Slicing, Chopping, and Path Conditions with Barriers

Jens Krinke
FernUniversität in Hagen[*]*, Germany*

Abstract. One of the critiques on program slicing is that slices presented to the user are hard to understand. This is mainly related to the problem that slicing 'dumps' the results onto the user without any explanation. This work will present an approach that can be used to 'filter' slices. This approach basically introduces 'barriers' which are not allowed to be passed during slice computation. An earlier filtering approach is chopping which is also extended to obey such a barrier. The barrier variants of slicing and chopping provide filtering possibilities for smaller slices and better comprehensibility. The concept of barriers is then applied to path conditions, which provide necessary conditions under which an influence between the source and target criterion exists. Barriers make those conditions more precise.

Keywords: program slicing, program dependence graph, path conditions

1. Introduction

Program slicing answers the question "Which statements may affect the computation at a different statement?". At first sight, an answer to that question should be a valuable help to programmers. After Weiser's first publication (Weiser, 1979) on slicing, almost 25 years have passed and various approaches to compute slices have evolved. Usually, inventions in computer science are adopted widely after around 10 years. Why are slicing techniques not easily available yet? William Griswold gave a talk at PASTE 2001 (Griswold, 2001) on that topic—*Making Slicing Practical: The Final Mile*. He pointed out why slicing is still not widely used today. One of the main problems is that slicing 'as-it-stands' is inadequate to essential software-engineering needs. Usually, slices are hard to understand. This is partly due to bad user interfaces, but mainly related to the problem that slicing 'dumps' the results onto the user without any explanation. Griswold stated the need for "slice explainers" that answer the question why a statement is included in the slice, as well as the need for "filtering". This work will present such a "filtering" approach to slicing; it basically introduces 'barriers' which are not allowed to be passed during slice computation. Especially for chopping, barriers can be used to focus a chop onto interesting program parts.

The next section will present slicing and chopping in detail. Section three will introduce barrier slicing and chopping together with examples. Path con-

[*] Most of the work was done at Universität Passau, Germany

[†] This article appeared in the Software Quality Journal, Volume 12, Number 4, December 2004. The original publication is available at `http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1023/B:SQJO.0000039792.93414.a5`

ditions are presented with their barrier variants in Section four. This work closes with a discussion of related work and conclusions.

2. Slicing and Chopping

A slice extracts those statements from a program that potentially have an influence onto a specific statement of interest which is the slicing criterion. Slicing has found its way into various applications. Currently, it is probably mostly used in the area of software maintenance and reengineering. It is often a base technique to ensure the quality of the developed software, like in testing (Gupta et al., 1992; Binkley, 1992; Bates and Horwitz, 1993; Binkley, 1998), checking a program for robustness (Harman and Danicic, 1995), impact analysis (Gallagher and Lyle, 1991), and cohesion measurement (Ott and Thuss, 1989; Bieman and Ott, 1994; Ott and Bieman, 1998).

Originally, slicing was defined by Weiser in 1979; he presented an approach to compute slices based on iterative data flow analysis (Weiser, 1979; Weiser, 1984). The other main approach to slicing uses reachability analysis in program dependence graphs (PDGs) (Ferrante et al., 1987). 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.

The extension of the PDG for *interprocedural programs* introduces more nodes and edges: For every procedure, a *procedure dependence graph* is constructed, which is basically a PDG with *formal-in* and *-out* nodes for every formal parameter of the procedure. A procedure call is represented by a *call* node and *actual-in* and *-out* nodes for each actual parameter. The call node is connected to the entry node by a *call* edge, the *actual-in* nodes are connected to their matching *formal-in* nodes via *parameter-in* edges, and the *actual-out* nodes are connected to their matching *formal-out* nodes via *parameter-out* edges. Such a graph is called *Interprocedural Program Dependence Graph (IPDG)*. The *System Dependence Graph (SDG)* is an IPDG, where *summary edges* between actual-in and actual-out have been added in order to represent transitive dependence due to calls (Horwitz et al., 1990).

To slice programs with procedures, it is not enough to perform a reachability analysis on IPDGs or SDGs. The resulting slices are not accurate as the *calling context* is not preserved: The algorithm may traverse a parameter-in edge coming from a call site into a procedure, may traverse some edges there, and may finally traverse a parameter-out edge going to a different call site. The sequence of traversed edges (the path) is an *unrealizable path*: It is impossible for an execution that a called procedure does not return to its call site. We consider an interprocedural slice to be *precise* if all nodes included in the slice are reachable from the criterion by a *realizable* path.

Definition 1. (Slice in an IPDG) 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.

We can extend the slicing criterion to allow a set of nodes $C \subseteq N$ instead of one single node:

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

These definitions cannot be used in an algorithm directly, because it is impractical to check paths whether they are interprocedurally realizable. Accurate slices can be calculated with a modified algorithm on SDGs (Horwitz et al., 1990): The benefit of SDGs is the presence of *summary* edges that represent transitive dependence due to calls. Summary edges can be used to identify actual-out nodes that are reachable from actual-in nodes by an interprocedurally realizable path through the called procedure without analyzing it. The idea of the slicing algorithm using summary edges (Horwitz et al., 1990; Reps et al., 1994) is first to slice from the criterion only ascending into calling procedures, and then to slice from all visited nodes only descending into called procedures. We refer the reader to (Krinke, 2002; Krinke, 2003) for a presentation of the algorithms.

Slicing identifies statements in a program which may influence a given statement (the slicing criterion), but it cannot answer the question why a specific statement is part of a slice. A more focused approach can help: *Chopping* (Jackson and Rollins, 1994) 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. This is basically the set of nodes which are lying on a path from s to t in the PDG.

Definition 2. (Chop) 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 t:

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

Here, $p \in s \rightarrow^{\star}_{\mathsf{R}} t$ denotes that path p is an interprocedurally realizable path from s to t.

Again, we can extend the chopping criteria to allow sets of nodes: The chop $C(S,T)$ of an IPDG from the source criterion nodes S to the target criterion nodes T consists of all nodes on which a node in T (transitively) depends via an interprocedurally realizable path from a node in $S \subseteq N$ to the node in $T \subseteq N$:

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

Jackson and Rollins restricted s and t to be in the same procedure and only traversed control dependence, data dependence, and summary edges, but not parameter or call edges. The resulting chop is called a *truncated same-level chop* C^{TS}; "truncated" because nodes of called procedures are not included. Reps and Rosay presented more variants of precise chopping (Reps and Rosay, 1995). A *non-truncated* same-level chop C^{NS} is like the truncated chop but includes the nodes of called procedures. They also present truncated and non-truncated *non-same-level* chops C^{TN} and C^{NN} (which they call *interprocedural*), where the nodes of the chopping criterion are allowed to be in different procedures. Again, the algorithms are explained in (Krinke, 2002; Krinke, 2003).

3. Barrier Slicing and Chopping

The presented slicing and chopping techniques compute very fixed results where the user has no influence. However, during slicing and chopping a user might want to give additional restrictions or additional knowledge to the computation:

1. A user might know that a certain data dependence cannot exist. Because the underlying data flow analysis is a conservative approximation and the pointer analysis is imprecise, it might be clear to the user that a dependence found by the analysis cannot happen in reality. For example, the analysis assumes a dependence between a definition `a[i]=...` and a usage `...=a[j]` of an array, but the user discovers that `i` and `j` never have the same value. If such a dependence is removed from the dependence graph, the computed slice might be smaller.

2. A user might want to exclude specific parts of the program which are of no interest for his purposes. For example, he might know that certain statement blocks are not executed during runs of interest; or he might want to ignore error handling or recovery code if he is only interested in normal execution.

3. During debugging, a slice might contain parts of the analyzed program that are known (or assumed) to be bug-free. These parts should be removed from the slice to make the slice more focused.

These points have been tackled independently: For example, the removal of dependences from the dependence graph by the user has been applied in Steindl's slicer (Steindl, 1999). The removal of parts from a slice is called *dicing* (Lyle and Weiser, 1987).

The following approach integrates both into a new kind of slicing, called *barrier slicing*, where nodes (or edges) in the dependence graph are declared to be a *barrier* that transitive dependence is not allowed to pass.

Definition 3. (Barrier Slice) The *barrier slice* $S_{\#}(C, B)$ of an IPDG $G = (N, E)$ for the slicing criterion $C \subseteq N$ with the barrier set of nodes $B \subseteq N$ consists of all nodes on which a node $n \in C$ (transitively) depends via an interprocedurally realizable path that does not pass a node in B:

$$\begin{aligned} S_{\#}(C, B) = \{m \in N \mid\ & p \in m \rightarrow^{\star}_{\mathsf{R}} n \wedge n \in C \\ & \wedge\ p = \langle n_1, \ldots, n_l \rangle \\ & \wedge\ \forall 1 < i \leq l : n_i \notin B\} \end{aligned}$$

The barrier may also be defined by a set of edges; the previous definition is adapted accordingly.

From barrier slicing it is only a small step to barrier chopping:

Definition 4. (Barrier Chop) The *barrier chop* $C_{\#}(S, T, B)$ of an IPDG $G = (N, E)$ from the source criterion $S \subseteq N$ to the target criterion $T \subseteq N$ with the barrier set of nodes B consists of all nodes on which a node in T (transitively) depends via an interprocedurally realizable path from a node in S to the node in T that does not pass a node in $B \subseteq N$:

$$\begin{aligned} C_{\#}(S, T, B) = \{n \in N \mid\ & p \in s \rightarrow^{\star}_{\mathsf{R}} t \wedge s \in S \wedge t \in T \\ & \wedge\ p = \langle n_1, \ldots, n_l \rangle \wedge \exists i : n = n_i \\ & \wedge\ \forall 1 < j < l : n_j \notin B\} \end{aligned}$$

The barrier may also be defined by a set of edges; the previous definition is adapted accordingly.

Again, the forward/backward, truncated/non-truncated, same-level/non-same-level variants can be defined, but are not presented here.

The computation of barrier slices and chops causes a problem: The additional constraint of the barrier destroys the usability of summary edges as they do not obey the barrier. Even if summary edges would comply with the barrier, the advantage of summary edges is lost: They can no longer be

Algorithm 1 Computation of Blocked Summary Edges

```
**Input:** $G = (N, E)$ the given SDG, $B \subset N$ the given barrier
**Output:** A set $S$ of blocked summary edges

$S = \emptyset, W = \emptyset$ *Initialization*
*Block all reachable summary edges*
foreach $n \in B$ do
    Let $P$ be the procedure containing $n$
    Let $S_P$ be the set of summary edges for calls to $P$
    $S = S \cup S_P, W = W \cup \{(m, m) \mid m$ is a formal-out node of $P\}$

repeat
    $S_0 = S$
    foreach $x \rightharpoonup y \in S$ do
        Let $P$ be the procedure containing $x$
        Let $S_P$ be the set of summary edges for calls to $P$
        $S = S \cup S_P, W = W \cup \{(m, m) \mid m$ is a formal-out node of $P\}$
until $S_0 = S$

*Unblock some summary edges, Invariants:*
*1.* $W \subseteq M$
*2.* $(n, m) \in M \Rightarrow n \rightarrow^\star m$ *via a barrier-free intraprocedural path,*
                        *$m$ is a formal-out node.*
$M = W$
while $W \neq \emptyset$ *worklist is not empty* do
    $W = W / \{(n, m)\}$ *remove one element from the worklist*
    if $n$ is a formal-in node then
        *A barrier-free path from formal-in n to formal-out m exists*
        foreach $n' \xrightarrow{\text{pi}} n$ which is a parameter-in edge do
            foreach $m \xrightarrow{\text{po}} m'$ which is a parameter-out-edge do
                if $n' \notin B \wedge m' \notin B \wedge n' \xrightarrow{\text{su}} m' \in S$ then
                    $S = S / \{n' \xrightarrow{\text{su}} m'\}$ *unblock summary edge*
                    foreach $(m', x) \in M \wedge (n', x) \notin M$ do
                        $M = M \cup \{(n', x)\}, W = W \cup \{(n', x)\}$
    else
        foreach $n' \xrightarrow{\text{dd,cd}} n$ do
            if $n' \notin B \wedge (n', m) \notin M$ then
                $M = M \cup \{(n', m)\}, W = W \cup \{(n', m)\}$
        foreach $n' \xrightarrow{\text{su}} n$ do
            if $n' \notin B \wedge (n', m) \notin M \wedge n' \xrightarrow{\text{su}} n \notin S$ then
                $M = M \cup \{(n', m)\}, W = W \cup \{(n', m)\}$
return $S$ *the set of blocked summary edges*
```

computed once and used for different slices and chops because they have to be computed for each barrier slice and chop individually.

The usual algorithm (Reps et al., 1994) can be adapted to compute summary edges which obey the barrier: The new version (algorithm 1) is based on blocking and unblocking summary edges. First, all summary edges stemming from calls that might call a procedure with a node from the barrier at some time are blocked. This set is a too conservative approximation, and the second step unblocks summary edges where a barrier-free path exists between the formal-in and -out node corresponding to the summary edge's actual-in and -out node. The algorithm propagates pairs (n, m) which state that formal-out node m is intraprocedurally reachable from n via a barrier-free path. The pairs are propagated via worklist W and kept in set M. If a pair from a formal-in to a formal-out node is encountered, all corresponding summary edges in calling procedures must be unblocked. That step must also propagate earlier encountered pairs along the now unblocked summary edge. (The propagation may have stopped at the blocked summary edge earlier.)

The first phase of the algorithm replaces the initialization phase of the original algorithm and the second phase does not generate new summary edges (like the original), but unblocks them. Only the version where the barrier consists of nodes is shown. This algorithm is cheaper than the complete re-computation of summary edges, because it only propagates node pairs to find barrier-free paths between actual-in/-out nodes if a summary edge, and therefore a (not necessarily barrier-free) path, exists.

Example 1. Consider the example in Figure 1: If a slice for `u_kg` in line 33 is computed, almost the complete program is in the slice: Just lines 11 and 12 are omitted. One might be interested in why the variable `p_cd` is in the slice and has an influence on `u_kg`. Therefore, a chop is computed: The source criterion are all statements containing variable `p_cd` (lines 9, 21, 23, 24 and 31) and the target criterion is `u_kg` in line 33. The computed chop is shown in Figure 2. In that chop, line 19 looks suspicious: variable `u_kg` is defined, using variable `cal_kg`. Another chop from all statements containing variable `cal_kg` to the same target consists only of lines 14, 19, 26, 28 and 33 (figure 3). A closer look reveals that statements 26 and 28 “transmit” the influence from `p_cd` on `u_kg`.

To check that no other statement is responsible, a barrier chop is computed: The source are the statements with `p_cd` again, the target criterion is still `u_kg` in line 33, and the barrier consists of lines 26 and 28. The computed chop is empty and reveals that lines 26 and 28 are the “hot spots”.

The barrier slice with the criterion `u_kg` in line 33 and the same barrier reveals the “intended” computation, which consists of lines 8, 13, 14, 16–19 and 33.

```
#define TRUE 1
#define CTRL2 0
#define PB 0
#define PA 1

void main()
{
  int p_ab[2] = {0, 1};
  int p_cd[1] = {0};
  char e_puf[8];
  int u;
  int idx;
  float u_kg;
  float cal_kg = 1.0;

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

Figure 1. An example

```
#define TRUE 1
#define CTRL2 0
#define PB 0
#define PA 1

void main()
{
  int p_ab[2] = {0, 1};
  int p_cd[1] = {0};
  char e_puf[8];
  int u;
  int idx;
  float u_kg;
  float cal_kg = 1.0;

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

Figure 2. A chop for the example in Figure 1

```
#define TRUE 1
#define CTRL2 0
#define PB 0
#define PA 1

void main()
{
  int p_ab[2] = {0, 1};
  int p_cd[1] = {0};
  char e_puf[8];
  int u;
  int idx;
  float u_kg;
  float cal_kg = 1.0;

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

Figure 3. Another chop for the example

3.1. C C

A specialized version of a barrier chop is a *core chop*, where the barrier consists of the source and target criterion nodes.

Definition 5. (Core Chop) A *core chop* $C_\circ(S,T)$ is defined as

$$C_\circ(S,T) = C_\#(S,T,S \cup T)$$

It is well suited for chops with large source and target criterion sets: Only the statements connecting the source to the target are part of the chop. Here is important that a barrier chop allows barrier nodes to be included in the criteria. In that case, the criterion nodes are only start or end nodes of the path and are not allowed elsewhere.

3.2. S C

When slices or chops are computed for large criterion sets, it is sometimes important to know which parts of the criterion set influence themselves and which statements are part of such an influence. If these statements have been identified, they can be handled particularly during following analyses. They can simply be computed by a *self chop*, where a set is both source and target criterion:

Definition 6. (Self Chop) A *self chop* $C_\bowtie(S)$ is defined as

$$C_\bowtie(S) = C(S,S)$$

It computes the strongly connected components of the SDG which contain nodes of the criterion. These components can be of special interest to the user, or they are used to make core chops even stronger:

Definition 7. (Strong Core Chop) A *strong core chop* $C_\bullet(S,T)$ is defined as

$$\begin{aligned} C_\bullet(S,T) = \; & C_\#(S \cup C_\bowtie(S), \\ & \quad T \cup C_\bowtie(T), \\ & \quad S \cup T \cup C_\bowtie(S) \cup C_\bowtie(T)) \end{aligned}$$

It only contains statements that connect the source criterion to the target criterion, none of the resulting statements will have an influence on the source criterion, and the target criterion will have no impact on the resulting statements.

Thus, the strong core chop only contains the most important nodes of the influence between the source and target criterion. This will be demonstrated with the example before:

Table I. Comparison of normal and core chopping

Program	LOC	nodes	chops	normal chops			core chops		
				time	nodes	%	time	nodes	%
agrep	3968	22823	8100	2680	4609	20	4346	4035	17
ansitape	1744	8509	5776	651	1022	12	1580	901	10
cdecl	3879	13339	2809	241	690	5	655	501	3
ctags	2933	12961	10201	2162	1753	13	5396	1567	12
football	2261	18879	5329	1318	2285	12	4420	2060	10
gnugo	3305	7787	1444	139	2645	33	171	2232	28
simulator	4476	14705	15376	6767	4753	32	7460	4503	30

Example 2. Let us compute a core chop with criteria similar to Example 1: The source criterion are all statements containing variable `p_cd` (lines 9, 21, 23, 24 and 31) and the target criterion consists of accesses of variable `u_kg` in lines 19 and 33. The computed chop is shown in Figure 4 and contains only statements that are involved in an influence between `p_cd` and `u_kg`.

If a strong core chop is computed instead, line 22 will no longer be in the computed chop, thus revealing an even stronger result.

3.3. E

To evaluate the barrier variants of slicing and chopping, we have performed a comparison of standard and core chopping. For a subset of the programs used in the evaluation in (Krinke, 2002), we computed standard and core chops between the program's procedures. The set of all nodes in a procedure was used as source or target criterion, and a chop was computed for every pair of the criteria (similar to the visualization before). The programs, their sizes (in lines of code, LOC), the number of nodes in their IPDGs, and the number of computed chops is shown in Table I. We have measured the time (in seconds) that was needed to compute all the chops, and their average size in nodes (absolute and as percentage of the complete IPDG). We can see that the computation of the core chops needs on average up to three times more time. The reason is the recomputation of blocked summary edges for every chop. On the other hand, the computed barrier chops are between 5% and 27% smaller than their normal counterparts.

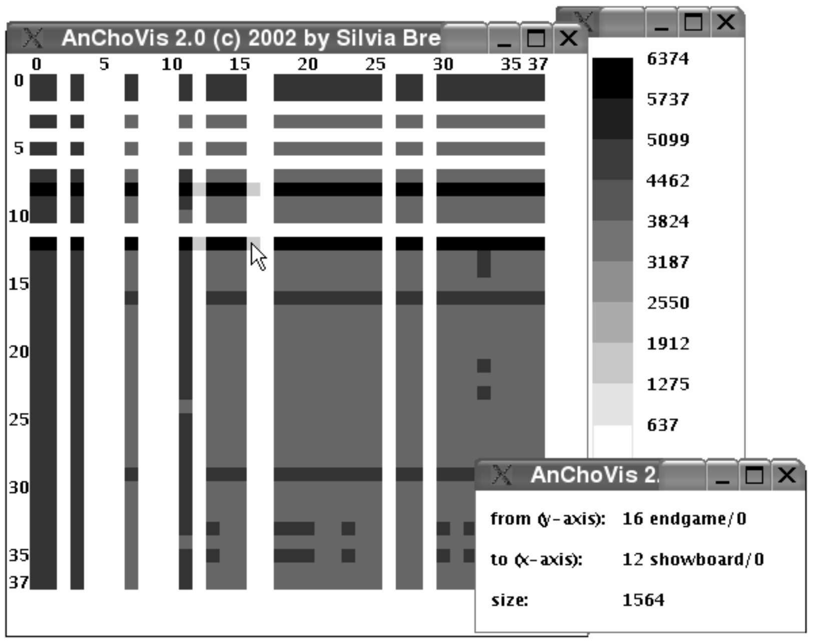
Whether the barrier variants of slicing and chopping are actually useful cannot easily be evaluated just by measuring times or sizes. The difference in results between the barrier and the usual variant depend on the chosen barrier.

```
#define TRUE 1
#define CTRL2 0
#define PB 0
#define PA 1

void main()
{
  int p_ab[2] = {0, 1};
  int p_cd[1] = {0};
  char e_puf[8];
  int u;
  int idx;
  float u_kg;
  float cal_kg = 1.0;

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

Figure 4. A core chop for the example



from (y-axis): 16 endgame/0

to (x-axis): 12 showboard/0

size: 1564

Figure 5. Visualization of the gnugo program

3.4. V

To understand a previously unknown program, it is helpful to identify the ‘hot’ procedures and global variables—those with the highest 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 (Krinke, 2004) 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_i to n_j. The matrix is drawn using a color that corresponds to the size for every entry—the bigger, the darker.

Example 3. Figure 5 shows such a visualization for the `gnugo` program. The three windows contain the chop matrix visualization (in this case for procedure-procedure-chops), a scale that maps colors to sizes, and a window that shows the names of the procedures and their chop’s size for the last chosen matrix element. The columns show the procedures 0-37 as source criteria and the rows the procedures as target criteria. Each entry $n_{i,j}$ of the matrix represents the size of a chop from all nodes of procedure n_j to all nodes of procedure n_i. For instance, the grey box at $(16, 12)$ represents the size of the chop from procedure `endgame` to procedure `showboard`, which is

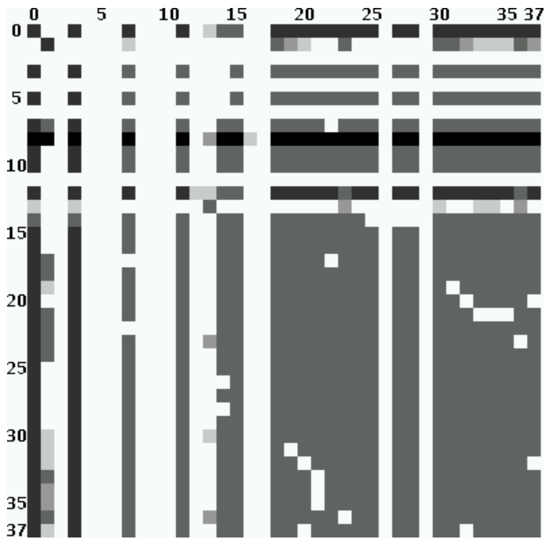


Figure 6. Visualization of the gnugo program with core chops

1564 nodes. Through the light color of column 16 for procedure endgame it is obvious that it has only a small impact on the rest of the system. A close look reveals that this procedure performs the board cleanup at a game's end. There are only two dark rows, they correspond to procedures printf (8) and showboard (12) which are called in endgame.

This visualization shows some typical features of programs like gnugo: The left and upper part of the matrix contain light colors, meaning the chops between the corresponding procedures are small. The procedures with small numbers are typically system routines that return some information from the operating system. Because that information is normally not influenced by the program, the chops with that procedures are small or even empty. The upper very dark row 8 represents the size of the chops with procedure printf as target criterion. Indeed, as most generated information is printed on the screen at some point, the chops must be large. The lower and right area of the matrix (rows and columns 13–37) has an almost uniform color, which means that the chops are of similar size. In programs like gnugo which have one central task (here to play Go), this can be observed very often. All chops contain a large share of the system, because everything influences everything else.

Figure 6 shows the same visualization but now with core chops instead. It shows a much more expressive picture. Of course, the chops are smaller on average, but there are many regions with extreme differences to the previous example. These differences are usually related to recursion: Programs like gnugo often have large components of recursive procedures. If a procedure of such a component is source or target criterion of a chop, the chop will always contain the complete component. This is different with core chops: If the source criterion is a procedure of the component, the core chop will omit the parts of the component which only reach the target criterion by passing through the source criterion again. If the target criterion is a procedure of the component, the core chop will omit the parts of the component which are only reachable from the source criterion by passing through the target criterion before. The omitted parts are usually not interesting, and thus, the core chops are more expressive.

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.

3.5. E B

The barriers are generated to capture observed behavior of a program. These observations may stem from profiling, audits, or even test data gathered from deployed software (Orso et al., 2003). Such data usually contains information about which statements or procedures of the deployed software have been executed or have not. The straightforward approach would just generate the barrier from the nodes of the not executed statements or procedures. However, the barrier can and should be made stronger. As an example, assume gathered data reveals that a procedure *p* was never executed. Instead of just including all nodes of *p* into the barrier, all nodes of all call sites to *p* can be included, too. Even all nodes of the basic blocks containing the call sites can be included. More formally, the barrier *B* is extended by all nodes *n*, where *n* has only predecessors in the control flow graph also contained in *B*, or where *n* has only successors in the control flow graph also contained in *B*. *B* is a *complete execution barrier* if no such node exist. For application of complete execution barriers, all previous definitions and algorithms of slices and chops stay unchanged—we only have to extend the barriers as described until they are complete execution barriers.

We should recall that the presented barrier slicing is not as precise as generally possible. Consider the following fragment:

```
void p(int b, int c) {
  int a;
  read(a);
  if (b) {
    a = c;
  } else {
    b = c;
  }
  print(a);
```

Here, two data dependence edges related to statement 9 exist: from the definition due to the `read(a)` in line 3 and from the definition in line 5. If gathered field data has revealed that statement 7 has never been executed, the definition of line 3 never reached line 7, because it was always killed in line 5. However, if a barrier slice for barrier $B = \{7\}$ and criterion $C = \{9\}$ is computed, it will contain line 3, because the data dependence from 3 to 7 is still in the IPDG.

The only way to reach that precision is to generate the barriers *before* data flow analysis: All nodes of the barrier are removed from the CFG, and the IPDG and the slices are computed based on the reduced CFG. The disadvantage of this approach is that whenever the barrier is changed, the expensive construction of the IPDG must be repeated. Thus, the previously presented approach of barrier slicing is much cheaper in comparison.

4. Path Conditions

Slicing can answer the question "*which statements have an influence on statement X?*", chopping can answer the question "*how does statement Y influence statement X?*", but neither slicing nor chopping can answer the question "*why does statement Y influence statement X*".

Example 4. Consider Figure 7 for an example: First, the question "*which statements influence the output of variable* `y` *in line 8*" is simply answered by computing a slice for this criterion. Figure 8 shows the result: Just one statement or node does not influence the criterion. Now, why is statement 1 included in the slice and how does statement 1 influence statement 8? This can be answered by a chop between statement 1 and statement 8. However, the result is a chop almost identical to the previous slice—which does not help at all.

Here, the computation of *path conditions* (Snelting, 1996) can assist. Path conditions give necessary conditions under which a transitive dependence between the source and target (criterion) node exists. These conditions give the answer to "why is this statement in the slice?" which Griswold (Griswold, 2001) categorized as a question hard to answer.

This section will first introduce the basic concepts of path conditions, and then show how the concept of barriers can be used for more precise path conditions.

4.1. S P C

A simple approach to compute path conditions between two nodes x and y in a program dependence graph consists of the following steps:

1. Compute all paths p_i from x to y in the program dependence graph.

```
1    if (i > 0)
2      a = x;
3      j = b;
4    c = z;
5    if (j < 5)
6      if (i < 8)
7        y = a;
8    print(y);
```

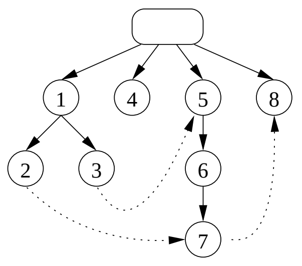


Figure 7. Simple fragment with program dependence graph

```
1    if (i > 0)
2      a = x;
3      j = b;
4
5    if (j < 5)
6      if (i < 8)
7        y = a;
8    print(y);
```

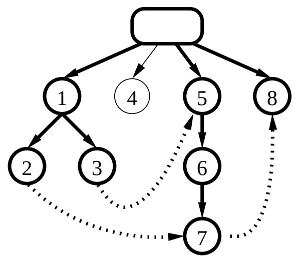


Figure 8. Slice of the fragment

2. For every node x that is part of a path p_i, compute the *execution condition* $E(x)$.

3. Combine the execution conditions to compute the path condition $\mathrm{PC}(x, y)$.

These three steps will be discussed next.

4.1.1. *Execution Conditions*

The execution condition $E(x)$ gives the conditions under which a node x may be executed. This can simply be computed by following the incoming control dependence edges and collecting the predicates of the ancestor nodes until the root (START) node is reached. In the example in Figure 7, the execution condition for statement 7 is $E(7) = (\texttt{j} < 5) \wedge (\texttt{i} < 8)$. More generally, an execution condition for a node x is computed by

$$E(x) = \bigwedge_{n \xrightarrow{\text{cd}} m \,|\, \text{START} \xrightarrow{\text{cd}}\star\, n \xrightarrow{\text{cd}} m \xrightarrow{\text{cd}}\star\, x} \gamma(n \xrightarrow{\text{cd}} m)$$

$$\gamma(n \xrightarrow{\text{cd}} m) = \begin{cases} \mu(n) & \text{if } \nu(n \xrightarrow{\text{cd}} m) = \mathsf{true} \\ \neg\mu(n) & \text{if } \nu(n \xrightarrow{\text{cd}} m) = \mathsf{false} \\ \mu(n) = \nu(n \xrightarrow{\text{cd}} m) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
E(1) &= \mathsf{true}\\
E(2) &= \mathtt{i} > 0\\
E(3) &= \mathtt{i} > 0\\
E(5) &= \mathsf{true}\\
E(6) &= \mathtt{j} < 5\\
E(7) &= (\mathtt{j} < 5) \wedge (\mathtt{i} < 8)\\
E(8) &= \mathsf{true}
\end{aligned}$$

Figure 9. Execution conditions

where $\mu(n)$ returns the (predicate) expression of a node n and $\nu(e)$ returns the label of edge e. Control dependence edges leaving predicates of if- and while-statements are labeled with either true or false, and control dependence edges leaving expressions of switch statements are labeled with the constant of the target case.

Example 5. Figure 9 shows the execution conditions for each of the example's statements.

In the presence of unstructured control flow, the control dependence subgraph may not be a tree, and there may be more than one single path from the root to the node of interest. Under such circumstances, the execution conditions compute as follows:

$$\begin{aligned}
E(p) &= \bigwedge_{n \overset{\text{cd}}{\to} m \mid p = \text{START} \overset{\text{cd}}{\to}{}^{\star} n \overset{\text{cd}}{\to} m \overset{\text{cd}}{\to}{}^{\star} x} \gamma(n \overset{\text{cd}}{\to} m)\\
E(x) &= \bigvee_{p = \text{START} \overset{\text{cd}}{\to}{}^{\star} x} E(p)
\end{aligned}$$

In the presence of unstructured control flow, the control dependence subgraph may even be cyclic, which causes the set of possible paths to be infinite. Snelting (Snelting, 1996) proved that cycles in execution conditions can be ignored, and thus, execution conditions are only computed over the finite set of cycle-free paths.

Other actions to handle unstructured control flow are not necessary.

4.1.2. *Combining Execution Conditions*

The execution conditions are used to form the path conditions. For a path p in a program dependence graph, the execution conditions are conjunctively combined to result in the conditions under which this path may be taken during execution:

$$\mathrm{PC}(p) = \bigwedge_{n \mid p=\langle\ldots,n,\ldots\rangle} E(n) \tag{1}$$

Usually, more than one path exists between two nodes x and y, and the path conditions for the single paths are combined disjunctively to form the path condition $\mathrm{PC}(x,y)$:

$$\mathrm{PC}(x,y) = \bigvee_{p=x\rightarrow^\star y} \mathrm{PC}(p) \tag{2}$$

A program dependence graph is typically cyclic which may result in an infinite number of paths between two nodes. Again, cycles in paths can be ignored (Snelting, 1996) and only cycle-free paths are used.

Example 6. The example in Figure 7 contains two paths from statement 1 to 8: $p_1 = \langle 1,2,7,8\rangle$ and $p_2 = \langle 1,3,5,6,7,8\rangle$. Their path conditions are:

$$\begin{aligned}
\mathrm{PC}(p_1) &= \mathsf{true} \wedge (\mathtt{i} > 0) \wedge (\mathtt{j} < 5) \wedge (\mathtt{i} < 8) \wedge \mathsf{true}\\
&= (\mathtt{i} > 0) \wedge (\mathtt{j} < 5) \wedge (\mathtt{i} < 8)\\
\mathrm{PC}(p_2) &= \mathsf{true} \wedge (\mathtt{i} > 0) \wedge \mathsf{true} \wedge (\mathtt{j} < 5) \wedge (\mathtt{j} < 5) \wedge (\mathtt{i} < 8) \wedge \mathsf{true}\\
&= (\mathtt{i} > 0) \wedge (\mathtt{j} < 5) \wedge (\mathtt{i} < 8)
\end{aligned}$$

Both path conditions can be combined disjunctively and simplified: The path condition from statement 1 to 8 is then $\mathrm{PC}(1,8) = (\mathtt{i} > 0) \wedge (\mathtt{j} < 5) \wedge (\mathtt{i} < 8)$.

Example 7. We now compute a path condition for the example in Figure 2. There, we have computed a chop between all statements containing variable `p_cd` (lines 9, 21, 23, 24 and 31), and the target criterion, namely the use of `u_kg` in line 33. To reveal the necessary conditions for the influence, we compute the path condition between line 9 and line 33:

$$\begin{aligned}
\mathrm{PC}(9,33) = {} & (\mathtt{p\_ab[CTRL2]\ \&\ 0x10} = \mathtt{0})\\
& \wedge (\mathtt{p\_cd[CTRL2]\ \&\ 0x01} \neq \mathtt{0})\\
& \wedge (\mathtt{p\_cd[CTRL2]\ \&\ 0x10} \neq \mathtt{0})\\
& \wedge (\mathtt{idx} < 7)\\
& \wedge ((\mathtt{e\_puf[idx]} = \text{‘}\mathtt{+}\text{’}) \vee (\mathtt{e\_puf[idx]} = \text{‘}\mathtt{-}\text{’}))
\end{aligned}$$

With some domain knowledge, this path conditions can be interpreted as “If the keyboard input is `+` or `-`, and (not necessarily at the same time) the ‘paper out’ signal is active, there is data flow from the keyboard to the displayed weight value”. A human would have a hard time to extract such statements!

This section has only explained the required basics of path conditions; more detailed presentations, including interprocedural and multi-threaded path conditions, are given by (Krinke and Snelting, 1998; Robschink and Snelting, 2002; Snelting et al., 2003; Krinke, 2003). The presented approach has been implemented and evaluated. Using sophisticated optimizations, it is possible to compute path conditions efficiently even in the interprocedural case (Robschink and Snelting, 2002; Snelting et al., 2003).

4.2. P C B

We will now extend the computation of path conditions in a way that barriers are obeyed. The barrier B now specifies all nodes that are guaranteed to not be part of an influence between x and y in a path condition $\mathrm{PC}_{\#}(x, y, B)$. First, we revisit equations 1 and 2, which are extended to obey the barrier B. For simplicity, the notation $p \rightsquigarrow B$ denotes that no node of path p is contained in the barrier B, $p = \langle n_1, \ldots, n_l\rangle \wedge \forall 1 < i \leq l : n_i \notin B$:

$$\mathrm{PC}_{\#}(x, y, B) = \bigvee_{p = x \rightarrow^{\star} y \,\wedge\, p \rightsquigarrow B} \mathrm{PC}_{\#}(p, B)$$

$$\mathrm{PC}_{\#}(p, B) = \bigwedge_{n \mid p = \langle \ldots, n, \ldots\rangle \,\wedge\, p \rightsquigarrow B} E_{\#}(n, B)$$

This definition just restricts the path conditions to paths not passing through the barrier. Second, the execution conditions must be adapted:

$$\begin{aligned} E_{\#}(x, B) &= \bigvee_{p = \mathrm{START} \overset{\mathrm{cd}}{\rightarrow}{}^{\star} x \,\wedge\, p \rightsquigarrow B} E_{\#}(p, B) \\ E_{\#}(p, B) &= \bigwedge_{n \overset{\mathrm{cd}}{\rightarrow} m \mid p = \mathrm{START} \overset{\mathrm{cd}}{\rightarrow}{}^{\star} n \overset{\mathrm{cd}}{\rightarrow} m \overset{\mathrm{cd}}{\rightarrow}{}^{\star} x \,\wedge\, p \rightsquigarrow B} \gamma(n \overset{\mathrm{cd}}{\rightarrow} m) \end{aligned}$$

Again, the introduced restriction is the omission of paths passing through the barrier. This restriction ensures that whenever a controlling predicate is part of the barrier, no controlled statement will be part of the path condition.

Example 8. We now turn back to the example in Figure 7, where we now specify a barrier $B = \{6\}$, which states that the predicate of line 6 has no influence. The barrier allows only one path from statement 1 to 8: $p_1 = \langle 1, 2, 7, 8\rangle$, because the other path $p_2 = \langle 1, 3, 5, 6, 7, 8\rangle$ passes through the barrier:

$$\begin{aligned} \mathrm{PC}_{\#}(x, y, B) &= \mathrm{PC}_{\#}(p_1, B) \\ &= E_{\#}(1, B) \wedge E_{\#}(2, B) \wedge E_{\#}(7, B) \wedge E_{\#}(8, B) \end{aligned}$$

The execution condition $E(7, B)$ now results in false, because there is only one path from START to 7, but this path contains line 6 which is an element of the barrier. Thus, the complete path condition is $\mathrm{PC}_{\#}(1, 8, \{6\}) = \mathsf{false}$. This is in contrast to the barrier slice $S_{\#}(8, \{6\}) = \{1, 2, 7, 8\}$ which contains statement 1, stating a barrier-free influence of statement 1 on statement 8.

5. Related Work

Chopping as presented here has been introduced by Jackson and Rollins (Jackson and Rollins, 1994), extended by Reps and Rosay (Reps and Rosay, 1995), and implemented in CodeSurfer (Anderson and Teitelbaum, 2001). An evaluation of various slicing and chopping algorithms has been done in (Krinke, 2002).

A *decomposition slice* (Gallagher and Lyle, 1991; Gallagher, 1996; Gallagher and O'Brien, 1997) is basically a slice for a variable at all statements writing that variable. The decomposition slice is used to form a graph using the partial ordering induced by proper subset inclusion of the decomposition slices for all variables.

Beck and Eichmann (Beck and Eichmann, 1993) use slicing to isolate statements of a module that influence an exported behavior. Their work uses *interface dependence graphs* and *interface slicing*.

Steindl (Steindl, 1998; Steindl, 1999) has developed a slicer for Oberon where the user can choose certain dependences to be removed from the dependence graph.

Set operations on slices produce various variants: Chopping uses intersection of a backward and a forward slice. The intersection of two forward or two backward slices is called a *backbone slice*. Dicing (Lyle and Weiser, 1987) is the subtraction of two slices. However, set operations on slices need special attention because the union of two slices may not produce a valid slice (De Lucia et al., 2003).

Orso et al (Orso et al., 2001) present a slicing algorithm which augments edges with types and restricts reachability onto a set of types, creating slices restricted to these types. Their algorithm needs to compute the summary edges specific to each slice (similar to algorithm 1). However, it only works for programs without recursion.

Path conditions have been introduced by (Snelting, 1996) as a way to validate measurement software. The main problem to compute path conditions is scalability and efficiency. Therefore, a divide-and-conquer strategy is applied: Paths and path conditions are decomposed before computation as shown in (Snelting, 1996; Krinke and Snelting, 1998) and (Robschink and Snelting, 2002; Snelting et al., 2003), which also contain case studies and evaluations.

Path conditions are somewhat similar to *conditioned slicing* (De Lucia et al., 1996; Canfora et al., 1998; Danicic et al., 2000; Fox et al., 2004). In conditioned slicing, a program is *conditioned* in a first step: Based on constraints over the input variables in first order logic formula, the program is partially evaluated via symbolic execution. After conditioning, slices are computed in the transformed and in this way smaller program. However, the computation of path conditions is not based on symbolic execution or partial evaluation, and does not handle control flow explicitly (only implicitly through the dependences). Both techniques use theorem provers or constraint solvers to simplify the generated predicates; we use Redlog (Sturm and Weispfenning, 1998). Though similar in spirit, both techniques have different goals: Conditioned slicing is used to compute smaller and more precise slices while path conditions explain *why* something is in a slice.

6. Conclusions

The presented variants of barrier slicing and chopping provide a filtering approach to reduce the size of slices and chops. The examples showed the helpfulness of this approach: Barriers can be used to focus slices and chops on interesting parts, to validate assumptions about slices and chops, and to improve the expressiveness of chop visualization.

Path conditions give necessary conditions under which a transitive dependence between source and target (criterion) nodes exists. These conditions give the answer to “Why is this statement in the slice?”. If barrier or core chops are used instead of traditional chops during path condition generation, only the important parts will be represented in the path condition, making it smaller and thus, more comprehensible. The adaption of path conditions to obey barriers make their use more flexible; instead of a pure static approach, path conditions with barriers can use data from dynamic executions of the analyzed programs. The use of barriers can make the generation of path conditions more precise.

Acknowledgements

Thomas Zimmermann implemented earlier versions of the presented algorithms. Silvia Breu implemented the visualization tool.

References

Anderson, P. and T. Teitelbaum: 2001, ‘Software Inspection Using CodeSurfer’. In: *Workshop on Inspection in Software Engineering (CAV 2001)*.

Bates, S. and S. Horwitz: 1993, ‘Incremental Program Testing Using Program Dependence Graphs’. In: *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 384–396.

Beck, J. and D. Eichmann: 1993, ‘Program and interface slicing for reverse engineering’. In: *IEEE/ACM* 15th *Conference on Software Engineering (ICSE’93)*. pp. 509–518.

Bieman, J. M. and L. M. Ott: 1994, ‘Measuring Functional Cohesion’. *IEEE Transactions on Software Engineering* **20**(8), 644–657.

Binkley, D.: 1992, ‘Using Semantic Differencing to Reduce the Cost of Regression Testing’. In: *Proceedings of the International Conference on Software Maintenance*. pp. 41–50.

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

Canfora, G., A. Cimitile, and A. De Lucia: 1998, ‘Conditioned Program Slicing’. *Information and Software Technology* **40**(11–12), 595–607.

Danicic, S., C. Fox, M. Harman, and R. Hierons: 2000, ‘ConSIT: A conditioned program slicer’. In: *International Conference on Software Maintenance*. pp. 216–226.

De Lucia, A., A. R. Fasolino, and M. Munro: 1996, ‘Understanding Function Behaviors through Program Slicing’. In: 4th *IEEE Workshop on Program Comprehension*. pp. 9–18.

De Lucia, A., M. Harman, R. Hierons, and J. Krinke: 2003, ‘Unions of Slices are not Slices’. In: *7th European Conference on Software Maintenance and Reengineering*.

Ferrante, J., K. J. Ottenstein, and J. D. Warren: 1987, ‘The Program Dependence Graph and Its Use in Optimization’. *ACM Transactions on Programming Languages and Systems* **9**(3), 319–349.

Fox, C., S. Danicic, M. Harman, and R. Hierons: 2004, ‘ConSIT: a fully automated conditioned program slicer’. *Software, Practice and Experience* **34**(1), 15–46.

Gallagher, K. and L. O’Brien: 1997, ‘Reducing Visualization Complexity Using Decomposition Slices’. In: *Software Visualization Workshop*. pp. 113–118.

Gallagher, K. B.: 1996, ‘Visual Impact Analysis’. In: *Proceedings of the International Conference on Software Maintenance*. pp. 52–58.

Gallagher, K. B. and J. R. Lyle: 1991, ‘Using Program Slicing in Software Maintenance’. *IEEE Transactions on Software Engineering* **17**(8), 751–761.

Griswold, W. G.: 2001, ‘Making Slicing Practical: The Final Mile’. Invited Talk, PASTE’01.

Gupta, R., M. J. Harrold, and M. L. Soffa: 1992, ‘An approach to regression testing using slicing’. In: *Proceedings of the IEEE Conference on Software Maintenance*. pp. 299–308.

Harman, M. and S. Danicic: 1995, ‘Using Program Slicing to Simplify Testing’. *Software Testing, Verification and Reliability* **5**(3), 143–162.

Horwitz, S. B., T. W. Reps, and D. Binkley: 1990, ‘Interprocedural Slicing Using Dependence Graphs’. *ACM Transactions on Programming Languages and Systems* **12**(1), 26–60.

Jackson, D. and E. J. Rollins: 1994, ‘A New Model of Program Dependences for Reverse Engineering’. In: *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*. pp. 2–10.

Krinke, J.: 2002, ‘Evaluating Context-Sensitive Slicing and Chopping’. In: *Proc. International Conference on Software Maintenance*. pp. 22–31.

Krinke, J.: 2003, ‘Advanced Slicing of Sequential and Concurrent Programs’. Ph.D. thesis, Universität Passau.

Krinke, J.: 2004, ‘Visualization of Program Dependence and Slices’. In: *Proc. International Conference on Software Maintenance*. pp. 168–177.

Krinke, J. and G. Snelting: 1998, ‘Validation of Measurement Software as an Application of Slicing and Constraint Solving’. *Information and Software Technology* **40**(11-12), 661–675.

Lyle, J. R. and M. Weiser: 1987, ‘Automatic Program Bug Location by Program Slicing’. In: *2 International Conference on Computers and Applications*. pp. 877–882.

Orso, A., T. Apiwattanapong, and M. J. Harrold: 2003, ‘Leveraging Field Data for Impact Analysis and Regression Testing’. In: *Proceedings of the 11th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*.

Orso, A., S. Sinha, and M. J. Harrold: 2001, ‘Incremental Slicing Based on Data-Dependences Types’. In: *International Conference on Software Maintenance*.

Ott, L. M. and J. M. Bieman: 1998, ‘Program Slices as an Abstraction for Cohesion Measurement’. *Information and Software Technology* **40**(11-12), 691–700.

Ott, L. M. and J. J. Thuss: 1989, ‘The Relationship between Slices and Module Cohesion’. In: *Proceedings of the* 11*th* *ACM conference on Software Engineering*. pp. 198–204.

Reps, T., S. Horwitz, M. Sagiv, and G. Rosay: 1994, ‘Speeding up Slicing’. In: *Proceedings of the ACM SIGSOFT ’94 Symposium on the Foundations of Software Engineering*. pp. 11–20.

Reps, T. and G. Rosay: 1995, ‘Precise Interprocedural Chopping’. In: *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*. pp. 41–52.

Robschink, T. and G. Snelting: 2002, ‘Efficient Path Conditions in Dependence Graphs’. In: *Proceedings of the 24th International Conference of Software Engineering (ICSE)*. pp. 478–488.

Snelting, G.: 1996, ‘Combining Slicing and Constraint Solving for Validation of Measurement Software’. In: *Static Analysis Symposium*, Vol. 1145 of *LNCS*. pp. 332–348, Springer.

Snelting, G., T. Robschink, and J. Krinke: 2003, ‘Efficient Path Conditions in Dependence Graphs for Software Safety Analysis’. Submitted for publication.

Steindl, C.: 1998, ‘Intermodular Slicing of Object-Oriented Programs’. In: *International Conference on Compiler Construction*, Vol. 1383 of *LNCS*. pp. 264–278, Springer.

Steindl, C.: 1999, ‘Benefits of a Data Flow-Aware Programming Environment’. In: *Workshop on Program Analysis for Software Tools and Engineering (PASTE’99)*.

Sturm, T. and V. Weispfenning: 1998, ‘Computational Geometry Problems in REDLOG’. In: *Automated Deduction in Geometry*. pp. 58–86.

Weiser, M.: 1979, ‘Program slices: formal, psychological, and practical investigations of an automatic program abstraction method’. Ph.D. thesis, University of Michigan, Ann Arbor.

Weiser, M.: 1984, ‘Program Slicing’. *IEEE Transactions on Software Engineering* **10**(4), 352–357.