Execution Trace Visualization in a 3D Space

Philippe Dugerdil & Sazzadul Alam
Department of Information Systems
HEG-Univ. of Applied Sciences,
7, route de Drize, CH-1227 Geneva, Switzerland
philippe.dugerdil@hesge.ch, sazzadul.alam@hesge.ch

Abstract

In program execution visualization, we wish to show where the action is unfolding when some high level use-case is played on the system. However, an important problem is to deal with the volume of data to display. One solution is to represent the architecture of the software in some familiar form. Then we proposed to represent it as a modern city, with buildings and districts, in a 3D space. In this paper we present the way the program execution is represented in such a space and we show how the understanding of this execution is facilitated. Since the display of the full execution trace is impossible because of its sheer size, we developed a trace segmentation technique. Then an information filter, called class omnipresence, is applied to further reduce the information to display. Finally, the resulting sequence of segments can be "played" on the screen, like in a movie.

Keywords – Software visualization, reverse engineering, dynamic analysis, trace segmentation.

1. Introduction

In reverse engineering projects, understanding the inner working of an industrial-size software from the execution trace represent a huge endeavor. In fact, the first problem encountered when analyzing the execution trace of some real-world program is the amount of information to process. Usually, a trace file can be as big as hundreds of thousands if not millions of events. In case of a loop or a recursion for example, the file may contain thousands of similar events or thousands of similar blocks of events. For an engineer to analyze such a trace file, this enormous quantity of information must be reduced. In the case of frequency spectrum analysis [2][1], the information is summarized by counting the occurrences of similar events. This is OK if the only information one wants to extract is the global frequency of the events. Another way to cope with the size of the trace is to compress it. Usually the compression techniques exploit trace element redundancies [9].

Besides, the traditional way to represent the result of the trace analysis is to display it as a flat graph. But a much less explored technique is to display the sequence of events of the program execution like a movie. However, this raises two problems:

- In what form to represent the thousands of software events involved in the trace "movie"?
- How to display such a big amount of information without having the "movie" to last hours?

Since software is a formal and abstract construct, there is no "natural" representation for it. Today's industrial software systems are tremendously complex, with size counting in millions of lines of code. One way to cope with complexity is to represent information hierarchically in several levels of abstraction and attach metrics to the displayed elements to represent aggregated information on their contents [12]. However, if the unsophisticated display of a few dozen of classes in a diagram can provide some insight to the structure and behavior of a system, we must find a better way to represent thousands of classes or components. In other words the chosen visual representations should be easily interpretable by the user, to let him concentrate on the program execution information. Then we exploited a familiar visual metaphor to help the user grasp a myriad of information in a single view: the Software City [1][3][16][17]. In this representation the classes and files are represented as buildings and the relationships as solid pipes between the buildings. The buildings are arranged in a 3D landscape where the city districts represent the packages in which the classes are located. Then the dimensions and appearance (texture) of the buildings are mapped to metrics. This system is called "Evospaces".

To represent the trace information, we extended the Software City metaphor, to represent the city at night. Then, the illuminated buildings are the ones in which “the people are working”. In summary we now have two views of the software city: the day view that is suited to the representation of structural and architectural information and the night view, which is well suited to represent program execution. But we must solve the problem of the quantity of information to represent. Then we used a segmentation technique coupled with statistical analysis to reduce the number of “frames” (movie images) to display. In our experiments, we used Mozilla as a test bench. All the figures are taken from this experiment. This paper is organized as follow. Section 2 presents the main concepts and implementation of the EvoSpaces environment that displays software systems in 3D. Section 3 presents the trace segmentation and statistical analysis techniques we used to reduce the trace information to display. Section 4 presents application of these techniques to a software city representing Mozilla. Section 5 presents the related work and section 6 concludes the paper and gives an outlook of the future work.

2. The EvoSpaces system

In the EvoSpaces system, the classes and files are represented as buildings and the relationships as solid pipes between the buildings. Metrics are used to set the height and textures of the buildings. However, a linear mapping between the metrics and the dimensions of the building is not very informative for the user. In fact it is very hard to visually compare building having a slight difference in height, especially in 3D. Then we decided to sort the metric values in three categories represented as 3 building heights and three building textures. (fig. 1).

Figure 1. Glyphs used in Evospaces

The objects representing the entities must now be distributed in the 3D space using some layout (topology). For example we could arrange the entities in rows and columns, in concentric circles, in spiral, etc. After having tried several layouts we realized that the most useful and the most easily interpretable distribution of the software elements in space is to represent containment. In figure 2 the containment (directories or package hierarchies) is represented as rectangular zones on the ground. The darker the zone, the higher in the containment hierarchy.

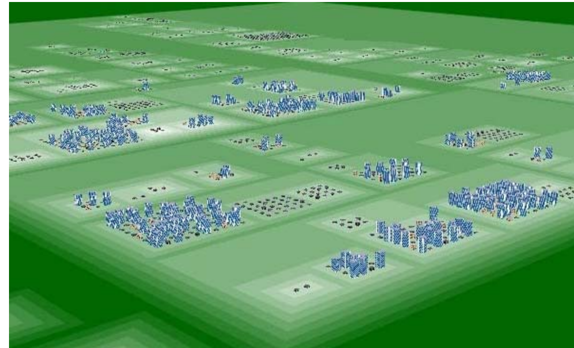

Figure 2. Containment layout

In this space the user can navigate between, and interact with, the displayed objects. Moreover the relationships can be displayed as solid pipes between the buildings (fig. 3). A moving red segment is displayed on each pipe to represent the directionality of the flow of information.

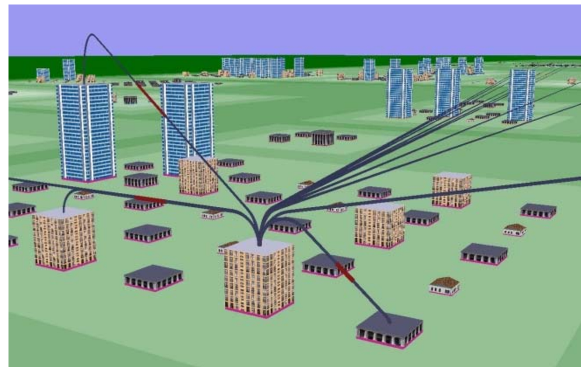

Figure 3. Relationships between classes

3. Trace segmentation

3.1 Need to segment the trace

To cope with the quantity of information of the execution trace, the current trend in literature is to compress it to remove the redundancies. There are two categories of compression algorithms: lossy and lossless. Basically the lossless algorithms try to identify and eliminate the recurring patterns in the trace. In the compressed trace, such an algorithm keeps only one occurrence of the pattern and replaces the others by a reference to the first. The simplest example

of this technique is when a set of contiguous similar lines are replaced by a single occurrence associated with the number of repetitions. As an example of a much more sophisticated algorithm, Hamou-Lhadj and Lethbridge took the *common subexpression algorithm* [8][10] used to identify patterns in DNA analysis. The lossy algorithms, i.e. the ones that do not preserve the information after compression, use approximate match techniques to find the recurring patterns to eliminate. For example De Pauw et al. [5] list half a dozen of such lossy techniques.

Since our work focuses on identifying the classes or files that are specific to the implementation of a given business function and the way it is implemented, our concern is less to find general techniques for trace compression than to actually "see" the involvement of classes and files in the trace as time passes. This visualization is based on the metaphor of the city buildings seen at night: the one in which people are working are the ones that are illuminated. Figure 4 shows a part of the city during day time and figure 5 presents the same buildings at night. The colored buildings show where the program execution happens.

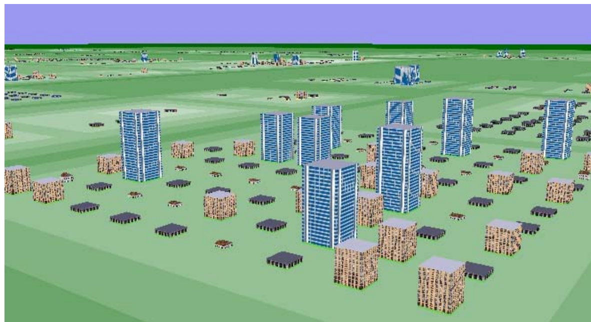

Figure 4. Day view

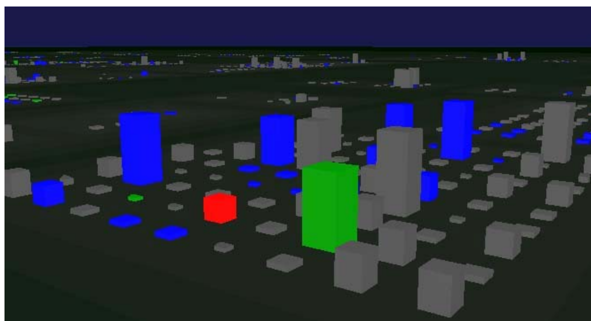

Figure 5. Night view

However, it is clear that one cannot present the sequence of building illuminations corresponding to each event in the trace. This sequence would take hours to display. Therefore, we developed a segmentation technique to summarize the information.

3.2 Segmentation technique

To segment the execution trace we split it up into contiguous segments of a given duration (width). Then we compute statistics in each of the segments (fig. 6). In this figure the text displayed vertically represents the events (method calls) in the trace.

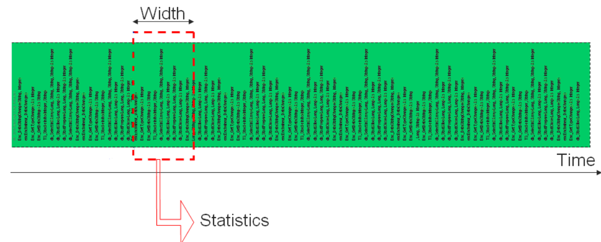

Figure 6. The segmentation of the trace

Starting from this segmentation idea we defined two views for the trace:

- The macroscopic view that represents the trace as the sequence of segments;
- The microscopic view that represents the sequence of events in a selected segment.

In the macroscopic view each "frame" (image) of the movie represents one segment. Then, in each segment, one computes the number of occurrences for each class i.e. the number of method executions (events) for the class. Next, this number is associated to one of the three user-defined categories of occurrences, mapped to a specific color. For example, the occurrences of a given class below 10 could be displayed in blue, the occurrences between 10 and 49 in green and the occurrence above 50 in red (fig. 7).

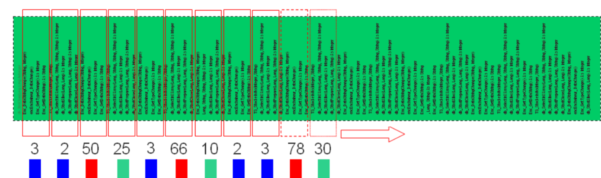

Figure 7. Class occurrences mapped to colors

Finally, each building in a "frame" (image) is illuminated using this color (fig 5). The resulting "movie" is obtained by the sequential display of the illuminated building for each segment. With this representation we can observe, by looking at the whole landscape and the color of the buildings, what are the active regions at any moment in time and what are the most active classes.

In the microscopic view the time scale is reduced to focus on a single segment as shown in figure 8. The method calls between the classes in the segment are

sequentially displayed by solid pipes between the buildings representing the classes (fig.9). While the increment of time in the macroscopic view is one segment, the increment of time in the microscopic view is the elementary invocation of a single method call (event). However, the buildings keep the same colors in both views.

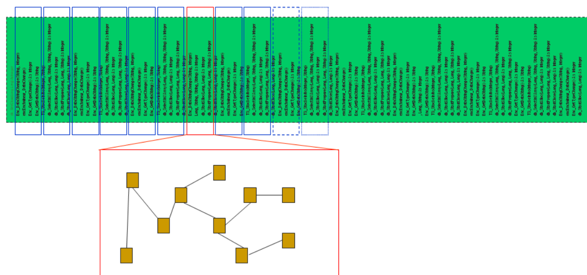

Figure 8. Events belonging to a segment

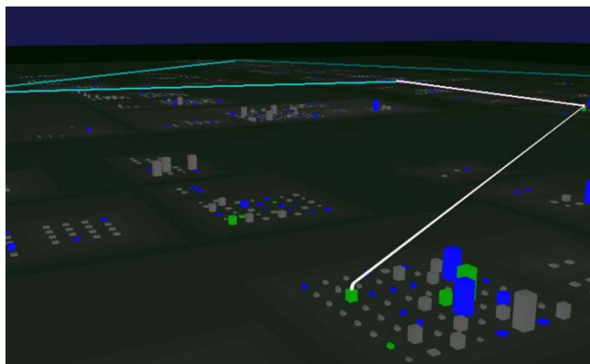

Figure 9. Microscopic view

3.3 Filtering uninteresting classes

Since we want to understand the role of the classes through the visualization of the program execution, we must relate the classes to the externally observable behavior of the program. Therefore since this behavior changes while the program is running, we naturally focus on the classes that appear in specific locations in the trace. In other words, the classes that are found everywhere in the trace cannot be linked to some specific business function. In fact these classes often have a transversal responsibility implementing some *aspect* of the software like logging, security check and the like. By analogy with signal theory, these classes are considered as *noise* and must be filtered out.

Traditional frequency analysis work defines a class frequency simply as the number of times it appears in the execution trace [2][1]. However, to identify the noise, we cannot simply count the occurrences of some classes but we must know where these occurrences happen in the trace. For example, a class that is found 10’000 times in the first 20% of the trace will be considered differently from a class that is found 10 times in 1000 evenly distributed locations. To use this idea with our Software City metaphor we could say that a building that stays illuminated all night long does not provide much information on when the people are working. A class whose location is evenly distributed throughout the trace is called *temporally omnipresent* [6]. By removing the omnipresent classes, one greatly simplifies the visualization of the trace on both the macroscopic and microscopic view.

4. Displaying the segmented trace

4.1 Execution trace storage in the database

The list of methods called during an execution of the analyzed software (the execution trace) is stored in a table of the EvoSpaces database. Each table’s row holds a sequence number, the identifier of the calling class, the identifier of the called class, the identifier of the method called and the level of the call. The sequence number let us know exactly in what order the methods have been invoked. Since we would like our system to display the “movie” of the trace smoothly, the number of class occurrences in a segment must be computed offline. This can be done for several values of the segmentation window width (fig.6). The results are stored in another table. To visualize the number of occurrences of each class in the segment, the user must define the occurrences categories: high, medium and low and assign them colors. This is done using a color mapper tool integrated in the Evospaces environment.

4.2 Playing the movie

When switching to the trace visualization mode, the night falls on the software city, the sky and the ground get dark and the buildings lose their textures. To navigate through the segments of the execution trace, a panel appears on the top of the window with several controls. The left part is used to select an existing segmented trace or to create a new one. The right part contains the controls used to display the “movie” associated to the selected trace. They look like the controls used to display a video (fig. 10). On the top, the arrowheads pointing to the right (the “play” button) together with the associated slider represent the Macroscopic view controls. It is used to launch the sequential display of the frames (segments) of the “movie”. In this case, the top slider moves according to the position of the frame actually displayed in the full trace. Moreover, one can set the

position of the top slider manually to set the frame to display. The user can enter the time in milliseconds each frame should be displayed. This sets the speed of the “movie”. When the Macroscopic view is in pause mode, the user can interact with the bottom slider that, together with the associated “play” button represents the Microscopic view controls. If the user clicks on this “play” button, the system displays the sequence of method calls in the current segment (frame). Then, the user can easily switch from the macroscopic view of a frame to the microscopic view of the execution trace. Finally, an effect of visual persistence of the display is implemented. This lets the user see the calls between two classes during a predefined number of frames (up to 10) with a decreasing level of luminosity as shown in figure 11. In the latter we show the microscopic view with a set of calls between classes. Each of the calls is displayed with a specific level of luminosity. The lower the luminosity level, the older in time.

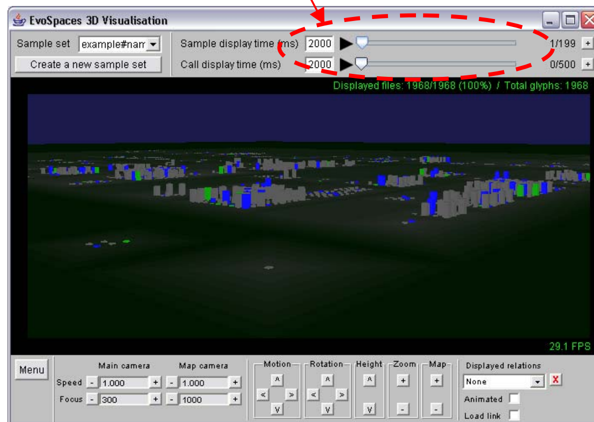

Figure 10. The “night” mode in Evospaces

5. Related work

Graphical representations of software have long been accepted as comprehension aids. The PolymetricViews system of Lanza [13] was a first step in presenting multiple metrics on the same 2D view. The use of 3D views to represent software architecture has been advocated by Feijs L. and De Jongin [7]. However they focused on the representation of the relations between modules, themselves represented as Lego bricks distributed in the 3D space. They did not investigate the use of “familiar” metaphors like the city. Langelier et al. [11] presented a visualization of software quality made of 3D boxes representing classes, whose dimensions are mapped to quality metrics. But they did not exploit the city metaphor to ease the interpretation of the view.

Very few authors have worked on a segmentation technique for trace analysis. One pioneering work is the one of Chan et al. [4] to visualize long sequence of low level Java execution traces in the AVID system (including memory event and call stack events). But their approach is to *sample* the trace i.e. selectively pick information from the source (the call stack for example) to limit the quantity of information to process.

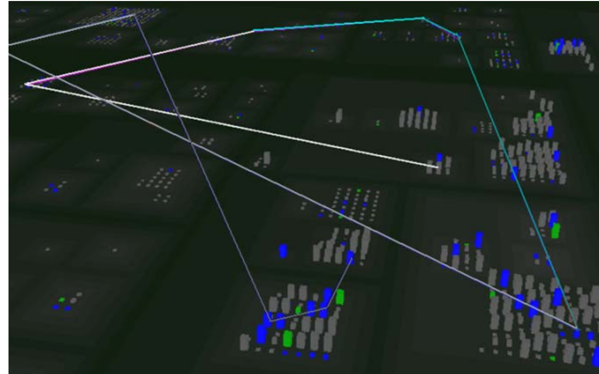

Figure 11. The calls with persistence

Our approach differs in that we segment the full trace to summarize it: we split the trace into consecutive “analysis windows” (segments) on which we apply statistics. Our noise reduction technique bears some similarity to the work of Hamou-Lhadj and Lethbridge on selective tracing and removal of the calls to utility classes [10]. However, in this work, utility classes are defined as classes whose methods are called from many other classes in the static call graph (the well known *omnipresent* classes of the pioneering Rigi system [14]). Then Hamou-Lhadj and Lethbridge define the utility hood metrics by counting the incoming and outgoing edge from the static call graph. Although their rationale for using the static call graph is sound, we claim that these metrics should be complemented by a dynamic measure we call “temporal omnipresence” [6].

6. Discussion and conclusion

In this paper we presented the trace visualization techniques we implemented in the EvoSpaces reverse-engineering tool. In fact, the execution trace is displayed like a movie using a trace segmentation technique. This “movie” allows the developer to see where the activity of the system is unfolding while it is running. Then the developer can focus on some particular moment in time and investigate the low-

level calls sequence between classes or files. To filter the quantity of information displayed, we introduced the concept of "omnipresent class" that let us remove all the classes that are not specific to some specific business function. As an experiment and proof of concept, we showed the use of these 3D techniques to represent a substantial part of a very large system, Mozilla, together with its execution trace. This experiment showed us that the display of a segmented trace like a movie is a very effective way to identify the areas in the code that are active at any moment in time. In fact, this representation allows the user to keep a global overview of the system (the Macroscopic view) while he is concentrating on some very specific part of the execution trace (the Microscopic view). The main contributions of this paper is to show how the Software City metaphor can be extended to represent execution traces in a condensed way and what techniques can be implemented to reduce the quantity of information to visualize. EvoSpaces is implemented in Java under Eclipse. As further research, we are investigating the relationship between the static and dynamic code analysis. Especially, we are studying the mixed use of static and dynamic information when displaying the buildings in Evospaces.

7. Acknowledgements

We gratefully acknowledge the financial support of the Hasler Foundation for the "Evospaces – Multi-dimensional navigation spaces for software evolution (project MMI-1976).

8. References

[1] Alam S., Dugerdil Ph. – EvoSpaces: 3D Visualization of Software Architecture. *Proc. of the Int. Conf. on Software Engineering and Knowledge Engineering (SEKE07),* 2007.

[2] Ball T. – The Concept of Dynamic Analysis. *Proc. 7th European Software Engineering Conf. (ESEC'99),* 1999.

[3] Boccuzzo S., Gall H., CocoViz: Towards Cognitive Software Visualizations. *Proc. of the IEEE Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2007.*

[4] Chan A., Holmes R., Murphy G.C., Ying A.T.T. - Scaling an Object-oriented System Execution Visualizer through Sampling. *Proc. of the IEEE Int. Workshop on Program Comprehension (ICPC'03),* 2003.

[5] De Pauw W., Lorenz D., Vlissides J., Wegman M. - Execution Patterns in Object-Oriented Visualization. *Proc. of the USENIX Conf. on Object-Oriented Technologies and Systems (COOTS),* 1998.

[6] Dugerdil Ph. - Using trace sampling techniques to identify dynamic clusters of classes. *Proc. of the IBM CAS Software and Systems Engineering Symposium (CASCON)* 2007.

[7] Feijs L., De Jongin R. – 3D Visualizations of Software Architectures. *Communications of the ACM (CACM) 41(12)*, Dec. 1998.

[8] Hamou-Lhadj A., Lethbridge T.C. – Compression Techniques to Simplify the Analysis of Large Execution Traces. *Proc. of the IEEE Workshop on Program Comprehension (IWPC),* 2002.

[9] Hamou-Lhadj A., Lethbridge T.C. - A Survey of Trace Exploration Tools and Techniques. *Proc of the Conf. of the Centre for Advanced Studies on Collaborative Research CASCON,* 2004

[10] Hamou-Lhadj A. Lethbridge T. – Summarizing the Content of Large Traces to Facilitate the Understanding of the Behavior of a Software System. *Proc. of the IEEE Int. Conf. on Program Comprehension (ICPC'06*), 2006.

[11] Langelier, G., Sahraoui, H., and Poulin, P. - Visualization-based analysis of quality for large-scale software systems. *In Proc. of the IEEE Int. Conf. on Automated Software Engineering ASE '05*. 2005.

[12] Lanza M., Marinescu R., Ducasse S. - *Object-oriented metrics in practice*. Springer, 2006.

[13] Lanza M., Ducasse S. – Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. *IEEE Trans. on Software Engineering 29(9),* Sept.2003.

[14] Muller H., Orgun M.A., Tilley S.R., Uhl J.S. - A Reverse Engineering Approach To Subsystem Structure Identification. *Software Maintenance: Research and Practice 5(4),* John Wiley & Sons, 1993

[15] Panas Th. - *A Framework for Reverse Engineering,* PhD Thesis, Växjö University, Dec. 2005

[16] Wettel R., Lanza M., Program Comprehension through Software Habitability. *Proceedings of the IEEE Int. Conf.e on Program Comprehension (ICPC),* 2007.

[17] Wettel R., Lanza M., - Visualizing Software Systems as Cities. *Proc. of the IEEE Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT),* 2007.