Effective Slicing: A Generalization of Full and Relevant Slicing [*]

Anne Mulhern
mulhern@cs.wisc.edu

Ben Liblit
liblit@cs.wisc.edu

University of Wisconsin–Madison

June 24, 2008

Abstract

A program slice, P', is the part of a program, P, that may affect the value of a set of variables, V, at a program point, p. Informally, P' is *well-behaved* if it calculates the same values for V at p as P. A well-behaved slice exhibits *good behavior*. A static slice must be well-behaved for every input while a dynamic slice must be well-behaved for just one input.

A *union slice* is the union of several dynamic slices calculated with respect to different inputs but the same V and some occurrence(s) of p in the program's execution history. A *realizable slice* is a union slice calculated with respect to all initial states. A realizable slice is, in general, not computable.

Well-behaved union slices allow reasoning about program behavior on sets of inputs, but existing union slicing algorithms may yield ill-behaved slices, i.e., for some input among those used to construct the dynamic slices, the union slice will calculate incorrect values for V.

We find that bad behavior of union slices is an artifact of the particular dynamic slicing algorithm, *full slicing*, used to calculate the individual slices. Full slicing is the name given to the originally proposed version of dynamic slicing, to distinguish this version from other variants of dynamic slicing that have arisen since. In contrast, the unions of *relevant slices* do yield well-behaved slices.

We propose a generalization of full and relevant slices, *effective slices*, that can be used to calculate unions of dynamic slices that are more precise than the unions of relevant slices, but still well-behaved. We extend the generalization to *scant slices*. We show that the nodes in the set differences between unions of different kinds of slices have certain properties useful in debugging.

1 Introduction

A *program slice* [15], P', is a subset of a program, P. Slicing algorithms are generally defined on an intermediate, graph-based representation of P. P' is then a subset of the nodes of P. A program slice is calculated with respect to a *slicing criterion*. The slicing criterion is a way of focusing an analysis of the program precisely and narrowly. Generally, a slicing criterion specifies at least a program point, p, and a set of variables, V.

Slices can be *static* or *dynamic*. A static slice must be correct for any initial state of the program. A dynamic slice must be correct for just one initial state. The slicing criterion for a dynamic slice includes a specification of the initial state, σ. In more precise slicing algorithms, the particular execution instance, p^i, of p is specified. The superscript indicates the position in the *execution history*, i.e., the sequence of instructions, for a given input state. An *execution slice* is the set of all nodes in the execution history. Since it contains all nodes executed for a given initial state it must be correct for that initial state; an execution slice is a dynamic slice, albeit a conservative one.

[*]Supported in part by AFOSR Grant FA9550-07-1-0210 and NSF Grants CCF-0621487, CCF-0701957, and CNS-0720565. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFOSR, NSF, or other institutions.

	P	Prec0	Prec1
1	`if(n == 0)`		
2	`  x = 3;`		`x = 3;`
	`else`		
3	`  x = 3;`	`x = 3;`	
4	`z = x;`		
	Criterion	$V = \{x\}$ $p = 4$	$V = \{x\}$ $p = 4$

Figure 1: Example showing a program with two precise slices

There are several definitions of correctness for program slicing. Weiser's original definition requires that the slice calculate the same values for V at p as the program itself. This definition requires that the slice be *executable*, i.e., that it be a program. A *closure slice* is a slice that contains all nodes that might have an effect on the criterion but which may not be executable. *Amorphous* slicing techniques [10, 11] transform a closure slice to an executable slice while preserving correct semantics. Additionally, this definition implies that the slice must be a *backward slice*, i.e., a set of nodes that precede p in program execution. A *forward slice* is a slice consisting of nodes that follow p during program execution. There is no analogous definition of correctness for forward slicing [5]. We call a slice that satisfies Weiser's criterion *well-behaved*.

A union slice [3, 14] is the union of dynamic slices where V and p are held constant but σ is allowed to vary. The criterion for a union slice thus includes not just one initial state, but rather a set of initial states. In earlier work [3], only the dynamic slice for the final instance of p was used to form the union. In later work [14], all instances are used. A *realizable slice* is a union slice calculated with respect to all inputs.

A *precise slice* is a minimal set of nodes that are correct for all possible executions for a particular slicing criterion. Precise slices are in general undecidable, but are contained in the corresponding static slice. We observe that a program may have multiple precise slices. Figure 1 shows such an example. Here the then branch and the else branch are identical so either is a minimal slice.

Well-behaved union slices, i.e., union slices that are well-behaved for every initial state used to construct the slice, have applications where a subset of initial states are of interest. For example, several inputs may be found to trigger just one bug [2]. In correcting the bug, it is useful to examine the well-behaved union of the dynamic slices for each initial state. However, union slices in general are not well-behaved [9, 13], i.e., they may yield incorrect values for one of the initial states used in forming the slice. A realizable slice may be ill-behaved as well.

Union slices are defined in terms of *full slices*, the original dynamic slices. We have observed that the unions of *relevant slices* [1], a variant of dynamic slices, are well-behaved regardless of the differences between their slicing criteria. *Semantic effect*, a concept introduced in the context of static slicing, allows to distinguish between relevant and full slices, and explain bad behavior of union slices. We develop the concept of *effective slicing*, a generalization of relevant and full slicing. We show that the unions of effective slices are more precise than the unions of relevant slices yet well-behaved, unlike the unions of full slices. We introduce another form of slicing, *scant slicing*, which extends our generalization further. We explore what it means for a node to be in the union of one kind of slice but not another, and suggest some uses for these difference sets in debugging.

2 Unions of Full Slices

Dynamic slicing algorithms vary in precision: some track dependencies, while other only record which nodes of a program were executed [17]. In the rest of the paper, we assume a fairly precise dynamic slicing algorithm that records which definition of a variable reaches a use of that variable each time a node is reached during execution of the program.

Different execution instances of the same node are distinguished by a superscript. The value of the superscript is the position of the execution instance of the node in the execution history. In the execution slice for input 2, E2, in Figure 2, node (6) is the fifth node to be reached in the execution of the program; this execution instance is uniquely

	P	X1	F1	X2	F2	U	R1	R2
1	`read(n);`	`read(n);`		`read(n);`			`read(n);`	`read(n);`
2	`x = 1;`	`x = 1;`	`x = 1;`	`x = 1;`		`x = 1;`	`x = 1;`	
3	`y = 2;`	`y = 2;`		`y = 2;`	`y = 2;`	`y = 2;`		`y = 2;`
4	`if (n == 1)`	`if (n == 1)`		`if (n == 1)`			`if (n == 1)`	`if (n == 1)`
5	`  y = 1;`	`  y = 1;`					`  y = 1;`	
6	`if (y == 2)`	`if (y == 2)`		`if (y == 2)`	`if (y == 2)`	`if (y == 2)`	`if (y == 2)`	`if (y == 2)`
7	`  x = 2;`			`  x = 2;`	`  x =2;`	`  x = 2;`		`  x = 2;`
8								
	Criterion	$V = \{x\}$ $p = 8^7$ n = 1	$V = \{x\}$ $p = 8^7$ n = 1	$V = \{x\}$ $p = 8^7$ n = 2	$V = \{x\}$ $p = 8^7$ n = 2	$V = \{x\}$ $p = 8^7$ n = $\{1, 2\}$	$V = \{x\}$ $p = 8^7$ n = 1	$V = \{x\}$ $p = 8^7$ n = 1

Figure 2: Example program showing the execution slice (X1) and a full slice (F1) of the program on input 1, the execution slice (X2) and a full slice (F2) of the program on input 2, the union slice (U) for inputs 1 and 2, and a relevant slice for input 1 (R1)

identified as 6^5.

2.1 Unions of Full Slices are Ill-behaved

Figure 2 shows an example where the union of two full slices, themselves well-behaved, is ill-behaved. X1 is the execution slice for input 1. In calculating the full slice, the definition of x that reaches node (8) is located and its dependencies are calculated. Since node (7) is not executed, the definition at node (2) is used. It has no dependencies, so the full slice, F1, is the singleton set, $\{2\}$.

X2 is the execution slice for input 2. Here node (7) is executed and so backward dependencies from node (7) are calculated. Node (7) is control dependent on node (6) and the assignment to y at node (3) is the one that reaches the conditional at node (6), so the slice is the set $\{3, 6, 7\}$.

Full slicing algorithms yield well-behaved slices.

Proof. Let F be a full slice for initial state σ. Let P be the original program. If F is not well-behaved there must be some first node, e, on which the execution of F differs from P and affects the slicing criterion. There must be some node in $P \setminus F$ that causes the difference in execution. If the node is in F then e is not the first node to have different behavior and so there is a contradiction. Call this node d. But d must be in F since it affects e during execution of P on σ. $\square$

However, the union of full slices may be ill-behaved. In the example the union of the two slices, $\{2, 3, 6, 7\}$, is ill-behaved on input 1. For both inputs, the slice did not contain the read statement at node (1). Yet the value of x at node (8) is determined by the input and is different depending on whether the input value is 1 or 2. Clearly, some nodes that matter to the value of x at node (8) are omitted in full slicing.

Unions of full slices taken with respect to the same initial state but different execution instances of p are also ill-behaved. Figure 3 is a slightly modified version of our previous example where n is a loop variable rather than an input variable. The full slices taken with respect to the two different execution instances of the same node are almost identical to those taken with respect to the two different inputs in the previous example. The only difference is that node (1) is included in the full slices for the example in Figure 3 because the other nodes are control dependent on it. Although the individual slices are well-behaved for their execution instance, their union is not, just as before.

2.2 Realizable Slices Are Ill-behaved

The program in Figure 2 has just two possible executions. One execution occurs exactly when n is 1; the other for every other value of n. The realizable slice is computable in this case; it is exactly U. We have already shown that U is ill-behaved. A realizable slice composed from the union of full slices on all initial states may be ill-behaved.

	P	F1	F1
1	`for (n = 1:2) {`	`for (n = 1:2) {`	`for (n = 1:2) {`
2	`  x = 1;`	`  x = 1;`	
3	`  y = 2;`		`  y = 2;`
4	`  if (n == 1)`		
5	`    y = 1;`		
6	`  if (y == 2)`		`  if (y == 2)`
7	`    x = 2;`		`    x = 2;`
8			
	Criterion	$V = \{$`x`$\}$ $p = 8^7$	$V = \{$`x`$\}$ $p = 8^{14}$

Figure 3: Example program showing the full slice with respect to the same node at different execution instances. F1 is the full slice taken with respect to node (8) on the first iteration of the loop, F2 on the second

	P	S1	S2
1	`x = 2;`	`x = 2;`	
2	`y = 2;`	`y = 2;`	
3	`...`		
4	`x = 1;`		`x = 1;`
5	`y = 3;`		`y = 3;`
6	`z = x * y;`	`z = x * y;`	`z = x * y;`
7			
	Criterion	$V = \{$`z`$\}$ $p = 7$	$V = \{$`z`$\}$ $p = 7$

Figure 4: Example program showing two well-behaved slices for the same slicing criterion

3 Semantic Effect

Good behavior in dynamic slices does not guarantee good behavior in their union. A similar problem arises in static slicing. It has been observed [12, 13] that Weiser's criterion allows multiple correct static slices. In fact, a correct, although computationally expensive algorithm, is one that enumerates every subset of P and selects any subset that calculates the correct values for V at p. A slice calculated in this way may be misleading. One expects that the nodes of P' should be those that affect the value of V at p when P executes, but it is possible that P' is a program that only coincidentally yields the correct values for V at p [12]. Figure 4, adapted from Kumar and Horwitz, shows two well-behaved slices taken with respect to the same slicing criterion. The assignments to `x` and `y` at nodes (1) and (2) are killed by the assignments at nodes (4) and (5). S1 is a well-behaved slice since it will always calculate the same value for `z` at node (7) as P. However, S2 is a more useful slice, since it contains the assignments to `x` and `y` that reach the computation of `z` in P.

Kumar and Horwitz [12] define correctness in terms of *semantic effect*. A correct slice is a slice that contains a superset of the nodes that have a semantic effect on the slicing criterion. A node has a semantic effect on the slicing criterion if changing the node in a prescribed way might cause the value of a variable in V at p to change. Semantic effect is decided based only on the data and control dependencies in the program; without any additional information from, e.g., symbolic evaluation. By this definition, S1 is an incorrect slice, since it does not include nodes (5) and (6). The algorithm which Kumar and Horwitz describe yields closure slices but does not guarantee that they be executable. We call slices that are executable and correct by Kumar and Horwitz's definition *intentionally well-behaved*, since they do not produce the correct values by accident.

The notion of semantic effect has not previously been used with respect to dynamic slicing. However, variants of dynamic slicing can be distinguished by examining their algorithms through the lens of semantic effect. In particular, relevant slicing and full slicing differ only in how semantic effect is decided.

3.1 Relevant Slicing

Relevant slicing [1] is a variant of dynamic slicing. A relevant slice is a superset of the corresponding dynamic slice. In the formalization of Binkley et al. [4], full slicing is *weaker* than relevant slicing. Full slices include only nodes with a *positive* effect, i.e., nodes that may transitively or directly affect the slicing criterion. Relevant slices also include nodes with a *negative* effect, i.e., that cause a statement that otherwise might affect the slicing criterion not to execute. Relevant slicing was motivated by the observation that bugs may be caused by errors of omission as well as commission.

Figure 2 shows R1, a relevant slice for input 1. A relevant slicing algorithm would determine that the value of `x` at node (8) has a potential dependence on node (6) via node (7), since, had the conditional at node (6) evaluated differently, the assignment to `x` at node (7) might have been reached. Therefore, node (6) is included in the relevant slice. Node (6) is data dependent on node (5) which is control dependent on node (4), itself data dependent on node (1). The entire relevant slice is $\{1,4,5,6\}$. Significantly, this slice includes the `read` statement, unlike the equivalent full slice.

Relevant slicing algorithms are generally defined by modification to full slicing algorithms [8]. In the next section we compare full and relevant slicing algorithms through the lens of semantic effect.

3.2 Semantic Effect in Dynamic Slicing

Unlike static slicing, dynamic slicing uses a record of the program's execution. This record yields important negative information, e.g., that a particular program point was not reached during execution or that a particular definition did not reach a use. Such negative information is not available in static slicing, hence is not considered when deciding semantic effect. In full and in relevant slicing, an unexecuted node may not have a semantic effect. However, they differ in the way unexecuted nodes are taken into account in deciding the semantic effect of executed nodes. In full slicing unexecuted nodes are ignored; in relevant slicing all unexecuted nodes are taken into account.

Figure 2 illustrates this difference. In the full slice for input 1 node (7) is ignored when deciding the semantic effect of other nodes; in the relevant slice, R1, node (7) is taken into account. Consider what would happen if `y` were assigned a different value at node (5). In that case, since the condition at node (6) is dependent on the definition of `y` at node (5), the branch might go in the opposite direction. If it did, node (7) would be executed and a different definition of `x` would reach node (8). In the full slice, since node (7) is ignored, this chain of possible changes in program execution is seen as irrelevant, and so node (5) is not included in the slice. In the relevant slice, on the other hand, node (7) is taken into account; this chain of possible changes in program execution is seen as significant and so node (5) is included in the slice.

Agrawal et al. [1] suggest *approximate relevant slicing* as an alternative to relevant slicing where the presence of pointers makes deciding semantic effect more difficult. In approximate relevant slicing, every conditional node that occurs in the execution slice is judged to have a possible semantic effect. Approximate relevant slices are cheaper to compute but less precise [8] than relevant slices.

3.3 Relevant Slice Unions are Well-Behaved

Since relevant slicing algorithms select a superset of the nodes that satisfy semantic effect with respect to all nodes in the program, unions of relevant slices are well-behaved. This is the case regardless of how the criteria differ.

Proof. Let U be the union of relevant slices $\{R_1,\ldots,R_n\}$ taken with respect to criteria $\{C_1,\ldots,C_n\}$. Assume that for some C_e in $\{C_1,\ldots,C_n\}$ there is a first node reached during execution of U on which U executes differently from P and where this execution affects the value of the slicing criterion. Call this node e. e can execute differently for one of two reasons. There may be a node in U, e', such that e' executed differently in P and this difference caused e to execute differently. If that is the case, then e is not the first node in U to execute differently and to affect the slicing criterion, and there is a contradiction. So it must be the the case that there is a node in P and not in U, d, such that its execution affected the execution of e. But there must be a node in P such that its execution prevented d from being executed when P was executed on C_e, otherwise d would be included in R_e and hence in U. Call this node c. c must be included in R_e, since c has a semantic effect on the slicing criterion through d. P and U must evaluate in the same way on c, since otherwise e is not the first node in P to evaluate differently and affect the slicing criterion. Therefore, d is not reached by P. Therefore, U is well-behaved with respect to criteria $\{C_1, \ldots, C_n\}$. $\square$

In the example in Figure 2, the union of the two relevant slices, R1 and R2, is the whole program, so R1 $\cup$ R2 is vacuously well-behaved. Suppose, however, that relevant slices are constructed using the set of inputs $\{2,3,4,5\}$. In each case, the program behaves exactly as for input 2. R2, R3, R4, and R5 are consequently equal. Call their union R. We show how the general proof above can be applied to this concrete case. Take each node in R in turn and consider whether it is possible for that node to be the first node reached during execution of R on any input in $\{2,3,4,5\}$ that behaves differently than in P when P is executed on the same input. Consider node (1). It cannot behave differently, as it is a `read` statement and not dependent on any previous statement. The same argument applies to node (3) and node (7), which are assignments of constants to variables. This leaves node (4) and node (6). If node (4) executes differently, then node (2) must have affected it, since node (2) is the only node in P and not in R that precedes node (4) in execution order. But node (2) did not affect node (4) since node (4) does not depend on the value of `x`. Finally, consider node (6). Then, node (6) must have been affected by the execution of node (2) or node (5). The conditional at node (6) does not depend on the value of `x`, so node (2) can be ignored. If node (5) is reached by P on any input state in $\{2,3,4,5\}$ then node (6) might execute differently in R than in P. But node (5) is control-dependent on node (4), which was included in R2, R3, R4, and R5 because it had a semantic effect on the slicing criterion through node (5). And, the conditional at node (4), when reached with initial state in $\{2,3,4,5\}$, evaluates to false, causing node (5) not to be reached, otherwise node (5) would have been included in one of R2, R3, R4, or R5. Thus node (5) was not reached during the execution of P on input 2. So R is well-behaved.

Since unions of relevant slices are well-behaved, a realizable slice constructed from relevant slices must be well-behaved for every initial state.

3.4 Relation to Other Kinds of Slices

The union of relevant slices, R, for the same V and over all execution instances of p, but where the initial state, σ, is allowed to vary, is a subset of the corresponding static slice, S. It is a subset of the union of execution slices, X, as well. $(S \cap X) \setminus R$ contains nodes that were executed, might have affected the slicing criterion, but certainly did not for any of the initial states. For the example in Figure 2, the reader will observe that the static slice for `x` at node (8) is the whole program. Node (2) occurs in the execution slices for inputs $\{2,3,4,5\}$ but does not occur in the union of relevant slices. It is excluded because, for these initial states, the assignment to `x` at node (2) is always killed by the subsequent assignment at node (7).

Consider the example in Figure 2. For inputs $\{2,3,4,5\}$, R2 $\cup$ R3 $\cup$ R4 $\cup$ R5 contains more nodes than are needed to ensure good behavior. This observation is the motivation for *effective slicing*, our generalization of full and relevant slicing.

4 Effective Slicing

In effective slicing the set of nodes taken into account in deciding semantic effect, *SE*, is a superset of the executed nodes. If *SE* is exactly the executed nodes, effective slicing is the same as full slicing. If *SE* is all nodes, effective slicing is the same as relevant slicing. There exist cases where a set in between these two extremes is desirable.

4.1 Unions of Effective Slices

4.1.1 In Debugging

Consider the case where m inputs are known to trigger one bug. In that case, there are m initial states, $\{\sigma_1, \ldots, \sigma_m\}$. If the bug manifests in a single location, the well-behaved union of dynamic slices taken with respect to all inputs captures all behaviors belonging to those inputs that are relevant to the bug. We have shown that the union of relevant slices is well-behaved. It is also the case that the union of effective slices, where *SE* is the set of all nodes in $\bigcup_{k \in \{1,\ldots,m\}} X_k$ where X_k is the execution slice for initial condition σ_k, is well-behaved for any σ_j in $\{\sigma_1, \ldots, \sigma_m\}$.

Proof. Let $\{E_1,\ldots,E_n\}$ be the set of effective slices taken with respect to input states $\{\sigma_1,\ldots,\sigma_n\}$. The proof proceeds just the same except for the d case. In that case, c must be included in E_e, unless d is not in *SE*. If c is in E_e then the argument is the same as for relevant slicing. If not, then d was never executed for any σ_k in $\{\sigma_1,\ldots,\sigma_n\}$. Thus, it can not have been reached during the execution of P on σ_e, so again there is a contradiction. □

Since *SE* may be significantly smaller than the whole program, the cost of pointer analysis in deciding semantic effect may be significantly less than that for calculating precise relevant slices. A realizable slice of effective slices, where *SE* is taken to be the union of all execution slices for every initial state, is well-behaved, unlike the union of full slices, and more precise than the union of relevant slices.

4.1.2 Dynamic Slicing is not Monotonic

Beszédes et al. use the full slice with respect to the final occurrence of p in forming union slices for their work on slicing C programs [3]. In subsequent work on Java programs [14], however, their slicer accumulates the nodes from slices calculated with respect to multiple execution instances, which yields larger, hence better, union slices. From the example in Figure 3 it is apparent that precise dynamic slicing in general is not monotonic in the execution order, i.e., the slice with respect to one execution instance may be neither a superset nor a subset of the slice with respect to a different execution instance. In forming union slices in order to approximate realizable slices, the union of the dynamic slices for all occurrences of p will be larger than the dynamic slice for the final occurrence of p.

4.1.3 Difference with Union of Relevant Slices

E, the union of effective slices, is contained in R, the union of relevant slices. The nodes in $R \setminus E$ have a particular kind of effect on the slicing criterion; they cause, transitively or directly, an assignment that would otherwise affect the slicing criterion to be omitted, and this assignment is omitted in every execution slice. Consider the union of effective slices for inputs $\{2,3,4,5\}$ for the example in Figure 2. Node (5) is not reached for any execution slice, so $SE = \{1,2,3,4,6,7\}$. Recall that the behavior of the program on each of these inputs is identical. Consider the nodes in E2. Node (1) and node (4) have no semantic effect on the slicing criterion, since the assignment at node (5) is ignored. Node (2) is excluded for the same reason it was excluded in calculating R2. Therefore, the effective slice for input 2, hence the union of the effective slices for inputs $\{2,3,4,5\}$, is exactly equal to the full slice, F2. The union of relevant slices for these inputs is exactly R2. The difference between R2 and F2 is $\{1,4\}$, the set of nodes that caused the assignment at node (5) to be omitted for every initial state. Another way of stating the effect of the nodes in $R \setminus E$ on the slicing criterion is that these nodes behaved the same way for the subset of initial states considered, and that their effect was always negative.

This sort of information is useful in debugging. If the cause of a bug is that some action was not taken, then it is the nodes in $R \setminus E$ that are responsible.

4.2 Unions of Full Slices

The union of full slices (F) is a subset of the union of effective slices (E) for the same set of initial states. F is not well-behaved even for the initial states for which the slices were constructed. The nodes in $E \setminus F$ are nodes that had only a negative effect, i.e, caused an assignment that otherwise would have affected the slicing criterion not to be executed, but where the assignment occurred in an execution slice for at least one of the initial states.

This behavior arises in our motivating example (Figure 2) and in other situations like it. The condition at node (4) evaluates differently depending on whether the input is 1 or 2. However, the effect in both cases is to cause an assignment, that otherwise would have affected the value of the slicing criterion, to be omitted. It is clear that some nodes in $E \setminus F$ must have had different behaviors for different initial states. If every node had behaved identically and negatively, then these nodes would have been in $R \setminus E$. It is not the case that all nodes had different behaviors; a conditional may have multiple data dependencies, only one of which has a different behavior. However, if the conditional is in E but not in F it is possible that all the assignment statements on which it is dependent will be omitted from F.

In the following section we introduce our extension of effective slicing, *scant slicing*. The unions of scant slices have a similar relationship to the unions of full slices as the unions of effective slices have to unions of relevant slices.

	Nodes with Effect				
Nodes in the Union of	None	Uniform Negative	Non-uniform Negative	Uniform Positive	Non-uniform Positive
execution slices	✓	✓	✓	✓	✓
relevant slices		✓	✓	✓	✓
effective slices (where *SE* is the union of execution slices)			✓	✓	✓
full slices				✓	✓
scant slices					✓

Table 1: Nodes contained in the unions of different kinds of dynamic slices

4.3 Unions of Scant Slices

The union of *scant slices* (*Sc*) is a subset of the union of full slices (F). For a scant slice, *SE* is calculated as for an effective slice. However, only edges that were followed on some execution are considered when calculating semantic effect. A conditional that always evaluated in the same way has only one outgoing edge in *SE*. Consider the example in Figure 2 for inputs $\{2,3,4,5\}$. For every execution, the true branch from node (6) was always taken. Therefore, in deciding semantic effect, the edge from node (6) to node (8) is ignored. A similar argument applies with regard to node (4). Therefore, node (1) has no semantic effect, since it cannot change the behavior of the conditional at node (4). The assignment at node (2) is overwritten in every execution, hence node (2) has no semantic effect. The assignment at node (3) can not affect node (6) since node (6) has only one outgoing edge. Node (4) has no semantic effect, as it has only one outgoing edge. The same argument holds for node (6). Thus, the only node in the slice is (7), the assignment to x that actually reaches node (8). A scant slice is not the same as a data slice, which includes only assignment statements. Had there been a conditional in the program that had branched in both directions then it might have been included in the slice.

The difference between F and *Sc* is analogous to the difference between R and E. Whereas, $R \setminus E$ contains nodes with a uniform negative effect, $F \setminus Sc$ contains nodes with a uniform positive effect. Table 1 characterizes the nodes contained in the unions of the variants of dynamic slicing discussed in the paper.

5 Applications in Dynamic Dicing

Dynamic program dicing [6] is a technique for using the difference between two dynamically calculated slices. During program execution, one variable may have the correct value while another may be incorrect. The nodes in the difference between the slice for the incorrect value and the slice for the correct value are likely to be nodes involved in the cause of the error. This idea has been extended recently by Zhang et al. [16] who take into account local semantic properties of nodes to exclude from a slice nodes that were necessary to calculate values that are known to be correct.

Dynamic dicing can be extended by using different kinds of slices and their unions. Consider, for example, the execution of a single program. A program point may be reached multiple times during execution, but for only one of those execution instances will a program display incorrect behavior. Consider the union of the effective slices for all correctly executing instances of a slicing criterion, taking *SE* to be the union of execution slices. The nodes in the union are all nodes involved in correct behavior. The difference between the full slice for the incorrectly executing instance and the union of effective slices contains nodes that are likely to contribute to the incorrect behavior. They are likely to cause incorrect behavior because they may have affected the slicing criterion in the incorrect execution, yet they never occurred in a correct execution. Other combinations of different dynamic slicing methods and unions of dynamic slices may also have interesting applications in dynamic dicing.

6 Related Work and Conclusion

That the unions of full slices are not well-behaved has been observed previously [9, 13]. Hall proposed simultaneous dynamic slicing to address this problem [9]. Hall's algorithm iteratively adds nodes to the union slice and does not explain why the unions of dynamic slices are not well-behaved. Conditioned slicing using a condition derived from the disjunction of the initial states is well-behaved [7], but conditioned slicing tools require heavyweight analysis, may be imprecise, and do not scale as well as dynamic slicing tools [17, 18]. Kumar and Horwitz [12] were the first to formalize the concept of semantic effect, but did not apply it to dynamic slicing. Binkley et al. [5] use a semantic meaning function that allows more formal reasoning about semantic effect. Chen and Cheung [6] discuss dynamic program dicing, which takes the difference between the same kind of slice on different criteria, and Zhang et al. [16] extend the idea of dynamic dicing by taking the semantics of individual nodes into account.

We exploit semantic effect to explain the underlying cause for the poor behavior of union slices. We observe that a precise dynamic slicing algorithm may yield slices that contain nodes omitted from a precise static slicing algorithm. We observe that the unions of relevant slices are well-behaved. We introduce effective slicing, a generalization of relevant and full slicing, which yields more precise but still well-behaved slices. We introduce scant slicing, a variant of effective slicing which takes into account the edges traversed in the program graph, rather than only the nodes visited. We examine the set differences between the unions of different kinds of slices and show that the nodes in the sets have certain properties that may prove useful in program understanding and maintenance.

References

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[2] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In S. Matwin and D. Mladenic, editors, *18th European Conference on Machine Learning*, Warsaw, Poland, Sept. 17–21 2007.

[3] A. Beszedes, C. Farago, Z. Szabo, J. Csirik, and T. Gyimothy. Union slices for program maintenance. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 12, Washington, DC, USA, 2002. IEEE Computer Society.

[4] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Ákos Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Sci. Comput. Program.*, 62(3):228–252, 2006.

[5] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, A. Kiss, and L. Ouarbya. Formalizing executable dynamic and forward slicing. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 43–52, Washington, DC, USA, 2004. IEEE Computer Society.

[6] T. Y. Chen and Y. Y. Cheung. Dynamic program dicing. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 378–385, Washington, DC, USA, 1993. IEEE Computer Society.

[7] S. Danicic, A. D. Lucia, and M. Harman. Building executable union slices using conditioned slicing. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, page 89, Washington, DC, USA, 2004. IEEE Computer Society.

[8] T. Gyimóthy, Árpád Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 303–321, London, UK, 1999. Springer-Verlag.

[9] R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Autom. Softw. Eng.*, 2(1):33–53, 1995.

[10] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *J. Syst. Softw.*, 68(1):45–64, 2003.

[11] M. Harman, A. Lakhotia, and D. Binkley. Theory and algorithms for slicing unstructured programs. *Information & Software Technology*, 48(7):549–565, 2006.

[12] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 96–112, London, UK, 2002. Springer-Verlag.

[13] A. D. Lucia, M. Harman, R. Hierons, and J. Krinke. Unions of slices are not slices. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR 2003)*, 2003.

[14] A. Szegedi, T. Gergely, A. Beszédes, T. Gyimóthy, and G. Tóth. Verifying the concept of union slices on Java programs. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 233–242, Mar. 2007.

[15] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[16] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 169–180, New York, NY, USA, 2006. ACM.

[17] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.

[18] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42, New York, NY, USA, 2005. ACM.