Visualization of Program Dependence and Slices

Jens Krinke
FernUniversität in Hagen, Germany
Jens.Krinke@FernUni-Hagen.de

Abstract

The program dependence graph (PDG) itself and the computed slices within the program dependence graph are results that should be presented to the user in a comprehensible form, if not used in subsequent analyses. A graphical presentation would be preferred as it is usually more intuitive than textual ones. This work describes how a layout for the PDGs can be generated to enable an appealing presentation. However, experience shows that the graphical presentation is less helpful than expected and a textual presentation is superior. Therefore this work contains an approach to textually present slices of PDGs in source code. The innovation of this approach is the fine-grained visualization of arbitrary node sets based on tokens and not on complete lines like in other approaches.

Furthermore, a major obstacle in visualization and comprehension of slices is the loss of locality. Thus, this work presents a simple, yet effective, approach to limit the range of a slice. This approach enables a visualization of slices where the local effects stand out against the more global effects. A second, more sophisticated approach visualizes the influence range of chops for variables and procedures. This enables a visualization of the impact of procedures and variables on the complete system.

1. Introduction

A slice extracts those statements from a program that potentially have an influence on a specific statement of interest which is the slicing criterion. Slicing has found its way into various applications. Nowadays it is probably mostly used in the area of software maintenance and reengineering [15], as in testing [17, 6, 5, 7], impact analysis [15] and cohesion measurement [24].

Originally, slicing was defined by Weiser in 1979; he presented an approach to compute slices based on iterative data flow analysis [31, 32]. The other main approach to slicing uses reachability analysis in program dependence graphs (PDGs) [11]. Program dependence graphs mainly consist of nodes representing the statements of a program as well as 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.

A slice can now simply be computed in three steps: Map the slicing criterion on a node, find all backward reachable nodes, and map the reached nodes back on the statements.

For the interprocedural variants IPDG and SDG the graphs are extended with additional interprocedural edges [18] (which are not discussed here). Our work is based on fine-grained dependence graphs, where the nodes are representing operands and operations (and thus a subset of the tokens) instead of statements.

The (*backward*) *slice* $S(n)$ of an IPDG $G = (N, E)$ at node $n \in N$ consists of all nodes on which n (transitively) depends via an interprocedurally realizable path:

$$S(n) = \{m \in N \mid m \rightarrow^{\star}_{\mathsf{R}} n\}$$

Here, $m \rightarrow^{\star}_{\mathsf{R}} n$ denotes that there exists an interprocedurally realizable path from m to n. A *forward* slice consists of all nodes that (transitively) depend on n.

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 subsequent analyses. As graphical presentations are often more intuitive than textual ones, a graphical visualization of PDGs is desirable. The next section describes how a layout for the PDGs can be generated to enable an appealing presentation. The presented visualizations were used with different users. The project started together with a measurement system certifying authority, where slicing was used to detect illegal influences on the measured values. The project members of the authority formed the first groups; they understood the concepts of program dependence and slicing. The other group consisted of researchers and students doing slicing research not necessarily related to the project. Both groups experience shows that the graphical presentation is less helpful than expected and a textual presentation is superior. Therefore Section 3 contains an approach to present slices in (fine-grained) PDGs textually in source code.

Furthermore, a major obstacle in visualization and comprehension of slices is the loss of locality. Thus, Section 4 presents a simple, yet effective, approach to limit the range of a slice. A second, more sophisticated approach in Section 5 visualizes the influence range of chops for variables and procedures. A discussion of related work and conclusions follow.

2. Graphical Visualization of PDGs

Layout of graphs is a widely explored research field with many general solutions available in many graph drawing tools. Some of these tools have been tested to lay out PDGs. The primary goal was to provide an aid for the developers of the slicing system to debug and verify generated PDGs. The tools we tried have been:

daVinci a visualization system for generating high-quality drawings of directed graphs [12].

VCG (Visualization of Compiler Graphs) is targeted at the visualization of graphs that typically occur as data structures in programs [26].

dot is a widely-used tool to create hierarchical layouts of directed graphs [21, 16].

The graphical representations generated by these tools have only been used by the slicing specialists, and their experience has been disillusioning. 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 or she is used to. The user (the slicing specialist) 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.

In a second experiment the layout was influenced as much as possible to generate a presentation that enables the viewer to map the graph structure to the syntactical structure based on the control-dependence subgraph. The control dependence subgraph is tree-like, and in structured programs it resembles the abstract syntax tree. The possibilities of influencing the layout were quite different in the evaluated tools, where *dot* had the greatest flexibility. However, it was not possible to manipulate the layout in a way that generated comprehensible presentations. The main obstacles have been:

1. The order of nodes was completely different than the order of the corresponding statements and their implicit control flow.
2. Nodes that were near in the laid out graph often had very distant statements in the source code.
3. It was hard to follow the data dependence edges.

These obstacles made the general tools practically unusable for visualization of program dependence graphs.

2.1. A Declarative Approach to Lay out PDGs

As the general algorithmic approach to lay out PDGs had failed, a declarative approach has been implemented. The main goal of this approach is to eliminate the three obstacles mentioned before. It is based on the following observations about the general properties of a PDG:

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 equivalent statements in the source code. This is essential as the order of statements implies control flow, which is not explicitly visualized in the layout (comprehension of the layout is much easier if the nodes' statements are executed left-to-right). 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.

2.1.1. Layout of the Control Dependence Graph

Instead of a specialized tree layout, an available implementation of the Sugiyama algorithm [30] has been reused, consisting of three phases:

1. The nodes are arranged into a vertical hierarchy based on a spanning tree of the graph. Also, the number of levels crossed by edges is minimized.
2. Nodes in a horizontal level of the hierarchy are ordered to minimize the number of edge crossings.
3. The coordinates of the nodes are calculated such that long edges are as straight as possible.

Because the control dependence graph is mainly a tree, phase one is simple and very fast. Phase two has been replaced completely as the order of nodes is defined by the statement order in the source code and is not allowed to change. In Phase three the original algorithm has been extended with a “rubber-band” improvement presented in [26].

2.1.2. Layout of Data Dependence Edges

The layout of data dependence edges is basically a routing between fixed start and end points. As most edges in a PDG are data dependence edges, the routing must be fast and efficient. Based on the observations at the beginning of this section, the routing is done according to the following principles:

1. The route of an edge is separated into three parts:
    - a vertical segment between the start node and the level above the end node,
    - a horizontal segment to the position of the end node, and
    - a vertical end segment to the end node.
2. Edges leaving the same node share the same first segment.

The layout of the three segments is done independently: The starting vertical segment is laid out straight if a node that would be crossed can be pushed aside. If this is not possible, the segment is split to circumvent the node. The horizontal segment is laid out with a sweep-line algorithm to minimize the space routes take passing a level. The third segment is routed to its entry point into the end node.

2.1.3. Presentation of System Dependence Graphs

The presented approach to laying out PDGs has been implemented in a tool that visualizes system dependence graphs [9]. Starting from a graphical presentation 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 laid out PDGs. Procedure crossing edges are visualized in the PDGs with anchors, indicating that the edge leaves the current procedure.

A visualization can be seen in Figure 1, where a slice is visualized in the dependence graph and source code through highlighting. Actually, the intersection of a forward and a backward slice is visualized there. The backward slice contains all statements that can influence the value of variable ‘`u_kg`’ in line 34, corresponding to node 111, and the forward slice contains all the statements that are influenced by variable ‘`p_cd`’ in line 9. The intersection[1] shows all statements that are involved in an influence of ‘`p_cd`’ on ‘`u_kg`’.

2.1.4. Navigation

The user interface for the visualized graph contains extensive navigational aids:

- Nodes and edges can be searched for by their attributes.
- Edges can be followed forward or backward.
- Selection of the anchors of a procedure crossing edge switches to the graph of the other procedure.
- A set of nodes can be expanded with all nodes reachable by traversing one edge.
- The visualization can be focused on each node of a node set by stepping through the set.
- Node sets can be saved and restored.
- Two node sets can be combined to a new node set by set operations.
- Node sets can be “filtered” through external tools; slicing and chopping is implemented that way.
- To compress the visualized graph, node sets can be folded to a single node.

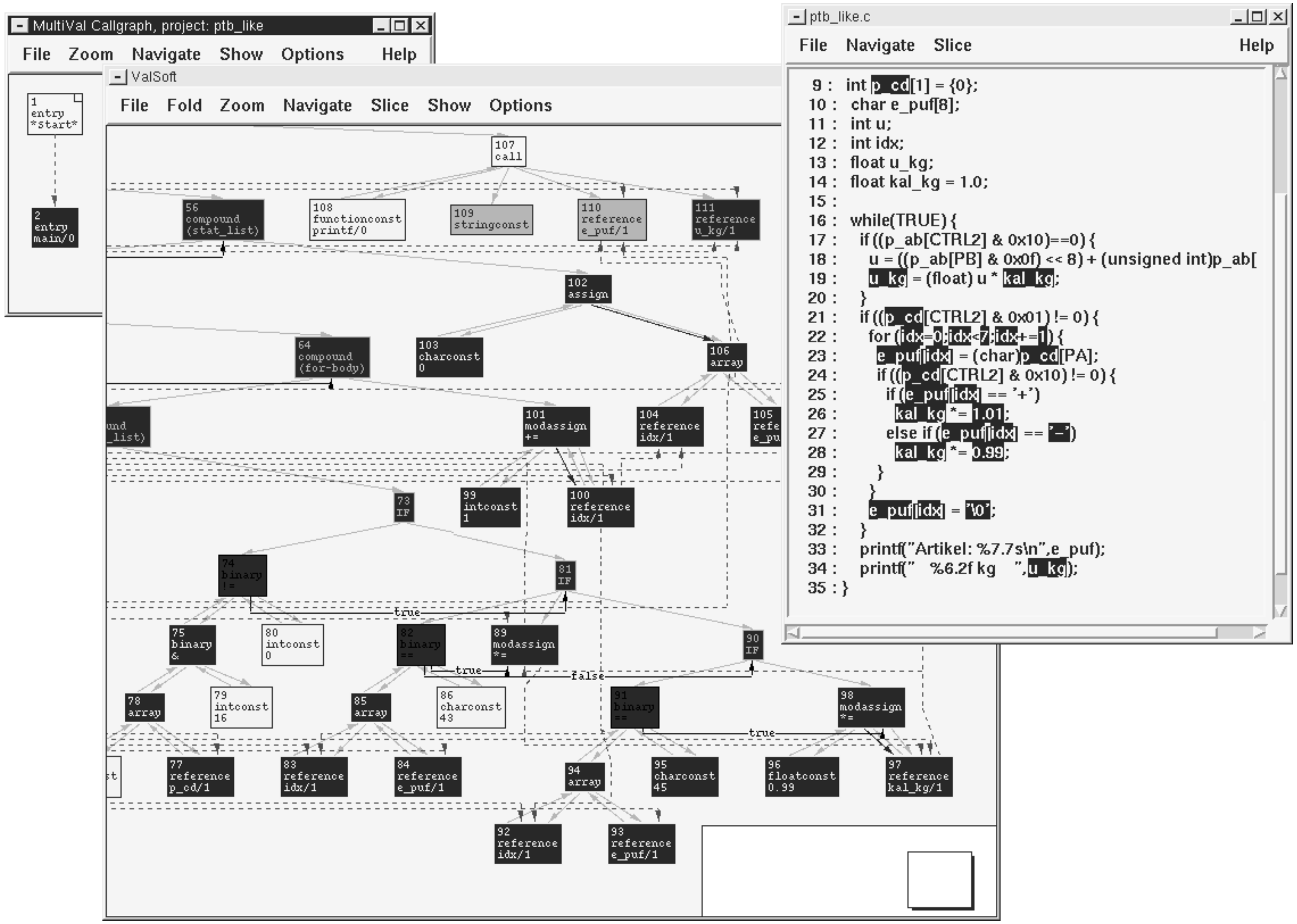
As discussed later, the user interface includes a textual visualization of the source code. Most of the navigational aids are also present there.

2.2. Evaluation

The presented tool is used by all researchers and students involved in slicing research. Their experiences show that the layout is very comprehensible up to *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 of switching to a source code visualization of the current procedure and back: 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 (see Figure 1 for an example). Together with the navigational aids, it is easy to see what statements influence which other statements and how. This is an important advancement to general purpose graph visualization tools, which are not able to provide the user with a comprehensible representation even for small procedures.

However, experience has shown that the graphical visualization is still too complex for *large* procedures. There, the number of nodes and edges is too big and it takes very long to follow edges across multiple pages by scrolling. Additionally, the users of the certifying authority were not able to use the graphical visualization. Although they understood the concepts of slicing and dependences, they were not able to use the tool to search for the reasons of a discovered illegal influence (i.e. why a certain statement is in a slice). We therefore reverted to textual visualization, which is presented next.

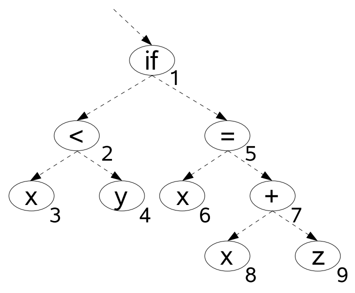
1 Actually not the intersection is computed, but a chop [20], which is different to the intersection in the interprocedural case.



```
int p_cd[1] = {0};
 char e_puf[8];
 int u;
 int idx;
 float u_kg;
 float kal_kg = 1.0;

 while(TRUE) {
   if ((p_ab[CTRL2] & 0x10)==0) {
     u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[
     u_kg = (float) u * kal_kg;
   }
   if ((p_cd[CTRL2] & 0x01) != 0) {
     for (idx=0;idx<7;idx+=1) {
       e_puf[idx] = (char)p_cd[PA];
       if ((p_cd[CTRL2] & 0x10) != 0) {
         if (e_puf[idx] == '+')
           kal_kg *= 1.01;
         else if (e_puf[idx] == '-')
           kal_kg *= 0.99;
       }
     }
     e_puf[idx] = '\0';
   }
   printf("Artikel: %7.7s\n",e_puf);
   printf("   %6.2f kg    ",u_kg);
}
```

Figure 1. The graphical user interface



$1:$	$[1-15]$	$1:$	$[1-2],[15-15]$
$2:$	$[3-7]$	$2:$	$[3-3],[5-5],[7-7]$
$3:$	$[4-4]$	$3:$	$[4-4]$
$4:$	$[6-6]$	$4:$	$[6-6]$
$5:$	$[9-14]$	$5:$	$[10-10],[14-14]$
$6:$	$[9-9]$	$6:$	$[9-9]$
$7:$	$[11-13]$	$7:$	$[12-12]$
$8:$	$[11-11]$	$8:$	$[11-11]$
$9:$	$[13-13]$	$9:$	$[13-13]$

Figure 2. A small code fragment with position intervals and after transformation

3. Textual Visualization of Slices

The graphical visualization presented in the previous section has been found to be overly complex for large programs and non-intuitive for visualization of slices. Therefore the graphical visualization has been extended with a visualization in source code. Because of the fine-grained structure, this causes a non-trivial projection of nodes on source code. The technique presented in this section not only visualizes slices (and chops [20]) in source code, but any set of nodes.

Textual visualization of source code is essential. Even with an accompanying graphical visualization, the user needs the reference to the source code to explain the origins of dependences. Most current slicing tools use a line-by-line visualization: if any part of a source code line is included in the visualized slice, the complete line is highlighted. This might be sufficient for traditional slicing, however, more advanced techniques like chopping [20] need a more fine-grained visualization. For example, a chop may contain only parts of an expression and a fine-grained visualization provides more precise information. Figure 1, line 19, contains such a situation: The detected influence of variable ‘`p_cd`’, line 9, on ‘`u_kg`’, line 34, involves variables ‘`u_kg`’ and ‘`kal_kg`’ in line 19, but not variable ‘`u`’.

The source code is represented as a continuous sequence of characters, such that any piece of source code can be represented as an interval in that sequence. Such an interval is described by a file/row/column position for start and end. During parsing while constructing the abstract syntax tree, every node is attributed with an interval. During analysis, the nodes of the abstract syntax tree are transformed into nodes in the program dependence graph, which still have the source code interval attribute (except for nodes that have no correspondence in the source code or the abstract syntax tree).

Consider the following example fragment with its program dependence graph shown in Figure 2, left column:

```
if (x < y) {
  x = x + z;
}
```

This program is represented as a sequence of characters, each character having a position, as shown in the following table (whitespace is ignored):

i	f	(	x	<	y	)	{	x	=	x	+	z	;	}
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

During transformation of this fragment into its program dependence graph the nodes are attributed with the position interval as shown in Figure 2, middle column.

If the visualization would just highlight the intervals of the nodes, the result would be disastrous: any visualization of a set that includes node 1 would highlight the complete fragment. The problem here is the nesting of intervals: an interval $r = [x_r - y_r]$ is nested in an interval $q = [x_q - y_q]$, written as $r \subseteq q$, if $x_q < x_r < y_r < y_q$. The intervals generated during construction of the abstract syntax tree have two properties (because of the tree structure):

1. All intervals are properly nested:

$$\forall r, q : r \subseteq q \vee q \subseteq r$$

2. All intervals are unique:

$$\forall r, q, r \neq q : x_r \neq x_q \vee y_r \neq y_q$$

It follows $\forall r, q : r \subset q \vee q \subset r$.

In order to get a comprehensible visualization, any position should only be highlighted if the smallest enclosing interval belongs to a node included in the highlighted set. An interval r is the smallest enclosing interval of a position x, if there is no interval q such that q encloses x ($x \in q$) and r ($r \supset q$). Therefore, the interval attribute of the nodes is changed to a set of attributes: If a node has an interval q that encloses an interval r of a different node, the interval r is removed by splitting the original interval q: Let $r = [x_r - y_r]$ be nested in interval $q = [x_q - y_q]$, the new interval is split into two new intervals $q_1 = [x_q - x_r[$ and $q_2 =]y_r - y_q]$.

If this transformation is applied thoroughly, every interval will be unique.

The resulting intervals for the example are shown in Figure 2, right column.

The nodes are now mapped to non-overlapping intervals. To highlight any set of nodes, the sets of intervals of the nodes are joined and only the intervals in the resulting set are highlighted.

The fragment below shows the visualization of a backward slice for node 9, which consists of the node set $\{1, 2, 3, 4, 5, 7, 9\}$:

```
if (x < y) {
  x = x + z;
}
```

The next example shows the node set $\{1, 5, 6\}$ highlighted.

```
if (x < y) {
  x = x + z;
}
```

We have presented only the basic visualization techniques. These techniques enable a fine-grained visualization of slices and arbitrary node sets in the source code. Most earlier approaches only visualize based on lines of source code, which is not sufficient for more advanced slicing techniques like chopping.

4. Distance-Limited Slices

Independent of visualization, one of the problems in understanding a slice for a criterion is to decide why a specific statement is included in that slice and how strong the influence of that statement is on the 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). Such navigation is central to the VALSOFT system [23] or to CodeSurfer [1]. However, such navigational aids don't offer an instant insight into the local effects of a statement.

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 an advantage over paths in control flow graphs: a statement having a direct influence on the criterion will be reached by a path with the length one, independent of the textual or control flow distance.

The *distance-limited* slice $S(c, k)$ of a PDG for the slicing criterion node c consists of all nodes on which c (transitively) depends via a realizable path consisting of at most k

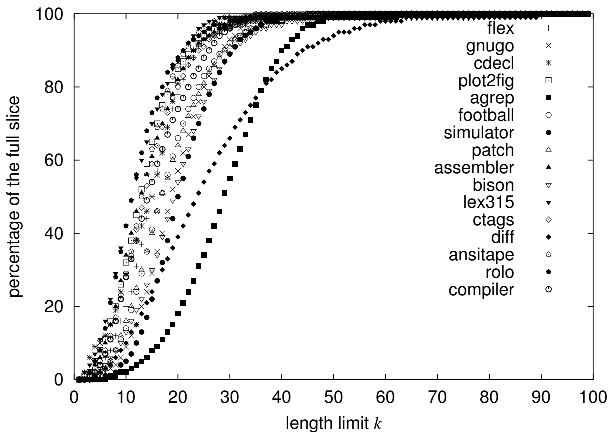


Figure 3. Evaluation of length-limited slicing

edges:

$$S(c, k) \quad = \quad \{m \mid p = m \rightarrow^{\star}_{\mathsf{R}} c \\ \wedge\ p = \langle n_1, \ldots, n_l \rangle \wedge l < k\}$$

A node m is said to have a distance $l = d(m, n)$ from a node n, if a realizable path from m to n consisting of l edges exists and no other path with fewer edges exists:

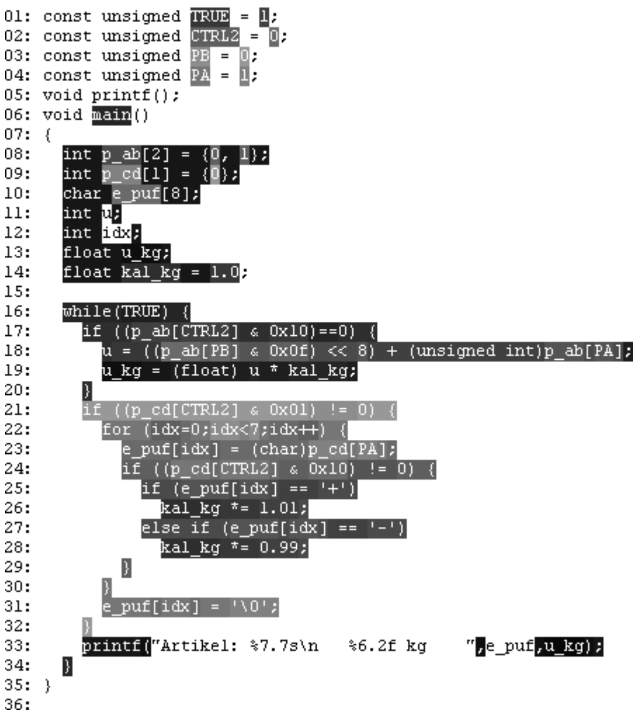
$$d(m, n) = \mathsf{min}(\{l \mid p = m \rightarrow^{\star}_{\mathsf{R}} n \wedge p = \langle n_1, \ldots, n_l \rangle\})$$

An efficient distance-limited slicing algorithm is a modified version of the interprocedural slicing algorithm [18, 22]. To omit a priority queue sorted by the actual distance, a breadth-first search is done where the worklist is implicitly sorted.

Figure 3 shows an evaluation: For a series of test cases, the average size of 1000 length limited slices has been computed, where the length limit ranges from 1 to 100 (x-axis). The y-axis shows the reached percentage of the full slices. This evaluation shows that length limited slices behave quite similar and independent of the analyzed program. Details of the test cases can be found in [22].

A more fine-grained approach is to replace the number of traversed edges by summarizing *distances* that have been assigned to the edges. Such distances can be used to give different classes of edges different weights. For example, a node that is reachable by a data dependence edge might be considered nearer than a node that is reachable by a summary edge. If the worklist is replaced by a priority queue sorted by the current sum of distances, the previous algorithm is able to compute such distance-limited slices.

Distance-limited slices can be visualized with the techniques presented in the previous sections without any modification. Another possibility is to illustrate the distances from the (slicing) criterion for any node in the (possibly



```
01: const unsigned TRUE = 1;
02: const unsigned CTRL2 = 0;
03: const unsigned PB = 0;
04: const unsigned PA = 1;
05: void printf();
06: void main()
07: {
08:    int p_ab[2] = {0, 1};
09:    int p_cd[1] = {0};
10:    char e_puf[8];
11:    int u;
12:    int idx;
13:    float u_kg;
14:    float kal_kg = 1.0;
15:
16:    while(TRUE) {
17:       if ((p_ab[CTRL2] & 0x10)==0) {
18:          u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
19:          u_kg = (float) u * kal_kg;
20:       }
21:       if ((p_cd[CTRL2] & 0x01) != 0) {
22:          for (idx=0;idx<7;idx++) {
23:             e_puf[idx] = (char)p_cd[PA];
24:             if ((p_cd[CTRL2] & 0x10) != 0) {
25:                if (e_puf[idx] == '+')
26:                   kal_kg *= 1.01;
27:                else if (e_puf[idx] == '-')
28:                   kal_kg *= 0.99;
29:             }
30:          }
31:          e_puf[idx] = '\0';
32:       }
33:       printf("Artikel: %7.7s\n   %6.2f kg    ",e_puf,u_kg);
34:    }
35: }
36:
```

Figure 4. Distance visualization of a slice

distance-limited) slice. The textual visualization of Section 3 is therefore modified not only to highlight the nodes in the textual representation, but to give any source code fragment a color representing the distance of the equivalent nodes to the criterion. The slicing algorithm does not need to be changed to accommodate the distance computation—it is sufficient to remember the distance of a node during breadth-first search.

Figure 4 shows an example visualization of a slice. A backward slice for variable '`u_kg`' in line 33 is displayed. Parts of the program that have a small distance to the slicing criterion are darker than those with a larger distance. With this presentation one can see that the initialization in lines 8–10 and 13–14 have a close influence on the criterion. It can also be seen that the first few lines of the loop (17–19) have a close influence and that the whole next if-statement has a varying influence, where lines 26 and 28 have the strongest effect. This visualization immediately shows why there is an influence of the variable '`kal_kg`' in lines 26 and 28 on '`u_kg`' in the criterion: The statement 19 uses '`kal_kg`' to compute a new value for '`u_kg`'. Despite the fact that statement 19 is textually far away from the criterion, the dark color shows its strong influence. Such information would not be visible in a simple slice visualization (e.g. Figure 1); the visualization of the distance guides the viewer to the important areas in the slice.

5. Abstract Visualization

For program understanding in-the-small the presented visualization techniques are very effective. However, for program understanding in-the-large they are not as helpful. If an unknown program is analyzed, the very detailed information of program dependence and slices is overwhelming and a much less detailed information is needed. The user trying 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 shows how slicing and chopping can help to visualize programs in a more abstract way, illustrating relations between variables or procedures. *Chopping* [20] 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 on the (target) node t: The *chop* $C(s, t)$ of an IPDG $G = (N, E)$ from the source criterion $s \in N$ to the target criterion $t \in N$ consists of all nodes on which node t (transitively) depends via an interprocedurally realizable path from node s to node t:

$$
\begin{aligned}
C(s,t) = \{n \in N \mid\ & p \in s \rightarrow^{\star}_{\mathsf{R}} t \\
& \wedge p = \langle n_1, \ldots, n_l \rangle \\
& \wedge \exists i : n = n_i \}
\end{aligned}
$$

5.1. Variables or Procedures as Criterion

It is possible to define slices for variables or procedures as criteria 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 of 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 adaptation of the needed algorithms. It will not be presented here, as it is straightforward.

5.2. Visualization of the Influence Range

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 and 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. Figure 5 shows such a visualization for the `ansitape` program. The columns show variables 0–34 as source criteria and the rows as target criteria. This matrix can be interpreted as follows:

- The global variables '`stdin`' (column two), '`stdout`' (3) and '`stderr`' (4) have empty chops with all other variables (light columns 2–4). This is obvious for '`stdout`' and '`stderr`' while '`stdin`' has no influence because the program only reads from tapes.
- The variable '`stdout`' (row 3) is not influenced (empty chops with stdout as target criterion), but '`stderr`' (row 4) is. The ansitape program normally writes all messages to '`stderr`' and produces no other output (except for writing to tapes).
- Row 12 has the biggest chops (and is the darkest row). This is variable '`tcb`', the tape control block, which is the main global variable of the program.

An implementation is shown in Figure 6: The three windows contain the chop matrix visualization (in this case for procedure-procedure-chops), a color scale and a window that shows the names of the procedures and their chop's size for the last chosen matrix element. With this tool, it is easy to get an overall impression of the software to analyze. Important procedures or global variables can be identified at first sight and their relationship 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.

6. Related Work

The SeeSlice slicing tool [3] includes some of the presented focusing and visualization techniques (the distance-limited slicing and visualizing distances). Files and procedures are not presented through source code but with an abstraction representing characters as single pixels. Files and procedures that are not part of computed slices are folded, such that only a small box is left. Slices highlight the pixels corresponding to contained elements.

In [4] the same problems with visualizing dependence graphs are reported and a decomposition approach is presented: Groups of nodes are collapsed into one node. The result is a hierarchy of groups, where every group is visualized independently. Three different decompositions are presented: The first decomposition is to group the nodes be-

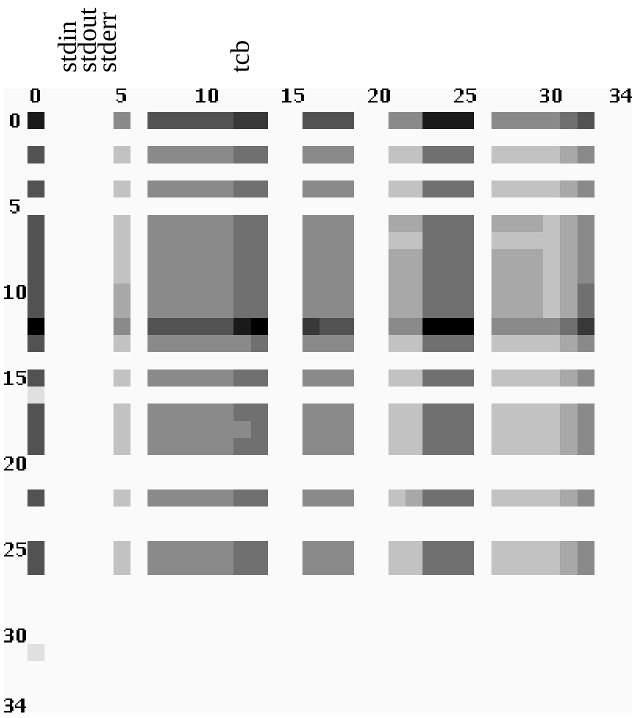
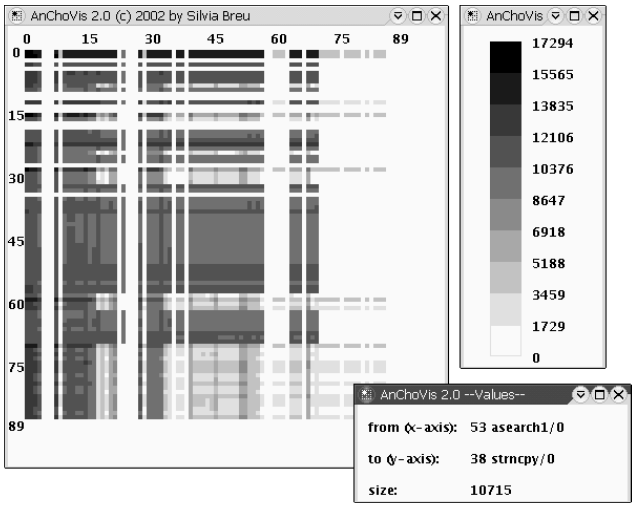


Figure 5. Visualization of chops for all global variables



from (x-axis): 53 asearch1/0
to (y-axis): 38 strncpy/0
size: 10715

Figure 6. GUI for the chop visualization

longing to the same procedure, the second is to group the nodes belonging to the same loop and the third is a combination of both. The result of the function decomposition is identical to the visualization of the call graph and the PDGs of the procedures presented in Section 2.1.3.

The CANTO environment [2] has a visualization tool PROVIS based on dot which can visualize PDGs (among other graphs). Again, problems with excessively large graphs are reported, which are omitted by only visualizing the subgraph which is reachable from a chosen node via a limited number of edges. It can be used in a stepping mode which is similar to distance-limited slicing: At each step the slice grows by considering one step of data or control dependence.

ChopShop [20, 19] is a tool to visualize slices and chops, based on highlighting text (in emacs) or laying out graphs (with dot and ghostview). It is reported that even the smallest chops result in huge graphs. Therefore, only an abstraction is visualized: normal statements (assignments) are omitted, procedure calls of the same procedure are folded into a single node and connecting edges are attributed with data dependence information.

The decomposition slice visualization of Surgeon's Assistant [14, 13] visualizes the inclusion hierarchy of decomposition slices as a graph using VCG [26].

An early system with capabilities for graphical visualization of dependence graphs is ProDAG [25]. Another system to visualize slices is [8].

Every slicing tool visualizes its results directly in the source code. However, most tools are line based, highlighting only complete lines. CodeSurfer [1] has textual visualization with highlighting parts of lines if there is more than one statement in a line. The textual visualization includes graphical elements like pop-ups for visualization and navigation along e.g. data and control dependence or calls. Such aids are necessary as a user cannot identify relevant dependences easily from source text alone. Such problems have also been identified by Ernst [10] and he suggested similar graphical aids. However, his tool, which is not restricted to highlighting complete lines, does not have such aids and offers depth-limited slicing instead (see Section 4).

Steindl's slicer for Oberon [27, 28, 29] also highlights only parts of lines, based on the individual lexical elements of the program.

CodeSurfer [1] also has a project viewer, which features a tree-like structural visualization of the SDG. This is useful for seeing "hidden" nodes, such as nodes that do not correspond to any source text.

7. Conclusions

All previous approaches to visualize slices and program dependence graphs in graph layouts used general purpose graph visualization tools. None of them were able to generate comprehensible visualizations of even small procedures. Our approach is the first with a dedicated, declarative approach to lay out dependence graphs that generates comprehensible graphs of small to medium sized procedures.

Despite the widespread use of graphical visualization in software maintenance and reverse engineering, our and others' experiences for graphical visualization of program dependence and program slices are different. For tasks related to "understanding in-the-large" 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 in-the-small" like program dependence and program slices suffer from the sheer amount of data to be visualized. Even our approach for graphical visualization has problems for large procedures. Our and others' experiences show that graphical visualization has more disadvantages than advantages in this area. Users outside slicing research just don't want to see the dependence graphs.

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, because the loss of locality. Distance-limited slicing and its visualization helps, 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 program "understanding in-the-large" none of the detailed visualizations of slices 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. It successfully generates a high-level abstraction of the procedures' and variables' impact on the complete system.

Acknowledgments. Much of the presented work has been done at Universität Passau, Germany. Frank Ehrich implemented the graphical user interface and the graph lay outing. Daniel Gmach implemented the textual visualization. Silvia Breu implemented the chop visualization tool.

References

[1] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering (CAV 2001)*, 2001.

[2] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo. Program understanding and maintenance with the CANTO environment. In *International Conference on Software Maintenance*, pages 72–81, 1997.

[3] T. Ball and S. G. Eick. Visualizing program slices. In *IEEE Symposium on Visual Languages*, pages 288–295, 1994.

[4] F. Balmas. Displaying dependence graphs: a hierarchical approach. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 261–270, 2001.

[5] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396, 1993.

[6] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 41–50, 1992.

[7] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11–12):583–594, 1998.

[8] Y. Deng, S. Kothari, and Y. Namara. Program slice browser. In *Ninth International Workshop on Program Comprehension (IWPC'01)*, pages 50–59, 2001.

[9] F. Ehrich. Entwurf und Implementierung eines Werkzeugs zur Visualisierung von Programmabhängigkeitsgraphen. Diplomarbeit, TU Braunschweig, 1996. (In German).

[10] M. D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 1994.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[12] M. Fröhlich and M. Werner. Demonstration of the interactive graph visualization system davinci. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing, DIMACS International Workshop GD'94*, volume 894 of *LNCS*. Springer, 1995.

[13] K. Gallagher and L. O'Brien. Reducing visualization complexity using decomposition slices. In *Software Visualization Workshop*, pages 113–118, 1997.

[14] K. B. Gallagher. Visual impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 52–58, 1996.

[15] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[16] E. R. Gansner, S. C. North, and K. P. Vo. DAG - A program that draws directed graphs. *Software, Practice and Experience*, 18(11):1047–1062, 1988.

[17] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, 1992.

[18] S. B. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[19] D. Jackson and E. J. Rollins. Abstraction mechanisms for pictorial slicing. In *Proceedings of the IEEE Workshop on Program Comprehension*, pages 82–88, 1994.

[20] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 2–10, 1994.

[21] E. Koutsofios and S. C. North. *Drawing graphs with dot*. Murray Hill, NJ, 1996.

[22] J. Krinke. Evaluating context-sensitive slicing and chopping. In *International Conference on Software Maintenance*, pages 22–31, 2002.

[23] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11-12):661–675, Dec. 1998.

[24] L. M. Ott and J. M. Bieman. Program slices as an abstraction for cohesion measurement. *Information and Software Technology*, 40(11-12):691–700, 1998.

[25] D. J. Richardson, T. O. O'Malley, C. T. Moore, and S. L. Aha. Developing and integrating prodag into the arcadia environment. In *Proceedings of the Fifth Symposium on Software Development Environments*, pages 109–119, 1992.

[26] G. Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD'94*, number 894 in LNCS, pages 194–205. Springer, 1995.

[27] C. Steindl. Intermodular slicing of object-oriented programs. In *International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 264–278. Springer, 1998.

[28] C. Steindl. Benefits of a data flow-aware programming environment. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, 1999.

[29] C. Steindl. *Program Slicing for Object-Oriented Programming Languages*. PhD thesis, Johannes Kepler University Linz, 1999.

[30] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, Feb. 1981.

[31] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[32] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.