$ equal to the set of all variables in $f$, function $f$ within a program, we create a bipartite graph $G = (U, V, E)$.

C. Low-Level Visualization of Program Slicing

Our work on the visualization of program slices is focused primarily on providing an efficient and scalable approach for depicting the relationship between a given program variable and its corresponding program slice. While we recognize that a number of different levels of abstractions are available for generating the visualizations, for example, file-level and function-level slices, we chose to focus on the visualization of slices contained within a given program function. Thus, the maintainer can tell which variables relate to different computations of a program easily. As such, we focus on the Slices attribute of the slicing profile. 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,

1) $U = \{\text{variables in } f\}$,
2) $V = \{\text{Slices}(u) \mid u \in V\}$, and
3) $E = \{(u, v) \mid u \in U \land v \in \text{Slices}(u)\}$

where Slices($u$) 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.

Figure 3 demonstrates a visualization of a function found in the Radius open source system. The bipartite graph uses D3 and depicts variables in the left-hand column and Slices 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. 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.

As shown in Figure 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 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 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.

In addition to the labeling of 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 with means for locating the neighbor for a given variable or slicing line. Not shown here is the source code view that we provide below this graph to allow maintainers to view the source code relevant to the graphs.

IV. TOOL UTILIZATION

A. Using the tool

We have prepared a video describing our tool and how to use it, and describe it in this section. The original motivation for the construction of vizSlice was to support slice-based software maintenance tasks by our research group. We use what was learned from the application of vizSlice at our research lab for the following tool demonstration. The idea is to have software maintainers and developers use the tool to visualize the impact that a change has on other parts of the software. This impact is calculated by our srcSlice tool, which outputs raw slicing results in an XML file. When we first showed the raw srcSlice textual results to developers/maintainers, they were not using the data due to the complexity and size of the output file. Once the data was visualized with vizSlice, they started to use it to determine the magnitude of the risk of introducing bug fixes into releases, as well as determining areas to regression test due to code changes.

To accomplish this task using vizSlice, developers/maintainers start with the output of srcSlice tool in XML format. With vizSlice, they input the data choosing the field from the slice profile to use for the visualization, for example, Slices. Once loaded, the entire treemap is shown with the corresponding hierarchy. The particular directory of interest is then chosen based on the average slice-size complexity. Additional information can be obtained by mouseover to show more detailed views of specific directories. The context of specific color value is viewed in the top of treemap. As developers/maintainers are exploring, they can select particular entities to view, with the tooltip demonstrating average slice size information. In addition, the entity information can be used to switch the display to another entity. For example, while considering the average slice size complexity of a specific function, they may observe that the number is high and use that to switch to the bipartite visualization.

https://youtu.be/ldGX0tzxrik
The vizSlic tool is a web-based tool that is run on the system dictionary "slices.dict" output file from srcSlice tool. The srcSlice tool is a command-line tool that is run on the srcML format of a source code file(s). All of this is available from the srcML website.

B. Insights

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. For specific variables, clicking on the bar adjacent to a variable isolates the slicing lines for that variable only, as shown in Figure 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 greq reveals that the corresponding slicing lines are 65925, 65934, 65938, 65941, 65942, 65943, 65947, and 65948.

A symmetric complexity-reducing visualization can be accomplished by users focusing the visualization on a particular line number. When selecting the bar adjacent to a slicing line, the visualization focuses attention on those variables that are referenced on that particular line. For example, for our sample graph in Figures 3 and 4 selecting line 65923 references three variables: input, inputsize, and output. In that case, the bar uses three colors, with each color corresponding to the colors used for the variables.

V. CONCLUSION & FUTURE WORK

There are several features of vizSlic we are currently working on. We intend to roll these extra features out as they are tested. We would like the ability to visualize using different slice profile fields. That is, in addition to the Slices field, we plan on allowing users to change the complexity color scale based on: (1) called functions using a slicing variable, (2) dependent variables on slicing variable, and (3) pointer aliases of the slicing variable.

The vizSlic tool allows a user to quickly request and examine large system slices. A reduced visual representation of a system that displays the various levels of system hierarchy makes it possible to query, display, and browse slices in a highly efficient and interactive fashion. There are many different possible applications for vizSlic. As part of a system for understanding and reverse engineering, the tool can be used to identify slice-based related functions and files to extract and refactor code. Applied to debugging, vizSlic provides help in identifying source code that impacted by a specific program artifact.

REFERENCES


\[\text{http://www.srcML.org}\]