Slicing Java Programs that Throw and Catch Exceptions

Matthew Allen
matthew@cs.wisc.edu

Susan Horwitz
horwitz@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St, Madison, WI 53706 USA

ABSTRACT

Exceptions are the preferred method for error handling in object-oriented languages like Java. Current program-slicing algorithms do not correctly deal with exception-handling constructs, because they do not account for the additional control and data dependences introduced by exceptions. This paper extends previous work on program slicing using the system dependence graph (SDG) to support slicing programs with exceptions.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Languages

Keywords

Program slicing, Java exceptions, Program dependence graph

1. INTRODUCTION

Program slicing is an operation introduced by Mark Weiser [18] that is useful for many applications, including program understanding, debugging, maintenance, and testing. This paper extends previous work on slicing to permit correct slicing of Java programs that include *try/catch/throw* constructs for throwing and handling exceptions.

A number of different definitions of slices have been proposed. For the purposes of this paper, the slice of a program from a component S (a statement or predicate) is the set of components that might affect whether and/or how often S executes, as well as the components that might affect the values of the variables used at S. Such slices can be computed efficiently using the *system-dependence graph* (SDG) representation of a program [7], in which vertices represent statements and predicates, and edges represent control and data dependences (more information on SDGs is provided in Section 2). Basically, a control-dependence edge $m \rightarrow n$ means that vertex m may affect whether and/or how often vertex n executes, while a data-dependence edge $m \rightarrow n$ means that vertex m may set the value of some variable used at vertex n. Therefore, the slice from a component S represented by a vertex v includes each component represented by a vertex w such that there is an interprocedurally-valid path in the SDG from w to v (i.e., w has a direct or transitive effect on v's execution).

An important question is which kinds of vertices w are considered to affect whether and/or how often a vertex v executes (thus ensuring that w is in the slice from v due to the control-dependence edge $w \rightarrow v$). Initial work on slicing considered only a very simple, structured language in which predicates were the only constructs that affected the flow of control, and thus only predicates were the sources of control-dependence edges. Later work [1, 3] extended slicing to handle programs with jumps (*goto*, *break*, *return*, etc). Since jumps also affect the flow of control, it made sense to let jumps (as well as predicates) be the sources of control-dependence edges. (Note that since a jump does not set the value of any variable, it cannot be the source of a data-dependence edge, and thus would not be in the slice from any component other than itself if it were not the source of a control-dependence edge. This is clearly not desirable in most contexts in which slicing is used. For example, if a statement S fails to execute because a procedure returns prematurely, and the programmer looks at the slice from S to try to understand why it did not execute, it is vital for the *return* statement to be in that slice.)

Now we are considering how to extend slicing to handle exceptions. Since a *throw* statement also affects flow of control, we feel that it is reasonable to let *throw* statements (like predicates and jumps) be the sources of control-dependence edges, and thus for a *throw* statement to be in the slice from a component that only executes if that exception is thrown, or only executes if that exception is *not* thrown (just as a jump statement is in the slice both from a component that only executes if the jump is taken, and from a component that only fails to execute if the jump is taken).

To illustrate the goals of slicing programs that include *try/catch/throw*, consider the two examples shown in Figure 1. In both examples, method f calls method g, which may throw *Ex1*, depending on the value of y.

First we examine the issues of control dependence discussed above. In Figure 1(a), the print statement at line 8 only executes if *Ex1* is thrown. That exception is thrown at line 14, in method *g*. Therefore, the slice from the print statement at line 8 must include the call to *g* at line 4, the throw of *Ex1* at line 14, and the *if* statement at line 13 on which the throw is predicated. The correct slice is indicated by plus signs in the margin in Figure 1(a).

The same components must be included in the slice from a statement that executes only if *Ex1* is *not* thrown; e.g., the print statement at line 5 in Figure 1(a). Therefore, the slice from that print statement must include the call to *g* as well as the *throw* statement and the *if* statement in *g*. The correct slice from the print statement at line 5 is indicated in Figure 1(a) using stars.

Next we examine how new kinds of data dependences can be introduced by exceptions. After a method call, a variable may have different values, depending on whether the method returned normally or because an exception was thrown. This must be reflected in a slice that includes a use of any such variable.

Consider slicing back from the statement on line 8 in Figure 1(b), which prints the value of *x*. This statement is only executed when *Ex1* is thrown at line 15; in that case, line 16 ($x = 1$) is not executed, and so only the definition of *x* at line 13 can reach the print statement. Conversely, the slice from the print of *x* at line 5 should only include the assignment to *x* at line 16, not the assignment at line 13. Line 5 is after the call to *g* in the *try* block, and will only execute if *g* returns without throwing an exception. Finally, the slice from the print of *x* at line 10 should include both assignments to *x*, since that print statement executes regardless of whether an exception is thrown or not. The three slices are indicated in Figure 1(b) using plus signs, stars, and x's, respectively.

The remainder of this paper is organized as follows: Section 2 gives background on system-dependence graphs and interprocedural slicing. Section 3 presents our extensions to interprocedural slicing to handle Java's *try/catch/throw* constructs. Section 4 discusses related work. Section 5 summarizes the contributions of this paper.

2. BACKGROUND

Slicing was originally defined by Weiser [18] for a simple, structured language as the solution to a dataflow problem specified using the program's control-flow graph (CFG). Ottenstein and Ottenstein [12] provided a more efficient algorithm for intraprocedural slicing (slicing a program that consists of just one procedure with no calls) that uses the program-dependence graph (PDG) [6]. Horwitz et al [7] extended the Ottenstein's algorithm to an interprocedural version that uses a collection of program-dependence graphs called the system-dependence graph (SDG). Although that algorithm was designed for procedural languages like C, the ideas apply to a subset of Java that excludes threads and exceptions. Later work by Ball/Horwitz [1] and Choi/Ferrante [3] extended the algorithm to handle jumps (*goto*, *break*, *return*, etc). The steps of the (extended) algorithm for building an SDG are given below. Figure 2 gives an exception-free version of the code from Figure 1(b), which is used to illustrate Steps 1–3; Figure 3 is used to illustrate Steps 4–6.

Step 1 (do static analysis): Do pointer analysis and use the results to determine the set of locations that may be used and defined by each statement or predicate in the program, as well as the set of methods that may be invoked when a call is made via a pointer. Using that information, compute *may-mod* and *may-use* sets for each method (the sets of non-local variables that might be modified and might be used, directly or transitively, by each method).

```
static int x, y;

static void f() {
    g();
    print(x);
}

static void g() {
    x = 0;
    if (y > 10) return;
    x = 1;
}
```

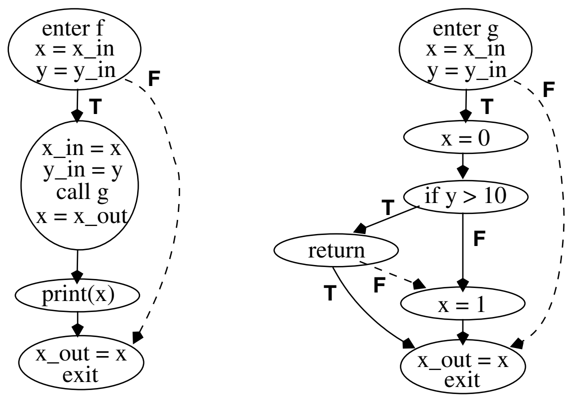


Figure 2: Example code and the corresponding CFGs. Non-executable edges are shown using dashed arrows.

Step 2 (build CFGs): Build a CFG for each method in the program. Each CFG starts with an *enter* vertex and ends with an *exit* vertex. There is one vertex for each statement and predicate. Each non-predicate vertex has one (unlabeled) outgoing edge. Each predicate vertex has two outgoing edges labeled *true* and *false*. The enter vertex and the vertices that represent jumps are treated as pseudo-predicates (predicates whose outgoing *false* edges are non-executable) in order to induce the correct control dependences. The enter vertex has an outgoing *true* edge to the vertex that represents the first statement in the method, and an outgoing *false* edge to the exit vertex. A jump vertex has an outgoing *true* edge to the target of the jump, and an outgoing *false* edge to the statement that would execute if the jump were a no-op.

Example: In Figure 2, the CFG for method *g* includes a jump (*return*) vertex. Its outgoing non-executable edge is shown using a dashed arrow. □

The vertex that represents a call to a method *M* also represents transferring to *M* the actual parameters plus all of the variables in *M*'s *may-mod* and *may-*

```
      ( 1) static int y;

++ ** ( 2) static void f() {
++ ** ( 3)   try {
++ ** ( 4)     g();
   ** ( 5)     print("no ex");
++ ** ( 6)   }
++    ( 7)   catch (Ex1 e) {
++    ( 8)     print("ex1");
++    ( 9)   }
      (10)   print("done");
++ ** (11) }

++ ** (12) static void g() throws Ex1 {
++ ** (13)   if (y > 10)
++ ** (14)     throw new Ex1();
      (15)   print("end g");
++ ** (16) }
```

(a)

```
         ( 1) static int x, y;

++ ** xx ( 2) static void f() {
++ ** xx ( 3)   try {
++ ** xx ( 4)     g();
   **    ( 5)     print(x);
++ ** xx ( 6)   }
++       ( 7)   catch (Ex1 e) {
++       ( 8)     print(x);
++       ( 9)   }
      xx (10)   print(x);
++ ** xx (11) }

++ ** xx (12) static void g() throws Ex1 {
++    xx (13)   x = 0;
++ ** xx (14)   if (y > 10)
++ ** xx (15)     throw new Ex1();
   ** xx (16)   x = 1;
++ ** xx (17) }
```

(b)

Figure 1: Two Java examples. In the left example, the slice from the print statement on line 8 is indicated with plus signs, and the slice from the print statement on line 5 is indicated with stars. In the right example, the slice from the print statement on line 8 is indicated with plus signs, the slice from the print statement on line 5 is indicated with stars, and the slice from the print statement on line 10 is indicated with x's.

use sets (the non-local variables that may be used or defined in M), and receiving back all of the variables in M's *may-mod* set. M's enter vertex represents receiving the values transfered in, and M's exit vertex represents transferring back the possibly modified values. This process is made explicit in the CFGs shown in this paper by labeling each call, enter, and exit vertex with a set of assignments:

- For a call vertex, the assignments are of the form $v\_in = v$ (one for each value transferred into the called method), and $v = v\_out$ (one for each value transferred back).
- For an enter vertex the assignments are of the form $v = v\_in$ (one for each value transferred in).
- For an exit vertex the assignments are of the form $v\_out = v$ (one for each value transferred back).

Example: In the code in Figure 2, field y is used (directly) in method g, and thus is in both f's and g's *may-use* sets (since f calls g); field x is used in method f, and is modified in g and thus is in f's *may-use* set and is in both f's and g's *may-mod* sets. Therefore, f's enter vertex includes assignments to receive the values of x and y; the call to g includes assignments to transfer the values of x and y, and to receive back the final value of x; g's enter vertex includes assignments to receive the values of x and y; both f's and g's exit vertices include assignments to transfer back the final value of x.[1] □

Step 3 (compute dependences): Compute each CFG's control and data dependences. Include non-executable edges when computing control dependences, but not when computing data dependences.

Vertex n is *control dependent* on vertex m iff n post-dominates one but not all of m's CFG successors.

Example: In the CFG for method f, all of the vertices except the exit vertex are control dependent on the enter vertex; in the CFG for method g, the return vertex is control dependent on the *if*, and the second assignment to x (*x=1*) is control dependent both on the *if* and on the return. The first assignment to x and the *if* are both control dependent on the enter vertex. □

Vertex n is *data dependent* on vertex m iff m (may) define a variable x, n (may) use x, and there is an x-definition-free path in the CFG from m to n.

Example: In f, the call vertex is data dependent on the enter vertex, and both the print vertex and the exit vertex are data dependent on the call. In g, the *if* vertex is data dependent on the enter vertex, and the exit vertex is data dependent on the two assignments to x. □

Step 4 (build a PDG for each CFG): The vertices of the PDG are the same as the vertices of the CFG except that the exit vertex is omitted, and the transfer of parameters and non-local variables from a method call to the called method and back (represented in the CFG by the sets of assignments used to label the call,

[1]In this example, field x *must* be modified in g, so there is actually no point in transferring its value into f or into g. However, since computing must-modify information is NP-hard in the presence of aliasing [10], the analysis of Step 1 only computes may-modify information. A variable that only *may* be modified by a method may also be unchanged; i.e., its final value may be the same as its initial value, and so the initial value must be transferred in so that it can be transferred back out.

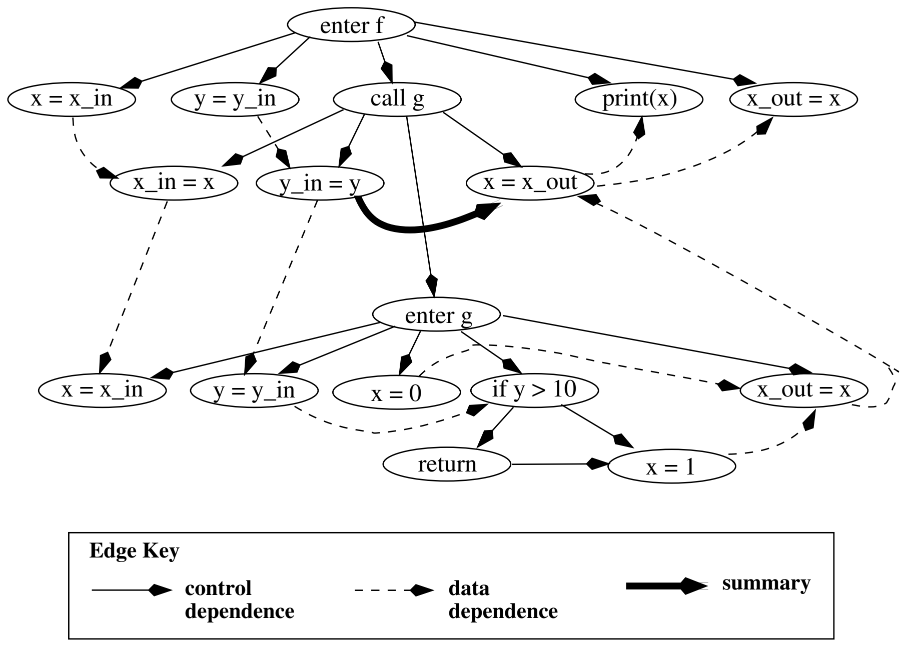


Figure 3: SDG for the code of Figure 2: The PDGs for f and g plus interprocedural and summary edges.

enter, and exit vertices) are represented in the PDG by four new kinds of vertices:

1. Each method includes one *formal-in* vertex $v = v\_in$ for each formal parameter and each variable in its *may-use* or *may-mod* sets.
2. Each method includes one *formal-out* vertex $v\_out = v$ for each variable in its *may-mod* set.
3. For each method call, there is one *actual-in* vertex $v\_in = v$ for each actual parameter and each variable in the called method's *may-use* or *may-mod* sets.
4. For each method call, there is one *actual-out* vertex $v = v\_out$ for each variable in the called method's *may-mod* set.

The edges of the PDG represent the data and control dependences computed in Step 3. All of the formal-in and formal-out vertices for a method are considered to be control dependent on the method's enter vertex, and for each call, all of the actual-in and actual-out vertices associated with the call are considered to be control dependent on the call vertex.

Example: The PDGs for methods f and g are shown in Figure 3. □

Step 5 (connect the PDGs to form the SDG): For each call to a method M, add a control-dependence edge from the call vertex to M's enter vertex. Add a data-dependence edge from each actual-in vertex associated with the call to the corresponding formal-in vertex in M, and from each formal-out vertex in M to the corresponding actual-out vertex associated with the call.

Example: Figure 3 shows the SDG for the code in Figure 2. □

Step 6 (add summary edges): For each pair of vertices (f_1, f_2) such that f_1 is a formal-in vertex and f_2 is a formal-out vertex of the same method, use the algorithm of [14] to determine whether there is an interprocedurally-valid path in the SDG from f_1 to f_2. If yes, for all corresponding actual-in/actual-out vertex-pairs (a_1, a_2), add the summary edge $a_1 \rightarrow a_2$.

Example: The SDG in Figure 3 has one summary edge from $y\_in = y$ to $x = x\_out$; this is because there is an interprocedurally-valid path between the corresponding formal-in/out vertices $y = y\_in$ and $x\_out = x$ in method g. The summary edge indicates that the value of y before the call to g can affect the value of x after the call. □

Once the SDG has been built, the slice from statement (or predicate) S can be computed in linear time in two passes that find all of the vertices from which there is a valid interprocedural path to S in the SDG. Pass 1 starts from the SDG vertex that represents S and follows edges backward in the SDG, ignoring interprocedural edges that run from a called method to the calling method. Pass 2 starts from the vertices reached in pass 1; it follows edges backward in the SDG, ignoring interprocedural edges that run from a calling method to the called method. The components of the slice are all of the vertices reached in pass 1 or pass 2.

3. EXTENDING INTERPROCEDURAL SLICING TO HANDLE TRY/CATCH/THROW

In this section we describe how to extend the interprocedural-slicing algorithm described above to handle

Java's *try/catch/throw* constructs. To simplify the presentation, we start by assuming there are no *finally* clauses and no unchecked exceptions; those are dealt with in Sections 3.3 and 3.4. Our approach involves defining how to represent *try/catch/throw* in the program's CFG and SDG; once the SDG is built, a slice can be computed using the two-pass approach described above.

The following two subsections describe how to extend the CFG and SDG so that the new control and data dependences introduced by *try/catch/throw* are handled correctly. Section 3.1 deals with control dependences and Section 3.2 deals with data dependences.

3.1 Handling control dependences

As discussed in the Introduction, a program component C may or may not execute depending on whether or not a *throw* statement T executes; i.e., T affects whether and/or how often C executes. This means that C should be (transitively) control dependent on T; i.e., there should be a path consisting of control-dependence edges in the SDG from T to C.

Consider the print statement on line 15 in Figure 1(a). That statement executes only if *Ex1* is not thrown, and thus it should be control dependent on the throw statement. This means that throw statements must be represented as pseudo-predicates in the CFG so that they induce control dependences.

Now consider the print statement on line 5 and the catch on line 7. The print executes only if *Ex1* is not thrown as a result of calling g, while the catch executes only if *Ex1* is thrown as a result of the call. Therefore, both the print and the catch should be control dependent on the call (as well as being transitively control dependent on the throw). This means that the call to g must be represented as a predicate so that it induces control dependences. It also means that there must be an interprocedural control-dependence path from the *throw* statement in g to the print statement and the catch in f. Note that this requires something quite new: in the SDGs defined in [7], there can be a control-dependence path from T to C only if vertices T and C are in the same method, or if T's method (transitively) calls C's method. To handle the control-dependence effects of a *throw* T, we now require a control-dependence path from T to C in some situations where C's method (transitively) calls T's method.

Finally, consider the print statement on line 8. That statement executes if the call to g causes *Ex1* to be thrown, only because the print is inside the catch of *Ex1*, and the call to g is inside the corresponding try. This means that both try and catch constructs must also be represented as pseudo-predicates.

Putting this all together, we define below how to build a CFG for code that includes try/catch/throw. Note that this takes into account the issues of control dependence discussed above but not yet the issues of data dependence.

Modifications to Step 2 of the SDG-Building Algorithm of Section 2 for try/catch/throw:

- For each method that might throw an exception, the method's CFG includes one *normal-exit* vertex and a set of *exceptional-exit* vertices, in addition to the usual exit vertex. There is one exceptional-exit vertex for each type of exception that may be thrown by the method. The normal-exit vertex follows the last statement in the method. The method's (original) exit vertex is the successor of both the normal- and exceptional-exit vertices.
- Each *try*, *catch*, and *throw* is represented by a pseudo-predicate vertex (recall that a pseudo-predicate has an executable outgoing *true* edge, and a non-executable outgoing *false* edge). The outgoing *true* edges of the try and catch vertices go to the first statements inside the try/catch block, and their outgoing *false* edges go to the first statement that follows the last catch block. The outgoing *true* edge of a throw goes to the corresponding exceptional-exit vertex, and the outgoing *false* edge goes to the statement that would follow the throw if it were a no-op.
- Each call vertex that represents a call to a method that might throw an exception is represented as a predicate with two kinds of outgoing edges (both executable):
  1. An edge to a new pseudo-predicate *normal-return* vertex. The outgoing *true* edge of the normal-return vertex goes to the first statement that executes when the call returns normally, or to the normal-exit vertex if the call is the last statement in the method. The outgoing *false* edge of the normal-return vertex goes to the first statement that executes whether the call returns normally or causes an exception to be thrown, or to the exit vertex if there is no such statement.
  2. For each possible exception type that might be thrown by the call, an edge to the corresponding catch vertex (if there is one), or to the corresponding exceptional-exit vertex at the end of the current method's CFG, otherwise.

Example: Figure 4 shows the CFGs for the code in Figure 1(a). □

The modifications to the design of the CFG listed above ensure that there will be a control-dependence path in the PDG from a throw to the corresponding exceptional-exit, and from a call to a method that might throw an exception to all vertices that represent program components whose execution may be affected by whether or not an exception is thrown (i.e., components in catch clauses, and components that follow the exception-throwing call). What remains is to define the appropriate modifications to the SDG:

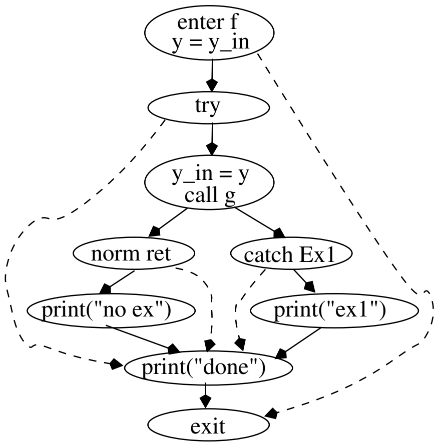
- We must define new interprocedural SDG edges, so that there is a control-dependence path in the SDG from a throw to all of the vertices whose execution it may affect.
- For calls to exception-throwing methods, we must also define the appropriate new kinds of summary edges to take into account the fact that the value of a parameter or non-local variable before the call may affect whether the method returns normally or throws an exception.

The new interprocedural edges are as follows; for each call to a method that may throw an exception:

- The normal-exit vertex in the called method is connected to the normal-return successor of the call vertex.

```
for each exception-throwing method $M$:
    for each formal-in vertex $v$:
        for each normal-exit and exceptional-exit vertex $w$:
            if there is an interprocedurally-valid path in the SDG from $v$ to $w$ (determined using the
               algorithm of [14]) then
                for each call to $M$:
                    add a summary edge from the actual-in vertex that corresponds to $v$ to the normal-return,
                    catch or exceptional-exit vertex that corresponds to $w$.
```

Figure 5: The algorithm for computing the new summary edges that represent the fact that the initial value of a parameter or non-local variable can affect whether a method returns normally or throws an exception.



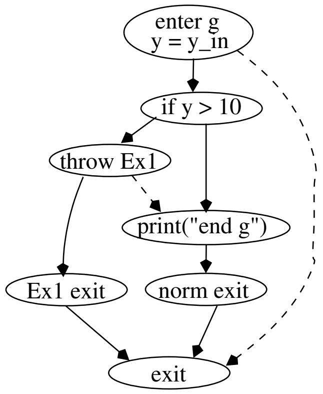


Figure 4: The CFGs for the code in Figure 1(a). (Dashed arrows represent non-executable edges.)

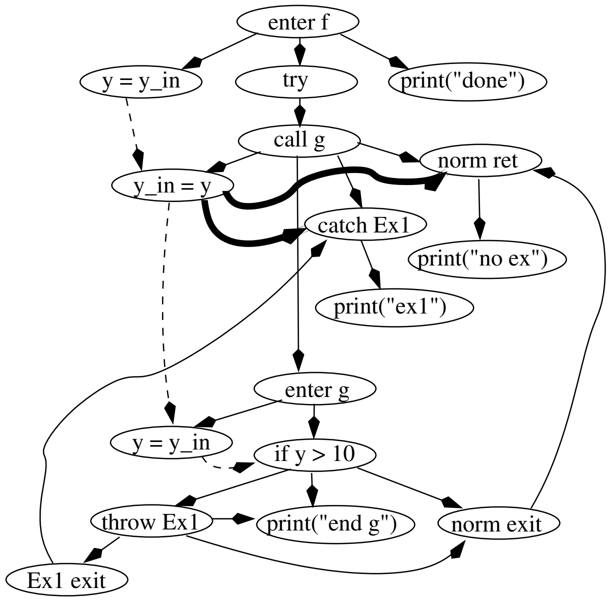


Figure 6: The SDG for the code in Figure 1(a). (See Figure 3 for the edge key.)

- Each exceptional-exit vertex in the called method is connected to the corresponding catch or exceptional-exit successor of the call vertex.

The algorithm for computing the new summary edges is given in Figure 5.

Example: Figure 6 shows the SDG for the code in Figure 1(a). Note that there are summary edges from the actual-in vertex $y\_in = y$ for the call to g to the normal-return and catch vertices associated with that call. This is because there are paths in g's PDG from $y = y\_in$ to both *normal exit* and *Ex1 exit*. The summary edges indicate that the value of y before the call determines whether g returns normally or throws *Ex1*. □

3.2 Handling data dependences

As discussed in the Introduction, the slices from each of the three print statements in the code in Figure 1(b) should include a different subset of the assignments to x: the slice from the print at line 5 should include only the assignment to x at line 16; the slice from the print at line 8 should include only the assignment to x at line 13, and the slice from the print at line 10 should include both assignments to x. However, if we compute those slices using an SDG that includes only one formal-out vertex for variable x we would (erroneously) include both assignments to x in all three slices, because there would be interprocedurally valid paths in the SDG from both assignments to all three print statements.

To solve this problem, we include formal- and actual-out vertices explicitly in the CFG, and we use multiple sets of formal- and actual-out vertices for methods that can throw exceptions, and for calls to those methods. In particular, for each method that might throw an exception, we associate a set of formal-out vertices with each of the method's exceptional-exit vertices, and with its normal-exit vertex.

Similarly, for each call to a method that might throw an exception, we associate a set of actual-out vertices with each of the call vertex's successors.

Example: The CFG for the code in Figure 1(b) is shown in Figure 7. Note that the normal-exit and exceptional-exit vertices are now pseudo-predicates, and that the call to g is no longer considered to be a definition of x—the (new) actual-out vertices for x now play that role. In this CFG only the assignment $x = 0$ at line 13 reaches the formal-out vertex associated with the exceptional-exit vertex, and only the assignment to $x = 1$ at line 16 reaches the formal-out vertex associated with the normal-exit vertex. □

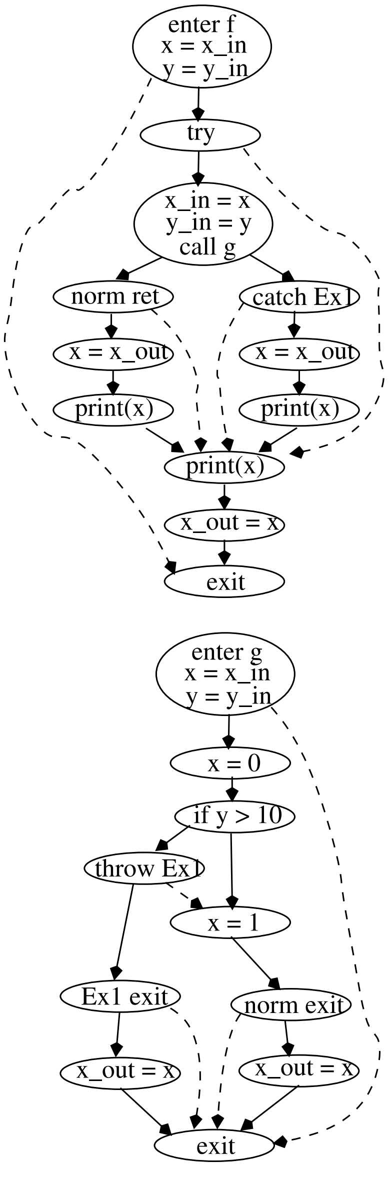


Figure 7: CFGs for the code in Figure 1(b).

When constructing the SDG, the formal-out vertices are connected by interprocedural data-dependence edges to the actual-out vertices associated with the corresponding exception type.

Example: The SDG constructed from the CFGs of Figure 7 is shown in Figure 8. Using this SDG, the slices from the print statements on lines 5 and 8 will each (correctly) include only one assignment to x, while the slice from the print statement on line 10 will include both assignments to x. □

3.3 Handling finally clauses

In Java, a *finally* clause can follow the last catch block associated with a try. The code in a finally block *always* executes, whether or not an exception is thrown. Thus, the code in a finally block can execute after the last statement in the try block (if no exception is thrown), or after the last statement in a catch block (if an exception is thrown and caught), or after a statement in a try or catch block that causes an uncaught exception to be thrown. If the try or catch blocks include non-local transfer of control (e.g., a break, continue, or return statement), the code in the finally block executes before control is transferred. Furthermore, any non-local transfer of control in the finally block takes precedence over non-local transfer of control in the try or catch blocks; i.e., if there is a transfer of control in both places, it is the one in the finally block that is actually executed. Therefore, the following four cases can arise:

1. There are no non-local transfers of control in the try, catch, or finally blocks.
2. The try and associated catch blocks do not contain non-local transfers of control, but the finally block does.
3. The try and associated catch blocks contain non-local transfers of control, but the finally block does not.
4. The try and associated catch blocks contain non-local transfers of control, and so does the finally block.

We discuss how to represent flow of control (in the CFG) for each of the four cases.

Cases 1 and 2: Cases 1 and 2 can be handled in a straightforward manner:

- The finally itself is represented as a pseudo-predicate, with an outgoing *true* edge to the vertex that represents the first statement in the finally block, and an outgoing *false* edge to the vertex that represents the first statement after the finally block.
- There is an edge from the last vertex in each try/catch block to the vertex that represents the finally.
- The finally vertex is also the target of the non-executable *false* edges out of the catch vertices.

Example: The CFG for these two cases is shown schematically in Figure 9 for a try block that has some statements, then a call to a method that might throw exceptions *Ex1* to *ExN*, then some more statements (represented in the figure by the box below the *normal-return* vertex). The finally block itself may include non-local transfer of control (for case 2); this is shown in the figure as a jump inside the box that represents the finally block. □

Case 3: Case 3 is illustrated by the example code in Figure 10 (it is assumed that g may throw exception *Ex1*, and that the finally block includes no non-local transfer of control).

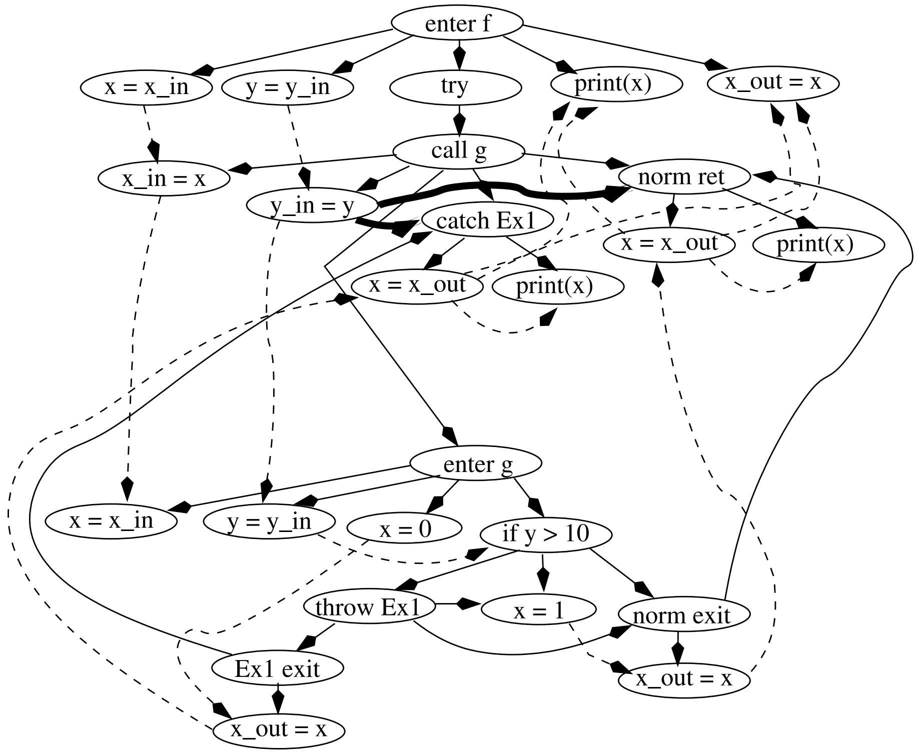


Figure 8: SDG for the CFGs of Figure 7. (See Figure 3 for the edge key.)

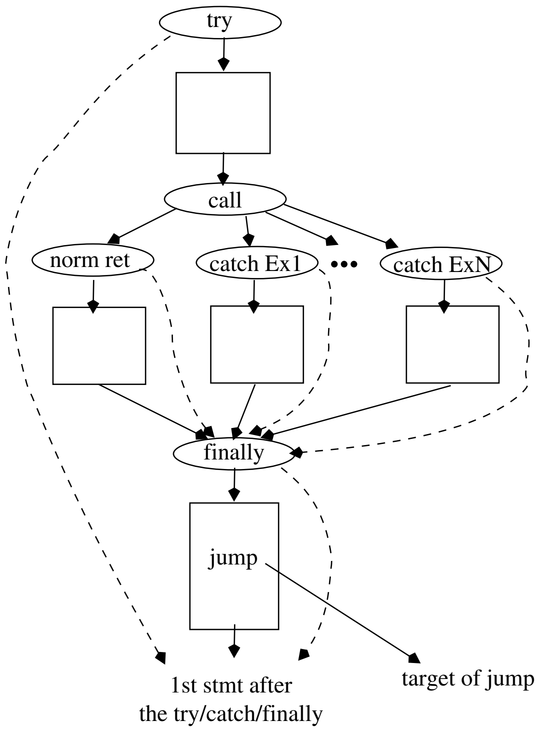
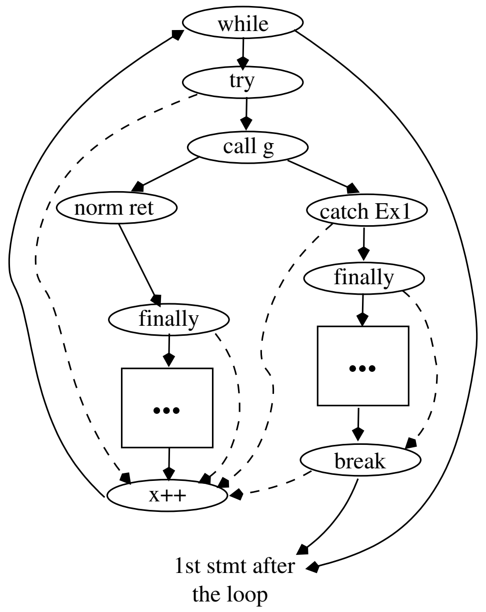


Figure 9: Handling finally clauses: The CFG for cases 1 and 2.

```
(1)  x = 0;
(2)  while ( x < 10 ) {
(3)     try {
(4)        g();    // may throw Ex1
(5)     }
(6)     catch (Ex1 e) { break; }
(7)     finally { ... }
(8)     x++;
(9)  }
```

Figure 10: Handling finally clauses: Code to illustrate Case 3.

In this example, if the exception is thrown, control flows from the end of the finally block to the break in the catch clause, while if the exception is not thrown, control flows from the end of the finally block to the increment of x. Thus, if we construct the CFG using the technique described above for cases 1 and 2, we would need multiple outgoing edges from the end of the finally block: one to the break statement on line 6, and one to the increment of x on line 8. This would introduce invalid paths into the CFG; for example, there would be a path from the end of the try block to the break statement (as if the break could be taken when no exception is thrown). While this would not lead to incorrect slices, it could lead to slices that include irrelevant components, and thus we propose two better ways to define the CFG in this case.



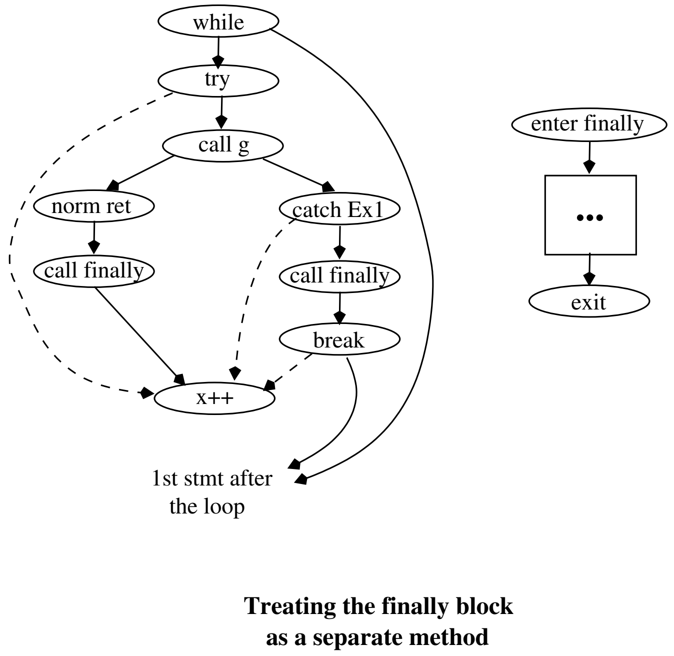


Figure 11: The CFGs for the code in Figure 10 using inlining and treating the finally block as a method.

The first possibility is to inline the finally clause as necessary:

- For each CFG vertex v that is part of a try/catch block and represents a non-local transfer of control, a copy of the entire finally clause is inserted between v's predecessor(s) and v itself.
- For each call in a try/catch block to a method that might throw an uncaught exception, a copy of the entire finally clause is inserted between the call vertex and the exceptional-exit vertex.

Although this approach can cause space blow-up, it is unlikely to do so in practice, since it is unlikely that non-local transfers of control out of try/catch blocks are frequently used in combination with finally blocks.

A second possibility is to treat the finally clause as a method: create a separate CFG for each finally clause, and insert a call vertex in the try/catch blocks wherever control would be transferred to the finally block (i.e., before each non-local transfer of control, between each call-vertex / exceptional-exit-vertex pair, and after the last statement in a try/catch block).

Example: Figure 11 illustrates these two approaches, showing the CFGs for the code in Figure 10. □

Case 4: Essentially the same two options used for case 3 can also be used to handle case 4 (non-local transfers of control in the try and/or catch blocks, as well as in the finally block). However, while the inlining approach can be used exactly as described above, treating the finally block as a method requires some special handling. The problem is that the finally block will be represented by a new method, but the targets of its non-local transfers of control will be in the original method. To handle this, each non-local transfer of control in the finally block must be represented by a new, special kind of exit vertex (e.g., a break would be represented by a *break-exit* vertex, and a continue would be represented by a *continue-exit* vertex) analogous to the special exceptional-exit vertices introduced in Section 3.1. Corresponding special return vertices (e.g., *break-return* and *continue-return*) would be used on the calling side, and the targets of those return vertices would be the original targets of the non-local transfers of control.

3.4 Handling unchecked exceptions

Java includes a large set of *unchecked* exceptions; for example: *ArithmeticException*, *ClassCastException*, *IndexOutOfBoundsException*. These exceptions can be thrown by many common expressions (arithmetic operations, casts, array or string indexing), and it is not required that such expressions be enclosed in a try block, or that the enclosing method list unchecked exceptions in its throws clause.

How unchecked exceptions should be treated in the context of slicing depends on the intended application. This question is related to the question of whether or not a statement S that follows a loop should be control dependent on the loop predicate, since S only executes if the loop eventually terminates. That issue was explored in [13, 2], and led to the definition of two kinds of control dependence: weak and strong. Similar notions may be useful in handling unchecked exceptions.

The most conservative approach would be to treat every program component C that might cause an unchecked exception to be thrown as a conditional throw statement; i.e., in the CFG, add an (executable) edge from C to the appropriate exceptional-exit vertex. However, this would make all components that follow C control dependent on it, and so slices computed using this approach would probably be so large as to be worthless for most applications.

Completely ignoring unchecked exceptions, is, however,

```
      ( 1) static int y;

++    ( 2) static void a() {
++    ( 3)   try {
++    ( 4)     b();
++    ( 5)   }
++ ** ( 6)   catch (Ex1 e) {
++ ** ( 7)     print("ex1");
++ ** ( 8)   }
++    ( 9) }

++    (10) static void b() throws Ex1 {
++    (11)   if (y > 10)
++    (12)     c();
++    (13) }

++ ** (14) static void c() throws Ex1 {
++    (15)   throw new Ex1();
++ ** (16) }
```

Figure 12: Example code. The correct slice from line 7 is indicated by plus signs, and the slice computed by the Sinha algorithm is indicated by stars.

not appropriate for code that includes catch clauses for them. In that case, the slice from code in the catch clause should include any expression that might cause the exception to be thrown. Handling these cases requires a (straightforward) analysis to determine, for each try block with an associated catch of an unchecked exception, which expressions in the (dynamic) scope of the try could cause the exception to be thrown; those expressions should be treated as conditional throw statements as described above.

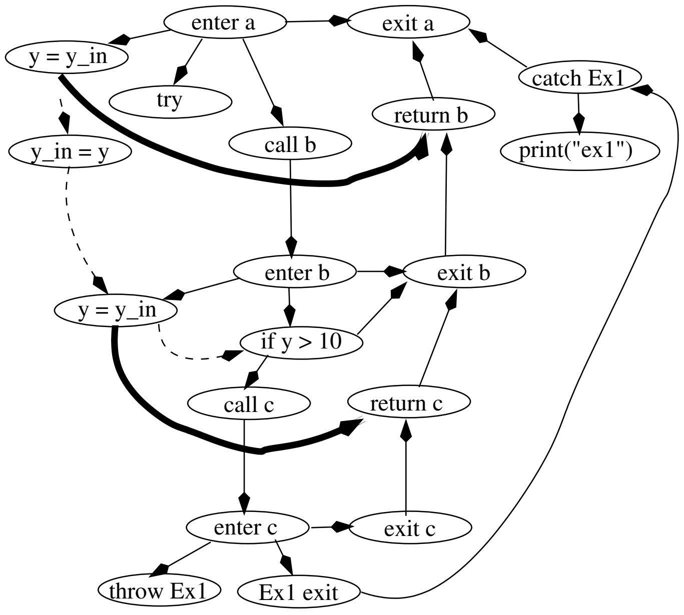
4. RELATED WORK

Sinha and Harrold previously addressed how to represent flow of control for try/catch/finally blocks in [15, 16], and Sinha, Harrold, and Rothermel addressed slicing programs with try/catch/throw in [17]. While some aspects of their approach are similar to ours, there are several omissions and errors that we have corrected.

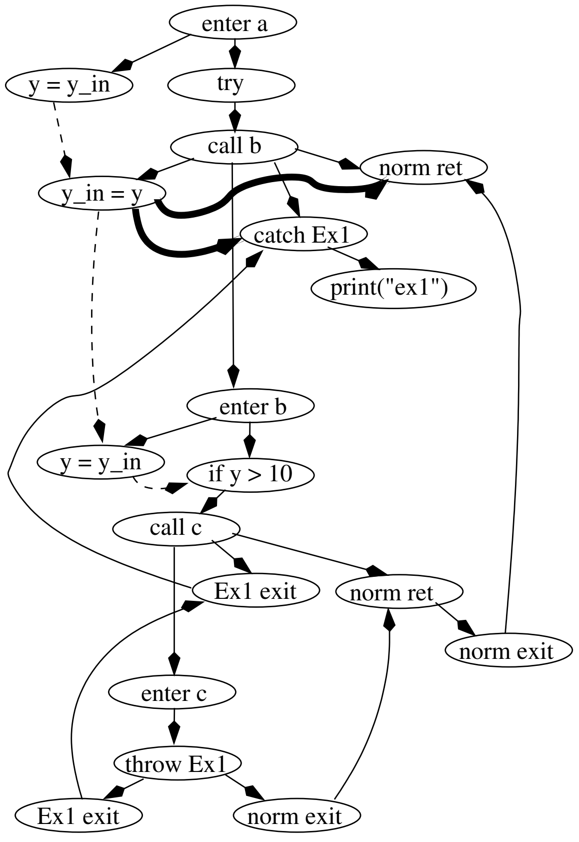
Handling finally clauses by inlining and by treating finally blocks as methods are both proposed in [15, 16]; however, they did not consider the simpler approach discussed in Section 3.3 for cases 1 and 2 (no non-local transfer of control, or such transfers only in the finally block). Also, they did not recognize the extra complications that arise in case 4 when the finally block is treated as a method.

We turn now to a comparison of our slicing algorithm with that of [17]. One problem with [17] is that try/catch/throw vertices are not treated as pseudo-predicates in the CFG, which means that they induce no control dependences, and thus are never included in slices. A more serious problem is that interprocedural control dependences are not represented correctly when the length of the call chain from a *try* to a *throw* is greater than one. These problems are illustrated by Figures 12 and 13.

Figure 12 gives example code; the correct slice from the print statement at line 7 is indicated by plus signs, and the slice computed by the Sinha algorithm is indicated by stars. Note that the Sinha slice includes no code from method



(a) The SDG built using Sinha et al's approach.



(b) The SDG built using our approach.

Figure 13: The SDGs built for the code of Figure 12

b, even though the *if* and the call to *c* there clearly control whether *Ex1* is thrown, and therefore whether the print statement executes. The reason for this problem is that the SDG defined by the Sinha algorithm (shown in Figure 13(a)) connects the exceptional-exit vertex in method *c* (where *Ex1* is thrown) directly to the catch vertex in method *a* (where the exception is caught). Thus, the first pass of slicing (which ignores interprocedural edges from called methods to call sites) reaches only vertices in method *a*, and the second pass (which ignores interprocedural edges from call sites to the called method) reaches only vertices in method *c*. In contrast, the SDG defined by the algorithm presented in this paper (shown in Figure 13(b)) "threads" the control dependences up the call chain, and so the appropriate vertices in methods *b* and *c* are reached during the second pass of slicing.

Another serious problem is that Sinha et al do not consider data dependences in the presence of exceptions at all. Using their approach, a slice from a statement in a catch block that uses a variable *v* whose value was set outside the catch block will not include *any* assignments to *v*. Furthermore, they only use one set of actual-out vertices, and thus do not account for the fact that different data dependences may be induced depending on whether a method exits normally, or through an exception.

5. CONCLUSIONS

Slicing is an important operation, and extending slicing algorithms to handle Java programs is an area of current interest [9, 8, 5, 4, 17, 11]. In this paper we have described how to extend the slicing algorithm of [7] (which uses SDGs) to handle try/catch/throw. In particular, we have defined how to represent try/catch/throw in the control-flow graphs from which SDGs are built so that correct control and data dependences are computed, as well as extending the set of interprocedural edges in the SDG itself.

6. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435.

7. REFERENCES

[1] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Lecture Notes in Computer Science*, volume 749, New York, NY, November 1993. Springer-Verlag.

[2] G. Bilardi and K. Pingali. A framework for generalized control dependence. In *Proc. of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 291–300. ACM Press, 1996.

[3] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[4] J. Hatcliff et al. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symp.*, 1999.

[5] M. Dwyer et al. Slicing multi-threaded Java programs: A case study. Technical Report 99-7, Kansas State University Computing and Information Sciences, 1999.

[6] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, January 1990.

[8] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Tools and Software Eng.*, June 1998.

[9] D. Liang and M. Harrold. Slicing objects using system dependence graphs. In *Proc. of the IEEE Int. Conf. on Software Maintenance*, pages 348–357, November 1998.

[10] E. Myers. A precise interprocedural data flow algorithm. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 219–230, 1981.

[11] M. Nanda and S. Ramesh. Slicing concurrent programs. In *Proc. Int. Symp. on Software Testing and Analysis*, August 2000.

[12] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 177–184, 1984.

[13] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. on Software Engineering*, 16(9):965–979, September 1990.

[14] T. Reps, S. Horwitz, and G. Rosay. Speeding up slicing. In *Proc. ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 11–20, December 1994.

[15] S. Sinha and M. Harrold. Analysis of programs with exception-handling constructs. In *ICSM*, pages 348–357, 1998.

[16] S. Sinha and M. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Trans. on Software Engineering*, 26(9):849–871, 2000.

[17] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Int. Conf. on Software Eng.*, pages 432–441, May 1999.

[18] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.