Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views

Danny Holten[1], Bas Cornelissen[2], and Jarke J. van Wijk[1]

[1]*Technische Universiteit Eindhoven – d.h.r.holten@tue.nl, vanwijk@win.tue.nl*
[2]*Delft University of Technology – s.g.m.cornelissen@tudelft.nl*

Abstract

One way of gaining understanding of a software system is the analysis of dynamic information, i.e., program execution traces. A problem regarding the analysis of such traces is the fact that these are often extremely large: hundreds of thousands and even millions of calls within a single trace are no exception. To aid a user in navigating and understanding these vast amounts of information, we propose to visualize execution traces using two linked views. One is an element interaction view that shows part of a trace, i.e., those calls within a specific time window, using Hierarchical Edge Bundles (HEBs). The other is a more detailed Massive Sequence View that is synchronized with the HEB View and offers detailed information regarding the temporal location and interleaving of the calls within the current time window. This view also introduces a novel way of antialiasing based on the importance of calls. In this paper, we focus on the technical aspects of the visualization and rendering techniques that are necessary to produce scalable visualizations using HEBs and Massive Sequence Views.

1. Introduction

Within software engineering, it is important to understand a system's behavior in order to perform specific maintenance tasks, like change requests.

It is important to support a software engineer (SE) in performing such program understanding tasks by facilitating the program understanding process, since this process is known to be very time-consuming according to Corbi [4]: approximately 50% of the time allocated for maintenance tasks is spent on acquiring system knowledge.

To aid an SE in program understanding tasks, it is often useful to provide the SE with more information than can be obtained by simply analyzing the static source code. Providing an SE with dynamic information, i.e., program execution traces gathered from a running program, has the benefit that new information becomes available – object identities, occurrences of late binding, and polymorphism – that would have been difficult to acquire if only static analysis had been used.

However, one of the major problems of performing dynamic analysis is the vast amount of information that has to be processed by an SE. Even a single execution trace corresponding to a couple of minutes of execution time of a relatively small program can contain hundreds of thousands or even millions of calls. Uncompressed execution traces can thus require hundreds of megabytes or even several gigabytes of storage space [21].

To provide an SE with a clear and concise representation that allows him or her to quickly gain insight in such amounts of data, we propose to use a visualization approach comprised of two linked views. One is an element interaction view that shows the calls within the current time window of the execution trace using Hierarchical Edge Bundles (HEBs) [10]. The other is a more detailed Massive Sequence View that is synchronized with the HEB View and offers detailed information regarding the temporal location and interleaving of the calls within the current time window. These views allow an SE to interactively explore an execution trace, to look for recurring patterns, and to inspect details on demand while providing a high degree of scalability.

In a companion paper [5], we characterized our approach using the task oriented framework introduced by Maletic et al. [14]. We showed how our OpenGL-based tool implementation called EXTRAVIS [1] – EXecution TRAce VISualizer – can be used by software developers

and (re-)engineers to perform trace exploration and phase detection, feature location, and feature comprehension, using three extensive case studies on an academic, an open source and an industrial software system.

In this paper, on the other hand, we focus on the technical aspects of the visualization and rendering techniques that are necessary to produce scalable visualizations using HEB Views and Massive Sequence Views. We furthermore introduce a novel way of visualizing calls within the Massive Sequence View. This technique guarantees the visibility of outlier calls, i.e., calls that are not frequently present within a certain time interval, when visualizing more than hundreds of thousands of calls.

Section 2 contains an overview of related work, followed by a global description of EXTRAVIS in Section 3. Sections 4 and 5 contain an in-depth description of the visualization and rendering techniques necessary to produce the HEB and the Massive Sequence View, respectively. Finally, Section 6 contains a summary of the proposed techniques and possible directions for future work.

2. Related Work

Most of the current methods for visualizing execution traces are based on two representations. One is a 2D or 3D node-link graph in which nodes represent elements, e.g., classes and objects, and edges (links) represent calls between elements. The other is a 2D, space-reduced representation in which calls between elements are time-ordered along one axis and the software hierarchy is depicted along the other axis; an Information Mural-like representation [11, 12]. Such a representation is, in essence, a zoomed-out, visually minimalistic version of a UML sequence diagram providing improved scalability.

Examples of trace visualization techniques that use node-link representations are the reference pattern view and class communication graph by De Pauw et al. [17, 18], the object and class interaction charts by Lange et al. [13], TraceCrawler by Greevy et al. [9], and the communication and creation interaction views by Ducasse et al. [7].

Although such node-link representations work well for the visualization of calls within small software systems, they tend to become highly cluttered when large numbers of calls within large software systems are visualized. A way to address this is provided by Hierarchical Edge Bundles [10], a technique that visually bundles the edges within a node-link representation together. HEBs reduce visual clutter and also visualize implicit calls between parent elements within the software hierarchy that are the result of explicit calls between their respective child elements. We make use of this technique for our element interaction view.

Trace visualization techniques that use Information Mural-like representations are visually similar to the visualizations produced by the SeeSoft system developed by Eick et al. [8], which shows source code instead of an execution trace by mapping each line of code to a row of pixels. Examples of techniques that visualize execution traces using Information Mural-like techniques are the Execution Mural by Jerding et al [12], ALMOST by Renieris et al. [19], in which the visualization is mapped onto a spiral view, and the zoomed-out execution view by De Pauw et al. [18].

Often, the elements visualized using Information Mural-like representations are so densely packed that their height and/or width is reduced to the size of a single pixel. If even more information needs to be represented within a single view, multiple elements are mapped onto the same pixel (or pixel line) and guaranteed visibility of each individual element, especially outliers, becomes a problem. Most pixel-based representations mentioned above do not address this problem and suffer from visual aliasing as a result of this.

Jerding et al. address this issue with their Information Mural representation by incorporating a linear antialiasing (AA) method [11, 12], but as we show in Section 5, this only partially solves the problem. In the accordion drawing technique by Munzner et al. [3, 16], important elements are marked and drawn in an importance-based order; the most important elements are drawn first and elements that do not fit on the screen anymore due to lack of space are not drawn at all. The importance-based antialiasing (IBAA) technique by Moreta et al. [15] is somewhat similar to the method that we use for rendering our Massive Sequence View; it uses a model that adjusts the individual contribution of calls to an antialiased pixel line to make outliers more visible. However, they use a different weighting model and they do not use a sliding window on the trace data to make the determination of the importance of a call more noise resistant (see Section 5).

3. EXTRAVIS

This section provides a global description of EXTRAVIS, its two main views, and the way in which these views are synchronized and linked.

The main interface of EXTRAVIS is shown in Figure 1. The HEB and Massive Sequence View visualize the same trace time window of an execution trace obtained from running JHOTDRAW [2], a highly customizable Java framework for graphics editing. The views are linked in the sense that highlighting of one or more calls or hierarchy elements in one view results in highlighting of the corresponding calls or hierarchy elements in the other view (see Figure 2). Both views offer textual details on demand by showing detailed tooltip information when hovering above a call or a hierarchy element (Figure 2). Linking to the actual source code corresponding to a certain call or hierarchy element is provided by both views as well.

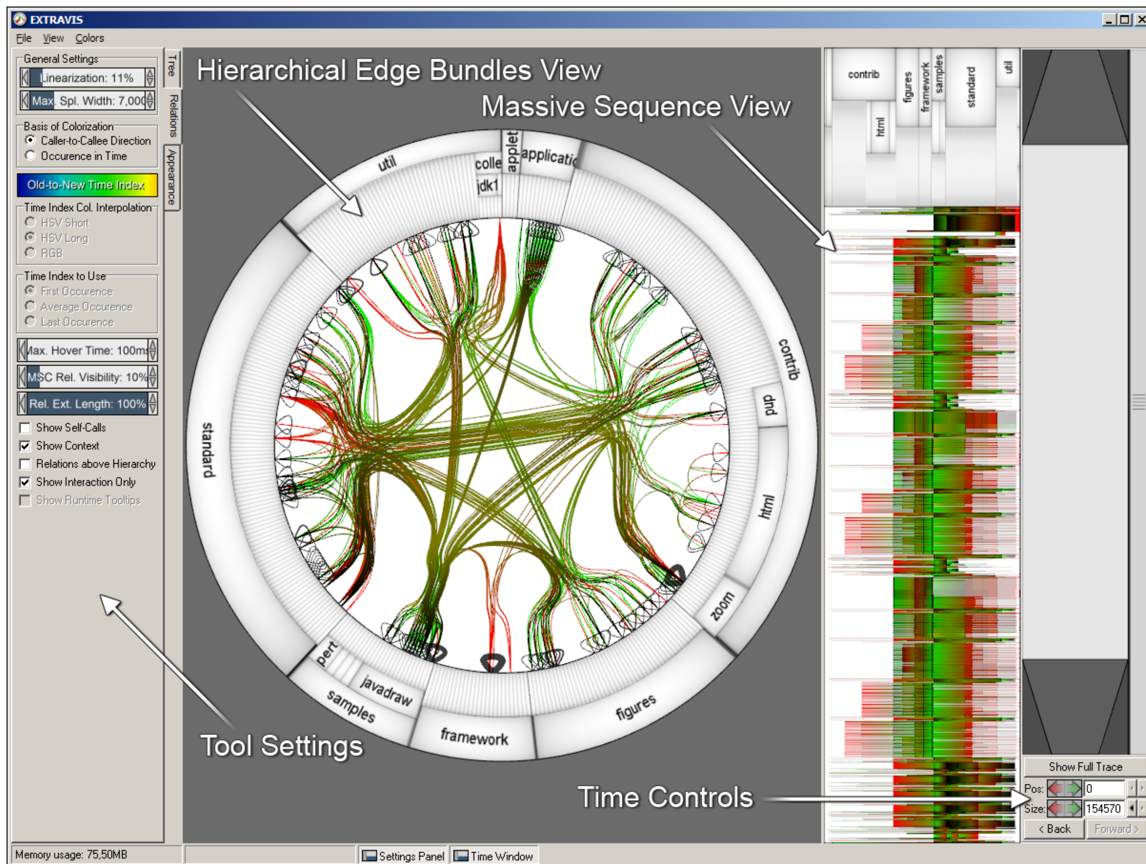

Figure 1. EXTRAVIS' **main interface showing an execution trace gathered from running** JHOTDRAW**. Recurring patterns are clearly shown within the trace – the creation of a new drawing and the insertion of five figures, executed three times in total (call direction: from green (caller) to red (callee)).**

3.1. Input Data

As input, EXTRAVIS needs a collection of parent-child relations that describe the hierarchy of the software system, e.g., the (sub)package-class structure in case of a Java program, and a collection of time-stamped call relations, e.g., Java class-to-class calls. Although we have been using trace data collected from Java programs, the input format is generic enough to enable the interpretation of parent-child and call relations collected from different programming languages as well, e.g., data collected from C++ programs.

In case of a Java program, we use a simple Perl tool to derive the system's class decomposition from its directory structure. This results in parent-child relations that define the system's structure in terms of (sub-)packages and classes.

The dynamic information is obtained by tracing a system's execution by monitoring for method invocations and registering the objects that are involved. We achieve this by extending the Java SDR framework from our earlier work [6]. The tracer registers unique objects, method names, information on the call sites, i.e., source-file names and line numbers, runtime parameters, and actual return values, and converts these events into an RSF file (Rigi Standard Format [20]).

3.2. Hierarchical Edge Bundles View

The HEB View shows the calls within the current time window using Hierarchical Edge Bundles; calls from one hierarchy element to another are visualized using a spline and color-encoded using a green-to-red gradient (by default) to indicate the caller-to-callee direction (see Figure 1). Spline thickness is used to indicate the number of calls from one hierarchy element to another within the current time window (see Figure 3). It is furthermore possible to switch to a visualization mode in which color is used to indicate the temporal location of a spline in time using a continuous color scale (Figure 3).

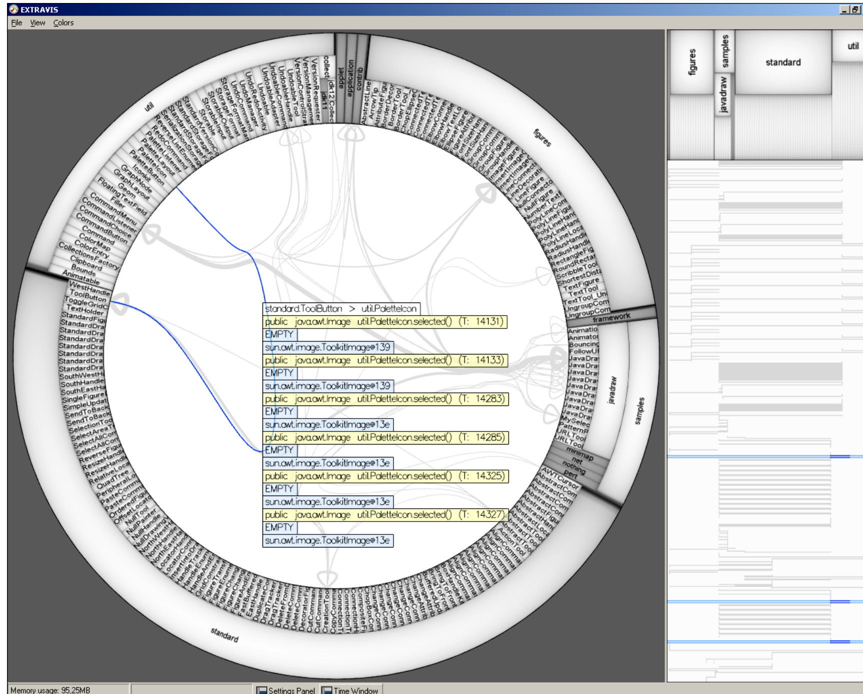

Figure 2. Highlighting visual elements in one view results in highlighting of the corresponding elements in the other view (hierarchy labels provide tooltips and can be scaled to increase readability).

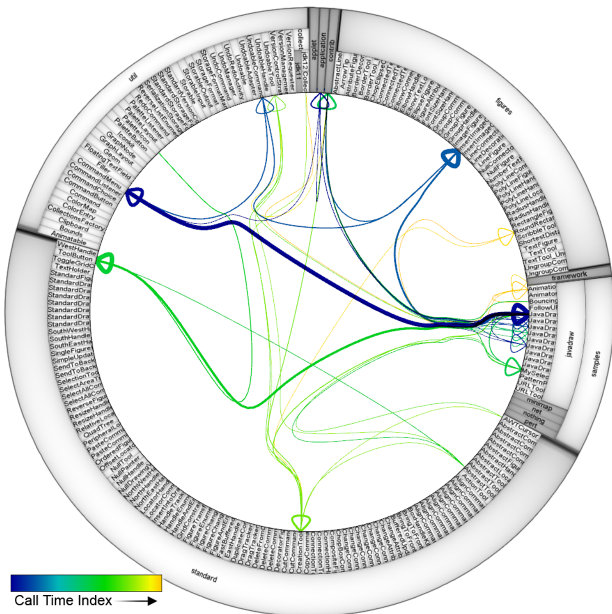

Figure 3. HEB View in temporal mode.

3.3. Massive Sequence View

Although the HEB View gives a good impression of the number of calls between hierarchy elements and the activity of certain parts of the software system, it is less suited for showing the exact temporal location and interleaving of calls, since it shows a “time-flattened” view of the current time window. To show detailed information regarding the temporal location and interleaving of calls, we use the Massive Sequence View.

The Massive Sequence View is a representation based on the Information Mural technique by Jerding et al. [11, 12]. The software hierarchy is depicted as an icicle plot at the top and calls between hierarchy elements are displayed as rectangles ordered from top (first call within the current time window) to bottom (last call within the current time window). Rectangles are color-encoded using a green-to-red gradient (by default) to indicate the caller-to-callee direction (see Figure 4).

4. Hierarchical Edge Bundles View

HEBs, as presented in [10], reduce visual clutter when dealing with large numbers of edges by using the layout provided by a tree visualization as a guide for bundling edges. Figure 5 illustrates how this is done by using part of a balloon tree layout as an example; $LCA(P_{Start}, P_{End})$ is the *least common ancestor* of P_{Start} and P_{End}. The approach is to use the path along the hierarchy between two nodes having a relation as the control polygon of a spline; the resulting spline

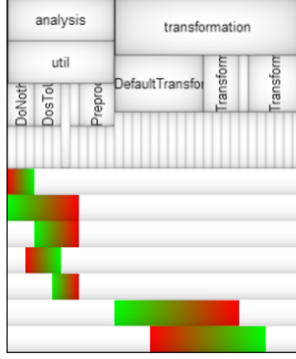

Figure 4. Massive Sequence View depicting the hierarchy as an icicle plot and calls as shaded rectangles (caller/callee = green/red).

is subsequently used to visualize the relation.

If this approach is used directly, ambiguity problems as shown in figure 6a may arise. The bundling strength β, $\beta \in [0,1]$, controls the amount of bundling by straightening the control polygon of the spline curve (Figures 6c and 6d). By varying the value of β, ambiguity problems can be reduced, as is illustrated in Figure 6e.

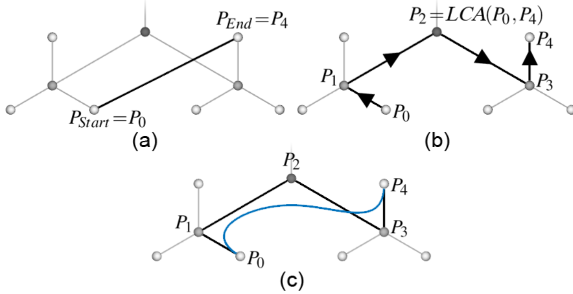

Figure 5. Bundling edges. (a) line connection between P_0 and P_4; (b) path along the hierarchy; (c) a spline depicting the connection using the path from (b) as control polygon.

Straightening of a control polygon is performed by straightening each of its control points P_i. These straightened control points P'_i are subsequently used as the new control polygon to generate the spline:

$$P'_i = \beta \cdot P_i + (1-\beta)(P_0 + \frac{i}{N-1}(P_{N-1} - P_0)), \qquad (1)$$

with
N : number of control points,
i : control point index, $i \in \{0,\ldots,N-1\}$,
β : bundling strength, $\beta \in [0,1]$.

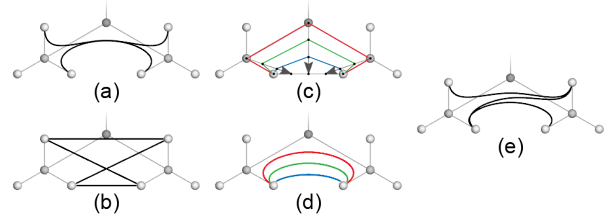

Figure 6. The bundle in (a) might contain each edge from (b); (c) and (d) illustrate the effect of β; (e) a high β value (0.8) can be chosen to retain bundles while still resolving ambiguity.

In case of bundles that are mainly comprised of splines having a similar direction, *minimum blending* can be used to draw these splines to spot individual splines within the bundle having an opposite direction (Figure 8). Minimum blending returns the minimum of two colors on a per-color-channel basis. Moreover, this blending mode – implemented in OpenGL using the `EXT_blend_minmax` extension – is commutative, resulting in identical visualizations regardless of the order in which splines are drawn.

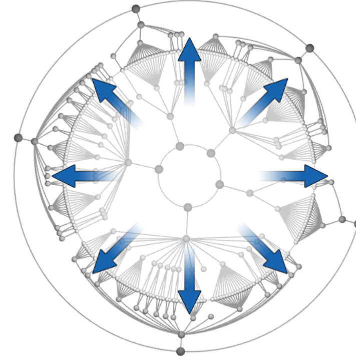

Figure 7. A radial tree layout is used for the inner circle and mirrored to the outside as the basis for an icicle plot. The structure of the inner layout is used to guide the bundling.

Figure 7 illustrates how the inner hierarchy that guides the bundling is constructed (invisible in final visualization) and how this hierarchy - when mirrored - serves as the basis for the outer icicle plot that depicts the hierarchy. The final visualization is depicted in Figure 8 and shows the benefits of HEBs over node-link graphs that use linear edges.

5. Massive Sequence View

In case of the Massive Sequence View, a problem arises if the number of calls to be displayed is larger than the

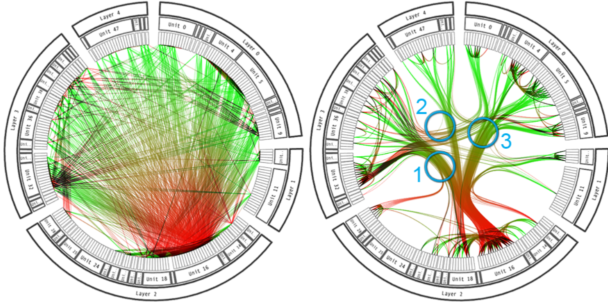

Figure 8. The HEB view (right) more clearly shows (1) main calling direction, (2) outliers, and (3) calls having an opposite direction.

number of available pixel lines; multiple calls need to be mapped onto a single pixel line in this case. This can be done through a linear AA technique; use as color a weighted average of the colors of the calls that need to be mapped onto a single pixel line. Figure 9 illustrates this procedure using an example in which five calls need to be mapped onto three pixel lines.

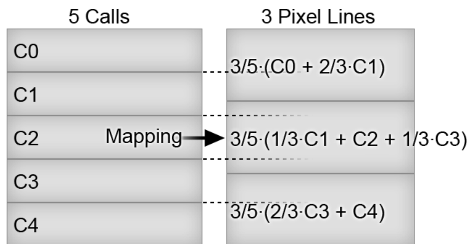

Figure 9. Mapping multiple calls onto a single pixel line is done through linear AA.

Although linear AA is an improvement when compared to approaches that do not use this technique, a problem becomes apparent if the technique is used to map multiple, (mainly) identical calls containing one or a couple of outlier calls onto a single pixel line. In this case, linear AA results in the (near) invisibility of outlier calls, even when only a modest number of calls is being blended; Figure 10 illustrates this using an example in which 16 calls are mapped onto a single pixel line; 15 calls from element B to element D and a single outlier call from element A to element C. Linear AA results in the outlier call being barely visible ($\frac{1}{16} \cdot 100 = 6.25\%$ visibility).

To increase the visible contribution of outlier calls, we introduce a model that modifies the weight of a call c based on its context, i.e., its neighboring calls. The context is defined by a window w that is centered on c. We determine the number of occurrences of c within w and subsequently normalize this value by dividing it by $|w|$, i.e., the total number of calls within w. This leaves us with the frequency of c, which is used to determine the visible contribution of call c. The complete model is as follows:

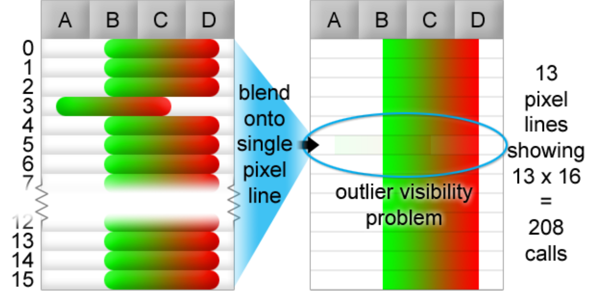

Figure 10. Using linear AA results in the (near) invisibility of outlier calls.

$$W_i = f_i^p, \tag{2}$$

with

W_i	:	contribution of call i,
p	:	contribution power (global), $p \in [-\infty, \infty]$,
f_i	:	frequency of call i within w, $f_i \in [0, 1]$.

Determining the contribution of each call thus becomes a prepass procedure: for each call i within the trace, its contribution W_i is determined using Equation 2 and stored for use during rendering. Furthermore, the window size $|w|$ should be sufficiently large to suppress noise; in practice, a window size of 25 is sufficient to accomplish this.

During rendering, the linear AA technique described above and depicted in Figure 9 is extended to take the call contributions W_i into account as well, resulting in IBAA (see Figure 11).

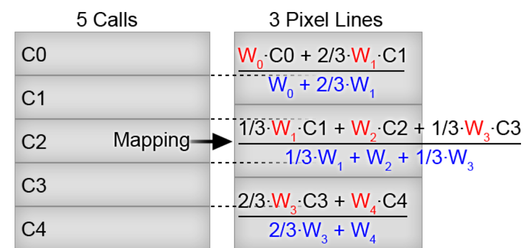

Figure 11. Linear AA is extended to IBAA by taking the call contributions W_i into account.

As is illustrated in Figure 11, the rendering of n calls onto a pixel line l depends on the fractionality F_s of the first call, the fractionality F_{s+n-1} of the last call, and the call contributions W_i, i.e.:

$$l = \frac{\sum_{i=s}^{s+n-1} (F_i \cdot W_i \cdot c_i)}{\sum_{i=s}^{s+n-1} (F_i \cdot W_i)}, \quad (3)$$

with

F_i : fractionality of call i, $i \in \{s, \ldots, s+n-1\}$:
$F_i \in [0,1]$ if $i = s$ or $i = s+n-1$,
$F_i = 1$ otherwise,

c_i : call i.

In theory, the value of the global contribution power p in Equation 2 can be set to any value in the range $[-\infty, \infty]$; a value of 0 results in linear AA (no discernable IBAA), positive values visually exaggerate the information by giving even more weight to calls that are already frequently present, and negative values give more weight to calls that are not frequently present, i.e., outlier calls.

In practice, values outside a range of approximately $[-5, 5]$ do not result in discernable visual changes anymore; it is therefore advisable to limit the value of p to this range. Figure 12 illustrates our IBAA method using $p = -1.0$.

6. Summary and Future Work

To tackle the scalability problem that is often associated with dynamic analysis due to the huge amounts of trace data, we proposed a scalable visualization solution centered around two synchronized and linked views of an execution trace.

The HEB View shows the calls within the current time window using Hierarchical Edge Bundles. The Massive Sequence View is synchronized with the HEB View and offers detailed information regarding the temporal location and interleaving of the calls within the current time window.

In a companion paper [5], we showed how our tool can be used to perform trace exploration and phase detection, feature location, and feature comprehension using three extensive case studies.

In this paper, we focused on the technical aspects of the visualization and rendering techniques that are necessary to produce scalable visualizations using HEB and Massive Sequence Views. To summarize, we showed how:

- HEBs can be used to reduce the visual clutter that is often present when using node-link representations to visualize a program's calling behavior. We furthermore showed how minimum blending can be used to more clearly visualize outlier calls within a HEB view;

- Both views can be used to complement each other: the HEB View gives a good impression of the number of calls between hierarchy elements and the activity of certain parts of the software system, while the Massive Sequence View shows detailed information regarding the temporal location and interleaving of calls;

- IBAA can be used to visualize calls within the Massive Sequence View in such a way that the visibility of outlier calls is guaranteed when visualizing more than hundreds of thousands of calls.

6.1. Future Work

Possible directions for future work surfaced as a result of presenting EXTRAVIS to the industrial partners within our project and due to our own use of EXTRAVIS. Those that we regard as the most useful and interesting are as follows:

- Pattern recognition within execution traces: marking certain parts of a trace could automatically reveal (near) identical parts at other locations within the trace. Pattern recognition could also be used to detect outliers, i.e., infrequent calls or calls that are not allowed to exist for some reason, e.g., because the elements belong to non-contiguous architectural layers;

- Execution trace comparison: comparing two or more traces in a visual way to quickly get an idea of where changes within a trace are located. Furthermore, observing two traces side by side might make feature location easier.

- Comparison of software hierarchies in support of execution trace comparison: two execution traces that are to be compared might differ from each other not only because the execution scenario has changed, but also because the software hierarchy has changed, e.g., as a result of classes that have been added, moved, or deleted. We are currently working on a way to visually compare two hierarchies and to interactively explore the differences between them.

Finally, we have made EXTRAVIS available online [1] so that users can try our tool and share their experiences with us, e.g., by providing feedback regarding new directions for future work or suggestions for improvement.

7. Acknowledgements

This project is funded by the Netherlands Organization for Scientific Research (NWO) Jacquard program under research grant no. 638.001.408 (*Reconstructor Project*).

References

[1] EXTRAVIS: EXecution TRAce VISualizer, `http://www.swerl.tudelft.nl/extravis/`.

[2] JHOTDRAW, `http://www.jhotdraw.org/`.

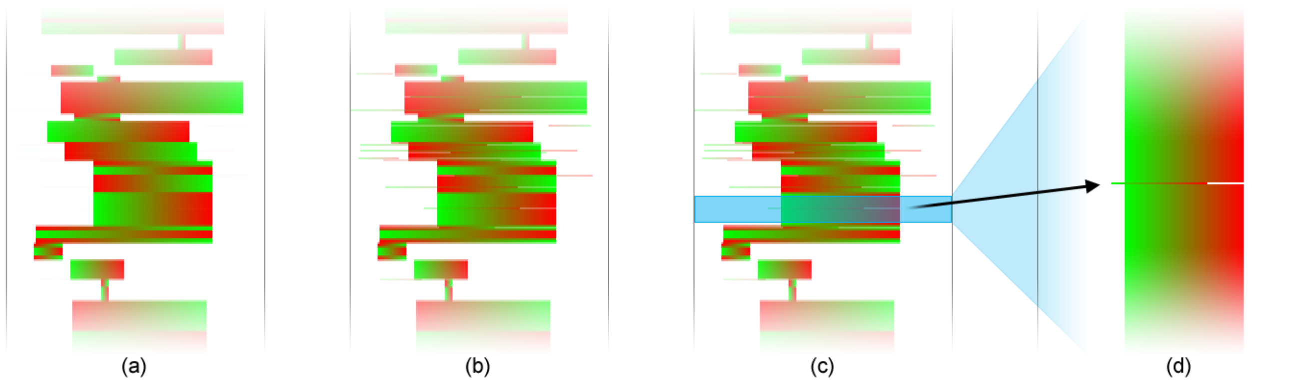

Figure 12. (a) linear AA results in outliers becoming (nearly) invisible; (b) IBAA ($p = -1.0$) preserves their visibility; (c) IBAA allows the selection of part of a trace containing outliers that would have normally been missed; (d) these can subsequently be inspected in more detail.

[3] D. Beermann, T. Munzner, and G. Humphreys. Scalable, robust visualization of large trees. In *Proceedings of the 3rd Eurographics / IEEE VGTC Symposium on Visualization (EUROVIS'05)*, pages 37–44. Eurographics / IEEE, 2005.

[4] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[5] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, A. van Deursen, and J. J. van Wijk. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 2007. To appear.

[6] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proceedings of the 11th IEEE European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 213–222. IEEE, 2007.

[7] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of the 8th IEEE European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318. IEEE, 2004.

[8] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. SeeSoft - A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[9] O. Greevy, M. Lanza, and C. Wysseier. Visualizing live software systems in 3D. In *Proceedings of the 3rd ACM Symposium on Software Visualization (SOFTVIS'06)*, pages 47–56. ACM, 2006.

[10] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of INFOVIS'06)*, 12(5):741–748, 2006.

[11] D. F. Jerding and J. T. Stasko. The Information Mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):257–271, 1998.

[12] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the 19th ACM International Conference on Software Engineering (ICSE'97)*, pages 360–370. ACM, 1997.

[13] D. B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, 1997.

[14] J. I. Maletic, A. Marcus, and M. L. Collard. A task oriented view of software visualization. In *Proceedings of the 1st IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 32–40. IEEE, 2002.

[15] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *Proceedings of the 5th Eurographics / IEEE VGTC Symposium on Visualization (EUROVIS'07)*. Eurographics / IEEE, 2007. To appear.

[16] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou. TreeJuxtaposer: Scalable tree comparison using focus+context with guaranteed visibility. In *Proceedings of the 30th ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'03)*, pages 453–462. ACM, 2003.

[17] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. *Visualizing the Execution of Java Programs*, pages 151–162. Springer-Verlag, 2001.

[18] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 219–234, 1998.

[19] M. Renieris and S. P. Reiss. ALMOST: Exploring program traces. In *Proceedings of the 5th ACM Workshop on New Paradigms in Information Visualization and Manipulation (NPIVM'99)*, pages 70–77. ACM, 1999.

[20] K. Wong. The Rigi user's manual - version 5.4.4. http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf (last visited January 30th, 2007), 1998.

[21] A. Zaidman. Scalability solutions for program comprehension through dynamic analysis. In *Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 327–330. IEEE, 2006.