System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow

Saurabh Sinha
Computer and Info. Science
Ohio State University
2015 Neil Avenue
Columbus, OH 43210 USA
sinha@cis.ohio-state.edu

Mary Jean Harrold
Computer and Info. Science
Ohio State University
2015 Neil Avenue
Columbus, OH 43210 USA
harrold@cis.ohio-state.edu

Gregg Rothermel
Computer Science
Oregon State University
Dearborn Hall 307-A
Corvallis, OR 97331 USA
grother@cs.orst.edu

Abstract

Many algorithms for automating software engineering tasks require program slices. To be applicable to large software systems, these slices must be computed interprocedurally. Slicing techniques based on the system dependence graph (SDG) provide one approach for computing interprocedural slices, but these techniques are defined only for programs in which called procedures necessarily return to call sites. When applied to programs that contain arbitrary interprocedural control flow, existing SDG-based slicing techniques can compute incorrect slices; this limits their applicability. This paper presents an approach to constructing SDGs, and computing slices on SDGs, that accommodates programs with arbitrary interprocedural control flow. The main benefit of our approach is that it allows the use of the SDG-based slicing technique on a wide class of practical programs to which it did not previously apply.

Keywords: Program slicing, interprocedural slicing, system dependence graph

1 Introduction

Program slicing has many applications to software engineering, including program comprehension, debugging, testing, maintenance, and integration of program versions (e.g., [11, 19]). A *program slice* of a program P with respect to a *slicing criterion* $<p,V>$, for p a location in P and V a set of variables in P referenced at p, is the set of statements and predicates in P that might affect the value of variables in V at p [12].[1]

To be applicable to large software systems, slices must be computed across procedure boundaries (*interprocedurally*). Interprocedural slicing techniques based on the system dependence graph (SDG) guarantee that slices are computed only along paths that represent legal call/return sequences [12], removing one source of imprecision of an earlier interprocedural slicing algorithm [19]. However, these techniques apply only to programs in which called procedures necessarily return to the caller at the call site. When applied to programs that contain arbitrary interprocedural control flow, SDG-based slicing techniques compute slices that not only contain unnecessary statements, but also omit necessary statements. Such incorrectness may be significant for tasks, such as integration of program versions [11], that require conservative solutions.

Languages that allow arbitrary interprocedural control-flow are common. For example, Fortran provides a `stop` statement, Ada provides the `raise-when` construct, C, C++, and Java provide `exit()` calls, C++ and Java provide `try-throw-catch` constructs, and C and C++ provide `setjmp()-longjmp()` calls. These

[1] This definition differs slightly from Weiser's original definition of program slicing [19].

language constructs cause situations in which procedure calls do not return to the call site. Our preliminary studies suggest that such constructs can occur frequently in practice: in a study of over 70 C programs, 63% contained `exit()` calls; in a study of over 1650 Java programs, 31% contained `try` or `throw` statements. The frequency of occurrence of these constructs in widely utilized languages suggests that the use of slicing techniques that do not consider the constructs could have significant effects in practical contexts.

One way to address this problem is to use an alternative slicing technique, such as the algorithm of [8]. That algorithm uses data-flow analysis to compute interprocedural slices, correctly considering programs that contain halt statements. The algorithm can be extended to accommodate arbitrary interprocedural control flow.

Slicing techniques based on data-flow analysis or on the SDG, however, form two general classes of slicing techniques. SDG-based techniques precompute, at potentially large expense, data-flow and other information for an entire program, constructing a representation on which slices can be computed by a simple graph-reachability approach in linear (in the size of the SDG) time. Data-flow-analysis-based techniques, in contrast, require less extensive precomputation of information; instead, they perform more computation while slicing. The relative costs/benefits of the two classes of techniques vary, depending on the program being sliced and on the sequence of slicing requests issued. It follows that there are situations in which either approach is more efficient than the other. Thus, an SDG-based interprocedural slicing technique that correctly handles arbitrary interprocedural control flow would be useful.

This paper presents a new approach for constructing SDGs and computing slices on SDGs that accommodates arbitrary interprocedural control flow. Our algorithm constructs, for individual procedures, control-flow representations that incorporate information on arbitrary interprocedural control flow, and uses these representations to construct an extended SDG. Our algorithm uses this graph to compute interprocedural slices that are precise with respect to arbitrary interprocedural control flow.

We implemented our algorithm and performed an experiment comparing the slices obtained using the traditional SDG with the slices obtained using our extended SDG. Our results show that the two sets of slices can differ significantly.

In the next section, we review the system-dependence-graph based approach to interprocedural slicing. In Section 3, we discuss the problems created by arbitrary interprocedural control flow. In Section 4, we present our algorithm for constructing and performing slicing on system-dependence graphs in the presence of arbitrary interprocedural control flow. Section 5 presents analytical and empirical evaluations of our approach, Section 6 discusses related work, and Section 7 presents our conclusions.

2 SDG-Based Slicing

A *system-dependence graph* (SDG) [12] is a collection of *procedure-dependence graphs* (PDG) [5] — one for each procedure — in which vertices are statements or predicate expressions. *Flow-dependence* edges represent flow of data between statements or expressions; *control-dependence* edges represent control conditions on which the execution of a statement or expression depends. Each PDG contains an *entry* vertex that represents entry into the procedure. To model parameter passing,[2] an SDG associates formal parameter vertices with each procedure-entry vertex: a *formal-in* vertex for each formal parameter of the procedure; a *formal-out*

[2]Global variables are treated as parameters.

```
program Sum1
   procedure M1                         procedure B1
1  begin M1                          9  begin B1
2     read i                         10a   call C1
3     read j                         10b   return C1
4     sum = 0                        11    sum = sum + j
5     while i < 10 do                12    read j
6a       call B1                     13    i = i + 1
6b       return B1                   14    end B1
      endwhile
7     print sum
8  end M1                               procedure C1
                                     15 begin C1
alternative version of 17:           16    if j < 0 then
17'   halt                           17       print("error")
                                           endif
                                     18 end C1
```

Figure 1: Programs `Sum1` and `Sum2`; `Sum2` is formed by substituting $17'$ for 17.

vertex for each formal parameter that may be modified [14] by the procedure. An SDG associates each call site in a procedure with a *call* vertex and a set of actual parameter vertices: an *actual-in* vertex for each actual parameter at the call site; an *actual-out* vertex for each actual parameter that may be modified by the called procedure.

An SDG connects PDGs at call sites. A *call* edge connects a call vertex to the entry vertex of the called procedure's PDG. Parameter-in and parameter-out edges represent parameter passing: *parameter-in* edges connect actual-in and formal-in vertices, and *parameter-out* edges connect formal-out and actual-out vertices.

Figure 2 shows the SDG for program `Sum1` of Figure 1. In the figure, ellipses represent program statements (labeled by statement numbers), parameter vertices, entry points, and call sites; various types of edges (shown by the key) represent dependencies and bindings.

Horwitz, Reps, and Binkley [12] compute interprocedural slices by solving a graph-reachability problem on an SDG. To restrict the computation of interprocedural slicing to paths that correspond to legal call/return sequences, an SDG uses summary edges to explicitly represent the transitive flow of dependence across call sites caused by data dependencies, control dependencies, or both. A *summary edge* connects an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex may affect the value associated with the actual-out vertex.

The interprocedural slicing algorithm consists of two phases. The first phase traverses backwards from the vertex in the SDG that represents the slicing criterion along all edges except parameter-out edges, and marks those vertices that are reached. The second phase traverses backwards from all vertices marked during the first phase along all edges except call and parameter-in edges, and marks reached vertices. The slice is the union of the marked vertices.

For example, consider the computation of a slice for $< 6b, \{i\} >$; this slicing criterion is represented in the SDG of Figure 2 by the actual-out vertex for i at the call site to `B1`. In its first phase, the HRB algorithm adds the vertices shaded darkly in Figure 2 to the slice. In its second phase, the algorithm adds the vertices reachable (backwards) from those added in the first phase, shaded lightly in the figure.

In the rest of this paper, we refer to the SDG, the SDG-construction algorithm, and the slicing algorithm of

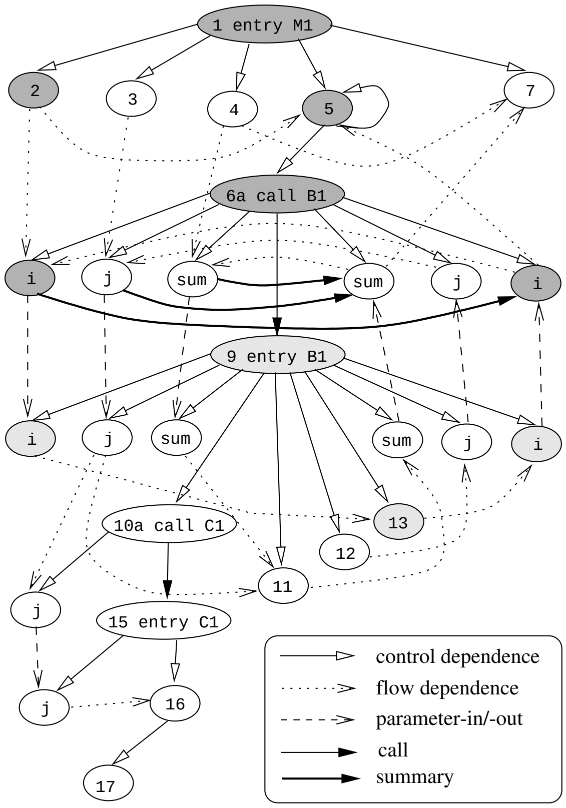

Figure 2: SDG for program `Sum1` of Figure 1.

Reference [12] as the HRB SDG, HRB SDG-construction algorithm, and HRB slicing algorithm, respectively.

3 The Problem

Consider the alternative version of `Sum1`, `Sum2`, shown in Figure 1, in which the print statement at line 17 has been changed to a halt statement (line 17′). In `Sum2`, control in `C1` may reach the halt statement instead of returning to the call site from which `C1` was invoked. Thus, the call to `C1` at line 10a in `B1` may not return, and thus, the call to `B1` at line 6a in `M1` may not return. We refer to the presence of control-flow paths through which called procedures do not return to call sites as *arbitrary interprocedural control flow.*[3]

Arbitrary interprocedural control flow can create control dependencies that differ from the dependencies computed by considering a program's procedures individually (e.g., by the algorithm of [5]). For illustration, consider the effects, on control dependencies, of the halt statement at line 17′ in `Sum2`. This halt statement causes statements that postdominate[4] the false branch of the `if-then` statement at 16 but do not postdominate the true branch of statement 16 to be control dependent on statement 16 being false. Thus, statements 10b, 11, 12, 13, and 14 in `B1`, and 5 and 6b in `M1`, are control dependent on statement 16. Furthermore, this halt statement causes statements that postdominate the false branch of the `while` statement at 5 but do not

[3]Like Ball and Horwitz [3], we use the term "arbitrary"; other terms, such as "unstructured" could also be used to describe this type of interprocedural control flow.

[4]Let $Pdom(N_j)$ be the set of N_i such that every path in the control flow graph from N_j to the exit of the program contain N_i. Then the vertices in $Pdom(N_j)$ *postdominate* N_j. The *immediate postdominator* of N_j is the unique N_k in $Pdom(N_j)$, such that N_k does not postdominate any other members of $Pdom(N_j)$.

```
program Sum3
   procedure M2
1  begin M2
2     read i
3     read j
4     sum = 0
5     try
6        while i < 10 do
7a          call B2
7b          return B2
         endwhile
8        print sum
      endtry
9     catch c: t1
10       print "error:",c
      endcatch
11 end M2

   procedure B2
12 begin B2
13a   call C2
13b   return C2
14    sum = sum + j
15    read j
16    i = i + 1
17    end B2

   procedure C2
18 begin C2
19    if j = 0 then
20       throw e  // type t1
21    elsif j < 0 then
22       halt
      endif
23 end C2
```

Figure 3: Program Sum3.

postdominate its true branch to be control dependent on statement 5 being false. Thus, statement 7 in M1 is control dependent on statement 5. None of these control dependencies are represented in the HRB SDG (Figure 2). A consequence of these dependencies is that the value of i on return from B1, which is dependent on statement 13, is also dependent (through the control dependence of statement 13 on statement 16) on statement 16, and thus, (through the data dependence of statement 16 on statement 3) on statement 3.

Through such effects on control dependence, arbitrary interprocedural control-flow can have significant effects on slice computation. To see this, consider applying the HRB slicing algorithm to program Sum2. In this case, the SDG for program Sum1 also serves as the SDG for program Sum2, except that vertex 17 now corresponds to the halt statement instead of the print statement. In this case, a slice computed for $< 6b, \{i\} >$ in the SDG for Sum2 yields the same slice as it did for Sum1. However, this slice omits statements 3 and 16 which, as previously stated, also influence the value of i at 6b. The slice also misses statement 12, and in fact, omits every statement in C1, even though some of these statements influence the computed value of i at 6b.

This example illustrates how the presence of a halt statement can affect control dependencies and slice computation. However, many other language constructs can create arbitrary interprocedural control flow. For example, consider a typical throw–catch construct. Figure 3 shows an extension of Sum2, Sum3, that contains a throw statement at line 20 that raises an exception. If control reaches line 20 in C2, an exception is raised, and control flows to line 9 in M2 where the exception is handled. Thus, like the halt statement, a throw-catch construct also affects control flow across procedures in a program: because of the throw statement at line 20, the call to C2 at line 13a in B1 may not return, and the call to B2 at line 7a in M2 may return to line 9 instead of to line 7b.

Throw-catch constructs differ from halt statements in that they do not necessarily cause execution of the program to terminate; rather, they may cause execution to continue from some targeted statement. Also, procedure calls that contain these constructs may return to the calling procedure, even after encountering the constructs. Like halt statements, however, these constructs can have effects on control dependencies not accounted for by the HRB SDG-construction algorithm.

Throw-catch and halt constructs, along with constructs such as raise-when and setjmp()-longjmp(), are

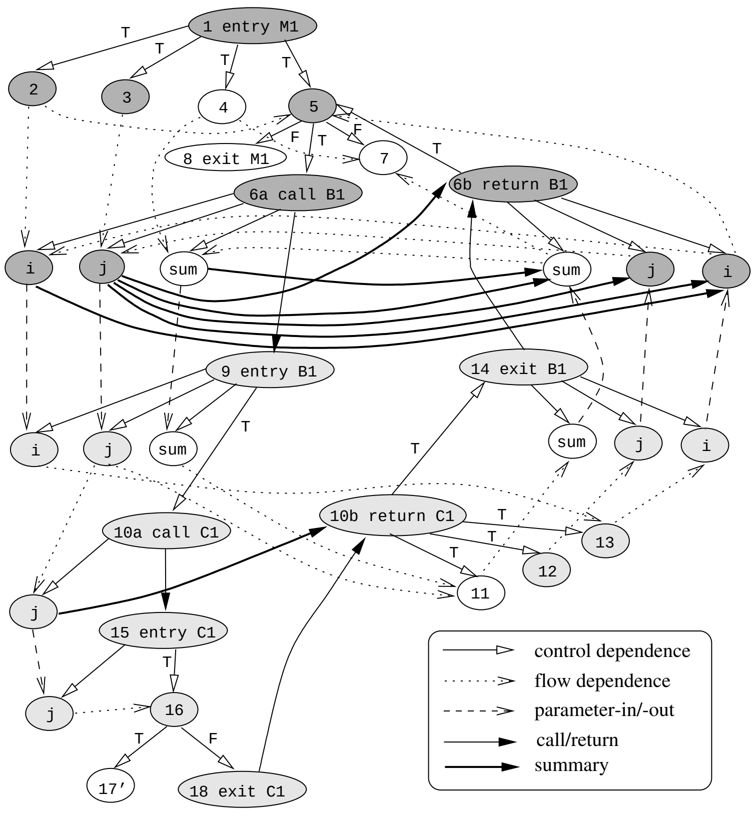

Figure 4: SDG for program `Sum2` of Figure 1.

examples of a class of interprocedural control constructs that may cause arbitrary interprocedural control flow. A common property of this class of constructs is that they may cause call sites to exist in which the control dependencies of the call statement differ from the control dependencies of statements that immediately follow the return from the call statement. It is this difference in dependencies, and its effects on slicing, that the HRB SDG and slicing algorithm do not accommodate.

A sufficient condition for the control dependencies of a call statement and statements following the return from the call to differ is that from the call statement, control may not return to the call site. We refer to such call sites, which are important for our algorithm, as *potentially non-returning call sites* (PNRCs).

4 Extended-SDG-Based Slicing

This section discusses our extensions to the SDG to represent the effects of arbitrary interprocedural control flow, our algorithm for constructing these SDGs, and the modifications required to the HRB slicing algorithm to perform the slicing on extended SDGs.

4.1 The Extended SDG

Our extended SDG differs from the HRB SDG in four ways, which we describe and illustrate in turn.

First, our SDGs use both call and return vertices to represent each PNRC, and entry and exit vertices to represent each procedure that can be reached from a PNRC. The SDGs also contain return edges that, similar to call edges in the HRB SDG, connect exit vertices in called procedures to the associated return vertices in calling procedures. Two vertices are associated with a PNRC to represent the differences in control dependencies of the call and return sites; two vertices are associated with procedures called from PNRCs for a similar reason. Previous approaches require only call and entry vertices, and thus, only call edges, because they assume that all called procedures return to their call sites.

For illustration, consider Figure 4, which depicts the SDG for program Sum2 (of Figure 1). The presence of the halt statement in line 17′ causes PNRCs 6a and 10a. Thus, there are both call and return vertices for each of these call sites, and entry and exit vertices for B1 and C1. The SDG contains return edges (14 exit B1, 6b return B1) and (18 exit C1, 10b return C1).

Second, in our SDGs, actual-in parameter vertices are associated with the call vertex, and actual-out parameter vertices with the return vertex; similarly, formal-in parameter vertices are associated with the entry vertex, and formal-out vertices with the exit vertex. The assignments represented by actual-out and formal-out vertices occur only if control reaches the corresponding return vertex or exit vertex; associating the actual- and formal-out vertices with return and exit vertices, therefore, correctly represents dependencies for parameter-out vertices in the SDG. Previous approaches associate both actual-in and actual-out vertices with call vertices, and both formal-in and formal-out vertices with entry vertices.

In Figure 4, for example, actual-in vertices for i, j, and sum are associated with 6a, and formal-in vertices for i, j, and sum are associated with the entry vertex for B1. Likewise, actual-out vertices are associated with 6b, and formal-out vertices with the exit vertex for B1.

Third, in addition to HRB SDG summary edges that represent transitive data and control dependencies for variables that are modified by a procedure, our SDGs contain summary edges from actual-in vertices to return vertices; these edges represent transitive data and control dependencies that influence return from a called procedure. To distinguish these types of summary edges, we call the former (summary edges described by HRB) *df-summary* edges because they affect the data flow in the calling procedure, and the latter *cd-summary* edges because they affect the control dependencies in the calling procedure. The cd-summary edges represent the effects on control dependence caused by PNRCs; they enable computation of interprocedural slices with minor changes to the HRB slicing algorithm. Note that the presence of cd-summary edges can force the inclusion of additional df-summary edges that would not be created by the HRB SDG-construction algorithm.

Summary edges are shown in bold in Figure 4. The edge from actual-in vertex for j at 6a to actual-out vertex for sum at 6b is a df-summary edge, and indicates that the value of sum following the call at line 6 depends on the value of j prior to that call. The summary edge from the same vertex for j to 6b, however, is a cd-summary edge, and indicates that return of control from the call at line 6 depends on the value of j prior to that call. Finally, the df-summary edge from actual-in vertex for j at 6a to actual-out vertex for j at 6b is induced by the cd-summary edge from actual-in vertex for j at 10a to 10b; the HRB SDG-construction algorithm would not compute such summary edges.

As a second example, consider the SDG for program Sum3 (of Figure 3) shown in Figure 5. The PDG for C2 in Figure 5 contains an exceptional-exit vertex because the procedure propagates an exception [18]. The call site at line 7 in Sum3 is a PNRC; therefore, the SDG contains call and return vertices corresponding to that call. The actual-in vertices for the call are associated with 7a, whereas the actual-out vertices are associated with 7b. The cd-summary edge from the actual-in vertex for j to 7b summarizes control-

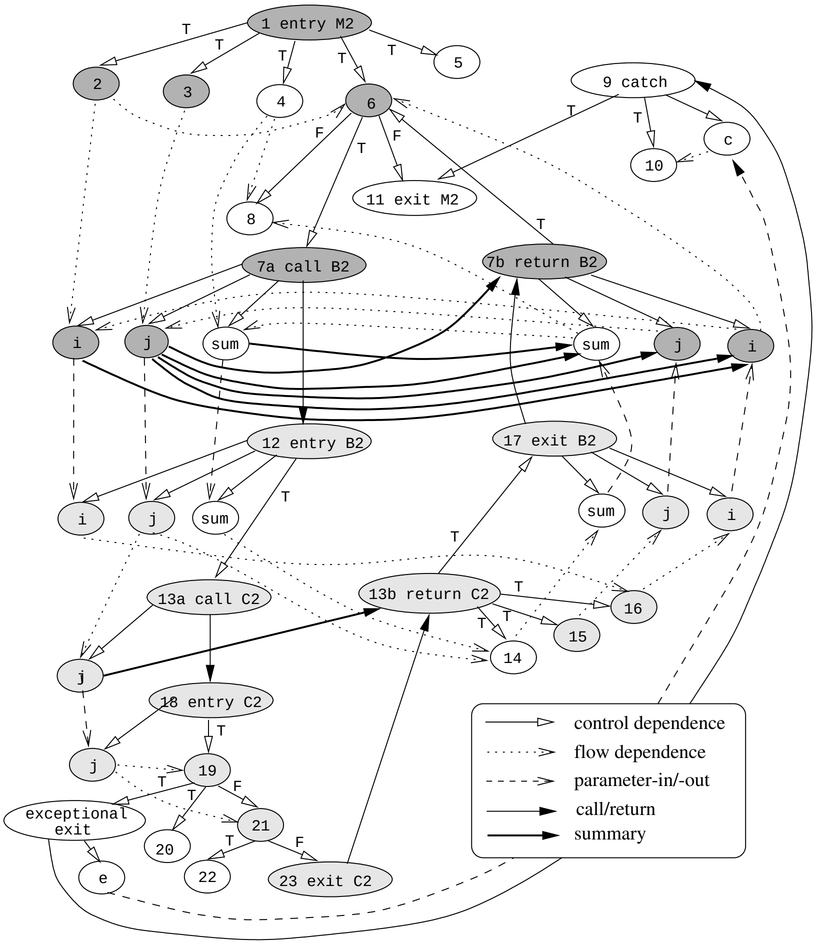

Figure 5: SDG for program `Sum3` of Figure 3.

---

dependence side-effects caused by the PNRC.

Fourth, in addition to the return edges connecting exit and return vertices, our SDGs contain additional interprocedural edges representing arbitrary flow of control, as required. For example, in Figure 5, there is an interprocedural edge connecting C2's exceptional exit with the catch handler at statement 9 in M2.

4.2 Constructing the Extended SDG

Our algorithm for constructing the SDG (Figure 6) (1) preprocesses program $\mathcal{P}$ to identify PNRCs, and performs MOD/REF analysis, (2) constructs PDGs for procedures in $\mathcal{P}$, and (3) constructs an SDG for $\mathcal{P}$.

4.2.1 Preprocess Program

The first step of `ConstructSDG` identifies PNRCs in a program.

PNRCs exist because of arbitrary interprocedural control constructs. These constructs are characterized by three elements:

Source element: a syntactic element that initiates a transfer of control;
Target element: a syntactic element that establishes a destination for a transfer of control;

```
algorithm ConstructSDG
input     CFG : control-flow graph for each procedure P in 𝒫
output    SDG : system-dependence graph for 𝒫
begin ConstructSDG
    /* Step 1 — preprocess program */
1a.  identify PNRCs
1b.  perform MOD/REF analysis
    /* Step 2 — construct PDGs for procedures */
2a.  construct ACFGs
2b.  compute intraprocedural control dependencies
2c.  compute intraprocedural data dependencies
2d.  construct PDGs
    /* Step 3 — construct SDG from PDGs */
3a.  connect PDGs
3b.  compute summary edges
end ConstructSDG
```

Figure 6: Algorithm for construction of an SDG.

Information element: a content element that represents the type of information transferred from source element to target element. An information element is generated at a source element, and determines the target for that source element; each target element also has an associated information element.

Information elements are relevant for identifying PNRCs because they determine the target element to which control flows following a procedure call. For constructs, such as halt, that do not have a target element (in the source program), we use a constant value as the information element associated with occurrences of source elements for those constructs; thus, for example, all procedures that contain halt statements propagate information element "halt."

The PNRC-identification algorithm performs reachability analysis on the the program's call graph[5] to build, for each procedure, a *propagation set* that contains information elements propagated by that procedure. To do this, the algorithm uses information regarding target elements that enclose a call site; such target elements may be destinations for information elements propagated by the called procedure.[6]

While iteratively building propagation sets, the algorithm also records, for each call site, target elements (enclosing that call site) in the calling procedure to which control can return following that call. A call site that has at least one such target element associated with it is, therefore, a PNRC. Information about target elements that are associated with PNRCs is used during Step 2 of `ConstructSDG`.[7]

Figure 7 shows the PNRC-identification algorithm, `ComputePNRC`. `ComputePNRC` takes three inputs: the program call graph, the propagation set for each procedure, and enclosing target elements for each call site. `ComputePNRC` returns a target-element set for each call site — a target-element set for a call site contains target elements enclosing the call site to which control can return following the call site. The algorithm initializes *workList* with those procedures that have non-empty propagation sets (lines 1–4). On each iteration, the algorithm removes a procedure P from *workList*, and processes call-graph predecessors of P (lines 6–7). For each predecessor *pred*, the algorithm saves the current propagation set for *pred* (line 8),

[5] A *call graph* for a program $\mathcal{P}$ contains a vertex N for each procedure P in $\mathcal{P}$, and an edge from vertex N_i to vertex N_j if the procedure corresponding to N_i contains a call site that calls the procedure corresponding to N_j. The precision of the call graph varies with the construction algorithm used (e.g., [7, 15]), and can impact the precision of graphs, such as the SDG, that rely on it.

[6] Information about target elements that enclose a call site can be saved during a syntax-directed construction of the CFGs: Reference [18] illustrates this for throw–catch constructs.

[7] If $\mathcal{P}$ contains statically unreachable code, we may identify call sites as PNRC that in fact are not; however, this identification leads only to larger-than-necessary graphs, and does not alter the precision of slices constructed on the extended SDG.

```
function ComputePNRC
input     CG : call graph for program 𝒫
          propSet[P] : information elements propagated by proc. P
          enclTargetElem[C]: target elements that enclose call site C
output    retTargetElem[C] : target elements to which control may return following call site C
declare   workList : list of procedures
          oldPropSet, infoElem : temporary sets for information elements
begin ComputePNRC
 1. workList = φ
 2. foreach P ∈ 𝒫 such that propSet[P] ≠ φ do
 3.     add P to workList
 4. endfor
 5. while workList ≠ φ do
 6.     remove P from workList
 7.     foreach predecessor pred of P in CG do
 8.         oldPropSet = propSet[pred]
 9.         foreach call site C in pred that calls P do
10.             infoElem = propSet[P] ∩ (information elements associated with enclTargetElem[C])
11.             retTargetElem[C] = enclTargetElem[C] whose associated information elements appear in infoElem
12.             if infoElem ≠ propSet[P] then
13.                 add (propSet[P] − infoElem) to propSet[pred]
14.                 add "super exit" to retTargetElem[C]
15.             endif
16.         endfor
17.         if oldPropSet ≠ propSet[pred] then
18.             add pred to workList
19.         endif
20.     endfor
21. endwhile
end ComputePNRC
```

Figure 7: Algorithm for computing PNRCs.

and next builds the new propagation set for *pred* from the propagation set for P (lines 9–16). To determine the new propagation set for *pred*, `ComputePNRC` examines call sites in *pred* that call P. For a call site C, the algorithm identifies those information elements (propagated by P) that match the information elements associated with target elements enclosing C (line 10), and records the corresponding target elements in the target-element set for C (line 11). If at least one information element propagated by P does not match the information element associated with any target element, the propagation set for *pred* may increase. Therefore, the algorithm adds such information elements to the propagation set for *pred*, while ensuring that those information elements do not already appear in the propagation set for *pred* (line 13). The algorithm also records "super exit" (explained in the next Section 4.2.2) as a target element for C (line 14). Finally, if the propagation set for *pred* changes from a previous iteration, the algorithm adds *pred* to *workList* (lines 17–19).

The algorithm in Figure 7 ignores the details of the way in which information elements are matched in line 10. For example, such matching may need to proceed in a certain order, determined by the order of the target elements, and may require a test of a subtype relationship among information elements; Reference [18] illustrates this process for throw–catch statements.

To illustrate the actions of `ComputePNRC` with an example, consider program `Sum3`, which contains a throw statement and a halt statement. Initially, the propagation set for procedure `C2` contains two information elements — `t1` and "halt"; the propagation sets for procedures `B2` and `M2` are empty. `ComputePNRC`, therefore, initializes `C2` on *workList*. Next, `ComputePNRC` examines call site 13a. There are no target elements enclosing that call site. Therefore, the algorithm adds `t1` and "halt" to the propagation set for `B2`, and adds "super exit" to the target-element set for call site 13a; the algorithm also adds `B2` to *workList*. In the iterations that follow, the algorithm removes `B2` from *workList* and processes call site 7a in `M2`, which calls `B2`. The target-element set for call site 7a contains a single element — vertex 9 — that has information element `t1` associated with it. Therefore, in line 10, the algorithm determines the contents of *infoElem* as `t1`, and in line 11, adds vertex 9 to the target-element set for call site 7a. Because the propagation set for `B2` contains `t1` and "halt," the condition in line 12 of `ComputePNRC` evaluates to true. `ComputePNRC` then adds "halt" to the propagation set for `M2`, and "super exit" to the target-element set for call site 7a. After the algorithm terminates, call sites 7a and 13a have non-empty target-element sets, and are, therefore, identified as PNRCs.

Following identification of PNRCs, `ConstructSDG` (line 1b) performs data-flow analysis (e.g., [14]) to compute two sets of variables for each procedure: MOD, which contains variables that are modified, and REF, which contains variables that are referenced.

4.2.2 Construct PDGs for Procedures

The second step of `ConstructSDG` creates PDG information for each procedure in $\mathcal{P}$. The PNRC computation in Step 1 identified call sites to which control may not return. These constructs may cause statements following the return from the call to be control dependent on conditional statements in other procedures. Although intraprocedural control-dependence analysis cannot identify these conditional statements, it can, using an augmented control-flow graph [10], create partial control-dependence information that can later be used to compute interprocedural slices.

An *augmented control-flow graph* (ACFG) for a procedure P is a control-flow graph, augmented with placeholder vertices that represent conditional statements in other procedures on which statements in P are control dependent. An ACFG has a unique vertex, *super entry*, that acts as a placeholder for entry to P. For each return vertex associated with a PNRC that calls procedure P_i, an ACFG contains a unique conditional vertex, *return predicate*, that acts as a placeholder for the conditional statements on which return from P_i is control dependent. A return-predicate vertex has an edge to the corresponding return vertex, and one edge to each target element associated with the corresponding PNRC; each edge from a return-predicate vertex is labeled with the target vertex of that edge. An ACFG also contains a unique vertex, *super exit*, that represents all exits from P. Finally, an ACFG contains control-flow edges that connect control-flow graph vertices with these new vertices.

The ACFG-construction algorithm, invoked in line 2a of `ConstructSDG`, adds return-predicate vertices to the control-flow graph of a procedure, using the target-element sets that were saved by the PNRC computation at call sites, to construct a partial ACFG. Next, the ACFG-construction algorithm determines the the immediate postdominator of each return-predicate vertex. Finally, the algorithm connects the call vertex for each PNRC to the immediate postdominator of the return-predicate associated with that PNRC.

When an ACFG is used to compute intraprocedural control dependencies using a control-dependence computation algorithm [5], the control dependencies for all vertices N that are control dependent on conditional statements in called procedures will contain a return-predicate vertex.

The graph on the left in Figure 8 depicts the ACFG for `M1`. In addition to super-entry and super-exit vertices, the ACFG contains one return-predicate vertex, RP6b, which represents the conditional statements on which return from procedure `B1` depends; RP6b has two out edges that are labeled '6b' and 'super exit' —

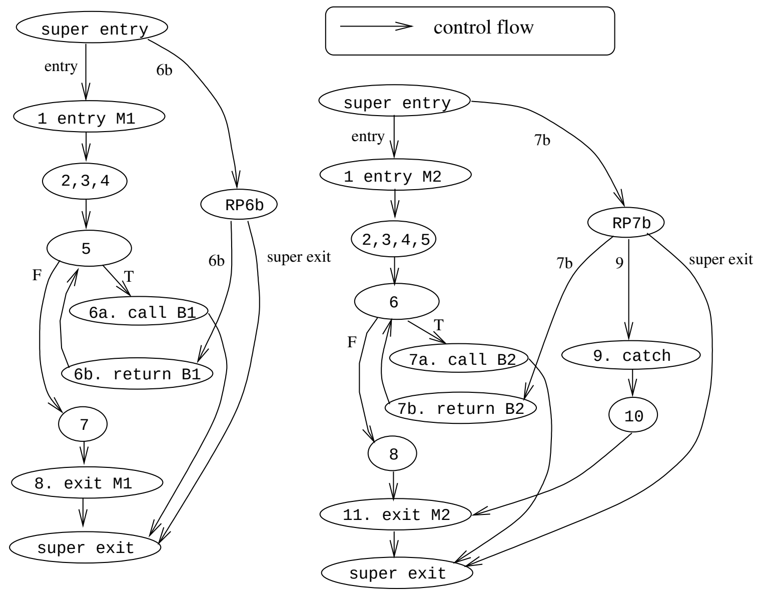

Figure 8: ACFGs for procedure `M1` of Figure 1 (left) and `M2` of Figure 3 (right).

the target vertices of the two edges. The ACFG also includes edges that connect control-flow graph vertices with these new vertices. When this ACFG is used in control-dependence computation, the computation finds that vertices 6b and 5 are control dependent on RP6b, and vertex 7 is control dependent on 5F. These intraprocedural control dependencies differ from those that would be obtained using an analysis that does not consider the potential side-effects that a called procedure may have on the control dependencies in calling procedures.

The graph on the right in Figure 8 shows the ACFG for procedure `M2`. The return-predicate vertex associated with the PNRC at vertex 7a has an edge to the corresponding return vertex, and one edge to each target element to which control can return following the call at vertex 7a; these edges are labeled '7b', '9', and 'super exit' — the targets of the edges.

Using a data-flow analysis algorithm [1], `ConstructSDG` next computes the data-dependence information needed for the program-dependence-graph construction.

As its final task in Step 2, `ConstructSDG` creates the PDG for each procedure P in $\mathcal{P}$. Using the vertices present in the ACFG for P, the algorithm creates a control-dependence edge (v_i, v_j) for each vertex v_j that is control dependent on v_i, with an exception: if v_i is a return-predicate vertex, v_j is control dependent on v_i with edge-label 'L', and v_j and L are the same vertex, the algorithm does not create an edge from vertex L to v_j. For example, in the control-dependence subgraph of the PDG for `M1` in Figure 4, vertex 5 is control dependent on return vertex 6b because in the ACFG for `M1`, vertex 5 is control dependent on a return-predicate vertex with edge-label '6b'; vertex 6b does not have any control-dependence edge incident on it because in the ACFG, vertex 6b is control dependent on a return-predicate vertex with edge-label '6b'. As another example, consider the PDG for `M2` from the SDG for `Sum3` shown in Figure 5: `ConstructSDG` creates a control-dependence edge from vertex 9 to vertex 10 because in the ACFG for `M2`, vertex 10 is control dependent on a return-predicate vertex with edge-label '9'; likewise, `ConstructSDG` creates a control-dependence edge from 7b to 6; `ConstructSDG` does not create a control-dependence edge incident on vertex

9, however, because vertex 9 is control dependent on a return-predicate vertex with edge-label '9'.

Next, the algorithm adds data-dependence information, and expands call and return sites to represent parameters and globals; the algorithm uses the HRB SDG-construction algorithm to expand call sites to represent actual and formal parameters and globals.

4.2.3 Construct Extended SDG from PDGs

Step 3 of `ConstructSDG` constructs an SDG from the individual PDGs created in Step 2. In line 3a, at each call site in a program, `ConstructSDG` connects the PDG for the called procedure to the PDG for the calling procedure using five types of edges: (1) a call edge connects a call vertex to the entry vertex of the called procedure; (2) a parameter-in edge connects each actual-in vertex at the call site to the corresponding formal-in vertex; (3) a parameter-out edge connects each formal-out vertex at the call site to the corresponding formal-out vertex; (4) if a call site is PNRC, a return edge connects the exit vertex of the PDG for the called procedure to the return vertex associated with the call site; (5) an interprocedural edge connects source and target elements of an arbitrary inteprocedural control construct. The first three types of edges are also used in the HRB SDG; the last two types of edges are required to represent PNRCs and arbitrary interprocedural control constructs.

Figure 4 shows the SDG for program `Sum2`, and illustrates the first four types of edges used to connect PDGs. A call edge connects call vertex 6a to entry vertex 9; parameter-in edges connect actual-in vertices at call vertex 6a to formal-in vertices at entry vertex 9; parameter-out edges connect formal-out vertices at exit vertex 14 to actual-out vertices at return vertex 6b; a return edge connects exit vertex 14 to return vertex 6b. Figure 5 illustrates the final type of edge: an interprocedural edge connects the exceptional-exit vertex for the throw statement to the vertex for the catch statement where the raised exception is handled.

In line 3b, `ConstructSDG` adds summary edges from actual-in vertices to actual-out vertices at each call site. In addition to df-summary edges created by the HRB algorithm, our algorithm also creates cd-summary edges from actual-in vertices to return vertices to represent transitive data and control dependencies that influence return from a called procedure.

Figure 9 presents `ComputeSummaryEdges`, an extended version of the algorithm for computing summary edges presented in Reference [16]; the extensions to the algorithm enable us to compute cd-summary edges in addition to df-dummary edges. The HRB SDG-construction algorithm uses a worklist approach to extend paths backwards along flow and control edges in the SDG, starting at all formal-out vertices. On reaching an actual-out vertex, that algorithm extends the path along control and previously-computed summary edges; on reaching a formal-in vertex, the algorithm creates a summary edge from an actual-in vertex to an actual-out vertex, and further extends paths that were partially extended up to that actual-out vertex. Our extensions to that algorithm (shown with asterisks in Figure 9) are three-fold. First, our algorithm also extends paths starting at exit vertices of procedures that may not return to call sites. Second, on reaching a return vertex, our algorithm extends the path along control and previously-computed summary edges. Third, on reaching a formal-in vertex, the algorithm considers the type of the final vertex along the path: if the final vertex is a formal-out vertex, the algorithm creates a df-summary edge from an actual-in to an actual-out vertex; if, however, the final vertex is an exit vertex, the algorithm creates a cd-summary edge from an actual-in to a return vertex; in either case, the algorithm further extends paths that were partially extended up to the target of the new summary edge.

```
function ComputeSummaryEdges
input    G : system dependence graph for program P
output   set of summary edges
declare  PathEdge, WorkList : sets of SDG paths
         SummaryEdge : set of summary edges
begin ComputeSummaryEdges
  1.  PathEdge = φ; SummaryEdge = φ; WorkList = φ
  2.  foreach w ∈ FormalOutVertices(G) do
  3.      insert (w → w) into PathEdge, WorkList
  4.  endfor
* 5.  foreach P ∈ PotentiallyNonRetProcs(P) do
* 6.      w = ExitVertex(P)
* 7.      insert (w → w) into PathEdge, WorkList
* 8.  endfor
  9.  while WorkList ≠ φ do
 10.      remove (v → w) from WorkList
 11.      case v ∈ ActualOutVertices(G)
 12.          foreach x such that (x → v) ∈ (SummaryEdge ∪ ControlEdges(G)) do
 13.              Propagate((x → w))
 14.          endfor
*15.      case v ∈ ReturnVertices(G)
*16.          foreach x such that (x → v) ∈ SummaryEdge do
*17.              Propagate((x → w))
*18.          endfor
 19.      case v ∈ FormalInVertices(G)
 20.          foreach c ∈ CallSitesTo(Proc(w)) do
 21.              x = CorrespondingActualIn(c, v)
*22.              if w ∈ ExitVertices(G) then
*23.                  y = CorrespondingReturn(c, w)
 24.              else y = CorrespondingActualOut(c, w) endif
 25.              insert (x → y) into SummaryEdge
 26.              foreach a such that (y → a) ∈ PathEdge do
 27.                  Propagate((x → a))
 28.              endfor
 29.          endfor
 30.      default
 31.          foreach x such that (x → v) ∈ (FlowEdges(G) ∪ ControlEdges(G)) do
 32.              Propagate((x → w))
 33.          endfor
 34.      endcase
 35.  endwhile
 36.  return SummaryEdge
end ComputeSummaryEdges
```

Figure 9: Algorithm for construction of summary edges in the presence of PNRCs.

`ComputeSummaryEdges` uses *PathEdge* and *WorkList* to store partially-extended paths, and *SummaryEdge* to store summary edges. `ComputeSummaryEdges` also uses several auxiliary functions: `Propagate`(e) inserts edge e into *PathEdge* and *WorkList*; `PotentiallyNonRetProcs`$(\mathcal{P})$ returns the set of potentially non-returning procedures in program $\mathcal{P}$; `Proc`(v) returns the procedure that contains SDG vertex v; `CallSitesTo`(P) returns the set of call sites that call procedure P; other functions, such as `FormalOutVertices`(G), return the set of named vertices from SDG G.

`ComputeSummaryEdges` initializes *PathEdge* and *WorkList* with formal-out and exit vertices (lines 1–8), and iterates until *WorkList* becomes empty. On each iteration, the algorithm extends paths backwards along flow- and control-dependence edges (lines 30–33). If a path reaches an actual-out vertex, the algorithm extends the path along control-dependence and previously-computed summary edges (lines 11–14); on encountering a return vertex, the algorithm extends the path along previously-computed summary edges (lines 15–18). When a path (with final vertex w) reaches a formal-in vertex v, the algorithm creates summary edges at all call sites that call the procedure containing w (lines 19–29). The algorithm first checks the type of w (line 22): if w is an exit vertex, the algorithm creates a cd-summary edge from the actual-in vertex corresponding to v to the return vertex that is connected to w at call site c (lines 23, 25); otherwise, `ComputeSummaryEdges` creates a df-summary edge from the actual-in vertex corresponding to v to the actual-out vertex corresponding to w at call site c (lines 24, 25). The algorithm next merges paths in *PathEdge* that were partially extended up to the target of the new summary edge with the new summary edge, and adds the merged paths to *WorkList* and *PathEdge* (lines 26–28).

4.3 Performing Slicing on the Extended SDG

Like the HRB slicing algorithm, our algorithm performs slicing in two phases; however, our algorithm contains a minor modification that lets it accommodate return and interprocedural edges that are not present in the HRB SDG. During the first phase, the algorithm traverses backwards along flow-dependence, control-dependence, call, parameter-in edges, and summary edges; the algorithm thus includes in the slice vertices that belong in the procedure in which slicing was initiated, or in some direct or indirect caller of that procedure. During the second phase, the algorithm traverses backwards along flow-dependence, control-dependence, return, interprocedural, parameter-out, and summary edges, starting at vertices included in the first phase; this phase includes vertices from procedures that are called directly or indirectly by the procedure in which slicing was initiated, or those procedures that are called by callers of the procedure in which slicing was initiated.

The vertices included during the first phase by slicing on the extended SDG for `Sum2` for slicing criterion $< 6b, \{i\} >$ are shown in the darker shade in Figure 4. The vertices included in the second phase are shown in a lighter shade.

Figure 5 shows the vertices included in the slice for criterion $< 7b, \{i\} >$. Because of the addition of exception-handling constructs in `Sum3`, the predicate vertex 18 that controls the throw vertex is included in the slice because it may affect the value of i at 7b. Interprocedural slicing, based on an SDG that does not represent side effects on control dependencies caused by PNRCs, may miss some statements that may affect the slicing criterion. The HRB SDG-construction algorithm would construct the same SDG representations for both `Sum1` and `Sum2`, and the HRB slicing algorithm would then compute the same slices for $< 6b, \{i\} >$ on `Sum1` and `Sum2`. A comparison of the slices computed in Figures 2 and 4 illustrates that the slice in Figure 2, when computed on `Sum2`, fails to include statements 3, 10a, 12, 13, and 16 — statements that may affect the value of i at statement 6b.

5 Evaluation

This section presents both an an analytical evaluation of our extended SDG, our SDG-construction algorithm, and our SDG slicing algorithm, and an empirical evaluation based on a prototype implementation.

5.1 Analytical Evaluation

We compare the complexity of our SDG-construction algorithm with the complexity of the HRB SDG-construction algorithm, and the size of the extended SDG with the size of the HRB SDG using the following parameters:

Procs	Number of procedures in a program
E_P	Maximum number of flow and control edges in the PDG for any procedure P
C_P	Maximum number of call sites in any procedure P
$PNRC_P$	Maximum number of PNRCs in any procedure P
$Params_P$	Maximum number of parameters in any procedure P
I_P	Maximum number of information elements propagated by any procedure P
T_P	Maximum number of information elements associated with target elements that enclose a call site in any procedure P

The HRB SDG-construction algorithm does not perform the actions of `ConstructSDG` in lines 1a and 2a, and therefore, avoids the computation costs associated with those actions. Line 1a of `ConstructSDG` calls `ComputePNRC`, which processes a call graph. In the worst case, the call graph may be fully connected; therefore, the upper bound for the number of iterations of the two outer loops (in lines 5 and 7) of `ComputePNRC` is $Procs^2$. For each procedure, `ComputePNRC` may process all call sites in that procedure (bounded by C_P), and for each call site, `ComputePNRC` computes the intersection of information elements propagated by the procedure called at that call site and information elements associated with target elements enclosing that call site (bounded by $I_P * T_P$). Therefore, the overall complexity of `ComputePNRC` is $O(Procs^2(C_P * I_P * T_P))$.

Line 2a of `ConstructSDG` constructs an ACFG for each procedure. To construct an ACFG, the algorithm creates a return-predicate vertex for each PNRC in a procedure, and creates out-edges from that return-predicate vertex to each vertex in target-element set for that PNRC. The size of a target-element set is bounded by T_P. Therefore, the overall complexity of line 2a of `ConstructSDG` is $O(Procs * PNRC_P * T_P)$.

The HRB SDG-construction algorithm has a lower cost for line 3a of `ConstructSDG` because the HRB SDG-construction algorithm does not create return edges for PNRCs, and interprocedural edges to represent arbitrary interprocedural control flow. However, because the additional cost of creating these edges is linear in the number of nodes in the SDG, the cost does not add to the asymptotic complexity of line 3a.

Finally, the HRB SDG-construction algorithm does not compute cd-summary edges, and therefore, has a lower cost for line 3b of `ConstructSDG`. The cost of computing summary edges for a single procedure in the HRB SDG-construction algorithm is:

$$O(\underbrace{Params_P}_{(i)}(\underbrace{E_P}_{(ii)} + \underbrace{(C_P * Params_P^2)}_{(iii)}))\ [16].$$

The equation represents the cost of summary-edge computation for a procedure as the product of the number of formal-out vertices for that procedure (term (i)), and the number of PDG edges for that procedure, which is the sum of flow and control edges (term (ii)) and summary edges (term (iii)). Term (i) in the equation results because each formal-out vertex of a procedure is initialized on the worklist to be processed (Figure 9). Our extension to the summary-edge algorithm may also initialize the exit vertex of a procedure on the worklist; therefore, term (i) for our summary-edge computation is $Params_P + 1$. Term (ii) remains the same in our summary-edge computation. Term (iii) in the equation represents the number of summary edges in a procedure; this term only includes df-summary edges in the HRB SDG-construction algorithm. Our summary-edge computation also creates cd-summary edges, in addition to df-summary edges; therefore, term (iii) for our summary-edge computation is $(C_P * Params_P^2) + (PNRC_P * Params)$. Although, our summary-edge computation creates additional df-summary edges that are not created by the HRB-SDG construction algorithm, these additional edges are subsumed in term (iii) of the above equation, and therefore, do not increase the cost. The overall cost of summary-edge computation in our SDG-construction algorithm is:

$$O((Params_P + 1)(E_P + (C_P * Params_P^2) + (PNRC_P * Params))).$$

Table 1: Subject programs for experiment

Program	Description	Lines of code	HRB SDG Number of nodes	HRB SDG Summary edges (df)	Extended SDG Number of nodes	Extended SDG Summary edges (df)	Extended SDG Summary edges (cd)
armenu.c	AAS User Interface [9]	8939	17797	13369	17871	23353	1816
dejavu1.c	Intraprocedural regression test selector [17]	4385	3958	987	4100	995	24
dejavu2.c	Interprocedural regression test selector [17]	5571	5389	1424	5571	1474	43
mpegplayer.c	MPEG player	6800	13755	8900	13923	12453	1607
space.c	Parser for antenna-array description language	11474	28609	34486	28834	34960	1724

Clearly the asymptotic complexity of summary-edge computation is the same in both versions of the SDG-construction algorithm.

The cost of the interprocedural slicing algorithm is linear in the number of edges in the SDG. The extended SDG contains several edges in addition to those in the HRB SDG — return edges for PNRCs, interprocedural edges for arbitrary interprocedural control flow, and cd-summary edges for side-effects on control dependence caused by PNRCs. These edges increase the size of the extended SDG, and thereby contribute to increased cost of slicing on the extended SDG. None of these additional edges, however, causes an asymptotic increase in the size of the extended SDG over the size of the HRB SDG. Therefore, the asymptotic cost of performing slicing on the extended SDG is the same as the cost of performing slicing on the HRB SDG.

The complexity analysis shows that the predominant expense of `ConstructSDG` over that of the HRB SDG-construction algorithm is the identification of PNRCs in Step 1. The cost of the interprocedural slicing algorithm is linear in the size of the extended SDG; that cost differs slightly from the cost of the HRB slicing algorithm because the HRB SDG does not have cd-summary edges.

Both the PNRC identification and MOD/REF analyses performed in Step 1 of `ConstructSDG` let us optimize the SDG representation: those call sites that are not PNRC do not require both call and return vertices; those formal parameters that are not defined in a procedure do not require formal-out vertices. An alternative approach creates ACFGs for procedures assuming that all call sites in the procedure are PNRCs, and then proceeds with Steps 2 and 3 of `ConstructSDG`, creating the PDG for the procedure, connecting it to the existing SDG, and computing summary edges. Slicing this extended SDG yields the same results as slicing an extended SDG that was constructed using the preprocessing step, at a cost of the additional space and time necessary to accommodate the additional PNRCs. This approach, however, can support the use of partial PNRC analyses in which portions of the program are not processed, or demand approaches to constructing the SDG in which PDGs for individual procedures are not created until those procedures are reached in the slicing.

5.2 Empirical Evaluation

To investigate the impact of arbitrary interprocedural control flow on slice computation, we implemented a prototype and performed an experiment on C programs; our prototype handles PNRCs caused by `exit()` calls. We implemented the HRB SDG, the extended SDG, and the SDG-based slicing algorithm using the Aristotle Analysis System (AAS) [9]. The sizes of the SDGs and the precision of the slicing depend to a large extent on the precision of the data-flow information used to construct the graphs. To account for the effects of aliasing on data-flow analysis, we replaced the AAS front-end with the PROLANGS Analysis Framework

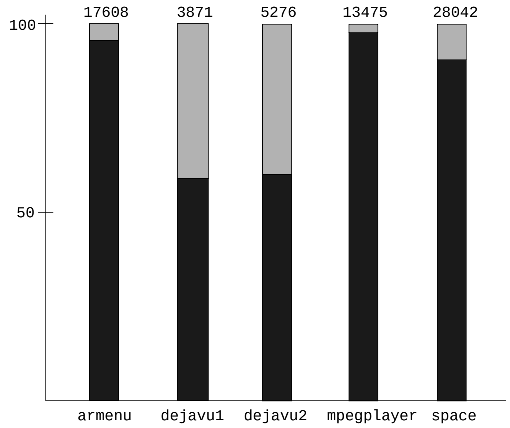

Figure 10: Slice differences.

(PAF) [6]; PAF provides a parser, and modules that perform flow- and context-sensitive alias analysis [13], and modification side-effect analysis [14]. We used PAF to gather the information required by the AAS tools: data-flow, control-flow graphs, symbol tables, and source-code mapping. The SDGs constructed in our implementation, therefore, account for data dependencies that arise because of pointer aliasing and parameter aliasing. Because PAF does not perform analysis for function pointers, our implementation does not currently incorporate the effects of function pointers. Our subject programs, however, contain no function pointers.

An SDG contains parameter nodes based on summary information for call sites and procedures [12]: the summary information for a call site specifies variables that are referenced (REF), and those that may be modified by the procedure called at that call site (MOD); the summary information for a procedure lists similar information for that procedure.

For our experiment, we used five C programs; Table 1 gives information about these subjects. In addition to a brief description and the number of lines of code, for each subject, the table lists the sizes of the HRB SDG and the extended SDG in terms of numbers of nodes and summary edges in each graph.

For each statement in each subject program, we computed an interprocedural slice using the HRB SDG and our extended SDG. We then compared the differences in the slices for each statement, and recorded the number of slices that differ. Figure 10 uses a segmented bar graph to illustrate our results. Each bar represents one of our subject programs. The lightly shaded segment represents the percentage of slices that were the same, and the darkly shaded segment represents the percentage of slices that differ. The number above each bar represents the number of slices that were computed for that program.

As the bar graph illustrates, the percentages of slices that differ range from 58% to 97%. Three of the subjects, `armenu.c`, `mpegplayer.c`, and `space.c`, contain PNRCs at the beginning of their entry procedures, which affects the control dependencies of almost every statement in those programs. Therefore, more than 90% of the slices for these three subjects that were computed on the HRB SDG differ from those computed on the extended SDG.

Given these differences, we would like to determine the extent to which the slices differ, and the cause of those differences — we are currently investigating this. Preliminary comparisons of the slices obtained for programs `dejavu1.c` and `dejavu2.c` indicate that the slices that differ may in fact fail to identify a significant percentage of statements that belong to those slices. Among the slices for `dejavu1.c` that differed, 40% failed to identify upto 20% of the statements that were in the slices, and 59% failed to identify between 20% and 40% of the statements; a very small percentage of the slices missed as many as 80% to 100% of the statements. Among the slices for `dejavu2.c` that differed, 99% missed up to 20% of the statements.

6 Related Work

In Reference [10], we discussed the effects of `halt` statements on interprocedural control dependencies, presented a definition of interprocedural control dependence that supports the relationship of control and data dependence to semantic dependence, and presented an algorithm for calculating interprocedural control dependencies. In that work, however, we did not explore the relationship between interprocedural control dependencies and slicing techniques. This paper addresses that relationship, and extends the focus beyond halt statements to arbitrary interprocedural control flow.

In Reference [8], Harrold and Ci presented a data-flow-analysis-based interprocedural slicing technique that correctly handles programs that contain `halt` statements. Although not discussed in [8], that approach can be extended to accommodate arbitrary interprocedural control flow. However, as discussed in Section 1, SDG-based and data-flow-analysis-based slicing techniques form two general classes of slicing approaches that offer distinct cost/benefits tradeoffs. This paper addresses SDG-based slicing, focusing on accomodating arbitrary interprocedural control-flow.

Construction of interprocedural graphs in the presence of function pointers or dynamic dispatch may affect the precision of SDGs, and thus, the precision of slices obtained using those SDGs. Our experimentation with real C programs in which there were no function pointers or dynamic dispatch shows that the problem of arbitrary interprocedural control flow can arise and be addressed in cases in which function pointers and dynamic dispatch do not occur. Nevertheless, problems of precision related to function pointers and dynamic dispatch must also be addressed, particularly for languages that more extensively employ these features. Various techniques have been reported that can improve the precision of interprocedural graphs (e.g., [2, 4, 7]). Our continuing work will investigate the incorporation of such techniques into our approach.

7 Conclusions

Program slicing has been rigorously researched as a potential aid in many common software engineering tasks. SDG-based slicing techniques form one important class of program slicing techniques. If SDG-based slicing techniques are to be useful for programs written in practice, however, they must function on programs that contain arbitrary interprocedural control flow. Our experiments show that arbitrary interprocedural control flow can have significant effects on slices computed on SDGs; these effects could contribute to unsafety in algorithms that depend on slice information. Our work addresses this problem by providing an approach to constructing SDGs, and computing slices on SDGs, that accomodates programs with arbitrary interprocedural control flow. We are implementing analysis tools for Java; when complete, these tools will let us extend our experimentation to object-oriented programs.

Acknowledgements

This work was supported in part by a grant from Microsoft Inc., by NSF NYI Award CCR-9696157 to Ohio State University, ESS Award CCR-9707792 to Ohio State University and Oregon State University, and CAREER Award CCR-9703108 to Oregon State University. The Programming Languages Research Group at Rutgers University developed, and freely distribute, the PROLANGS Analysis Framework. Alberto Pasquini provided the source code for the Space program. Jim Jones helped with the implementation.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Prin., Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[2] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proc. of the 18th Int'l. Conf. on Softw. Eng.*, pages 16–27, Mar. 1996.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Proc. of 1st Int'l Workshop on Automated and Algorithmic Debugging*, volume 749 of *Lec. Notes in Computer Science*, pages 206–222. Springer-Verlag, Nov. 1993.

[4] R.K. Chatterjee, B. G. Ryder, , and W. A. Landi. Complexity of concrete type-inference in the presence of exceptions. In *Proc. of Euro. Symp. on Prog.*, Apr. 1998.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Sys.*, 9(3):319–349, July 1987.

[6] Programming Languages Research Group. PROLANGS Analysis Framework, 1998.

[7] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. of Object-Oriented Prog. Sys., Lang. and Applications*, pages 108–124, Oct. 1997.

[8] M. J. Harrold and Ning Ci. Reuse-driven interprocedural slicing. In *Proc. of the Int'l Conf. on Softw. Eng.*, April 1998.

[9] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, Mar. 1997.

[10] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis*, March 1998.

[11] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. on Prog. Lang. and Sys.*, 11(3):345–387, July 1989.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Prog. Lang. and Sys.*, 12(1):26–60, Jan. 1990.

[13] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of SIGPLAN '92 Conf. on Prog. Lang. Design and Implem.*, pages 235–248, June 1992.

[14] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proc. of SIGPLAN'92 Conf. on Prog. Lang. Design and Implem.*, pages 56–67, June 1993.

[15] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Trans. on Softw. Eng. and Meth.*, 7(2):158–191.

[16] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *Proc. of Second ACM Conf. on Foundations of Softw. Eng.*, pages 11–20, Dec. 1994.

[17] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng.and Meth.*, 6(2):173–210, Apr. 1997.

[18] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Proc. of Int'l Conf. on Softw. Maint.*, Nov. 1998.

[19] M. Weiser. Program slicing. *IEEE Trans. on Softw. Eng.*, 10(4):352–357, July 1984.