Correspondence

Comments on Program Slicing

HARETON K. N. LEUNG and HASSAN K. REGHBATI

Abstract—This correspondence points out some of the problems with Weiser’s algorithm [5] for computing program slices. Corrections are made to Weiser’s algorithm. It is shown how Weiser’s algorithm can be amended to handle loops. Advantages of the Bergerett [27] and Carré’s approach [1] are discussed.

Index Terms—Data flow analysis, debugging, information flow, slicing.

In a recent paper Weiser introduces the method of program slicing [5]. Program slice is a useful program element with applications in program testing, parallel processor distribution [4], maintenance, and debugging [3]. We found that Weiser’s paper [5] contains several errors which warrant correction. There are several places where the notations used are confusing and ambiguous. For instance, a minor improvement can be made in the definition of Proj_{c,v}\:s:

Definition: Let \( T = (t_1, t_2, \ldots, t_n) \) be a state trajectory, \( n \) any node in \( N \) and \( \ldots \). To clearly show that node \( n \) does not affect the endpoint of the trajectory \( T \), \( t_i \) should be used instead of \( t_n \) in the expression for \( T \).

A major inconsistency occurs in Weiser’s definitions of the projection function \( \text{Proj} \) and the program slice. This inconsistency shows up in the second property outlined in his definition of a slice. The claim about \( \text{Proj}_c(T) \neq \text{Proj}_c(T') \) does not hold. Consider the example program shown in Fig. 1. Let \( C = <9, \{X\}> \) and \( S_c \) be the program slice generated from the slicing criterion \( C \) (\( S_c \) is also shown in Fig. 1). If follows that \( C' = <\text{succ}(9), \{X\}> = <12, \{X\}> \), since line 12 is the nearest successor of line 9 in the original program which is also in the slice. Note that line 9 itself is not in the slice. Let the input be \( I = (2, 3, 4) \). One can easily show that the state trajectory \( T \) for the program, based on input \( I \), would be \((1, s_1), (2, s_2), (3, s_3), (4, s_4), (5, s_5), (7, s_7), (8, s_8), (9, s_9), (10, s_{10}), (11, s_{11}), (12, s_{12})\). The state trajectory \( T' \) for \( S_c \) based on input \( I \) would be \((1, s_1), (2, s_2), (12, s_{12})\). According to the definition of \( \text{Proj} \):

\[
\text{Proj}_c(T) = \text{Proj}_c(1, s_1) \text{Proj}_c(2, s_2) \cdots \text{Proj}_c(12, s_{12})
\]

\[
= \lambda \lambda \lambda \lambda \lambda \lambda (9, s_9) \lambda \lambda
\]

\[
= (9, s_9)
\]

and

\[
\text{Proj}_c(T') = \text{Proj}_c(1, s_1) \text{Proj}_c(2, s_2) \text{Proj}_c(12, s_{12})
\]

\[
= \lambda \lambda (12, s_{12})
\]

\[
= (12, s_{12})
\]

Thus, \( \text{Proj}_c(T) \neq \text{Proj}_c(T') \), contradicting Weiser’s definition of a slice.

The above difficulty can be corrected by redefining the projection function as follows:

Definition: Let \( T = (t_1, t_2, \ldots, t_n) \) be a state trajectory, \( n \) any node in \( N \), and \( s \) any variable from variable names to values. Then:

\[
\text{Proj}_{c,v}(T, s) = \begin{cases} 
\lambda & \text{if } n \text{ does not affect the value of } V \text{ at } i \\
(s, v) & \text{otherwise}
\end{cases}
\]

where \( s \mid V \) is \( s \) restricted to domain \( V \), and \( \lambda \) is the empty string.

Applying the above definition to the same example, we have:

\[
\text{Proj}_c(T) = \text{Proj}_c(T') = (2, s_2).
\]

This satisfies the property of a slice as defined by Weiser.

The algorithm given for finding a slice requires improvement. The difficulty is caused by Weiser’s ambiguous use of “+” for immediate successor. Weiser defines the statements which have direct effects on the slicing criterion as

\[
S_c = \{ \text{for all } n \text{ such that } R_c(n + 1) \cap \text{DEF}(n) \neq \emptyset \}
\]

The relationship between nodes \( n \) and \( n + 1 \) requires clarification. Is node \( n + 1 \) the immediate successor of node \( n \)? How to handle the case of a node having more than one immediate successor (for example, if \( + \) then \( \cdots \) else statement)? We propose a more appropriate definition of \( S_c \):

\[
S_c = \{ \text{for all } n \text{ such that } R_c(n) \cap \text{DEF}(n) \neq \emptyset \}
\]

for any node \( n \) which is an immediate successor of node \( n \).

Another problem with Weiser’s algorithm is that, according to the definition of \( B_c \), \( B_c \) will be empty for any program. Since for any branch statement \( n \), \( \text{DEF}(n) = \emptyset \), it is easy to verify that \( B_c \) will not contain any branch statements. Thus, for any node \( n \in \text{INFL}(n) \), \( B_c \) will be empty. Consequently, the definition

\[
B_c = \bigcup_{n \in \text{INFL}(n)}
\]

defines an empty set. The statements of interest are the branch statements which can select to execute or not execute some state-
ment in $S_C^b$. A correct definition of $B_C^0$ should resemble the following:

$$B_C^0 = \text{all nodes } b \text{ s.t. } n \in S_C^b \quad \text{and} \quad n \in \text{INFL}(b).$$

It is not clear how Weiser's algorithm deals with loops. Consider the program in Fig. 2. Suppose the slicing criterion is $C = <4, \{X\}>$. Then, following Weiser's algorithm, node 4 will be included in $S_C^b$. Since there are many paths to reach a slicing node (node 4 in our example) within a loop, the fundamental question is which path should be used to generate the desired $R_C^0(n)$, $B_C^0$, and $S_C^b$. When a loop is iterated more than once, the statements near the end of the loop may affect those at the beginning. Therefore, for a slicing criterion $<j, V>$ where the slicing node $j$ is in a loop, the procedure in computing $R_C^0(n)$, $B_C^0$, and $S_C^b$ must include the effect of every statement within the loop. Due to the data flow relation [1], one can easily verify that iterating a loop more than twice will not change the statements included in $S_C^b$.

Thus, for a slicing node $j$ within a loop, the proper procedure in finding $R_C^0(n)$, $B_C^0$, and $S_C^b$ should treat every node within the loop as a predecessor of $j$.

Based on the above observations, Weiser's algorithm should be amended as follows:

The full definition of the influence at level $i + 1$ is the following. For all $i \geq 0$:

$$R_C^{i+1}(n) = R_C^i(n) \cup R_C^0(n)$$

where $BC(b)$ is the branch statement criterion, defined as $<b$, \text{REF}(b) $>$. $S_C^{i+1} = \text{all nodes } n \text{ s.t. } n \in B_C^i$ or $R_C^{i+1}(n) \cap \text{DEF}(n) \neq \emptyset$, $m$ is an immediate successor of $n$.

$B_C^{i+1} = \text{all nodes } b \text{ s.t. } n \in S_C^{i+1}$ and $n \in \text{INFL}(b)$.

For a slicing node within a loop, all nodes within the loop must be treated as predecessors of the slicing node.

Consider applying the above algorithm to the program of Fig. 2, using the slicing criterion $C = <4, \{X\}>$. Intuitively, one would expect lines 2-5 to be in the slice since all these lines are indirectly involved in computing the value of $X$. Table I shows some of the intermediate results in computing the slice. We had treated node 5 as an immediate predecessor of node 4. From the results of $R_C^0$, $S_C^b$ is derived to be $\{4\}$. $B_C^0$ is then found to be $\{3\}$. It follows that the branch statement criterion is $<3, \{K\}>$, since $K$ is referenced in line 3. The values for $R_C^0(n)$ can then be computed. Next, $R_C^1(n)$ is formed by taking the union of $R_C^0(n)$ and $R_C^0(n)$. Finally, $S_C = \{2, 3, 4, 5\}$ is derived. Further iterations do not change the content of the slice. Thus, the slice on criterion $<4, \{X\}>$ is

1. BEGIN
2. $K = 1$
3. WHILE $K < 10$ DO
   4. $X = K + 1$
   5. $K = K + 1$
END;
8. END.

If one had used Weiser's algorithm, one would have obtained $S_C^b = \{4\}$, $B_C^0 = \emptyset$. Further iteration would give the slice as having only:

1. BEGIN
2. $X = K + 1$
3. END.

In a recent paper, Bergeretti and Carre [1] described the information-flow relations. The information-flow relations are helpful both in program testing and in checking the consistency of assertions with a program text. These relations are also useful in automatic detection of ill-posed statements, ill-posed imported values and undefined variables during the static analysis of a program. Out of the three binary relations: $\lambda$, $\mu$, and $\rho$, the relation $\mu$ can be used to compute program slices. The $\mu$ relation relates the values of the "expression" parts within any program statement $S$ and the values of the program variables on exit from $S$. Bergeretti and Carre's slicing algorithm involves first calculating all the information-flow relations, $\mu$, which has a time bound of $O(|E| \times |V|^2)$ and then determining the slice in $O(|E|)$ time, where $|E|$ is the number of expressions and $|V|$ is the number of variables in the program. Let $n$ be the number of statements in the program. For a worst case assumption, $|E| = n$ and $|V| = n$. Consequently, Bergeretti and Carre's approach gives a time bound of $O(n^3)$. By contrast, Weiser's algorithm has a time bound of $O(ne \log e)$, where $e$ is the number of edges in the flow graph of the program. Under the worst case assumption of $e = n^2$, the time bound becomes $O(n^7 \log n^2)$, which is slightly better than Bergeretti and Carre's algorithm. However, Bergeretti and Carre have found the running times of their algorithm to be quite acceptable on their test cases because the relations are extremely sparse. More empirical data are needed to determine the relative merits of these two algorithms. One advantage of Bergeretti and Carre's approach is that once all the $\mu$ relations are calculated, any program slice can be computed in $O(|E|)$ additional time by scanning the $\mu$ relation matrix for the variable(s) of interest.

### REFERENCES