Visualizing Constituent Behaviors within Executions

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

***Abstract*—In this *New Ideas and Emerging Results* paper, we present a novel visualization, The Brain, that reveals clusters of source code that co-execute to produce behavioral features of the program throughout and within executions. We created a clustered visualization of source-code that is informed by dynamic control flow of multiple executions; each cluster represents commonly interacting logic that composes software features. In addition, we render individual executions atop the clustered multiple-execution visualization as user-controlled animations to reveal characteristics of specific executions—these animations may provide exemplars for the clustered features and provide chronology for those behavioral features, or they may reveal anomalous behaviors that do not fit with the overall operational profile of most executions. Both the clustered multiple-execution view and the animated individual-execution view provide insights for the constituent behaviors within executions that compose behaviors of whole executions. Inspired by neural imaging of human brains of people who were subjected to various external stimuli, we designed and implemented The Brain to reveal program activity during execution. The result has revealed the principal behaviors of execution, 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 The Brain and envisage the uses to which it can be put, and we provide two example usage scenarios to demonstrate its utility.**

I. Introduction

Visualizing runtime interactions in an execution of a software system can assist software developer in tasks that need program understanding. This importance of visualizing runtime interactions is emphasized by Jerding *et al.* [6], 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".

Traditional GUI-based debuggers assist in understanding executions; however, they localize the view of an execution, restricting the developers' attention to a single file, class or method at a time. More recent research has built upon the notion of Execution Murals to visualize varying runtime interactions. Such works include that by De Pauw *et al.* [9], Cornelissen *et al.* [1] and Trümper *et al.* [11]. However, such works visualize a single execution at a time, encoding program behavior that is potentially only relevant to that execution.

In contrast, Deng *et al.* [2] represented multiple-execution behaviors by using a force-directed graph of source-code-instruction nodes and control-dependence and data-dependence edges. They then used dynamic statement-coverage data to inform weights to edges to induce clustering of instructions that were evoked in the same executions. The

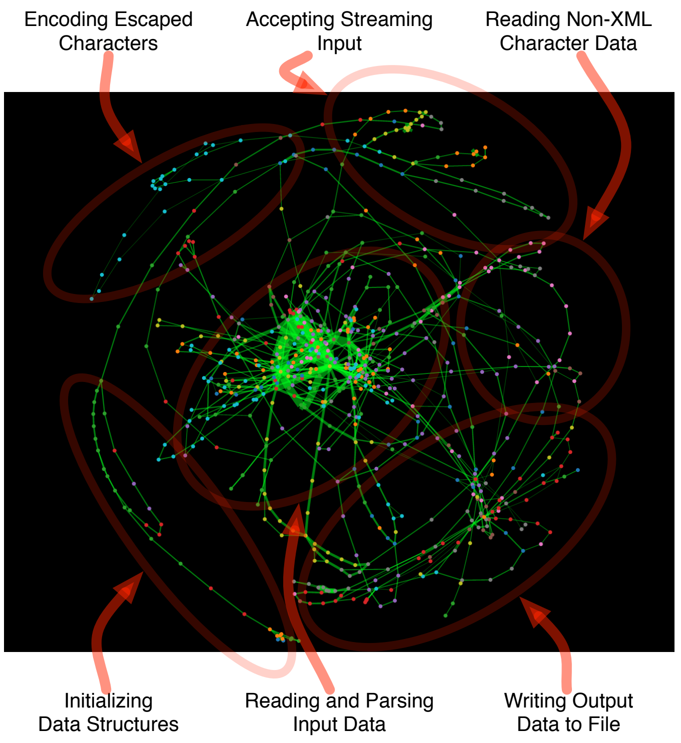

Encoding Escaped Characters
Accepting Streaming Input
Reading Non-XML Character Data
Initializing Data Structures
Reading and Parsing Input Data
Writing Output Data to File

Fig. 1: The Brain visualization of the NanoXML program.

resulting visualization, Constellation visualization, revealed clusters of instructions that shared static-analysis dependencies and were executed in similar sets of executions. However, it is subject to inaccurate inference of dependence coverage from the lower-fidelity statement coverage. Moreover, it can only reveal overall clusters of code across all executions.

Our objective is to provide a global view of program behavior due to, both, an individual execution and an aggregate effect of multiple executions, to assist developers' understanding of programs and executions via insight and discovery. We meet this objective with a visualization that we are calling The Brain, where we render frequently interacting source-code instructions as clusters, and overlay individual executions with user-controllable animations. The clustered visualization takes inspiration from our earlier Constellation visualization work, but instead uses statement traces and dynamic control-flow data to identify definitive (*i.e.,* not heuristically inferred) code relationships and sequentially ordered interactions of source code. In another divergence from our previous work, the clustered view is informed by purely dynamic information (*i.e.,* no static dependence analysis is required) and as such does not suffer from the over-approximations of such analyses.

As such, the resulting clusters are entirely informed by actual dynamic behavioral and functional features within and across executions. Much like our earlier work, we weight the edges according to frequency of dynamic occurrence.

We provide insight into individual executions by animating source-code executions, overlaid on the clustered view. The animation enables comparison against the aggregate behavior of all executions, which may reveal anomalies.

Figure 1 demonstrates the clustered view of THE BRAIN on the NANOXML program. We have overlaid labels upon the behavioral features that can be observed as distinct occurrences with the single-execution animations.[1] Nodes represent source-code instructions, which are placed nearby other instructions with with they commonly execute; and edges are optionally drawn with thickness denoting frequency of traversal. The visualization is described, in detail, in the following section.

The key contributions of our work are as follows.

- THE BRAIN: *a novel visualization that renders program behavior* due to an individual execution against the landscape of a more general program behavior due to multiple executions.
- *Two example usage scenarios*, demonstrating how THE BRAIN can assist software developers.

II. THE BRAIN VISUALIZATION

To provide a detailed overview of THE BRAIN, we adopt the 5-dimensional framework proposed by Maletic *et al.* [8] for describing software visualizations. Their five dimensions are: "Task," "Audience," "Target," "Presentation" and "Medium." Each of these dimensions address the "why," "who," "what," "how," and "where" questions, respectively. Finally, in this section we describe our choice of the "brain" metaphor.

Task. *Why is the visualization needed?* As this paper presents emerging work, all potential use cases for THE BRAIN are yet to be identified. However, we have discovered its applicability to diagnose anomalous execution behavior and provide insights into execution patterns that exist within and across executions of the code. In addition to these discovery activities, we envision its use for many other tasks, such as reverse-engineering or refactoring potentially disparate parts of the code that are responsible for software features. THE BRAIN presents a global view for program executions that typically are large information spaces. As Jerding *et al.* [6] highlight, global views provide users with "immediate insight into large information spaces" and also serve as effective tools for "visually searching" large information spaces that can guide subsequent analyses. Finally, we think that THE BRAIN provides an inspiring alternate conception of executing software, which may inspire software-visualization researchers.

Audience. *Who will use the visualization?* We envision the visualization to be useful in the previously described ways to (1) software programmers trying to understand code interactions to implement new functionality (2) software architects trying to understand code interactions for either re-architecting or verifying architectural compliance, (3) software testers and debuggers trying to understand and fix bugs, and (4) software-visualization researchers seeking new paradigms for representing program structure and execution.

[1]Due to our inability to demonstrate animations, we encourage readers to view a video of THE BRAIN at http://vimeo.com/spideruci

Target. *What is the data source to represent?* THE BRAIN represents source code and dynamically observed white-box behavior. Specifically, we represent each source-code instruction, along with its modular location, and the dynamically observed control flows among the instructions.

Presentation. *How to represent it?* We represent source-code instructions as dot-sized nodes in a force-directed graph. The visualization is composed of the following elements:

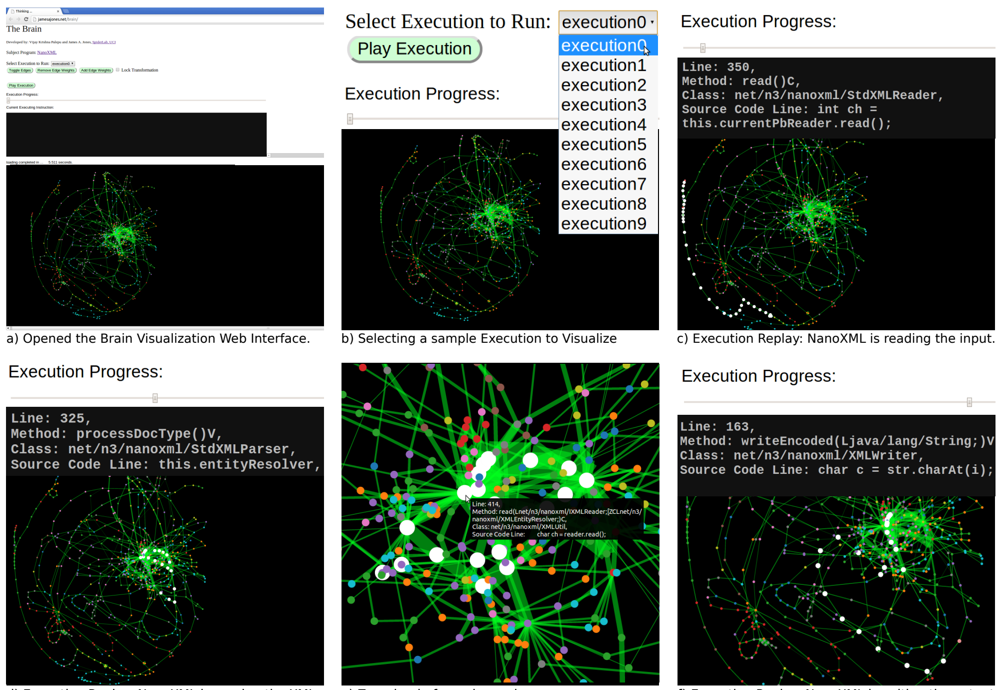
Nodes — Annotations, Color. The nodes of the graph are visualized as dots, each representing a unique source code instruction of the program being visualized. Each node is annotated with the class name, method name and the actual text of source-code instruction that it represents. The annotation for a node becomes visible when a mouse pointer is hovered over it. The color of each node is defined by the class to which it belongs, which enables the identification of cohesive or cross-cutting dynamic behavioral features.

Edges — Thickness, Length. Edges are optionally displayed, controlled with a toggle. When drawn an edge between two nodes represents a dynamically observed control-flow relation between the two source-code instructions represented by the two nodes. The thickness of the edge is directly proportional to the frequency of traversals of the represented flow, across all executions. The default length (and strength) of each edge is inversely proportional (positively for strength) to the frequency of traversals of the represented control flow. However, the eventual length is subject to the layout algorithm, which heuristically optimizes placement of nodes and their edges.

Layout — Force Directed Graph. The force directed layout is composed of a constant repulsive force among nodes and binding forces imposed by the edges between nodes. The binding edge force between two connected nodes is weighted according to the frequency of execution of the represented dynamic control flow. These forces influence the positions of all nodes in the graph after the layout algorithm is allowed to iterate and stabilize.

Animation — Execution Replay. An execution is animated in the visualization by highlighting the instructions from an execution trace in execution order. The highlight is accomplished with a white, larger-diameter node.

Interactivity — Pan and Zoom, Progress, Selection. The entire visualization is represented in vector form, thus allowing zooming and panning to focus on specific areas and movement among them. The progress of an execution can be viewed and controlled with a slider. Finally, regions of the graph can be lassoed, which produces a summary of the represented selected instructions in an accompanying pane of the visualization. In the future, we will provide brushing to complementary visualizations, such as SeeSoft [5].



a) Opened the Brain Visualization Web Interface.

b) Selecting a sample Execution to Visualize

```
Line: 350,
Method: read()C,
Class: net/n3/nanoxml/StdXMLReader,
Source Code Line: int ch =
this.currentPbReader.read();
```

c) Execution Replay: NanoXML is reading the input.

```
Line: 325,
Method: processDocType()V,
Class: net/n3/nanoxml/StdXMLParser,
Source Code Line: this.entityResolver,
```

d) Execution Replay: NanoXML is parsing the XML.

```
Line: 414,
Method: read(Lnet/n3/nanoxml/IXMLReader;[ZCLnet/n3/
nanoxml/XMLEntityResolver;)C,
Class: net/n3/nanoxml/XMLUtil,
Source Code Line:      char ch = reader.read();
```

e) Zooming-in for a closer view.

```
Line: 163,
Method: writeEncoded(Ljava/lang/String;)V
Class: net/n3/nanoxml/XMLWriter,
Source Code Line: char c = str.charAt(i);
```

f) Execution Replay: NanoXML is writing the output.

Fig. 2: THE BRAIN — Animated replay of an Execution for NANOXML

Medium. *Where to represent the visualization?* The current implementation of THE BRAIN runs in any modern web browser that supports the HTML5 and SVG standards.

The "Brain" Metaphor. The choice of the "brain" metaphor is inspired by visualizations of brain activity of people who are subject to external stimuli. Our use of the metaphor is rooted in the entity being presented in this visualization, *i.e.,* the working behavior of a program or computational unit that accepts input; has modules or sub-units with specific functions and work for specific inputs; and whose behavior is not well understood by its users. Similarly, the human brain can be considered a computational unit that responds to stimulus (*i.e.,* input); contains sub-units with specific functions that work for specific stimuli. For example, the frontal brain lobe is particularly active during problem solving; and in contrast, visual recognition particularly activates the Occipital lobe.

III. EXAMPLE USAGE SCENARIOS

We present two example usage scenarios for THE BRAIN involving, program comprehension and anomaly detection. The scenarios render NANOXML, which is a Java XML parser with >7,000 LOC, in a prototype implementation of THE BRAIN. The implementation and its use are illustrated in Figure 2. NANOXML's source code and test cases were obtained from the Subject-artifact Infrastructure Repository [4].

Scenario: Program Execution Comprehension. This scenario showcases how THE BRAIN can help in understanding the internal behavior of a program execution. Specifically, we illustrate the identification of execution patterns for different components, during a sample execution of NANOXML.

Early Execution. After rendering the program, Figure 2(a), we replay a sample execution as shown in Figure 2(b): select a sample execution from the drop-down list labeled *Select Execution to Run* and click the *Play Execution* button. As a result, a stream of instruction nodes light up in the visualization, as shown in the left-hand side of Figure 2(c), in the beginning of the execution replay. The execution's progress is continuously indicated by a slider, moving from left to right. Simultaneously, the most recently replayed source-code instruction is displayed. The example in Figure 2(c) depicts one-time, early execution of instructions from the `StdXMLReader` class, which initializes and sets up the reader functionality.

Middle Execution. The next phase of execution constitutes the bulk of the processing, as evidenced by the thick edges in the center of the visualization. Figures 2(d) and 2(e) depict the repeated execution of instructions from the classes `StdXMLParser`, `XMLUtil`, and several others. These instructions are visibly localized to a large cluster of nodes in the center of the visualization for the majority of the execution replay. This illustrates the dominant operation of the reading and parsing behavioral features.

Late Execution. The final phase of the execution performs writing of the data to output. Figure 2(f) depicts the execution of instructions from the `XMLWriter` class, which performs the outputting phase of execution.

Discussion. From the clustered visualization, the dominant behavior is clearly visible, and the auxiliary flows and clusters are set apart. Upon animating the execution, the sequence of the behavior and the way in which the features interact becomes apparent. Moreover, the heavy interaction of the classes that constitute the reading and parsing functionality reveals the magnitude of the code and execution that is devoted to this one phase of functionality. Although NanoXML may be considered to have a mostly comprehensible execution pattern, that execution pattern was not obvious to us before viewing the visualization, particularly the dominance of the reading and parsing functionality for all executions. Moreover, we imagine the use of such visualizations for programs with much larger feature sets which can reveal opportunities for development tasks such as code refactoring, performance optimizations, and design recovery.

Scenario: Execution Anomaly Detection. This scenario showcases how The Brain can help in detecting anomalous program behavior.

Malformed Input Anomaly. We found an anomaly in our testing with an execution replay that took remarkably less time to complete than the other execution replays. An incoherent and distinct animation of the instruction nodes, for this execution replay, further indicated an anomaly. A subsequent inspection of the corresponding input XML file for this execution replay revealed a mismatch in the starting and ending tags for an XML element. Fixing this the input XML file and rendering the resulting trace produced a regular execution replay.

Missing Behavior Anomaly. We found an anomaly when a cluster was not executed by a particular execution trace. Upon investigation, the unexecuted cluster was revealed to process string data in leaf XML tags, which was confirmed upon inspection of the input file.

Discussion. Using our own visualization in its testing revealed not only the common pattern of execution, but also some outliers. In the first execution anomaly, we discovered a flawed input file (which may or may not have been intentional by the tester). In the second execution anomaly, we discovered a correct input file but also became aware of the difference in the functionality that it tested and better understood the role of the unexecuted cluster.

IV. Related Works

Execution Trace Visualization. Several researchers have proposed visualizations of program execution traces. As examples, De Pauw *et al.* [9] created Jinsight, Cornelissen *et al.* [1] created Extravis, Reiss [10] created Jive, and Trümper *et al.* [11] recently created TraceDiff. Much like The Brain, these tools visualize a program during its execution and map runtime information to source-code components. However, in contrast, The Brain 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.

Visual Clustering of Source Code. Several researchers have proposed visualizations to cluster source code. As examples, Deng *et al.* [2] created Constellation visualization to show co-executing static dependencies, Kuhn *et al.* [7] cluster based on textual analyses, and Dietrich *et al.* [3] created Barrio to cluster based on static modular structure. Much like The Brain, these tools cluster source-code elements. However, in contrast, The Brain clusters source-code instructions for the purpose of revealing behavioral features within executions and provides facilities to visualize individual dynamic executions of the program.

V. Conclusions

In this paper we present a novel visualization — The Brain — 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; (2) the composition of the source code that cooperates to facilitate those features, which often includes multiple files and modules; and (3) anomalous behaviors of individual executions. We illustrate, through usage scenarios, how the visualization may assist developers in their development tasks. In the future we plan to evaluate the scalability of both the visual elements and the underlying mechanics of trace collection for larger systems. Such evaluations would focus on understanding the usefulness of such a visualization to developers; thus guiding future evaluations by user studies. The resulting qualitative results for such visualizations make scientific validation challenging. We hope to discuss evaluation strategies, using empirical studies and theoretical frameworks, to assess the usefulness to developers.

VI. Acknowledgements

This material is based upon work supported by the National Science Foundation under award CCF-1116943.

References

[1] 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.

[2] 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.

[3] 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.

[4] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 2005.

[5] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11), 1992.

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

[7] 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.

[8] J. I. Maletic, A. Marcus, and M. L. Collard. A task oriented view of software visualization. In *International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–40, 2002.

[9] 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.

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

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