COMPUTING AMORPHOUS PROGRAM SLICES USING DEPENDENCE GRAPHS

David Binkley

Loyola College in Maryland
Computer Science Department
4501 North Charles Street
Baltimore, MD 21210
binkley@cs.loyola.edu

Keywords
Slicing, Dependence Graphs, Debugging, Program Transformation, Testing.

ABSTRACT

This paper presents an algorithm for computing amorphous program slices. Amorphous program slicing relaxes the syntactic condition of traditional slicing and can therefore produce considerably smaller slices. Existing algorithms slice the program's control-flow graph and then apply a collection of axiomatic rules. A more unified approach is obtained using the *system dependence graph* for both steps. Slicing this graph requires linear time. Furthermore, it contains sufficient information to allow the rules to be easily specified and applied. Some of these rules treat the system dependence graph as a data-flow graph and use a data-flow evaluation model, while others perform standard compiler optimizations. The application of amorphous slicing to the problem of determining array access safety is considered.

1. INTRODUCTION

Program slicing is a decomposition technique that extracts from a program statements relevant to a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the computation of variable v at statement s?" An observer viewing the execution of a program and its slice cannot determine which is which when attention is focused on statement s and variable v.

supported by National Science Foundation grant CCR-9803665

The utility and power of program slicing comes from its ability to assist in tedious and error prone tasks such as program debugging, testing, parallelization, integration, software safety, software understanding, software maintenance, software metrics, and program comprehension [4, 14]. Slicing does this by extracting an algorithm—whose computation may be scattered throughout a program—from intervening irrelevant statements. It should be easier for a programmer interested in a subset of the program's behavior to understand the slice.

The above definition of a program slice places a semantic constraint on the slice; it requires the slice to have a certain behavior. In his original definition, Weiser also placed a syntactic constraint on a slice [15]: a slice is determined by deleting statements from a program. For debugging, the application Weiser was studying, this is a desirable property. However, for some applications, such as program comprehension, it can be overly restrictive.

This paper presents an efficient algorithm for computing *amorphous program slices*[8]. An amorphous program slice has the same semantic constraint as a traditional slice, but a more relaxed syntactic constraint. As illustrated in Figure 1, this allows certain transformations to reduce the slice of the slice. The original implementation of amorphous program slicing repeats the following two steps until the first step fails to delete any statements [7, 8]:

(1) slice the program,
(2) apply a set of semantics preserving axioms and rules.

This implementation is a "proof of concept." It uses the program's control-flow graph to compute program slices and provides no guidance for applying the axioms and rules. An efficient implementation requires an efficient slicing algorithm and an efficient strategy for applying the axioms and rules.

The algorithm developed in this paper slices and then transforms the program's *system dependence graph* [11]. Slicing the system dependence graph is a linear time operation [11]. The transformations then applied include some that

Initial Program	Traditional Slice	Amorphous Slice
main()	*main*()	*main*()
{	{	{
$a = 1$	$a = 1$	
$b = a + 2$	$b = a + 2$	$b = 3$
$c = 5$		
}	}	}

Figure 1. A traditional and an amorphous slice of a program. The slice is taken with respect to b at the end of the program. Traditional slicing deletes the statement "$c = 5$." Amorphous slicing further reduces the program's size using constant propagation and folding.

treat the system dependence graph as a data-flow graph in which vertices of the graph are "fired" (executed) under certain conditions. Other transformations define, as graph rewritings, standard compiler optimizations such as partial redundancy elimination.

The remainder of this paper is organized as follows: Section 2 presents some background information. Section 3 presents an algorithm for computing amorphous program slices. The application of amorphous program slicing to the problem of array accesses safety is presented in Section 4, followed by a summary in Section 5.

2. BACKGROUND

The amorphous program slicing algorithm presented in Section 3 is based on a program's system dependence graph (SDG) [11]. An SDG is a collection of vertices and edges (the SDG for the last while loop in Figure 3 is shown in the right of Figure 4). The vertices represent the components of the program (*e.g.*, assignments statements and predicates). The edges of an SDG represent dependences between components. There are two kinds of edges: control dependence edges and data dependence edges. A control dependence edge, which is labeled either **true** or **false**, connects a vertex v to a vertex u if during execution, whenever the predicate represented by v is evaluated and its value matches the label on the edge to u, then the program component represented by u will eventually be executed provided the program terminates normally.

The only data-dependence edges used in this paper are *flow dependence edges*. A flow dependence edge runs from a vertex that represents an assignment of a variable to a vertex that represents a use of the variable reached by that assignment. Flow dependence edges are categorized as either *loop-independent* or *loop-dependent*: loop-dependent edges represent values passed from one loop iteration to another; all other flow dependence edges are loop-independent.

3. ALGORITHM

This section presents an algorithm for computing amorphous slices. The algorithm has the four steps: build the program's SDG, slice this SDG, transform the slice using a set of rewriting rules, and then reconstitute a program from the transformed graph. The complete algorithm is shown in Figure 2. To compute the amorphous program slice of P with respect to vertex v where v uses or defines variable x, the interprocedural slice of P's SDG with respect to v is first taken [11, 12]. This slice removes computations from P that do not affect the value of x at v.

Two sets of transformations are then applied to the slice. The first set treats the SDG as a data-flow graph [3, 13]. Each rule in this set chooses a vertex that has no incoming control or loop-independent flow dependence edges and then *fires* it: the vertex is removed from the graph along with its outgoing

```
function AmorphousSlice(P, S)
declare
   SlicedP, P: programs
   S: a statement
   G: an SDG
   WorkList, Fireable: sets of vertices
   v: a vertex
begin
   G := BuildSDG(P)
   G := Slice(G, VertexFor(S))
   WorkList := { Vertices that a Table 2
                 rule applies to }
   Fireable := { Enter_Main }

   while WorkList ≠ ∅ and Fireable ≠ ∅ do
      while Fireable ≠ ∅ do
         Select and remove a vertex v from Fireable
         Fire v using its Table 1 rule,
            updating G, Fireable, and WorkList
      od
      while WorkList ≠ ∅ do
         Select and remove a vertex v from WorkList
         Apply a Table 2 rule, to v
            updating G, Fireable, and WorkList
      od
   od
   G := MakeFeasible(G)
   SlicedP := Reconstitute(G)
   return(SlicedP)
end
```

Figure 2. Function AmorphousSlice computes an amorphous slice of P with respect to the vertex for a statement S.

edges. Firing may also add, remove, or update other vertices and edges in the graph. For example, to fire an if-predicate, the value of the predicate is first computed. Assuming that the predicate evaluates to **true**, all the vertices reachable along control dependence edges labeled **false** are removed together with any vertices transitively control dependent on them. (This removal assumes that the control dependence subgraph forms a tree. This assumption is violated by unstructured programs in which case a vertex is removed only if it has no other remaining incoming control dependence edges.) For the five kinds of vertices in the SDG, Table 1 informally describes their rewriting (these rules can be formalized using the models of Selke [13] and Binkley [1]).

To implement these rules, a simple work list of vertices with no incoming loop-independent edges is maintained. This list initially contains the entry vertex of procedure *main*. As edges are removed, their targets are checked; a target with no remaining loop-independent incoming edges is added to the work list.

The second set of rewriting rules essentially performs standard compiler optimizations (*e.g.*, redundant expression elimination, constant propagation [3], and loop-unrolling [6]) on the SDG. Table 2 contains several such rules; section 4 will add rules specialized to solving a particular problem. As the SDG is built, opportunities for applying these rules are recorded. The algorithm then considers a vertex every time its incident edges or label change. As with the firing of the rules from Table 1, a simple work list provides the necessary control structure.

The fourth and final step in the algorithm produces a program from the rewritten SDG. To do so requires that the SDG be *feasible* (a feasible SDG is the SDG of some program). This requires replacing any tagged constants with a variable and introducing an assignment vertex which represents the assignment of the constant to the variable. These vertices are control dependent on the vertex that the use is control dependent on (it is possible for two uses to share an assignment vertex). After this step an entry vertex is added back to the graph with control dependence edges to all vertices without an incoming control dependence edge. The graph can now be made feasible and reconstituted into a program [2, 5, 10].

4. AN EXAMPLE APPLICATION

This section considers the application of amorphous slicing to the problem of array bounds correctness. Traditional program slicing is not directly applicable to this problem because it computes a slice based on a variable, not a concept such as "array access safety." Using a precursor to amorphous slicing, Harman and Danicic overcame this limitation by first augmenting a program and then taking a slice. The augmentation includes assignments to a new variable *robust* such that *robust* has value **true** iff the program has no out of bounds array references [7]. More specifically, it adds at the beginning of the program, the assignment "*robust* = **true**" (at the start of execution there have been no array bounds errors). At the end of the program the statement "*print robust*" is added. Finally, before each array reference $a[i]$ the assignment statement

$$\begin{aligned} &robust = robust \;\&\& \\ &\qquad i \geq 0 \;\&\&\; i < upper\_bound(a) \end{aligned}$$

is added immediately before the statement with the reference (this assumes that arrays start at index 0). The transformed program computes the robustness of the original program. That is, for a given input, this program computes whether or not the original program will fail due to an array out of bounds error.

The remainder of this section describes the computation of a *robustness slice* (the amorphous slice of the augmented program) using the rules given in Section 3. For ease of comparison, the example used is essentially that used by Harman and Danicic [7]. Figure 3 shows this program after it has been augmented with assignments to *robust*. In this example, the slice taken with respect to *robust* at the end of the program includes most of the original program.

Now consider the graph rewriting of this slice. The rewriting begins with the firing of the entry vertex for main. Next the assignments to *robust* and i are fired updating nodes in the body of the while loop. The firing of the while loop essentially unrolls the loop. This unrolling continues until a single vertex labeled "*robust* = 1" remains.

The bulk of the remaining rewriting is concentrated on the two other while-loops in the program. The unrolling of the last loop (shown in Figure 3) is similar to that of the first loop except the control dependence of the use of *robust* on the statement "if $(result[i] > bc)$" causes each unrolling to leave a copy of the entire if subgraph. This can be avoided by introducing the new rewriting rule illustrated in Figure 4. This rule is the graph adaptation of a rule given by Harman and Danicic.[1] It applies to an if vertex v and the vertices reachable from v by control dependence edges with the same label (**true** or **false**) and their transitive control dependence successors, when these vertices can be divided into two groups. The first group is an assignment to *robust* that is reached by a flow dependence edge from an identical assignment to *robust* that is not control dependent on v (the assignments must be of the form "*robust* = *robust* && $\cdots$"). The second group, S, has no

[1] In some circumstances, the rule given by Harman and Danicic is incorrect: it removes a branch of an if statement if the branch does not directly affect *robust*. However, it is possible for the branch to indirectly affect *robust*.

Table 1	
Vertex Kind	Rule
Enter	No additional actions.
Assignment	For an assignment vertex labeled "$x = exp$," modify the targets of all outgoing edges as follows: replace x with c *tagged* by x, written c^x, where c is the constant value of *exp*. The value of *exp* is constant because the assignment vertex has no incoming flow dependence edges (the lack of incoming control edges implies that the vertex has no incoming loop-dependent edges). Tags allow subsequent rewritings to overwrite previous ones [1, 13].
If-predicate	Assume the predicate's value is **true** (the **false** case is symmetric). Remove all vertices representing statements in the **false** branch of the if statement along with their incoming and outgoing edges.
While predicate	If the predicate evaluates to **false** then all vertices representing the loop are removed along with their incoming and outgoing edges. Otherwise, if the predicate evaluates to **true** then copies of the vertices that represent the body of the loop are added to the graph. Copies are added as if one iteration of the loop had been unrolled and placed immediately before the original while-loop.
Call-site	Perform procedure inlining: replace the call-site and actual parameter vertices with copies of the vertices for the called procedure.

Notes

- Vertices that represent statements with side effects (*e.g.*, printing) are never fired.
- The while-loop and call-site rules may actually increase the size of the graph. In some cases this is only temporary as further rewritings will remove the added vertices. In cases where the added vertices are not removed, the use of these rules must be restricted.

Table 2	
Rule	Explanation
constant folding and propagation	For an assignment vertex labeled "$x = e$", if e is constant (perhaps the result of previous rewritings), then it is evaluated and the resulting value for x is placed "on" all outgoing flow dependence edges. If all the incoming flow dependence edges for a variable x are labeled with the same constant value, then x is replaced with that constant value.
flow edge removal	For a flow dependence edge from an assignment vertex v labeled "$a = e$" to a vertex u, the expression e can be propagated along the edge to u. For this rewriting to be allowed, u must be the target of exactly one flow-dependence edge for variable a. The SDG is updated by copying the incoming flow dependence edges of v to u and rewriting the label of u by substituting e for a. If u is an assignment vertex that assigns to a, then its rewriting requires rewriting all flow-edges from v in order to avoid situations in which the graph cannot be transformed back into a sequential program.
loop (anti-) fusion	Loop fusion joins two similar loops in the hope of reducing loop overhead and of creating a larger basic block within the loop body upon which to perform optimization [6]. Loop anti-fusion separates a loop with two separate computations in its body (perhaps to allow parallel execution). While the exact conditions under which this optimization applies are too complex to state here, two (sets of) statements that have no dataflow connections are trivial candidates.

Notes

- Vertices with no outgoing edges can be removed form the graph. (Note that a subsequent slice will not include them.)

outgoing edges to vertices outside of S (note that S includes v). When this pattern is found, then the assignment to *robust* that is control dependent on v and the vertices of S except v are removed. If this removal leaves no vertices control dependent on v, then v too is removed.

Applying the if-rule to the last while loop of the example program, produces the graph on the bottom of Figure 4. Note that removal of the two vertices that involve *result* removes the only two uses of *result* in the graph. Thus, the vertices that define *result* can also be removed. This can be done by a recursive vertex removal procedure or by a subsequent slice with respect to the final value of *robust*.

A sequence of rewritings similar to the unrolling of the first loop reduces the subgraph on the bottom of Figure 4 to a single assignment vertex labeled "*robust* = *robust* && $127^i \geq 0$ && $127^i \leq 127$," which simplifies to "*robust* = *robust*" (a subsequent rewriting rule removes such self assignment vertices).

```
robust = 1; /* starts out robust */
printf("Enter a string");
scanf("%s",source);
i = 0;
while (i<128)
{
   robust = robust && i>=0 && i<=127;
   result[i] = 0;
   i=i+1;
}
i = 0;
robust = robust && i>=0 && i<=MAX;

while (source[i] != '\0')
{
   robust = robust && i>=0 && i<=MAX;
   x = source[i];
   i = i+1;
   robust = robust && x>=0 && x<=127;
   robust = robust && x>=0 && x<=127;
   result[x] = result[x]+1;
}

b = 0;
bc = 0;
i = 'a';

while (i<128)
{
   robust = robust && i>=0 && i<=127;
   if (result[i] > bc)
   {
      b = i;
      robust = robust && i>=0 && i<=127;
      bc = result[i];
   }
   i = i+1;
}
printf("Most frequent: %c ",b);
printf("occurs %d times\n",bc);
printf("robust = %d",robust);
```

Figure 3. An example program that computes the most frequently occurring ASCII character in its input. The slice of this program with respect to printing of the final value of *robust* excludes only the boxed statements. (The constant MAX is the declared size of the string *source*.)

Finally, Figure 5 shows the rewriting of the middle loop. The initial loop, shown in Figure 5(a), has three flow edges that are candidates for flow-edge removal. Figure 5(b) shows the result of removing the edge labeled $flow^1$. After rewriting, the flow dependence edge from the first assignment of *robust* to the second has been copied to the final assignment; the copy is labeled $flow^2$ in Figure 5(b). This rewriting also updates the expression labeling the target of edge $flow^1$ to be "*robust* = *robust* && $x \geq 0$ && $x \leq 127$ && $x \geq 0$ && $x \leq 127$" which simplifies to "*robust* = *robust* && $x \geq 0$ && $x \leq 127$" as shown in Figure 5(b). Figure 5(c) shows the result of removing the edges labeled $flow^2$ and $flow^3$. In all three removals, the source vertex has no other outgoing edges and thus is removed from the graph as part of the rewriting.

The remaining while-loop cannot be rewritten further because the value of *source* is not known until runtime. The final code is

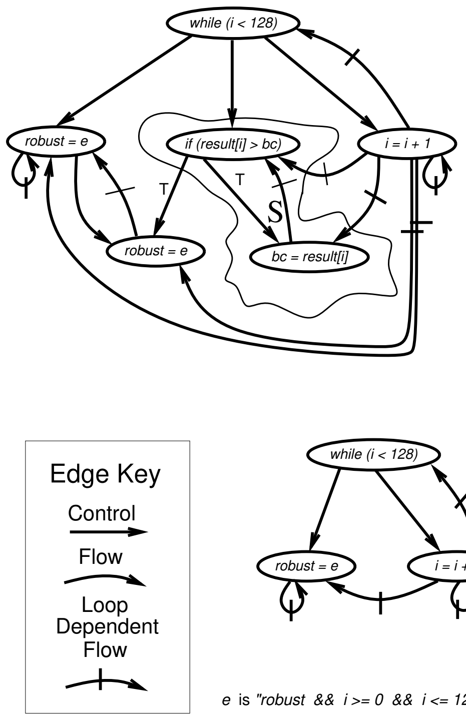

Figure 4. An illustration of the if rule to the last while loop in the example program.

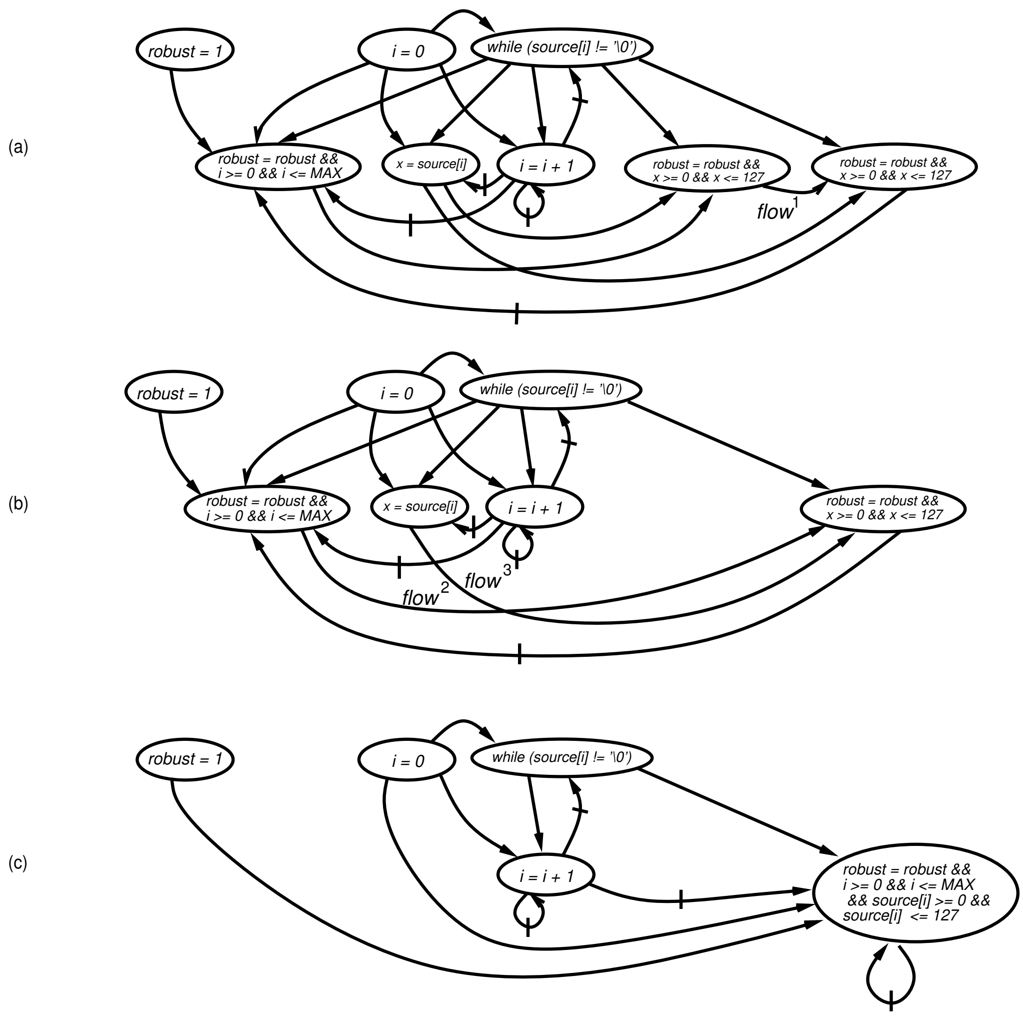

Figure 5. The middle while loop of the example program. The vertex labeled "$robust = 1$" is not from the original program. It is the result of rewriting the first loop. The vertex labeled "$result[x] = result[x] + 1$" has been removed from the loop by loop-anti-fusion.

```
scanf("%s", source);
robust = 1;
i = 0;
while(source[i] != '\0')
{
   robust = robust && i>=0 && i<=MAX
                && source[i]>=0
                && source[i]<=127;
   i = i+1;
}
printf("robust = %d", robust);
```

This program outputs **true** iff in the loop the following condition is never violated:

```
i>=0 && i<=MAX &&
source[i]>=0 && source[i]<=127
```

In other words, (since i is never less than 0) the input string is not too long (*i.e.*, i is never greater than MAX) and no input character (held in *source*[]) is outside the range $0 \cdots 127$.

5. SUMMARY

Several different kinds of slices have been proposed to address different problem domains. Amorphous program slicing is useful in reducing the size of a slice when the resulting slice need not be composed solely of statements from the original program. This paper presents an algorithm for computing amorphous program slices that uses the SDG as an underlying representation. The SDG is an ideal representation for computing amorphous slices because the data and control dependences contained in the graph make it is easy to compute the initial static slice and to specify the transformation rules.

The example application considered in Section 4 involves transforming a program to allow a slice to be computed on some less tangible aspect of the program. In addition to array safety, this idea has been used to analyze the safety of dynamic memory access [9]. It should also be possible to extract type safety from C++ programs using a similar approach. This idea is a topic for future research.

Acknowledgments
The comments of Mark Harman improved and clarified several parts of this paper.

REFERENCES

1. Binkley, D., Horwitz, S., and Reps, T., "The multi-procedure equivalence theorem," TR-890, Computer Sciences Department, University of Wisconsin, Madison, WI (November 1989).

2. Binkley, D., "Precise executable interprocedural slices," *ACM Letters on Programming Languages and Systems* **2**(1-4)(1993).

3. Binkley, D., "Interprocedural constant propagation using dependence graphs and a data-flow model," in *International Conference on Compiler Construction,* (Edinburgh, Scotland, April 1994), (1994).

4. Binkley, D. and Gallagher, K., "Program slicing," *Annals of Computing*, (1996).

5. Choi, J. and Ferrante, J., "Static slicing in the presence of goto statements," *ACM Transacton on Programming Languages and Systems* **16**(4) pp. 1097-1113 (July 1991).

6. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).

7. Harman, M. and Danicic, S., "Using program slicing to simplify testing," *Journal of Software Testing, Verification, and Reliability* **5** pp. 143-162 John Wiley & Sons Ltd, (1995).

8. Harman, M. and Danicic, S., "Amorphous program slicing," pp. 70-79 in *IEEE International Workshop on Program Comprehension,* (Dearborn, Michigan May, 1997), IEEE Computer Society, Washington, DC (1997).

9. Harman, M., Sivagurunathan, Y., and Danicic, S., "Analysis of dynamic memory access using amorphous slicing," in *to appear in IEEE International Conference on Software Maintenance,* (Washington, DC, Nov, 1998), IEEE Computer Society, Washington, DC (1998).

10. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

11. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).

12. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (New Orleans, LA, December 7-9, 1994)*, ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994), pp. 11-20.*, (1994).

13. Selke, R, "A graph semantics for program dependence graphs," pp. 12-24 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, January 11-13, 1989), ACM, New York, NY (1989).

14. Tip, F., "A survey of program slicing techniques," *Journal of Programming Languages* **3**(3) pp. 121-189 (september 1995).

15. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).