Reduction of Visual Complexity in Dynamic Graphs

Doug Kimelman, Bruce Leban,
Tova Roth, Dror Zernik

IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598

Abstract. Graphs are used extensively in software visualization to represent both static aspects of software structure and dynamic aspects of execution-time behavior. However, for realistic subject software systems, there are far too many nodes and edges in the displayed graphs to be comprehensible to an end user. Further, for presentation of dynamics, continual change and redisplay of such large graphs is too demanding for conventional workstation computational resources. This paper poses the problem of "reduction" or "abstraction" in dynamically changing graphs, and proposes a combination of techniques that can be used to reduce the visual complexity of a graph, without obscuring the significant information that it was meant to convey. The abstract graph can be comprehended more readily and it changes far less frequently than the full graph. As well, when the abstract graph does change, it requires far less computation for layout and redisplay. These abstraction techniques are illustrated by way of examples showing their use in systems for visualization of object-oriented and multi-layer software systems.

1 Introduction

Software visualization [17] [9] [3] [18] often makes extensive use of graphs to represent software systems. Graphs have been used within the "PV" system [7] to represent both static aspects of software structure, such as class hierarchies, static call graphs, and program dependence graphs [8], and to represent dynamic aspects of execution-time behavior, such as process hierarchies and dynamic call graphs. In some cases, behavioral dynamics have been shown by animating static graphs (e.g. by varying the color of nodes and edges in a static call graph to show profiling information), while in other cases truly dynamic graphs have been displayed, with structure varying over time.

The "Ovation" system [4] introduced the "call cluster", a dynamic undirected graph laid out to reflect the degree of interaction between various pairs of objects or classes in an object-oriented system (the closer two objects or classes are, the more they have interacted). The "Visage" system [19] includes three dimensional layouts of causality graphs to show individual communication operations between processors over time in a parallel system.

Unfortunately, for realistic subject software systems, there are often far too many nodes and edges in these graphs to be comprehensible to an end user. Programs with thousands of functions or classes, tens of thousands of objects, and hundreds of thousands of communications are not uncommon. As graphs approach such sizes, conventional technology and graph display techniques become inadequate. Displays first become cluttered and eventually just become flooded with a sea of color until no discernible structure remains. Further, changes occur frequently in these large graphs, making the cost of repeated layout and redisplay prohibitive. With the computational resources available on conventional workstations, it becomes impractical to keep up with ongoing activity within the graph.

This paper proposes a combination of "abstraction" or "reduction" techniques for reducing the visual complexity of a graph, while preserving or even enhancing the significant information that it was meant to convey. A number of means are provided for automatically selecting nodes and edges in a graph: thresholds on weights, regular expressions on labels, relationships within the graph, and constraints related to the semantics of the graph; and operations are provided for then "disposing" of these selected nodes: "ghosting" (relegating nodes to the background visually), "hiding" (removing nodes from the display entirely), and "grouping" (grouping nodes under a single new meta-node). Multiple selection and disposal operations can be applied in sequence. After a user arrives at a suitable sequence of operations for a given graph through interactive experimentation, the system automatically displays the "abstract" or "reduced" form of all subsequent instances of the graph, as it evolves through continual change. Note that, with this form of abstraction, not every change to the full graph results in a change to the abstract graph.

An abstract graph achieved in this way can be comprehended more readily by an end user, and it changes far less frequently than the full graph. As well, whenever the abstract graph does change, it can be laid out and displayed more quickly. Thus, it becomes feasible for conventional systems to keep up with ongoing activity within the graph.

2 Problem Statement

The basic abstraction problem introduced by this paper can be stated as follows.

The focus is on large and continually changing directed or undirected graphs. Each node and edge has a label (an arbitrary string), a weight (an arbitrary positive integer), and a "context" property (an arbitrary integer), all of which are set and continually updated at the request of the application which is a client of the graph package. The context property is used to establish equivalence classes related to semantics. For example, in a dynamic call graph, all of the functions in a particular module or file could be grouped into a single equivalence class (that is, they serve as context for each other). In an object interaction graph, all of the nodes and edges related to the current call stack could form a single equivalence class.

The application continually makes requests to add and delete nodes and edges, and to changes labels, weights, and properties. In the case of a system for visualizing dynamics of software behavior, the visualization system is the client application. As it consumes event records describing execution history, it makes requests to the graph package. As well, as time passes, the client may effect gradual "aging" or "decay" of information by making requests to the graph package to decrease the weight of nodes and edges.

The requirement, as discussed above, is to reduce the complexity of the graph, or de-emphasize components of the graph, so that: it can be comprehended more readily by an end user, it changes less frequently, and it can repeatedly be laid out and displayed quickly by the system. As well, stability should be induced between successive instances of the dynamic graph as it evolves through continual change. Stability is important both for its own sake and so that changes in the graph are more apparent, that is, so that artifacts of graph layout and display do not obscure the actual changes in the graph. Finally, some means of maintaining context or frame of reference as the graph evolves should be provided to the end user.

Throughout, the information which was intended to be conveyed by the graph, i.e. the essence of the graph, must be preserved or enhanced. The actual layout of the graph may be significantly altered, but the meaning of the layout and dynamics should remain the same. For example, in an object interaction graph, the clustering should continue to reflect which groups of objects work closely together.

3 Proposed Abstraction Techniques

3.1 General Approach

The general approach taken in this work is for abstraction to take place outside of, and independent of, actual graph layout. This approach was adopted for a number of reasons: layout packages are often provided in "black box" form, and hence there is no possibility of incorporating abstraction extensions into them; an abstraction technique which is independent of layout is more broadly applicable than one tied to a particular layout algorithm; independence allows a user to employ domain-specific layout algorithms wherever applicable, yet still be able to retain the benefits of abstraction; and, an independent abstraction facility can mask the instability or sensitivity of underlying layout algorithms to small changes (thereby freeing a user with dynamic graphs from having to locate a suitable incremental graph layout package).

An important aspect of generality of the abstraction techniques is that they are "graph abstraction" rather than "semantics-driven abstraction in the application domain". That is, these techniques are cast strictly in terms of the abstract properties of the graph e.g. the actual structure of the graph, weights, labels, and context properties; and they are based as little as possible on "hard-coding" concerning semantics of specific uses of the graph. For instance, nothing is built into these techniques concerning dynamic call graphs specifically.

3.2 Overall Process

The proposed abstraction facility maintains two graphs: the actual or "raw" graph, and a reduced or "abstract" graph. (Note that the cost of the additional graph is small, as it is orders of magnitude smaller than the raw graph.) The abstract graph is obtained from the raw graph by disposing of a number of nodes and edges, and then invoking the same layout package as would normally be used for the raw graph. Minor adjustments might then be applied to the layout, and the abstract graph is displayed.

Operations are carried out on the raw graph as they are requested by the client application, but the raw graph is not repeatedly laid out (unless the end user has asked to have the raw graph displayed along with the abstract graph). As each operation is carried out on the raw graph, it is considered, to determine whether it affects the abstract graph. (This process is less costly than either laying out the entire raw graph, or mapping the entire raw graph into an abstract graph.) In cases where the abstract graph is affected, it is updated, laid out, adjusted, and then redisplayed.

3.3 Specific Techniques

Because nodes and edges are treated in much the same way with respect to abstraction, for the sake of simplicity this discussion will refer only to nodes.

A number of criteria are provided for automatically selecting nodes and edges in a graph:

- thresholds on weights e.g. above or below some value, or within some range (thresholds on the *gradients* of weights allows selection of nodes which have recently become active but which have yet to accumulate much weight)
- regular expressions on labels e.g. all matching, or all not matching
- relationships within the graph, e.g. any edge touching a selected node, or any node which is a parent (or a child) of a selected node
- constraints related to the semantics of the graph e.g. any node identified as being in the context of a selected node (based on the value of the context property established by the client application according to semantics).

Once a set of nodes has been selected, a number of operations are provided for "disposing" of selected nodes:

- "ghosting" — relegating nodes to the background visually, e.g. for displays with gray backgrounds, coloring nodes a slightly different shade of gray, and coloring labels and edges white
- "hiding" — removing nodes from the display entirely
- and "grouping" — grouping nodes under a single new meta-node, or in the case of nodes selected using a regular expression, forming a number of distinct groups, one for each distinct string matched by the regular expression, and then placing into a given group all nodes for which the regular expression matched the string corresponding to that group e.g. where labels are of the

form 'file:function', and a regular expression is constructed to match the 'file' part of the labels, a distinct group would be formed for each file, and all of the functions from a file would be placed in that file's group.

Multiple selection and disposal operations can be applied in sequence. After a user arrives at a suitable sequence of operations for a given graph through interactive experimentation, the system automatically displays the "abstract" or "reduced" form of all subsequent instances of the graph, as it evolves through continual change.

Stability, a common deficiency of conventional layout algorithms for dynamic graphs, is addressed in two ways for the abstract graph. In both cases, the approach involves altering the coordinates of a node (and related edges) after the layout algorithm has positioned the elements of the graph, but before the graph is actually displayed. To combat "jitter", or oscillation of a node among a number of almost identical positions, very small changes in the position of a node are simply eliminated. To combat radical shifts of nodes in the graph, smoothing is achieved by limiting the amount that a node can travel on each successive instance of the abstract graph.

A means for the end-user to maintain context, or basic frame of reference, as the graph evolves, is provided in the form of correlation mechanisms. At any time, the user can continuously vary the thresholds applied to the weights in order to reveal successively more of the hidden nodes of the graph. This allows the user to see the relationship between the nodes in the abstract graph and other nodes in the raw graph (possibly less transient and hence more stable as a point of reference). As well, the user can cause the layout of a more abstract graph to transform smoothly to that of a less abstract graph i.e. to one in which the positions of the nodes corresponds more closely to their position in the less abstract graph. This allows a user to correlate the nodes of one graph to their counterparts in another.

One abstraction alternative that experience has shown *not* to be an effective means of achieving preservation of context is that of attempting to maintain spatial similarity between the raw and abstract graphs. One approach might involve removing nodes from the graph and then employing some form of non-linear transformation to scale the graph so that the basic overall shape of the graph is preserved, while remaining groups of elements of the graph are brought closer together, and individual groups are expanded, in order to better utilize the available space (in effect, a multiple fisheye) [15]. Any such approach attempted was found to lead to displays which were still too sparse.

4 Examples

As discussed above, the domain of interest for this work is software visualization. In particular, the focus is on large and continually changing graphs representing dynamic aspects of execution-time behavior of a software system. Two examples of graphs which are candidates for abstraction are: a dynamic call graph (DCG) showing profiling information, and an object interaction graph (OIG) showing degree of interaction among objects.[1]

For DCGs displayed by the "PV" system [7] [2], sequences of abstraction operations which proved to be effective included: hiding all nodes whose weights are be[illegible] threshold; and grouping function nodes according to module, while preserving [illegible]ts and context. For OIGs displayed by the "Ovation" system [4], sequences of abstraction operations which proved to be effective included: ghosting applied to nodes with weights below a threshold; and hiding all nodes whose weights are below a threshold, yet allowing the layout to be transformed so that node positions are the same as in the graph to which hiding had not been applied.

5 Related Work

Many systems allow zooming and panning through a large graph, but scrolling is not an effective means of "getting a feel" for the overall structure of a graph.

The importance of reducing the visual complexity of the overall graph is addressed in [2] and [16], where sophisticated and hierarchical displays are considered. These approaches are based solely on altering the display of a graph, without removing any elements from the graph or otherwise reducing the complexity of the underlying graph itself. An attempt to provide automatic reduction of graph complexity by applying an abstraction operation is described in [14]. In this case, only the number of edges in the graph is reduced, not the number of nodes. Further, only the structure of the graph is considered. Weights, labels, and other information in the graph are not exploited.

A more general tool for abstraction and parsing of graphs can be achieved using graph grammars [13]. An approach which uses graph grammars for layout is described in [1]. The complexity of such an approach for abstraction, however, would make it computationally infeasible.

Abstraction of graphs used to represent databases is addressed in [11] [10] and [6], but none of these consider the issue of dynamically changing graphs.

Much work has been reported recently concerning the related issue of incremental graph layout algorithms [5] [12]. This work concentrates on reducing the cost of repeatedly laying out a graph, by requiring only computation proportional to the amount of change in the graph. There is no reduction in the visual complexity of the displayed graph.

The issue of continuity of the display of a series of graphs over time is mentioned in [6] but not addressed.

[1] An expanded version of this paper, containing definitions for DCG and OIG, detailed examples, and color figures, is available from any of the authors as an IBM Research Center Technical Report, or on the World Wide Web from `http://www.ibm.com` under Technology and Research and Program Visualization.

[2] PV uses the "NARC" package [20] for all layout and display of directed graphs.

6 Conclusion

Dynamic graphs representing realistic subject software systems are often far too large and changing far too rapidly to be handled effectively with conventional techniques and display technology. However, with suitable combinations of "reduction" and "abstraction" techniques, the complexity of such graphs can be reduced to the point where they are comprehensible to an end user, they change less frequently, and they can be laid out and displayed more quickly, yet they still convey the information that was originally intended. Further, it becomes feasible to display these graphs with conventional technology, at the rates required for software visualization.

Experience has shown that sequences of selection operations on nodes and edges, based on weights, labels, structure, and semantics-related properties, followed by disposal operations including ghosting, hiding, and grouping, are an effective means of automatically reducing complexity in a series of instances of an evolving dynamic graph. We believe that this form of abstraction will prove to be essential for effective use of graphs in software visualization.

Acknowledgments

Thanks to Wim De Pauw for the Ovation cluster views and much of the work on Ovation, to Bryan Rosenburg for the exponential decay algorithm, the dynamic call graph view, and much of the work on PV, and to Vance Waddle for the NARC package for graph layout and display.

References

1. F.J. Brandenburg "Layout Graph Grammars: The Placement Approach", Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science 532, Springer-Verlag, 1991, pp. 144–156.
2. M.J. Carpano "Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysis", IEEE Trans. Systems, Man, and Cybernetics, SMC-10(11), 1980, pp. 705–715.
3. M.P. Consens, A.O. Mendelzon, and A.G. Ryman "Visualizing and Querying Software Structures" Proc. 14th Intl. Conf. on Software Engineering, 1992, pp. 138–156.
4. W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides "Visualizing the Behavior of Object-Oriented Systems", Proc. OOPSLA '93 The Eighth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices 28(10), October 1993, pp.326–337.
5. G. Di Battista and R. Tamassia "Incremental Planarity Testing", Proc. 30th IEEE Symp. on Foundations of Computer Science, 1989, pp. 436–441.
6. K.M. Fairchild, S.E. Poltrock, and G.W. Furnas "SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases", in *Cognitive Science and its Applications for Human-Computer Interaction*, R. Guindon Ed., Lawrence Erlbaum Associates Publishers, 1988, pp. 201–233.

7. D. Kimelman, B. Rosenburg, and T. Roth "Strata-Various :-) Multi-Layer Visualization of Dynamics in Software System Behavior", Proc. IEEE Visualization '94, October 1994, pp. 172–178.
8. D. Kimelman and G. Sang'udi "Program Visualization by Integration of Advanced Compiler Technology with Configurable Views", Proc. CNRS-NSF Collaboration Workshop on Environments and Tools for Parallel Scientific Computing, Saint Hilaire du Touvet, France, J.J. Dongarra and B. Tourancheau, editors, Elsevier Science Publishers, September 1992, pp. 73–84.
9. E. Kraemer and J. Stasko "The Visualization of Parallel Systems: An Overview", Journal of Parallel and Distributed Computing 18(2), 1993, pp. 105–117.
10. Y. Hara, A. Keller, P. Rathmann, G. and Wiederhold "Implementing Hypertext Database Relationships Through Aggregations and Exceptions", Stanford Technical Report STAN-CS-91-1381, September 1991.
11. M.P. Consens, F.C. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista "Architecture and Applications of the Hy+ Visualization System", IBM Systems Journal 33(3), 1994, pp. 458–476.
12. K. Miriyala, S.W. Hornick, and R. Tamassia "An Incremental Approach to Aesthetic Graph Layout", Proc. International Workshop on Computer-Aided Software Engineering (CASE '93), 1993.
13. M. Nagl "A Tutorial and Bibliographical Survey on Graph Grammars", Proc. of International Workshop on Graph Grammars and Their Applications to Computer Science and Biology, Lecture Notes in Computer Science, 1978.
14. F.J. Newbery "Edge Concentration: A Method for Clustering Directed Graphs", Proc. 2nd International Workshop on Software Configuration Management, 1989, pp. 76–85.
15. E.G. Noik "A Space of Presentation Emphasis Techniques for Visualizing Graphs", Proc. Graphics Interface '94, May 1994, pp. 225–234.
16. S.C. North "Drawing Ranked Digraphs with Recursive Clusters", Private Communication, Draft from November 1993.
17. B.A. Price, R.M. Baecker, and I.S. Small "A Principled Taxonomy of Software Visualization", Journal of Visual Languages and Computing 4(3), 1993, pp. 211–266.
18. S.P. Reiss "A Framework for Abstract 3D Visualization", Proc. IEEE Symposium on Visual Languages, August 1993, pp. 108–115.
19. A. Rudich, D. Zernik, and G. Zodik "Visage - Visualization of Attribute Graphs: A Foundation for a Parallel Programming Environment", Proc. CNRS-NSF Collaboration Workshop on Environments and Tools for Parallel Scientific Computing, Saint Hilaire du Touvet, France, J.J. Dongarra and B. Tourancheau, editors, Elsevier Science Publishers, September 1992, pp. 171–192.
20. V. Waddle and A. Malhotra "NARC/X Nodes and ARCs Graph Widget API Definition", IBM Research Center Technical Report, 1993.