SoftArchViz: An Automated Approach to Visualizing Software Architectures

Naveen Bali*
Performance Engineering Group,
Network Appliance, Inc.
Research Triangle Park, NC, USA

Amit P. Sawant†
Department of Computer Science,
North Carolina State University
Raleigh, NC, USA

Abstract

This paper presents a programmatic mechanism for visualizing large software architectures using Multi-Dimensional Scaling. We begin with a short survey of standard representations of software architecture used in the industry and then describe how our software visualization technique was applied to Network Appliance Data ONTAP®7G (storage server operating system). We show how our visualization tool known as SoftArchViz can be used to study different architectural views that represent different attributes of the software components. Finally we discuss potential future work to enhance the capabilities of SoftArchViz and the application of these enhancements to study new aspects of software architecture.

Keywords: software architecture, architecture visualization, software visualization, information visualization, perception

1 Introduction

The Network Appliance storage server operating system known as Data ONTAP 7G is a very large software system with more than 10,000 files and close to 1 million lines of code. It is almost impossible for a single developer or engineer working with this system to comprehend all its components and their interactions. A number of ways to look at the components of Data ONTAP 7G exist, but they all oversimplify the world and leave out critical details. A software architecture that accurately and completely represents Data ONTAP 7G would help software engineers, but we found that such an architecture does not exist at Network Appliance. In fact, this situation is not unusual in the industry.

This paper is the result of our attempt to define a software architecture for Data ONTAP 7G that neatly encapsulates the way the system is partitioned and enables us to think about components and the interactions between them. Our work utilizes existing categorizations of Data ONTAP 7G as embedded directly in the code and uses visualization technologies to graphically identify and browse the components of Data ONTAP 7G. Although our work has dealt exclusively with Data ONTAP 7G, the techniques described in this paper can be easily applied to other large software systems such as operating systems, middleware and application software.

The remainder of this paper proceeds as follows. In Section 2 we survey related work. In Section 3 and Section 4 we discuss software architectures and multi-dimensional scaling visualization technique. Section 5 describes the building of an architecture programmatically for Data ONTAP 7G. Section 6 provides details on visualizing software architecture. In Section 7, we present some user feedback. Finally, Section 8 and Section 9 discuss conclusions and future work.

*e-mail: naveen.bali@netapp.com
†e-mail: amit.sawant@ncsu.edu

2 Related Work

Knodel et al. [15] propose that software architecture visualizations should be thoroughly assessed by software architects in an industrial environment. They successfully applied the approach of software development, visualization, and validation to their software architecture visualization tool. Byelas et al. [6] use a technique that minimizes clutter to visualize areas of interest in system architecture diagrams and thereby gain more understanding about complex software systems. Auer et al. [3] applied Multi-Dimensional Scaling techniques to visualize high-dimensional software portfolio information gathered from different sources. They proposed a simple method to define raw software metrics data that can be analyzed and processed using Multi-Dimensional Scaling methods. They also conducted a validation study to demonstrate the usability of their proposed approach for software decision support. Eick et al. [8] created and implemented a number of concepts for visualizing the change in software data. These concepts can be combined to form perspectives to understand software change by exploring software change data and thereby managing the software development cycle. GASE (Graphical Analyzer for Software Evolution) [12] is another tool to visualize software structural change. This tool uses color to represent new, common and deleted parts of a software system, and thereby enables the software developer to gain insight about the structural change that might otherwise be difficult to detect.

Mancoridis et al. [16] developed techniques to facilitate the automatic recovery of the modular structure of a software system from its source code. They conducted validation experiments by asking the actual system designers for their feedback. Their experimentation showed good results. The RECONSTRUCTOR project focuses on interactive visualization of software systems and is concerned with the representation of architecture information in different forms [13].

We took inspiration from the above mentioned related work while developing *SoftArchViz*.

3 Software Architectures

There are many well-known styles of software architecture that designers and developers regularly use [4]. Terms that describe style, such as “client-server”, “object-oriented”, “distributed”, “multiprocessor”, “layered”, “message passing” and “preemptive” are widely used and software designers and developers have an intuitive understanding of their meaning. Garlan et al. [10, 11] describe software architecture as a collection of components together with a description of the interactions between these components - the connectors.

The benefits of having a well-defined, complete and accurate software architecture that can be easily visualized are manifold: software engineers can easily comprehend the system; developers can anticipate and analyze the impact of their source code modifications on the rest of the system; software architects can visually

inspect and validate proposed alterations to the architecture. Unfortunately, most software systems do not have well-defined architectures and those that do are almost always incomplete or inaccurate or both. This situation, we believe, is due to the lack of availability of tools for easily creating and updating software architectures.

There are programmatic ways of creating or encoding architectures, known as Architecture Description Languages (ADLs) [9, 1]. A major drawback with using ADLs is that there are many different ADLs available but each one is specific to a particular software domain, such as network protocols. There is no general-purpose ADL available that we are aware of for encoding all types of software. Even if such an ADL exists, it is nevertheless cumbersome to expect all developers in an organization to learn yet another new language.

Our visualization mechanism provides a software architectural representation that is both complete and accurate. Moreover, since we use raw source code to generate the software architecture, the architecture can be updated frequently to capture changes as they happen. With ADLs an architecture must be first envisioned, then specified in the chosen ADL, and finally implemented as code.

One of the limitations of ADLs is that tracking deviations in the implementation from the architectural representation is not easy, and the problem becomes worse as the implementation evolves over time. In contrast, we represent the architecture exactly as it is implemented and we can easily track it as the implementation evolves.

4 Multi-Dimensional Scaling Visualization Technique (MDS)

Multi-dimensional scaling (MDS) technique [18, 7, 19, 5] provides visual representation of the pattern of proximities among a set of objects. The relationship between input proximities and distances among points on the map can be:

- Positive: The smaller the input proximity, the closer the distance between points, and vice versa (e.g., visualization of cities).
- Negative: The smaller the input proximity, the farther the distance between points, and vice versa (e.g., visualization of software architecture).

MDS finds a set of vectors in p-dimensional space such that the matrix of Euclidean distances among them correspond as closely as possible to some function of the input matrix according to a criterion function called stress. The simplified algorithm is as follows:

1. Assign points to arbitrary coordinates in p-dimensional space.
2. Compute Euclidean distances among all pairs of points, to form the *Dist*' matrix.
3. Compare the *Dist*' matrix with the input *Dist* matrix by evaluating the stress function. The smaller the value, the greater the correspondence between the two.
4. Adjust coordinates of each point in the direction that best reduces the stress value.
5. Repeat steps 2 through 4 until stress won't get any lower.

4.1 Visualization of Cities

We started to tackle the problem of visualizing software architecture by first visualizing the distances between cities. This is a classic example and has been visualized using MDS techniques. We selected nine cities: Seattle, Minneapolis, Boston, Reno, Denver, Raleigh, Los Angeles, Austin, and Miami. The distances between these cities are as shown in Figure 1 taken from http://www.thepalmbeachtimes.com/TravelNavigator.

Figure 2 shows three different views generated with three different random starting positions for the cities while running the MDS visualization software. One important thing to note here is that any of these views can be transformed into the other by changing the perspective (e.g., translation and rotation). Figure 2c represents a topographically accurate view of the nine cities.

5 Building an Architecture Programmatically for Data ONTAP 7G

We found the only way to realistically extract a software architectural view of Network Appliance Data ONTAP 7G is to examine and visualize the source code. Kazman et al. [14] describe a workbench called Dali, which can extract, manipulate and interpret software architectural information using raw source code as input. We have borrowed some of the methodology of Dali in our work. The following subsections describe the steps we took in arriving at an architectural representation.

5.1 Identifying Software Components

All architectural styles have components and connectors, as described earlier. In order to generate a more complete representation of Data ONTAP 7G, it is necessary to first decide what components to use and how to connect them. BURT is a bug-reporting tool that is widely used at Network Appliance. All bugs have a type and a subtype. Since every line of code in Data ONTAP 7G has a BURT associated with it and since every BURT has a subtype, it follows that the entire code base can be partitioned into BURT's software subtypes. After eliminating obsolete and redundant software subtypes in BURT and merging overlapping ones, we were left with a set of 38 software components. The connectors in our software architecture are direct procedure calls and accesses to global variables.

5.2 Building Cross-Reference Tables for Software Components

The method of code partitioning using subtypes in BURT lends itself naturally to code isolation. Usually, there is a one-to-one mapping between software subtypes in BURT and engineering development teams. Most engineering development teams work on code that resides in a small set of subdirectories in the code tree. So it was easy to map these software components to code. An analysis of these software components involved the following steps:

1. Create a list of symbols (functions and global variables) exported by each software component.
2. Create a symbol cross-reference table that lists the number of references from each component to every other component.

We used a commercially available tool called Understand C++ [2] to extract the functions, global variables and macros defined within each file in the source tree and obtained cross-references for these symbols. We then used homegrown Perl scripts to extract the symbols and symbol cross-references in HTML format and created a new mapping of symbols to software components. We finally distilled global cross-reference symbol tables.

Figure 4 shows a subset of the global functions cross-reference table with the software component *platform* highlighted, and Figure 6 shows a subset of the global variables cross-reference table with the software component *kernel* highlighted. In these figures,

	Seattle	Minneapolis	Boston	Reno	Denver	Raleigh	LA	Austin	Miami
Seattle	0	1377	2456	557	1003	2318	942	1739	2685
Minneapolis	1377	0	1106	1384	683	966	1514	1023	1483
Boston	2456	1106	0	2485	1739	603	2572	1670	1243
Reno	557	1384	2485	0	781	2206	386	1380	2427
Denver	1003	683	1739	781	0	1425	837	759	1692
Raleigh	2318	966	603	2206	1425	0	2205	1143	693
LA	942	1514	2572	386	837	2205	0	1219	2307
Austin	1739	1023	1670	1380	759	1143	1219	0	1091
Miami	2685	1483	1243	2427	1692	693	2307	1091	0

Figure 1: Distance table between the nine cities

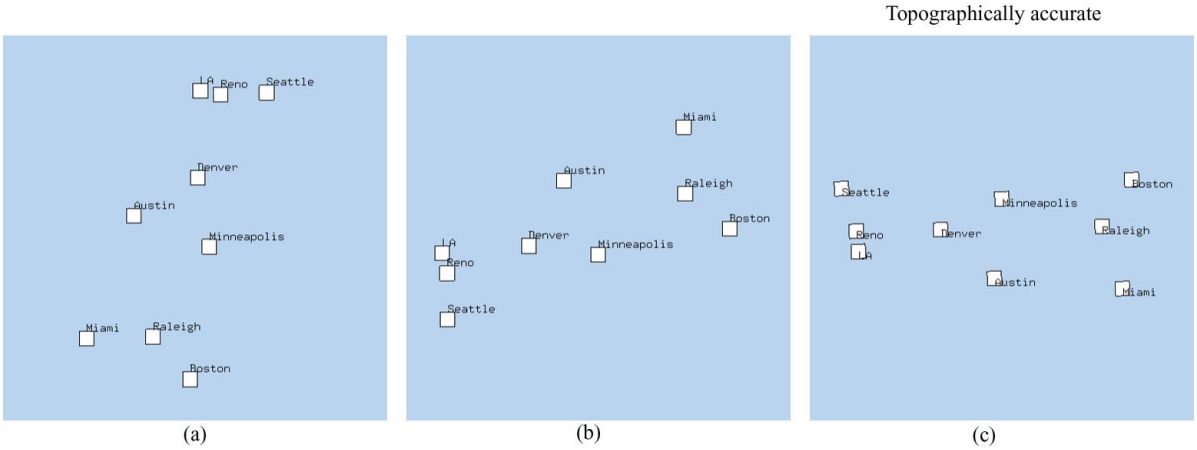

Figure 2: Three different views of the visualization of nine cities

the number in any given component tuple (*row*, *column*) represents the number of references from *column* into *row*. For example, in Figure 4 101 for the component tuple (*platform*, *admin*) represents the number of global function references from *admin* into *platform*.

6 Visualizing Software Architecture

We used the principle discussed in Section 4 to visualize software architecture. When we design visualization, properties of the data and the visual features used to represent its data elements must be carefully controlled to produce an effective result [17]. Important characteristics that must be considered include:

1. Dimensionality (number of attributes in the dataset)
2. Number of elements
3. Visual-feature salience (strengths and limitations that make it suitable for certain types of data attributes and analysis tasks)
4. Visual interference (different visual features can interact with one another, producing visual interference; this must be controlled or eliminated to guarantee effective exploration and analysis)

Using the knowledge of the above perceptual guidelines, we choose visual features that are highly salient, both in isolation and in combination. We map features to individual data attributes in ways that draw a viewer's focus of attention to important areas in a visualization.

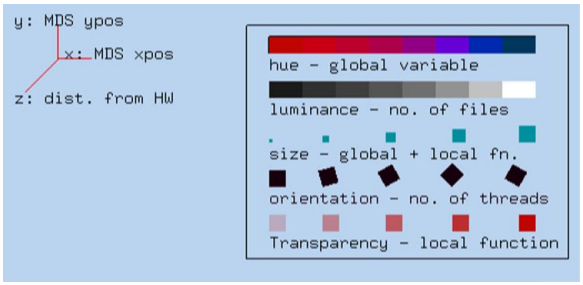

Figure 3: Default data-feature mappings. Each glyph represents a software module, *number of global variables* mapped to hue, *number of files* mapped to luminance, *number of global functions + number of local functions* mapped to size, *number of threads* mapped to orientation, *number of local functions* mapped to transparency, *distance from hardware* mapped to height

6.1 Data-Feature Mappings

Our datasets, global functions, and global variables cross-reference tables consist of values representing the number of references between software components (the higher the value, the more connected the pair of components are, and vice versa) along with six attributes: number of files, number of threads, number of global variables, number of global functions, number of local functions, and distance from hardware layer.

We use simple 3D geometric objects called "glyphs" that support variation of spatial position (*x position*, *y position*), color (*hue*, *luminance*), and texture (*height*, *size*, *orientation*, *line thickness*, and

transparency) properties to represent the attribute values of the software component. The most important attributes should be mapped to the most salient features, and secondary data should never be visualized in a way that would lead to visual interference. The order of importance for visual features is luminance, hue, and then various texture properties. In our visualizations, we selected initial default data-feature mappings, as shown in Figure 3, in consultation with domain experts, such that:

- Spatial position (x position and y position) and line thickness represent the *number of global function references* between connecting software components
- *Number of global variables* mapped to hue (red for low values to blue for high values)
- *Number of files* mapped to luminance (brighter for higher values)
- *Number of global functions + number of local functions* mapped to size of 3D glyphs (larger for higher values)
- *Number of threads* mapped to orientation (more tilted for higher values)
- *Number of local functions* mapped to transparency (more transparent for lower values and less transparent for higher values)
- *Distance from hardware* mapped to height (taller for higher values)

6.2 Visualization of Global Functions

Figure 5 shows a visualization of the global functions cross-reference table (Figure 4), along with five other attributes: number of files, number of threads, number of global variables, number of global functions, and number of local functions. The data-feature mappings are as shown in the legend. Spatial position and line thickness represent the number of global function references between connecting software components.

The visualization in Figure 5a shows connections between all the software components with a non-zero value in the global functions cross-reference table. Figure 5b shows the outgoing global function calls from the software component *platform* (represented by the highlighted column). Similarly the incoming global function calls to the software component *platform* can be shown.

6.2.1 Interpretation

1. Figure 5a shows that most components talk to just a few other components, which suggests that the architecture might lend itself to a layered representation. But the architecture is not strictly layered, because most components communicate with more than two other components.
2. Figure 5b shows that modules *network*, *storage*, *kernel*, *nvram*, *target* and *ha* are very closely related to *platform* because they are placed close to one another and are connected by lines.
3. The same information can be gleaned from the global functions cross-reference table by closely examining the numbers in the highlighted row and column in Figure 4, but it is hard to mentally process and comprehend a large matrix with 38×38 entries.
4. Also, it is easy to miss certain subtle details: for example, the components *nvram* and *target* are closely related to *platform* because although the number of references from these modules into *platform* is relatively small, they are nonetheless large as a fraction of all references from these modules.
5. *SoftArchViz* understands these details and illuminates them without taxing the user and allows the user to directly interact with the data.

6.3 Visualization of Global Variables

Figure 7 shows a visualization of the global variables cross-reference table (Figure 6) along with five other attributes: number of files, number of threads, number of global variables, number of global functions, and number of local functions. The data-feature mappings are as shown in the legend. Spatial position and line thickness represent the number of global variable references between connecting software components.

The visualization in Figure 7a shows the outgoing global variable references from the software component *kernel* (represented by the highlighted column) and Figure 7b shows the incoming global variable references into the software component *kernel*.

6.3.1 Interpretation

1. The *kernel* module is central to the architecture because it provides common services to all the other modules. This is clearly illustrated in Figure 7 because there are many more incoming connections than outgoing ones.
2. Similar observation can be made from the global functions cross-reference table for the *kernel* module.
3. The same information can be gleaned from the global variables cross-reference table by closely examining the numbers in the highlighted row and column in Figure 6.

7 User Feedback

We presented our tool (*SoftArchViz*) to a few software engineers at Network Appliance, and observed their interaction with it and gathered their feedback after they used the tool. Below are some of their reactions:

1. “I can figure out exactly which other engineering teams I need to consult with regarding my new feature”
2. “I never realized that the component that I work with has interactions with so many other components”
3. “We could visualize a dynamic workload using this approach by analyzing the runtime profile data”
4. “This tool helps me see the big picture and where my module fits in”
5. “Very useful and powerful”

This feedback leads us to believe that we have taken a positive step in visualizing software architectures.

	admin	autosupport	cifs	cmds	coredump	counters	asis	dump	filerview	ha	platform	raid	# Files	# sk Threads	# Global Variables	# Global Functions	# Local Functions	Hardware distance
admin	10151	154	862	1638	0	40	12	42	0	193	370	108	491	500	812	2187	2670	4
autosupport	16	294	0	11	2	0	0	0	0	11	120	23	21	2	386	93	89	4
cifs	206	2	6398	683	0	12	0	0	0	23	51	5	491	60	666	3309	1973	3
cmds	333	0	220	5241	0	5	2	0	0	57	216	127	329	26	516	1463	3608	4
coredump	8	0	0	2	161	0	0	0	0	2	10	7	25	2	217	108	179	2
counters	50	2	61	260	0	132	0	0	0	0	0	0	407	1	12	121	94	4
asis	10	0	0	87	0	1	239	0	0	0	0	0	26	4	48	92	88	4
dump	0	0	0	1	0	0	0	336	0	0	0	0	21	1	8	115	175	4
filerview	0	0	0	0	0	0	0	0	0	0	0	0	289	0	0	0	0	4
ha	169	18	20	149	39	1	0	0	0	7650	440	95	696	94	912	2774	2425	2
platform	101	13	3	318	52	12	7	0	0	538	20999	171	1281	60	2113	5669	7500	1
raid	117	0	3	656	37	18	0	0	0	88	45	12491	222	36	1288	2574	1895	2
....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....

Figure 4: Subset of the Global Functions cross-reference table with column and row *platform* highlighted

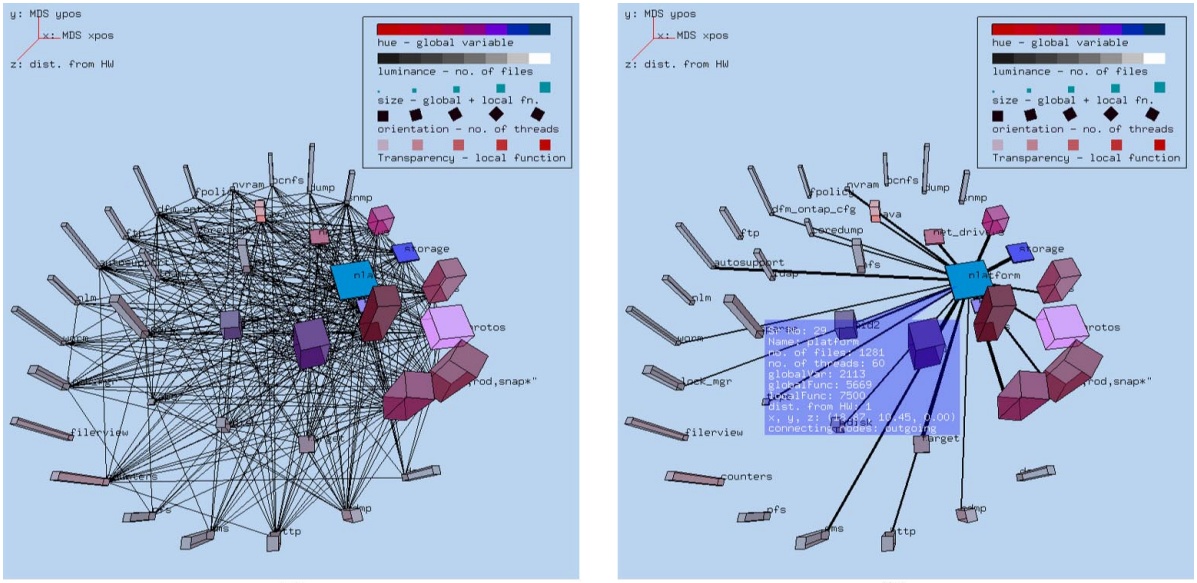

Figure 5: Visualization of Global Functions cross-reference table: (a) Mesh and (b) outgoing global function calls from the module *platform*

8 Conclusions

Kazman et al. [14] pose the following question: "Is software architecture a mass hallucination that we, as developers, gladly and glibly ascribe to?" We believe that our work is a small step toward shattering this mass hallucination. We hope that our work will encourage software developers to think about how their day-to-day work affects the overall software architecture of the system that they are working with. We use rules of human perception to build displays that harness the strengths and avoid the limitations of low-level human vision. Individual software components are presented using graphical "glyphs" that vary their spatial position, color, and texture properties to encode the component's attribute values. The result is a display that can be used by viewers to rapidly and accurately analyze, explore, compare, and discover within the software architecture.

9 Future Work

We plan to fully automate the process of generating the various cross-reference tables used by *SoftArchViz* by running a tool such as Understand C++ in conjunction with our home-grown scripts to analyze the entire source code at some regular time interval (nightly or weekly or monthly). We plan to enhance *SoftArchViz* to enable it to represent differences in the architecture at two or more points in time so that a user can quickly identify all the modules that changed and the extent of those changes.

We would also like to visualize the architecture of individual components (and possibly subcomponents) within Data ONTAP 7G by using the same algorithm that we used to generate the high-level software architecture described in this paper. We plan to analyze performance profiles from a few benchmarks and other workloads of interest in order to visualize the dynamic interactions of the software components and subcomponents within Data ONTAP 7G at run-time. Finally, we would like to visualize the software architecture of another large system - Data ONTAP GX (Network Appliance next generation distributed storage server operating system).

10 Acknowledgments

This work is supported by Network Appliance, Inc. We would like to acknowledge Steve Rodrigues and Neil Joffe for their support and guidance. We would also like to thank Sarat Kocherlakota for his help with the MDS implementation.

References

[1] Architectural Description Languages http://www.sei.cmu.edu/str/descriptions/adl_body.html Retrieved on [05/15/2006], 2006.

	admin	autosupport	cifs	cmds	coredump	counters	asis	dump	filerview	kernel	platform	raid	....	# Files	# sk Threads	# Global Variables	# Global Functions	# Local Functions	Hardware distance
admin	1847	17	176	423	0	0	8	21	0	142	102	0	....	491	500	812	2187	2670	4
autosupport	2	775	0	4	0	0	0	0	0	3	18	0	....	21	2	386	93	89	4
cifs	2	0	619	44	0	28	0	0	0	2	5	0	....	491	60	666	3309	1973	3
cmds	108	2	11	5211	11	0	0	9	0	75	862	14	....	329	26	516	1463	3608	4
coredump	0	0	0	7	302	0	0	0	0	0	0	0	....	25	2	217	108	179	2
counters	0	0	0	5	0	63	0	0	0	0	0	0	....	407	1	12	121	94	4
asis	0	0	0	9	0	0	101	0	0	0	0	0	....	26	4	48	92	88	4
dump	0	0	0	1	0	2	0	278	0	0	0	0	....	21	1	8	115	175	4
filerview	0	0	0	0	0	0	0	0	0	0	0	0	....	289	0	0	0	0	4
kernel	775	87	1041	1589	260	98	35	95	0	5129	2122	925	....	696	94	912	2774	2425	2
platform	13	1	0	110	23	12	0	0	0	243	10821	5	....	1281	60	2113	5669	7500	1
raid	4	0	0	96	0	7	0	0	0	11	74	2891	....	222	36	1288	2574	1895	2
....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....

Figure 6: Subset of the Global Variables cross-reference table with column and row *kernel* highlighted

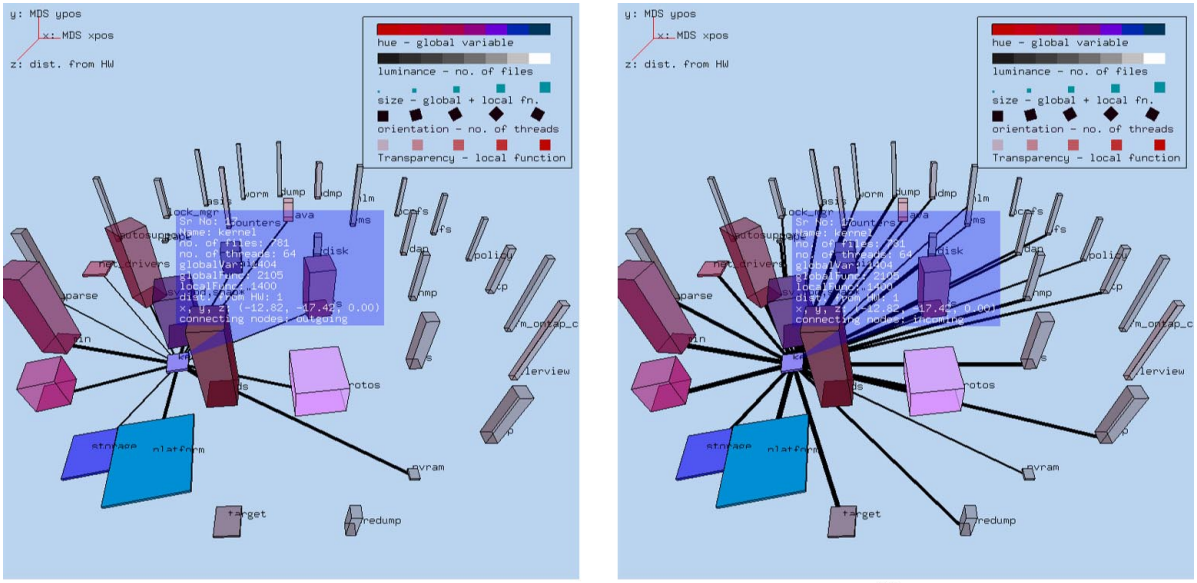

Figure 7: Visualization of Global Variables cross-reference table: (a) outgoing global variable references from the module *kernel* and (b) incoming global variable references into the module *kernel*

[2] Understand for C++ http://www.scitools.com/products/understand/cpp/product.php Retrieved on [03/10/2006], 2006.

[3] Martin Auer, Bernhard Graser, and Stefan Biffl. An approach to visualizing empirical software project portfolio data using multidimensional scaling. In *Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration, IRI - 2003, October 27-29, 2003, Las Vegas, NV, USA*, pages 504–512. IEEE Systems, Man, and Cybernetics Society, 2003.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[5] Stephen P. Borgatti. Multidimensional scaling. 1997.

[6] H. Byelas and A. Telea. Visualization of areas of interest in software architecture diagrams. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 105–114, New York, NY, USA, 2006. ACM Press.

[7] M. L. Davidson. Multidimensional scaling. 1983.

[8] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002.

[9] David Garlan, Robert T. Monroe, and David Wile. Acme: architectural description of component-based systems. pages 47–67, 2000.

[10] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[11] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[12] R. Holt and J. Y. Pak. Gase: visualizing software evolution-in-the-large. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 163, Washington, DC, USA, 1996. IEEE Computer Society.

[13] Danny Holten. Interactive software visualization within the reconstructor project (reconstructor: Reconstructing software architectures for system evaluation purposes). In *Proceedings of Visual Analytics Science and Technology Symposium*, Baltimore, MD, USA, 2006.

[14] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. *CMU Technical Report CMU/SEI-97-TR010*, 1997.

[15] Jens Knodel, Dirk Muthig, Matthias Naab, and Dirk Zeckzer. Towards empirically validated software architecture visualization. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 187–188, New York, NY, USA, 2006. ACM Press.

[16] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 45, Washington, DC, USA, 1998. IEEE Computer Society.

[17] A. P. Sawant, M. Vanninen, and C. G. Healey. Perfviz: A visualization tool for analyzing, exploring, and comparing storage controller performance data. In *SPIE-IS&T Visualization and Data Analysis*, volume 6495, pages 0701–0711, San Jose, CA, 2007.

[18] W. S. Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17:401–419, 1952.

[19] Forrest W. Young. Multidimensional scaling. *Encyclopedia of Statistical Sciences*, 5, 1985.