Slicing Unconditional Jumps with Unnecessary Control Dependencies

Carlos Galindo[0000−0002−3569−6218], Sergio Pérez[0000−0002−4384−7004], and Josep Silva[0000−0001−5096−0008]

VRAIN
Universitat Politècnica de València
Camí de Vera s/n
E-46022 València, Spain
{cargaji,serperu,jsilva}@dsic.upv.es

Abstract. Program slicing is an analysis technique that has a wide range of applications, ranging from compilers to clone detection software, and that has been applied to practically all programming languages. Most program slicing techniques are based on a widely extended program representation, the System Dependence Graph (SDG). However, in the presence of unconditional jumps, there exist some situations where most SDG-based slicing techniques are not as accurate as possible, including more code than strictly necessary. In this paper, we identify one of these scenarios, pointing out the cause of the inaccuracy, and describing the initial solution to the problem proposed in the literature, together with an extension, which solves the problem completely. These solutions modify both the SDG generation and the slicing algorithm. Additionally, we propose an alternative solution, that solves the problem by modifying only the SDG generation, leaving the slicing algorithm untouched.

Keywords: Program analysis, Program slicing, Unconditional jumps

1 Introduction

Program slicing [20, 18] is a technique for program analysis and transformation whose main objective is to extract from a program the set of statements that affect a given set of variables in a specific statement, the so-called *slicing criterion*. The programs obtained with program slicing are called *slices*, and they are used in many areas such as debugging [1], program specialization [2], software maintenance [7], code obfuscation [13], etc.

There exist several algorithms and data structures to represent programs that can be used to compute slices, but the most efficient and broadly used data structure is the *system dependence graph* (SDG), introduced by Horwitz et al. [9]. It is computed from the program's source code, and once built, a slicing criterion is chosen and mapped to the graph, that is then traversed with the algorithm proposed in [9] to compute the corresponding slice.

The SDG is the result of assembling a set of graphs that represent information about a program. Figure 1 depicts how the SDG is built using the control-flow graph (CFG) as the starting graph. First, using the CFG of each function definition in the code, two different graphs are built: (i) the control dependence graph (CDG) [6] and (ii) the data dependence graph (DDG) [19, 6]. The union of both graphs results in the program dependence graph (PDG) [14, 6], which represents all data and control dependencies inside a concrete function. Finally, PDG's function calls, definitions and their parameters are linked with interprocedural arcs, generating the final SDG. The SDG can be traversed from a slicing criterion to produce a slice in linear time with the algorithm proposed in [9].

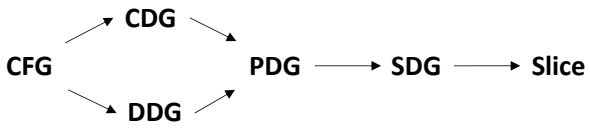

Fig. 1. Sequence of graphs generated to build the SDG.

As all the aforementioned graphs conforming the SDG represent different relationships of the program, an improvement in the accuracy of these graphs results in a direct impact on the accuracy of the SDG. Throughout the years, the SDG has been augmented with different dependencies, and several techniques have been defined to properly represent complex situations: interprocedural alternatives to compute executable slices [4], extensions of the CFG to represent interprocedural control dependencies [17], object-oriented language representations and slicing [12], or program slicing in concurrent environments [10, 5] are some examples of the evolution of the SDG.

For the purpose of this paper, we are interested in the evolution of the unconditional control flow treatment for program slicing. In this specific area, the initial proposal was the one introduced by Ball and Horwitz [3]. In their work, the authors considered a simplified language with scalar variables and constants, assignment statements, jump statements (`goto`, `break`, `halt`, etc.), conditional statements (`if-then`, `if-then-else`), and loops (`while` and `repeat`). Despite the simplicity of the given programming language, the ideas proposed can be applied to any kind of unconditional jumps present in other programming languages. In this paper, we provide examples using the `break` statement in the Java programming language, even though the problem presented and its solution can be applied to any statement that represents an unconditional jump. The following example illustrates the problem identified by Ball and Horwitz after their proposal.

Example 1 (Unconditional jump subsumption [3]). Consider the Java method shown below on the left-hand side:

```
1  public void f() {
2    while (X) {
3      if (Y) {
4        if (Z) {
5          A;
6          break;
7        }
8        B;
9        break;
10     }
11     C;
12   }
13   D;
14 }
```

Original program

```
1  public void f() {
2    while (X) {
3      if (Y) {
4        if (Z) {
5
6          break;
7        }
8
9        break;
10     }
11     C;
12   }
13
14 }
```

SDG slice

```
1  public void f() {
2    while (X) {
3      if (Y) {
4
5
6
7
8
9        break;
10     }
11     C;
12   }
13
14 }
```

Minimal slice

This method contains a `while` statement, from which the execution may exit naturally or through any of the `break` statements. To represent the rest of statements and conditional expressions, uppercase letters are used; and, for simplicity, we can assume that there are no data dependencies between them.

Now consider statement `C` as the slicing criterion: each input that produces a computation in the original program that reaches `C` must produce a computation in the slice that also reaches `C`. Note that `C` is only executed when `X` is `true` and `Y` is `false`.

The code in the centre displays the computed slice by Ball and Horwitz's approach; the code on the right-hand side is the minimal slice. As can be observed, the `break` in line 6 and its surrounding `if` statement (`if (Z)`) have been unnecessarily included in the slice, since the evaluation of `Z` does not influence the execution of a `break` after being the `Y` statement evaluated to `true`. Their inclusion would not be specially problematic, if it were not for the condition of the `if` statement (`Z`), which may include extra data dependencies that are unnecessary in the slice and that may lead to include other unnecessary statements, making the slice even more imprecise.

The rest of the paper is structured as follows: Section 2 illustrates the rationale behind the problem shown in Example 1, detailing how dependencies are generated, identifying when the problem shows up, and describing the solution proposed by Kumar and Horwitz in [11], where the authors introduced changes in two steps of the process shown in Figure 1. Section 3 proposes an alternative solution that is simpler and does not need to change the slicing algorithm, lowering the time complexity while preserving completeness at all times. Section 4 explains the problem in presence of `switch` statements and how to represent them to solve the problem. Section 5 outlines our implementation of the proposed solution. Finally, Section 6 concludes the article outlining the main contributions.

2 Unconditional jumps and the PPDG

To keep the paper self-contained, we start with the definition of control flow graph.

Definition 1 (Control-flow graph). *Given a program P, the control flow graph of P is a graph (N, A) where N is a set of nodes that contains one node for each statement in the program, and A are arcs that represent the execution flow between the nodes:*

Statement node. *Any statement that is not a conditional jump. These nodes have one outgoing edge pointing to the next statement of the program.*

Predicate node. *Any conditional jump statement, such as* `if`*,* `while`*, etc. These nodes have two outgoing edges labelled true and false, leading to the statements that would be executed regarding the condition evaluation.*

The CFG of the `Original program` in Example 1 is shown in Figure 2 (left), where we can ignore the dashed red arcs for now, since they are not part of the CFG. In this graph, all nodes with just one outgoing arc represent statements, while all nodes with two outgoing arcs labeled with T or F represent predicates. In the graph, unconditional jumps, such as `break` are represented with a node whose outgoing arc leads to the statement that will be executed after the jump. Other representations of unconditional jumps, such as representing them with a single arc connecting the previous statement with the jumps' target are inadequate for program slicing, as we require a mapping from each statement in the source code to a node in each graph.

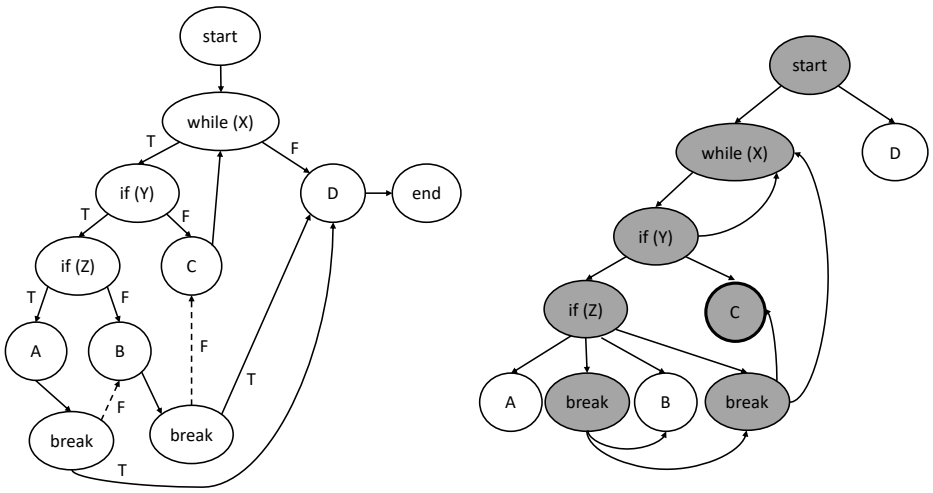

Fig. 2. ACFG (left) and CDG (right) of the code in Example 1.

The control flow graph is the basis to calculate control dependencies in a program and, thus, the control dependence graph.

Definition 2 (Control dependence). *Let G be a CFG. Let X and Y be nodes in G. A node Y post-dominates a node X in G if every directed path from X to*

the End node passes through Y. Node Y is control dependent *on node X if and only if Y post-dominates one but not all of X's CFG successors.*

Definition 3 (Control dependence graph). *Given a program P and its associated CFG $G_{CFG} = (N, A)$, the* control dependence graph *(CDG) of P is a graph $G_{CDG} = (N, A')$ where $(x, y) \in A'$ if and only if node $y \in N$ is control-dependent on node $x \in N$.*

Unconditional jump statements distort the usual understanding of control dependence, and they invalidate the standard representation of control dependencies in the CDG. Example 2 shows that the standard definition of control dependence is insufficient in presence of unconditional jumps.

Example 2 (Control dependencies induced by unconditional jumps). Consider the following code on the left-hand side and the slicing criterion x in the last line.

```
1  x = 0;
2  while (true) {
3    x++;
4    if (x>10)
5      break;
6  }
7  print(x);
```

Original program

```
1  x = 0;
2  while (true) {
3    x++;
4    if (x>10)
5
6  }
7  print(x);
```

Wrong slice

The slice of this code is the whole code (everything is needed to reach the slicing criterion). Nevertheless, according to Definition 2, the `break` statement in line 5 does not control any other statement, that is, no statement depends on the `break` statement. Therefore, the (wrong) slice computed with the standard definition of control dependence would be the code on the right. This is an infinite loop that never reaches the slicing criterion. Clearly, the execution of `print(x)` is in some way controlled by the execution of `break` and, thus, unconditional jumps induce some kind of control dependencies that are not captured in Definition 2.

To deal with this problem (i.e. unconditional control flow statements), Ball and Horwitz [3] proposed a modification of the CFG in presence of unconditional control flow statements, which result in a CDG with augmented dependencies. This approach is the most popular one and the one used in most of the subsequent literature [15, 11, 16]. The main modification applied to the CFG consists in the introduction of a third category of nodes in the definition of the CFG:

Pseudo-predicates. Unconditional jumps (i.e. `break`, `goto`, `return`[1], etc.) are treated like predicates, where the outgoing edge labelled *false* is marked as non-executable—because there is no possible execution where such edge would be possible, according to the definition of the CFG [8]. For unconditional jumps, the *true* edge leads to the statement at the jump destination, and the *false* edge to the statement that would be executed if the jump was skipped.

The graph obtained from adding the *false* arcs to the pseudo-predicate nodes of a CFG is called the *Augmented CFG* (ACFG). As a consequence of the appearance of pseudo-predicate nodes, in an ACFG every statement between an unconditional jump and its destination is control-dependent on it (see Definition 2), as can be seen in Example 3.

Example 3 (Control dependencies generated by unconditional jumps). Consider again the ACFG in Figure 2 (left), which represents the code in Example 1. Here, solid arrows represent edges that come out from statements, predicates, and *true* pseudo-predicate branches; and dashed red arrows represent the non-executable (*false*) branches of pseudo-predicates. When we transform this ACFG to a CDG, we obtain the CDG in Figure 2 (right), where the slice with respect to variable `C` is represented with grey nodes.

Even though Ball and Horwitz solved the exposed problem with the definition of the ACFG, there was still a problem they were not able to solve. This problem is represented in the code of Example 1. It appears when there are two different unconditional jumps with the same jump destination. Due to the *false* pseudo-predicate arcs in the ACFG, all the statements between the first unconditional jump and the second one become directly control-dependent on the first jump, including the second one. Similarly, all the statements located between the second jump and the destination statement become directly control-dependent on the second jump. As a result of the transitive dependence, when any statement between the second jump and the destination statement is required, the inclusion of both unconditional jump statements in the slice is unavoidable. The inclusion of the first jump statement will increase the size of the slice with all its dependencies, leading to an imprecise slice. The solution proposed in [3] is complete, but not as accurate as it was expected to be.

Ball and Horwitz were aware of the aforementioned problem and, some years later, Kumar and Horwitz proposed a solution in [11]. Their solution was based on two main modifications:

1. **A new definition of control dependence in the presence of pseudo-predicates**. *"Node Y is control-dependent on node X if and only if Y post-dominates, in the CFG, one but not all of X's ACFG successors"*. The resulting graph was called the pseudo-predicate PDG (PPDG).
2. **A new slicing algorithm**. The new algorithm established some restrictions in the slicing traversal. *"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."*

By the introduction of these novelties, the accuracy of the slice was improved, since it is not possible to add in the slice two pseudo-predicate nodes that jump

[1] The target of the jump in a `return` statement is the *End* or *Exit* node of the procedure it's in, from which control will be handed back to the previous procedure in the call stack.

to the same destination unless one of them is the slicing criterion itself. This approach solved the problem of Example 1, proposed in [3].

3 Alternative solution: unnecessary control dependencies

In this section, we propose an alternative solution to the unconditional jump problem shown in the previous section. The key idea of our approach is to identify which edges of the CDG are responsible for the inaccurate slices and define a method to avoid building them in the graph generation process.

To properly reason about the accuracy of our approach, we provide a formal definition of slicing criterion and slice.

Definition 4 (Slicing criterion). *Let P be a program. A slicing criterion C of P is a tuple $\langle s, v\rangle$ where s is a statement in P and v is a set of variables that are used or defined in s.*

Definition 5 (CDG slice). *Given a CDG $G = (N, A)$ and a slicing criterion $\langle s, v\rangle$, where $n \in N$ represents s in G, a* CDG slice *of n is a subgraph $G' = (N', A')$ such that:*

1. $N' \subseteq N$.
2. $\forall n' \in N'$, *n is control dependent on n' and n' is needed to execute n the same number of times as in G (the original program).*
3. $A' = \{(x, y) \in A \mid x, y \in N'\}$.

The standard slicing algorithm, denoted $slice(G, C)$, collects all nodes that are reachable from the node in G associated with the slicing criterion C traversing backwards the CDG arcs.

We have identified a general situation in which some control dependencies should be omitted. If those control dependencies are removed from the CDG, then the standard slicing algorithm is still complete and precision is kept the same or improved. Consider a CDG G with two unconditional jump statements x and y that jump to the same destination, with an arc (x, y) in G. There exists a CDG G' with the same set of nodes and a set of arcs obtained by deleting all the control arcs in G with y as target, that produces more accurate program slices.

Theorem 1. *Let $G = (N, A)$ be a CDG. Let $x \in N$ be any unconditional jump statement. Let $y \in N$ be an unconditional jump statement without any variable use or definition that jumps to the same destination as x. Let $G' = (N, A')$ where $A' = (A \setminus \{(w, y) \mid w \in N\})$. For all slicing criteria C, $slice(G', C)$ is a CDG slice.*

Proof. We prove the theorem by means of a generic code that captures all possible scenarios that can happen under the conditions of the theorem. We consider two unconditional jump statements, x as the first jump statement and y as the second one. First, x and y cannot be sequential statements because in that case

y would be dead code. This forces us to enclose x inside a conditional structure. As y does not define or use any variable, we add the statement s_1 and place an external conditional structure to also prevent it to be dead code. This generic code is depicted in Figure 3 (left). Any statement or groups of them added to this code before or after x or y would produce a similar topology that would not affect the proof. The reason is that any statement represented by a set of nodes has only one successor in the CFG and can never be the source of a control dependence (see Section 2.3 in [3]).

We graphically illustrate this proof by means of figures 3 and 4. Figure 3 represents the ACFG (centre) of the aforementioned code with the ACFG extra arcs represented with dashed red arrows, and its associated CDG (right). Figure 4 represents the same CDG removing two control dependence arcs (dashed red arcs). The figure represents two program slices with respect to two different slicing criteria: x (left) and s_1 (right).

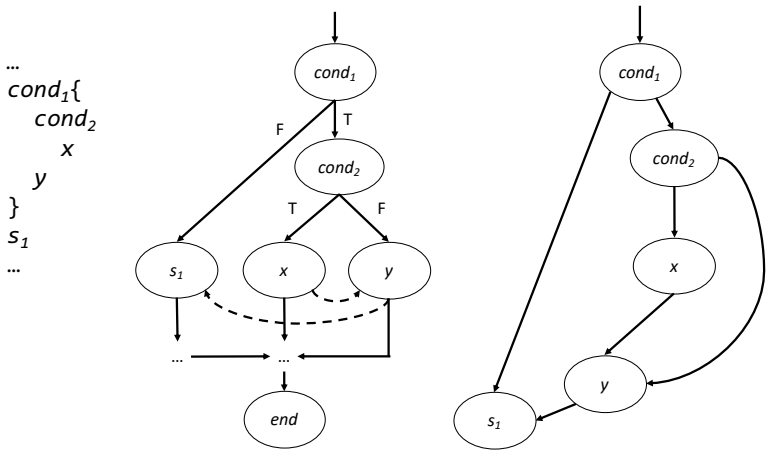

Fig. 3. Piece of code (left), its ACFG (centre) and its associated CDG (right).

We distinguish two possible scenarios according to the slice computed by $slice(G', n)$:

(i) $y \notin slice(G', n)$ (Figure 4, left). In this case, node y is not needed to execute n and, thus, the removal of the arcs that end in y do not affect the computation of $slice(G', n)$ because they are never traversed. Therefore, all nodes needed to execute n belong to the slice (condition 2 in Definition 5) and also all arcs induced by them are kept in the slice (condition 3 in Definition 5). Hence, $slice(G', n)$ is a CDG slice.

(ii) $y \in slice(G', n)$ (Figure 4, right). First, according to Definition 4, node y cannot be selected as slicing criterion, as it does not define or use any variables of the program according to the theorem conditions imposed on

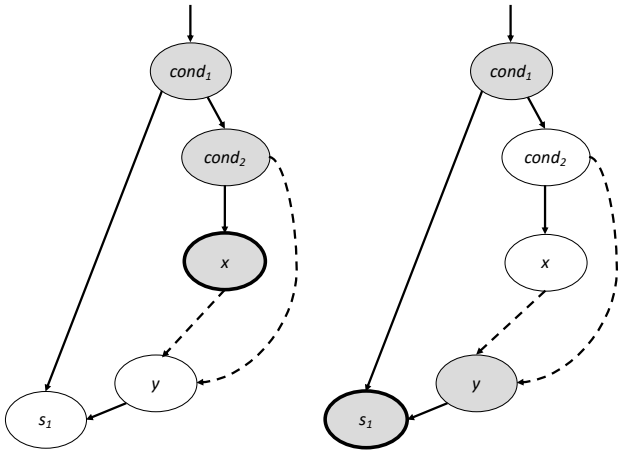

Fig. 4. CDG of our approach and CDG slices w.r.t. x (left) and s_1 (right).

y. Then, because no data dependence exists on y, the only possibility to include y in $slice(G', n)$ is because some statement between y and the jump destination of y is included in the slice (s_1 in our graph in Figure 4 (right)). Because of that, there is an execution path where y affects the execution of this statement. In the case that $cond_1$ was a loop, y would be control dependent on $cond_1$ itself, including $cond_1$ in $slice(G', n)$ but, in this case, we would obtain the same result because s_1 is also control dependent on $cond_1$ and thus, included in $slice(G', n)$.

We have two possible scenarios to execute n (see the ACFG in Figure 3 (centre)):

- s_1 is executed. Then, $cond_1$ is *false* and $cond_2$, x, and y are not executed (they can be excluded from $slice(G', n)$).
- Either x or y are executed. As the result of executing x and y is functionally the same (the program execution continues at the destination of y), there is no difference between taking one path of $cond_2$ or another. Therefore, $cond_2$, x and y can be replaced by y without modifying the behaviour of the program; making the control dependency arcs from $cond_2$ and x to y unnecessary.

In the three cases, the removal of the arcs that end in y ensure that the three conditions in Definition 5 hold. Thus, $slice(G', n)$ is a CDG slice.

□

Theorem 1 proves that slices produced with this solutions are complete, as their result is a CDG slice. Additionally, due to the fact that the same nodes, but fewer arcs exist, all slices produced from G' will be equal or smaller than those generated from the original CDG G, thus guaranteeing that our solution is, at least, as correct as the previous solution. Finally, a CDG with fewer arcs

means that the input size for the slicing algorithm is smaller, therefore lowering the time required to slice the graph once generated.

Algorithm 1 CDG Generation Algorithm

Input: An ACFG $G = (N, A)$.
Output: A CDG $G' = (N', A_c)$.
1: $A_c = genControlArcs(G)$
2: **for all** $(n_s, n_e) \in A_c$ **do**
3: **if** $(n_s, n_e \in un_{\text{jumps}} \wedge jumpDest(n_s) == jumpDest(n_e))$ **then**
4: $A_c = A_c \setminus (x, n_e) \; \forall \; x \in N$
5: **end if**
6: **end for**
7: $N' = N \setminus \{\text{End}\}$
8: $G' = (N', A_c)$

Algorithm 1 formalizes the new CDG generation process, which removes the unnecessary arcs. To perform that task, the algorithm uses an ACFG as the starting point. The algorithm uses the following functions and sets:

- *genControlArcs*\1. It inputs an ACFG and outputs all control arcs that can be obtained according to Definition 2.
- un_{jumps}. This is a set with all nodes that represent an unconditional jump.
- *jumpDest*\1. This function inputs a CDG node n that represents an unconditional jump statement and outputs the destination of the jump.

Algorithm 1 first generates all control dependencies in the ACFG. Then, each control dependency $n \to n'$ is inspected to determine whether both n and n' are unconditional jumps with the same destination. If this is the case, then all control arcs that target node n' are removed. This forms the set A'. Finally, N' is calculated by removing the *End* node from N and the CDG $G' = (N', A')$ is obtained.

With this generation process, the CDG produced is more accurate than the one produced by Ball and Horwitz. For instance, the CDG associated to the *Original program* in Example 1 is shown in Figure 5. The CDG slice associated to the slicing criterion `C` is shown in grey, and it corresponds to the *Minimal slice* in Example 1. As can be seen, nodes `break` and `if(Z)` are no longer part of the slice. The structure of this graph represents now a more realistic control dependence, where unconditional jumps to common destinations are not dependent on each other.

It is worth remarking the main difference between the solution presented in [11] and our approach: the amount of steps of the slicing process that are modified. Both approaches introduce a modification in the CDG generation process. While the amount of arcs generated by Kumar and Horwitz may be lower or greater than the amount of arcs generated in the initial proposal by Ball and Horwtiz [3], the amount of arcs generated in our approach is always equal

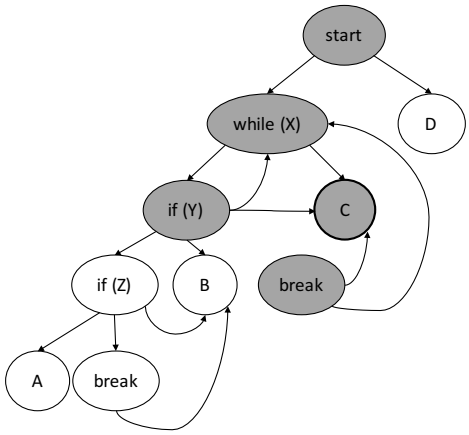

Fig. 5. CDG obtained by applying Algorithm 1 to the code in Example 1.

or lower than in the initial proposal. In addition, the approach by Kumar and Horwitz needs to change the standard SDG-traversal algorithm, introducing an overhead when calculating slices. On the contrary, in our approach the SDG-traversal algorithm remains untouched, keeping the slicing process as a graph reachability problem and ensuring the slicing cost proposed by Ottenstein and Ottenstein in [14].

4 The representation of `switch` statements

Another frequent structure where this problem appears is the `switch` statement. It is considered good practice to terminate each `case` with an unconditional jump, such as `return` or `break`. This creates an environment where there are multiple unconditional jumps with the same target, a perfect example of the situation where our technique applies.

In comparison with Kumar and Horwitz's approach, ours behaves in the same way, with one caveat: it includes the appropriate `case` statements, while theirs ignores them.

`switch` statements can be represented in many ways. Regarding its control-flow representation for program slicing, the following patterns are applied (where s is the `switch` statement):

- The selector or argument of s (e.g. `switch (a)`) is connected to each `case` statement. Additionally, if s has no default case, it is connected to the first instruction after s. Otherwise, it is connected to the first instruction after s via a non-executable arc.
- Each `case` c is connected to the first instruction that will be executed if the selector matches it, which is typically the first instruction in its body.[2]

[2] Multiple languages allow chaining `case` statements.

Furthermore, c is connected via a non-executable arc to the default case, if present, or otherwise to the first instruction after s.
- Statements within a `case` are represented as they would be in any other part of the program. The only caveat is: let s be the last statement of a `case` (c) statement's body, and s' the first statement in the following `case` (c') statement's body; if s does not jump, the following instruction would not be c', but s'. As an example; in the sequence of statements `a = 10; case 1; b = 1;`, the first will only be connected to the third, and not to the `case` statement. Thus, any unconditional jump at the end of a `case`, which is a common construct, will have a non-executable arc connected to the first instruction of the following `case` statement.

All these rules follow two maxims: (1) executable control-flow arcs connect instructions that may be executed sequentially and (2) non-executable control-flow arcs connect i to j, where i is any instruction and j is the instruction that would be executed in the case that i was a no-operation (a blank instruction that affects nor control, neither data). Regarding the control dependence graph, each part of the `switch` statement has sensible control dependencies:

- The selector is the source of data dependencies towards all instructions in its body, and this is the desired outcome: the selector is included if any `case` is; or in other words, the selector affects the execution of each `case`.
- Each case is the source of data dependencies towards the instructions in its body, and towards the default case (if present). The effect a `case` has on its body is clear, but the one on the default case may not be, though it is present. Consider a `switch` with n `case` statements and a default case. If any were removed, the default case would be affected, as the executions that previously passed through the deleted statement will now traverse the default case. Thus, the presence of each `case` statement affects the number of times the default case is run.
- Statements within a `case` c are control dependent on c, and possibly on unconditional jumps in previous `case` statements.

Example 4. Comparison of our technique against Kumar and Horwitz's in a simple `switch` statement

Consider the code displayed in Figure 6, where a simple `switch` statement is declared. On the right-hand side, its slice with respect to `S3` (line 11). Only one of the `break` statements is necessary, as they perform an equivalent effect on the slicing criterion.

Figure 7 shows the augmented control-flow graph of this simple procedure, with non-executable arcs shown with dashed edges. Figure 8 shows the resulting SDG built using our technique, and the slice obtained matches the one in Figure 6. Finally, Figure 9 shows the result of applying Kumar and Horwitz's technique. Note how their approach, though it generates more arcs, traverses fewer of them, leading them to the same result as ours.

```
1  public class Switch {
2      void f() {
3          switch (cond) {
4              case e1:
5                  S1;
6                  break;
7              case e2:
8                  S2;
9                  break;
10             case e3:
11                 S3;
12                 break;
13         }
14     }
15 }
```

```
1  public class Switch {
2      void f() {
3          switch (cond) {
4
5
6
7
8
9                  break;
10             case e3:
11                 S3;
12                 break;
13         }
14     }
15 }
```

Fig. 6. A program with a simple `switch` statement and its slice with respect to `S3`.

5 Implementation

Both our approach and Kumar and Horwitz's have been implemented in an open-source Java slicer, available at the URL `https://github.com/mistupv/JavaSDGSlicer`. In the branch `TAPAS-2020`, one can generate graphs using our approach with the flag "`-t TSDG`", and using Kumar and Horwitz's with "`PSDG`".

The specific implementation can be seen in the following classes for our approach:

TapasPDG Extends the implementation of the Augmented PDG (a PDG based on the ACFG), applying Algorithm 1 after generating the control dependency arcs.

TapasSDG Extends the implementation of the Augmented SDG (a SDG based on the APDG), by basing it instead on the aforementioned TapasPDG.

CFGBuilder, ACFGBuilder They implement the creation of the control-flow graph, specifically regarding the handling of the `switch` statement. The specific change can be seen in commit `3c771a29`.

As for Kumar and Horwitz's approach, it is implemented in the following classes:

PPDG Extends the implementation of the APDG, replacing the generation of control dependencies with their own.

PSDG Extends the ASDG to use PPDGs instead of APDGs, and uses a compatible slicing algorithm instead of the classic one.

PseudoPredicateSlicingAlgorithm Extends the classic slicing algorithm, with the additional restriction added in this approach.

6 Conclusions

Ball and Horwitz proposed the first program slicing technique with a specific treatment for unconditional jumps. Even though their technique produces complete slices in all cases, they were aware that accuracy could be improved, and

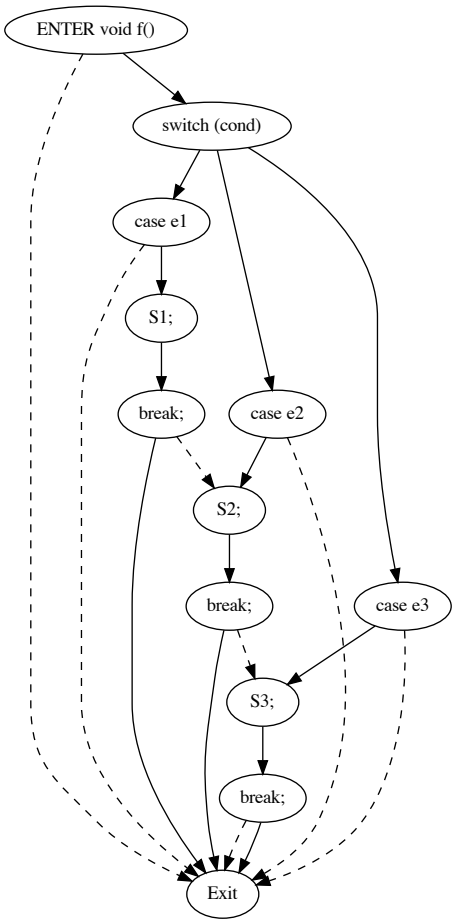

Fig. 7. The CFG obtained with our technique (common to both Kumar and Horwitz's and our technique).

they proposed a challenging example (analogous to Example 1) where the computed slice was bigger than needed. Some years later, Kumar and Horwitz solved this accuracy problem changing the definition of control dependencies and redefining the standard slicing algorithm.

In this paper, we propose an alternative approach that solves the problem performing fewer changes to the standard approach. Our approach only needs to change the CDG produced, and all the other phases of program slicing (including SDG traversal) remain unchanged. We have theoretically proven the correctness of our approach.

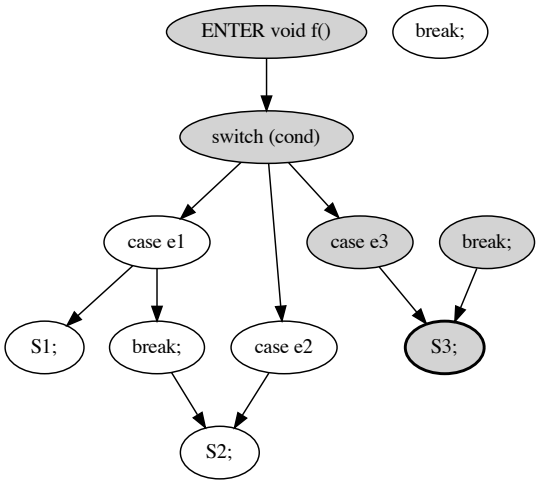

Fig. 8. The SDG and slice (gray nodes) w.r.t. to `S3`, obtained by applying our technique.

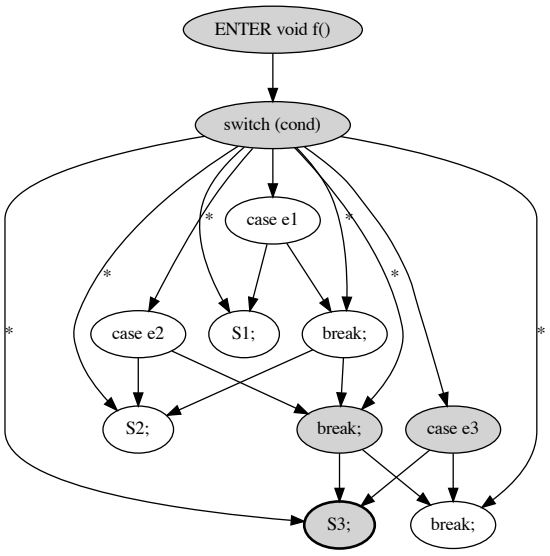

Fig. 9. The SDG and slice (gray nodes) w.r.t. to `S3`, obtained by applying Kumar and Horwitz's technique.

7 Acknowledgements

This work has been partially supported by the EU (FEDER) and the Spanish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

References

1. C. ai Sun, Y. Ran, C. Zheng, H. Liu, D. Towey, and X. Zhang. Fault localisation for WS-BPEL programs based on predicate switching and program slicing. *Journal of Systems and Software*, 135:191 – 204, 2018.
2. M. Aung, S. Horwitz, R. Joiner, and T. Reps. Specialization slicing. *ACM Trans. Program. Lang. Syst.*, 36(2):5:1–5:67, June 2014.
3. T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pages 206–222, London, UK, UK, 1993. Springer-Verlag.
4. D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2(1-4):31–45, March 1993.
5. Z. Chen and B. Xu. Slicing concurrent java programs. *SIGPLAN Not.*, 36(4):41–47, Apr. 2001.
6. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
7. A. Hajnal and I. Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software Maintenance*, 24(1):67–82, 2012.
8. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.
9. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.
10. J. Krinke. Static slicing of threaded programs. *SIGPLAN Not.*, 33(7):35–42, July 1998.
11. S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *Lecture Notes in Computer Science (LNCS)*, pages 96–112. Springer, 2002.
12. L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.
13. A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: Design, correctness, and evaluation. In *Proceedings of the 2007 ACM Workshop on Digital Rights Management*, DRM '07, pages 70–81, New York, NY, USA, 2007. ACM.
14. K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Software Engineering Notes*, 9(3):177–184, 1984.

15. T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, December 1994.
16. T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 41–52, New York, NY, USA, 1995. Association for Computing Machinery.
17. S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 432–441. IEEE, May 1999.
18. F. Tip. A survey of Program Slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
19. R. A. Towle. *Control and Data Dependence for Program Transformations.* PhD thesis, USA, 1976. AAI7624191.
20. M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.