# Better Slicing of Programs with Jumps and Switches

Sumit Kumar[1] and Susan Horwitz[1,2]

[1] University of Wisconsin
{sumit,horwitz}@cs.wisc.edu
[2] GrammaTech, Inc.

**Abstract.** Program slicing is an important operation that can be used as the basis for programming tools that help programmers understand, debug, maintain, and test their code. This paper extends previous work on program slicing by providing a new definition of "correct" slices, by introducing a representation for C-style switch statements, and by defining a new way to compute control dependences and to slice a program-dependence graph so as to compute more precise slices of programs that include jumps and switches. Experimental results show that the new approach to slicing can sometimes lead to a significant improvement in slice precision.

## 1 Introduction

Program slicing, first introduced by Mark Weiser in [13], is a topic of on-going interest. For example, Jens Krinke maintains a website [7] with over 100 references to published work on slicing. This paper makes the following four contributions in the area of program slicing:

**Defining Correct Slices:** Weiser defined a *correct* slice of a program $P$ to be a projection of $P$ with certain properties (see Section 3). Podgurski and Clarke [10] defined a notion of *semantic dependence* that can also be used as the basis for a definition of a correct slice; however, their definition did not take jump statements (`goto`, `break`, etc.) into account. We give an example to illustrate a shortcoming of Weiser's definition, and offer a new definition, similar to the one for semantic dependence, that overcomes the problem with Weiser's definition, and also makes sense for programs with jump statements.

**Language Extension:** We discuss how to represent C-style switch statements in a program's control-flow and program-dependence graphs. To our knowledge, this is the first time switch statements have been discussed as such, rather than assuming that they have been implemented at a low level using gotos. Handling switch statements is important because many slicing applications involve displaying the result of a slice to the programmer, or using the results to create new source code. Thus, for those applications, if a slice includes code from a switch, it needs to be displayed/represented in the new code as a switch rather than in some low-level form. Representing and slicing a switch in a low-level form

© Springer-Verlag Berlin Heidelberg 2002

and then mapping the results back to the source level may lead to a final result that is less precise than the one produced by working on the switch directly.

**Improved Precision:** Finding correct, minimal slices is an undecidable problem, whether correctness is defined according to Weiser, Podgurski/Clarke, or using the new definition proposed here. However, it is still a reasonable goal to design a slicing algorithm that is more precise than previous ones; i.e., to define a new algorithm that is correct, and also produces smaller slices than previous algorithms. In this spirit, we introduce some example programs with jumps and switches for which previous slicing algorithms produce slices that include too many components. While the examples with jumps are somewhat artificial, the examples with switches are motivated by code from real programs. We show that the reason extra components are included in the slices has to do both with how control dependences are defined, and how slices are computed. We then give a new definition of control dependence and a new slicing algorithm that is more precise than previous algorithms in the presence of jumps and/or switches. Due to space constraints, we discuss only intraprocedural slicing; extending the algorithm to be interprocedural is straightforward [8].

**Experimental Results:** While it is possible to produce artificial examples in which our new approach to slicing provides arbitrarily smaller slices than previous approaches, it is important to know how well it will work in practice. We provide some experimental results that show that while in most cases slice sizes are reduced by no more than 5%, there are examples of reductions of up to 35%.

## 2 Background

### 2.1 Assumptions

We assume that we are dealing with *well formed* programs; in particular, that there is neither unreachable code (i.e., there is a path in the program's control-flow graph from the enter node to every other node) nor explicit infinite loops (i.e., there is a path from every node in the control-flow graph to the exit node).

### 2.2 Slicing Using the PDG

Informally, the slice of a program from statement $S$ is the set of program components that might affect $S$, either by affecting the value of some variable used at $S$, or by affecting whether and how often $S$ executes. More precise definitions have been proposed, and are discussed below in Section 3.

Slicing was originally defined by Weiser [13] as the solution to a dataflow problem specified using the program's control-flow graph (CFG). Ottenstein and Ottenstein [9] provided a more efficient algorithm that uses the program-dependence graph (PDG) [4]:

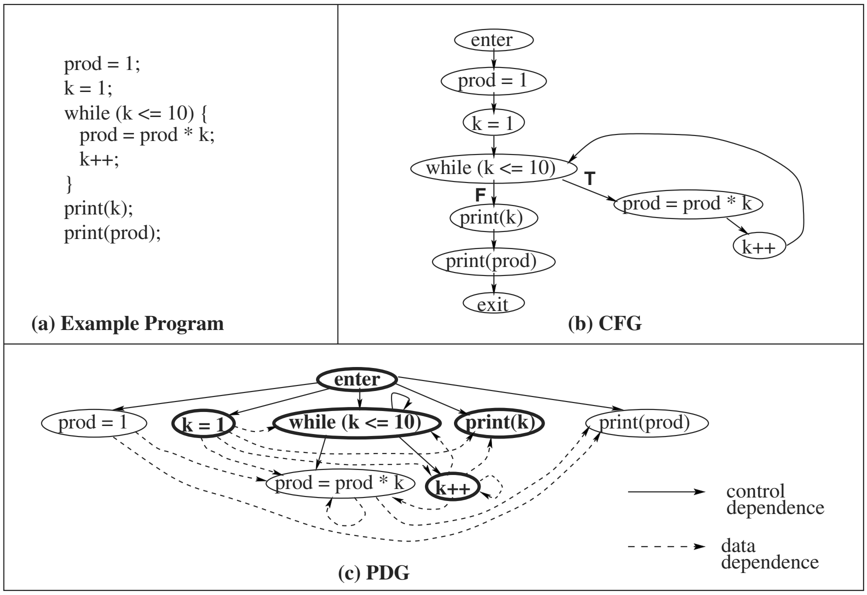


**Fig. 1.** Example program, its CFG, and its PDG. The PDG nodes in the slice from "`print(k)`" are shown in bold

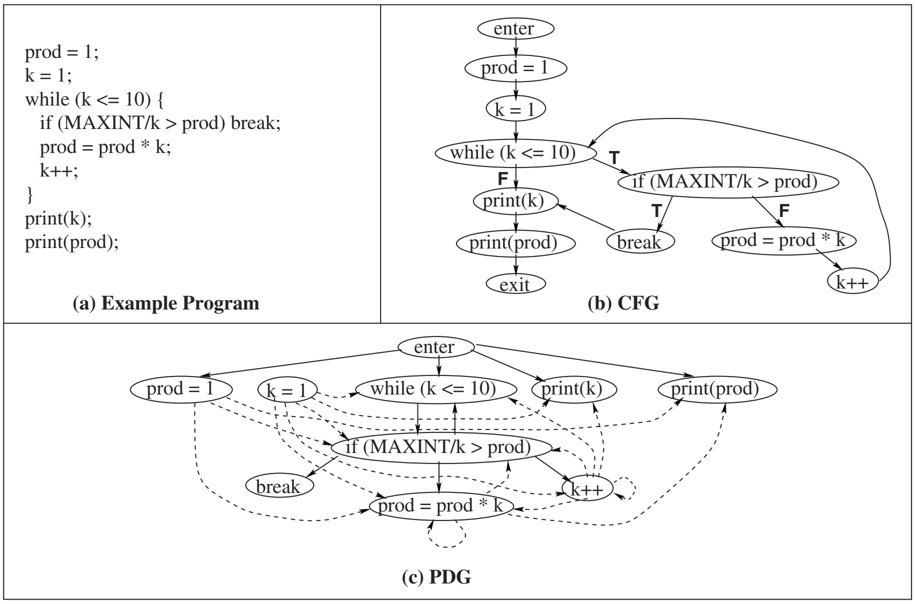
**Algorithm 1** *(Ottensteins' Algorithm for Building and Slicing the PDG)*

**Step 1:** *Build the program's CFG, and use it to compute data and control dependences: Node N is* ***data dependent*** *on node M iff M defines a variable x, N uses x, and there is an x-definition-free path in the CFG from M to N. Node N is* ***control dependent*** *on node M iff N postdominates one but not all of M's CFG successors.*

**Step 2:** *Build the PDG. The nodes of the PDG are almost the same as the nodes of the CFG: a special enter node, and a node for each predicate and each statement in the program; however, the PDG does not include the CFG's exit node. The edges of the PDG represent the data and control dependences computed using the CFG.*

**Step 3:** *To compute the slice from statement (or predicate) S, start from the PDG node that represents S and follow the data- and control-dependence edges backwards in the PDG. The components of the slice are all of the nodes reached in this manner.*

Example: Figure 1 shows a program that computes the product of the numbers from 1 to 10, its CFG, and its PDG. The nodes in the slice of the PDG from "`print(k)`" are shown using bold font. (For the purposes of control-dependence computation, an edge is added to the CFG from the *enter* node to the *exit* node; to avoid clutter, those edges are not shown in the CFGs given in this paper).



```
prod = 1;
k = 1;
while (k <= 10) {
  if (MAXINT/k > prod) break;
  prod = prod * k;
  k++;
}
print(k);
print(prod);
```

**Fig. 2.** Example program with a `break` statement, its CFG, and its PDG

## 2.3 Handling Jumps

Early slicing algorithms (including Weiser's and the Ottensteins') assumed a structured language with conditional statements and loops, but no jump statements (such as `goto`, `break`, `continue`, and `return`). Both [2] and [3] pointed out that if a CFG is used in which a jump statement is represented as a node with just a single outgoing edge (to the target of the jump), then no other node will be control dependent on the jump, and thus it will not be in the slice from any ot her node. For example, Figure 2(a) shows a modified version of the program from Figure 1, now including a `break` statement. Figures 2(b) and 2(c) show the program's CFG and the corresponding PDG. Note that in this PDG, there is no path from the `break` to "`print(k)`" or to "`print(prod)`", and therefore the `break` is (erroneously) not included in the slices from those two print statements even though the presence of the `break` can affect the values that are printed.

The solution proposed by [2] and [3] involves using an augmented CFG, called the ACFG, to build a dependence graph whose control-dependence edges are different from those in the PDG used by Algorithm 1. We will refer to the new dependence graph as the *APDG*, to distinguish it from the PDG.

**Algorithm 2** *(Building and Slicing the APDG)*

**Step 1:** *Build the program's ACFG. In the ACFG, jump statements are treated as pseudo-predicates. Each jump statement is represented by a node with two outgoing edges: the edge labeled true goes to the target of the jump, and the (non-executable) edge labeled false goes to the node that would follow the*

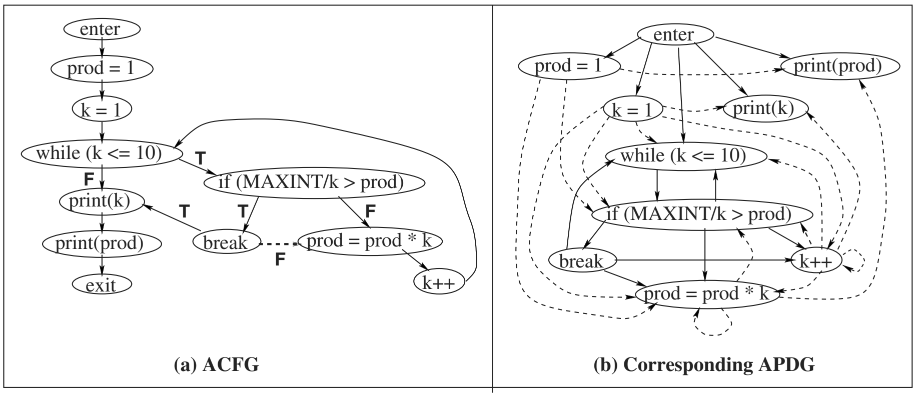


**Fig. 3.** ACFG and the corresponding APDG for the example program from Figure 2

*jump if it were replaced by a no-op. Labels are treated as separate statements; i.e., each label is represented in the ACFG by a node with one outgoing edge to the statement that it labels.*

**Step 2:** *Build the program's APDG. Ignore the non-executable ACFG edges when computing data-dependence edges; do not ignore them when computing control-dependence edges. (This way, the nodes that are executed only because a jump is present, as well as those that are not executed but would be if the jump were removed, are control dependent on the jump node, and therefore the jump will be included in their slices.)*

**Step 3:** *To compute the slice from node S, follow data- and control-dependence edges backwards from S as in Algorithm 1. A label* `L` *is included in a slice iff a statement "*`goto L`*" is in the slice.*

Example: Figure 3 shows the ACFG for the program in Figure 2(a), and the corresponding APDG. (The non-executable *false* edge out of the `break` in Figure 3(a) is shown using a dotted arrow.) Note that in Figure 3(b), there are control-dependence edges from the `break` to "`prod = prod * k`" and to "`k++`"; therefore, the `break` is (correctly) included in every slice that includes one of those two nodes.

## 3 Semantic Foundations for Slicing

In his seminal paper on program slicing [13], Weiser defined a slice of a program $P$ from point $S$ with respect to a set of variables $V$ to be any program $P'$ such that:

| Program | Intuitive Slice | Also Correct by Weiser's Definition |
|---|---|---|
| `[1] x = 2;` | | `[1] x = 2;` |
| `[2] y = 2;` | | `[2] y = 2;` |
| `[3] w = x * y;` | | |
| `[4] x = 1;` | `[4] x = 1;` | |
| `[5] y = 3;` | `[5] y = 3;` | |
| `[6] z = x + y;` | `[6] z = x + y;` | `[6] z = x + y;` |

**Fig. 4.** Example illustrating a shortcoming of Weiser's definition of a correct slice

– $P'$ can be obtained from $P$ by deleting zero or more statements.
– Whenever $P$ halts on input $I$, $P'$ also halts on input $I$, and the two programs produce the same sequences of values for all variables in set $V$ at point $S$ if it is in the slice, and otherwise at the nearest successor to $S$ that is in the slice.

One problem with this definition is that it can be inconsistent with the intuitive idea that the slice of a program from point $S$ is the set of program components that might *affect* $S$. For example, Figure 4 shows a program, the slice that a programmer would probably produce if asked to slice the program from statement `[6]` with respect to variable `z`, and another slice that is correct according to Weiser's definition, but that does not match our intuition about slicing. Furthermore, the requirement that a slice be an executable program may be too restrictive in some contexts (e.g., when using slicing to understand how a program works, or to understand the effects of a proposed change). In those cases, it might be more appropriate to define the slice of a program simply to be a subset of the program's components, rather than an executable projection of the program.

Given these observations, we propose to define the slice of program $P$ from component $S$ to be the components of $P$ that might have a semantic effect on $S$. But what does it mean for a statement or predicate $X$ to have a semantic effect on another statement/predicate $S$? To make that notion more precise, we consider what happens when a new program $P'$ is created by modifying $X$ or removing it from program $P$ as follows:

***X* is a normal predicate:** $P'$ is created by replacing $X$ with a different predicate that uses the same set of variables as $X$. (For example, in the program whose ACFG is shown in Figure 3, the predicate "`MAXINT/k > prod`" could be replaced by any other predicate that uses only variables `k` and `prod`, such as: "`k < prod`", or "`k != 0 && prod > 22`".)

***X* is a pseudo-predicate (a jump statement):** $P'$ is created by removing statement $X$ from $P$.

***X* is a non-jump statement:** $P'$ is created by replacing $X$ with a different statement that uses and defines the same sets of variables as $X$. (For example, in the program whose ACFG is shown in Figure 3, the statement "`prod = prod*k`" could be replaced by any other statement that uses only variables `prod` and `k`, and that defines variable `prod`, such as: "`prod = k + prod`", or "`prod = prod-k-4`".)

**Definition 0.** ***(Semantic Effect)**: X has a **semantic effect** on S iff there is some program $P'$ created by modifying or removing X from P as defined above, and some input I such that:*

- *Both P and $P'$ halt on I.*
- *The two programs produce a different sequence of values for some variable used at S.*

Note that the sequence of values produced for a variable used at $S$ can differ either because the two sequences are of different lengths, or because their $k^{th}$ values differ (for some $k$).

Definition 0 is similar to the definition of *finitely demonstrable semantic dependence* given by Podgurski and Clarke in [10]. However, that definition did not take jump statements into account: according to their definition, no program component is ever semantically dependent on a jump; therefore, if a correct slice from $S$ is defined to include all components on which $S$ is semantically dependent, jump statements will never be included in a slice. This is clearly contrary to one's intuition, and therefore is a shortcoming of the Podgurski/Clarke definition.

As usual with any interesting property of a program, determining which components have a semantic effect on a given component $S$, according to Definition 0, is an undecidable problem. Therefore, we must say that a (correct) slice of program $P$ from component $S$ is any superset of the components of $P$ that have a semantic effect on $S$.

Note that using Definition 0, statements `[4]` and `[5]` in the example program in Figure 4 (but not statements `[1]` and `[2]`) have a semantic effect on statement `[6]`. Therefore, a correct slice from statement `[6]` must include statements `[4]` and `[5]` (but not statements `[1]` and `[2]`), which is consistent with our intuition about that slice.

## 4 Representing Switch Statements

Consider a C switch statement of the form:

```
switch (E) {
   case e1: S1; break;
     ...
   case en: Sn; break;
   default: S;
}
```

Clearly, "`switch(E)`" should be represented in the CFG (and the ACFG) using a (normal) predicate node with $n + 1$ outgoing edges: one to each `case` including the `default`. If there were no `default`, the $n + 1^{st}$ edge should go to the first statement following the switch (because in C, if the value of the

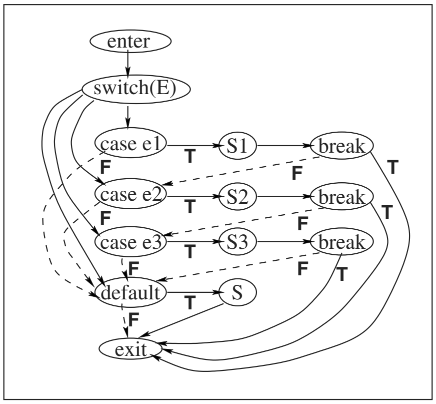


**Fig. 5.** ACFG for a switch statement

switch expression does not match any case label, and there is no `default` then execution continues immediately after the switch).

Now consider how to represent the case labels. One's initial intuition might be that they are similar to other labels in a program (the targets of `goto` statements). However, there is an important difference: if a program includes "`goto L1`", then label `L1` must be in the program, or it is not syntactically correct. If there is *no* "`goto L1`", then it doesn't matter whether label `L1` is in the program: its presence or absence has no semantic effect. However, these observations are not true of a case label. Removing a case label from a program never causes a syntax error, but *can* have a semantic effect. For example, if expression `E` in the code given above evaluates to `e1`, then statement `S1` will execute. However, if "`case e1`" is removed, then statement `S1` will not execute; instead, statement `S` will execute. Therefore, it makes sense for "`case e1`" to be in the slice from `S1` as well as in the slice from `S`.

This suggests that, like jumps, case labels should be represented using pseudo-predicates in the ACFG. The target of the outgoing *true* edge from a case-label node should be the first statement inside the `case`, and the target of the outgoing *false* edge should be the node that represents the default label if there is one, and otherwise the first statement that follows the switch (because if the case label is removed, and the switch expression matches that value, then execution proceeds with the first statement after the switch). The target of the outgoing *false* edge from the default case should always be the first statement that follows the switch.

Example: Figure 5 shows the ACFG for the switch statement given above (for $n = 3$).

| Code Fragment | Ideal Slice | Slice computed using Algorithm 2 |
|---|---|---|
| `switch (E) {`<br>`case e1: S1; break;`<br>`case e2: S2; break;`<br>`case e3:` $\boxed{\texttt{S3;}}$ `break;`<br>`}` | $\boxed{\texttt{switch (E)}}$ `{`<br>`case e1: S1; break;`<br>`case e2: S2;` $\boxed{\texttt{break;}}$<br>$\boxed{\texttt{case e3:}}$ $\boxed{\texttt{S3;}}$ `break;`<br>`}` | $\boxed{\texttt{switch (E)}}$ `{`<br>$\boxed{\texttt{case e1:}}$ `S1;` $\boxed{\texttt{break;}}$<br>$\boxed{\texttt{case e2:}}$ `S2;` $\boxed{\texttt{break;}}$<br>$\boxed{\texttt{case e3:}}$ $\boxed{\texttt{S3;}}$ `break;`<br>`}` |
| `switch (E) {`<br>`case e1: if (P) return;`<br>`break;`<br>`case e2:` $\boxed{\texttt{S;}}$<br>`}` | $\boxed{\texttt{switch (E)}}$ `{`<br>`case e1: if (P) return;`<br>$\boxed{\texttt{break;}}$<br>$\boxed{\texttt{case e2:}}$ $\boxed{\texttt{S;}}$<br>`}` | $\boxed{\texttt{switch (E)}}$ `{`<br>$\boxed{\texttt{case e1:}}$ $\boxed{\texttt{if (P)}}$ $\boxed{\texttt{return;}}$<br>$\boxed{\texttt{break;}}$<br>$\boxed{\texttt{case e2:}}$ $\boxed{\texttt{S;}}$<br>`}` |
| `if (P1) goto L1;`<br>`if (P2) goto L3;`<br>`goto L2;`<br>`L1:` $\boxed{\texttt{S;}}$<br>`L2: ...`<br>`L3: ...` | $\boxed{\texttt{if (P1)}}$ $\boxed{\texttt{goto L1;}}$<br>`if (P2) goto L3;`<br>$\boxed{\texttt{goto L2;}}$<br>$\boxed{\texttt{L1:}}$ $\boxed{\texttt{S;}}$<br>$\boxed{\texttt{L2:}}$ `...`<br>`L3: ...` | $\boxed{\texttt{if (P1)}}$ $\boxed{\texttt{goto L1;}}$<br>$\boxed{\texttt{if (P2)}}$ $\boxed{\texttt{goto L3;}}$<br>$\boxed{\texttt{goto L2;}}$<br>$\boxed{\texttt{L1:}}$ $\boxed{\texttt{S;}}$<br>$\boxed{\texttt{L2:}}$ `...`<br>$\boxed{\texttt{L3:}}$ `...` |

**Fig. 6.** Examples for which Algorithm 2 produces slices with extra components. The first column gives a code fragment, with one statement enclosed in a box. The second column shows the ideal slice from the boxed statement (according to Definition 0 given above in Section 3). The third column shows the slice computed using Algorithm 2

## 5 Motivation for a New Slicing Algorithm

Figure 6 gives three examples where Algorithm 2 (see Section 2.3) produces slices that include unwanted components. (In these examples, we assume that switch statements are represented in the ACFG as discussed above in Section 4.) The first column in Figure 6 gives a code fragment, with one statement enclosed in a box. The second column shows the ideal slice from the boxed statement (according to Definition 0 given above in Section 3). The third column shows the slice computed using Algorithm 2. The first two examples involve switches, while the third example involves only gotos.

Note that in the first example the slice from `S3` should include the `break` from the previous case, because the presence/absence of that `break` affects whether or not `S3` executes. In particular, consider what happens when expression `E` evaluates to `e2`. If the `break` is *not* in the program, `S3` executes, while if the `break` *is* in the program, `S3` does not execute.

In the second example, the slice from `S` should include neither "`if (P)`" nor "`return`". Whatever the value of predicate `P`, statement `S` will not execute (because either the `return` or the `break` prevents execution from "falling through" from "`case e1`" to "`case e2`"). Similarly, whether or not the `return` is in the program makes no difference since it is followed by the `break` (and thus `S` is always prevented from executing).

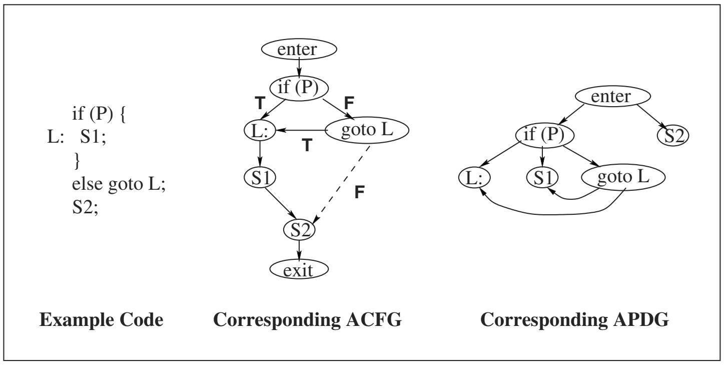


**Fig. 7.** Example in which a control dependence does not reflect a semantic effect. The APDG includes a control-dependence edge from "`if (P)`" to `S1`. However, "`if (P)`" cannot in fact affect the execution of `S1`; it always executes, regardless of whether `P` evaluates to *true* or to *false*

In all three examples, extra components are included in the slices computed using Algorithm 2 because of a chain of control-dependence edges. For instance, the APDG for the second example includes the following chain: `case e1` $\rightarrow$ `if (P)` $\rightarrow$ `return` $\rightarrow$ `break` $\rightarrow$ `case e2` $\rightarrow$ `S`. Thus, since Algorithm 2 follows all control-dependence edges backwards, all of those components are included in the slice from `S`[1]. In this example, each individual control-dependence edge represents a possible semantic effect: "`case e1`" has a semantic effect on "`if (P)`", which has a semantic effect on "`return`", which has a semantic effect on "`break`", which has a semantic effect on "`case e2`", which has a semantic effect on `S`. However, the backwards closure of the control-dependence relation starting from `S` yields a superset of the components that have a semantic effect on `S`; i.e., the "semantic-effect" relation is not transitive.

It is also possible to have an example in which even an individual control dependence (computed using the ACFG) does not reflect a semantic effect, as illustrated in Figure 7. In this example, the APDG includes a control-dependence edge from "`if (P)`" to `S1` because `S1` postdominates the *true* successor of the `if` in the ACFG, but does not postdominate its *false* successor (because the `goto`'s non-executable *false* edge bypasses `S1`). However, "`if (P)`" cannot in fact affect the execution of `S1`; it always executes, regardless of whether `P` evaluates to *true* or to *false*.

These examples motivate the need for a new definition of control dependence to avoid control-dependence edges like the one in Figure 7 that do not reflect a semantic effect. They also motivate the need for a new way to compute slices

[1] Furthermore, the entire backward closure from predicate `P` of the control- and data-dependence relations will be included in the slice computed by Algorithm 2, making it arbitrarily larger than the ideal slice.

that does not involve taking the transitive closure of the control-dependence edges, since, as discussed above, the semantic-effect relation is not transitive.

## 6 New Definition of Control Dependence and New Slicing Algorithm

Recall that the definition of control dependence used in Algorithm 1 is as follows:

**Definition 1.** ***(Original Control Dependence)****: Node N is* ***control dependent*** *on node M iff N postdominates, in the CFG, one but not all of M's CFG successors.*

To permit control dependence on jumps, Algorithm 2 replaces "CFG" with "ACFG" in the definition of control dependence:

**Definition 2.** ***(Augmented Control Dependence)****: Node N is* ***control dependent*** *on node M iff N postdominates, in the ACFG, one but not all of M's ACFG successors.*

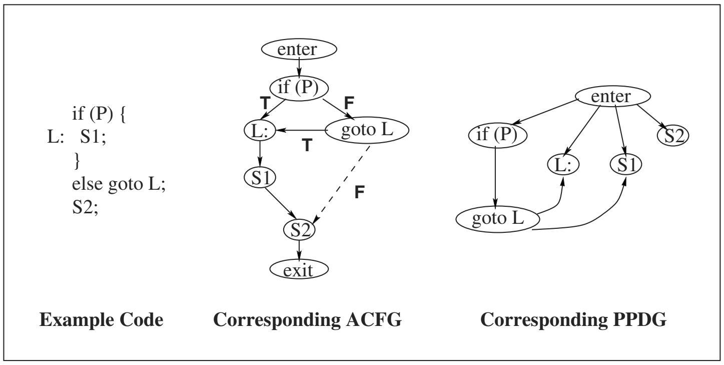
Unfortunately, as illustrated in Figure 7, Definition 2 is too liberal; it can cause a spurious control dependence of $N$ on $M$ due to the presence of an intervening pseudo-predicate. For example, in the ACFG in Figure 7, node `S1` fails to postdominate the *false* successor of the `if` only because of the non-executable edge from "`goto L1`" to `S2`. Since the execution of `S1` *is* affected by the presence/absence of the `goto` it *should* be considered to be control dependent on the `goto`; however, (as noted previously), `S1` will execute regardless of the value of predicate `P`, and therefore it should *not* be considered to be control dependent on the `if`. So in this case, the actual influence of "`goto L1`" on statement `S1` causes an apparent (but spurious) influence of "`if (P)`" on `S1`.

The solution to this dilemma is to replace only the *second* instance of "CFG" with "ACFG" in Definition 1:

**Definition 3.** ***(Control Dependence in the Presence of Pseudo-predicates)****: Node N is* ***control-dependent*** *on node M iff N postdominates, in the CFG, one but not all of M's ACFG successors.*

We will refer to a dependence graph that includes control-dependence edges computed using Definition 3 as a PPDG (pseudo-predicate PDG) to distinguish them from the PDGs whose control-dependence edges are computed using Definition 1, and the APDGs whose control-dependence edges are computed using Definition 2.
Example: The program and ACFG from Figure 7 are given again in Figure 8, with the corresponding PPDG. Note that neither label `L` nor statement `S1` is control dependent on "`if (P)`".

Definition 3 addresses the problem of control-dependence edges that do not reflect semantic effects. The next problem that needs to be addressed is the fact that even when every control-dependence edge does represent a semantic effect, the backward closure of control-dependence edges from a node $S$ may include



```
    if (P) {
L:   S1;
    }
    else goto L;
    S2;
```

**Fig. 8.** Example code and ACFG from Figure 7 with the corresponding PPDG

nodes that have no semantic effect on $S$. For example, consider again the PPDG in Figure 8. If the slice from node `S1` includes all nodes reached by following control-dependence edges backwards, then "`if (P)`" will (erroneously) be in the slice because of the chain of control-dependence edges: `if (P)` $\rightarrow$ `goto L` $\rightarrow$ `S1`.

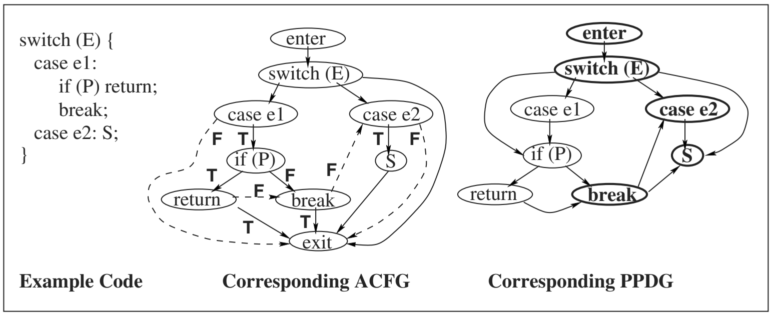
To address this problem, we need the following definition:

**Definition 4.** *(IPD): The **immediate post dominator (IPD)** of a set of ACFG nodes is the node that is the least-common ancestor of that set of nodes in the CFG's postdominator tree.*

Consider a (normal or pseudo) predicate $P$, with ACFG successors $n_1 ... n_k$, and let $D = \text{IPD}(n_1 ... n_k)$. Intuitively, $P$ may affect the execution of a program component $S$ only if there is a path in the CFG from one of $P$'s ACFG successors to $S$ that does not include node $D$. (If there is such a path, we say that $S$ ***is controlled by*** $P$.) The value of $P$ (for a normal predicate), or its presence/absence (for a pseudo-predicate) determines which of its ACFG successors is executed. The execution of the nodes along the paths from those ACFG successors to $D$ are also affected by the value (or presence/absence) of $P$. However, since whenever $P$ is executed, execution will always reach $D$ (barring an infinite loop or other abnormal termination), the execution of nodes "beyond" $D$ are not affected by $P$.

As discussed above, following control-dependence edges backwards from $S$ in the PPDG can cause extra nodes to be included in the slice from $S$. In terms of the "is controlled by" relation, this is because there may be a chain of control-dependence edges in the PPDG from a predicate $P$ to $S$, yet $S$ is not controlled by $P$. However, we have proved the following Theorem [8]:

**Theorem:** Node $S$ is controlled by (normal or pseudo) predicate $P$ iff there is a chain of control-dependence edges in the PPDG: $P \rightarrow M_1 \rightarrow M_2 \rightarrow ... \rightarrow M_k \rightarrow S$, such that every $M_i$ in the chain is a normal predicate node. (Note that there may also be no $M_i$'s at all; i.e., there may be a single control-dependence edge $P \rightarrow S$.)



```
switch (E) {
  case e1:
      if (P) return;
      break;
  case e2: S;
}
```

**Fig. 9.** The code, ACFG, and PPDG for the second example in Figure 6. The PPDG nodes in the slice from `S`, computed using Algorithm 3, are shown in bold

This Theorem tells us that it is not necessary to follow control-dependence edges back from a pseudo-predicate; for any predicate $P$ such that there is a node $S$ in the slice that is controlled by $P$, $P$ will be picked up by following chains backwards only from normal predicates.

The new algorithm for building and slicing the PPDG is given below.

**Algorithm 3** *(Building and Slicing the PPDG)*

**Step 1:** *Build the ACFG as described above for Algorithm 2.*

**Step 2:** *Build the PPDG: Ignore the non-executable ACFG edges when computing data-dependence edges; compute control-dependence edges according to Definition 3.*

**Step 3:** *To compute the slice from node $S$, include $S$ itself and all of its data- and control-dependence predecessors in the slice. Then follow backwards all data-dependence edges, and all control-dependence edges whose targets are not pseudo-predicates; add each node reached during this traversal to the slice. Include label `L` in the slice iff a statement "`goto L`" is in the slice.*

Examples: (1) Using Algorithm 3, the slice from `S1` of the program in Figure 8 would include the nodes for `S1`, "`goto L`", `L`, and the *enter* node. It would not include the node for "`if (P)`" because, since "`goto L`" is a pseudo-predicate, its incoming control-dependence edge would not be followed back to the `if` node. (2) Figure 9 shows the code, ACFG, and PPDG for the second example in Figure 6. Bold font is used to indicate the nodes that would be in the slice from statement `S` computed using Algorithm 3. Note that "`case e1`", "`if (P)`", and "`return`" are correctly omitted from the slice.

## 6.1 Complexity

The time required for Algorithm 3 includes the time to build the PPDG and the time to compute a slice. Like previous slicing algorithms that use a dependence

| | lines of source code | number of APDG/PPDG nodes | number of slices | Av. slice size (# of nodes) | |
|---|---|---|---|---|---|
| | | | | Alg 2 | Alg 3 |
| gcc.cpp | 4,079 | 16,784 | 1,932 | 11,693 | 11,670 |
| byacc | 6,626 | 21,239 | 468 | 2,119 | 2,110 |
| CADP | 12,930 | 35,965 | 499 | 7,921 | 7,905 |
| flex | 16,236 | 31,354 | 1,716 | 8,150 | 8,082 |

**Fig. 10.** Information about the C programs used in the experiments

graph, the time for slicing itself is extremely efficient, requiring only time proportional to the size of the slice (the number of nodes and edges in the sub-PPDG that represents the slice). The only difference in the time required to build the PPDG as compared to the time required to build the APDG is for the computation of control dependences. Computing control dependences can be done for both the APDG and the PPDG in time $O(E + C)$, where $E$ is the number of edges in the ACFG and $C$ is the number of control-dependence edges. However, $C$ may be different for the APDG and PPDG. For example, in Figure 9, the PPDG includes edges from "`switch (E)`" to "`if (P)`" and to `S` that would not be in the corresponding APDG. Figures 7 and 8 illustrate control-dependence edges that are in the APDG but not in the PPDG.

## 7 Experimental Results

To evaluate our work, we implemented Algorithms 2 and 3, and used each of them to compute slices in four C programs (information about the programs, the number of slices taken in each, and the average sizes of those slices is given in the table in Figure 10). Slices were taken from all of the nodes that could be reached by following one control-dependence edge forward from a node representing a switch case, and then following five data-dependence edges forward. This ensured that every slice would include a switch, but (by starting further along the chain of data dependences) avoided, for example, slices that would include *only* switch cases and breaks.

More details about the experimental results are given in the tables in Figures 11 and 12. Figure 11 presents information about the differences in the sizes of the individual slices taken using the two algorithms. The first column gives the number of cases where the two algorithms produced slices of exactly the same size. The other columns give the number of cases where the slice produced by Algorithm 2 was larger than the slice produced by Algorithm 3; the second column gives the number of cases where the size difference was between 1 and 10, the third column gives the number of cases where the size difference was between 11 and 20, etc.

Figure 12 presents information about how much the use of Algorithm 3 reduced the sizes of the slices. The first column gives the number of cases where there was no reduction in slice size (a 0% reduction). The other columns give the number of cases where the reduction in size falls within the range specified

| | 0 | 1-10 | 11-20 | 21-30 | 31-40 | 41-50 | 51-60 | 61-70 | 71-80 | 81-90 |
|---|---|---|---|---|---|---|---|---|---|---|
| gcc.cpp | 2 | 0 | 48 | 1881 | 0 | 1 | 0 | 0 | 0 | 0 |
| byacc | 0 | 229 | 239 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| CADP | 18 | 152 | 160 | 169 | 0 | 0 | 0 | 0 | 0 | 0 |
| flex | 0 | 0 | 5 | 127 | 48 | 41 | 8 | 79 | 1405 | 3 |

**Fig. 11.** Differences in slice sizes using the two algorithms. Each entry gives the number of cases where the size difference in the slices produced by Algorithms 2 and 3 falls into the range given at the top of the column

| | 0% | 5% | 10% | 15% | 20% | 25% | 30% | 35% |
|---|---|---|---|---|---|---|---|---|
| gcc.cpp | 2 | 1918 | 3 | 1 | 8 | 0 | 0 | 0 |
| byacc | 0 | 438 | 18 | 7 | 5 | 0 | 0 | 0 |
| CADP | 18 | 481 | 0 | 0 | 0 | 0 | 0 | 0 |
| flex | 0 | 1572 | 0 | 5 | 13 | 52 | 66 | 8 |

**Fig. 12.** Percent reduction in slice sizes achieved using Algorithm 3. Each entry gives the number of cases where the reduction in slice size falls into the range specified by the previous and current column headings

by the previous and current column headers. For example, the second column gives the number of cases where there was a size reduction greater than 0% and less than or equal to 5%; the third column gives the number of cases where there was a size reduction greater than 5% and less than or equal to 10%.

Note that in almost all cases Algorithm 3 did produce smaller slices than Algorithm 2. Although this led to only a small reduction in the total size of the slice in most cases, there were some cases in both gcc.cpp and byacc where Algorithm 3 provided reductions in slice sizes of more than 15%, and some cases in flex where it provided reductions in slice sizes of more than 30%.

## 8 Related Work

***Choi-Ferrante:*** The paper by Choi and Ferrante [3] that presents Algorithm 2 also includes a second algorithm: Given a node $S$, it starts with the slice from $S$ computed using Algorithm 1, then adds `goto` statements to the slice to form a program that will always produce the same sequence of values for the variables used at $S$ as the original program. This technique may produce smaller slices than those produced using Algorithm 2. However, the `goto`s that are added are not necessarily in the original program; therefore, it satisfies neither Weiser's definition of a correct slice, nor Definition 0 from Section 3.

***Agrawal:*** Agrawal [1] also gives an algorithm that involves adding jump statements to the slice computed using the standard PDG, but the statements that he adds *are* from the original program. He states that this algorithm produces the same results as Algorithm 2; however, no proof is provided.

***Harman-Danicic:*** More recently, Harman and Danicic [5] have defined an extension to Agrawal's algorithm that produces smaller slices by using a refined criterion for adding jump statements (from the original program) to the slice computed using Algorithm 1. When applied to programs without switches, it may or may not produce slices that satisfy Definition 0. This is because their algorithm includes some nondeterminism: when there are cycle-free paths from a predicate to its immediate-postdominator both via its *true* and its *false* branches, then the jump statements along either of the paths can be chosen to be in the slice. Unfortunately, when applied to programs with switch statements, this algorithm can be as imprecise as Algorithm 2. For example, when used to slice the switch statement in the first example in Figure 6, it produces exactly the same slice as Algorithm 2. Another disadvantage of this algorithm as compared to ours is that the worst-case time to compute a slice can be quadratic in the size of the CFG, while our algorithm is linear in the size of the computed slice.

***Sinha-Harrold-Rothermel:*** In [12], Sinha, Harrold, and Rothermel discuss interprocedural slicing in the presence of arbitrary interprocedural control flow; e.g., statements (like `halt`, `setjmp-longjmp`) that prevent procedures from returning to their call sites. That issue is orthogonal to the one addressed here (better slicing of programs with jumps and switches); thus, the two approaches can be combined to handle programs with arbitrary interprocedural control flow as well as jumps and switches.

***Schoenig-Ducassé:*** An algorithm for slicing Prolog programs is given in [11]. It is observed that some arguments of some clauses (referred to as *failure positions*) must be included in a slice because they may cause a relevant goal to fail (so excluding them from the slice might change the control-flow of the program). This idea is related to our notion of the semantic effect of a jump statement (because in that case too, removing the statement might change the control-flow of the program).

## 9 Summary

We have provided a new definition for a “correct” slice, a new definition for control dependences, and a new slicing algorithm. The algorithm has essentially the same complexity as previous algorithms that compute slices using program dependence graphs, and is more precise than previous algorithms when applied to programs with jumps and switch statements.

The motivation for this work was the observation that slices of code with switch statements computed using the approach to handling jumps proposed by [2,3] (as implemented in the CodeSurfer [6] programming tool) often include many extra components, which is confusing to users of the tool. We expect that the new approach will have an important practical benefit (to users of slicing tools) as well as being an interesting theoretical advance.

**Acknowledgments**

This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435.

## References

1. H. Agrawal. On slicing programs with jump statements. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 302–312, June 1994.
2. T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Lecture Notes in Computer Science*, volume 749, New York, NY, November 1993. Springer-Verlag.
3. J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. on Programming Languages and Systems*, 16(4):1097–1113, July 1994.
4. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.
5. M. Harman and S. Danicic. A new algorithm for slicing unstructured programs. *Jrnl. of Software Maintenance*, 10(6):415–441, Nov./Dec. 1998.
6. `http://www.codesurfer.com`.
7. `http://www.infosun.fmi.uni-passau.de/st/staff/krinke/slicing/`.
8. S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. Technical Report TR-1429, Computer Sciences, University of Wisconsin-Madison, 2001.
9. K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 177–184, 1984.
10. A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. on Software Engineering*, 16(9):965–979, September 1990.
11. S. Schoenig and M. Ducassé. A backward slicing algorithm for Prolog. In *Lecture Notes in Computer Science*, volume 1145, pages 317–331, New York, NY, 1996. Springer-Verlag.
12. S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Int. Conf. on Software Engineering*, pages 432–441, May 1999.
13. M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.