7-2001

A Framework for Visualizing the Behavior of Component-Based Software Systems

Matthew O. Ward
Worcester Polytechnic Institute, matt@cs.wpi.edu

George T. Heineman
Worcester Polytechnic Institute, heineman@cs.wpi.edu

A Framework for Visualizing the Behavior of Component-Based Software Systems

Matt Ward and George Heineman[1]
Computer Science Department
WPI
Worcester, MA 01609
WPI-CS-TR-01-19
{matt,heineman}@cs.wpi.edu

1. Introduction

It is commonly accepted that software systems have grown too large to statically verify and analyze. This is true even when the software is decomposed into well-defined software components. Until the software engineering community develops more powerful analysis techniques, there is a need for developers to assess the run-time behavior of complex software systems. We propose a framework for visualizing the execution of component-based software systems to answer such questions as:

- What interface elements of component X are accessed when the system performs functionality Y?
- What is the order of method calls (over time) for a particular object-oriented class C?
- Is the function `void f (char *s)` ever called with a `NULL` pointer?
- What is the temporal relationship between the method invocations on components X and Y?

Visualization provides the means for developers to ask questions (such as these) that they could not have known in advance, but only formulated as they viewed a visual representation of the system in execution.

An inherent difficulty in general software visualization, whether static or dynamic, is the lack of an agreed-upon set of "ideal" metrics for component-based software systems. In contrast, network quality of service (QoS) characterizes deviations from ideal network behavior (e.g., infinite capacity, zero delay and corruption) as a multi-dimensional distortion space, which becomes visible only when delivered to the end-points. QoS visualization and control thus becomes a framework to discuss which of several forms of distortion is least disruptive to the application and what the resource cost of controlling these distortions is to the network [9]. It is possible to build very large networking systems whose overall behavior cannot be formally analyzed, but whose day-to-day operation can be practically visualized and managed. In the software engineering community, we face a similar situation because we must manage the composition and integration of component-based software systems that we are unable to formally analyze within acceptable cost and time bounds.

The DASADA (Dynamic Assembly for Systems Adaptability, Dependability, and Assurance) research program funded by the Department of Defense is divided into two interconnected infrastructures [10]. A probe infrastructure is responsible for extracting meaningful events from the execution of a working component-based software system. A gauge infrastructure processes and delivers these events to specific gauges programmed to listen for events. A primary aim of DASADA is to develop a partnership between gauges and probes to evaluate a software system. However, gauges are carefully designed to observe specific situations and report results back to system administrators. In this paper, we present our proposed approach to visualizing the events of the software system. In a way, the visualization provides a standard means for observing the behavior.

There are several important benefits of the monitoring infrastructure. First, the probe infrastructure decouples the model (of the system) from the realization (implementation) of the system being monitored. Second, the approach is language independent. Third, it allows for exploration and customization, as different visualizations are activated in real or near-real time. One of the authors of this paper has implemented a probe run-time infrastructure as part of the DASADA program. To date, however, this infrastructure has only been plugged into simplistic pre-programmed gauges. This paper sketches the principles behind a new visualization framework that we will build on top of the probe infrastructure. To

[1] Heineman is supported in part by Air Force Cooperative Agreement No. F30602-00-2-0611.

date, we have focused our effort on defining the models we will use as a foundation for visualization. The pictures in this position paper show the type of visualizations we plan to implement.

2. Models

We have developed a set of interrelated models to form the basis of the visualizations. The *structural model* captures the hierarchical structure of any software system. This model is language-independent and captures the way software is decomposed into finer units of granularity. A *behavior model* captures the dynamic behavior of a software system as it executes. We incorporate the results of our DASADA project for the continual validation of the dynamic functional and extra-functional properties of component-based systems [4]. To continually validate a software system, probes are injected to extract information as the system executes. These probes emit events that identify the individual software elements (i.e., functions, methods, or components) that execute and their attributes. We correlate the dynamic events generated from the probes with the structural model. The final *visualization model* enables users to explore large amounts of system information to debug the system, validate functional properties, or monitor on-going performance.

2.1. Structural Model

The basic structural model is stratified into five hierarchical levels: *system, layer, components, entity, sub-entities.* A system is composed of multiple layers. For our purposes, we identify four layers: *Graphical User Interface, Business Logic, Wrapping and Legacy Services,* and *Data and Operating System Services* [1]; the number (and type) of layer is customizable. Each layer is further decomposed into components that have been assigned to that layer. To ensure that the model can be applied to multiple programming languages, we decompose components into *entities*, which can be interpreted according to the particular component implementation. For example, an entity could be an object, a module, a java package, or a function. *Sub-entities* enable further decomposition, for example, into attributes, or private functions. Our model is extensible and through scaling, multiple levels can be added to the model (e.g., sub-layers, decomposed components, or even sub-systems).

2.2. Behavior Model

We attach "probes" to each component's interface, introducing before and after callbacks wrapped around each entry point, in the style of our active interfaces paradigm [2] - but now extended beyond procedure call to other connector types, including events, data accesses, linkages, streams, arbitrators, adaptors, and distributors (as in the taxonomy of Mehta *et al*. [3]). We are thus able to extract detailed semantic information about the software system as it executes and correlate this information with the structure of the component-based software system.

2.3. Visualization Model

Given a hierarchical structural model, we developed structures upon which to base our visualizations. We are interested in exploring both static and dynamic attributes of a software system, and have developed four interconnected views of the software model:

Node view

As the layers, components, and entities define a hierarchical relationship, we use a radial visualization (similar to [6]), with the radius corresponding to the depth of the hierarchy, to visualize the building blocks of the software system. Color is used to associate sibling nodes with each other and their parents. A *node* is any wedge within a visual level. Direct manipulation enables selective drill-down (to reach a more detailed level) and roll-up (to aggregate child nodes into their parent node). Sets of nodes can be selected as slices of interest, and may include nodes from multiple levels.

Link view

Links represent communication or flow between nodes. We generate visualizations by exploding the node view outward from the center to help preserve the nodal relationships while making room for explicit links between nodes. Links can be used to convey many types of relations and dynamic attributes, such as call graphs and class inheritance. Because of the large number of links possible, it is essential that users be able to select links of interest, which results in other links being hidden. We can specify a "chain" of links by selecting a node of interest and displaying the paths leading into or out of the node. The user can constrain

a chain length in a number of ways, including a fixed link count and an ending node. A set of chains (which we call a "fence") can be created and explored in the same manner as a slice of nodes. Links can be colored either according to a single attribute (e.g., communication volume or chain ID) or using the two colors associated with the nodes sharing the link.

Time view

Each node in a selected slice – or link in a selected chain or fence – can generate events, and these events can either affect the node/link views or add an entry to the time view, which is simply a horizontal line with vertical colored lines to mark the events (similar to PV [7]). Each line is colored to correspond to its node or link to enable easy matching of events to their location in the node or link structures. Aggregation events, which can be based on time, accumulated values/counts, or other node/link characteristic, can be used to reduce the amount of clutter in the time view. The user can filter the time view, either by type of event of by bounding the time period of interest, and replay the events to watch their effect on the other visualizations.

Attribute view

Each node or link may have a large number of attributes that could be the focus of exploration and analysis. We might be interested in the values of these attributes at a single point in time, or how their values change over time. At present we restrict ourselves to the display of numeric quantities, and we use a suite of multivariate visualization techniques (from XmdvTool [8], developed at WPI) to convey the values for a selected node or link. Techniques such as data glyphs are useful for showing the values of an attribute set at a single point in time, while scatterplot matrices and parallel coordinates are effective for showing behavior over time. Because different nodes or links may have different attributes, we cannot, at present, show the attributes of multiple nodes/links in a single visualization. Instead we can juxtapose these visualizations to help identify correlations or anomalies.

3. User Interaction with the Visualization

We are exploring and integrating several methods to improve the effectiveness of visual exploration of the behavior of a large-scale software systems, including:

Linking

We use color and selective highlighting as the main devices for conveying associations between views. The user can activate multiple views that are automatically linked to each other. A selection in one view is automatically applied to all other open views.

Brushing

Users click on or brush over parts of the display as mechanisms for selection. This in turn can result in an action such as highlighting, drill-down, or filtering.

Multiresolution viewing

Each visualization can be viewed at a number of resolutions, using drill-down/roll-up interactions to specify where more or less detail is needed and various forms of aggregation to represent values within groups of nodes, links, or events.

Filtering

As the user builds a mental model of the behavior of different aspects of the system, different parts become more or less interesting. By filtering uninteresting aspects, as selected by the user, the available screen space is used more effectively. Because visualizations can persist over time, so must all filters selected by the user.

Distorting views

Another mechanism commonly used to manage dense information displays is to redistribute screen space for the different components of the display. Thus we can use lens effects and other non-linear distortions of the pixels, structures, or data to examine parts of the display at higher resolution [11] while preserving the overall context. Distortion is mostly independent from the other user interactions.

4. Visualization Sketch

To give a sense of the types of visualizations we envision, consider Figures 1 and 2. These visualizations are correlated by color and selection (not shown here). Figure 1 shows the radial display of the structural model. The innermost circle represents the system. The second circle is divided into four layers. Layers L_1 and L_2 could be the GUI and processing layers of a client application, while Layers L_3 and L_4 could represent two layers within a server application. Layers $L_2 - L_4$ are exploded out one more layer of detail to show the components contained within the respective layers. Finally, several components are further exploded to show the entities within the component, for example, individual method calls, attributes, or low-level events. Figure 2 shows a temporal radial depiction of the behavior of a component from time T_0 through time T_1. Each black dot represents an interface access, method invocation, or attribute access.

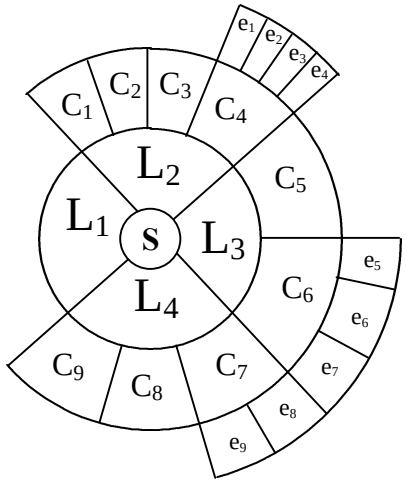

Figure 1: Visualization of Structure

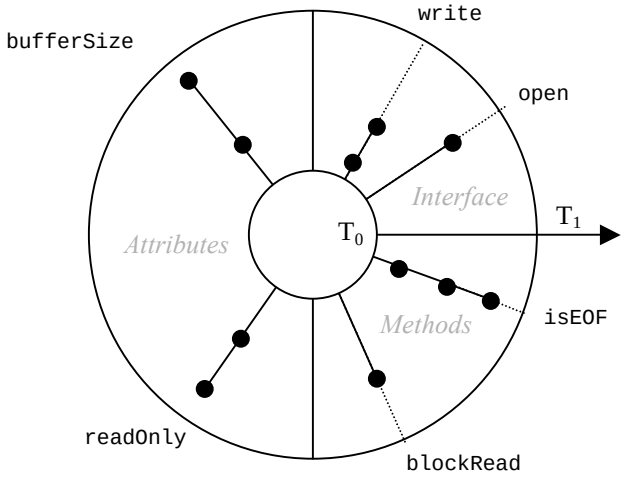

Figure 2: Visualization of Component

Acknowledgements

We would like to acknowledge the efforts of all groups funded through DASADA. For the full list, visit http://www.rl.af.mil/tech/programs/dasada

References

1. S. Latchem, "The Design of Component Infrastructures", in George T. Heineman and William T. Councill, eds., *Component-Based Software Engineering: Putting the Pieces Together,* Addison-Wesley, Longman, 2001.
2. G. T. Heineman. A Model for Designing Adaptable Software Components. 22nd Annual International Computer Software and Applications Conference, Aug., 1998.
3. N. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. 22nd International Conference on Software Engineering, June, 2000, pp. 178-187.
4. P. Gill, Probing for a Continual Validation Prototype, M.S. Thesis, Aug., 2001.
5. *Software Visualization: programming as multimedia experience*, J. Stasko, J. Domingue, M. Brown, B. Price, Eds., MIT Press, 1998.
6. J. Stasko and E. Zhang, "Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations", Proceedings, Information Visualization 2000, pp. 57-65.
7. D. Kimelman, B. Rosenburg, and T. Roth, "Strata-Various: multi-layer visualization of dynamics in software system behavior", Proceedings, Visualization 1994, pp. 172-178.
8. M. Ward, "XmdvTool: integrating multiple methods for visualizing multivariate data", Proceedings, Visualization 1994, pp. 326-333.
9. DARPA ITO Sponsored Research: Massachusetts Institute of Technology (MIT). 1998 Project Summary: Robust Multi-Scalable Networks, http://www.darpa.mil/ito/psum1998/F443-0.html.
10. DASADA, Dynamic Assembly for Systems Adaptability, Dependability, and Assurance, http://www.rl.af.mil/tech/programs/dasada.
11. Y. Leung and M. Apperley, "A review and taxonomy of distortion-oriented presentation techniques", ACM Transactions on Computer-Human Interaction, Vol. 1, No. 2, 1994, pp. 126-160.