The Dominance Tree in Visualizing Software Dependencies

Raimar Falke, Raimund Klein, Rainer Koschke, Jochen Quante
University of Bremen, Germany
http://www.informatik.uni-bremen.de/st/
{rfalke, ray, koschke, quante}@informatik.uni-bremen.de

Abstract

Dominance analysis from graph theory allows one to locate subordinated software elements in a rooted dependency graph. It identifies the nesting structure for a dependency graph as a dominance tree, and, hence, adds information not immediately visible in large and complex graphs. Moreover, the subordination (or locality) can be leveraged for drawing dependency graphs.

This paper envisions ways to leverage the dominance relation for structuring and presenting large dependency graphs. To explore the feasibility of these kinds of visualization, we measure dominance trees for large software systems written in different programming languages. These measurements give us the necessary information to design a usable visualization.

1 Introduction

Koschke's survey of software visualization has shown that graphs are the most popular means to convey information on software systems visually [7]. The survey has also shown that the most often visualized artifacts are those at the mid-level, i.e., artifacts between the conceptual architecture and source code, namely, call graphs, module dependencies, global variables, routines, and user-defined types and their interrelationships, and class and object models. All these examples fall in the class of dependency graphs. The classical tool Rigi [8] uses graphs to visualize dependencies among software units. Other reverse engineering tools have followed this paradigm. Examples include Shrimp [9] and its successor Creole as well as Bauhaus [1].

Graphs are a 'natural' representation[1] of dependencies, and they offer an immediate 'natural' visualization as nodes and edges in two or three dimensional space. Graph drawing is an active field of research, which has proposed many automatic graph layout algorithms. Still, dependency graphs grow in the size of the software systems from which they are derived. Large graphs are difficult to layout automatically due to the often polynomial asymptotic time complexity of layout algorithms. Worse, large graphs are also difficult to grasp for a human reader even if the graphs could be laid out efficiently and pleasantly.

One way to make larger graphs more accessible is to impose a structure, which allows one to merge several related nodes into a summarizing node; that is, one uses hierarchical graphs whose composite nodes may hide subsumed nodes. The question then is what structure that should be? Software systems often have a hierarchical structure as specified by the programming language (attributes and methods – classes – packages) or by the file system (global declarations – files – directories). But what if this is not the structure under which we want to view the system, or if it cannot be used, for instance, because we want to see the call graph? In such cases, one can use software clustering techniques which attempt to group related entities. However, the underlying clustering criteria might not always be the one a viewer would agree to (for instance, clustering by similar names). The clustering criteria might be orthogonal to the dependency relation; a mixture of both in one view might lead to visually unpleasant results.

One possible solution to a more closely related kind of clustering is dominance analysis. Dominance analysis is a well known concept from graph theory for rooted graphs. Dominance analysis has been used in reverse engineering to identify modules and subsystems [3, 5, 2]. Here, we explore its use to visualize dependency graphs. We develop ideas on how to visualize dependency graphs enriched by dominance information. Then we measure dominance trees for large software systems. These measures are used to explore the feasibility of visualization based on dominance. This work

[1] Whatever 'natural' means when it comes to immaterial software.

is part of ongoing research. We invite other researchers to use these data to develop their own ideas on how to leverage dominance information for visualization.

2 Dominance Tree

Dominance for rooted graphs is defined as follows. Let R be the root of a graph. Node D *dominates* node N if and only if every path from R to N contains D. In other words, we need to pass by D to get to N. Hence, N can be considered local to D. Every node, say N, —except the root—has a unique direct dominator: the dominator of N that is dominated by all other dominators of N.

The direct dominance relation, hence, forms a tree. Fig. 1 shows an example dominance tree. The dominance tree highlights the scopes in the underlying dependency graph. If N is dominated by D, only nodes also dominated by D view[2] N. That means, there cannot be any node outside the subtree rooted by D that has a dependency on N. In other words, the subtree is closed under incoming dependencies. Inversely, there can be, however, outgoing dependencies. Fig. 1 shows the nodes visible to N as the gray area (let N be the highlighted node). N views its immediate children, but not their children. We call dependency edges starting at a node at level l in the dominance tree toward a node at level k where $k > l$ a *downward edge* (levels increase from 1 for the root toward the leaves of the tree). N may also view its siblings. Such dependencies staying at the same level are called *sibling edges*. Finally, N can view all its dominators and their direct children. Dependencies from dominance tree levels l to levels $k < l$ are called *upward edges*

Beyond its linear time complexity [4], the advantage of dominance analysis as a clustering technique is that it immediately derives from the dependencies which ought to be visualized. It provides additional information; that is, it extends the dependency information. It tells a human viewer the nesting in terms of visibility in the original dependency graph. Every dominator constitutes a kind of interface to the node it dominates (all dependencies to them must pass through the dominator). Hence, it is an immediate way of abstracting many nodes into one.

One fundamental prerequisite of dominance analysis is the existence of a root. For complete systems, we can use the main program. For some languages, such as C and C++, it is trivial to identify the main program (which is the `main` function). In other languages, such as Ada it is more difficult because any parameterless

[2]may have access to

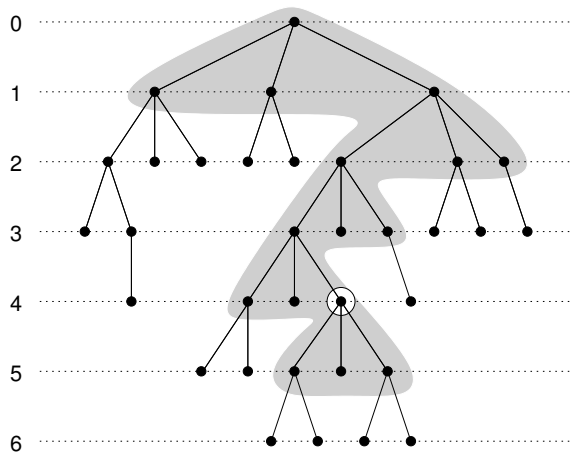


Figure 1. Dominance tree and visibility.

procedure delivered as own compilation unit could take on the role of the main program. The definition of the actual main program is delivered to the compiler. Here, the user would need to determine the main program.

More difficult is the fact that some software systems do not have a main program at all. For instance, if one wants to analyze a library. In such cases, one has to determine the interface of the library and, similarly to multi-main programs, add an artificial root accessing every interface item. Heuristics may be helpful to detect an interface (such as everything that is declared public), but likely this task cannot be automated completely, in which case—again—the user needs to be asked.

Dominance analysis, further, requires that there exists a path from the root to every node. For systems with dead code some will not be reached, of course. If one still wants to take them into account, one could detect connected components in the graph and then connect them through a new artificial root. However, only for the subgraph reachable from the main program, the root is known in advance. For all others, the user must be asked.

3 Visualizing Dependency Graphs

The dominance tree imposes a structure on the tree directly derived from the original dependencies. Nodes in a common subtree of the dominance tree are also close in the original dependency graph (there cannot be any incoming edge from outside of the subtree, although there can be outgoing edges). These properties make it a promising help for visualizing dependency graphs. In this section, we explore some initial ideas on how this information can be leveraged.

3.1 Hierarchical Graphs

The most obvious way of taking advantage of dominance information is to turn the dependency graph into a hierarchical graph where all children of a dominator are collapsed into the dominator at every level. To maintain the dependency edges from collapsed nodes, edges may be lifted from collapsed nodes to their composite node. A user can then view the top level nodes of the dominance tree as first overview and step-wise unfold the nodes on demand to dig into details as, for instance, supported by Shrimp/Creole.

If there are considerable fewer nodes directly dominated by the root, the visual and cognitive complexity is reduced. The visual complexity relates to the number of nodes to be visualized, the cognitive complexity is reduced through the abstraction of subordinate nodes into their dominators.

Abstraction is an important means to better understand a software, but the hiding of nodes also has its drawbacks: Nodes the user is interested in might be hidden in lower levels. Getting a complete picture of the dependencies is diffcult.

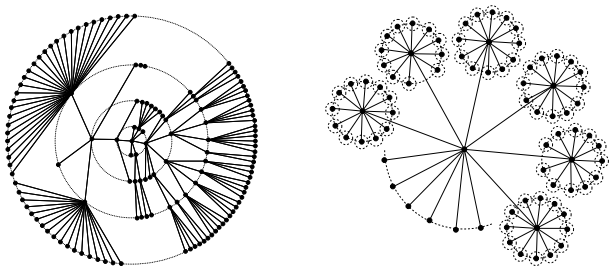
It would be nice if we had structure in terms of dominance and all details at the same time. One way to achieve this is to use the direct dominance relation as yet another type of edge that can be visualized. The next section delves into that.

3.2 Tree-Based Graph Drawing

One class of common graph drawing algorithms for general graphs is based on trees. First, the layout of a spanning tree of the graph is calculated, and then, the additional edges in the graph are inserted into the drawing. The spanning tree defines a hierarchy that can be used for layer assignment as in Sugiyama's algorithm. The tree can always be drawn without crossings very easily, while the additional edges might require advanced crossing minimization techniques. The problem in these approaches usually is to find a good spanning tree that also has some meaning. We are proposing to use the dominance tree for this purpose. The dominance tree is not necessarily a spanning tree, but it definitely has some meaning for the graph because it is directly derived from it.

Concerning graph drawing, the advantages of trees compared to general graphs are that they can easily be drawn without edge crossings, and that they define a hierarchy which can be directly used for the drawing. Tree layout algorithms have a low complexity and are easy to implement. The following layouts are usually used for drawing general rooted trees [6]:

- Parallel levels (Fig. 1): Each node is put onto one of a set of parallel levels according to its distance to the root. This is the classical tree layout. There are several well-known algorithms that optimize for different aesthetic criteria to make the tree more readable.
- Circular levels (Fig. 2(a)): The root is put into the middle of a set of concentric circles. Each node is then put onto one of these circles according to its depth in the tree.
- Balloon view (Fig. 2(b)): Projection of the three dimensional cone tree onto the plane. The children of each node are placed on a circle around their parent.



(a) Circular levels (b) Balloon view

Figure 2. Other tree layouts [6].

Another question is—once the dominance relation is laid out as a tree—how the original dependencies should be fit in? One aesthetic criterion is to minimize the number of edge crossings. Luckily, because the dominance tree is derived from the dependencies, it has certain properties that may be leveraged here. These properties relate to the scopes as discussed in Section 2:

- there cannot be any incoming edges from outside the subtree
- sibling edges may be drawn through planarization (if possible) as they are at the same level between nodes placed next to each other
- downward edges are also local; they remain within the subtree
- upward edges are somewhat problematic, but even these cannot have arbitrary targets; they stay within the scope corridor as visualized in Fig. 1 (shaded area); sharing can be used to summarize multiple edges going up to the same node

Since these properties impose strong restrictions on the allowable edges between any pair of nodes, it should be possible to take advantage of them for finding an appropriate layout.

4 Measuring Dominance Dependency Graphs

Before we actually attempt to implement an algorithm that leverages the properties stated in the previous section, we should gather data on how typical large dominance trees for real software systems look like. For instance, we expect that the second level (direct children of the root) tends to be very broad as opposed to all other levels. A circular layout would draw a large circle around the root. The data for these graphs would give us some hints whether a layout could work in principle before we invest time implementing it. This section describes measurements for dependency graphs and their dominance trees that are useful to that end.

The dominance tree is a rooted tree that is calculated on the dependency graph. However, as mentioned above, not all nodes of the dependency graph might be reachable from the root. Therefore, the dominance tree usually contains less nodes than the dependency graph. Additionally to the size of the graphs, the first metrics check how many nodes and edges are retained, and how many are lost (names in brackets denote the corresponding row names in Fig. 4):

- Number of nodes in the dominance tree (#Nodes)
- Share of nodes in the dependency graph that occur in the dominance tree (%Nodes)
- Number of dependency edges with source and target node within the dominance tree (#Edges)
- Share of such edges compared to all dependency edges (%Edges)
- In- and out-degree of each node; maximum (max in/out deg) and average

Several tree metrics that might be relevant for the layout can be collected on the dominance tree:

- Depth (Depth)
- Nodes per level: Height profile, minimum, maximum (Width), average
- Leaves per level: Height profile, minimum, maximum, average, share of leaves (%Leaves/L)

All following measurements relate to the nodes and edges reachable from the root in the dependency graph. We look at the degree of nodes and number of dependency edges with respect to the level of the nodes in the dominance tree:

- Number of dependency edges from each level in the dominance tree to each other level
- Number of dependency edges crossing n levels
- Number of dependency edges from a certain level to a lower (Down), the same (Same), or a higher level (Up)
- Absolute number of levels that a upward edge spans on average (LUE), and this number of levels compared to the tree's depth (Rel.LUE)
- Level with the highest number of incoming upward edges (MCL)
- Percentage of upward edges that lead into this level (%EMCL)
- Percentage of nodes in this level (%NMCL)

As described earlier, the upward dependency edges always follow the path to the root in the dominance tree with an optional side step to a sibling on the target level. Let us call the step from one level to the next or to the side a *segment*. Certain parts of a path can be shared by multiple edges. So it is possible that edges with the same target node will share the last segments on that path, or edges with the same source node may share the first segments. Those edges could be drawn as a hyperedge and this way save space in the drawing. For both cases, we calculated the number of these shared segments and the number of segments without sharing. In detail, we measure:

- Number of parallel segments between each pair of nodes with/without sharing: minimum, maximum, average (PAS for sharing)
- Overall number of segments with and without sharing
- Percentage of segments that is needed with sharing compared to no sharing (Sharing)

Subgraphs closed under outgoing dependency edges in the projected dependency graph could also be interesting for drawing. These subgraphs could be drawn anywhere on the plane, since they have only one node that is connected to the remaining graph. Therefore, for each node in the dominance tree, we check the subtree of that node for closedness with respect to outgoing dependency edges.

5 Case Study

To get an impression about the concrete properties of dependency graphs that are relevant in software reverse engineering, we investigated the call graphs of several larger software systems. In particular, we measured three C programs (gcc's cc1, sdcc, and xfig), four C++ programs (the server of BZFlag, doxygen, jikes and mozilla), and four Java programs (AntLR,

ArgoUML, DiaGen and Maven). The C and C++ call graph creation was supported by the Das points-to analysis[3] (our pointer analysis does not work for Java yet). Therefore, the call graphs for the Java systems are quite incomplete. Fig. 4 shows the results of the measurements that were described in the previous section.

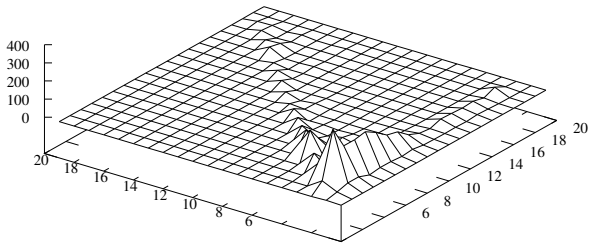


Figure 3. Mountain chains for Maven.

General tree layout If the dominance tree of the call graph is deep and narrow, this means that the system has a lot of locality and can thus be understood more easily. So this could be an indication for good quality. Our Java subject systems have that property. On the other hand, as mentioned above, many call edges are missing there, which probably makes the call graph much less complex. The dominance tree for sdcc marks the other extreme: It is shallow and wide.

For drawing, we would be interested in a tree that is not too wide and has a good depth. Even with a great width, this would not be too much of a problem if most of the nodes on the same level would be leaves. For our subjects, about 70-80% of the nodes in each level were in fact leaves of the dominance tree.

Call edges While the ratio of upward edges is independent of the programming language, the Java programs expose a higher amount (∼15%) of downward calls at the expense of the calls at the same layer. The length of an upward edge is about a fifth of the tree height (Rel.LUE), and those edges cross between 1.6 and 4.4 layers on average (LUE). This means that most of those upward edges do not cross too many layers, so drawing the nodes on a dominance tree base would place them quite close to each other (if neighboring layers are located close to each other).

Sharing The results show that the sharing based on the same source node will yield a reduction to 55% of the normal amount of segments on average (SD-Sharing). With destination based sharing it is about 45% (SS-Sharing). After sharing, the average number of lines which would have to be drawn in parallel for each segment is 3.6 for the same-source case and 2.9 for the other. However, there are still segments which contain a much larger number (up to several hundred) which can be considered not drawable. So sharing can help reduce the amount of lines necessary to draw all the edges, but it is still not enough.

Mountain Chains Fig. 3 shows the number of calls from a source to a target level. Visible here are two constructs which are similar to mountain chains. The diagonal one shows the calls down to the next level (down edges). The second mountain chain represents the tool functions. These are located on one of the top levels (often the second one after root, but sometimes also deeper; the most called layer MCL) and get called from all other places. This layer is called by two thirds of all calls for the programs we examined (%EMCL). Only Jikes shows a much lower percentage. The reason is that for Jikes, layer 4 is the target for 28.9% of all incoming backward edges, so that layer 3 and 4 together provide 62.5% (a normal value). On average, the most called layer contains a fifth of all nodes.

Closedness In the examined graphs, only two extreme kinds of closedness were found: The root node has no outgoing upward edges, and certain small subtrees are closed. These subtrees usually contain no or in very rare cases only up to 9 nodes with 10 edges inside. So closedness is not a common property that could be leveraged.

6 Conclusions

In this paper, we have recalled the basics of dominance analysis and explored employing it for visualizing dependency graphs. We have applied several measures to dominance trees derived from different software systems that should be of interest for that.

Some of the dominance tree's properties are similar across programming languages. These include the average percentage of leaves per layer and the ratio of upward edges towards all edges within the tree. These form the primary basis for general assumptions about a dominance tree. The properties we found have encouraged our initial assumption that the dominance tree could be helpful for visualization.

The proposed technique may be useful in several aspects. It might improve graph layout algorithms by making layout calculations easier, since trees are easier to draw in general. The resulting graphs could make the dominance relation explicit and this way help the viewer to better understand the graph. Also, the aesthetics of the drawing might be improved.

We think there are several possible applications of dominance analysis for drawing a dependency graph.

[3] except cc1

	cc1	sdcc	xfig	BZFl.	doxyg.	jikes	Moz.	Argo	DiaG.	Antlr	Maven
#Nodes	4,149	2,787	1,766	2,044	3,571	2,192	464	1,063	820	366	1,168
%Nodes	1.5%	95.7%	76.0%	16.5%	19.3%	40.0%	11.9%	2.1%	9.0%	12.2%	3.5%
#Edges	20,888	22,653	8,119	7,111	20,680	7,796	761	2,241	2,076	1,031	2,766
%Edges	67.1%	97.0%	68.3%	60.7%	49.0%	37.2%	29.5%	2.0%	11.6%	9.3%	4.0%
Depth	17	9	10	11	13	15	9	15	20	12	20
Width	787	1,106	649	613	789	366	150	213	160	84	242
%Leaves/L	73%	81%	81%	75%	80%	75%	74%	70%	35%	70%	73%
max in deg	705	419	138	205	542	59	24	64	141	44	120
max out deg	183	239	191	195	238	114	60	81	42	30	47
Down	12%	9%	12%	21%	12%	17%	47%	35%	24%	27%	32%
Same	22%	34%	38%	27%	65%	31%	30%	16%	16%	9%	17%
Up	66%	57%	50%	52%	23%	52%	23%	49%	60%	64%	51%
LUE	4.3	2.0	1.6	2.4	2.3	2.6	2.0	3.1	2.8	3.5	4.4
Rel.LUE	25%	22%	16%	22%	18%	17%	22%	21%	14%	29%	22%
MCL	3	1	1	1	1	3	1	1	1	2	2
% EMCL	64%	84%	96%	53%	78%	34%	64%	75%	63%	60%	78%
% NMCL	19%	22%	37%	30%	19%	14%	16%	20%	16%	23%	21%
SD-Sharing	35%	60%	63%	54%	50%	55%	83%	68%	55%	40%	55%
SD-PAS	4.3	3.7	3.2	3.2	5.0	3.5	2.3	3.2	3.5	3.4	3.7
SS-Sharing	37%	43%	51%	43%	36%	42%	61%	47%	44%	39%	45%
SS-PAS	4.6	2.6	2.5	2.6	3.5	2.7	1.7	2.2	2.8	3.3	3.0

Figure 4. Metrics results for the subject systems. See Section 4 for details (SD=Same Destination sharing, SS=Same Source sharing).

We suggest two topics which should be examined further:

Tree-based drawing When drawing the dependency graph based on the tree layout, the set of possible target nodes is quite limited for a given source node. It should be possible to take advantage of this restriction when inserting the additional edges. Also, a special crossing minimization algorithm might utilize this property. As shown in the case studies, the number of segments that have to be drawn can be significantly reduced by joining the edges which share source and/or destination node. This could help to make the graph more readable.

Hierarchical drawing The analysis can be used to remove nodes in the initial view of a "flat" display of the dependency graph. All nodes below a certain level would be hidden. However, the hidden edges have to be lifted into the visible part of the graph and will create more edges there. This technique could be used in an interactive graph exploration tool.

References

[1] Bauhaus. `http://www.bauhaus-stuttgart.de/`, June 2005.

[2] E. Burd and M. Munro. Evaluating the use of dominance trees for c and cobol. In *ICSM*, pages 401–410. IEEE Press, 1999.

[3] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems and Software*, 28:117–127, 1995.

[4] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice and Experiences*, 4:1–10, 2001.

[5] J.-F. Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *ICSM*. IEEE Press, 1997.

[6] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Vis. Comput. Graph.*, 6(1):24–43, 2000.

[7] R. Koschke. Software visualization in software maintenance, reverse engineering, and reengineering: A research survey. *Journal Software Maintenance and Evolution*, 15(2):87–109, March/April 2003.

[8] H. A. Müller and K. Klashinsky. Rigi—a system for programming-in-the-large. In *ICSE*, pages 80–86. ACM Press, 1985.

[9] M.-A. Storey. Shrimp views: an interactive environment for exploring multiple hierarchical views of a java program. In *ICSE 2001 Workshop on Software Visualization*. ACM Press, 2001.