Visualizing and Analyzing Software Infrastructures

Adam Buchsbaum, Yih-Farn Chen, Huale Huang, Eleftherios Koutsofios, John Mocenigo, and Anne Rogers, *AT&T Labs—Research*

Michael Jankowsky, *AT&T Business Services*

Spiros Mancoridis, *Drexel University*

Large corporations typically run complex infrastructures involving hundreds or thousands of software systems. As marketplaces change, these infrastructures must be redesigned. The Enterprise Navigator system lets architects visualize and analyze the system interconnections of selected products and services.

Companies frequently need to redesign their software infrastructures in response to marketplace changes, but they must do so carefully so that the new architecture will not disrupt existing operations or increase operating costs unnecessarily. To support these goals, system architects have long recognized the need to build a repository of information about all of their company's systems and their interfaces. Using this information, architects create system interface diagrams to help them study the existing architecture. At AT&T, these diagrams used to be created and updated manually, published annually, and distributed throughout the business units.

The process of manually drawing system interface diagrams is tedious and error-prone: a simple diagram showing all the interconnections to a single system could take 30 minutes or more to draw, and the diagram often becomes obsolete before it is published. Moreover, it is not easy, through the draw-and-publish mechanism, to get a system interface diagram in real time based on an ad hoc query because the need for the diagram might not have been anticipated. For example, a manager in charge of reengineering billing operations across the company might want to generate diagrams that show the systems involved in bill calculations for more than 200 products and service offerings. Since these queries are unexpected and therefore the diagrams not published, manually producing all these diagrams could take a long time. This situation would likely delay the reengineering decision process.

We built a system called Enterprise Navigator to let users make ad hoc queries about an enterprise software architecture and then automatically generate the corresponding system interface diagram in real time on the Web. Figure 1 shows a typical diagram EN generated for a particular ad hoc query. Each node represents a system, and each link represents an interface between the two connected systems. With EN, users can

- study a system architecture's evolution over time,

0740-7459/01/$10.00 © 2001 IEEE

- find substructures embedded in complex diagrams, and
- determine which systems dominate information flows.

EN runs as a collection of stand-alone tools using a set of database visualization tools, called Ciao,[1] or as an integrated Web service. This article focuses on the latter.

Our work builds on established research in source code analysis, graph drawing, and reverse engineering. Acacia[2] and Chava[3] are examples of reverse engineering tools for analyzing C, C++, and Java programs, respectively. These systems store source code analysis results in an entity–relationship database so that users can extract software structure information through ad hoc queries without relying on customized parsers. Software engineers often use visualization tools that employ automatic graph-drawing algorithms[4,5] (see the "Graph Drawing" sidebar) to help them comprehend the results of their analyses. Many reverse engineering techniques, including techniques for software clustering[6] and dominator analysis,[7] have underpinnings based on optimization theory, statistics, and graph theory.

To date, these techniques and tools have been applied mainly to individual software systems written in a variety of programming languages. The work described here takes the next step by showing how to model, query, analyze, and visualize the entire software infrastructure of a large enterprise such as AT&T when the infrastructure information is available in a database.

Architecture

Figure 2 presents a high-level view of EN's architecture. You interact with EN by means of a Java applet (as shown in Figure 3). The applet establishes a two-way socket connection to a Java application running on a server. The Java application communicates with a database of software infrastructure specifics via a JDBC (Java Database Connectivity) connection (http://java.sun.com/products/jdbc). The applet passes your visualization requests to the server application, which formulates an SQL query to retrieve the necessary information from the database. The server application then constructs a system interface graph and opens a connection to a graph layout pro-

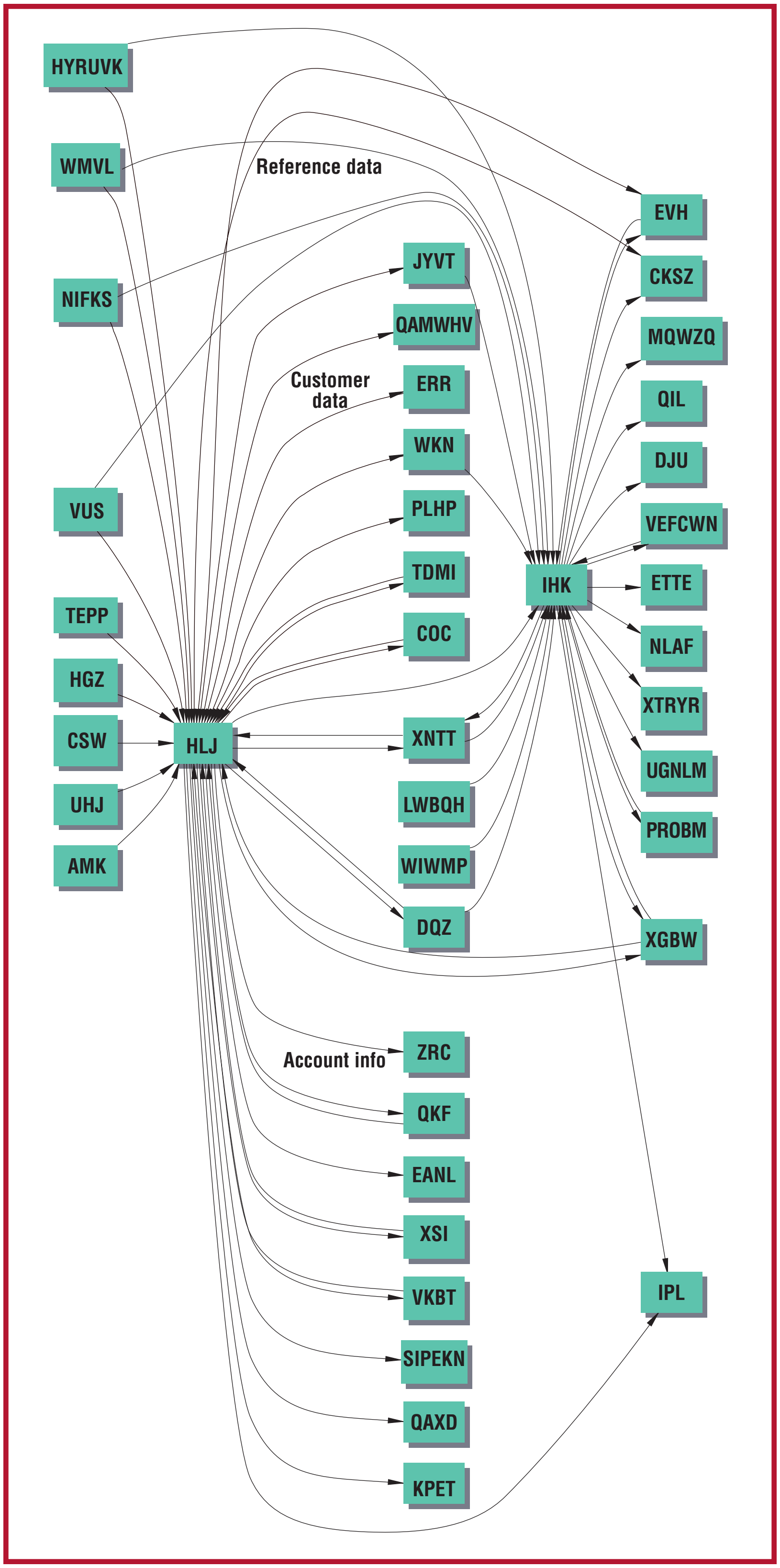


Figure 1. A typical system interface diagram generated by Enterprise Navigator. To protect proprietary information, we have replaced real system names with randomly generated ones and omitted certain interface names.

gram to position the graph's elements automatically. When the server application finishes the layout, it sends the graph using Java object serialization (http://java.sun.com/products/jdk/1.1/docs/guide/serialization) to the applet, which creates a visualization window and displays the requested system interface diagram. You can select nodes and edges to view their attributes, alter the graph display based on those attributes, and return the graph to the server for additional processing by graph clustering or graph dominator algorithms.

The glue holding EN together is the Java application on the server machine. The EN components linked by the server application include

- System Profile Database, an infrastructure database;
- Grappa, a graph manipulation and display tool;
- Bunch, a graph clustering tool; and
- Dominator, a graph dominator tool.

System profile database

SPDB, the underlying database supplying EN, contains key information about all system entities and interfaces within the enterprise of interest. In AT&T's SPDB, EN primarily uses these three tables:

- The *system table* contains basic information about each system in the entire business enterprise. It also includes such entities as work centers, network elements, databases, and Web sites as well as external systems that participate in data flows to or from other systems within the enterprise. Information about a system can include system type, name, owner, and status; business unit owner; phase-in and phase-out dates; and its parent system.
- The *interface table* gives information about flows between systems and other entities described in the system table. This information can include interface type, owner, and status; the "from" and "to" systems; the business unit owner; and transmission media, frequency, and mode.
- The *mapping table* links other entities such as products and services or business functions to systems in the system table.

In constructing a system interface diagram, EN constructs a graph of the components, setting the systems as nodes and the interfaces as edges. It stores additional pieces of information about those entities as attributes to the graph elements.

Although this article focuses on the SPDB used at AT&T, you can easily modify EN to work with other databases as long as similar information on systems, interfaces, and mappings is available.

Graph Drawing

Graph drawing addresses the problem of visualizing structural information by constructing geometric representations of abstract graphs and networks. The automatic generation of graph drawings has important applications in key technologies such as database design, software engineering, VLSI, network design, and visual interfaces in other domains. In any setting, effective visualization should reveal interesting characteristics of data while avoiding distractions and irrelevancy. Most objective properties for graph layout algorithms correspond to a few simple visual principles:

- Favor recognition and readability of individual objects. Identifying objects should be easy—for example, by giving them legible text labels or by choosing certain shapes, colors, or styles. This principle implies efficient use of available layout area.
- Avoid aliases, including edge crossings, sharp bends, and the intersection of unrelated objects.
- Control eye movement to help users trace edges and paths in diagrams and find sources and sinks. Short and straight or at least monotonic edges are good.
- Reveal patterns by emphasizing symmetry, parallelism, and regularity. Layouts having these characteristics are often easier to read and memorize than ones lacking such organization.

Three families of graph layout algorithms have been particularly successful: hierarchical layouts of trees and directed acyclic graphs, virtual physical layouts of undirected graphs (for example, spring model layouts), and orthogonal grid layouts of planarized graphs.

Graphviz, a set of tools for Unix, Windows, and OSX, has components for the first two families of layouts just mentioned. Source code and binary executables for common platforms are available at www.research.att.com/sw/tools/graphviz. One of the Graphviz tools, called Dot, was used to create most of Enterprise Navigator's layouts.

Grappa is a Java graph-drawing package that simplifies the inclusion of graph display and manipulation capabilities in Java applications and applets. Grappa does not have graph layout capabilities built into it, but integrates easily with tools such as Dot. Moreover, because Grappa stores graph structure information, it simplifies the coding of custom layout algorithms in Java. Grappa also enables questions about the graph structure to be answered easily (for instance, finding nodes that are directly connected to a given node). Grappa is available from the Graphviz Web site. It provided the interactive graph displays we used in Enterprise Navigator.

Graph manipulation and display

A Java package called Grappa[8] handles graph manipulation and display in EN in both the client applet and the server application. It can also build and manipulate graphs independent of display considerations. Although Grappa does not contain layout algorithms, it has methods for simplifying communication with graph layout programs, particularly the Dot layout program.[5]

Mouse interactions with the nodes and edges that Grappa displays can trigger additional actions, including initiating a new query specific to a selected element and viewing or storing additional data about an element. You can study architecture evolution over time because Grappa colors systems according to a reference date (see the Status Reference Date field in Figure 3) and each system's status at that date. Grappa also colors systems that have been phased out of service and those yet to be introduced differently from active systems. Visualizing these changes helps architects determine the effects of business reengineering on various products and services.

Grappa is designed to be extensible—it allows additional graph manipulation methods to be integrated. Without recoding anything, we integrated the next two tools described—Bunch and Dominator—into EN through the server application, which acts as a bridge between those applications and the display applet.

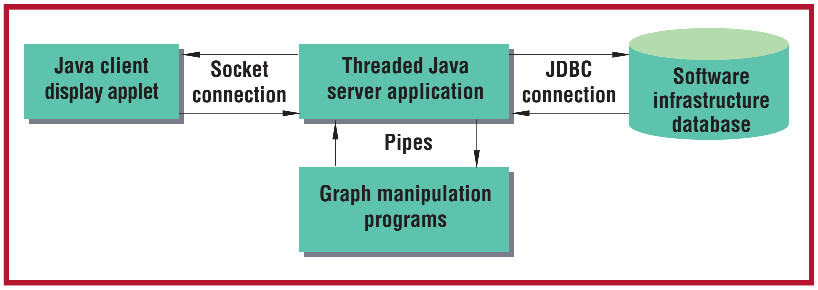
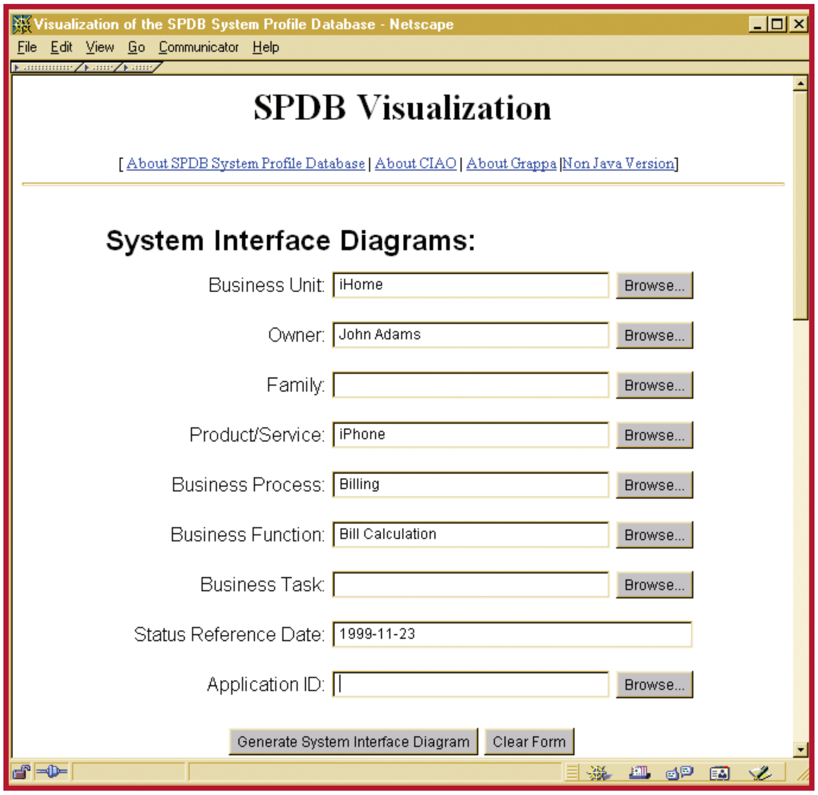

Java client display applet
Socket connection
Threaded Java server application
JDBC connection
Software infrastructure database
Pipes
Graph manipulation programs

Figure 2. The Enterprise Navigator architecture.


Visualization of the SPDB System Profile Database - Netscape
File Edit View Go Communicator Help
Browse...
Generate System Interface Diagram
Clear Form

SPDB Visualization

[About SPDB System Profile Database | About CIAO | About Grappa |Non Java Version]

System Interface Diagrams:

Business Unit:	iHome
Owner:	John Adams
Family:	
Product/Service:	iPhone
Business Process:	Billing
Business Function:	Bill Calculation
Business Task:	
Status Reference Date:	1999-11-23
Application ID:	

Figure 3. The Enterprise Navigator query interface.

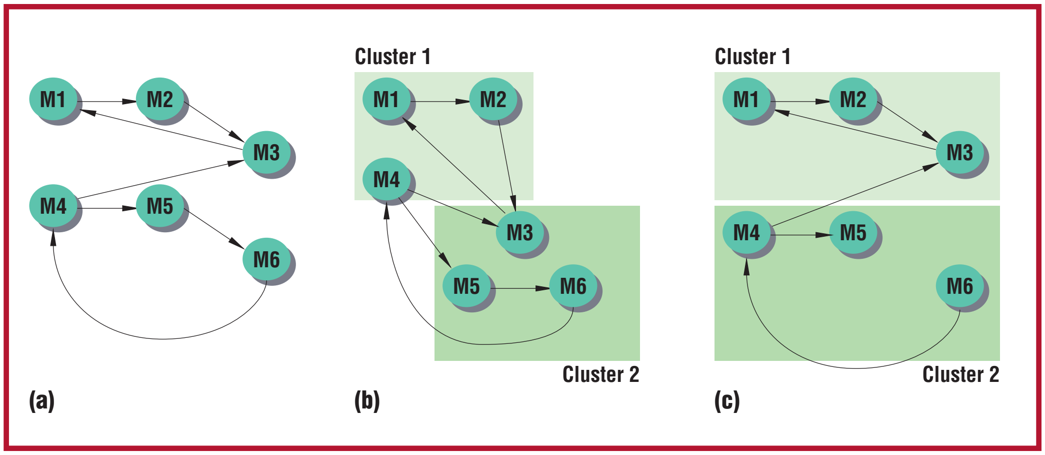
Graph clustering

EN uses the Bunch tool (available at http://serg.mcs.drexel.edu/bunch) to cluster components in a system interface diagram.[6] Clustering is particularly useful to system architects who are trying to understand large and complex software infrastructures from their graph representations.

Bunch accepts a graph as input and outputs it partitioned into a set of nonoverlapping clusters of nodes. Using Grappa, EN can show this partitioned system interface diagram as a graph with node clusters enclosed in rectangles. The Java server application communicates with Bunch through the latter's application programming interface.

Bunch attempts to partition the software graph so that system entities (nodes) in the same cluster are more closely related and system entities in different clusters are relatively independent of each other. Creating a meaningful partition, however, is difficult because the number of possible partitions is large, even for a small graph. Also, small differences between two partitions can yield very different results. As an example, consider Figure 4a, which presents a graph with a small number of entities and relationships. The two partitions of the graph shown in Figures 4b and 4c are similar, with only two nodes (M3 and M4) swapped. Despite this seemingly small difference, the partition defined in Figure 4c better captures the graph's high-level structure because it groups the more interdependent nodes.

Bunch treats graph clustering as an optimization problem, in which the goal is to maximize an objective function that favors creating clusters with a high degree of *intra-*



Figures 4. (a) A small system interface graph; (b) a partition of that graph; (c) a better partition.

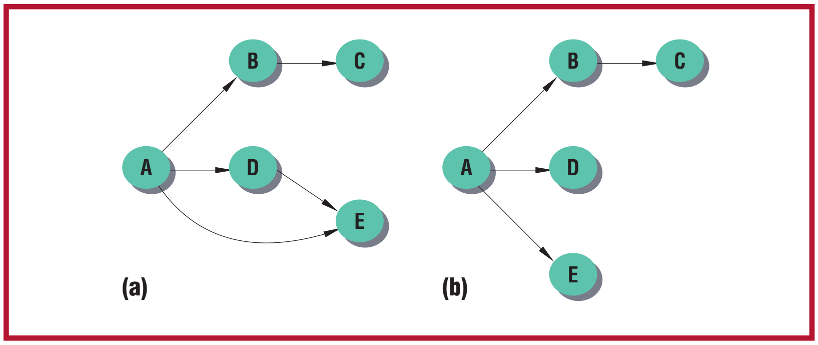


Figure 5. (a) A system interface graph and (b) its dominator tree.

edges, the edges between nodes of the same cluster. The same function penalizes pairs of clusters that exhibit a high degree of *interedges*, the edges between nodes that belong to different clusters. A large number of interedges indicates poor partitioning, which complicates software maintenance because changes to a software system might affect other systems in the software infrastructure. A low degree of interedges, indicating that the individual clusters are largely independent, is a desirable system architecture trait. Changes applied to such a system are likely to be localized to its cluster, thereby reducing the likelihood of introducing errors into other systems.

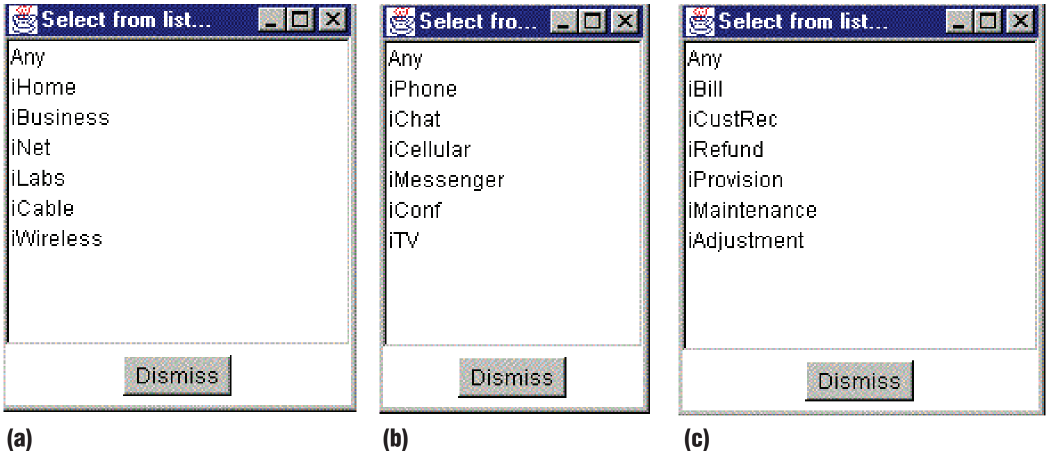
Graph dominators

EN uses the Dominator tool to determine a graph's dominators. In a graph with a selected root node R, node X *dominates* node Y if every path from R to Y goes through X. When EN generates a system interface diagram, each interface link between systems represents an information flow. For example, in Figure 5a, the link from A to B means that information flows from A to B. Figure 5b shows the dominator tree derived from that infrastructure graph. A link in the dominator tree between two nodes means that any information flow from the root node (selected from the original graph) to the target node must flow through the source of the link. For example, if A is the source node and there is a dominator link from B to C, then there is no way to get from A to C without going through B. In other words, if B were to be removed, C would be cut off from any information derived by A. On the other hand, consider the links from D to E and A to E in Figure 5a. The direct link from A to E provides a way to get to E from A without going through D; therefore, D is not a dominator of E. In fact, A is the sole dominator of E.

The root node of a dominator tree represents the system where the information flow logically begins. In cases where a single root is not available, the user can manually choose multiple roots from the graph to act together as the global information source.

Our tool uses the Dominators Algorithm devised by Thomas Lengauer and Robert Endre Tarjan.[7] Adam Buchsbaum and his colleagues[9] provide a history of dominator algorithms as well as theoretical improvements to the Lengauer-Tarjan algorithm. Here are two sample applications for dominator trees in EN:

- Performing a sanity check on system evolution. Removing a system disconnects all systems dominated by it (and by extension systems dominated by those, and so on) from the original information source. Therefore, systems that are scheduled to retire should not dominate any systems that are not retiring. EN lets you check the situation visually by uniquely coloring systems to be retired on a dominator diagram.



Any
iHome
iBusiness
iNet
iLabs
iCable
iWireless

Any
iPhone
iChat
iCellular
iMessenger
iConf
iTV

Any
iBill
iCustRec
iRefund
iProvision
iMaintenance
iAdjustment

Figure 6. Lists of (a) business units and (b) products and services under iHome; (c) systems under iHome that are involved with the iPhone service.

- Qualitatively assessing the dependency complexity. Flat dominator trees—that is, trees in which many systems are directly connected to root nodes—can represent highly interconnected systems, because there are few systems whose removal would disconnect the graph. Such high interconnectivity can be good due to replicated resources for system dependability, or bad due to unnecessary or duplicated information flows. On the other hand, deep dominator trees—that is, trees in which many systems are far from the root nodes—can represent less connected systems because many systems critically depend on many others for connectivity from the root. Again, this may be good or bad, depending on the application.

Case study

Figure 3 shows the query interface presented by the Java applet on the client side. The interface lets you select systems from different business units, owners (managers), products and services, business functions, and so on before generating a system interface diagram. The parameters you select in some categories constrain what choices are available in other categories. For example, when you click the Browse button next to the Business Unit category, the list of all business units appears (see Figure 6a). If you choose iHome and then click the Browse button next to the Product/Service item in Figure 3, Figure 6b appears and shows all products and services under the iHome business unit only. (All business unit names, product names, and system names have been replaced with imaginary names to protect proprietary information.) If you select iPhone and browse the systems under that service, the list of systems appears (see Figure 6c). You can refine the query further by setting the values for Business Process or Business Function, and so on.

You can pick any system from the list to generate a system interface diagram. Figure 7 shows a typical diagram for the fictitious HLJ system. The picture clearly shows that

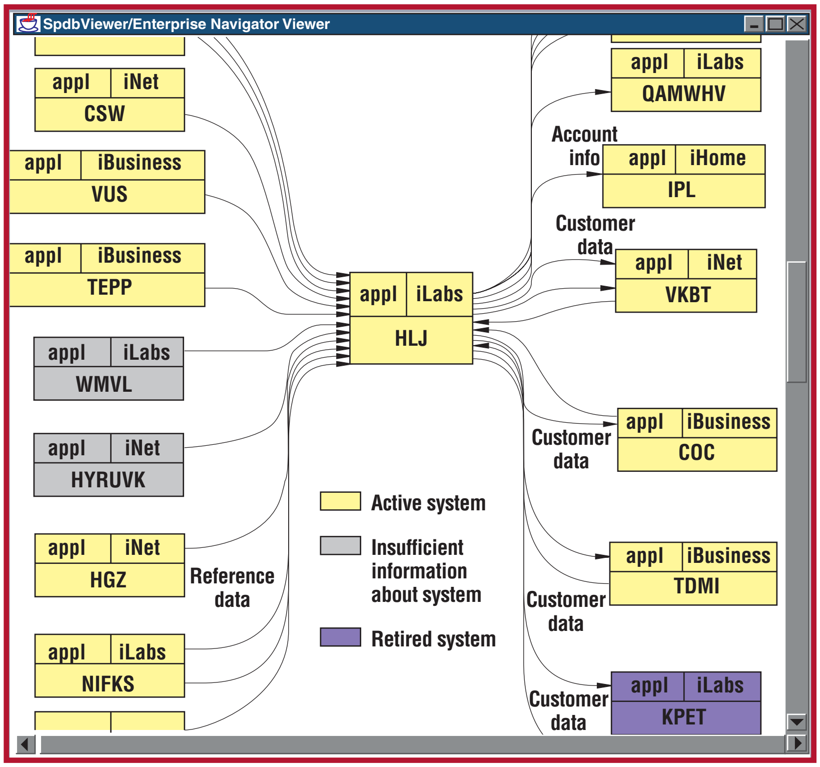


Figure 7. System interface diagram of HLJ (a fictitious system name).

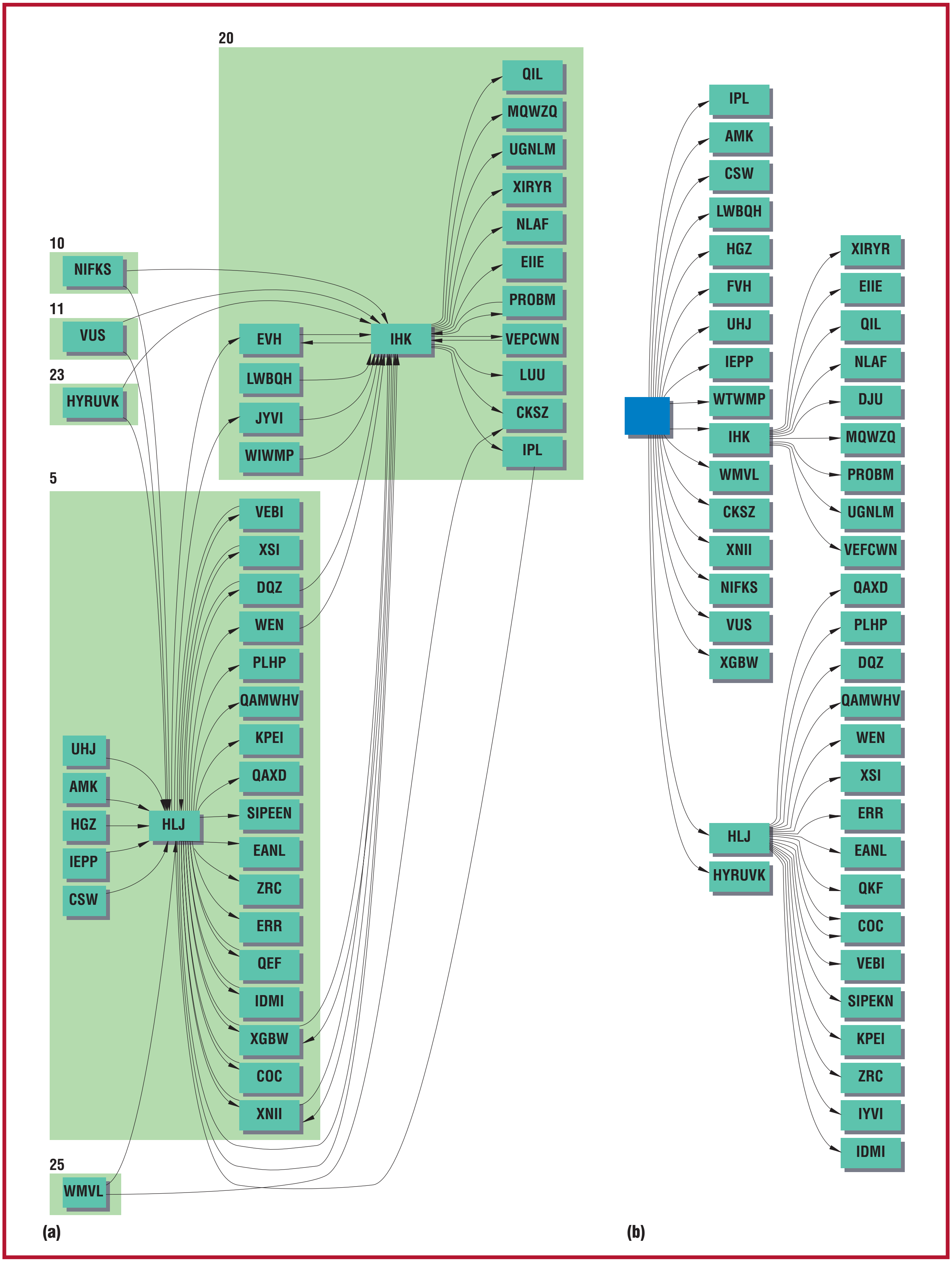


Figure 8. Two diagrams generated from the system interface diagram shown in Figure 1: (a) the clustering diagram and (b) the dominator diagram.

HLJ collects reference data and then distributes accounting and customer data (among other things) to other systems.

In a system interface diagram, you can color each node according to different system attributes. In Figure 7, we use the status shading scheme, in which a yellow node indicates an active system and a gray node indicates insufficient information about that system. Using the reference date (1999-11-23) shown in the query interface page (Figure 3), a blue node indicates that this system is planned and will be introduced soon, and a purple node indicates that this system has retired. Because it is important to know how a business reengineering plan affects the system architecture, the architect can use different reference dates to see how the system interface diagram of a particular product or service has evolved or will change. Another shading scheme we used in the past was Y2K shading, in which nodes were colored according to whether they were Y2K compliant. This scheme let us quickly verify the Y2K readiness of a product or service (assuming the Y2K compliance data was available).

The control bar in the bottom of the window shown in Figure 7 provides several other features. Hitting the "Convert to …" button converts the current interface diagram to various graphics and database formats so that other tools can import it easily. When you click on a system node, the default action is to generate another system interface diagram centered on that node. If, however, you check the Page Link checkbox, a Web browser is invoked to bring up a Web page showing all the system details (software, hardware, contacts, and so on) extracted from SPDB.

Instead of focusing on a single system, you can choose to generate a system interface diagram for all systems involved in a product or service (or any systems that satisfy a particular query). Figure 1 shows a typical system interface diagram of a particular service. If you would like to discover clusters of systems embedded in a complex structure, you can invoke the Bunch tool to convert Figure 1 to a clustering diagram, shown in Figure 8a. The diagram shows two clusters with a similar architectural pattern: each has a data hub that receives data from several sources and distributes processed data to many other destinations. Identifying clusters from complex system interface diagrams without the help of automated tools is not always easy.

If you want to discover the dominating information flows starting from a particular system, you can select the node and perform a dominator analysis. Alternatively, EN can perform topological sorting to rank the nodes and add a virtual root to all the top-level nodes before starting the dominator analysis. Figure 8b shows a dominator tree created with this method for the system interface diagram of Figure 1. Clearly, any attempt to remove the HLJ system will affect all the other systems that it dominates, because any information flows to the product or service represented by Figure 1 must go through HLJ. Such information can help system architects plan their reengineering efforts.

Enterprise Navigator's usefulness depends heavily on the underlying data's timeliness. We are working on facilities that would let architects update architecture data directly from graphs, thus eliminating the delays associated with the old data collection process. We also plan to add node and link operators that will let users examine in more detail the corresponding systems and data transmitted on a link. With the addition of operational-data-like system availability to the database, we might be able to perform end-to-end enterprise architecture simulations. Finally, we welcome the opportunity to apply the EN concept to other forms of enterprise data. Most of the tools and libraries we use are already available on the Web, and we plan to provide a reusable package that would simplify their integration with other infrastructure databases. For more information on EN and pointers to the software components it uses, visit www.research.att.com/~ciao/en.

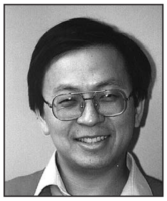
References

1. Y. Chen et al., "Ciao: A Graphical Navigator for Software and Document Repositories," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 66–75.
2. Y. Chen, E.R. Gansner, and E. Koutsofios, "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection," *IEEE Trans. Software Eng.*, vol. 24, no. 9, Sept. 1998, pp. 682–693.
3 J. Korn, Y. Chen, and E. Koutsofios, "Chava: Reverse Engineering and Tracking of Java Applets," *Proc. 6th Working Conf. Reverse Eng.*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 314–325.
4. G. Di Battista et al., *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, Upper Saddle River, N.J., 1999.
5. E.R. Gansner et al., "A Technique for Drawing Directed Graphs," *IEEE Trans. Software Eng.*, vol. 19, no. 3, Mar. 1993, pp. 214–230.
6. S. Mancoridis et al., "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," *IEEE Proc. Int'l Conf. Software Maintenance* (ICSM '99), IEEE CS Press, Los Alamitos, Calif., 1999.
7. T. Lengauer and R.E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Trans. Programming Languages and Systems*, vol. 1, no. 1, 1979, pp. 121–141.
8. N. Barghouti, J. Mocenigo, and W. Lee, "Grappa: A Graph Package in Java," *Proc. 5th Int'l Symp. Graph Drawing*, Springer-Verlag, Berlin, 1997, pp. 336–343.
9. A.L. Buchsbaum et al., "A New, Simpler Linear-Time Dominators Algorithm," *ACM Trans. Programming Languages and Systems*, vol. 20, no. 6, Nov. 1998, pp. 1265–1296.

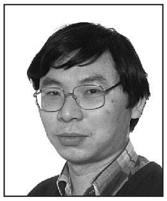
For more information on this or any other computing topic, please visit our Digital Library at http://computer/publications/dlib.

About the Authors

Adam Buchsbaum is a principal technical staff member in the Network Services Research Lab at AT&T Labs. He specializes in the design and analysis of algorithms and data structures. His research interests also include graph problems, massive data sets, and combinatorics. He received his PhD in computer science from Princeton University. Contact him at Room E203, Building 103, 180 Park Ave., PO Box 971, Florham Park, NJ 07932-0971; alb@research.att.com; www.research.att.com/info/alb.



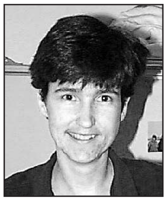
Yih-Farn Chen is a technology consultant in the Software Systems Research Department, Network Services Research Center, at AT&T Labs. His research interests include mobile computing, software engineering, and the Web. His recent work includes a mobile service platform, Web-site tracking services, personal proxy servers, a reverse engineering portal, and Enterprise Navigator. He was a vice chair of the WWW10 and WWW11 conferences. He received his PhD in computer science from the University of California, Berkeley. Contact him at Room E219, Building 103, 180 Park Ave, PO Box 971, Florham Park, NJ 07932-0971; chen@research.att.com; www.research.att.com/info/chen.



Huale Huang is a senior technical staff member in the Information Sciences Research Lab at AT&T Labs, where he is working on mobile communications. He is interested in Web and mobile computing. He received an MS in mathematics from the Chinese Academy of Sciences, an MS in computer science from the New Jersey Institute of Technology, and a PhD in mathematics from City University of New York. Contact him at Room C258, Building 103, 180 Park Ave, PO Box 971, Florham Park, NJ 07932-0971; huale@research.att.com.

Eleftherios Koutsofios is a technology consultant at AT&T. His research interests include interactive techniques, display and user interaction technologies, and information visualization. He has worked on graph layouts, programmable graphics editors, tools for visualizing large data sets, and program animation. He received his PhD in computer science from Princeton University. Contact him at Room E223, Building 103, 180 Park Ave, PO Box 971, Florham Park, NJ 07932-0971; ek@research.att.com; www.research.att.com/info/ek.

John Mocenigo is a principal technical staff member with the Network Services Research Lab at AT&T Labs. He enjoys problem solving and writing code for graph visualization and database transaction logging. Currently he is working on a scripting language called Yoix that runs under Java (www.research.att.com/sw/tools/yoix). He received his PhD in electrical engineering–control theory from Brown University. Contact him at Room D225, Building 103, 180 Park Ave, PO Box 971, Florham Park, NJ 07932-0971; john@research.att.com; www.research.att.com/info/john.



Anne Rogers is a technology consultant in the Network Services Research Lab at AT&T Labs. She specializes in programming languages, compilers, and systems for processing large volumes of data. She received a BS from Carnegie Mellon University and an MS and a PhD from Cornell University. Contact her at Room E205, Building 103, 180 Park Ave, PO Box 971, Florham Park, NJ 07932-0971; amr@research.att.com; www.research.att.com/info/amr.

Michael Jankowsky is a senior technical staff member in the Enterprise IT Security Group within AT&T Business Services. He specializes in database design and data management. He received a BS in mathematics from Montclair State University. Contact him at AT&T Business Services, Room E5-2A03, 200 Laurel Ave. South, Middletown, NJ 07748; jankowsky@ems.att.com.

Spiros Mancoridis is an associate professor in the Department of Mathematics and Computer Science at Drexel University and founder and director of the Software Engineering Research Group there. His research involves reverse engineering of large software systems, program understanding, software testing, and software security. In 1998, he received a Career Award for young investigators from the US National Science Foundation. He received his PhD in computer science from the University of Toronto. Contact him at the Dept. of Math & CS, Drexel University, Philadelphia, PA 19104; smancori@mcs.drexel.edu; www.mcs.drexel.edu/~smancori.