Computing on-the-fly the Relevant Program Flows to a Control Dependency

Husni Khanfar

School of Innovation, Design, and Engineering, Mälardalen University,
SE-721 23 Västerås, Sweden
husni.khanfar@mdh.se

Abstract. Control dependence analysis is a key step in many program analysis techniques such as program slicing. The current approaches that compute control dependencies have two limitations. First, they are monolithically computed for the whole program at once. Second, these approaches do not compute the relevant program flows with the control dependency. They consider the relevant flows a slicing issue.
This work extends our previous work that computes on demand the control dependencies for a particular program statement without making a comprehensive analysis. It defines new concepts in static program analysis discipline, that will be a new foundation for many future works in static program analysis. Moreover, it significantly optimizes our previous work. Finally, it presents a new approach that can compute the relevant predicates and goto statements to a particular control dependency. The contribution of this work is well-proved, and it has succeeded with hundreds of examples. However, this is the first work that claims that it can compute accurately and on demand the control dependencies with its relevant flows.

Keywords: Program Analysis, Control Dependence, Program Slicing, Label-axis

1 Introduction

In programming languages, the program flow from a statement, or a predicate to another shows a possible order of execution between these two statements in the source code. In 1970, the program representation Control Flow Graph (CFG) was invented [1]. The main purpose of this graph is to abbreviate the source code and to represent its program flows. The abbreviation is made based on the program flows. Since then, the CFG becomes the base for many static program analysis techniques such as dataflow analyses, dominance analysis and points-to sets algorithms. The CFG plays an important role in developing the computer science in the last decades. It is noticeable that most of the techniques built on CFG use fixed-point iterations analysis paradigm. However, one of the disadvantages of the CFG is that it distracts the syntactic structure of the source and destroys its location-based information.

Control dependency is a relationship between a predicate and a statement, in which the outcome of the predicate at the run-time determines the possible execution of the statement. The state-of-the-art approach that computes the control dependence is based on the CFG that analyses the whole program at once [2,3,4,5,6,7,8]. Our previous work [9] shows an approach that finds on demand the minimal set of predicates that control the execution of a particular statement

MRTC Report, Mälardalen Real-Time Research Centre - Mälardalen University - Sweden
ISRN: MDH-MRTC-334/2021-1-SE
Received in 15-April-2021 Accepted in 16-April-2021

s. Our algorithm relies on understanding the location of each program flow with regards to other flows. This algorithm determines the control dependencies of a given statement in three phases; the first phase checks whether the statement is syntactically inside a conditional statement that does not include a jump flow. This phase can make a fast decision without further computation. The second phase excludes all the predicates that they certainly do not control the statement of interest. The remaining predicates are the candidates to be the controllers of s. This phase makes a fast over-approximation which it is efficient because the number of the candidates is few to the total predicates in the program. The third phase uses a (costly) algorithm called Exploring Paths by Stopping (EPS) to determine the candidates that control s.
This work adds the following contributions to the previous one:

- Introducing the new definition *Sequence of Overlapping Flows* and distinguish between it and the definition *Set of Overlapping Flows* which was presented in [9].
- Introducing the label-axis program representation, which is derived from the CFG, but it illustrates better the control flows with their locations.
- Determining the sub-figure in the CFG that we use to investigate the control dependency between a predicate and a statement.
- Computing the relevant program flows that establish a control dependence relationship.

The paper is organized as follows, Section 2 provides the background. Section 3 presents new definitions in this discipline. Section 4 summarizes our previous approach. Section 5 presents a new efficient optimization for Theorem 2 that is described in our previous work [9]. Section 6 shows our new approach in computing on-the-fly the relevant flows to a particular control dependency. Section 7 explains how to avoid missing relevant flows. Section 8 presents the related work and work conclusion is presented in Section 9.

2 Background

This section provides a brief description of the While language, the state-of-the-art program representation CFG, the post-domination concept and the control dependencies.

2.1 While Language and CFG

The While language [10] is a small model imperative programming language that is used to develop and test new approaches and methods specialized in the analysis of source codes. Using the While language for such developments gets rid of unnecessary details included in the real languages.

A While program is a statement s, which might be an elementary statement (es), conditional statement (cs) or a composite statement ($s_1; s_2$). In [10], every elementary statement or a predicate has a unique integer label. This work extends the labeling scheme in [10] by giving a unique label to every conditional statement. Moreover, a `goto` statement is added to the syntax of the While language. The statements are labeled in ascending order according to their locations in the source code, from left to right and from top to bottom. With this labeling system, all program flows except backward jumps go from statements with lower labels to statements with higher labels [9].

For introducing the abstract syntax of the While language, we suppose that a denotes arithmetic expressions, and the predicate b denotes boolean expressions. Based on that, the abstract syntax of the While language is:

$$cs ::= \texttt{if } [b]^\ell \texttt{ then } s' \mid \texttt{if } [b]^\ell \texttt{ then } s' \texttt{ else } s'' \mid \texttt{while } [b]^\ell \texttt{ do } s'$$
$$es ::= [x := a]^\ell \mid [skip]^\ell \mid [goto\ \ell']^\ell$$
$$s ::= es \mid s'; s'' \mid cs$$

If clear from the context, we will abuse notation and write "predicate p", or "statement s" instead of "the label of predicate p" or "the label of statement s".

Definition 1. ***Control Flow Graph:*** *The Control Flow Graph for an intra-procedural program s is a 4-tuple $(N, E, Entry, End)$.*

1. *N is a set of statements, where each statements represents an elementary program statement in s.*
2. *E is a set of program flows, where each program flow represents a possible program flow from a statement to another. $E \subset (N \times N)$.*
3. *Entry: is a unique start statement. $Entry \in N$.*
4. *End: is a unique exit statement. $End \in N$.*
5. *There is a path from Entry to every $n \in N$.*
6. *There is a path from every $n \in N$ to End.*

We refer the reader to the book "Principles of Program Analysis" [10] for further details about the construction of the CFG from a While program. The formal definition of building a CFG from a While program in [10] defines five types of functions: *init, final, blocks, labels* and *flow* to construct a CFG from a While program. In these definitions, Lab refers to the set of all labels in the program. These definitions are extended to include `goto` statements as follows:

$$init : \text{Stmt} \to \text{Lab}$$

which gets the initial label of a statement:

$$\begin{aligned}
init([x := a]^\ell) &= \ell \\
init([skip]^\ell) &= \ell \\
init([goto\ \ell']^\ell) &= \ell \\
init(\text{S}_1; \text{S}_2) &= init(\text{S}_1) \\
init(\text{if } [b]^\ell \text{ then S}) &= \ell \\
init(\text{while } [b]^\ell \text{ do S}) &= \ell \\
init(\text{if}[b]^\ell \text{ then S}_1 \text{ else S}_2) &= \ell
\end{aligned}$$

$$final : \text{Stmt} \to \text{P(Lab)}$$

which gets the set of last labels in a statement:

$$\begin{aligned}
final([x := a]^\ell) &= \{\ell\} \\
final([skip]^\ell) &= \{\ell\} \\
final([goto\ \ell']^\ell) &= \{\ell\} \\
final(\text{S}_1; \text{S}_2) &= final(\text{S}_2) \\
final(\text{if } [b]^\ell \text{ then S}) &= \{\ell\} \cup final(\text{S}) \\
final(\text{while } [b]^\ell \text{ do S}) &= \{\ell\} \\
final(\text{if}[b]^\ell \text{ then S}_1 \text{ else S}_2) &= final(\text{S}_1) \cup final(\text{S}_2)
\end{aligned}$$

2.2 Basic Definitions

In this subsection, we introduce few definitions that are related to the control dependencies.

Definition 2. ***Post-domination:*** *In a CFG G, any node n* post-dominates *node y if all the paths from y to Exit contain n.*

Definition 3. ***Immediate Post-dominator:*** *In a CFG, the node n* immediately post-dominates *another node m if n strictly post-dominates m but does not strictly post-dominate any node that strictly post-dominates m.*

Definition 4. ***Standard Control Dependence:*** *In accordance to [3], node n is* standard control dependent *on node m in program s if:*

1. *There exists a non-trivial*[1] *path π from m to n such that every node $n' \in (\pi - m, n)$ is post-dominated by n; and*
2. *m is not strictly post-dominated by n.*

This relationship is expressed in this context as a $\overset{cd}{\underset{st}{\Longrightarrow}}$ b, where a is a predicate controls b. According to [9], both a and b should belong to the same SOF st, and the SOF (st in our case) is shown under $\Rightarrow$. The SOF is defined later in Def. 14.

Definition 5. ***The Conditional Statement of a Label:***
The conditional statement cs of a label ℓ refers to the innermost conditional statement where ℓ exists.

3 Program Flows

This subsection introduces many definitions that are related to program flows. Since this work is an extension to our previous work [9], we will mark each repeated definition by

Definition 6. ***The program flow notation ($\rightarrow$)***[2] *refers to a pair of labels defining a program flow, such as:*

$$\ell \rightarrow \ell'$$

where ℓ is the outgoing label and ℓ' is the ingoing label. O symbol denotes the outgoing label in a flow. I symbol denotes the ingoing label in a flow.

So,O($\ell \rightarrow \ell'$) =ℓ, and I($\ell \rightarrow \ell'$) =ℓ'.

Definition 7. *A* ***Normal Program Flow*** *occurs between two labels $a \rightarrow b$ wherein $b = a + 1$.*

Definition 8. *A* **Jump (Unstructured) Flow***[9] is a program flow wherein the outgoing side is an* unstructured jump statement *(e.g. `goto`).*

Definition 9. *A* **Structured Flow***[9] is a program flow wherein* the outgoing side is a structured jump statement *(e.g. `if` or `while`).*
The structured flows in the conditional statements in the While language are defined as follows:

1. *`if-then-else`. suppose the code: if b^c then S_1 else S_2; S'.*
   The flows of [if b^c then S_1 else S_2] are:

[1] Path π is non-trivial if it contains at least two nodes [11]
[2] The right arrow is used also in this paper to define the terms and concepts.

- $c \rightarrow init(S_2)$: *the structured flow.*
- $c \rightarrow init(S_1)$.
- $final(S_1) \rightarrow init(S')$: *a jump flow.*
- $final(S_2) \rightarrow init(S')$.

2. `if-then`. *suppose the code:* $if\ b^c\ then\ S_1;\ S'$.
   The flows of $[if\, b^c\ then\ S_1]$ *are:*
   - $c \rightarrow init(S')$: *the structured flow.*
   - $c \rightarrow init(S_1)$.
   - $final(S_1) \rightarrow init(S')$: *a jump flow.*
3. `while`. *suppose the code:* $while\ b^c\ then\ S_1;\ S'$
   The flows of $[while\ b^c\ then\ S_1]$ *are:*
   - $c \rightarrow init(S')$: *the structured flow.*
   - $final(S_1) \rightarrow c$: *a jump flow.*

3.1 Label-Axis

The **label-axis** is a program representation derived from the CFG, where most of the unidirectional edges are replaced by a straight horizontal line (label-axis) and the circles of the nodes in the CFG are replaced by small vertical ticks on the label-axis. The values of the ticks appearing in their underneath represent the labels in the program. The label-axis runs from left to right. Thus, the label value in the left is always less than any to the right. The straight line between two consecutive ticks represents a flow from the label in the left to the label in the right. The structured and jump flows in the program are represented by arrows on the label-axis.

Figure 1 shows a label axis representation for an unstructured source code. This examples shows how the labels and the flows are represented on a label-axis. Notice there is a flow from 3 to 4 and this flow is represented by a straight line from tick 3 to tick 4. In this example, the structured flows are plotted by continuous lines, whereas the jump flows are plotted by dashed lines. The difference between the arrow shapes helps in understanding better the syntactical structure of the program, while the ascending locations of the ticks helps to figure-out directly the effect of the locations, whereas the arrows and straight lines between the ticks represent the program flows. So, the label-axis show three types of information in one diagram.

Definition 10 (Interval of Labels). *is a set of consecutive labels in the label axis that are bounded between two label values.*

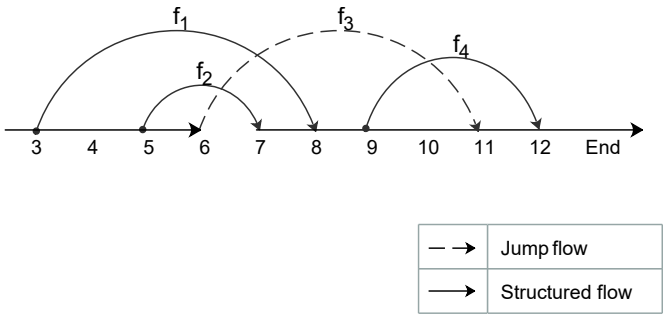
The bounding relation can map four types of data to an interval as follows:

$$\begin{aligned} \text{Label}\times\text{Label} &\rightarrow \text{Interval} \\ \text{SOF} &\rightarrow \text{Interval} \\ \text{SQC} &\rightarrow \text{Interval} \\ \text{Flow} &\rightarrow \text{Interval} \end{aligned}$$

Notice that the definition of SOF and SQC will follow later in this work. There are four bounding operators, which are [...] ,]...] , [...[and]...[. Each of these operators maps the four data types to an interval. In Mathematics, the differences between these operators can be shown by the following examples: $[i, w]$ defines an interval of labels from i to w. $]i, w]$ defines an interval of labels from $i + 1$ to w. $[i, w[$ defines an interval of labels from i to $w - 1$.

```
if[b0]3 then
   [x:=x+1]4;
   if[b1]5 then
      [goto L1]6;
   [y:=y+1]7
[x:=x+2]8;
if[b2]9 then
   [x:=x+3]10
   L1:[y:=y+2]11;
[h:=10]12
```

(a) Unstructured program



(b) The flows at the label axis

Fig. 1: Example of an unstructured program and the label-axis of its flows.

3.2 Overlapping of Program Flows

Herein, the interleaving of program flows is introduced and classified into two main categories; namely overlapping and intersecting.

Definition 11. ***Overlapping Flows:****[9] the program flow* $d \to h$ overlaps $c \to f$ *when* $c > d \geq f$ *or* $c < d < f$ *as well as* h *is either less than* c *and* f *or larger than* c *and* f*.*

Fig. 2 depicts the concept of overlapping and intersecting flows.

Definition 12. ***Bypassing:****[9] the program flow* $j \to v$ bypasses *the label* t *if either* $j < t < v$ *or* $j > t \geq v$*.*

Definition 13. SQC *is an abbreviation of a sequence of overlapping flows, where each flow overlaps a predecessor flow in the sequence. The sequence are surrounded by left and right angle braces, and the flows are separated by commas, for instance,* $\langle f_i, f_{i+1}, ..., f_n \rangle$*.*

The SQC is always addressed by its first flow, which should be in this context a structured program flow. Thus, we can say the SQC of f_i, or the SQC of p→w. However, the SQC of f_i is a subset from SOF that contains f_i. Now, to idenitfy the SQC of f_i existing in a SOF, we use the relation Q:

Definition 14. SOF *is an abbreviation of a Set of Overlapping Flows. This set is formed from the flows of an SQC or many SQCs. For each SQC, one of its flows should be overlapped with a flow from another SQC. The SOF is represented by a finite number of flows surrounded between two cutely braces, and the flows are separated by commas, for instance,* $\{f_i, f_{i+1}, ..., f_n\}$*. Each flow should exist once in the set even it belongs to many SQCs.*

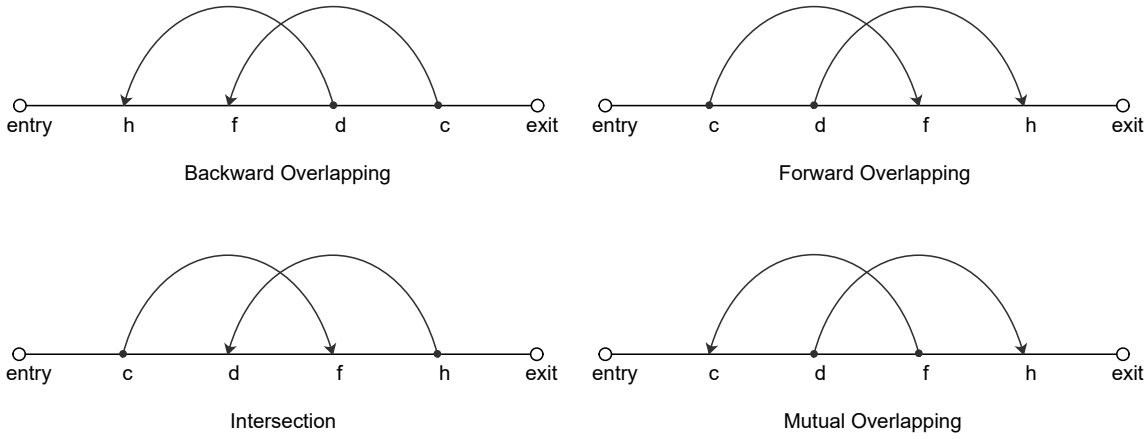


Fig. 2: Overlapping and intersecting flows [9]

Definition 15. $\mathcal{S}$*:* Flow $\rightarrow$ SOF
Let Flows is a set of flows. $\mathcal{S}_f(Flows)$ *denotes an SOF from Flows that includes* f*.*

It is permissible also to replace the structured flow by its out label. Thus, we can replace $\mathcal{S}_f(Flows)$ by $\mathcal{S}_{O(f)}(Flows)$.

Definition 16. Q*:* SOF $\times$ Flow $\rightarrow$ SQC
Let st be a SOF *that includes the structured flow* f*.* Q_f^{st} *denotes the* SQC *of* f *in st.*

It is permissible to drop the SOF from Q relation because the flow cannot exist in more than SOF. In this case, it is known that we mean the SOF which includes f. Moreover, it is permissible also to replace the structured flow by its out label (predicate). Thus, we can replace Q_f by $Q_{O(f)}$ or by Q_p .

In our work with the data types: Flow, SQC and SOF types, there is a need to define the maximum and minimum labels of each of these relationships. However, *Min* and *Max* relationships map the instance of each of these data types to the minimum or maximum label in these instances respectively.

Definition 17. *Min*(*Flow*) *: it maps a Flow to the minimum label among its end labels.*

Definition 18. *Min*(*SOF*) *: it maps a SOF to the minimum label among the end labels of its flows.*

Definition 19. *Min*(*SQC*) *: it maps a SQC to the minimum label among the end labels of its flows.*

In the same manner, *Max* is defined by these three data types and the interval operators can work with these data types. If we suppose that *st* is a SOF and *sq* is a SQC, then:
$[st]$=$[Min(st), Max(st)]$.
$]sq]$=$]Min(sq), Max(sq)]$.

Definition 20. Scope*:* Predicate → Interval
The Scope of predicate p is the interval [*Min*(Q_p) , *Max*(Q_p) [

Definition 21 ($\overline{st}$). *is the set of labels in the SOF st.*

4 On-the-fly Computation of Control Dependencies

This section presents the set of rules (theorems) that we presented in [9] to compute on-the-fly standard control dependencies. The proofs of these theorems are shown in our previous work [9].

Theorem 1. *Let p a label of a predicate with a scope interval from k to v. Then no possible control dependence relationship can be established between p and a label outside its scope.*

Theorem 2. *The label ℓ is control dependent on the predicate p if and only if in Exploring the Paths from the two immediate successors of p, where the stopping list for both explorations is: {ℓ,End}, the collection of one of the explorations will not contain End where the other includes End* [3]*.*

Theorem 3. *Let ℓ a label with conditional statement cs, and let p the predicate of the conditional statement cs. Further, assume that cs does not comprise any jump flow (it has neither ingoing or outgoing label of a jump flow). Then ℓ is control dependent on p, and it is not control dependent on any other predicate.*

In finding the predicates that control a particular statement s, there are three phases. The first phase is an implementation of Theorem 3. In this phase, the algorithm checks whether the conditional statement cs of s comprises an ingoing or outgoing label of a jump flow. If it does not, then s is solely control dependent on the predicate of cs. Otherwise, we proceed to the second phase which is an implementation of Theorem 1. The second phase conditions that both s and any predicate controlling s exist in the boundaries of an SOF (Def. 14). By this, few candidates of predicates remain and most of other predicates in the program are excluded from the consideration. The role of this phase is important because the third phase applies a costly algorithm with respect to time and space. Thus, it is important to pass a few number of candidates to the third phase. To check the control dependence between each candidate and s, the third phase (the implementation of Theorem 2) employs a first-depth search technique [12], called here an *Exploring Paths by Stopping* (EPS). To determine whether s is control dependent on the candidate p; the third phase makes two EPS explorations from p and its final judgment is made by the rule: if an EPS exploration from p traverses s and another EPS from p does not traverse s, then s is control dependent on p.
In addition to the above theorems, we need also to borrow the following lemmas and definitions from [9]:

Lemma 1 *Let p the label of a predicate with scope interval from k to v. Then v post-dominates p.*

Lemma 2 *ℓ is control dependent on the predicate with label p, if and only if one of the immediate successors of p is post-dominated by ℓ and the other immediate successor of p is not post-dominated by ℓ.*

[3] For better understanding of this theorem, interested readers are advised to read [9]

Informally, we can define the EPS as a technique that conducts a depth-first search in a tree of forward paths whose root is an already determined label ℓ. The search starts at ℓ. Then it moves to visit its immediate successors and so on. Each new visited label is added to a special collection *clctn*. The search does not visit the immediate successors of some labels, which are in a special list called the *stopping list*. To prevent the occurrence of infinite loops, the search does not add the already visited labels to *clctn*.

5 Subfigure

Theorem 2 checks a potential control dependence $p \overset{cd}{\underset{sf}{\Longrightarrow}} \ell$ by launching two EPS searches from p. While one of the searches should reach End, thus making a bad time complexity especially with large programs containing several thousands of program statements. This section shows how we can optimize the searching to be inside the boundaries of the SOF *sf* to which both the predicate p and the statement of interest ℓ belong. Therefore, rather than finding a path from p to END for checking the truth of $p \overset{cd}{\underset{sf}{\Longrightarrow}} \ell$, the subfigure concept allows to condition reaching to $Max(sf)$. As a result, a considerable amount of unnecessary explorations and traversing of statements are eliminated. This concept is applied as what Theorem 4 states.

Theorem 4. *Suppose that sf is a SOF,* $p \in [\ sf\ [$, $\ell \in [\ sf\ [$ *and two EPSs from the two immediate successors of p, with the stopping list:* $\{\ell, Max(sf)\}$*, are carried out. The relation* $p \xrightarrow[st]{cd} \ell$ *is true if and only if the collection of an EPS contains* $Max(sf)$ *whereas the other EPS includes* $Max(sf)$*.*

Proof. This theorem is an extension to Theorem 2. However, the difference is in replacing End by $Max(st)$. We suppose that this replacement is correct because Lemma 1 states that Max(st) postdominates ℓ and the two immediate successors of p. Thus, if the two EPSs reach Max(st), then both will reach End because $\ell \notin [Max(st)$,End]. Therefore, we consider that *st* is the subfigure we use to check the standard control dependence inside its boundary and there is no need to go beyond it. □

6 Computing the Relevant Flows

So far, the approach shown earlier in this paper focuses on finding the predicates that control the execution of a statement; however, this does not include the relevant program flows which are essential to build the paths to satisfy the conditions in Def. 4. This section presents a new theorem to extract the relevant flows. It is worth mentioning that these flows are a subset of the entire flows in the SOF which we applied Theorem 4 on.

Definition 22 (Relevant Program Flows to a Control Dependency). *Creating a standard control dependence between a predicate p and a label* ℓ *requires the existence of paths from p in accordance to Def. 4. The Relevant program flows to* $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$ *are the structured and jump flows in these paths. This relationship is written as* $\mathcal{R}(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)$*.*

The control dependency relationship consists of three main elements, the predicate (controller), the controlled statement, and the relevant predicates and `goto` statements that draw the paths to build this relationship in accordance to Def. 4. In contrast to all other related works, this literature focuses on computing the relevant program flows to a control dependency rather

than the relevant statements, which are simply the outgoing labels of the relevant flows. Although the difference between *Relevant Flows* and *Relevant Statements* is small, it hides a difference in the way of the manipulation. This work uses the SOFs to compute the control dependencies with their relevant facts. By this, all the elements of a control dependence relationship are easily computed once. Other works use the post-domination facts to compute the control dependencies, but these facts are completely ineffective in computing the relevant statements. The post-domination facts are formed due to the existence of paths, whereas the relevant statements emerge due to the absence of paths. Thus, using the post-domination facts to compute the relevant statements is not possible. Hence, the researchers considered the problem of the relevant statement a program slicing concern, thereby, they tried to find the relevant statements that affect a particular point. This is why we use the term *Relevant Flows* rather than *Relevant Statements*.

Historically[4], the computation of the Relevant Statements was one of the most difficult research problems in program slicing that required 22 years (from 1980 to 2002) to be solved. The twists and turns occur because the Relevant Statements do not affect the second element in the control dependence relationship. So, it was completely illogical to solve relevant statement problems through a program slicing paradigm. In our opinion, these ways of manipulation are consequences to jump over the main obstacle that was hindering the researchers. The obstacle was that the researcher was and still tied up to the CFG.

This section shows how we can find all the predicates controlling a particular statement with their relevant flows. So, we compute once all the elements. This section initializes a set of definitions and lemmas to state the final theorem.

Lemma 3 *Suppose $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$, where st is a SOF. Theorem 4 is implemented to check this dependency and this implementation produces two collections clct′ and clct″, the labels in the set $clct' \cap clct''$ are not relevant to any potential control dependency relationship between p and ℓ.*

Proof. Definition 4 and Theorem 4 show that there are two distinct paths from p. The first is from p to ℓ, where ℓ postdominates each label in it. The second path is from p to Max(st) and it does not include ℓ. Assuming $\ell' \in]st[$, and the two paths meet at ℓ', then ℓ does not postdominate ℓ'. Consequently, the first condition in Definition 4 is not fulfilled. □

Lemma 4 *Suppose $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$, where st is a SOF. Theorem 4 is implemented to check this dependency and its implementation produces two collections clct′ and clct″. All the relevant predicates to this dependence exist in the subset union. union $\subseteq \overline{st}$, and it denotes to:*

$$union = p \ \cup \ clct' \ \cup \ clct'' \setminus clct' \cap \ clct'' \tag{1}$$

Proof. Since the labels of the paths from p to ℓ and from p to $Max(sq)$ are collected in $clct'$and $clct''$, all other labels of predicates and `goto` statements in st are not relevant to $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$, because they do not exist in the paths forming $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$. Furthermore, in accordance to Lemma 3, we exclude from the subset union all the labels in $clct' \cap clct''$. Hence, the equation is verified. □

Definition 23. *$\mu(Predicate \overset{cd}{\underset{st}{\Longrightarrow}} Label)$ is a subset of labels in $\overline{st}$. It is formed from the united of the two collections clct′ and clct″ that are produced to compute $p \overset{cd}{\Longrightarrow} \ell$ in accordance to Theorem 4 after excluding the repeated labels in the two collections. Mathematically, this can be represented as:*

$$\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell) = clct' \ \cup \ clct'' \setminus clct' \cap \ clct'' \tag{2}$$

[4] All these details will be shown in the Related Work section

Definition 24. $\mathcal{P} : Labels \rightarrow Labels$
The relation $\mathcal{P}$ returns the predicates and goto labels from a set of labels.

In applying the relation $\mathcal{P}$ on $\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)$, we can get the following:

$$\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)) = \{\ell' | \ell' \in \mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell), \ell' \text{ is predicate} \vee \ell' \text{ is } \texttt{goto} \ \}$$

Definition 25. $\mathcal{F} : SOF \rightarrow Flows$
The relationship $\mathcal{F}$ returns the set of flows that form an SOF.

Definition 26. $\overrightarrow{\mathcal{F}} : Labels \rightarrow Flows$
The relationship $\mathcal{F}$ returns the set of flows from a set of labels.

For calculating $\mathcal{R}(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)$, we follow specific steps. First, the implementation of Algorithm 4 runs to calculate the set $\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)$. Then, the subset of the predicates and goto labels in $\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)$ is computed and it is denoted by $\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell))$. Next, the flows whose out labels are in $\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell))$ are computed and denoted by $\overrightarrow{\mathcal{F}}(\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell))$. Afterward, the SOF st'= $\mathcal{S}_p(\overrightarrow{\mathcal{F}}(\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell))))$ is computed. Finally, $\mathcal{R}(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell) = \{f | f \in st'\}$. The formulation of this approach can be summarized by the following theorem.

Theorem 5. *If $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$ exists, then*
$\mathcal{R}(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell) = \mathcal{F}(st')$ *where* $st' = \mathcal{S}_p(\overrightarrow{\mathcal{F}}(\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell))))$.

Proof. Lemma 4 states that the labels in the set $\{\overline{st} - \mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)\}$ are not relevant to $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$. As a result, the labels $\mathcal{P}(\overline{st} - \mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell))$ and the flows $\mathcal{F}(\mathcal{P}(\overline{st} - \mu(p \overset{sc}{\underset{st}{\Longrightarrow}} \ell)))$ are irrelevant too. Based on that, the relevant flows are included in the set $\overrightarrow{\mathcal{F}}(\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)))$. This is written as: $\mathcal{R}(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell) \subseteq \overrightarrow{\mathcal{F}}(\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)))$. Since Theorem 1 requires that p and ℓ belong to the same SOF, we construct from the flows in $\overrightarrow{\mathcal{F}}(\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell)))$ a SOF $st' = \mathcal{S}_p(\overrightarrow{\mathcal{F}}(\mathcal{P}(\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell))))$. Then, $\mathcal{R}(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell) = \mathcal{F}(st')$. □

Theorem 1 and Theorem 5 uses the overlapping of program flows as a foundation in its work. Theorem 1 uses the overlapping flows to exclude all the predicates that are certainly not related to the label of interest. Later, Theorem 5 uses again the overlapping flows to find the relevant flows.
We can explain Theorem 5 as follows:

- The **First Step**: we create the set *union* from the collections $clct'$ and $clct''$ that are produced from the implementation of Theorem 2 as shown in Equation 1.
- The **Second Step**: a small algorithm can be written as:
  1. The set of the predicates and `goto` labels in *union* are determined.
  2. The flows of the predicates and `goto` labels in (Label 1) are determined.
  3. The sequences from the flows in (2) are recomputed.
  4. The flows of the sequence in (3) which includes ℓ and p are the relevant flows to the control dependency between p and ℓ.

6.1 Example 1

The steps for finding the predicates that control label 4 in Fig. 1a are as follows:
The first phase (Theorem 3): The statement 4 is in the body of the conditional statement whose predicate is 3. Since this conditional statement comprises a jump program flow at 6, it is not possible to decide that 4 is control dependent on 3.
The second phase (Theorem 1): there is one sequence of a SOF (st) that bypasses 4. $st = [3 \rightarrow 8,\ 6 \rightarrow 11, 9 \rightarrow 12]$. From this set, we find that the predicates that might control the execution of 4 are: $\{3, 9\}$.
The third phase (Theorem 2): explores the paths from the immediate successor of each predicate computed in the second step to determine whether it controls 12. Since $Max(st) = 12$, the stopping list for all the explorations is {4,12}. The EPS searches are:

- For 3, whose immediate successors are 4 and 8: $clct'(4)$={4}. $clct''(8)$=$\{8, 9, 12\}$. Since 12 exists in one of the collections only and it does not appear in the other collection, 3 is control dependent on 4.
- For 9, whose immediate successors are 10 and 12: $clct'(10) = \{10, 11, 12\}$, and $clct''(12)$=$\{12\}$. Since 12 exists in both collections, we conclude that 4 is not control dependent on 9.

Finding the relevant predicates and goto statements for the control dependence between 3 and 4:

- The first step: the union is $\{3, 4, 8, 9, 12\}$. Fig. 3-a depicts the flows from this union.
- The second step: there is one sequence of overlapping flows that bypasses 4. it is $[3 \rightarrow 8]$ (see Fig. 3-b). Since the flow in this sequence is the structured flow of the predicate 3, we can say that there are no relevant flows to the control dependence between 3 and 4.

6.2 Example 2

In Fig.1, For finding the predicates that control the statement 7:
First phase: The conditional statement where 4 exists comprises a `goto`.
Second phase: The overlapping flows $[3 \rightarrow 8,\ 6 \rightarrow 11, 9 \rightarrow 12]$, and $[5 \rightarrow 7,\ 6 \rightarrow 11, 9 \rightarrow 12]$ bypass 7. The candidate predicates are: $\{3, 5, 9\}$.

Third phase: For 3: $clct'(4)$={4,5,6,11,12}. $clct''(8)$=$\{8, 9, 12\}$. For 5: $clct'(6)$={6,11,12}. $clct''(7)$={7}. For 9: $clct'(10) = \{10, 11, 12\}$, and $clct''(12)$=$\{12\}$. Based on the above collections, we find that 7 is control dependent only on 5.
Finding the relevant predicates and goto statements for the control dependence between 5 and 7:

- The first step: the union is $\{5, 6, 7, 11, 12\}$ (Fig. 3-c).
- The second step: the overlapping flows that bypasses 7 are $[5 \rightarrow 7, 6 \rightarrow 10]$ (Fig. 3-d). So, 6 is a relevant statement.

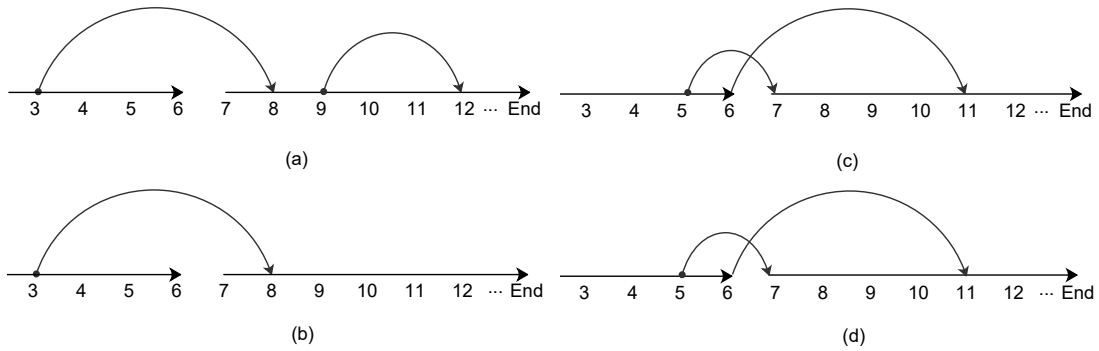


Fig. 3: The flows in the label-axis from (a) First step in Example 1 (b) Second step in Example 1 (c) First step in Example 2 (d) Second step in Example 2.

7 Missing Relevant Flows

Given that there is a predicate p, whose structured flow belongs to the SOF st, its two immediate successors are p' and p'', and the label $\ell \in st$. Lemma 2 states that if the relation $p \overset{cd}{\underset{st}{\Longrightarrow}} \ell$ exists, then if run an EPS from one of the immediate successors p', then EPS does not reach $Max(st)$. In other words, the condition Def. 4-A is satisfied, and in accordance to Theorem 4, the labels are collected due to the running of EPS from p' do no include $Max(st)$. We name this collection during this context as $clct'$. In accordance to Theorem 4, another EPS should run from the second immediate successor p'' and this EPS reaches $Max(st)$. The labels visited during this exploration are collected in the collection $clct''$. This section discusses the effect of the emergence of many paths satisfying the condition Def. 4-A from p' and the emergence of many paths from p'' that satisfy the condition Def. 4-B. This discussion is important to eliminate under-approximated results from the implementation of Theorem 5.

Studying Figure. 4 shows that many paths satisfy Def. 4-B from p''. The two immediate successors of 1 are 2 and 12, the SOF $st = \{f1, f2, f3, f4, f5, f6, f7\}$, and $Max(st)$=17. To study the potential control dependence between 1 and 14, we run an EPS from 2 and another EPS from 12 with a stopping list={14,17}. Considering label 12, there is one path from 12, which is 12,13 and the labels in this path are post-dominated by 14. Regarding label 2; there are three paths that bypass 14 and reaches 17. The first path is 2,3,17, whereas the second is 2,4,5,6,16,17; meanwhile the third is 2,4,5,6,8,9,15,16,17. Since the implementation of Theorem 4 stops as soon as it reaches 17, if the EPS of 2 traverses the labels 2,3,17, then $clct''$won't include the out labels in the second and third paths. Consequently, many relevant flows are excluded from $\mathcal{R}(1 \overset{cd}{\underset{st}{\Longrightarrow}} 14)$, namely, $f3, f4, f5, f6, f7$.

However, to include all the relevant flows in the paths that satisfy Def. 4-B from 2; the implementation of Theorem 4 should be changed as follows: the EPS should not stop exploring the paths as soon as it reaches $Max(st)$ and it should return to the labels that are in joints but they are not visited. For instance, the EPS of label 2 starts its exploration from 2, whose immediate successors are 3 and 4. When the EPS visits 2 and decides to continue toward 3, then it should save 4 in a worklist. Next, when the EPS reaches 17 and finds that the worklist is not empty, then it removes 4 from the worklist, adds it to the collection and continues the

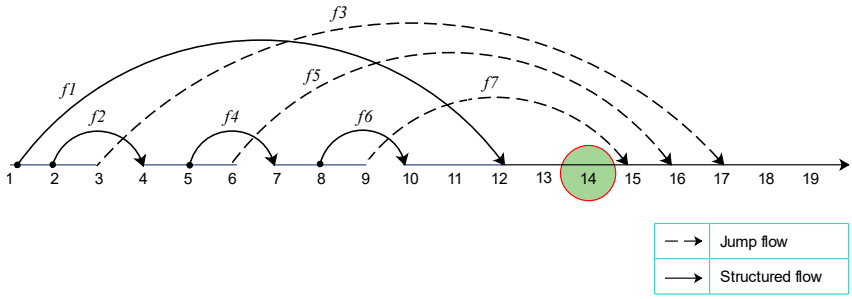


Fig. 4: Many paths bypass the statement under analysis

exploration from 4, thereby all the paths from 2 are traversed and added to the collection. As a result, all the out labels of the flows $f3, f4, f5, f6$ and $f7$ will be added to the collection.

Regarding the paths from p' that satisfy Def. 4-A, Theorem 4 mentions that all of these paths should not reach $Max(st)$. Thus, all the paths should be explored to check the satisfaction of this condition, and the worklist is already implemented in a somehow. However, should the paths from p' to ℓ be considered relevant? Practically, they can be discarded because they do not influence the execution of ℓ.

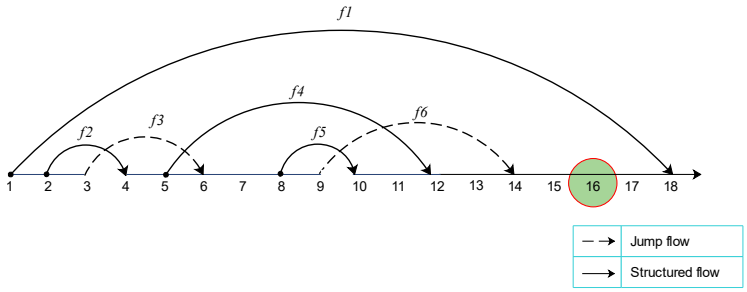


Fig. 5: Post-dominating many paths

Fig. 5 shows two nesting SOFs, which are $st = \{f1\}$, and $st' = \{f2, f3, 4, f5, f6\}$. For checking $1 \overset{cd}{\underset{st}{\Longrightarrow}} 16$, we find many paths from 2, all of which are post-dominated by label 16; e.g. the path 2,4,5,12,13,14,15, and the path 2,3,6,7,8,9,14,15. These paths include the out labels of the flows $f2, f3, f4, f5$ and $f6$. Since these flows start and end in a path from 2 to 16, they can be discarded from considered as relevant paths.
Based on that, $clct'$in equation (1) can be eliminated and be replaced by the following equations:

$$union = p \ \cup \ clct'' \tag{3}$$

Accordingly, Def. 23 can be changed to the following:

Definition 27. *$\mu(Predicate \overset{cd}{\underset{st}{\Longrightarrow}} Label)$ is a subset of labels in $\overline{st}$, which contains the labels from the paths starting from the immediate successor of p that satisfies the condition in Def. 4-B. This*

can be expressed mathematically as:

$$\mu(p \overset{cd}{\underset{st}{\Longrightarrow}} \ell) = clct'' \tag{4}$$

8 Related Work

For debugging purposes, Weiser [2] proposed the utilization of backward slicing, which is based on dataflow equations, but in working with unstructured jumps, this approach misses proper relevant program flows. Ottenstein and Ottenstein [3] invented the Program Dependence Graph (PDG). In these graphs, the program statements, expressions, inputs, parameters, and global variables are represented by vertexes; while edges represent control and data dependencies. Horwitz et al. [13] extended the PDG to System Dependence Graph (SDG) that represents inter-procedural programs.

There are many algorithms [14,15,16,17,18,19,20,21] that are developed to find the post-domination facts. All these algorithms rely on fixed-point iterations, which require comprehensive analysis along with all the program statements or nodes in the program. Nevertheless, they differ in the time complexity.

The control dependence relationship is formed due to the existence of two paths from the predicate that control a particular statement to the last statement in the program. Sometimes, one of the paths contains one or more `goto` statements; in this case, these statements have to be included in the slice because it is considered as a part of the control dependence relation. Weiser [2] could not properly include relevant goto statements, while Ottenstein [3] did not design the PDG-based slicing to address unstructured programs. Many works tried to address this problem such as Ball and Horwitz [4] as well as Choi and Ferrentai [5] who proposed to treat the `goto` statements as predicates that have two successors. These works suffer from over-approximation due to including irrelevant or fake predicates [6]. Harman and Danicic extended Agrawal's algorithm [7] by using a refined criterion for slicing the `goto` statements. Harman and Danicic's algorithm is imprecise with `switch` statements [8]. Kumar and Horwitz [8] suggested an improvement to the PDG-based slicing approach for slicing relevant jump (`goto`) statements.

Most of the works fail to compute the relevant `goto` statement. This research question took about one decade to reach an acceptable solution. Most of these works did not solve this research question because they separate the computations of the control dependencies and their relevant `goto` statements. They obtained the control dependencies from the post-domination facts, while they considered the relevant `goto` statements as a slicing issue. Moreover, they neglected the relevant predicates whose structured flows are also involved in the paths forming the target control dependence. This separation between (or computing gradually) the elements of one control dependence causes the loss of a solid work that draws a complete picture. This work deals with this challenge through computing all the elements of the control dependency as described earlier.

The first program representation that was proposed to preserve the location-based information was emerged in our earlier works [22,23]. These two works present a new light-weight program representation that is based on Predicated Control Blocks (PCB). While each PCB represents a conditional statement, that preserves the syntactic structure of the program. Moreover, it enables us to introduce an efficient on-demand slicing approach. Both PCB-graph and label-axis collect location-based information with the program flows in their graphs; however the PCB-graph aims at preserving the syntactic structure; whereas the label-axis saves the locations of the flows.

9 Conclusions

This paper presents the first approach that can obtain the control dependencies with its relative flows without making a comprehensive analysis. This approach employs multiple types of information; besides, it builds on the location-based information, the syntactic structure and the program flows. The location-based information improves the analysis because it allows getting on-the-fly the control scope of every predicate, the statements or nodes that are certainly not controlled by any predicate, the statements which post-dominate all their previous statements, and the statements which are controlled by the predicate of their conditional statements. In comparison with the state-of-the-art approach, the proposed approach uses much less amount of information to compute one control dependence with its relative facts. Therefore, there is a reason to believe that the new approach is faster than the state-of-the-art approach in computing the control dependencies for few number of statements.

A significant enhancement to our previous work [9] is done by introducing the sub-figure concept. This work enhances the previous work in two aspects. Firstly, it optimizes significantly the work of Theorem 2 by introducing the sub-figure concept that allows eliminating unnecessary traversing of paths beyond the boundaries of overlapping flows. Secondly, it introduces the label-axis program representation, which is an efficient in detecting fast many facts such as the boundaries of the overlapping flows, the overlapping and intersection between the flows, the scope of the predicates and the nesting between the overlapping flows sets. Finally, it is used to facilitate the computation of the control dependencies.

This work converts the overlapping flows initialized in [9] from an algorithm to a foundation that could be used in many static program analyses. The new foundation that we present in this paper enables us to compute on-the-fly the post-domination facts, control dependencies with its relevant flows together for a particular statement. This indicates the efficiency of the new foundation.

References

1. Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.
2. Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
3. Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.
4. Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *International Workshop on Automated and Algorithmic Debugging*, pages 206–222. Springer, 1993.
5. Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1097–1113, 1994.
6. Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance: Research and Practice*, 10(6):415–441, 1998.
7. Hiralal Agrawal. On slicing programs with jump statements. In *ACM Sigplan Notices*, volume 29, pages 302–312. ACM, 1994.
8. Sumit Kumar and Susan Horwitz. Better slicing of programs with jumps and switches. In *International Conference on Fundamental Approaches to Software Engineering*, pages 96–112. Springer, 2002.
9. Husni Khanfar, Björn Lisper, and Saad Mubeen. Demand-driven static backward slicing for unstructured programs. Technical Report MDH-MRTC-324/2019-1-SE, School of Innovation, Design and Engineering. Malardalen University, May 2019.
10. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 2015.
11. Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):27, 2007.
12. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
13. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.
14. Alfred V Aho and Jeffrey D Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., 1977.
15. Alfred V Aho and Jeffery D Ullman. Lr (k) grammars. In *The theory of parsing, translation, and compiling*, volume 1, pages 371–379. Prentice-Hall Englewood Cliffs, NJ, 1972.
16. Stephen Alstrup, Dov Harel, Peter W Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.
17. Adam L Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 279–288. ACM, 1998.
18. Matthew S Hecht. *Flow analysis of computer programs*. Elsevier Science Inc., 1977.
19. Matthew S Hecht and Jeffrey D Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal on Computing*, 4(4):519–532, 1975.
20. Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.
21. Paul W Purdom Jr and Edward F Moore. Immediate predominators in a directed graph [h]. *Communications of the ACM*, 15(8):777–778, 1972.
22. Husni Khanfar, Björn Lisper, and Abu Naser Masud. Static backward program slicing for safety-critical systems. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 50–65. Springer, 2015.
23. Husni Khanfar and Björn Lisper. Enhanced PCB-based slicing. In *Fifth International Valentin Turchin Workshop on Metacomputation*, page 71, 2016.