Scaling Dynamic Architectural Software Visualizations

Andrew Catton and Gail C. Murphy
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC, Canada V6T 1Z4
{catton, murphy@cs.ubc.ca}

Introduction

Architectural views of software systems—views in terms of entities at a higher-level of abstraction, such as subsystems, than typical source-level constructs, such as classes and methods—can be helpful to developers performing tasks on large and complex systems. For example, an understanding of the architecture of a system can help a developer reason about the kinds of changes a system may easily accept [GS93]. These views can be helpful because they abstract information that may be spread amongst numerous source-level constructs. This abstraction capability can help address a problem of scale faced by more detailed visualizers. Specifically, one view can represent a large amount of source code. However, even when employing this approach, many nontrivial systems have characteristics that make the information load on the visualization system and/or the user impractical, causing even dynamic architectural visualizers—visualizers oriented at displaying dynamic information collected from a system through an architectural view—to suffer scaling problems. Specifically, how can the visualizer represent and display information collected from long-running systems? As well, how can complex multi-threaded system information be visualized to maximize the relevant information presented while minimizing the 'noise'?

In this position paper, we provide an overview of a dynamic architectural visualization tool for object-oriented systems under development at the University of British Columbia, in conjunction with OTI, Inc [WMF+98]. We then briefly outline various facilities we are investigating to help address the problem of visualizing long-lived and multi-threaded systems.

Architectural Visualizer

Our visualization tool allows a developer to analyze the execution of a system off-line in terms of an architectural view chosen by the developer [WMF+98]. The visualization consists of a temporally-ordered series of pictures or *cels*, each detailing information about a corresponding point in the execution of the system being analyzed, and a summary of the execution to that point. Using the visualization tool, a developer can navigate across the trace, either one event at a time or as an animation, seeing how objects mapped to the entities in the architectural view call each other, are allocated and are deallocated.

Figure 1 shows a screen snapshot of the tool in action at a point about halfway through the execution of an implementation of a hierarchical agglomerative reverse engineering algorithm. This algorithm automatically clusters entities, such as procedures in a C program, based on a similarity function to determine a subsystem organization for the system. In the visualization, the classes implementing the algorithm are mapped to four architectural entities (the grey squares): Clustering, representing the class performing the clustering analysis; SimFunc, representing the class containing methods for computing the similarity function; `ModulesAndSuch` representing the classes denoting the modules whose similarity is being compared; and `Rest`, representing all other classes involved in the algorithm. The solid arrows show the calls between objects mapped to the separate architectural entities. The dashed arrow represents the current call stack. Object allocation and deallocation information is shown in two ways: the total number of objects allocated and deallocated per architectural entity are shown within each grey square; and, the histograms provide a view on the memory usage over time by an architectural entity. This particular visualization was used in a case study that discovered the source of an execution problem in the implementation of the reverse engineering algorithm; further details about the case study are available elsewhere [WMF+98].

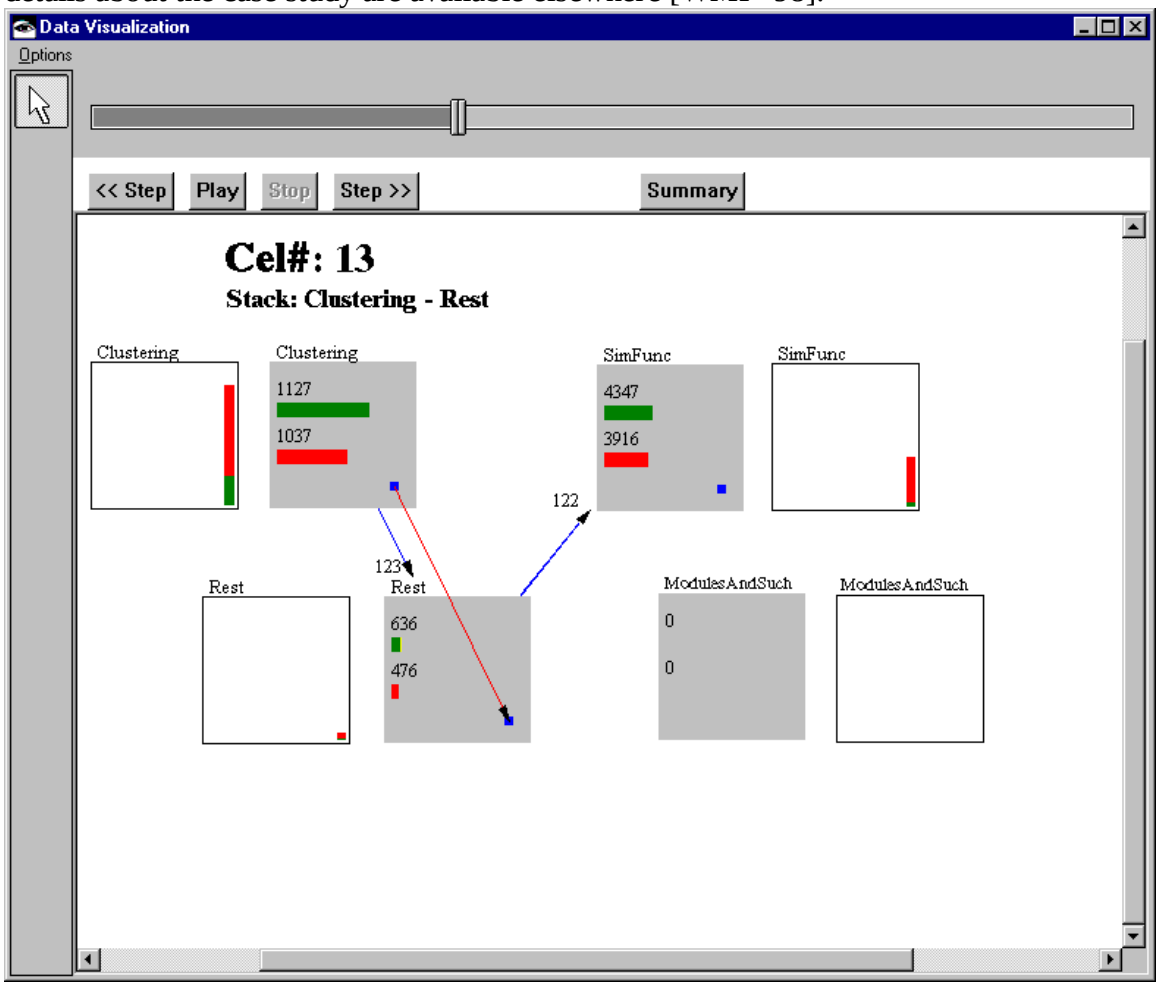

Figure 1 Architectural Visualizer

The software developer using the visualization tool is in control of the particular architectural view used. The following specification maps the lower-level source code entities to the architectural entities chosen for the reverse engineering system.

```
category Clustering class "ArchClusteringAnalysis"
category ModulesAndSuch class "Arch(Procedure|Symbol)
category SimFunc class "ArchSimFunc"
category Rest class "Schwanke.*"
```

This specification uses regular expressions to identify classes in the system to map to a particular architectural entity. For instance, the first line states that any class named `ArchClusteringAnalysis` is to be associated with the architectural entity named `Clustering`.

Visualizing Long-lived Systems

Our visualization tool enables a developer to look at large portions of the dynamics of a system at one time. However, using our tool, it is still difficult for a developer to consider large portions of the execution of a system as the traces of the system grow large. We have added two additional features to our tool to try to address this problem: filtering and sampling.

Filtering

To enable us to experiment with different approaches, our tool currently works from traces collected using the IBM Jinsight system. The Jinsight traces are converted into our format. As part of the conversion process, we have implemented a filtering system which allows a developer to specify which classes should be retained in the trace and which should be discarded. A developer specifies the classes to be included and excluded based on a regular expression match of the Java fully-qualified names.

The filtering of classes is intended to help a developer eliminate sources of noise in the visualization. For instance, for the purposes of some tasks, events related to the Java io package may be considered noise.

Such filtering is common in visualization. We plan to experiment with the filtering to understand its impact on scale in the context of architectural visualization.

Sampling

Filtering eliminates information about the execution from the visualization. Often, in the early stages of performing a task, such as a performance tuning task, a developer may not be able to determine which portions of the execution it is reasonable to filter. Yet, the developer may still need to deal with scale. For instance, in the initial stages of a performance tuning task, the developer may be trying to isolate a portion of the execution of the system to consider in detail for attacking the performance problem of interest. We believe that in such cases, it may be helpful to visualize information sampled from the execution of a system.

To investigate the use of sampling, we have implemented both time-based and event-based sampling for our architectural visualizer. In contrast to filtering, sampling requires support both in forming the execution information presented to the visualizer, and in the visualization of the information. We have chosen to visualize sampled information in a similar form to information that is traced. As a result, developers can seamlessly move between sampled and traced information over the course of an execution of the system. Early experimentation on medium sized systems with the sampling approach has been positive: We have been able to pinpoint locations of performance problems without using an overwhelming amount of execution information. We plan to continue these investigations in the context of larger, industrial systems.

Visualizing Multi-Threaded Systems

In order to use the architectural visualization tool with nontrivial systems, we have had to provide capabilities accounting for multiple Java threads, while keeping the information load on users at a reasonable level. When stepping through the execution trace, a user requires information regarding which threads are currently active, and state information for each, such as the current call stack . This was achieved with minimal addition of visible information, by keeping everything in a single display, and color-coding threads to keep them distinct. Early experimentation has shown the usefulness of this type of visualization in understanding thread interactions. Also, the capability to focus on a particular thread or group of threads has been added to allow more detailed analysis of the target system's trace at given point, or in summary. A typical usage pattern might include examining the trace with all threads displayed, determining which contain information of interest, then displaying only these threads in later visualization. Preliminary case studies have shown that this approach to multi-threading allows for further gains in effective information management over large traces.

Acknowledgments

This work was supported by the Canadian Consortium for Software Engineering Research (CSER) in conjunction with OTI, Inc.

References

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering,* Vol. 2, World Scientific Publishing Company, Singapore, pp. 1-39, 1993

[WMF+98] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright and Darin Swanson. Visualizing Dynamic Software System Information through High-level Models. In Proc. Of OOPSLA '98, p. 271-283, 1998.