Barrier Slicing and Chopping

Jens Krinke
Universität Passau
Passau, Germany

Abstract

One of the critiques on program slicing is that slices presented to the user 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. 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.

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 [20] on slicing in 1979, 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 [8] 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.

This approach 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 two sections will present slicing and chopping in detail. Section four will introduce barrier slicing and chopping together with an example. This work is closed with a discussion of related work and conclusions.

2. Slicing

A slice extracts those statements from a program that potentially have an influence onto a specific statement of interest which is the slicing criterion. Originally, slicing was defined by Weiser in 1979; he presented an approach to compute slices based on iterative data flow analysis [20, 21]. The other main approach to slicing uses reachability analysis in program dependence graphs (PDGs) [4]. 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 representing transitive dependence due to calls [9].

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 finally 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 interprocedural 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 interprocedural realizable path from m to n.

We can extend the slice criterion to allow a set of nodes $C \subseteq N$ instead 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 interprocedural realizable. Accurate slices can be calculated with a modified algorithm on SDGs:

The benefit of SDGs is the presence of *summary* edges that represent transitive dependence due to calls. Summary edges cam be used to identify actual-out nodes that are reachable from actual-in nodes by an interprocedural realizable path through the called procedure without analyzing it. The idea of the slicing algorithm using summary edges [9, 15] 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 [11] for a presentation of the algorithms.

3. Chopping

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: Jackson and Rollins [10] introduced *Chopping* which 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 interprocedural realizable path from node s to node 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 interprocedural 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 of T (transitively) depends via an interprocedural realizable path from a node of $S \subseteq N$ to the node in $T \subseteq N$:

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

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. In [16] Reps and Rosay presented more variants of precise chopping. 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 [11].

4 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 happen. 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, when 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.

Both 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 [18, 19]. The removal of parts from a slice has been presented by Lyle and Weiser [13] and is called *dicing*.

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 interprocedural realizable path that does not pass a node of 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 of T (transitively) depends via an interprocedural realizable path from a node of S to the node in T that does not pass a node of $B \subseteq N$:

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

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

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

Initialization
$S = \emptyset$, $W = \emptyset$
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
  $S = S \cup S_P$
  $W = W \cup \{(n, n) \mid n$ 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
    $S = S \cup S_P$
    $W = W \cup \{(n, n) \mid n$ is a formal-out node of $P\}$
until $S_0 = S$

Unblock some summary edges
P = 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**
    foreach $n' \overset{\text{pi}}{\rightarrow} n$ which is a parameter-in edge **do**
      foreach $m \overset{\text{po}}{\rightarrow} m'$ which is a parameter-out-edge **do**
        if $n' \overset{\text{su}}{\rightarrow} m' \in S$ **then**
          $S = S - \{n' \overset{\text{su}}{\rightarrow} m'\}$ *unblock summary edge*
          foreach $(m', x) \in P \wedge (n', x) \notin P$ **do**
            $P = P \cup \{(n', x)\}$
            $W = W \cup \{(n', x)\}$
  else
    foreach $n' \overset{\text{dd,cd}}{\rightarrow} n$ **do**
      if $n' \notin B \wedge (n', m) \notin P$ **then**
        $P = P \cup \{(n', m)\}$
        $W = W \cup \{(n', m)\}$
    foreach $n' \overset{\text{su}}{\rightarrow} n$ **do**
      if $n' \notin B \wedge n' \overset{\text{su}}{\rightarrow} n \notin S \wedge (n', m) \notin P$ **then**
        $P = P \cup \{(n', m)\}$
        $W = W \cup \{(n', m)\}$
return S *the set of blocked summary edges*

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 cause a minor problem: The additional constraint of the barrier destroys the usability of summary edges as they do not obey the barrier. Even when summary edges would comply with the barrier, the advantage of summary edges is lost: They can no longer be computed once and used for different slices and chops because they have to be computed for each barrier slice and chop individually. However, the original algorithm 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 very conservative approximation and the second step unblocks summary edges where a barrier-free path exists between actual-in and -out nodes. The first phase replaces the initialization phase of the original algorithm and the second phase does not generate new summary edges, but unblocks them. Only the version where the barrier consists of nodes is shown.

This algorithm is cheaper than the complete recomputation 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 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` 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, where the variable `u_kg` is defined, using variable `kal_kg`. Another chop from all statements containing variable `kal_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 is line 26 and 28. The computed chop is empty and reveals that lines 26 and 28 are the "hot spots".

4.1 Core Chop

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

```
 1  #define TRUE 1
 2  #define CTRL2 0
 3  #define PB 0
 4  #define PA 1
 5
 6  void main()
 7  {
 8    int p_ab[2] = {0, 1};
 9    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  }
```

Figure 1. An example

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.

4.2 Self Chop

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. After identifying such statements, they can specially be handled during following analyses. They can be computed simply by a *self chop*, where a set is both source and target criterion:

```
 1  #define TRUE 1
 2  #define CTRL2 0
 3  #define PB 0
 4  #define PA 1
 5
 6  void main()
 7  {
 8    int p_ab[2] = {0, 1};
 9    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  }
```

Figure 2. A chop for the example in figure 1

```
 1  #define TRUE 1
 2  #define CTRL2 0
 3  #define PB 0
 4  #define PA 1
 5
 6  void main()
 7  {
 8    int p_ab[2] = {0, 1};
 9    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  }
```

Figure 3. Another chop for the example

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:

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

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.

5. Related Work

Chopping as presented here has been introduced by Jackson and Rollins [10], extended by Reps and Rosay [16] and implemented in CodeSurfer [1]. An evaluation of various slicing and chopping algorithms has been done in [11].

A *decomposition slice* [7, 6, 5] 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 [2] use slicing to isolate statements of a module that influence an exported behavior. Their work uses *interface dependence graphs* and *interface slicing*.

Steindl [18, 19] 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 [13] 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 [3].

Orso et al [14] presents 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.

6. Conclusions

The presented variants of barrier slicing and chopping provide a filtering approach to reduce the size of slices and chops. The example showed the helpfulness of this approach. Now we are integrating these slicing and chopping algorithms into our VALSOFT slicing system [12]. We are confident that the usefulness of our approach will be shown in following experiments.

The size reduction of chops is especially important for the generation of path conditions [17]. 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?". When 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.

Acknowledgments. Thomas Zimmermann implemented earlier versions of the presented algorithms. Silvia Breu and Maximilian Störzer provided valuable comments.

References

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

[2] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518, 1993.

[3] A. De Lucia, M. Harman, R. Hierons, and J. Krinke. Unions of slices are not slices. In *7th European Conference on Software Maintenance and Reengineering*, 2003.

[4] 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.

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

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

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

[8] W. G. Griswold. Making slicing practical: The final mile, 2001. Invited Talk, PASTE'01.

[9] 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.

[10] 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.

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

[12] 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.

[13] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *2 International Conference on Computers and Applications*, pages 877–882, 1987.

[14] A. Orso, S. Sinha, and M. J. Harrold. Incremental slicing based on data-dependences types. In *International Conference on Software Maintenance*, 2001.

[15] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[16] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 41–52, 1995.

[17] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *Proceedings of the 24th International Conference of Software Engineering (ICSE)*, pages 478–488, 2002.

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

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

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

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