Revealing Runtime Features and Constituent Behaviors within Software

Vijay Krishna Palepu and James A. Jones
Department of Informatics, University of California, Irvine, USA
{vpalepu, jajones}@uci.edu

***Abstract*—Software engineers organize source code into a dominant hierarchy of components and modules that may emphasize various characteristics over runtime behavior. In this way, runtime features may involve cross-cutting aspects of code from multiple components, and some of these features may be emergent in nature, rather than designed. Although source-code modularization assists software engineers to organize and find components, identifying such cross-cutting feature sets can be more difficult. This work presents a visualization that includes a static (*i.e.,* compile-time) representation of source code that gives prominence to clusters of cooperating source-code instructions to identify dynamic (*i.e.,* runtime) features and constituent behaviors within executions of the software. In addition, the visualization animates software executions to reveal which feature clusters are executed and in what order. The result has revealed the principal behaviors of software executions, and those behaviors were revealed to be (in some cases) cohesive, modular source-code structures and (in other cases) cross-cutting, emergent behaviors that involve multiple modules. In this paper, we describe our system (CEREBRO), envisage the uses to which it can be put, and evaluate its ability to reveal emergent runtime features and internal constituent behaviors of execution. We found that: (1) the visualization revealed emergent and commonly occuring functionalities that cross-cut the structural decomposition of the system; (2) four independent judges generally agreed in their interpretations of the code clusters, especially when informed only by our visualization; and (3) interacting with the external interface of an application while simultaneously observing the internal execution facilitated localization of code that implements the features and functionality evoked externally.**

I. INTRODUCTION

Software programmers decompose their program code into explicit components and modules to enable scalability of the codebase, reuse of modules, and comprehensibility of the code design. Early in software-engineering literature (1971), Wirth observed that decomposition and refinement can be performed on both program structures and data structures. In later work, object-oriented design enabled programmers to write code that was principally organized by data structures and entities, rather than the algorithmic decomposition that previously dominated. In other words, object-oriented code today, is principally structured by "actors" rather than "actions."

Because of this style of programming, behavioral features of the software often cross-cut the structural design of the code, and thus may be difficult to find and identify. Software behaviors can be identified by the *runtime* composition and cooperation of objects' interfaces, which makes comprehension of such interactions more difficult. This importance of visualizing runtime interactions is emphasized by Jerding *et al.* [10], in their work on Execution Murals, where they show how global overviews for a software execution provide "immediate insight into different phases of the execution" and enable an execution, specifically execution traces, to be "searched visually".

To help programmers understand the runtime behavior of their code and thus the constituent features within executions, researchers and tool builders have created interfaces and visualizations. For example, traditional symbolic debuggers allow programmers to step through a program execution. Jerding *et al.* [10], De Pauw *et al.* [15], Cornelissen *et al.* [4] and Trümper *et al.* [17], for example, visualize various runtime interactions of code. An alternative approach was taken by Kuhn *et al.* [13] by structuring visual representations of the codebase by the frequently used words (or *concepts*) expressed within the source code. These common concepts may have been expressed in structurally disparate code modules.

The execution visualizations (*e.g.,* Jerding [10], De Pauw [15]) are limited by their ability to represent only a single execution at a time, which hinders their ability to reveal commonly cooperating code subsets that perform various tasks and thus reveal runtime features that are in common across multiple executions. Moreover, the concept-mapping visualizations (*e.g.,* Kuhn [13]) are limited in their ability to reveal which textually-similar modules cooperate (or do not) during runtime to perform tasks.

In this work, we take a different approach. Much like the concept mapping approach of Kuhn *et al.* [13], we represent an overview of the source code. However, instead of the textual similarity of the modules informing the placement of the code representations, we use the runtime cooperation of the code in order to emphasize and reveal flows of execution that perform tasks. Much like the execution visualizations (*e.g.,* Jerding [10], De Pauw [15]), we visualize runtime execution. However, our visualization is principally informed by multiple executions (as many or as few as the user chooses), and as such, provides clusters of commonly cooperating code that perform features and task that are observed across executions.

Our visualization, which we are calling CEREBRO, presents clusters of code elements (in our implementations thus far, these are source-code instructions). The placement and clustering of the code is informed by execution traces to reveal commonly interacting and cooperating instructions from those executions. The visualization provides the ability to interact to

explore and reveal the structural composition of the clusters, and it provides the ability to animate individual executions atop the clustered views in order to discover the sequence of execution behaviors and internal responses to external inputs.

To assess the visualization, we conducted three main evaluations. The first evaluation was a series of case studies of software subjects to determine if we could identify meaningful clusters that represented common functionality. We found that the clusters were not only meaningful, but they also demonstrated how object-oriented design often obfuscates feature-sets that cross-cut the structural design. The second evaluation involved four independent judges of the visualization to determine the quality and consistency of their identified clusters. We found that the four judges identified consistent clusters when presented with only the execution-informed layout, and we observed that when the judges were provided with the structural decomposition their results had slightly greater divergence. The third evaluation involved interacting with the external interface of a software system whilst visualizing the internal code execution to determine if features could be identified and located. We found that (1) different subsections of the codebase were involved in each external feature, (2) multiple clusters were often involved in each external feature, and (3) the common and distinct clusters involved in different external features provided insight into the way in which features were implemented in the code.

The main contributions of this work are as follows:

1) A novel visualization of fine-grained code elements that (a) reveals internal loci of functionality, (b) enables discovery of runtime, internal behaviors across multiple executions, (c) enables discovery of the structural components that are involved and cooperate to perform functionalities, and (d) enables exploration of individual executions in the context of the clusters that are revealed.
2) An evaluation of the CEREBRO visualization and interface that includes four diverse and real-world software subjects that includes both independent judges and analyses of the resulting clusterings.
3) An extention and maturation of our prior work [14], including evaluations, new functionality, and a dissemination of the implementation artifacts that form the CEREBRO visualization.

II. MOTIVATION

Code Design and Decomposition. Programmers can use different methods for decomposing a functional task in code. Wirth referred to the process as *stepwise refinement* and proposed that "Refinement of the description of program and data structures should proceed in parallel." [18] In this early and astute observation, Wirth points out that such refinement occurs for both programmatic tasks and for data structures.

Later, Booch elucidated the benefits of decomposing code primarily according to entities or objects, which he referred to as *object-oriented decomposition*, and then secondarily decomposing programmatic tasks, which he referred to as *algorithmic decomposition*. He states, "The algorithmic view highlights the ordering of events, and the object-oriented view emphasizes the agents that either cause action or are the subjects on which these operations act," [1, p. 20] and moreover, "We must start decomposing a system either by algorithms or by objects and then use the resulting structure as the framework for expressing the other perspective." [1, pp. 22–23]

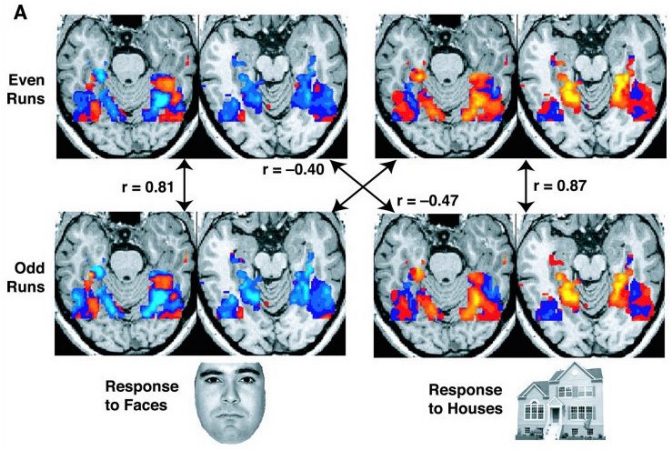


Fig. 1: Example of a Functional Magnetic Resonance Imaging (fMRI) scan of a human brain, which serves as inspiration for our visualization. In the scans, the subject: (left) viewed a face and (right) viewed a house. Red shows activation and blue shows deactivation, which reveals different brain activity for different tasks and stimuli.

Today, many software projects have adopted the object-oriented method of problem decomposition — the classes and entities form the dominant hierarchy that organizes the code. By making this choice, functionality (*i.e.,* "action") within the code is composed of multiple interacting objects (*i.e.,* "actors"), and can be more effectively understood with a codebase recomposition that emphasizes the interactions between "actors.".

To enable such recomposition, a global overview of the codebase is needed to reveal the composition of different subsets of the codebase, and their cooperation across multiple structures. We also posit that recomposing code according to functional cooperation may reveal planned features as well as reveal *emergent* internal behaviors. Such emergent behavior may reveal behaviors that result from planned (but undocumented designs) or may help identify inadvertent misbehavior (*e.g.,* bugs).

Inspiration: Human Brain Imaging. We took inspiration for our approach to visualizing the internal functional structure of the codebase from human brain imaging technology, namely functional magnetic resonance imaging (fMRI). fMRI technology detects and visualizes bloodflow within a living brain to measure and detect brain activity. Cognitive neuroscientists have found that different regions of the human brain are activated by different external stimuli. Figure 1 shows an fMRI scan for a brain of a human subject who was perceiving a photograph of a person's face (left) and for the same subject

who was perceiving a photograph of a house (right) [9].[1]

Following this inspiration, the elements that compose program code should be placed in spatial locations that reduces the distance between interacting elements. Our implementation of CEREBRO uses source-code instructions[2] as the granularity of the program-code element. Following our analogy, an instruction is analogous to a neuron; a cluster of instructions is analogous to a lobe of the brain; the runtime connections between the instructions is analogous to synapses; program input is analogous to perceptual stimulus; and CEREBRO is analogous to an fMRI scanner.

Envisioned Usage Scenarios. We envisage multiple potential uses of the CEREBRO visualization. For example, newcomers to a software project may use such visualizations to understand the system's structure and behavior (*i.e.,* "onboarding"). Another example is to support feature localization by providing the external stimulus to evoke a software feature and observing the internal behaviors that correlate. We also envisage its use in diagnostics to support maintenance for both preventative maintenance and corrective debugging. We can imagine software architects using CEREBRO to perform a functional architectural recovery, or to support detection of unintentional violations of the prescribed architecture (perhaps due to software erosion). Finally, much like how fMRI technology has influenced and informed cognition and neuroscience researchers, we envision that using a visualization such as CEREBRO may inform areas of software research and future visualizations.

III. CEREBRO VISUALIZATION

The "brain" metaphor is a reflection of the data or entities that are presented in this visualization. As an analogy to the idea of the human brain — a set of complex parts working together to respond to real-world stimulus — this visualization represents software systems as individual parts that work in concert to produce outputs for specific external inputs. To elucidate, we address two central questions about the visualization in this section: (a) what data or elements does the visualization present?; and (b) how does the visualization present the data?

CEREBRO presents source code and dynamically observed white-box behavior. Specifically, we represent an entire software system as a composition of individual, constituent source-code instructions that are executed during the working of the system. To reveal how the executing instructions work together, possibly from different parts of the system's structural organization, the visualization also presents the dynamically observed control flows among the instructions and the structural location of the source-code instructions. Figure 2 depicts the CEREBRO visualization, with each of its constituent components, which we will now describe.

Data Source — Software Executions. We monitor software executions, and treat them as our data source, to identify the executed set of source-code instructions and the flow of the individual executions from one instruction to the next. We represent executed source-code instructions as nodes in a system-wide, dynamic control-flow graph. The nodes in the graph are rendered with a force-directed layout, based on the frequencies of execution flows, between source-code instructions, during any number of executions or runs, of the software system in question. The actual visualization is composed of the following elements.

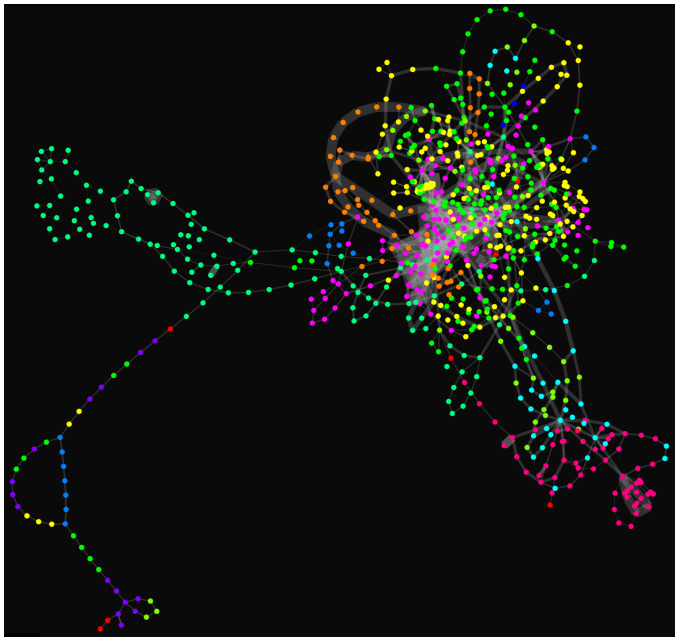


Fig. 2: CEREBRO visualization of the NANOXML program. Nodes represent source-code instructions; edges represent execution-trace control-flow; clusters of nodes represent commonly cooperating instructions; and node color represents the object-oriented class in which the instruction belongs.

Nodes — Source-code Instructions. The nodes of the graph are visualized as dots, each representing a source-code instruction that was executed, in at least one run of the program. Each node is annotated with its respective source-code instruction's line-number and the names of the instruction's owner-class and -method. The annotations for a node becomes visible when a mouse pointer is hovered over it.

Additionally, nodes are colored by the source-code class to which their corresponding instructions belong. A large number of colors are often required in the presence of a large number of classes in a software system. Such a situation makes it difficult to identify different shades of the same color that are assigned to different classes; thus making it difficult to visually distinguish between classes. To overcome this problem, nodes that represent source-code instructions from the same package hierarchy are colored with different shades of a common color. This scheme of coloring ensures that classes from the same package hierarchy are assigned similar colors. Thus, allowing the identification of the common structural hierarchy of source-code, if not the exact class.

[1]The image in Figure 1 is public domain and used under the Creative Commons Public Domain declaration.

[2]In this work, a "source-code instruction" refers to an actual source-code line in a software codebase, unless mentioned otherwise.

As such, annotations and colors for nodes, which reveal the structural organization of the source-code, enable the identification of dynamic behavioral features as cohesive or cross-cutting through the spatial layout of the nodes, which is dynamically informed.

Force-Directed Layout — Cooperating Instructions. The positions of the nodes in the visualization that represent executed source-code instructions, are informed by a force directed layout. The force-directed layout is composed of repulsive and binding forces between nodes. These forces influence the final positions of all nodes in the graph after the layout algorithm is allowed to iterate and stabilize. A constant repulsive force among nodes causes all nodes to repeal each other. Simultaneously, a dynamically observed flow of execution from one source-code instruction to another, establishes a binding force between the corresponding nodes in the layout. Further, the binding force between the two nodes is positively weighted by increasing frequencies of such execution flows between source-code instructions — these frequencies are gathered both within an execution (*e.g.,* a block of code executed within a loop, or a method called multiple times) and across executions (*e.g.,* multiple runs that each invoke the same execution path). Such a dynamically informed layout, forces nodes to be closer when the representing source-code instructions exhibit higher frequencies of execution flow between them — indicative of greater runtime cooperation between such instructions. We refer to such a layout of nodes, for a single system, as the "brain" of the software system.

Edges — Execution Flows. An edge between two nodes represents dynamically observed flow(s) of execution between the two source-code instructions represented by the two nodes. The edges in the visualization are optionally rendered, but regardless of whether they are drawn, they inform the layout of the nodes to place cooperating nodes near each other. When rendered, the thickness of edges is directly proportional to the frequency of traversals of the represented execution flow, across all executions. As such, edge thickness is an indicator of the weights of the individual binding forces that inform the final layout of the visualized nodes.

Animation — Execution Enactment. In addition to the inanimate elements of the visualization, *i.e.,* nodes, layout and edges, CEREBRO also facilitates the animation of software executions, themselves. An execution is animated in the visualization by highlighting the nodes for instructions in execution order. The highlight is accomplished with a white, larger-diameter dot for the currently executing instruction(s). Animating multiple concurrent threads is also supported to enable observation of how each thread's execution interacts. Finally, the animation can enact either a pre-recorded execution trace or a live execution that is occurring simultaneous to, and alongside, the visualization.

IV. APPROACH

Given our discussion of how CEREBRO presents the interaction between executed source-code instructions, it is important

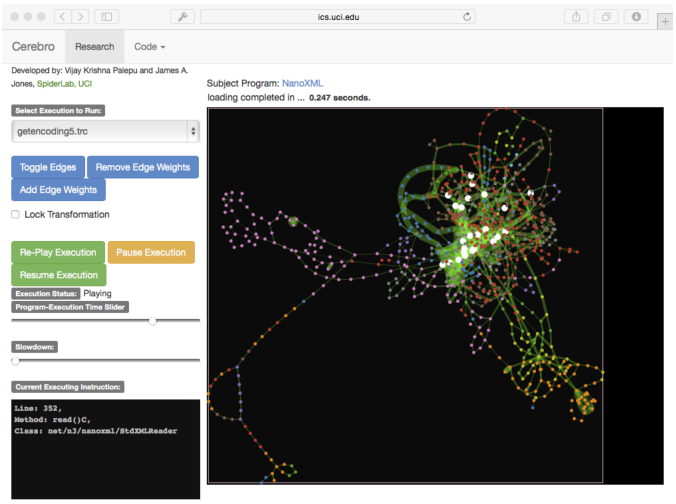


Fig. 3: Screenshot of the CEREBRO visualization tool running in a web browser.

to note the non-trivial mechanics to collect, recompose, and finally present such data. In this section, we will provide a brief overview on the principal parts of our approach, namely — (a) collecting instruction execution traces for multiple system executions; (b) recomposing execution traces to inform the force-directed layout of executed source-code instructions from their execution flows; and (c) visualizing the force-directed source-code instructions and the executions — either live or recorded — that flow through the instructions.

Collecting Execution Traces. As a first step to our approach we need to monitor the executions of the software system that we intend to visualize using CEREBRO. The monitoring of program executions involves instrumenting the executable binaries of the software system with probe instructions that relay data about each executing source-code instruction during the software system's execution. Such instrumentation probes capture and relay the necessary data about a specific source-code instruction, just before its actual execution. We refer to a stream of executing source-code-instruction data, as relayed by the instrumentation probes, as an *execution trace*. Such execution traces serve a dual purpose: to derive the execution flows to inform the force-directed layout of the execution instructions, and to visualize the execution itself.

Composing Force Directed Layouts from Execution Flows. The first step towards computing the force-directed layout is to identify the executed source-code instructions and the flows of execution between them. We scan the recorded execution traces to identify the set of source-code instructions that are visualized as nodes. Simultaneously, we record the flow of execution from one instruction to the next, along with their frequencies, by observing the execution order of the source-code instructions in the execution traces. We establish an execution flow between two source-code instructions, if the execution instance of one instruction was immediately followed by the other, for each thread.

We treat the executed source-code instructions and the exe-

cution flows that connect them as nodes and directed-edges in a directed graph. We use the frequency of the execution flows as weights to the corresponding directed-edges in the graph. We then use the graph as the input to a force-directed graph-layout algorithm, which employs a constant repulsive force among the nodes and binding forces between the connected nodes, to render the final layout of the nodes in a two-dimensional space. The resulting layout renders node clusters and patterns, as shown in Figure 2, which are indicative of how the groups of source-code instructions interacted with each other via the flow of execution, often towards executing a common feature of the underlying software system.

It is worth noting that we often analyze multiple execution traces to inform the set of executed source-code instructions and the flows between them – within and across individual executions — to generate the force-directed layout. Despite our ability to use multiple executions, our approach can also work with a single execution trace, to generate the force-directed layout of the executed instructions within the execution. However, it is worth noting that potentially a wider range of node clusters and patterns, *i.e.,* behaviors are revealed when using more than a single execution.

Visualizing Source-Code Instructions and Executions. As a final step in our approach, we visualize force-directed layout of nodes and edges that represent the executed source-code instructions and their execution flows, respectively. Aside from using execution traces to compute the force-directed layout of the nodes and edges, we also use execution traces to visualize the executions themselves through the means of animating the nodes in the order of execution of their corresponding source-code instructions. Further, we support the visualization of live executions by supplying the execution trace obtained from a live execution, directly into the visualization, without necessarily persisting the execution trace to disk. That said, a prerequisite of our approach to visualize a live execution of a software system, is the *a priori* generation of the visualization layout of the software system, which can be informed by a set of *training* executions.

Interactivity: Pan and Zoom, Execution Progress, Selection. In addition, we present the visualization within the context of an interactive user interface that enables the exploration of the nodes and edges, and the source-code executions that they represent. The vectorized form of the visualization allows zooming and panning to focus on specific areas and movement among them. The progress of an execution animation, from a prerecorded trace, can be viewed and controlled with a slider. Finally, regions of nodes within the visualization can be selected with a free-hand lasso, which produces a summary of the selected instructions in an accompanying pane of the visualization.

Implementation. We implemented our approach, *i.e.,* the execution-tracing, computation of force-directed layout, and the interactive visualization, as three different components.

Source-Code Execution Tracer. We implemented a source-code instruction-level execution tracer that we call BLINKY. BLINKY is a Java bytecode instrumentation framework that is accompanied with an online profiler for dynamic collection and relay of execution traces to other systems.[3] BLINKY is primarily based on the ASM bytecode re-engineering framework [3]. We analyze the execution traces from BLINKY to identify the executed source-code instructions and the execution flows between them to generate a force-directed layout.

Force-Directed Layout. We compute the force-directed layout of nodes and edges in our visualization with an implementation of the Frutcherman-Reingold force layout algorithm [8], as implemented in the GraphStream dynamic graph manipulation and processing framework [7].

Interactive Visualization. We implemented the interactive visualization for rendering the force-directed layout of executed source-code instructions with a web-based HTML5 application that relies on SVG for rendering the visualization in a vectorized format. Figure 3 shows a screenshot of the CEREBRO interactive visualization, running in a web browser. Our implementation of the interactive visualization makes substantial use of D3.JS — a data-driven, declarative Javascript library to create and manipulate interactive, web-based graphics [2]. Implementing the entire visualization with web-based standards and technology allows our implementation of the visualization to be portable and inter-operable across different platforms.[4]

V. EVALUATION

We employed our implementation of CEREBRO to address three main research questions.

RQ1: Does the visualization assist in revealing common software functionality across the structural organization of source-code instructions?
RQ2: Does the visualization aid in consistently identifying clusters of source-code instructions?
RQ3: Does the visualization enable localization of external software features during live monitoring of execution?

We ask **RQ1** in an attempt to understand the extent of overlap, between the algorithmic decomposition of software systems, which is largely influenced by the "actions" of the software system, as against the structural organization of source-code instructions that is often focused on the "actors", like in object-oriented code design. **RQ2** is designed to study the quality of code clusters that are revealed by the CEREBRO. We are specifically trying to understand the consistency with which four independent judges identify similar sets of clusters. Finally, we ask **RQ3** to study if CEREBRO helps in identifying, when and where different software features are enacted during the execution of a software system.

[3]BLINKY is available as an open-source project at https://github.com/spideruci/blinky

[4]The open source implementation of the interactive visualization along with its live demo is accessible at http://spideruci.github.io/cerebro.

Experimental Subjects. In order to carry out our investigation we used our implementation of CEREBRO to visualize the software executions of four real-world subject programs that provide non-trivial functionalities to their users. The software programs of our evaluation are — NANOXML (>2,600SLOCs), JPACMAN (>3,000SLOCs), JAVAC (>70,000SLOCs), and JEDIT (>100,000SLOC).

Qualitative Case Study.

In order to answer **RQ1**, we carried out a series of case-studies, where we created the CEREBRO visualizations for multiple executions of NANOXML, JPACMAN, and JAVAC, and tried to identify the major functionalities exhibited by executions of those systems. We identified software functionality in each system by investigating how various source-code instructions were clustered together in the layout, thus indicating cooperation between the instructions of a cluster towards common features or "actions". We labeled each cluster that we recognized with a natural-language phrase that best described to us, the common functionality or "actions" that they perform. We gained an understanding of the common functionality of each cluster, and thus their natural-language label, by manually investigating the range of classes and methods that owned a cluster's composite source-code instructions. The natural language descriptors within the names of such classes and methods helped us better understand the purpose of such instruction clusters. In addition, we explored the systems' executions by replaying recorded execution traces to identify the order in which instructions clusters *i.e.,* software functionalities were enacted — thus revealing potential interactions between functionalities and in effect pointing to higher-level features of the software systems under study. The findings of our exercise in identifying various high-level features of the three software systems are depicted in Figure 4.

Our goal was to study the consensus or divergence between the functionality exhibited by the systems and the cooperation of source-code instructions towards those functionalities from across various parts of the structural hierarchy of those systems. The structural hierarchy of the source-code instructions is conveyed by the color of the nodes in the visualization that indicate the range of object-oriented classes that the constituent instructions of an annotated cluster belong to. We recognize a consensus between the action-oriented features and the object-oriented classes when an annotated cluster is dominated by the nodes, *i.e.,* source-code instructions, from a single class — recognized by a dominant shade of colors. Similarly, we identify a divergence between the features and the classes, when an annotated cluster is constituted by source-code instructions from multiple different object-oriented classes — recognized by a mixture of different colors.

Instances of Consensus. As shown in the figures, the annotated CEREBRO visualizations of the different systems reveal cases of both consensus and divergence between the action-oriented features and the object-oriented structure. Consider the instances of the "start reading input data" cluster in Figure 4a, for NANOXML that is dominated by green nodes — those nodes are representative of the class `StdXMLReader`, which reveals a class whose purpose is action-oriented rather than entity-oriented.

JPACMAN also exhibits cases of consensus in Figure 4b. Consider the "loading sprite image" cluster, dominated by red colored instructions of the `ImageLoader` class and whose modularization is informed by the action that it performs.

There also appears to be substantial agreement in the case of JAVAC, shown in in Figure 4c. Consider the annotated clusters, "generating code" and "writing classfiles" that are colored in purple and blue respectively. Those clusters depict the code generation phase of the JAVAC compiler, and represent instructions from the classes `Code`, `Gen` and `ClassWriter`.

Instances of Divergence. In contrast, NANOXML also exhibits a large cluster annotated as "parsing and reading input data", with nodes of a number of colors. These instructions span multiple classes that are responsible for reading while building the data structures in memory.

Such a trend is observed to a greater extent in the CEREBRO visualization of JPACMAN. In Figure 4b, we particularly identify two clusters — "creating a new game" and "moving pacman and ghosts" — that display a high variety of color compositions. The cluster "creating a new game" is the initialization code of the system that initializes object instances from the entire range of different classes in the system, and is thus expected to cross-cut the structural hierarchy of the system. Similarly, "moving pacman and ghosts" represents the main game-loop of JPACMAN and is thus, a composite of multiple instructions from the various model and view classes of JPACMAN.

The most notable case of divergence in JAVAC emerges in the "compiling parse tree" cluster that exhibits a mixture of different colors. This cluster represents the main compilation phase within JAVAC's execution, and composes with instructions from within the parser, code generation, and various utility classes, aside from the classes in the `com.sun.tools.javac.comp` package that are responsible for the various stages of compilation.

Visual Clustering Study.

In order to answer **RQ2**, we asked four independent judges to identify clusters for different CEREBRO visualizations for the three software systems: NANOXML, JPACMAN, and JAVAC. The independent judges first assessed the composition of visual clusters based on the colorless structure of the graphs, and then reassessed the composition of the visual clusters when color was added. As such, we recognize and note the potential bias that this order of events has: judges may be influenced by the visual structure in the colored version. Thus, the results should be interpreted as such. The scope of this study is limited to how the visual elements (force-layout and color) impact the human-perception of node-clustering.

We initially provided all judges with a colorless rendition of CEREBRO, where all nodes were colored in white. This forced the judges to identify clusters based on just the layout of the nodes with the CEREBRO visualization for each system. As

(a) NANOXML (b) JPACMAN (c) JAVAC

Fig. 4: Software subjects with clusters labeled with the functionalities that they implement.

such, they were not influenced by the structural organization of code. For this study, we call this treatment the *white*-treatment.

We then asked our independent judges to repeat the same task for identifying clusters in the layout of nodes, with the aid of colored nodes — colors representing the location of the code in the structural organization of the systems. We refer to this treatment as the *color*-treatment.

We recorded the contents of the individual clusters as identified by the four judges, for all three software programs and with both treatments — white and color. We next visually compared the consistency with which the judges identified similar node clusters. For each treatment and system, we assigned a judge-identified-cluster with a specific color, and colored all nodes within that cluster with that color — this enabled a side-by-side visual comparison of the constituency of each of the manual clusterings. Given the space limitations we present representative samples of the resulting visual comparisons[5] of the judge-identified-clusters in Figures 5–6.

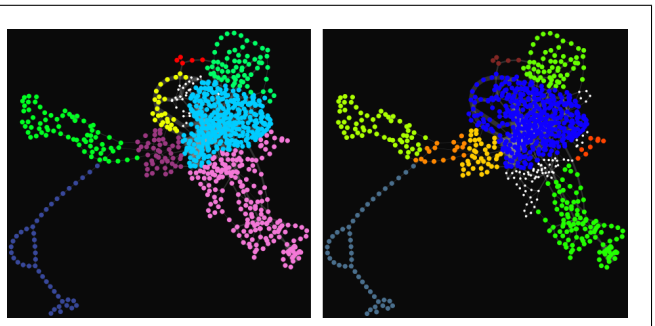
Figure 5 samples the visual comparisons of the clusters identified by the four judges for the *white*-treatment. Each row in Figure 5 shows the visual comparison of the clusters by the judges for a single subject, starting with NANOXML. For instance, Figure 5(b) shows the visual comparison of the clusters by the second and third judges for JPACMAN, with left-hand side showing the second judge's clusters and the right-hand side showing the clusters identified by the third judge. Similarly, in Figure 6(a)–(c) we depict a sample of the visual comparison of the three systems (NANOXML, JPACMAN, and JAVAC) with the *color*-treatment. The colors in both figures simply demarcate nodes from different cluster as identified by the four judges; the colors do not carry any other special meaning for these images. A visual inspection of the comparisons, depicted in-part in Figures 5 and 6, shows a perceptible degree of similarity in the clusters identified by the judges, across the two treatments. Moreover, in most cases of disagreement, a large cluster identified by one judge would be indicated by two or more distinct subset clusters by another judge (the lower part of JPACMAN in Figure 5(b)).

That said, we observe a slightly greater consistency in the clusters identified by the judges in the *white*-treatment, as against that in the *color*-treatment. This is best observed in the case of JPACMAN (Figure 6(c)), where the clusters identified by the judges seem to diverge more than the others. It is interesting to note such an effect with the *color*-treatment, despite the obvious learning effect towards the force-directed layout and against color. However, the reader should note, that the clustering identified by the judges with the *color*-treatment resemble closer to the clusters that we identified in Figures 4a, 4b and 4c, from our initial case studies, where we were informed by the actual execution animations, in addition to the layout and color of each nodes.

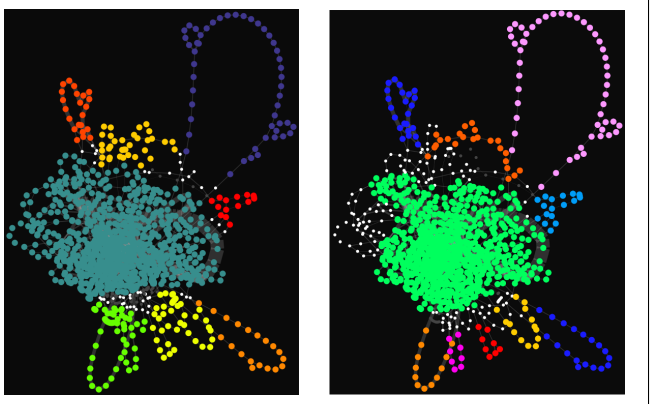
In summary, we found that for our three subjects and four judges, there was a general consistency of cluster identification and composition. And although introducing structurally informed colors for the nodes caused a small degree of disagreement, those disagreements actually appear to be in harmony with the clusters that we identified, informed in addition by the execution-replays.

External Stimuli to Internal Response Causal Study. We answer **RQ3** by studying the animation of nodes for the software system JEDIT by interacting with the external interface of JEDIT and visualizing its internal response in real-time. For the purpose of this study we created a simple scenario: open a new file by using JEDIT's file-browser, type changes to the file, perform a textual find-and-replace to fix an error in the text, before terminating JEDIT. We wanted to identify

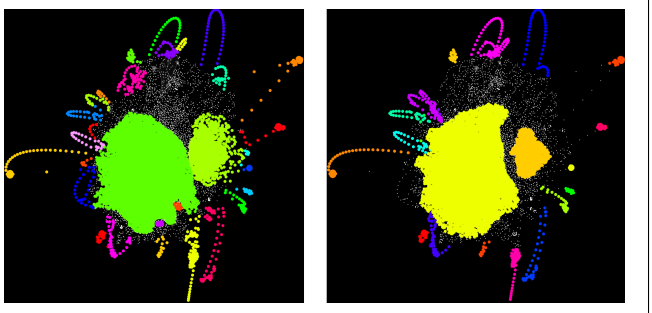
[5]The entire set of visual comparisons is made available at http://spideruci.github.io/cerebro/data



(a) Clusters identified by Judges 1 and 2 for NANOXML

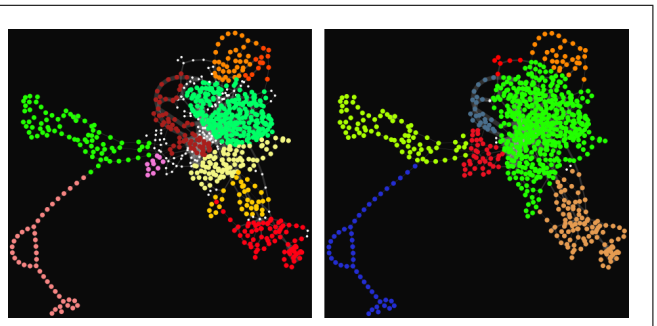


(b) Clusters identified by Judges 2 and 3 for JPACMAN

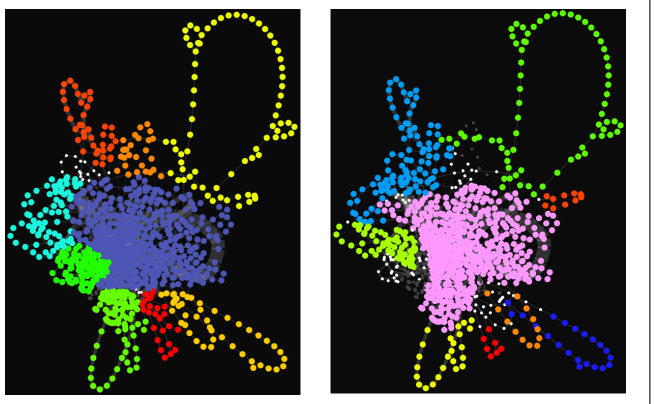


(c) Clusters identified by Judges 3 and 4 for JAVAC

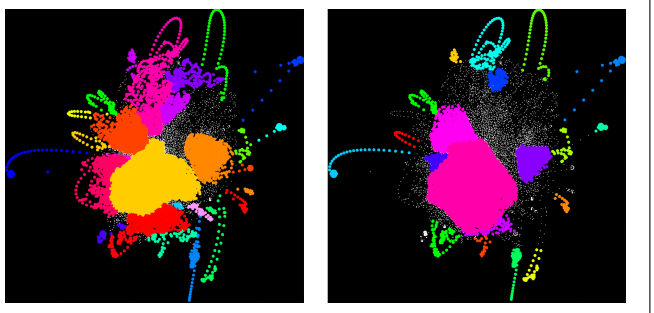
Fig. 5: Samples of visual comparison of clusterings identified by four independent judges with *white*-treatment.



(a) Clusters identified by Judges 1 and 2 for NANOXML



(b) Clusters identified by Judges 2 and 3 for JPACMAN



(c) Clusters identified by Judges 3 and 4 for JAVAC

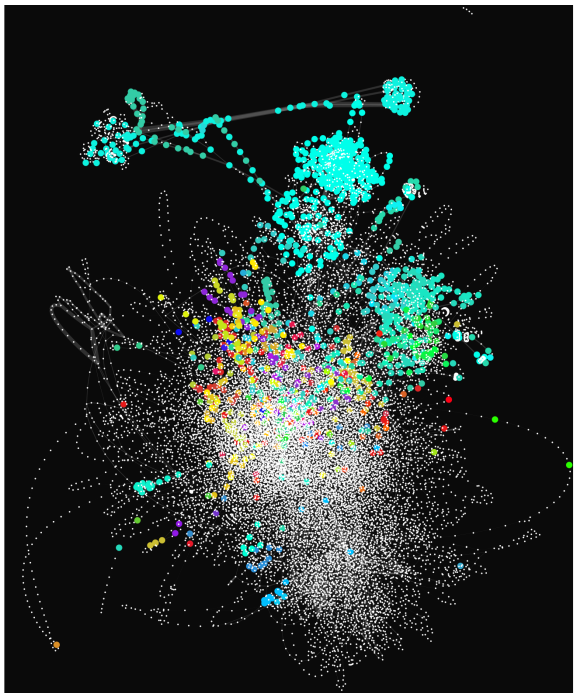
Fig. 6: Samples of visual comparison of clusterings identified by four independent judges with *color*-treatment.

the instructions that were executed for each task separately, so as to locate possible clusters of instructions that support each individual task — thus enabling us to identify the location of the specific features associated with each task.

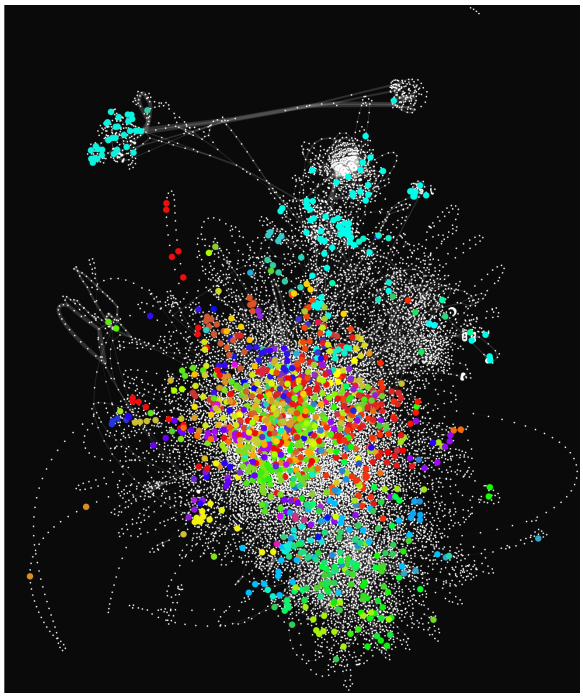
The resulting renditions of the partially colored visualizations, as informed by the four stages of our scenario are presented in Figure 7. Different parts of CEREBRO for JEDIT are highlighted for each individual task. That said, notice that the clustering for "type text" and "perform find-and-replace" are more similar than the other two clusterings. Upon further investigation, we found that the two green clusters for "type text" and "perform find-and-replace" (shown in Figures 7(c,d)) represent classes that work to provide functionality within text area of JEDIT. Moreover, given that the "find-and-replace" functionality in JEDIT renders results in the text editor itself, the similarity between the two clusterings in Figures 7(c,d) may be expected.

VI. RELATED WORKS

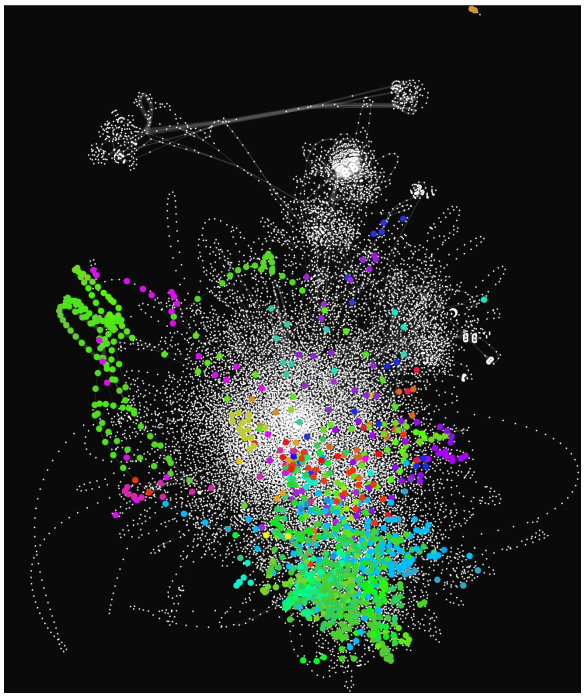
Execution Trace Visualization. Various researchers have proposed solutions for visualizing program execution traces. De Pauw *et al.* [15] created the JINSIGHT tool that visualizes the relations among runtime entities of a program based on execution traces. Cornelissen *et al.* [4] created the EXTRAVIS tool to visually encode dynamic method-call relationships among structural components of programs and navigate those relations using dynamic call-trace views. Reiss [16] proposes the JIVE visualization to illustrate dynamic properties about the memory and threads of a program in execution. Both JINSIGHT and JIVE visualize a program during its execution. Further, both



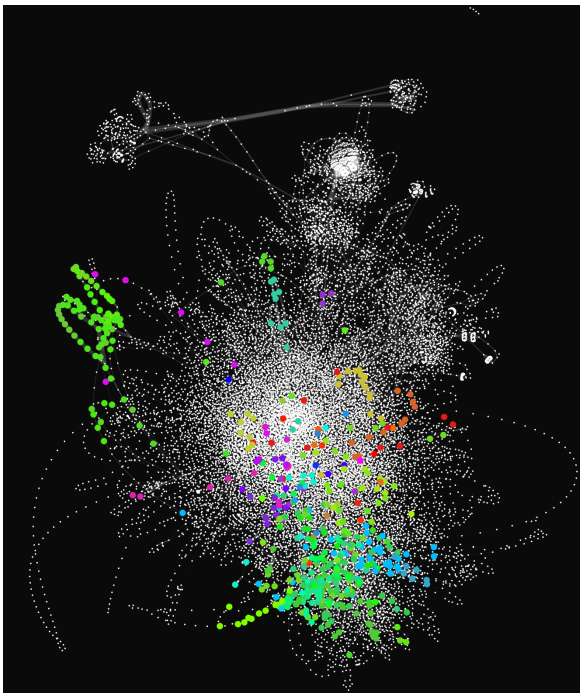
(a) During startup.



(b) While browsing the file-system to open a new file.



(c) While typing text in an open file.



(d) While doing a textual find-and-replace.

Fig. 7: CEREBRO for JEDIT at different stages during its live execution.

EXTRAVIS and JIVE map a program's runtime information to the corresponding structural source code components. Karran *et al.* [11], recently proposed SYNCTRACE, a technique to capture the runtime interplay of concurrent threads in a program using program execution traces. Much like CEREBRO, these tools visualize a program during its execution and map runtime information to source-code components. In contrast, CEREBRO targets the discovery of behavioral features, across multiple executions, which may crosscut the program structure to reveal how disparate structural components interact in a single aggregate, clustered view. The individual program execution replays are overlaid on the clustered view to reveal dynamic patterns of a single execution.

Visualizing Source Code Dependencies. Deng *et al.* [5] encode static dependencies between program instructions with the CONSTELLATION visualization to highlight cross-cutting aspects in a program. Kuhn *et al.* [13] cartographically visualize software code clusters based on the "topics" contained therein. Dietrich *et al.* [6] created BARRIO to visually cluster entities in Java programs to model their modular structure using the dependence relations therein. Krinke [12] visualizes program slices within a framework of static dependencies in a program. Similar to such works, CEREBRO clusters source code instructions; however, CEREBRO clusters instructions that form dynamic, functional aspects of the software program, using only dynamically collected information.

VII. CONCLUSIONS

In this paper we present a novel visualization — CEREBRO — that renders executions of a software program overlaid on a dynamically informed, global view of the program's source code. The visualization reveals: (1) dynamic, behavioral features that compose executions; and (2) the composition of the source code that cooperates to facilitate those features, which often includes multiple files and modules. We empirically found that the clusters identified by the visualization were meaningful and directed attention toward functionality that cross-cut the object-oriented, structural decomposition of the software subjects. We also found that the four independent judges displayed a greater degree of consistency in identifying emergent, crosscutting functional modules, especially when not distracted by the object-oriented decomposition. Finally, we found that visualizing live software execution enabled the identification and localization of code clusters that interact to implement external software features.

In the future, we plan to explore the use of other graph-layout algorithms to identify clusters of code. Additionally, we are working toward automatic and algorithmic identification of the boundaries of code clusters that represent internal constituent behaviors. Also, we plan to develop and evaluate use of CEREBRO for various of the envisaged usage scenarios described in Section II.

VIII. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under awards CAREER CCF-1350837 and CCF-1116943.

REFERENCES

[1] G. Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[2] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[3] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, November 2002.

[4] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *International Conference on Program Comprehension*, pages 49–58, 2007.

[5] F. Deng, N. DiGiuseppe, and J. A. Jones. Constellation visualization: Augmenting program dependence with dynamic information. In *International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8, 2011.

[6] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of Java dependency graphs. In *Symposium on Software Visualization*, pages 91–94, 2008.

[7] A. Dutot, F. Guinand, D. Olivier, and Y. Pigné. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. In *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS'2007)*, 2007.

[8] T. M. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[9] J. V. Haxby, M. I. Gobbini, M. L. Furey, A. Ishai, J. L. Schouten, and P. Pietrini. Distributed and overlapping representations of faces and objects in ventral temporal cortex. *Science*, 293(5539):2425–2430, 2001.

[10] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *International Conference on Software engineering*, pages 360–370, 1997.

[11] B. Karran, J. Trümper, and J. Döllner. Synctrace: Visual thread interplay analysis. In *Working Conference on Software Visualization*. IEEE Computer Society, 2013.

[12] J. Krinke. Visualization of program dependence and slices. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 168–177, 2004.

[13] A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz. Software cartography: thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):191–210, 2010.

[14] V. K. Palepu and J. A. Jones. Visualizing constituent behaviors within executions. In *1st IEEE Working Conference on Software Visualization, New Ideas and Emerging Results Track (VISSOFT-NIER)*, pages 1–4, September 2013.

[15] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002.

[16] S. P. Reiss. Visualizing Java in action. In *Symposium on Software Visualization*, pages 57–65, 2003.

[17] J. Trümper, J. Döllner, and A. Telea. Multiscale visual comparison of execution traces. In *International Conference on Program Comprehension*, 2013.

[18] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, Apr. 1971.