Constellation Visualization: Augmenting Program Dependence with Dynamic Information

Fang Deng
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: fdeng@ics.uci.edu

Nicholas DiGiuseppe
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: ndigiuse@ics.uci.edu

James A. Jones
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: jajones@ics.uci.edu

Abstract—This paper presents a scalable, statement-level visualization that shows related code in a way that supports human interpretation of clustering and context. The visualization is applicable to many software-engineering tasks through the utilization and visualization of problem-specific meta-data. The visualization models statement-level code relations from a system-dependence-graph model of the program being visualized. Dynamic, run-time information is used to augment the static program model to further enable visual cluster identification and interpretation. In addition, we performed a user study of our visualization on an example program domain. The results of the study show that our new visualization successfully revealed relevant context to the programmer participants.

I. Introduction

To perform various software-engineering tasks, developers often need to understand large software systems and how the various components of the software influence and relate to others. Software systems often contain many modules, files, and lines of code. To a certain degree, these software components are often organized in a logical, coherent fashion. For example, related files are placed in packages, and lines of code are organized with other, related lines in methods and classes. However, classes and files between packages often interact in fundamental and important ways. Similarly, lines within methods may call other methods or set field or global variables that are read by other lines in other methods. These relationships often form so-called cross-cutting, yet scattered functionalities. Understanding these relationships at both the high and low levels of granularity can be a difficult and time-consuming task for a developer, especially as the size of the software increases.

One popular software visualization that is capable of showing both high- and low-level program components is the SeeSoft visualization proposed by Eick *et al.* [4]. The SeeSoft visualization scales well to large programs and encodes problem-specific meta-data through its use of color for the software component representations. Its strengths come from the fact that it scales well, is able to present problem-specific information through color, and preserves the structure of the code. The preservation of the structure of the code helps developers in that it presents the code in a way that is familiar. However, SeeSoft does not explicitly encode the relationships among the components in the code, leaving that task to developers' manual search and interpretation.

The low-level relationships among lines of code can be obtained from dependence graph models of the software such as program and system dependence graphs (PDGs and SDGs). These models are seldom visualized as they tend to be quite large, both in terms of the number of nodes (statements) and edges (dependencies or relationships). While there are some exceptions, such as those proposed by Krinke [11] and Würthinger *et al.* [15], they generally do not present the program in a scalable form for medium to large programs. Dietrich *et al.* [1] enabled a certain degree of scalability by visualizing dependence graphs of programs at a higher level of granularity: the class level. These works initiated an exploration of the possibilities for leveraging these dependence graph models of programs to highlight relationships among program components.

In this work, we extend such notions of dependence graph visualizations to enable a system-wide visualization of software at a low-level of granularity. Our objectives are to make this visualization: scalable, helpful in enabling developers to identify relationships among statements in the code, applicable to problem-specific software-engineering tasks, and capable of interfacing with other visualizations.

To address these objectives, we created a novel program visualization that we are calling the *Constellation Visualization*. We present the system-dependence graph at the statement level. We make it scalable by hiding edges (by default) and presenting nodes as singular points. We depict relationships among statements in the program based on their proximity in the visualization; related code forms visual clusters. These clusters reveal related and relevant code context, often revealing core cross-cutting, yet scattered functionality. The static dependency model of the program is augmented with various forms of dynamic, run-time information that further supports the ability for the user of the visualization to identify related and relevant code context specific for a particular software engineering task. This dynamic information can be gathered from commonplace testing tools which impose low run-time overhead. Our visualization supports a number of software-engineering tasks by incorporating problem-specific meta-data and encoding it with color. It also works seamlessly with other visualizations through interactivity to leverage the strengths of each.

The main contributions of this work are:

- A novel visualization: the Constellation Visualization. We augment the scalable view of the system-dependence graph with lightweight dynamic information in a way that depicts related, clustered code. The visualization supports various software-engineering tasks through the use of problem-specific meta-data.
- A user study that compares two visualizations, the SeeSoft and Constellation visualizations, to determine if a benefit would be had by incorporating dependency information for a task which requires context to be ascertained.
- Examples of several software-engineering tasks for which we speculate that the new visualization will be helpful as an augment to traditional software visualizations, and describe how the visualization can be mapped to those domains.

II. BACKGROUND

A. SeeSoft Visualization

Eick *et al.* [4] created the SeeSoft visualization as a scalable program visualization that presents source code in a miniature, or zoomed-away, view. It presents each line of code as a single line of pixels that are colored to represent some aspect of the software. Eick *et al.* proposed a number of classes of source-code meta-data that can be mapped to colors, such as code authorship, program slices, and code age. An advantage of the SeeSoft visualization is that the code structure (e.g., indentation, statement length, and blank lines) is preserved in the visualization so that it is easily interpretable by developers. A potential area for improvement and extension could be had in enabling developers to understand how parts of the program can affect other parts of the program — this is the motivation for our new visualization.

As an example, one use of SeeSoft is for fault-localization. Jones *et al.* [9] proposed a technique (called TARANTULA) for identifying lines of code that are "suspicious" of causing failures, i.e., being the bug or fault causing those failures. The technique determines the strength of the correlation between statement execution and test-case failure — those statements that are primarily executed by failing test cases, but rarely executed by passing test cases are considered more likely of causing those failures. Those statements whose execution is highly correlated with failure are considered "suspicious" and are colored in a red hue; those statements whose execution is highly correlated with passing test cases are considered non-suspicious and are colored in a green hue; and statements whose execution is correlated to a degree in between these extremes are colored according to a continuous sub-spectrum of hues varying from red to yellow to green.

Figure 1 presents a program and a problem-domain that we will use as a running example in this and the next section. In this figure, we demonstrate the use of the SeeSoft visualization using this fault-localization technique. The program being visualized is a version of the Gzip Unix utility which contains
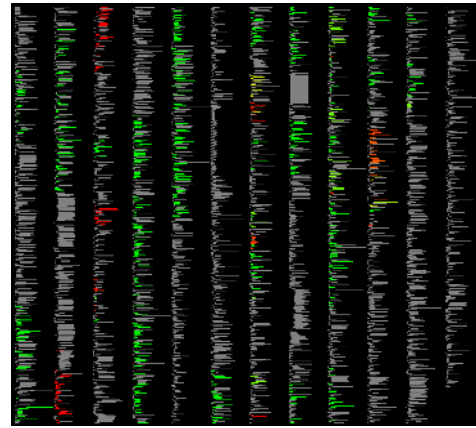

Fig. 1: An example of a SeeSoft view with coloring from the TARANTULA tool.

over 9000 lines of code and a single fault causing test case failures. Notice that the fault-localization technique determines that a subset of the program is suspicious of causing the failures. This visualization would be presented to the developers, whose responsibility would be to interpret the results, investigate those statements, and ascertain the relationships (or absence of relationships) among the disparate suspicious (red) statements.

Notice that in this visualization, the red lines are scattered into different parts of the source code. As such, developers would have no indication of where to begin their search or how the disparately located suspicious code interrelates. Anecdotally, we have found that in interpreting and exploring the code, the suspicious lines are most often interrelated and form a logical cohesion that is descriptive of the faulty behavior causing failures. Based on this experience, we sought to incorporate information that describes relationships between statements, however disparately located they may be.

B. Dependency Graphs

To describe these relationships between statements in code, researchers have proposed various dependency graphs. These dependency graphs often describe two types of relationships within the software: control and data. Control dependencies occur when a part of the program (e.g. a statement) controls the execution of another part of the program. For example, an `if` statement controls whether the code in its block is executed. Data dependencies occur when a part of the program assigns a value to a variable that another part of the program uses. For example, a read of a variable depends upon the definition of the variable that assigned its value.

Ottenstein and Ottenstein [14] proposed the program-dependence graph (PDG), which is an intra-procedural model that captures both control and data dependence among statements. Horwitz *et al.* [8] extended the PDG model by producing an inter-procedural model that in addition conveys dependencies between procedures. This model is called a system-dependence graph (SDG).

The traditional form of graph visualization — where a node is represented as an oval whose label describes the entity that it represents, and where an edge is represented as a line between nodes — is typically unscalable. The difficulty in scaling such a visualization stems from the fact that the number of nodes is roughly proportional and consistent with the number of statements in the source code, and more so for the multitude of interconnecting dependency edges.

Nevertheless, researchers have visualized such program dependence graphs. Krinke [11] proposed a visualization approach which uses a dedicated, declarative layout for program dependence graphs. It aims at making the graph more comprehensible than generic graph layout algorithms. Würthinger *et al.* [15] developed a tool to visualize program dependence graphs of Sun Microsystems' Java HotSpotTM server compiler. They provide additional features such as filtering mechanisms to allow manipulation on the program elements and a mechanism that shows the evolution of the graph. Dietrich *et al.* [1] visualize a higher-level dependency graph (i.e., dependencies among classes) using a "betweenness" clustering algorithm to compute the modular structure of programs so as to assist identifying component boundaries.

In addition to the difficulty in scaling such visualizations, these depict the static model of the program: all dependencies that *may* be fulfilled in any execution are given equal strength in informing node positions regardless of whether any executions actually do fulfill them. This static-only view of the dependency graph can be problematic because not all dependencies are fulfilled during execution, and those dependencies that are fulfilled are given equal strength regardless of fulfillment frequency. As such, such visualizations are unable to inform users of the degree to which program components actually depend on each other. We think that incorporating dynamic information is likely to give a better intuition of which dependencies are feasible or infeasible, weak or strong.

III. VISUALIZATION

In this section, we describe our new visualization by describing, in turn, each of the refinement steps that we applied to traditional dependence graph visualizations. Each step is described in its own section along with a running example visualization to demonstrate the benefit that that refinement brought. First, we describe a traditional method of visualizing dependence graphs in Section III-A. Then, we describe each of our refinement steps:

Step 1: For scalability of layout computation and human interpretation, do not visualize edges. (Section III-B)

Step 2: For scalability of interpretation, visualize each node as a single point. (Section III-C)

Step 3: For applicability to specific problem domains, color each node to encode problem-specific meta-data. (Section III-D)

Step 4: For ease of interpretation and pattern recognition, inform edge strength with dynamic information of the visualized program. (Section III-E)

Step 5: For further optimizing problem-specific interpretation and pattern recognition, allow for preferential treatment of categories of executions informing the dynamic information. (Section III-F)

Finally, we discuss our prototype implementation which supports interactive exploration of the visualizations in Section III-G. Each section contains a figure that presents an evolving visualization resulting from the refinement step.

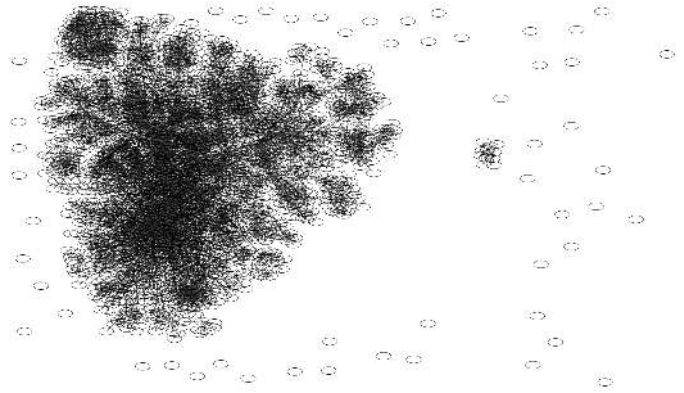### A. *(Step 0) Traditional Dependence Graph Visualization*

Fig. 2: An example of a traditional representation of a system-dependence graph for the Gzip source code. *(Step 0)*

Traditionally, program dependence graphs are seldom visualized, especially for large programs and at a fine-level of granularity (e.g., the statement level). Tools and developers seldom visualize such graphs primarily due to their inability to scale, both in terms of computational time and in terms of human interpretation of a large and complex graph.

The difficulty in allowing large, fine-grained graphs to be visualized comes, in part from the computational expense in providing a useful graph layout. Graph-layout algorithms typically attempt to minimize edge crossings to enable users of the graph visualization to easily visually traverse them. Such edge-crossing minimization accounts for a significant portion of such layout computation [6].

Another difficulty relating to scalability is that the resulting graph visualization is often too large to be represented on a typically computer display or print-out. Nodes are represented as ovals, which often contain text to allow the user to understand each node's purpose. Despite the best edge-crossing algorithms and their required inordinate amounts of time for computation, for large and complex graphs, edges still cross. The result often resembles a nest of oval nodes and connecting edges such that the developer cannot easily traverse the graph or display the graph in its entirety.

Graph visualizations often employ a layout algorithm such as a force-directed graph (FDG) algorithm [5]. In such FDG algorithms, nodes repel each other due to an *electro-static* repulsion force, and edges bind their incident nodes with a *spring* force. The initial position of each node is randomly

assigned on a Cartesian coordinate system of a given size. Then, iteratively, each node is moved to lower its affecting forces: away from other repelling nodes (due to the node repulsive forces) and toward other connected binding nodes (due to the edge spring forces). The state of the layout iterates until the graph has found a local minimal overall energy and the layout reaches a stasis.

Figure 2 demonstrates a resulting graph visualization of the system-dependence graph for the same Gzip program from the running example first presented in Section II. In this visualization, each statement in the program is represented by a node which is depicted by an oval. Dependencies between statements are depicted as lines between their corresponding nodes. This naïve graph visualization demonstrates the difficulties that a user would likely face if attempting to interpret the program, its elements, and their relationships.
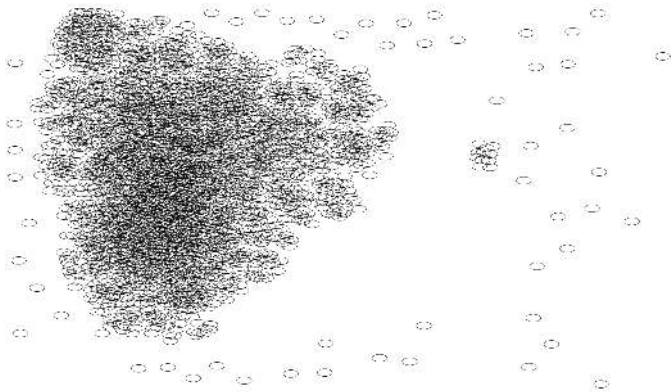
B. (Step 1) Omit Edge Visualization

Fig. 3: Edges are omitted by default to enable greater scalability. *(Step 1)*

As a first refinement step to allow greater scalability of visualizing traditional dependence graphs, we propose to omit the rendering of edges by default. The choice to do so may be inapplicable to some uses of the visualization, however in cases where not all edges need to be visualized, all the time, we consider this a useful and enabling step. The omission of the drawing of these edges has two beneficial effects on scalability: (1) graph layout algorithms need not optimize for minimizing edge crossings and (2) users will be not be overwhelmed with all edges at all times. Edges inform node positions in the graph layout in their traditional ways (described in Section III-A) despite not being rendered by default. We allow the edges which are incident on a particular node to be drawn through user interaction with the interface to allow for exploration. However, by default, edges are hidden, and their crossings do not need to be minimized. While the edges are removed generally, they can be observed by interacting with the visualization.

Figure 3 demonstrates the same program as as our running example. Notice that the graph is less cluttered, although it is quite difficult to interpret given the large number of nodes.

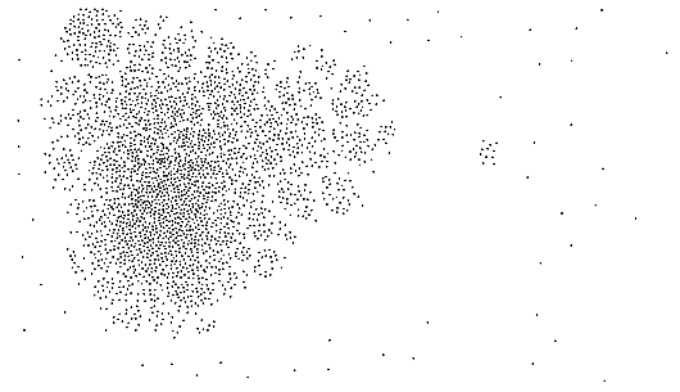### C. (Step 2) Represent Nodes as Points

Fig. 4: Nodes are represented as a single point to enable greater scalability. *(Step 2)*

Our second refinement step to allow greater scalability of visualizing dependence graphs is to represent each node not as an oval containing text which describes it, but instead to represent each node as a single point or dot. In doing so, more nodes are able to be visualized with less clutter. However, the identifying information which was previously contained within the node label is lost. We overcome this through interactivity of the visualization which will be described in Section III-G.

Figure 4 demonstrates the effect of Step 2 on the same program depicted in Figures 2 and 3. While proximity of nodes indicates relationships, edges can be observed through interaction, and node identity can be queried through inter-action (described in Section III-G). In this visualization, the graph scales well, but may be difficult to interpret without the assistance of problem-specific information.

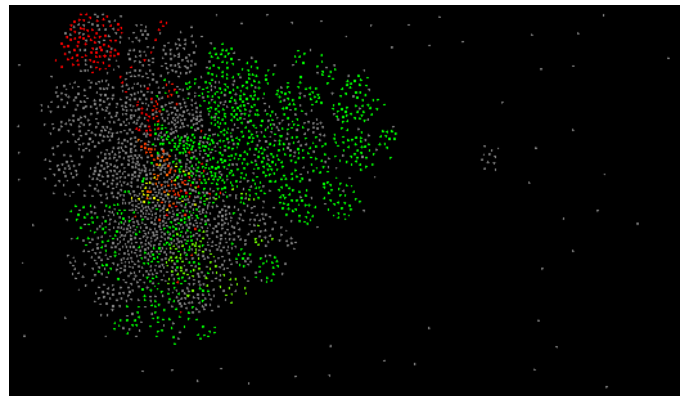### D. (Step 3) Color to Encode Problem-specific Meta-data

Fig. 5: Color encodes problem-specific meta-data. *(Step 3)*

To help the user of the visualization interpret its results, we encode specific problem-specific meta-data using color. We color nodes according to properties attributed to the

statements that they represent. For example, for automated fault-localization techniques, much like the TARANTULA technique [9], red can be used to encode suspicious statements in the program and green to encode non-suspicious statements (with a continuous spectrum going through yellow in between). Alternatively, slices can be visualized using color, code authorship (with each developer assigned a different color), and many other possible uses of color. Eick *et al.* described a number of uses of color for individual program statements [4]. We provide more such example uses of color in Section III-H and the potential uses of viewing these along with their related and dependent statements.

Figure 5 demonstrates the effect of coloring nodes according to Step 3. This visualization depicts the same program as our running example. The color in this visualization is informed by dynamic, run-time information and the TARANTULA fault-localization technique. Notice that now, the most suspicious statements (i.e., *red* statements, which number over 200) produce a loose cluster in the upper, left-hand corner. Notice that the equivalent SeeSoft visualization from Figure 1 depicts these same red statements as scattered throughout the program.

E. (Step 4) Inform Edge Strength with Dynamic Information
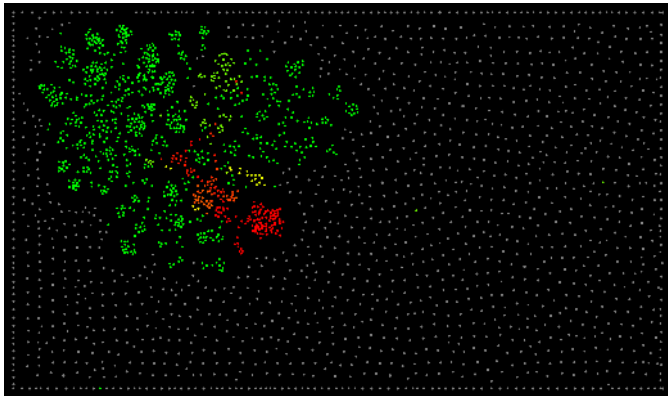

Fig. 6: Dynamic information used to influence node positions. *(Step 4)*

To further help the user interpret the results and patterns found in the visualization, we utilize dynamic, run-time information to influence node positions. In this way, we want to highlight and give prominence to dependencies that are realized during actual execution. In each of the visualizations thus far (Steps 0–3), the graph being visualized represents a static dependency graph. That is, each dependency (which is represented visually as an edge) represents a dependency that *may* occur in *any possible* execution (and these are often over-approximations of dependency possibilities). By enhancing dependencies that are realized often, and diminishing dependencies which are seldom or never realized, we hope to better allow users to identify clusters or patterns in the resulting visualization.

To do so, we assign each edge a weight which influences its spring force. The weight is informed by lightweight

and commonplace statement-coverage information. Using such statement-coverage information, we calculate, for each edge (dependency), the set similarity of the set of test cases which executed each of its incident nodes (statements). We use the Jaccard set-similarity metric to assess the similarity and thus inform the edge weight. We use this as a lightweight approximation of the likelihood of dependency traversal.

Figure 6 demonstrates the effect of utilizing dynamic information to inform edge strength which influences the force-directed graph layout algorithm. Notice that the suspicious (red) nodes cluster to a greater degree than the static-only layout presented in Figure 5. Another interesting quality of this visualization is that the unexecuted statement nodes (colored gray) have no spring force and are thus free to "float away." The executed statements are attracted to the statements that are most often executed with them.

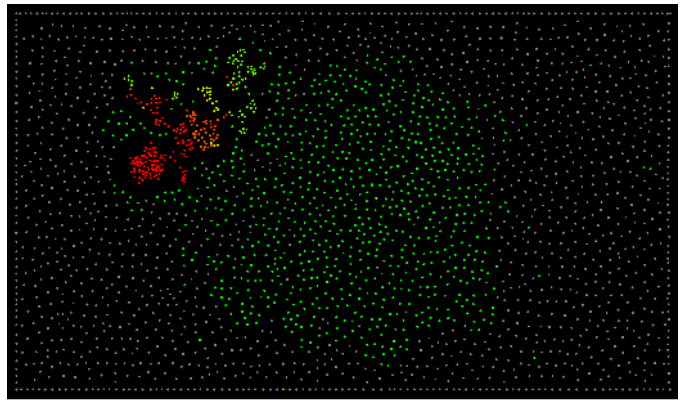*F. (Step 5) Bias Dynamic Information*

Fig. 7: Dynamic information is categorized and differentially utilized to inform node positions. *(Step 5)*

To tailor the visualization further for the specific problem domain, we next differentiate the treatment of dynamic data informing the edge strengths. In our running example of showing fault-localization information with its context, we may choose to treat passing and failing information differently. Because in this example domain the user is most interested in failure-related behavior, we can give a greater weight to failing test-case traversals than passing.

Other problem domains may require a different differentiation of test-case execution information. For example, if attempting to highlight and locate all cross-cutting functionality that relate to a particular feature (say, the GUI) those test cases that exhibit that behavior can be treated more strongly. Or, in another example, test cases which exhibit poor performance may be used to highlight and cluster code that may give rise to such performance issues.

To attribute edge strength with *biasing* (i.e., differentiation or preferencing), we compute the set similarity in the same way as described in Step 4, however, we treat each subclass of test cases separately. Each subclass's similarity is scaled to

the appropriate proportion as desired, and summed to produce the overall edge strength.

Figure 7 demonstrates the effect of utilizing biased dynamic information to inform edge strength. In this figure, failing test-case similarity accounts for 98% of the total edge strength and the passing test-case similarity accounts for just 2%. We can observe that the (red) failure-correlated code — consisting of over 200 lines — clusters to an even greater degree than before, while the (green) passing-correlated code is allowed to float away.

G. Supporting Interactive Exploration

Because the Constellation visualization provides strengths such as enabling relationships, patterns, and clusters of code to be observed, yet has the limitation of not displaying the code in a way that is familiar to the developer, we envisioned its use in concert with the SeeSoft view. While both are scalable, the SeeSoft view has, in some ways, opposite strengths: it displays code in a way with which developers should be familiar. By incorporating these two visualizations, plus a standard full-size source-code listing, the benefits of each can be had.

We implemented a three-paned view which incorporates all three of these views of the software: (1) the Constellation visualization, (2) the SeeSoft visualization, and the (3) the full-sized source-code view. Each of these enables *brushing* [10] with the other two. For example, a node can be clicked in the Constellation, and the appropriate code is loaded and highlighted in the other two views.

H. Applications

Much like the SeeSoft visualization from which we were inspired by its use of color to encode problem-specific meta-data, the Constellation visualization accommodates a plethora of applications such as assessing code authorship, evaluating system modularity, and performing feature searching. To give the reader a sense for such possibilities and for how the new visualization can be useful in these contexts, we describe a few such applications here.

Code authorship: Many development teams contain multiple developers that are working on the same code base. In these circumstances a project manager benefits from knowing the author of a section of code. The Constellation visualization would take as input developer authorship data (presumably gained through a repository) — each developer may be assigned a distinct color. The Constellation visualization then colors and clusters the system based upon dependency data and (if available) run-time data, allowing users to identify which sections of code were created by which developer. Additionally, through the use of the visual clustering, developers who have code that depends closely upon each other can be made aware of those close relationships, potentially informing collaborative maintenance efforts.

System modularity: Similar in aim to work by Kuhn *et al.* [12], the clustering of system dependence nodes based on structure and execution may enable developers to become aware of prominent features and their modularity in the code.

Tracking the coupling and cohesion of a system between lines of code within different classes and packages can inform developers of crucial system design decisions and potential maintenance problems. The Constellation visualization could take class and package hierarchy information as input and color statement nodes accordingly. It then could cluster the system based upon dependency data and (if available) run-time data, allowing developers in the early stages of design to understand more explicitly how the system is evolving, which classes and components across class and package boundaries form cross-cutting functionality. These revealed features may inform potential future areas to refactor the code.

Feature search: If a developer were interested in identifying all code that related to certain functionality, and test cases exist that specifically target that functionality, a technique very similar to the fault-localization example presented in Sections II and III could be utilized. In such an application, the test cases that test for that functionality could be correlated with statement execution. The statements could be colored to indicate the degree to which they are correlated with the feature of interest. Additionally, the dynamic information that informs edge strength can be treated separately according to whether it is from the test cases which test the feature of interest. Even if the code that implements the feature is scattered among multiple parts of the program, the visualization is likely to cluster this code, revealing related and relevant code, and also revealing feature-correlated code that does not cluster, which may indicate duplicated functionality (i.e., clones). Additionally, the visualization is likely to not only expose the code that is directly feature-related, but also reveal code that provides the context for it.

IV. User Experiment

The goal of our study was to assess the degree to which relevant context is encoded into our visualization, as well as the traditional SeeSoft visualization. We included the SeeSoft visualization because it is also a scalable program visualization that works at the statement level and encodes problem-specific meta-data using color. We wanted to assess *whether the code structure as implemented by the developers was sufficient to encode such context*. Our choice of application for the visualizations is the same as that of our running example presented in Sections II and III: the Tarantula fault-localization technique.

We describe our user study by presenting first our object programs, then our experimental variables, next our experimental setup, and finally our results and analysis.

A. Objects of Analysis

In this experiment we used four C-language subjects: Flex version 2.5.4, Gzip version 1.0.7, Sed version 3.02, and Space. All four subjects, are real world programs: Flex, Gzip, and Sed, are Unix utility programs; while Space is an interpreter for an array definition language created by the European Space Agency. These programs vary in size with Flex comprising 15895 LOC, Gzip 9251 LOC, Sed 11699 LOC, and Space

6445 LOC. All four subjects were downloaded from the Subject Infrastructure Repository (SIR) [3] together with their faulty versions and test suites. Like Reference [2] mutants are utilized to create a sufficient fault base and replace faults when SIR's faults cause no failures. Our mutant creation adheres to the framework Offut *et al.* establish in [13] — employing random mutant operators at random positions in the code. After our mutation, each subject program has at least 20 different single fault versions. In all, we produced over 300 visualizations: at least one Constellation and one SeeSoft visualization per fault per object program.

B. Experimental Setup

Our experimental subjects were 30 information and computer science students at the University of California, Irvine. These students varied in degrees of expertise from 1 to 20 years of software-development experience.

We began by creating a visualization for every faulty version from each program. To create the visualizations, we injected each fault, one at a time, into its object program, ran its test suite, gathered execution coverage information, and computed the TARANTULA suspiciousness metric for each source-code statement. For the Constellation visualization, we utilized the CodeSurfer tool by GrammaTech [7] to produce the system-dependence graphs.

Then, random visualizations were selected (without replacement until all versions were used) from the SeeSoft and Constellation visualizations. Each participant viewed five SeeSoft and five Constellation visualizations. The subjects were informed about the basic layout of each visualization and the colored meta-data. Each participant identified a location where they believed the fault to be located according to the visualization.

To evaluate the effectiveness of the conveyed context, we determined the total lines of code which would need to be inspected starting from the subject-prescribed fault location to find the fault. Because each visualization demonstrates information differently, we use the following two methods to evaluate them. For the Constellation visualization, we employ increasingly large concentric circles centered at the subject-prescribed fault location until the fault is circumscribed. We then assessed the quantity of nodes in that circle. This method is chosen because it mimics a breadth-first traversal, which fits well with Constellation visualization's graph structure. For the SeeSoft view, we started at the subject-prescribed fault location and expanded the context in both the forward and backward directions in the source file until the fault was found. This method is chosen because it utilizes the traditional code structure which is the foundation on which SeeSoft is built. Additionally, it *enables us to determine whether relevance is already encoded by developer choice of code location*, and thus whether the additional Constellation visualization is needed at all. Our goal is to most nearly approximate a structural traversal or intended use.

(a) Search results for Flex. (b) Search results for Gzip.

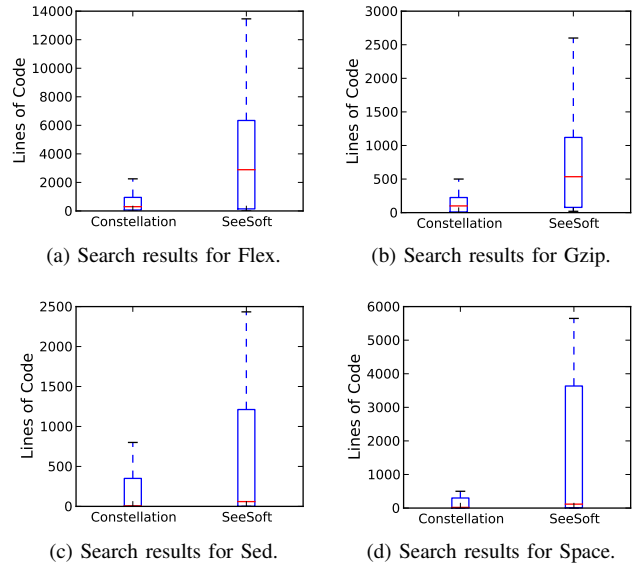(c) Search results for Sed. (d) Search results for Space.

Fig. 8: Number of statements which need to be explored until the fault is found, for each program and visualization type.

C. Results and Analysis

The results of our user study are presented in Figure 8. Each subfigure displays a boxplot which depicts the lines of code that needed to be examined for each subject with both visualizations. Each figure shows the Constellation visualization in the left box and SeeSoft in the right box. Our data shows that in every program, the Constellation visualization enables developers to locate the fault with fewer lines of code. This is manifested by a smaller median and standard deviation in the Constellation visualization in every subject. Further, every Constellation visualization maximum requires fewer lines of code than SeeSoft's first quartile — and in two subjects, even fewer than SeeSoft's median.

We noticed that the subjects primarily selected a red location for their assessed fault location: within SeeSoft they selected the largest group of continuous red lines, while in Constellation they selected the center of the largest cluster of red nodes. This behavior is expected and (we believe) typical given the implications for fault-localization-coloring data and the structure of each visualization. Thus, when a fault was colored yellow or green, the subjects unsurprisingly never guessed that line as the fault. However, the Constellation visualization typically placed those statements in close proximity to the subjects' guess, which was red. SeeSoft often depicted these lines in distant locations from the red lines.

From this observational data and Figure 8, we draw the following conclusions:

1) While SeeSoft is successful for many tasks, it often does not effectively convey relevant context. Suspicious code (those colored red) are often in disparate locations. This result implies that the developers did not structure the code in a way to consolidate relevant context, and thus methods to extract such context may be useful.

2) The Constellation visualization is able to transmit context quite effectively. Disparate locations are associated with dependency and run-time data and are clustered together in Constellation view.

3) When the suspicious lines of code do not include the fault, the suspicious code is often in close structural proximity and directly related to the fault in a run-time execution. We expect a developer to build up an understanding of the fault context as they examine the associations around their selected location.

4) The Constellation visualization has a small standard deviation (as manifested by its quartile locations). The small standard deviation indicates a consistent proximity across all faults in all of our object programs.

V. Threats to Validity

A difficulty in creating external validity for our study is generalizability. We only have 30 study participants, all from the University of California, Irvine, and therefore are unable to draw conclusions about developers worldwide. However, because the choices by these subject follow intuition, and due to the large variety of developer experience, and that developer results are similar across all four different subjects and across two different visualizations, we believe it likely that other developers will behave similarly.

A difficulty in creating construct validity is how we measure context in each visualization. We assume that developers will use each visualization in a certain way — essentially to traverse each visualization based upon the layout of the visualization — but without a study of developers using these tools across multiple applications, we cannot verify this assumption. Notwithstanding this, we believe that measuring each visualization this way — focusing on its structure and purpose — is a reasonable way to determine the strengths of each visualization's layout and necessity.

VI. Conclusions and Future Work

Our paper presents a novel visualization — the Constellation visualization — which enables a system-wide view while simultaneously enabling statement-level analysis, reveals crosscutting functionality, enables clustering through structural dependencies and run-time data, and uses problem-specific metadata applicable to a number of software-engineering tasks. This visualization can empower developers — in a wide range of applications — to better understand their system's context and visualize meta-data patterns. We also describe our prototype tool which implements our visualization.

We present a user study which evaluates our new framework and a traditional visualization. Each visualization is evaluated for how effectively it enables a developer to understand their system's meta-data context. This study found that our new visualization is effective at revealing relevant context. We believe that existing visualizations such as SeeSoft to be effective at presenting the code in a familiar way to developers, while neglecting relationships between code elements. As such, we think that our new visualization to be a desirable compliment to such existing visualizations.

In the future, we expect to perform a user study to evaluate how effectively developers can use this tool in different stages of software development, along with assessing which aspects of the visualization are the most beneficial and which are not. We also intend to perform qualitative studies evaluating user experience in interacting with the tools that implement our visualizations for a variety of software-engineering tasks. Lastly, we will implement more interactive exploration techniques to better enable developer search.

VII. Acknowledgments

References

[1] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings of the Symposium on Software visualization*, SoftVis '08, pages 91–94, New York, NY, USA, 2008. ACM.

[2] N. DiGiuseppe and J. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 9th ACM/IEEE International Symposium on Software Testing and Analysis*, ISSTA '11, page To Appear, New York, NY, USA, 2011. ACM.

[3] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 60–70, Washington, DC, USA, 2004. IEEE Computer Society.

[4] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18:957–968, November 1992.

[5] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[6] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, March 1993.

[7] GrammaTech, www.grammatech.com, Ithica, NY. *CodeSurfer*.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12:26–60, January 1990.

[9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[10] D. Keim. Information visualization and visual data mining. *Visualization and Computer Graphics, IEEE Transactions on*, 8(1):1 –8, jan/mar 2002.

[11] J. Krinke. Visualization of program dependence and slices. In *Proceedings of the International Conference on Software Maintenance*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[12] A. Kuhn, D. Erni, and O. Nierstrasz. Embedding spatial software visualization in the ide: an exploratory study. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 113–122, New York, NY, USA, 2010. ACM.

[13] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5:99–118, April 1996.

[14] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, 1984.

[15] T. Würthinger, C. Wimmer, and H. Mössenböck. Visualization of program dependence graphs. In *Proceedings of the Joint European Conferences on Theory and Practice of Software*, CC'08/ETAPS'08, pages 193–196, Berlin, Heidelberg, 2008. Springer-Verlag.