Textual vs. Graphical Visualization of Fine-Grained Dependences

Extended Abstract

Jens Krinke
FernUniversität in Hagen

1 Introduction

A slice extracts those statements from a program that potentially have an influence onto a specific statement of interest that is the slicing criterion. Slicing has found its way into various applications. It is mostly used in the area of software maintenance and reengineering, e.g. in testing, impact analysis, and cohesion measurement.

One of the main slicing approaches uses reachability analysis in program dependence graphs (PDGs). Program dependence graphs mainly consist of nodes representing the statements of a program, and control and data dependence edges:

- Control dependence between two statement nodes exists if one statement controls the execution of the other (e.g. through if- or while-statements).
- Data dependence between two statement nodes exists if a definition of a variable at one statement might reach the usage of the same variable at another statement.

For the interprocedural variants IPDG and SDG the graphs are extended with additional interprocedural edges (which are not discussed here). The (*backward*) *slice* $S(n)$ of an IPDG at node n consists of all nodes on which n (transitively) depends via an interprocedurally realizable path.

The program dependence graph itself and the computed slices within the program dependence graph are results that should be presented to the user if not used in following analyses. As graphical presentations are often more intuitive than textual ones, a graphical visualization of PDGs is desirable.

2 Visualization of PDGs

Layout of graphs is a widely explored research field with many general solutions available in graph drawing tools. We evaluated some of these tools (*daVinci*, *VCG* and *dot*) to lay out PDGs. Our experience with these tools to layout PDGs has been disappointing. The resulting layouts were visually appealing but unusable, as it was not possible to comprehend the graph. The reason is that the viewer has no cognitive mapping back to the source code, which is the representation he is used to. The user expects a representation that is either similar to the abstract syntax tree (as a presentation of the syntactical structure), or a control-flow-graph like presentation.

Because this general approach to layout PDGs had failed, a declarative approach has been implemented. It is based on the following observations:

1. The control-dependence subgraph is similar to the structure of the abstract syntax tree.
2. Most edges in a PDG are data dependence edges. Usually, a node with a variable definition has more than one outgoing data dependence edge.

The first observation leads to the requirement to have a tree-like layout of the control dependence subgraph with the additional requirement that the order of the nodes in a hierarchy level should be the same as the order of the corresponding statements in the source code. The second observation leads to an approach where the data dependence edges should be added to the resulting layout without modifying it. As most data dependence edges would now cross large parts of the graph, a Manhattan layout is adequate. This enables an orthogonal layout of edges with fixed start and end points. This approach has been implemented in a tool that visualizes system dependence graphs. Starting from a graphical representation of the call graph, the user can select procedures and visualize their PDGs. Through selection of nodes, slices can be calculated and are visualized through inverted nodes in the PDGs laid out.

Experience with the presented tool shows that the layout is very comprehensible for medium sized procedures and the user easily keeps a cognitive map from the structure of the graph to the source code and vice versa. This mapping is supported by the possibility to switch between a textual visualization of the source code and the graphical layout of the current procedure. Sets of nodes marked in the graph can be highlighted in the source code and marked regions in the source code can be highlighted in the graph. Together with additional navigational aids, it is easy to see what statements influence which other statements and how.

However, experience has shown that the graphical visualization is still too complex. For larger procedures the number of nodes and edges is too high, and it takes very long to follow edges across multiple pages by scrolling.

The presented graphical visualization has been found to be far too complex for large programs and non-intuitive for visualization of slices. Therefore the graphical visualization has been extended with a visualization in source code. This causes a non-trivial projection of nodes onto source code, because of the fine-grained structure of the dependences between statements.

3 Visualization of Locality

Independent of visualization, one of the problems in understanding a slice is to decide why a specific statement is included in that slice and how strong the influence of that statement is onto the slicing criterion. A slice cannot answer these questions as it does not contain any qualitative information. Probably the most important attribute is *locality*. Users are more interested in facts that are near the current point of interest than on those far away. A simple but very useful aid is to provide the user with navigation along the dependences: For a selected statement, show all statements that are directly dependent (or vice versa).

A more general approach to accomplish locality in slicing is to limit the length of a path between the criterion and the reached statement. Using paths in program dependence graphs has instead of paths in control flow graphs has an advantage. A statement that has a direct influence on the criterion will be reached by a path with length one, independent of textual or control flow distance.

Distance-limited slices cannot simply be visualized with the techniques presented in the previous section without any modification. Another possibility is to indicate the distances from the (slicing) criterion for any node in the (possibly distance-limited) slice. The textual visualization from the previous section is therefore modified not only to highlight the nodes in the textual representation, but also to give any source code fragment a color that represents the distance of the equivalent nodes to the criterion. The slicing algorithm needs not to be changed in order to accommodate the distance computation—it is sufficient to remember the distance of a node during breadth-first search.

4 Abstract Visualization

For large-scale program understanding the presented visualization techniques are not very helpful. If an unknown program is analyzed, the very detailed information of program dependences and slices is overwhelming, and a much less detailed information is needed. The user who tries to understand the program will start with variables and procedures and not with statements. To understand a previously unknown program, it is helpful to identify the 'hot' procedures and global variables—the procedures and variables with the highest impact on the system.

This section will show how slicing and chopping can help to visualize programs in a more abstract way, illustrating relations between variables or procedures. *Chopping* reveals the statements involved in a transitive dependence from one specific statement (the source criterion) to another (the target criterion). A chop for a chopping criterion (s, t) is the set of nodes that are part of an influence of the (source) node s onto the (target) node t.

It is possible to define slices for variables or procedures informally:

1. A (backward) slice for a criterion variable v is the set of statements (or nodes in the PDG) which may influence variable v at some point in the program.
2. A (backward) slice for a criterion procedure P is the set of statements (or nodes in the PDG) which may influence a statement of P.

These definitions can be adapted to the other slicing and chopping variants, including the adaption of the needed algorithms. It will not be presented here, as it is straightforward.

As previously noted, it is helpful to identify the 'hot' procedures and global variables. However, to identify them, we have to measure the procedures' and variables' impact on the system. A simple measurement is to compute slices for every procedure or global variable and record the size of the computed slices. However, this might be too simple. A slightly better approach is to compute chops between the procedures or variables. A visualization tool has been implemented that computes a $n \times n$ matrix for n procedures or variables, where every element $n_{i,j}$ of the matrix is the size of a chop from the procedure or variable n_j to n_i. The matrix is painted using a color for every entry, corresponding to the size—the bigger, the darker. With this tool, it is easy to get an overall impression of the software to analyze. Important procedures or global variables can be identified on first sight and their relationship can be studied. Doing this as a preparing stage aids in later, more thorough investigations with traditional slicing visualizations like the ones presented in the previous sections.

5 Conclusions

Despite the widespread use of graphical visualization in software maintenance and reverse engineering, our and other's experiences for graphical visualization of program dependence and program slices are different. For tasks related to large-scale understanding graphical visualization has proven to be successful. The main reason is that the number of nodes (or objects) to be visualized is kept very low by clustering techniques. Tasks related to understanding dependences in detail (like program dependences and slices) suffer from the sheer amount of data to be visualized. The various experiences show that graphical visualization has more disadvantages than advantages in this area.

The visualization of slices in textual form has shown to be much more effective, because the programmer is accustomed to representations similar to source code. However, slices are still hard to understand due to the loss of locality. Distance-limited slicing and its visualization can help, because it limits the distance of the influence to the current point of interest. The visualization of the distance shows immediately how important a statement is for the current influence.

For large-scale program understanding none of the detailed slicing visualizations are helpful. The presented approach to visualize the influence range of variables and procedures by visualizing the size of chops can help the user to identify "hot spots" of the program very fast.