Visualizing Multiple Program Executions to Assist Behaviour Verification

Chunying Zhao Kang Zhang Jie Hao W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
Richardson, TX, USA
{cxz051000, kzhang, jxh049000, ewong}@utdallas.edu

***Abstract*—Visualization techniques have been widely used in representing software artifacts. They play a central role in conveying program information to software developers. While numerous tools have been developed to visualize information such as static software architectures, dynamic program behaviors, and debugging processes in different ways, little attention has been paid to visualizing correlations and variations among program representations. This paper investigates the visualization of cross-references across multiple program executions upon different testing inputs so that meaningful and viewable properties can be presented to the viewpoint from different perspectives. Visualizing such a comparison can help feature location and program behavior verification. It also helps programmers better understand and test their software which can have a significant impact on improving its reliability.**

Keywords-software visualization; dynamic behavior comparison; correlation visualization

I. Introduction

Software visualization [3][9][32][39] techniques have been widely used in the lifecycle of program development spanning from requirements and design to software testing and maintenance, where program structures and analytical results are represented in an intuitive way that fits humans' mental models. So far, many visualization tools have been developed to aid comprehension of software systems from various perspectives, including static software architecture [2][4][13][28][29][33], source code navigation[22], dynamic program runtime behaviors [8][19][25][26], debugging and fault location [10][20], statistic analysis [11], and program evolution [5][6][12].

Inherited from information visualization, existing software visualization tools utilize various graphical ingredients and topological theories to visualize software data, e.g. force directed layouts [14], zooming techniques [31], and fisheyes [15]. Colors [16], metaphors [2], multiple dimensions [23][34], and animations [30][35] are also popular visualization methods for conveying software information to end-users.

While prior software visualization techniques can display information such as static software architectures, dynamic program behaviors, and debugging processes in different ways, less attention has been paid to highlighting variations and correlations across multiple software representations, e.g. visualizing a comparison of program executions. VisLink [7], a methodology for revealing the relationship amongst information representations, emphasizes the importance of constructing correlations between information representations. It sheds light on the visualization of correlations across software artifacts.

This paper aims at visualizing the comparison of multiple program executions from different program inputs by highlighting their similarities and differences. Visualizing the cross-referencing of execution traces can help developers to better understand and test their software which can significantly improve its reliability. According to Miranskyy et al. [24], comparing program execution traces is beneficial for several reasons: it helps to improve test coverage and identify duplicate test cases; it also helps to locate common features [37] pertaining to particular behaviors that occurred in different executions.

GAMMATELLA [19][26] is one of the well-known tools for visualizing program execution and debugging. It combines treemaps, color hues, and brightness to illustrate program execution information. Ware et al. [35] used stereo and motion cues to maintain connections of related events in consecutive information representations. To compare multiple representations of program executions, a more efficient way is to visualize the representations of interest in the same scene. By presenting the similarities and differences in a human-centric manner, users can easily identify common and unique events. Events unique to an execution can be considered as a feature relating to a specific test case.

To realize such a comparison, we encountered several challenges, such as a large amount of traces, visual modeling of program executions, constructing similarities and differences among individual executions, and efficient navigation supports. To address them, we firstly reduce execution traces to different abstraction levels with a folding/unfolding functionality, so that only necessary information is displayed in a limited visualization space. Then we visually model both individual executions and their connections using a multi-plane layout based on the conceptual closeness of testing inputs. Finally we provide a reactive navigation interface with an eye tracker to browse the visual space in a human-centric manner.

The rest of the paper is organized as follows. Section 2 presents an overview of the approach, data abstraction, and visual models. Section 3 illustrates a multi-plane layout and mapping specifications. Section 4 describes multiple perspectives of the visual space and a reactive navigation interface. Section 5 presents an experiment on an open source software and observations from preliminary results. Section 6 reviews related work. Section 7 concludes the paper and discusses our future work.

II. Modeling Multiple Executions

A. Overview

Creating an effective layout for comparing multiple executions is not easy. Since there have been a number of useful 2D representations (e.g. sequence diagrams, call graphs) that could visualize various aspects of a system, these representations need to be correlated.

To correlate such representations, a 3D space is advantageous over a 2D plane because it provides more perspectives for users to observe from [34]. We choose a multi-plane 3D layout by displaying individual executions on 2D planes and constructing their connections across the planes in a 3D space. The multi-plane layout can clearly separate correlations from individual representations.

For instance, in Figure 1(a), correlations are clearly separated from individual representations compared with the multi-box representation in Fig.1 (b) that clusters information of individual executions in the boxes. Different from existing multi-plane layouts that are primarily used for visualizing software hierarchical structures (e.g. Rigi [33]), this paper aims at constructing correlations among multiple executions and deriving meaningful observations.

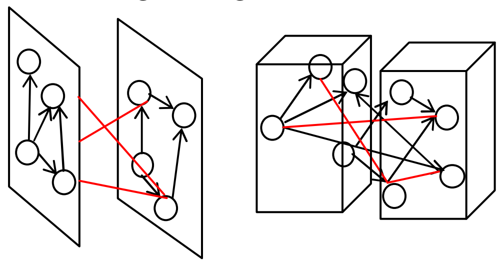


Figure 1. Different 3D visualization arrangements

Figure 2 depicts an overview of our approach. Execution traces are reduced based on user-defined criteria. Representations are ordered as multiple planes. Then information on adjacent planes is mapped, and graphical elements are rendered on the 3D space with multiple perspectives.

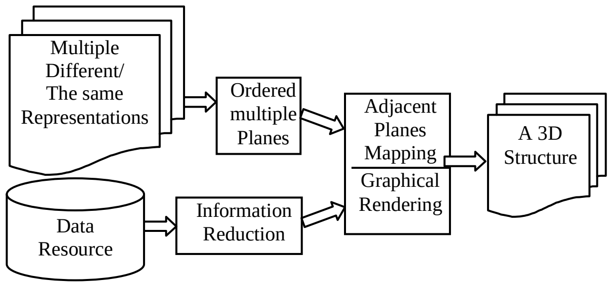


Figure 2. Approach Overview

B. Execution Trace Abstraction

Handling program execution data involves two major steps: trace collection and abstraction. We collect execution traces using an aspect-oriented approach. Instrumentation aspects are created and compiled seamlessly together with the Java byte codes of source programs. In the current implementation, we record method invocations, and build a call graph of method invocations represented in a standard GraphML format. Multiple executions on various inputs are traced.

A pre-processing [38] is conducted to preliminarily filter the collected trace by abstracting away detailed information not to be visualized. The information abstracted away includes method calls that contribute little to the representation of program interactions. Such method calls may include:

(1) Repeated method invocations.
(2) Lower-level method invocations.
(3) Intra-object method invocations.

Reducing these method invocations makes it possible to display a large amount of information in a limited visual space. The reduction is a multi-level abstraction from the highest level to the lowest level by unfolding events, and vice versa. Based on the abstraction criteria, when events are folded, the new event representing the folded information will be the highest level event of the folded ones. For instance, if we collapse a call chain: *A* calls *B*, *B* calls *C*, and *C* calls *D*. The new event denoting the call chain will be *A*. That is, we collapse the low level events, and could iteratively unfold them upon users' requests. Similarly, intra-object method invocations could be folded inside inter-object method invocations. Details of our trace reduction technique could be found elsewhere [38].

C. Visual Models for Executions

We use sequence diagrams to visualize individual executions on 2D planes. Although sequence diagrams have been used widely in software design and program execution, visualization of cross-referencing multiple executions has not appeared in the literature to our knowledge.

Each object is represented as a small green solid sphere in the space. As each object may include one or more objects due to the abstraction, the more objects a sphere includes, the larger the sphere will be. We could assign different colors to objects belonging to different threads for a multi-threading program. Method calls between objects are represented by horizontal edges arranged on the occurring order of events. Edges within one execution (i.e. intra-execution edges) have the same color (red in our current implementation). Mapping between executions are highlighted with yellow lines. Figure 3(a) depicts a typical representation of a single execution, and Figure 3(b) describes a multi-execution representation.

Choosing the sequence diagram to visualize individual executions has several merits: (1) it emphasizes object interactions; (2) it makes the layouts of multiple executions

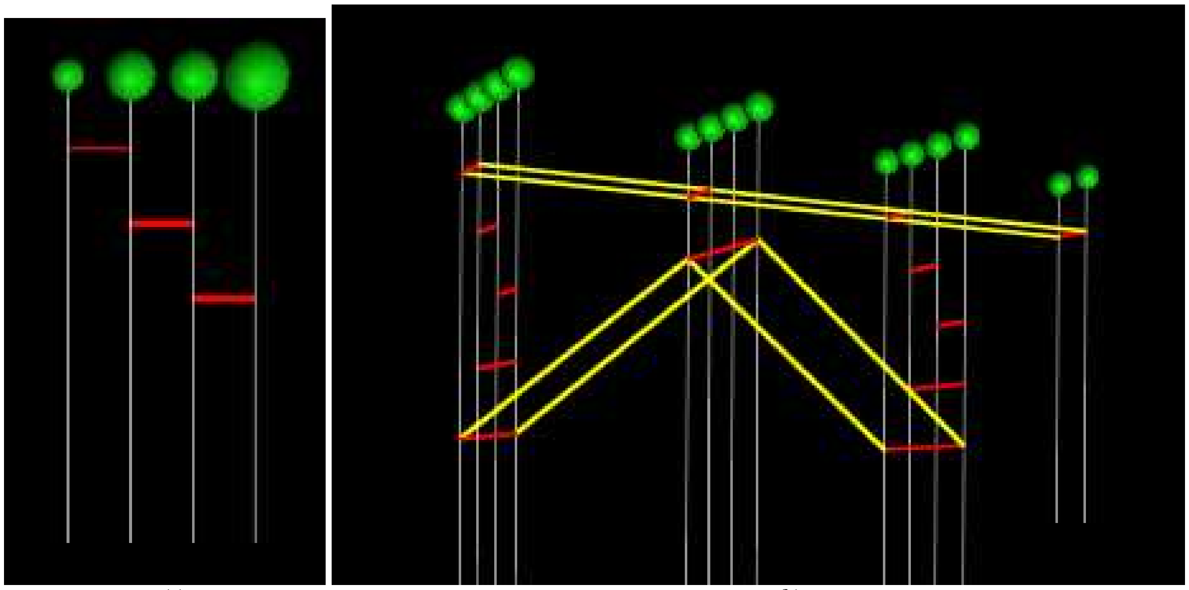


Figure 3. Basic models of multiple planes

uniform, and thus eases the intuitive cross-referencing compared with other 2D node-link graphs.

III. MAPPINGS BETWEEN EXECUTIONS

Challenging issues for visualizing multi-plane correlations include:

(1) Visual properties of individual planes, i.e. orders, orientations, and positions of planes in a 3D space;

(2) The optimal layout of planes for the best observation;

(3) Mappings between information in two adjacent planes, i.e. the criteria of judging whether two (clustered) events on different planes are the same or not.

A. *The Layout of Planes*

The order of planes is determined by the semantic relationships between the planes. The general principle is to position planes having close relationships together. The assumption is that representations with close relationships are apt to share more common properties. In multi-execution visualization, executions using similar test cases or performed consecutively can be spatially positioned together, so that the user can easily observe conceptually related clusters.

More formally, we arrange multiple planes based on concept analysis as follows:

Let R ($r_1, r_2,..., r_i,...$) be the set of different relationships among all representations (planes), where each $r_i \subset R$ is semantically defined based on the meanings of the contents it represents. Let P ($p_1, p_2,\ldots, p_i,...$) be the set of planes for different representations. Let A ($a_1, a_2,..., a_i,...$) be the set of common attributes for the relationships in R.

A cluster of planes S is defined as a triple (r_i, A_i, P_i) such that

(1) $r_i \in R$.

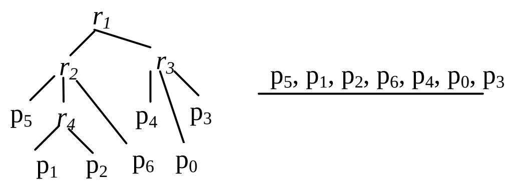
(2) $A_i = \{a_m, a_{m+1},\ldots, a_n\} \subset A$, are common attributes for r_i.

(3) $P_i = \{p_m, p_{m+1},\ldots, p_n\} \subset P$.

(4) $\forall p_j \in P_i$, $m \leq j \leq n$, contents on p_j have all the attributes of A_i.

(5) $\forall p_k \notin P_i$, $\exists\, a_i \in A_i$, contents on p_k do not have the attribute a_i.

A concept lattice can be created from the semantics of the relationships in R. The attributes and relationships can be annotated by developers using textual labels on each execution. We first turn the concept lattice into a tree structure by breaking any possible cycles. Figure 4(a) illustrates a concept lattice with four types of relationships $\{r_1, r_2, r_3, r_4\}$ and seven planes $\{p_0, \ldots., p_6\}$. Figure 4(b) depicts the ordered planes based on the concept lattice using a tree- traversal algorithm.



(a) (b)

Figure 4. Ordering of planes

The following tree-traversal algorithm can order planes in a concept lattice.

```
Input: root $rt$ of the lattice; Output: an ordering of the planes
PlaneOrder($rt$)
FOR each child $c$ of $rt$,
IF $c$ is a plane ($p_i$), put $p_i$ in the next adjacent available position in the order
```

```
    ELSE IF $c$ is a relationship ($r_i$), call $PlaneOrder(c)$;
    Mark $p_i$ as rt-cluster.
```

The multiple planes are rotatable to provide a desirable observation. We associate the orientations of planes with the user's viewpoint so that the rotatable planes can dynamically present the maximal viewable information to the viewer by dynamically forming an angle facing the current position of the viewpoint and showing the properties interesting to the user. The change of the viewpoint will trigger changes of plane orientations, and highlights in the new view, in a similar fashion as a camera model [1].

Figure 5 illustrates an example position of the viewpoint where the planes rotate to obtain the best exposure of the inter-plane relationships that are the current observing focus of the user.

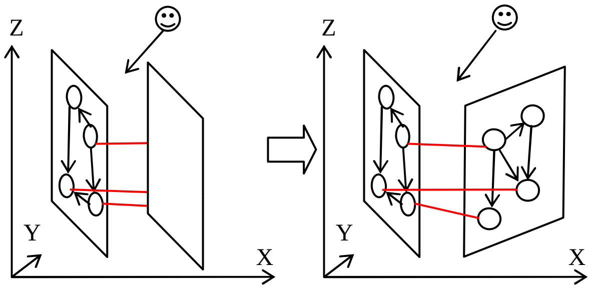


Figure 5. Planes adjusting automatically against the viewpoint

B. *Mapping Specifications*

In visualizing multiple representations, an information set includes multiple individual representations and correlations among them. We define the correlation between two representations as a *concept mapping*, denoted as $M(X, Y)$, where X and Y are two visual representations, i.e., mapping information sharing the same attributes from X to Y. Each visual representation refers to an information subset with a specific layout. $x \subset X$ is a visual entity in X, and $y \subset Y$ is a visual entity in Y. $m(x, y) \subset M(X, Y)$ maps x to y. The *concept mapping* requires that two connected entities share common attributes.

The *concept mapping* in cross-referencing has several properties:

(1) It is a concatenation of $M(X_1, X_2)$, $M(X_2, X_3)$,…, where X_i is the i^{th} visual representation.

(2) $m(x, y) \subset M(X, Y)$ could be a one-to-more or more-to-one mapping.

(3) $M(X, Y)$ is transitive, i.e. $M(X_i, X_j) \cup M(X_j, X_k) \rightarrow M(X_i, X_k)$.

To visualize correlations among multiple representations, we consider:

(1) The visualization of individual representations X_i and X_{i+1};

(2) The visualization of mappings $M(X_i, X_{i+1})$ between individual representations X_i and X_{i+1}.

Figure 6 illustrates mappings between entities among multiple executions. Two entities considered the same will be visually connected. In multiple program executions, two entities (events) are connected if they are:

(1) invoked by objects having the same class names, and

(2) both callers (callees) of the same method invocation.

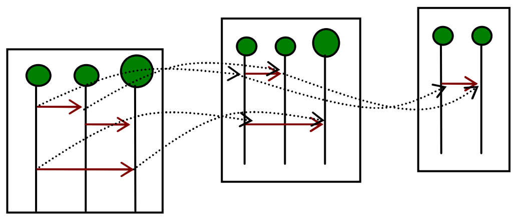


Figure 6. Mappings between planes

Definition 1: Events e_1 and e_2 in two abstracted call graphs S_1 and S_2 are the same if and only if the nodes n_1 and n_2 in the graphs denoting e_1 and e_2 are isomorphic, and have the same names [38].

The isomorphism requires that two abstracted nodes have the same structural connections with their neighboring nodes. Two nodes have the same name if they have the same method names, class names, thread names, and arguments.

IV. MULTIPLE PERSPECTIVES

A. *Visual Coordinates*

Multiple planes in parallel form a cube, and present several meaningful properties when the user observes the 3D structure from different perspectives. Figure 7 depicts three major perspectives: the top view, the front view, and the side view.

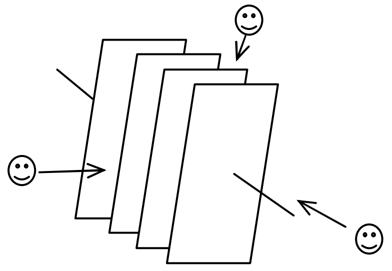


Figure 7. Multiple perspectives

We map visual properties of the 3D structure to a Cartesian coordinate system. Figure 8 depicts the corresponding information on each axis. Each perspective is formed by two axes in the space. The property of each axis is determined by the information topology of the 3D structure. In this paper, the X-axis, -Y-axis, and Z-axis are mapped to object interactions, event orders, and execution instances (abbreviated as O, V, and E, respectively).

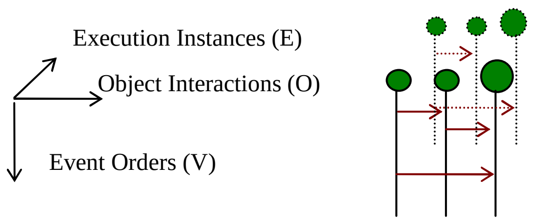


Figure 8. Visual coordinates

The perspectives in the visual space are generalized into three views: the front view (i.e. the *O-V* plane in the visual coordinate system), the top view (i.e. the *E-O* plane in the visual coordination system), and the side view (i.e. the *V-E* plane in the visual coordinate system). Each view is formed by two axes as described below.

B. *Program Properties from Multiple Perspectives*

To compare executions, we not only need to locate common behaviors but also need to understand the contextual information around the common behaviors, such as the interactions with other objects and temporal properties. We take advantage of structural properties of the cube, and derive the meaning of each view as follows:

(1) The front view on *O-V* plane: Method invocations within each execution, i.e. object interactions.

(2) The top view on *O-E* plane: Objects' participations in multiple executions, i.e. common activities of objects in different executions.

(3) The side view on *V-E* plane: Events' occurrences in multiple executions. It helps to compare the temporal properties of common or unique events in different executions.

Figure 9 shows a front view on the *O-E* plane. The planes for multiple executions are in parallel. The object interactions within each execution are represented using a sequence diagram.

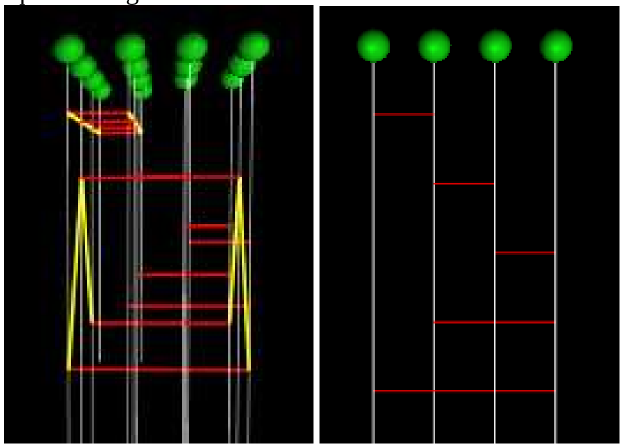

(a) (b)

Figure 9. A front view on the *O-V* plane

Different from a traditional sequence diagram, our approach clusters objects and interactions in different levels of abstraction. The interactions between objects indicate the communications between different object clusters. The user can select one of the presented executions by clicking (or gaze at it through an eye tracker) an object in that execution, and view it separately. For instance, Figure 9(b) shows the first execution in Figure 9 (a) separately.

Viewing the executions on the *O-E* plane in Figure 10, the user can identify and compare how objects participate in the different executions. In this view, objects having the same activities are connected by yellow lines. Some objects may participate in all the executions, while others may appear in fewer executions. For instance, in Figure 10 following the yellow lines, we can conclude that the top two objects in each execution have participated in all the four scenarios, and involve the same method invocations.

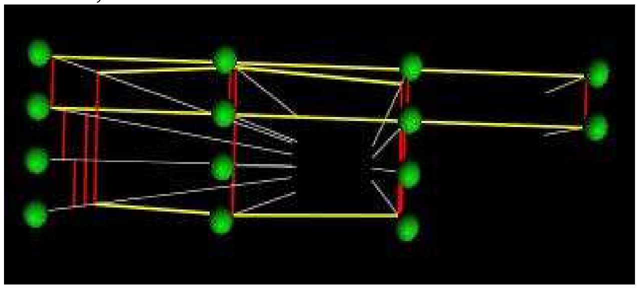


Figure 10. A top view on the *O-E* plane

The side view of the executions on the *V-E* plane is shown in Figure 11. Since the V-axis records the occurring order of events, viewing the space on the *V-E* plane, the user can obtain temporal properties of object interactions. Common events occurring at the same or different instances of time in multiple executions connected by yellow lines can be seen clearly. For instance, the common events connected by the top straight yellow lines in Figure 11 happened at the beginning of each execution. The bottom angled yellow lines connecting the same events in three executions indicates that these events did not occur in the last execution, and they happened earlier in the second execution than those in the other two scenarios. It reveals the temporal distribution of common events among multiple executions.

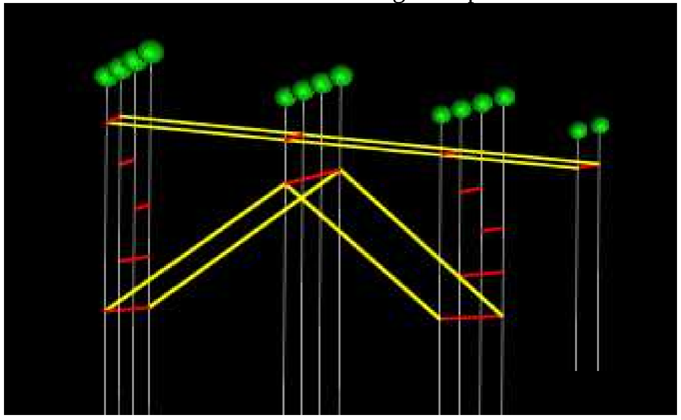


Figure 11. A side view on the *V-E* plane

C. *Navigation Supports with Eye Tracking*

In addition to traditional navigation supports that use the mouse/keyboard to move, rotate, and zoom in (out) a 3D scene, we use an eye tracker to capture users' visual targets and aid navigation.

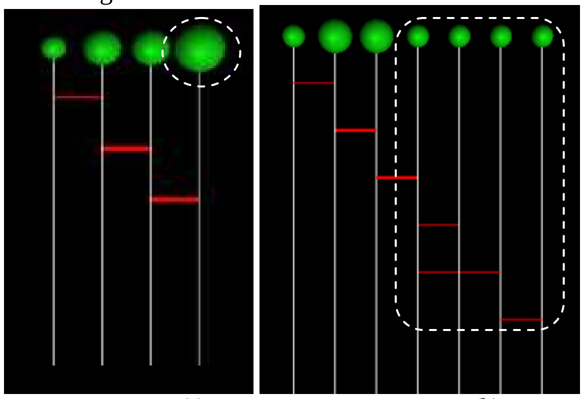

(a) (b)

Figure 12. Folded/unfolded views with an eye tracker

The eye tracker is a cursor controlled by the pupil movements that occur while the viewer physically gazes at an object of interest [21]. Fixation (the stabilization of eyes on an object of interest for a certain period of time) and saccade (quick movements of eyes from one location to the next) are the two most common types of eye movements [21]. We use fixation to determine the user's target. The ordinate of the pupil in the eye tracking controller screen is mapped to that in the visual space.

The user can choose to see the details of an object by gazing at it for a certain amount of time (e.g. 2 sec.). Then the object can be unfolded on demand. Assume that the big node (in the dotted circle) in Figure 12(a) is the observer's current focus; it could be unfolded to the object interactions shown in the dotted rectangle in Figure 12(b). Iteratively, the user can trace down from the highest level to the lowest to see method invocations in detail. A large green sphere indicates that it can be unfolded. Similarly, a thicker red line includes more interactions between objects. Textual information (e.g. object names) will be displayed upon users' requests.

Semi-transparency is used to aid highlighting. The information related to a specific object interesting to the user can be automatically highlighted by applying transparency to the contextual information. For instance, when a user gazes at the second object (the dotted circle) of the leftmost execution in Figure 13 for a certain amount of time, the interactions and correlations caused by the object and its correlations with other executions are highlighted by making the contextual information semi-transparent. The transparency is tunable for obtaining the best contrast. The user can get a quick impression about the influence of this object among multiple program executions via highlights.

V. EXPERIMENTS

A. *Experiments with JHotDraw*

We have evaluated the multiple-plane approach on an open source project, JHotDraw. JHotDraw is a GUI framework supporting simple drawing activities written in Java, and was initially designed to illustrate the application of design patterns. We used Version 6.0 that contains 136 classes, 1,380 methods, and 19 interfaces.

The experiment aims at evaluating the effectiveness of visualizing correlations and differences among multiple program executions. The scalability of the approach for a large amount of data is also investigated. We design different testing scenarios for the program, and present traces in different abstraction levels.

The four testing scenarios include:

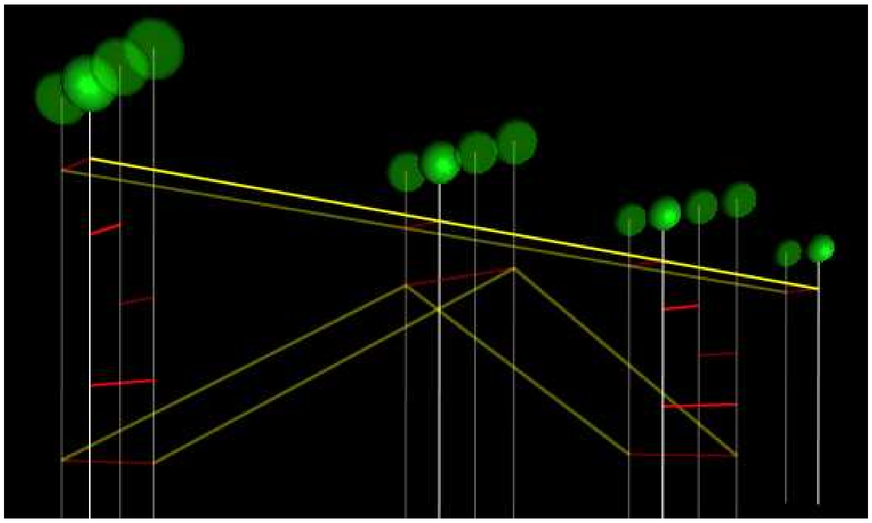


Figure 13. Highlighted information with semi-transparency

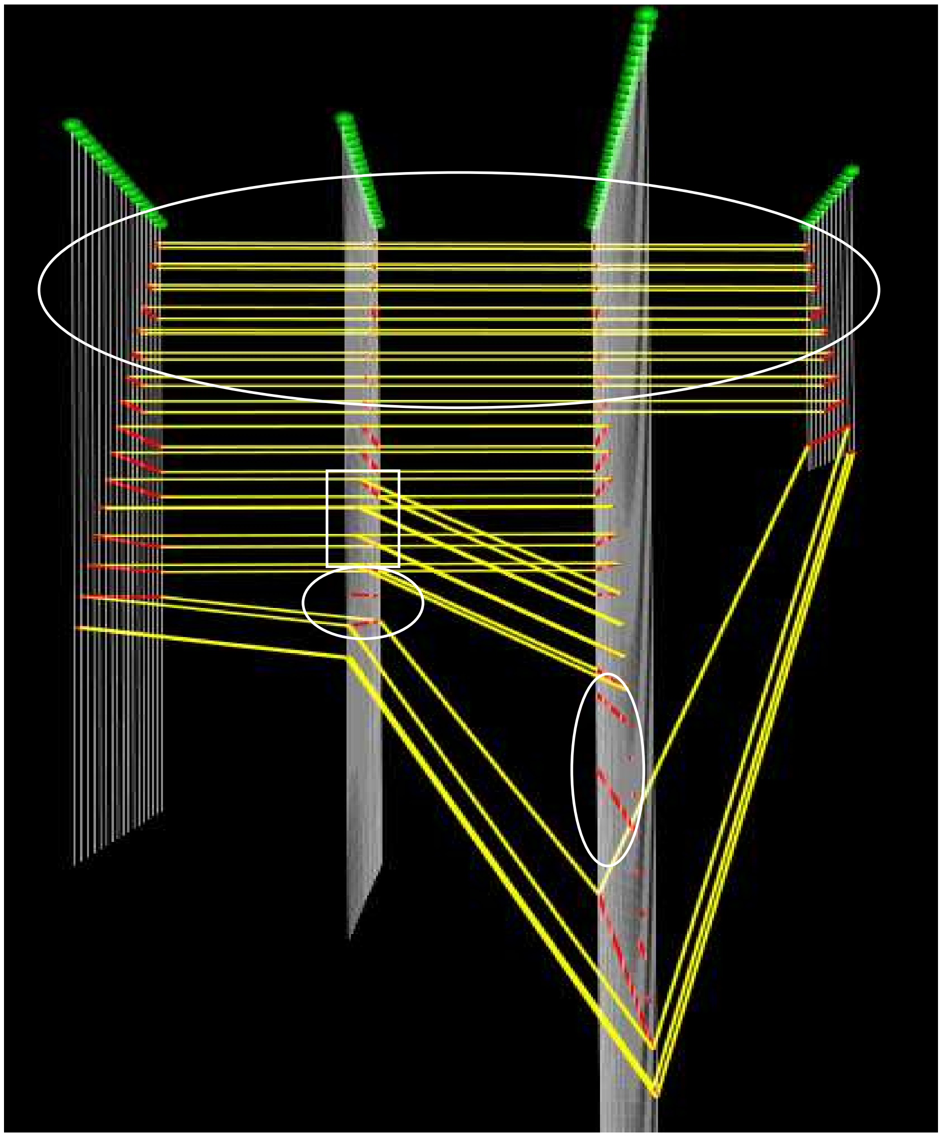


Figure 14. An abstracted visual representation

(1) Run JHotDraw → create new view→ draw one round rectangle → close JHotDraw.
(2) Run JHotDraw → create new view→ draw one ellipse → save the view→ close JHotDraw.
(3) Run JHotDraw → create new view→ draw one round rectangle → draw one ellipse → start and stop animation→ close JHotDraw".
(4) Run JHotDraw → close JHotDraw.

The above scenarios have both common and different behaviors. The traces were collected from the package *org.*jhotdraw.*sampls.javadraw*. This package contains the most essential classes and methods that can initiate the drawing environment and support drawing applications.

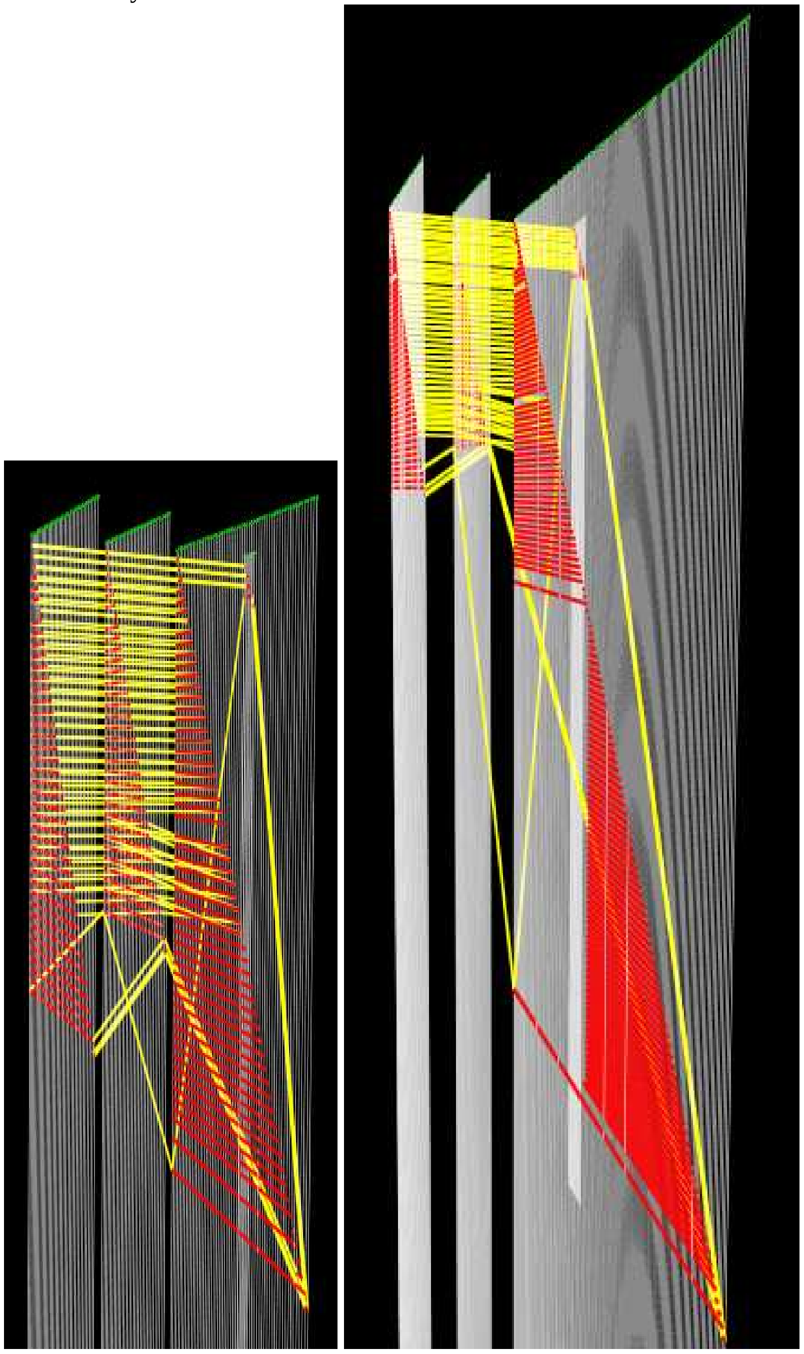
The four planes in Figure 14 from left to right correspond to the four scenarios with the same level of abstraction, respectively. The interactions within each execution are illustrated by red lines. The yellow lines connect the same events across each pair of adjacent planes.

Following the yellow lines in the side view, the user can easily identify common and different activities across the four scenarios. The representation can also be used to verify a program's expected behavior.

In Figure 14 the inter-execution connections highlighted by the white ellipse on the top indicates that those events connected by the yellow lines exist in four executions and occurred at an early stage in the scenarios, which conforms to the fact that the initialization of JHotDraw exists in all the scenarios, which is a common behavior. Similarly, the angled yellow lines at the bottom in Figure 14 also connect some common behaviors in the four scenarios.

The scalability is also investigated in our experiments. It is achieved by collapsing information at different abstraction levels. Figure 15 illustrates the same four scenarios in two different abstraction levels. The executions in Figure 15(a) were abstracted by collapsing the method invocations with a depth greater than three in a call chain. The executions in Figure 15(b) did not collapse any method invocations based on call depths, and thus include more data (displayed in a scale larger than (a)). By comparing the representations at different levels, the user can identify which behaviors are less significant and have been collapsed away in the abstraction process.

From a high-level view, the user can observe and infer the basic structures of program behaviors. With navigation supports, the user can track down to low-level details.



(a) (b)

Figure 15. An abstracted visual representation

B. Discussion

The multi-plane approach is useful for comparing and analyzing multiple representations having intrinsic connections. The planes could dynamically rotate so that the best exposure of inter-plane relationships representing properties interesting to the user is presented towards the direction of the user's viewpoint.

To date, we have only developed the approach that arranges and visualizes 2D planes in parallel or dynamically presents inter-plane relationships between two planes as discussed in Section 3.1. For multiple planes to present viewable properties towards the viewpoint with minimum occlusions, the optimal angles for individual planes need to be explored. Calculating the angles of multiple planes geometrically involves many factors, such as the viewpoint, the number of planes, the layouts on individual planes and the visualization tasks, etc.

The presented approach addresses the scalability issue by information abstraction, which is currently a preprocessing step, prior to the visualization process. The abstraction step compresses only low-level and uninteresting information to reduce the 3D visual complexity.

VI. Related Work

There has been a huge amount of work on program visualization and analysis.

Feijs and Jong [13] visualized the organization of software architectures in a 3D space, in which software modules were mapped onto bricks of different shapes. The bricks are then connected by lines to simulate the relationships between objects. Similarly, Alam *et al.* [2] used a metaphor between software architectures and city buildings. They represented components and metrics of complex software in a 3D virtual world. Navigation is simulated as walking in the street of a city. These visualization approaches only consider one representation of a static architecture, and do not support comparisons between different executions or architectures.

Achievements in visualizing the changes or evolution of software systems includes the work of Beyer *et al.* [5], which created an animated 2D storyboard to show the evolution of software architectures. The software artifacts on 2D storyboards are displayed as a bunch of clusters. The clusters are efficient in visualizing software modulation but are difficult to be compared. Also, the changes of software modulation are achieved via the animation of the displays on 2D planes. The animated 2D plane is good at visually illustrating the evolution process, which is continuous and incremental. In contrast, our 3D environment displays multiple executions simultaneously. The commonalities and differences across multiple executions are easy to be identified.

Apart from visualizing software architectures and evolutions, understanding program behavior has been a research topic in both program comprehension and software visualization. One of the major issues related to visualizing program behavior is the compression technique for execution traces. Hamou-Lhadj *et al.* [17][18] have proposed different abstraction approaches to summarizing the contents of a program behavior. Reiss [27] defined an automated model for dynamic visualizing program execution using user abstractions.

GAMMATELLA [19][26] visualizes executions in three levels in 2D: a file-level representation in a miniaturized view, a system-level view using treemaps, and a statement-level view using colors' hue and brightness. GAMMATELLA does not support comparisons either. Similarly, TraceVis [29] visualizes program behaviors by displaying microprocessor instructions in a 2D plane. It supports queries, arbitrary levels of zooming, and annotations on colorful blocks. Okamura *et al.* [25] visualized executions and debugging using visual programming principles. They used different 3D objects to represent software artifacts, and simulated state transitions by applying animation on the 3D objects. These approaches, however, do not support a comparison of different program executions.

VII. Conclusion and Future Work

This paper has presented a 3D visualization technique for comparing and analyzing multiple representations. Specifically, multiple program executions are represented as multiple planes in parallel. The correlations and differences are constructed and highlighted. A 3D visualization tool has been implemented and experimented on an open source project. Locating the correlations among multiple program behaviors can help programmers better understand and test their software which can have a significant impact on improving its reliability.

Multiple executions presented in this paper could be generalized to be multiple representations, e.g. for different software versions, different visual formalisms (such as those in UML), or different platforms. This is one of our future directions. Our immediate future work is on the exploration of adjustable angles for the 2D planes according to the viewpoint position, more visual features, focus-contextual presentation, and so on. More experiments and user studies on larger application programs will also be performed.

References

[1] A.G.N. Ahmed, "High Quality Camera Paths for Navigating Graphs in Three Dimensional Space", *PhD Thesis*, University of Sydney, 2008.

[2] S. Alam and P. Dugerdil, "EvoSpace: 3D Visualization of Software Architecture", *Proc. of SEKE'07,* 2007, pp.500-505.

[3] T. Ball and S.G. Erik, "Software Visualization In the Large", *IEEE Computer*, Vol.29, No.4, April 1996, pp.33-43.

[4] M. Balzer and O. Deussen, "Hierarchy Based 3D Visualization of Large Software Structures", *Proc. IEEE Visualization*, 2004, pp. 4p-4p.

[5] D. Beyer and A.E. Hassan, "Animated Visualization of Software History Using Evolution Storyboards", *Proc. WCRE'06,* 2006, pp.199-210.

[6] C.S. Collberg, S.G. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A System for Graph-Based Visualization of the Evolution of Software", *Proc. SoftVis'03,* pp.77-86.

[7] C. Collins, S. Carpendale, and G. Penn, "VisLink: Revealing Relationships Amongst Visualizations", *IEEE Trans. on Visualization and Computer Graphics (InfoVis'07))*, Vol. 13, No.6, 2007, pp.1192-1199.

[8] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J.J. V Wijk, and A. van Deursen, "Understanding Execution Traces Using Massive Sequence and Circular Bundle Views", *Proc. ICPC'07*, pp.49-58.

[9] P. Eades and K. Zhang (Eds.) Software Visualization, World Scientific Co., Singapore, 1996.

[10] J. Eagan, M.J. Harrold, J.A. Jones, and J. Stasko, "Technical Note: Visually Encoding Program Testing Information to Find Faults in Software", *Proc. InfoVis'0*, pp. 33-36.

[11] S.G. Eick, J.L. Steffen, E.E. Sumner, "Seesoft- A Tool For Visualizing Line Oriented Software Statistics", *IEEE Trans. on Software Engineering*, Vol. 18, No. 11, November 1992, pp.957-968.

[12] S.G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and Paul Schuster, "Visualizing Software Changes", *IEEE Trans. Software Engineering*, Vol. 28, No.4, 2002, pp.396-412.

[13] L. Feijs and R.D. Jong, "3D Visualization of Software Architectures", *Communications of ACM*, Vol. 41, No. 12, 1998, pp.73-78.

[14] T.M.J. Fruchterman and E.M. Reingold, "Graph Drawing by Force-Directed Placement", *Journal of Software Practice & Experience*, Vol. 21, No. 11, November 1991, pp.1129-1164.

[15] G. Furnas, "Generalized Fisheye Views", *ACM SIGCHI Bulletin*, Vol.17, No.4, ACM Press, 1996, pp.16-23.

[16] H. Gall, M. Jazayeri, and C. Riva, "Visualizing Software Release Histories: The Use of Color and Third Dimension", *Proc. ICSM'99*, pp. 99-108.

[17] A. Hamou-Lhadj and T. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces", *Proc. IWPC'02*, pp.159-168.

[18] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behavior of a Software System", *Proc. ICPC'06*, pp.181-190.

[19] J.A. Jones, R. Orso, and M.J. Harrold, "GAMMATELLA: Visualization of Program-Execution Data for Deployed Software", *Information Visualization*, Vol.3, No.3, 2004, pp. 173-188.

[20] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization", *Proc. ICSE'02*, pp.467-477.

[21] H. Kagdi, S. Yusuf, and J.I. Maletic, "On Using Eye Tracking in Empirical Assessment of Software Visualizations", *Proc. WEASELTech'07*, pp.21-22.

[22] G. Lommerse, F. Nossin, L. Voinea, and A. Telea "The Visual Code Navigator: An Interactive Toolset for Source Code Investigation", *Proc. INFOVIS'05*, pp.24-31.

[23] A. Marcus, L. Feng, and J.I. Maletic, "3D Representations for Software Visualization", *Proc. SoftVis'03*, pp. 27-36.

[24] A.V. Miranskyy, N.H. Madhavji, M.S. Gittens, M. Davison, M. Wilding and D. Godwin, "An Iterative, Multi-Level, and Scalable Approach to Comparing Execution Traces", *Proc. SIGSOFT/FSE*, pp.537-540.

[25] T. Okamura, B. Shizuki, and J. Tanaka, "Execution Visualization and Debugging in Three-Dimensional Visual Programming", *Proc. 8th Int'l. Conf. on Information Visualization*, pp.167-172.

[26] A. Orso, J.A. Jones, M.J. Harrold, and J. Stasko, "GAMMATELLA: Visualization of Program-Execution Data for Deployed Software", *Proc. ICSE'04*, pp.699-700.

[27] S.P. Reiss, "Visualizing Program Execution Using User Abstractions", *Proc. SOFTVIS'06*, pp.125-134.

[28] J. Rilling and S.P. Mudur, "3D Visualization Techniques to Support Slicing-based Program Comprehension", *Computers & Graphics*, 2005, Vol.29, No.3, pp.311-329.

[29] J.E. Roberts, "TraceVis: An Execution Trace Visualization Tool James Roberts", *M.S. Thesis*, University of Illinois at Urbana-Champaign, 2004.

[30] G.G. Robertson, S.K. Gard, and J.D. Mackinlay, "Information Visualization Using 3D Interactive Animation", *Communications of ACM*, Vol.36 , No.4, April 1993, pp.57-71.

[31] D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, J. Dill, S. Dubs, and M. Roseman, "Navigating Hierarchically Clustered Networks Through Fisheye and Full-zoom Methods", *ACM Trans. on Computer-Human Interaction*, Vol.3, No.2, June 1996, pp.162-188.

[32] J. Stasko, J. Domingue, M.H. Brown, and B.A. Price (Ed.), Software Visualization: Programming as a Multimedia Experience, MIT Press, Cambridge, MA, 1998.

[33] M.D. Storey, K. Wong, and H.A. Müller, "Rigi: A Visualization Environment for Reverse Engineering", *Proc. ICSE'97*, pp.606-607.

[34] C. Ware and P. Mitchell, "Visualizing Graphs in Three Dimensions", *ACM Trans. on Applied Perception*, Vol.5, No.1, 2008, pp.2:1-15.

[35] C. Ware and G. Franck, "Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions", *ACM Trans. on Graphics*, Vol. 15, No.2, 1996, pp.121-140.

[36] C. Ware, and G. Franck, "Viewing a Graph in a Virtual Reality Display is Three Times as Good as a 2D Diagram", *Proc. IEEE Symposium on Visual Languages*, October 4-7, 1994, pp.182-183.

[37] N. Wilde, J.A. Gomez., T. Gust, and D. Strasburg, "Locating User Functionality in Old Code", *Proc. 8th ICSM'92*, pp.200-205.

[38] C. Zhao and K. Zhang, and Y. Lei, "Abstraction of Multiple Executions of Object-Oriented Programs". *Accepted as a poster in Proc. 24th Annual ACM Symposium on Applied Computing*, 2009.

[39] K. Zhang (Ed.), Software Visualization - From Theory to Practice, Kluwer Academic Publishers, Boston, April 2003.