Some Notes on Interprocedural Program Slicing

K. B. Gallagher*
Computer Science Department
University of Durham
South Road
Durham DH1 3LE, UK
k.b.gallagher@durham.ac.uk

Abstract

Weiser's algorithm for computing interprocedural slices has a serious drawback: it generates spurious criteria which are not feasible in the control flow of the program. When these extraneous criteria are used the slice becomes imprecise in that it has statements that are not relevant to the computation. Horwitz, Reps and Binkley solved this problem by devising the System Dependence Graph with an associated algorithm that produced more precise interprocedural slices. We take a "step backward" and show how to generate exactly the interprocedural slicing criteria needed, using the program's call graph or a stack. This technique can also be used on a family of program dependence graphs that represent all procedures in a program and are not interconnected by a system dependence graph. Then we show how to use the Horwitz, Reps and Binkley interprocedural slicing algorithm to generate criteria and show that the criteria so generated are equal to those generated by the call-graph/stack technique. Thus we present alternative, equivalent ways to generate precise slicing criteria across procedure boundaries. And finally we show that under certain circumstances, Weiser's technique for slicing across procedures is a bit "too strong," for it always generates sufficient criteria to obtain the entire program as a slice on any criteria.

1 Introduction

Program slicing is a pretty simple idea. It's fundamental concepts are easily grasped by novices. It's a good idea because it applies to many problem domains; who doesn't want help in restricting a complicated problem to the relevant focus of inquiry?

We are so familiar with the concepts and arcana of program slicing that we are liable to forget that there are some deep and sophisticated concepts "under the hood;" when it comes to actually

*On leave from Loyola College in Maryland constructing algorithms, data structures and especially *writing* a program slicer, significant difficulties arise. (Note: how many production quality slicers do you know of? And as soon as you know of one you ask: "Does it work on language / dialect X?" The answer to the second question is usually negative.) The problems are myriad: getting the language to an invertible representation, so that the statements not in the slice are elided; implementing slicing algorithms on the representation; consideration of parameter passing, object creation and aliasing.... Then we move on to improved algorithms, better data structures; it seems every technical conference has and "Improved Slicing" paper. Indeed, the Source Code and Manipulation Conference (SCAM) is one of the venues in which one can discuss these sophisticated ideas without the necessity of a fully functioning slicing system and an application on which it can be applied.

This paper is the result of a series lectures that the author gave at the University of Durham, entitled "Program Slicing." The audience was third (final) year students in Computer Science and Software Engineering. The initial lectures were as above: This is a slice (Everybody understands.) Here is how to compute one. The fog of confusion rolls in. (Followed by the question: "Are going to make us *write* one?")

The following lecture approach was used. Use the simple data-flow control-flow approach outlined by Weiser, so that the computational *concepts* were understood; followed by the Program Dependence / System Dependence Graph (PDG/SDG) approach to improve the algorithms and address weaknesses of the data-flow control-flow method. The results from the early method could be used to check solutions and compare for accuracy with the SDG approach.

The particular problem was interprocedural program slicing. The ideas for this paper came while preparing and giving the lectures and as such do not offer any improved technique for computing slices. It was in trying to explain these concepts that some insights and simplifications came that connected some heretofore unnoticed concepts. These connections helped the students understand the central point of the lectures: how to *compute* an interprocedural program slice.

First, we show how to generate exactly the interprocedural slicing criteria needed, using the call graph. Then we show how to use the Horwitz, Reps and Binkley[2] interprocedural slicing algorithm (hereafter referred to as the "HRB algorithm" or just "HRB") can also be used to generate criteria and show that the criteria so generated are equal to those generated by the call-graph technique. Thus we present an alternative, equivalent way to generate precise slicing criteria across procedure boundaries. And finally we show that under certain circumstances, Weiser's technique for slicing across procedures is a bit "too strong," for it always generates sufficient criteria to obtain the entire program as a slice on any criteria.

2 Background

Wieser's paper on program slicing [4] has a method for going through procedure calls.

> For each criterion C for a procedure P, there is a set of criteria $\mathrm{UP}_0(C)$ which are those needed to slice callers of P and a set $\mathrm{DOWN}_0(C)$ which are those needed to slice procedures called by P ... $\mathrm{UP}_0(C)$ and $\mathrm{DOWN}_0(C)$ can be extended by functions UP and DOWN which map sets of criteria into sets of criteria. Let CC be any set of

```
main {
  sum := 0;
  i := 1;
  while (i < 11) do
      call A( sum, i);
  od;
end

A (x, y) {
   call Add(x,y);
   call Increment(y);
   return;
  }

Add (a, b) {
   a = a + b;
   return;
}

Increment ( z ) {
  Call Add ( z, 1);
  return;
}
```

Figure 1: Interprocedural program from [2] used for illustration.

criteria. Then

$$\mathrm{UP}(CC) = \bigcup_{C \in CC} \mathrm{UP}_0(C)$$

$$\mathrm{DOWN}(CC) = \bigcup_{C \in CC} \mathrm{DOWN}_0(C)$$

The union and transitive closure are defined in the usual way for relations. $(\mathrm{UP} \cup \mathrm{DOWN})^*$ will map any set of criteria into all those necessary to to complete the corresponding slices through all calling and called routines. The complete interprocedural slice for criterion C is then just the union of the interprocedural slices for each criterion in $(\mathrm{UP} \cup \mathrm{DOWN})^*(C)$.

The noted problem with Weiser's method is the generation of too many criteria which are extraneous to the program's feasible call graph. Criteria are generated which has a procedure returning from one other than the one from which it was called. An inspection of the call graph

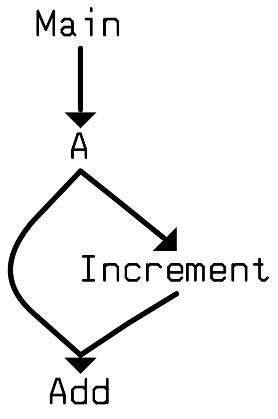


Figure 2: Call graph of the program of Figure 1.

of Figure 2 shows what the problem is: When slicing from procedure `Increment` a criterion is generated that goes DOWN into `Add`; from there a criterion is generated that goes UP into `A`. It is this criterion that does not respect the feasibility of return from `Increment` through `A`.

Horwitz, et al., [2] present a thorough discussion of this problem:

> Using Weiser's algorithm to slice this program with respect to the the variable *z* and the **return** statement of procedure *Increment*, we obtain everything from the original program. ... The reason these components are included in the slice computed by Weiser's algorithm is as follows: The initial criterion "<end of procedure *Increment*, z >" is mapped by the DOWN relation to a slicing criteria "<end of procedure *Add*, a >." The latter criterion is then mapped by the UP relation to *two* slicing criteria — corresponding to *all* the sites that call *Add* — the criterion "<call on *Add* in *Increment*, z >" and the (irrelevant) criterion "<call on *Add* in *A*, x >." Wieser's algorithm does not produce as precise a slice as possible because transitive closure fails to account for the calling context (*Increment*) of a called procedure (*Add*) and thus generates a spurious criterion (<call on *Add* in *Increment*, z >).
>
> ... The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure.

Recall that the HRB language is a simple but sufficient reduction of production languages, and that in their model, parameters are passed by value result.

We will use a <`ProcedureName.VariableName, point`> form to refer to criteria. For example, the criteria used in the HRB example will now be noted <`Increment.z, return`>. When a criterion is placed in a code sample the point will be omitted: <`Increment.z`>. Only the variable of interest will be noted as the point of interest is precisely where the criterion is inserted.

3 Generating Accurate Interprocedural Criteria

The problem of extraneous criteria arose from the generation of a criterion from a `return` back through an uncalled procedure; we want to eliminate this criterion. That is, we only want criteria that are on *feasible* call paths. The feasible call sequences can be obtained from an inspection of a slightly modified call graph, described next.

The call graph of Figure 3 has the same information as that of Figure 2, albeit in a different form. In Figure 3 each call to `Add` is noted explicitly. Using this graph it is relatively straightforward to get the precise criteria. Starting with the criteria <`Increment.z, return`>, we go DOWN into procedure `Add` and get criteria <`Add.a, return`>. Once we have gone backward across `Add` we go back UP into `Increment` with criteria <`Increment.z, Call Add`>. We have concluded slicing across `Increment` and now go UP into `A` with criteria <`A.y, Call Increment`>. Proceeding backward, we go DOWN into `Add` again with criteria <`Add.b, return`> (collecting no statements as variable `Add.b` is not defined) and come back UP with criteria <`A.y, Call Add`>, and so on. Figure 4 shows the program of Figure 1 with the criteria inserted as they were produced in this discussion.

What we have done here is to "slow down" the generation of criteria and insert them only as needed from the context. Weiser's UP/DOWN relations, in an attempt to generate complete criteria early in the slicing process, generate the infeasible criteria. The above method is akin to the Weiser's original approach of adding referenced variables to the original slicing criterion as they were needed. For instance, when a statement was determined to be in the slice, all variables referenced in all controlling predicates were added to the current slicing criteria. Of course, this only works in the iterative data-flow control-flow technique that Weiser used. This method is not suited to the PDG/SDG of HRB in which slices can be taken only where a variable is defined or referenced, unless the structure is augmented. It does, however, help the students grasp what is going on as an inter-procedural slice is computed. And it helps the student grasp some of subtlety of the HRB technique.

This procedure can also be done without explicit reference to the modified call graph. A stack with a can be used to keep track of the calling context. The intuitive mapping of the DOWN operations to push and the UP operations to pop fails. Consider the example of Figure 4 in which there is one DOWN followed by two UPs. A simple generalization of UP and DOWN to ENTER and LEAVE, regardless of the UP/DOWN direction suffices nicely. We ENTER a procedure, without regard to direction (UP or DOWN), slice through the procedure and LEAVE it, without regard to the direction. If, while slicing through a procedure, we ENTER another procedure, we will get a corresponding LEAVE. Now we map ENTER and LEAVE to push and pop and the problem of popping an empty stack goes away. This is similar to the *realizable path* (balanced parentheses) approach used in [3].

This still leaves the problem of recursive calls, which fortunately can be easily addressed by using a set to guard the insertion of criteria. More precisely, regard each criterion as an element of a set. Then, before each ENTER (push) operation, we determine if the new candidate criterion is in the set of criteria. If it is not, we continue as above; if it is in the set, do nothing.

By using a call graph or a stack, we have accounted for the calling context and give an accessible

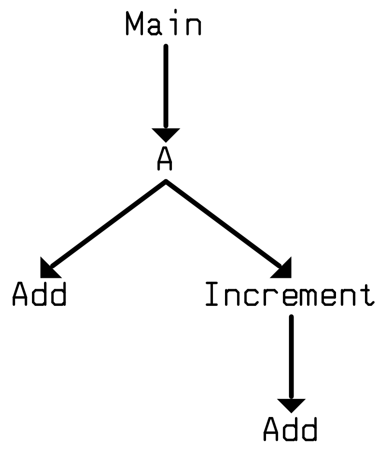


Figure 3: Call graph of the program of Figure 1 with each procedure call/return made explicit.

solution to the spurious criterion problem. To reiterate, the point of this section is show students how slicing criteria can be accurately generated in a data-flow control-flow environment. Once they understand this, it is easier for them to comprehend the SDG approach.

4 Using SGD's to Generate Slicing Criteria

This section shows a few ways to use SDG's to generate the same *criteria* that were derived in the previous section. In the first place, this can be done by noting that the SDG has a call graph embedded in it. Since we know how to use the call graph to generate criteria, we can use this embedded call graph to generate criteria. This is not a typical use if the SDG!

4.1 Linkage Grammars

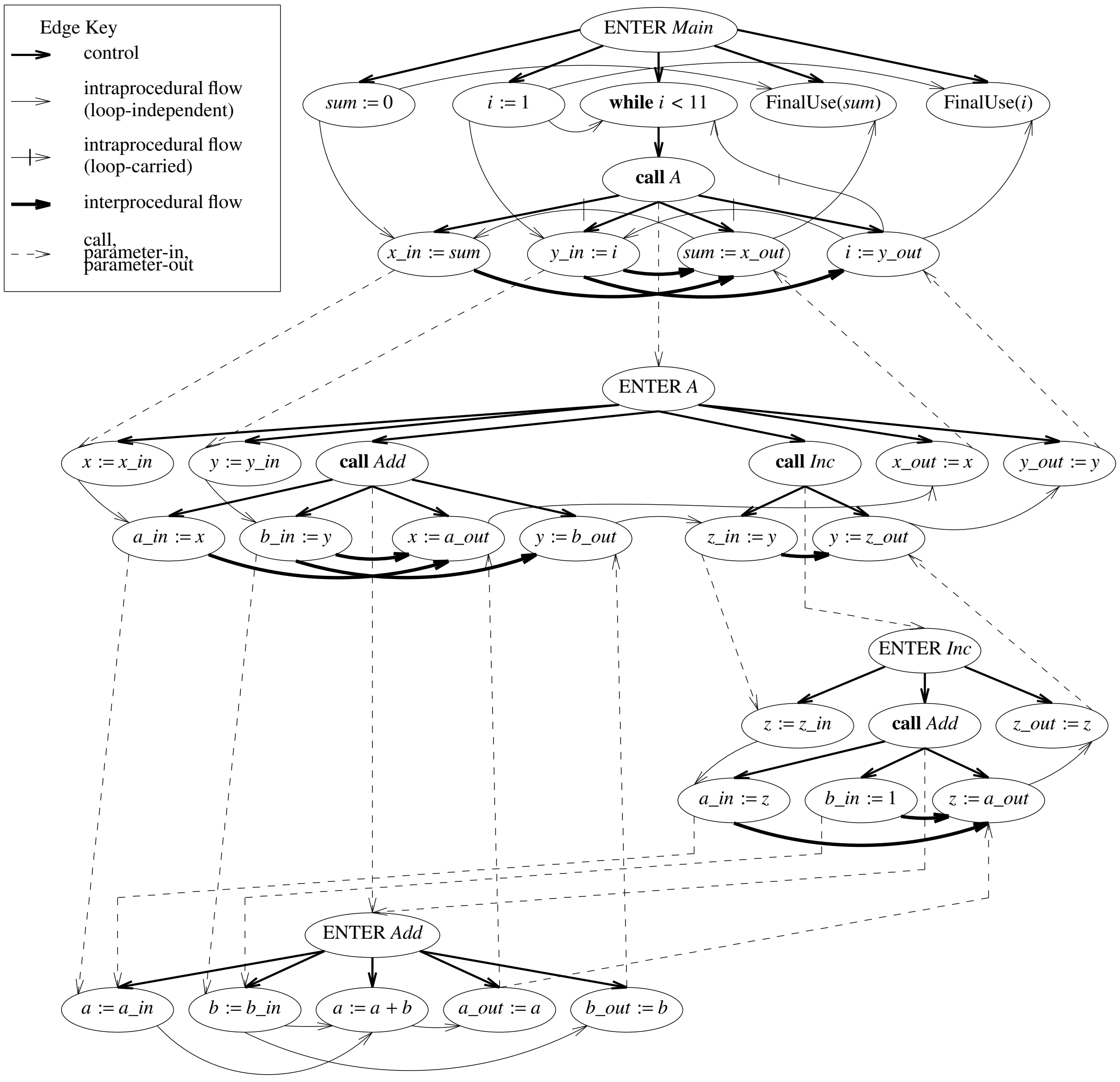
A linkage grammar is used in HRB to compute the transitive dependence due to procedure calls. The linkage grammar is an attribute grammar. The attributes of the linkage grammar are used the obtain the in/out variable flow across procedure boundaries; its context free part is used to determine the calling structure (the call graph). This context free part to generates a call tree. This is also similar to the *realizable path* approach used in [3]. Once we have the call tree in place, we proceed as previously outlined.

The HRB algorithm may also be used to emit criteria. The HRB algorithm proceeds in two phases. The first phase delivers the slice that is above the procedure of interest; that is it "identifies veritices that can reach [variable] s [in procedure P], and are either in P itself or a procedure that calls P..."[2]. The second phase does the other half of the work: it follows (transitively) the flows that reach s when called by P.

This gives an alternative (explanatory) names to some of the edges added to PDG's to create an SDG. The *parameter-out* edges are the *not-followed-in-phase-1* edges. (Why this is so makes a good exam question.) Similarly, the *parameter-in* and *call* edges are *not-followed-in-phase-2* edges. (Another good exam question!)

```
main {
  sum := 0;
  i := 1;
  while (i < 11) do
      call A( sum, i);
  od;
end

A (x, y) {
   /* 6. < A.y >
         UP from Add        */
   call Add(x,y);
   /* 4. < A.y >
         UP from Increment   */
   call Increment(y);
   return;
  }

Add (a, b) {
   a = a + b;
   return;
   /* 2.  < Add.a >
           DOWN from INCREMENT
           using the call graph  */
   /* 5.  < Add.b >
          this time DOWN from Add    */
}

Increment ( z ) {

  /* 3. < Increment.z >
         coming UP from Add      */
  Call Add ( z, 1);
  return;
  /* 1.  <Increment.z>  */
}
```

Figure 4: Program from [2] and Figure 1 with criteria inserted as `C` comments and showing the order in which they were inserted.



Example system's system dependence graph. Control dependences, shown unlabeled, are represented using medium-bold arrows; intraprocedural flow dependences are represented using arcs; transitive interprocedural flow dependences (corresponding to subordinate characteristic graph edges) are represented using heavy bold arcs; call edges, parameter-in edges, and parameter-out edges (which connect program and procedure dependence graphs together) are represented using dashed arrows.

Figure 5: System Dependence Graph of Figure 1.

4.2 A Combined Method

So, in the middle of the System Dependence Graph lecture (as the students' eyes are clouding over, trying to keep all the new edge and vertex types in their heads) I ask "Can we do this with out the System Dependence Graph?". It turns out that the answer is "yes."

Instead of building the SDG, we build only a named family of PDG's, joined by a call-graph. Starting at the relevant point in the procedure of interest, we chase the edges in each procedure's PDG. Crossing over a `Call` we match up the parameters and generate a new criteria for the called procedure, F. Slice across that and so on. Then when slicing across F is concluded, map the parameters from F back to the caller. When an ENTER node is traversed in the PDG, generate new criteria for each procedure that calls, and continue.

This is the same approach as outlined in section 3. This is a one-pass linear time method. It is one pass because we are using the as-needed technique for criteria generation and one-pass because we are using the PDG.

4.3 Are They The Same Criteria?

The argument the emitted criteria are the same comes from the observation that they are generated from the (perhaps embedded) call-graph and not from the SDG/PDG or the control-flow graph. As we pass up or down marking the calling sequence, the form from which the criteria are generated is moot.

Note that this uses SDG's to generate the criteria, not the slice.

5 Concurrent Assignments

There is one more observation to be made in relation to slicing across procedure boundaries. Reps, et al. [3] note:

> A procedure call is treated like a multiple [concurrent] assignment statement "$v_1, v_2, \ldots, v_n := x_1, x_2, \ldots, x_{m[sic]}$", where the v_i are the set of variables that might be modified by the call, and the x_j are the set of variables that might be used by the call. Thus, the value of every v_i is assumed to depend on the value of of every x_j before the call. This may lead to an overly conservative slice (*i.e.*, one that includes extra components...)

In Weiser's slicing method, we then have that the DEF'ed set of the above statement is $\{v_1, v_2, \ldots, v_n\}$ while the REF'ed set is $\{x_1, x_2, \ldots, x_m\}$. This would mean that when any of the x_i are REF'ed, *all* of the v_i are DEF'ed. A program slice in a language that uses value-result parameter passing, as in HRB, must proceed carefully when slicing across concurrent assignment statements. In this context, if a concurrent assignment is used to map actuals to formals, then when *any* actual parameter is passed, all formal parameters are captured. This is clearly too strong and would lead to the entire procedure being included in the slice, along with all calling and called procedures: the whole program! (If a procedure were not in such a slice, it would be dead code.)

This situation can be easily remedied by serializing the concurrent assignment as described in [1], chapter 3.

6 Conclusion

Interprocedural slicing is an interesting problem. Getting the criteria correct across calls and returns requires careful attention to detail. Attending to the feasible calling sequences adds another layer of particulars. The System Dependence Graph and its associated algorithms solve all of these problem all at once. This is why is it is a superior representation.

Trying to *learn* about interprocedural program slicing is another matter. It helps the novice to have the both the issues and their respective solutions broken into more accessible concepts. Once these individual problems and solutions are discussed, the true power and beauty of the SDG representation can be fully appreciated. This paper has given some direction on how to do that.

References

[1] J. Gannon, J. Purtilo, and M. Zelkowitz. *Software Specification: A Comparisopn of Formal Methods*. Ablex, 1994.

[2] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):35–46, January 1990.

[3] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 11–20, 1994. Published in ACM SIGSOFT Notes v19 n4.

[4] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.