1992

Rigorous Data Flow Testing through Output Influences†

Evelyn Duesterwald Rajiv Gupta Mary Lou Soffa

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
duester@cs.pitt.edu (412) 624-8850

Abstract - This paper presents a refinement of existing data flow testing criteria through the notion of the output-influence in a program. Previous data flow testing criteria considered exercising a definition-use (def-use) pair in a successful test case as sufficient evidence of its correctness. We argue that the correctness is not demonstrated unless exercising the def-use pair has an influence on the computation of at least one correctly produced output value. We refine existing data flow criteria by requiring that an exercised def-use pair must be output-influencing to be considered tested by a test case. By incorporating the notion of output-influence into existing criteria, we present a more rigorous testing strategy while still relying on the efficiency of data flow analysis based approaches to testing. We have developed several techniques that are based on the concept of static and dynamic program slicing to efficiently compute the output-influencing def-use pairs in a test case. By utilizing the concept of a dynamic slice, we effectively determine the data flow coverage of programs with pointer and array references.

1. Introduction

Various data flow testing criteria have been developed to determine when a program is sufficiently tested. In data flow testing [5, 9, 14, 20], a variable assignment at a point in a program is tested by selecting test cases that execute subpaths from the assignment (i.e., definition) to points where the variable's value is used (i.e, use). Definition-use (def-use) pairs are computed using data flow analysis techniques, and data flow testing criteria are used to select a particular set of def-use pairs to test. A data flow criterion is satisfied if each selected def-use pair has been covered, i.e., tested, by at least one test case. One criterion, 'All-du' (all def-use pairs), for example, is satisfied when each def-use pair in a program has been tested. A critical issue for uncovering program errors during testing is the **effectiveness** of a testing criterion. The effectiveness of a criterion is measured as the likelihood that errors, if they exist, are revealed by the execution of paths selected under the criterion.

The effectiveness of existing data flow criteria is limited due to their common assumption that merely exercising def-use pairs provides sufficient evidence of their correctness. However, exercising a def-use pair does not necessarily demonstrate its correctness or incorrectness. Typically, an incorrect def-use pair is detected by the programmer if its erroneous effect is reflected in the computed output values. Thus, exercising def-use pairs does not aid in revealing program errors unless the exercised definitions have an influence on the computed output values. Consider, for example, the task of testing an

† Partially supported by the National Science Foundation under a Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh.

optimizing compiler. Optimizers analyze the input program and perform optimizations on program code based on the results of the analysis. The analysis portion of the optimizer is exercised in every test run independent of whether the optimizations are actually applied. However, an error in the analysis portion is unlikely to be revealed unless the error had some influence on an incorrect output (e.g., by the application of incorrect optimizations or the lack of applying correct ones). Thus, for a particular test case that produces correct output values, only those components of the program should be considered tested that have actually contributed to the correct output computation.

In this paper, we introduce the notion of **output-influencing (OI)** def-use pairs as a novel approach to more rigorous and effective data flow testing. Our approach is based on the notion that a program component to be considered tested must have demonstrated evidence of correctness through its contribution to the produced program output. Output-influence is used to refine previous data flow testing criteria by requiring that an exercised def-use pair influences the computation of at least one correct output value to be considered tested. We restrict the discussion in this paper to the refinement of the All-du criterion to an **output-influencing-All-du (OI-All-du)** criterion, although other existing data flow criteria can also be refined using output-influence.

Incorporating the concept of output-influence in a data flow criterion presents new challenges in determining the def-use pairs that have been **covered** by a test case. Determining data flow coverage for the conventional All-du criterion requires recording all def-use pairs in the program that are exercised. Our refined OI-All-du criterion, in addition, requires the determination as to whether an exercised def-use pair is also output-influencing. We utilize the concept of a **program slice** [25] to determine data flow coverage for the OI-All-du criterion. A program slice, first introduced by Weiser for debugging [25], is a subset of a program that contains the statements that may influence the value of a selected set of variables at selected program points. We use slicing to capture the output-influencing def-use pairs by computing **output-slices**, which are slices on the output statements in the program. By formulating the coverage problem as a slicing problem, we develop several techniques with varying costs and accuracies to determine the output-influencing def-use pairs in a test case.

Program slices can be computed statically over all executions of a program (static slice), or dynamically, providing information for a specific execution of a program (dynamic slice). A static output-slice provides a conservative compile-time approximation of the actual def-use pairs in a test case. The advantage of using static analysis is that the data flow information for a program is computed only once prior to execution. Although static data flow analysis is sufficiently accurate in its application in optimizing compilers, static information may be overly conservative in the presence of dynamic variable references, such as array and pointer references. To obtain accurate information for array and pointer variables, we compute dynamic output-slices. By using the same approach for both static and dynamic slicing, we are also able to combine the two slicing concepts. By incorporating both static and dynamic data flow information, we can move as much work as possible to compile-time and only collect dynamic information, where compile-time analysis is overly conservative, i.e., for array and pointer references.

The remainder of this paper is organized as follows. Section 2 introduces the OI-All-du criterion and discusses related work. Section 3 presents the dependence graph that we use to represent a program. Section 3 also defines our slicing algorithm that operates on the dependence graph. In Section 4, we demonstrate how we use the presented slicing concepts to develop several approaches with varying accuracies and costs to determine the data flow coverage for the OI-All-du criterion. Concluding remarks are given in Section 5.

2. Output Influence and Testing

Previous data flow criteria considered a def-use pair to be covered by a test case t if it was exercised in t, assuming that merely exercising a def-use pair provides sufficient evidence of its correctness. We refine previous data flow criteria by requiring that in order for a test case t to cover a def-use pair, the def-use pair should not only be exercised during execution of t, but also influence the computation of an output value produced by t. The output-influencing All-du (OI-All-du) criterion is defined as follows:

Definition: Let P be a program and T_P a test suite for P. T_P satisfies the **output-influencing-All-du (OI-All-du) criterion** if, and only if, for each def-use pair p in P there is some test case t in T_P such that, when P is executed on t, the pair p is exercised and exercising p directly or indirectly affects the computation of at least one correct output value produced by t.

We illustrate our refined criterion using the example shown in Fig. 1 (i). The program fragment computes the minimum and the sum over an input array a. However, we have introduced several errors that are documented in the comments in Fig. 1 (i). We consider the test case t_0: n=4 and a=(0,0,0,4) for which the fragment accidently produces the correct output values (i.e., the output is 0, 4). When executing the fragment on this input, all statements are executed, and in particular all def-use pairs inside the loop computation are exercised. Therefore, using the traditional All-du criterion leads to premature conclusions about the correctness of the def-use pairs inside the loop computation that actually contains errors.

```
(1)  input(n,a);
(2)  i:=2;
(3)  p:=1;
(4)  m:=a[p];
(5)  While i<n Do Begin        /* should correctly be i<=n  */
(6)    If a[p]<=a[1] Then      /* should correctly be a[p]<=a[i]  */
(7)      p:=i;
(8)    a[i]:=a[i]+a[i-1];
(9)    i:=i+1;
     EndWhile;
                               /* omitted: m:=a[p]  */
(10) output('min is',m);
(11) output('sum is',a[n]);
```

(i)

```
(1)  input(n,a,b);
(2)  i:=2;
(3)  p:=1;
(4)  m:=a[p];
(5)  While i<n Do Begin
(8)    a[i]:=a[i]+a[i-1];
(9)    i:=i+1;
     EndWhile;
(10) output('min is',m);
(11) output('sum is',a[n]);
```

(ii)

```
(1)  input(n,a,b);
(2)  i:=2;
(3)  p:=1;
(4)  m:=a[p];
(10) output('min is',m);
(11) output('sum is',a[n]);
```

(iii)

Fig. 1: Original program fragment (i), the static slice on the two output values in statements 10 and 11 (ii), and the dynamic slice for the two output values on input n=4, a=(0,0,0,4) (iii).

The new OI-All-du criterion enables a more rigorous approach to data flow testing. Def-use pairs that are exercised in a test case but have not demonstrated any evidence of their correctness (like the ones inside the loop computation in Fig. 1) are accordingly not considered covered. Thus, it is no longer sufficient to determine data flow coverage from the def-use pairs that were exercised during a test case. In addition, it must be determined which of the exercised def-use pairs were of output-influence. We utilize the concept of a program slice to determine the output-influencing def-use pairs in a test case.

For the purpose of capturing the output-influencing def-use pairs in a test case, we are interested in a particular class of slices, the **output-slices**. Output-slices are determined for the computed output values in a program and contain the statements that directly or indirectly affect the computation of output values. We use both static and dynamic output-slices to develop several techniques with varying costs and accuracies to determine the data flow coverage of a test case. Consider the example in Fig. 1. The static output-slice on the two output values in statements 10 and 11 is shown in Fig. 1 (ii). The slice does not contain statements 6 and 7 inside the loop and thus correctly indicates that no def-use pair involved in the execution of those statements should be covered by the test case.

Although using static slices eliminates a number of def-use pairs from coverage that do actually not contribute to the computed output values, the static nature of this approach may yield too conservative information in the presence of dynamic structures such as arrays and pointers. The static output-slice in Fig. 1 (ii), for example, does not express that the output value `a[n]` is actually not a result of the loop computation (due to the error in the loop condition). To obtain accurate information in the presence of arrays and pointers, we use a dynamic approach, since all static ambiguities of array and pointer accesses can be resolved at run-time. The dynamic output-slice for the two output values produced by our test case t_0 is shown in Fig. 1 (iii). The slice reveals that actually no def-use pair involved in the loop computation should be covered by the test case and therefore the errors in Fig. 1 (i) will not avoid detection during testing.

Two approaches to compute static slices have been proposed. The first approach, proposed by Weiser [25] computes a static slice by iteratively solving a set of data flow equations over the program's control flow graph representation [3]. The second approach uses a variation of the program dependence graph to represent a program [12,22]. The program dependence graph (PDG) [7,17] explicitly represents both control and data dependencies among the statements in a program. The computation of a static slice reduces to a simple vertex-reachability problem when using the PDG. There have also been two approaches to compute dynamic program slices. In Korel and Laski's approach [15] a complete execution trace is generated at run-time and a dynamic slice is computed by solving data flow equations over the generated trace. A different approach to compute dynamic slices based on dependence graphs has been presented in [1,2]. The notion of a dynamic slice used in this approach is slightly different from Korel and Laski's in that a dynamic slice is not necessarily an executable subset of the original program and thus may contain fewer statements. In the work by Agrawal, DeMillo and Spafford a dynamically expanded program dependence graph is defined based on the execution history of a program [2]. Agrawal and Horgan describe how a reduced dynamic expansion of the program dependence graph can be built at run-time [1]. Dynamic slices are determined after execution from the dynamic dependence graph or the reduced dynamic dependence graph, respectively. The authors also describe a dependence graph based approach that can be used to computed executable dynamic slices of the notion used by Korel and Laski. In this approach, the static program dependence graph is built first and data dependence edges are marked during execution as they arise. A dynamic slice is determined by considering only the marked portion of the graph.

We follow the dependence graph based approach for slicing and use the notion of a slice as an executable subset of a program. We use a variation of the program dependence graph for both static and dynamic slicing. However, our notion of a dynamic dependence graph differs from the one given in [1]. Existing dynamic dependence graphs [1, 19] form some dynamic expansion of a static program dependence graph and are capable of distinguishing dependencies that hold for different instances of a statements (for example, for statements inside a loop). Our primary interest, however, in using dynamic dependence information is to resolve compile-time ambiguities of array and pointer accesses. For this purpose we have adapted a slightly different definition of a dynamic dependence graph that does not represent an expansion but a dynamic subset of the program's static dependence graph with respect to a specific execution. We have developed an efficient dynamic slicing algorithm that does not require the generation of execution traces or run-time analysis. Instead of statically determining the data flow in a program, we collect the data flow information that is needed to construct dynamic slices on-the-fly as the program executes through a simple pointer mechanism.

By using dependence graphs for both static and dynamic slicing, we can efficiently combine the two slicing concepts. The resulting slices are based on both static and dynamic information. A hybrid between a static and a dynamic slice was first introduced by Venkatesh [24], the so called 'quasi-slice'. In a quasi-slice some input values are fixed while others may vary. Our notion of a hybrid slice is slightly different in that our hybrid is a static slice with respect to some variables and a dynamic slice with respect to others, independent of the input variables.

3. Dependence Graphs and Slicing

This section presents our approach for computing output-slices of a program. An output-slice is extracted from a program's dependence graph that represents the control and data dependencies among statements. Thus, the major portion of our slicing technique is the construction of the dependence graph. The control dependence information is derived from the program structure and we present three alternative methods for collecting data dependence information: statically over all executions, dynamically with respect to a specific execution, or in a combined static/dynamic fashion. The three methods are described in detail in the following sections. Using any of these methods, we obtain a dependence graph for the construction of output-slices. However, if the data dependence information in the graph was computed statically the resulting output-slices are static output-slices. Similarly, if dynamic (combined static/dynamic) data dependence information is used, the output-slices are dynamic (combined static/dynamic) output-slices.

3.1. The Dependence Graph

The dependence graph for a program represents two dependence relations among statements: control and data dependence. A statement s_1 is **control dependent** on a statement s_2, if s_2 is a control predicate and control reaches s_1 depending on the result of evaluating s_2. In structured programs the control dependences can be directly derived from the nesting structure of statements. The computation of control dependencies in arbitrary programs is described in [7]. **Data dependence** describes the def-use relationships in a program: static data dependencies describe the potential def-use pairs and dynamic data dependencies the actual def-use pairs for a specific input. Thus, dynamic data dependence is a subrelation of static data dependence. A statement s_1 is **statically data dependent** on a statement s_2 if s_2 contains a definition of a variable v and s_1 uses v and control *may* reach s_1 after s_2 without passing through a redefinition of v. A statement s_1 is **dynamically data dependent** on s_2 with respect to program input I if, during the execution on input I, s_1 computes a value that is used in s_2.

The **dependence graph** for a program P is a directed graph $G = (N, C \cup D)$. N is a set of nodes, $C \subseteq N \times N$ a set of control dependence edges, and $D \subseteq N \times N$ a set of data dependence edges. The nodes in N represent the statements and the control predicates in P with one distinguished predicate called *entry*. There is a control dependence edge $v_1 \rightarrow v_2 \in C$ if v_1 is **immediately** control dependent on v_2. Note that by the restriction to immediate control dependent statements, there are no transitive control dependence edges in the graph. There is a data dependence edge $v_1 \rightarrow v_2 \in D$ if v_1 is data dependent on v_2.

3.1.1. Static Data Dependence

If the data dependencies in the dependence graph for a program P are determined statically, the graph is called a **static dependence graph**, denoted G_s. A static dependence graph for P represents the dependencies that hold for all executions of P. The complete static graph is built at compile-time. Traditional static data flow analysis techniques are used to determine a conservative approximation of the def-use pairs in P and to establish corresponding static data dependence edges in the graph. Although there have been some efforts in distinguishing individual array elements in static data flow analysis [13], arrays are typically treated like one scalar. Each definition/use of an array element is treated as a definition/use of the entire array. Pointer variables form another source of difficulty in static data flow analysis. Some approaches to statically analyze programs with pointers have been described in [4, 11, 18]. Fig. 2 shows the static dependence graph for the program example in Fig. 1.

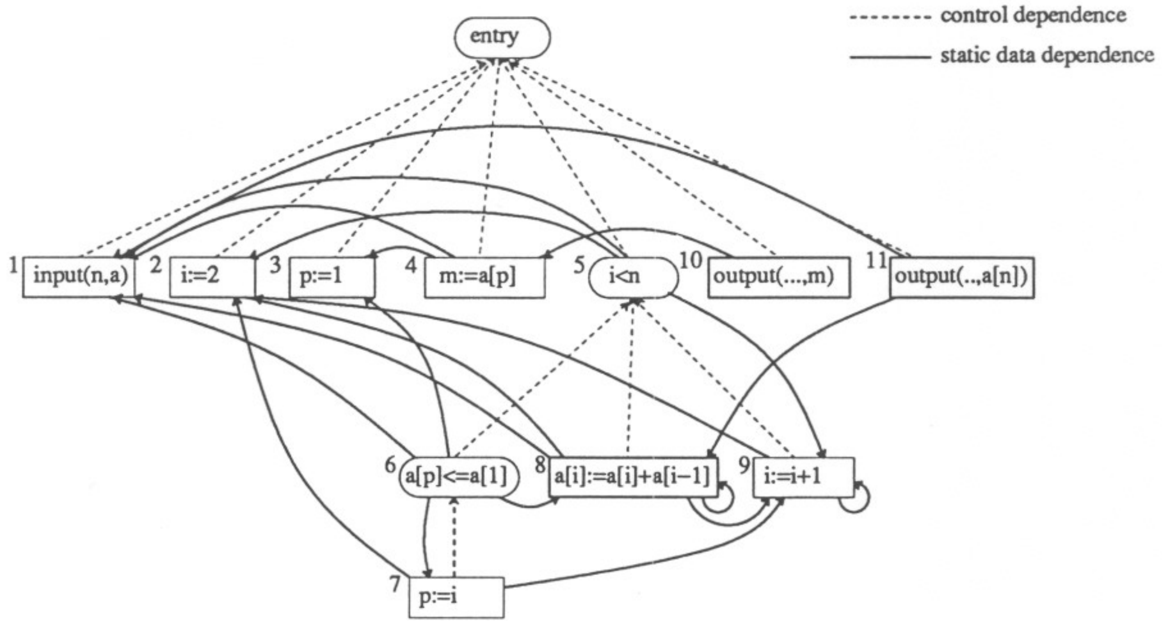


Fig. 2: The static dependence graph for the program in Fig. 1 (i).

3.1.2. Dynamic Data Dependence

An alternative to applying static data flow analysis is to collect dynamic data dependence information at run-time. A dependence graph for a program P that is built using the dynamic data flow information for a specific input I is called a **dynamic dependence graph** for P with respect to I and is denoted G_d. In contrast to the static dependence graph, G_d represents the specific data dependencies that hold in a particular execution. Thus, for a program P the relation $G_d \subseteq G_s$ holds with respect to any execution. To construct a dynamic dependence graph G_d, we first statically build the control dependence subgraph of G_d. Information about dynamic def-use pairs is collected **on-the-fly** as the program executes. We utilize a scheme similar to Korel's dynamic data flow analysis [16]. The original code is instrumented to determine the dynamic reaching definitions at run-time . This way, dynamic data dependence edges are inserted in the graph on-the-fly without requiring a static approximation of the potential def-use pairs.

We associated with each variable v a pointer *v.dptr*. At any point during execution *v.dptr* points to the program statement that last defined a value for v. Thus, for each executed use of v, the dynamic reaching definition is immediately found through *v.dptr* and a corresponding dependence edge is created. Every definition of a variable v that is executed in a statement s causes an update of *v.dptr* to point to s.

To handle composite structures such as records, the individual record components are treated in the same way as described above. If, however, the entire record is used or defined, code instrumentation is inserted for each component of the record. The same procedure follows for a reference to an entire array. For the reference of individual array elements, code instrumentation is added for the evaluated array element and also for every variable that occurs in the subscript expression. The use of pointer variables requires special treatment. Pointer variable references are similar to array references in that they may access a different variable in different instances. However, the reference of a pointer may actually access two variables: the pointer itself and the variable pointed to by the pointer. If a pointer p points to a variable v and v is accessed through p, we also insert code for the appropriate run-time actions for the reference to v. A high-level illustration of the code instrumentation on source code level is shown in Fig. 3.

```
              ...
(100)      x:=*p;                         CreateEdge( 100 → p.dptr)
                                          CreateEdge( 100 → eval(*p).dptr)
                                          x.dptr:= 100

(101) a[x]:=0;                            CreateEdge( 101 → x.dptr)
                                          a[eval(x)].dptr:= 101

(102)      q:=p;                          CreateEdge( 102 → p.dptr)
                                          q.dptr:= 102

(103)     *q:=x;                          CreateEdge( 103 → x.dptr)
                                          CreateEdge( 103 → q.dptr)
                                          eval(*q).dptr := 103


            (i)                                    (ii)
```

Fig. 3: A program fragment (i) and code instrumentation (ii).

The code instrumentation enables the construction of the dynamic data dependence subgraph as the program executes. Since the variables accessed by a subscripted reference or by a pointer may vary in different instances of the reference, a new data dependence edge is created for each distinct access. After execution the complete dynamic dependence graph is available for the extraction of program slices. The dynamic dependence graph for the program from Fig.1 with respect to the input n=4, and a=(0,0,0,4) is depicted in Fig. 4.

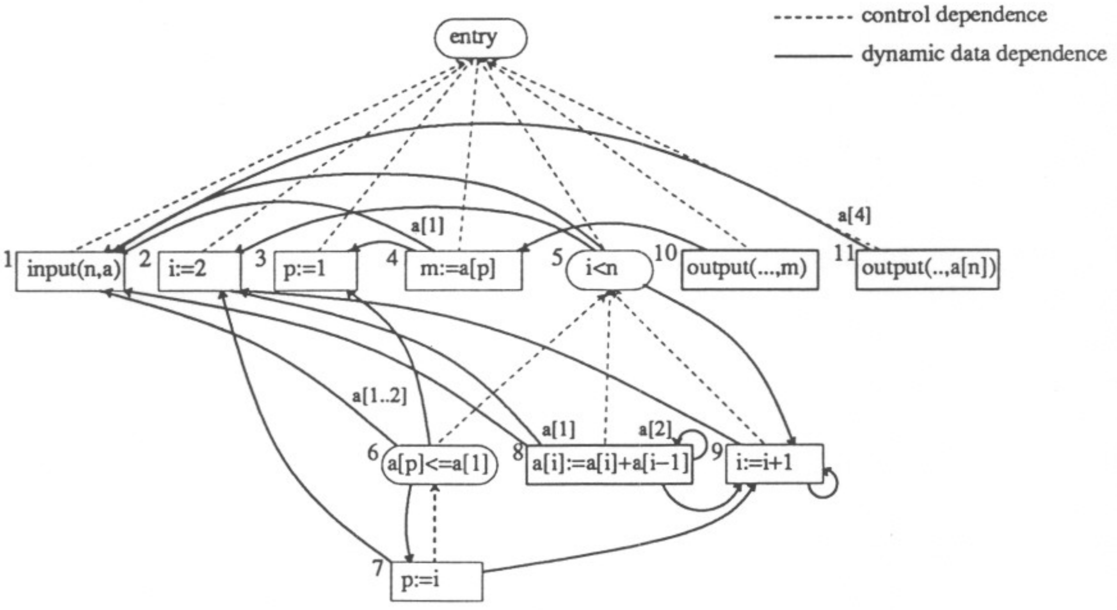


Fig. 4: The dynamic dependence graph for the program in Fig. 1 (i) with respect to the input n=4, a=(0,0,0,4).

3.1.3. Combining Static and Dynamic Dependence

Constructing the dynamic dependence graph does not require static data flow analysis; however it does require some execution overhead. Although only a constant amount of execution time overhead is required for each executed statement, it may still be preferable to move as much work to compile-time as possible. We may want to compute static data dependence information for the class of variables for which static data flow analysis does not become overly conservative (i.e., scalar variables) and use dynamic analysis otherwise (i.e., arrays and pointers). Thus, the static dependence graph can be constructed partially prior to execution and dynamic dependence information is collected for the remaining portion of the graph at run-time. The resulting graph, denoted $G_{s/d}$, contains both static and dynamic data dependence edges and forms a more accurate approximation of the actual def-use pairs than the purely static dependence graph.

Intuitively, the set of variables V in a program is partitioned into two sets V_1 and V_2. Static data flow information is computed only for variables in V_1 and corresponding static dependence edges for variables in V_1 are created in the dependence graph prior to execution. For variables in the other set V_2 code instrumentation is added to enable the creation of dynamic dependence edges for accesses of variables in V_2 at run-time. However, care must be taken in partitioning the program variables in a static set V_1 and a dynamic set V_2. In order to obtain correct data dependence information with respect to the actual dependencies in an execution, it must be ensured that no variable in one set is used to access a variable from the other set. A partition of the variables in a program that satisfies this requirement is called a **feasible partition.** For example, if there are no pointers in the program, a feasible partition divides the program variables, such that array references are analyzed dynamically and other scalar variables statically. If there are pointer variables but the class of variables that can be pointed to by the pointers can be restricted (for example by typing and pointer analysis [4, 11]), the pointer variables together with the variables they may point to can be analyzed separately. A feasible partition for the program in Fig. 1 is V_1 = {`n,i,m,p`} and V_2 = {`a`}, where V_1 contains all scalar variables to be analyzed statically and V_2 contains the array `a` to be analyzed dynamically.

3.2. Output-Slices

An output-slice of a program P is an executable suprogram of P defined as follows:

Definition: Given a set of output statements S in a program P and an input I for P.

(1) A **static output-slice** of P with respect to S is a subprogram P' of P that, when executed, computes the same values in S as P does.

(2) A **dynamic output-slice** of P with respect to S and input I is a subprogram P'' of P that, when executed on input I, computes the same values in S as P does.

Note, that if a combination of static and dynamic data dependence information was used to create the dependence graph with respect to an input I, the resulting output-slices are dynamic output-slices, i.e., their execution is only defined for input I.

There may be more than one static output-slice for a program P and a set of output statements S. Similarly, there may be more than one dynamic slice with respect to some input. In particular, a static slice for a set S is also a dynamic slice for S with respect to each input I. The problem of determining the statement-minimal static slice is undecidable [25], as is the problem for dynamic slices. Thus, data flow analysis is used to construct a conservative approximation of the statement-minimal slice.

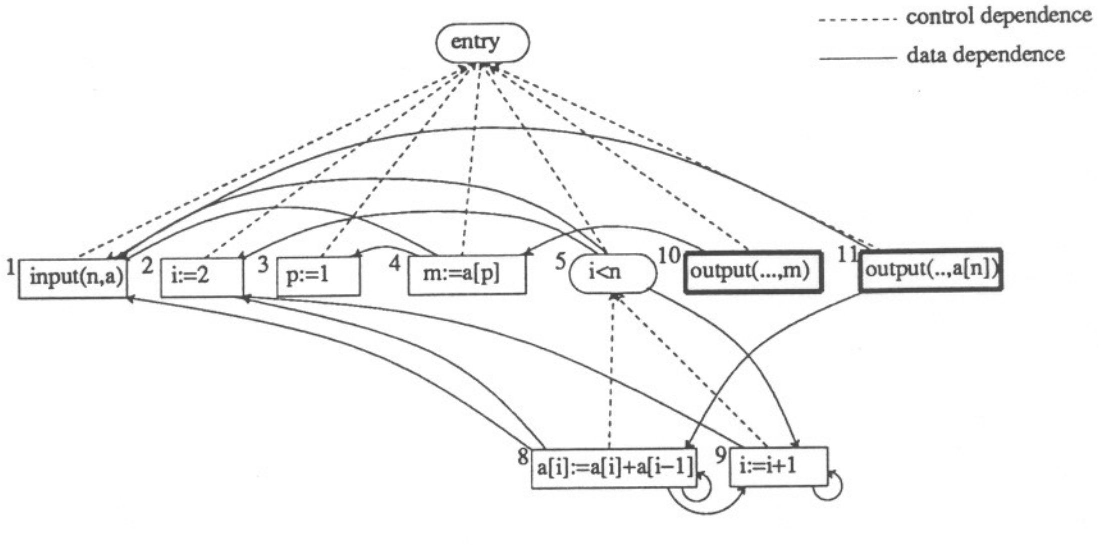
We extract static output-slices from the static dependence graph for a program P, and extract dynamic output-slices from a dynamic dependence graph for P. For a given set of output nodes S in a dependence graph G[1] for a program P, the output-slice with respect to S corresponds to a subgraph of G denoted G/S. The subprogram is obtained from G/S by restricting P to only those statements and predicates that occur in G/S. We refer to both the subgraph G/S and the corresponding subprogram as an output-slice. If G is a static dependence graph for P, G/S is a static output-slice, otherwise if G is a dynamic dependence graph with respect to some input I, G/S is a dynamic output-slice.

In order to define the subgraph of a program's dependence graph that corresponds to an output-slice, we utilize the following terminology. A node n_1 is reachable from a node n_2, denoted $n_2 \Rightarrow n_1$, if there is a path from n_2 to n_1. An edge $e=(n_1, n_2)$ is reachable form a node n, denoted $n \Rightarrow e$, if there is a path starting at n that contains e. An output-slice G/S is the restriction of G to only those nodes and edges that are reachable from a node in S. Thus, given a dependence graph G = (N, E) for a program P and a set S of output nodes in G, the **output-slice** with respect to S is the subgraph G/S = (N′, E′), where

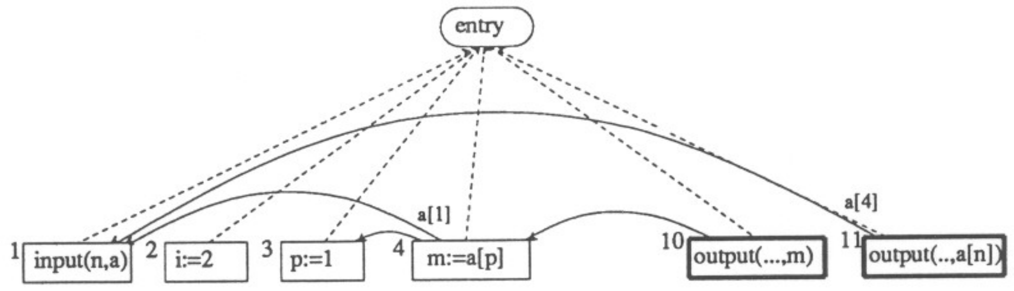
$$N' = \{\, n \in N \mid \exists\, s \in S:\ s \Rightarrow n \,\} \text{ and } E' = \{\, e \in E \mid \exists\, s \in S:\ s \Rightarrow e \,\}.$$

1 If a distinction among a static, dynamic or combined static/dynamic dependence graph is irrelevant, we merely use the term *dependence graph.*

An output-slice G/S is computed in a single traversal over a program's dependence graph G. A simple algorithm to walk the dependence graph and compute a slice has been presented in [10]. The static output-slice for the two output nodes 10 and 11 in the static dependence graph from Fig. 2, and the dynamic output-slice for the dynamic dependence graph from Fig. 4 are shown in Fig. 5 (i) and (ii), respectively.



(i) static output-slice



(ii) dynamic output-slice

Fig. 5: The static output-slice (i) for the static dependence graph from Fig. 2, and the dynamic output-slice (ii) for the dynamic dependence graph from Fig. 4.

4. Determining Data Flow Coverage

A data flow testing criterion describes a subset of the potential def-use pairs in a program to test. The goal of data flow testing is to satisfy a testing criterion, i.e., to execute as many test cases for a program as necessary to cover all def-use pairs described in the criterion. When the set of def-use pairs is completely covered, the program is considered tested with respect to the specific criterion. In general, it may not be possible to fully satisfy a criterion and exhaustively test a program. Due to the limitation of static program analysis, there may be infeasible def-use pairs included in the set of def-use pairs to test, such that no test case exists that can cover them. However, the goal remains to cover as many def-use pairs as possibly during testing and thus to get as close as possible to satisfying a particular criterion. Reaching this testing goal involves two critical issues: appropriate test case generation and accurate data flow coverage determination for a test case. If test cases are generated in an ad hoc fashion, it may be possible that during successive test cases no progress towards the testing goal is made; that is, no new def-use pairs are exercised. If data flow coverage is determined incorrectly, i.e., more def-use pairs than actually tested are covered, it may be possible that testing is terminated prematurely, leaving the programmer in the illusion that the program is sufficiently tested, although there may still be errors that could have been caught by the respective criterion.

We do not consider the problem of test case generation, which has been widely discussed in the literature [6, 16, 23], and assume that test cases are generated 'reasonably' either manually by the programmer or by an automatic test case generator. The problem of determining the data flow coverage of a test case has not received as much attention in the past, although it has been pointed out that the accuracy and efficiency of determining when a criterion is satisfied is critical for the usefulness of a testing strategy [26]. We focus on the problem of determining the data flow coverage for the OI-All-du criterion for both programs with and without arrays and pointers.

4.1. The Cover Set

The data flow coverage problem for a program P and a test case t with respect to the OI-All-du criterion can be formulated as the problem of computing the set $Cover(t)$. The set $Cover(t)$ is a subset of the potential def-use pairs in P and contains only those pairs that had influence on the computation of at least one correct output value produced by t. We present in the following sections three approaches that differ in cost and accuracy to compute the set $Cover(t)$ for a test case t.

4.2. Static Output-Slices and the Cover Set

Our first approach uses static output-slices of a program to capture the output-influence in a test case. If S is the set of output statements that are executed by test case t, we are interested in the static output-slice G_s/S. The set of data dependence edges in G_s/S contains the def-use pairs that may be of output-influence for a class of test cases, specifically, all test cases that execute exactly the output statements that are contained in S. Thus, the data dependence information in an output-slice must be tailored to a specific test case t in order to determine $Cover(t)$. For this purpose, we consider the execution path $\Pi(t)$. $\Pi(t)$ is the set of nodes in the program's dependence graph that are executed by test case t. Intuitively, to compute $Cover(t)$ we filter the information in a static output-slice using the execution path $\Pi(t)$; $Cover(t)$ is determined as the set of data dependence edges in the slice that are induced by $\Pi(t)$.

Approach 1 - Static Output-Slice: Given a program P and the execution path $\Pi(t)$ of a test case t for P. Let S be the set of output nodes in $\Pi(t)$ and G_s/S be the static output-slice with respect to S. The set of def-use pairs covered by t is determined as:

$$Cover_s(t) = \{ (n_1, n_2) \mid n_1 \rightarrow n_2 \text{ is a data dependence edge in } G_s/S \text{ and } \{n_1, n_2\} \subseteq \Pi(t) \}.$$

Approach 1 is easily implemented by using an adjacency matrix implementation for the data dependence edges in the slice G_s/S. $Cover_s(t)$ is then obtained by considering only the columns and rows of the matrix for nodes in $\Pi(t)$.

We apply Approach 1 now to the example from Fig. 2 and our test case t_0: `n=4, a=(0,0,0,4)`. Test case t_0 executes all statements and in particular both output statements. Thus, the static output-slice with respect to the executed output statements corresponds to the one depicted in Fig. 5 (i). Since all nodes in the slice are executed, $Cover_s(t_0)$ contains all data dependence edges in the slice. The set $Cover_s(t_0)$ correctly indicates that no def-use pair involved in the loop computation of the minimum element of the input array should be considered tested by t_0, although they have been exercised. In particular the incorrect use in statement 6 is not included in the cover set.

4.3. Dynamic Output-Slices and the Cover Set

Although using static output-slices prevents a number of def-use pairs from coverage that have not contributed to the computed output values, the static nature of this approach may yield too imprecise information in the presence of pointers and arrays. Consider for example the static output-slice from Fig. 5 (i). Statically, the array `a` is treated like one scalar so that the use of `a[n]` in node 11 is assumed to depend on both definitions of `a`, the definition in node 1 and the one in node 8. However, due to the error in the loop condition actually only the definition in node 1 can reach the use in node 11. Thus, the statically determined def-use pair among node 8 and node 11 is dynamically infeasible.

It has recently been shown that data flow testing requires some form of monitoring of actual references during execution to provide useful information in the presence of pointers and arrays [21].. We present now a dynamic slicing approach to accurately determine $Cover(t)$ if t contains array and pointer accesses. The dynamic dependence graph for a program P with respect to a test case t represents the actual def-use pairs among statements. Thus, the set of dynamic data dependence edges accurately describes the exercised def-use pairs. To capture the exercised def-use pairs that were of output-influence, we consider dynamic output-slices. The cover set for a test case is then determined as the set of dynamic dependence edges in a dynamic output-slice.

Approach 2 - Dynamic Output-Slice: Given a program P and a test case t for P. Let S be the set of output nodes executed in t and G_d/S the dynamic output-slice with respect to S. The set of def-use pairs covered by t is determined as:

$$Cover_d(t) = \{\ (n_1, n_2) \mid n_1 \rightarrow n_2 \text{ is a data dependence edge in } G_d/S\ \}.$$

Let us apply Approach 2 to our example. The dynamic output-slice with respect to our test case t_0 was shown in Fig. 5 (ii). The cover set $Cover_d(t_0)$ contains the dynamic dependence edges of the slice depicted in Fig. 5 (ii). By utilizing dynamic output-slices, it is clear that the produced output values do not depend on the loop computation. Thus, none of the exercised def-use pairs involved in the loop computation is prematurely covered.

4.4. Combined Static and Dynamic Output-Slices

Approach 1 that uses static output-slices provides a conservative estimate for the covered def-use pairs. Due to the limitation of static data flow analysis $Cover_s(t)$ may contain more def-use pairs than the actually output-influencing ones. The amount of inaccuracy in static data flow analysis of scalar variables is insignificant in applications such as compiler optimizations. However, as we have also demonstrated, static data flow information may be overly conservative for array and pointer variables. The dynamic Approach 2 to compute $Cover_d(t)$ overcomes this inaccuracy and provides precise information in the presence of dynamic variable accesses. Thus, for a given test case the relation

$$Cover_s(t) \supseteq Cover_d(t)$$

holds. However, the cost of the accuracy in $Cover_d(t)$ is the increased execution overhead required to collect dynamic data dependence edges. Although, the run-time overhead in our approach is kept low, as we require no run-time analysis, it is still desirable to incorporate dynamic information only if static analysis cannot provide sufficient accuracy. For this purpose, we employ in the third approach the combined static/dynamic slicing concept developed in Section 3.1.3. The computation of $Cover_{s/d}(t)$ is identical to the one in Approach 1, however instead of the static dependence graph, a combined static/dynamics dependence graph for some feasible variable partition is used.

Approach 3 - Combined Static/Dynamic Output-Slice: Given a program P and the execution path $\Pi(t)$ of a test case t for P. Let S be the set of output nodes in $\Pi(t)$ and $G_{s/d}/S$ be the static output-slice with respect to S and a feasible partition of the variables in P. The set of def-use pairs covered by t is determined as:

$$Cover_{s/d}(t) = \{ \ (n_1, n_2) \mid n_1 \rightarrow n_2 \text{ is a data dependence edge in } G_{s/d}/S \text{ and } \{n_1, n_2\} \subseteq \Pi(t) \ \}.$$

In general, $Cover_{s/d}(t)$ lies in accuracy between $Cover_s(t)$ and $Cover_d(t)$, i.e. the relation

$$Cover_s(t) \supseteq Cover_{s/d}(t) \supseteq Cover_d(t)$$

holds. Provided that variables are partitioned such that all and only the dynamically referenced variables are analyzed at run-time, we expect, however, in practice, the accuracy of $Cover_d(t)$ to be very close to $Cover_s(t)$ while the cost could be significantly lower.

5. Conclusions

We have introduced the concept of output-influence as a refinement to existing data flow testing criteria. Incorporating output-influences allows for a more rigorous approach to data flow testing that is based on the intuition that a def-use pair to be considered tested must have demonstrated evidence of correctness in terms of its contribution to a produced output value. Unlike previous data flow testing criteria, not all exercised def-use pairs are covered by a test case. Instead we cover only those exercised def-used pairs that influenced the computation of at least one correctly produced output value. We have presented several techniques based on the concept of static and dynamic program slicing to compute the output-influencing def-use pairs in a test case. By using slicing based techniques we can effectively and efficiently determine the data flow coverage including programs that contains arrays and pointers.

We are currently implementing the refined OI-All-du criterion as part of a data flow testing system. A number of empirical and analytical studies have been presented to compare the cost and effectiveness of previous testing strategies [8,26-28]. We expect to obtain insights in the practical implications of our refined criterion through comparative studies to previous approaches with respect to cost and effectiveness. When a traditional data flow criterion is satisfied for a program and a specific test suite, the corresponding OI-criterion may not be satisfied if the exact same test suite is used demonstrating the more rigorous approach using OI-criteria. In general, it is however, difficult to analytically compare the OI-criteria with previous criteria in terms of the number of required test cases. Since a refined OI-criteria and its corresponding traditional criterion may lead to a different coverage of def-use pairs in a specific test case, different test cases may be generated even if the same test case generation strategy is used. We expect to perform empirical comparisons of (1) the average number of test cases required to satisfy a traditional criterion and its refined OI-version and (2) the number of additional errors that can be caught using our refinement, but that circumvent detection using the traditional approach.

References

1. H. Agrawal and B. Horgan, "Dynamic program slicing," *Proceedings of the SIGPLAN 1990 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, vol. 25, no. 6, pp. 246-256, June 1990.

2. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic slicing in the presence of unconstrained pointers," *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 60-73, Victoria, British Columbia, October 1991.

3. A. V. Aho, R. Sethi, and J. D. Ullman, in *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Massachusetts, 1986.

4. D.R. Chase, M. Wegman, and F.K. Zadeck, "Analysis of pointers and structures," *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, vol. 25, no. 6, pp. 296-310, White Plains, New York, June 1990.

5. L.A. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A comparison of data flow path selection criteria," *Proceedings of the 8th International Conference on Software Engineering*, pp. 224-251, London, UK, August 1985.

6. R.A. DeMillo and A.J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. SE-17, no. 9, pp. 900-910, September 1991.

7. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, June 1987.

8. P. Frankl and S.N. Weiss, "Is data flow testing more effective than branch testing? An empirical study," *Proceedings of Quality Week 1991*, May 1991.

9. P.G. Frankl and E.J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering* , vol. SE-14, no. 10, pp. 1483-1498, October 1988.

10. S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *Computer Science Technical Report #756*, University of Wisconsin-Madison, March 1988.

11. S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 28-40, Portland, Oregon, June 1989.

12. S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, January 1990.

13. A. D. Kallis and D. Klappholz, "Reaching definitions analysis on code containing array references," *Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1991.

14. B. Korel and J. Laski, "A tool for data flow oriented program testing," *ACM Softfair Proceedings*, pp. 35-37, December 1985.

15. B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155-163, October 1988.

16. B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, vol. SE-16, no. 8, pp. 870-879, August 1990.

17. D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, "Dependence graphs and compiler optimizations," *8th annual ACM Symposium on Principles of Programming Languages*, pp. 207-218, Williamsburgh, Virginia, January 1981.

18. W. Landi and B.G. Ryder, "Pointer-induced aliasing: A problem classification," *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pp. 93-103, Orlando, Florida, January 1991.

19. B. P. Miller and J.-D. Choi, "A mechanism for efficient debugging of parallel programs," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 135-144, Atlanta, Georgia, 1988.

20. S. C. Ntafos, ''An evaluation of required element testing strategies,'' *Proceedings of the Conference on Software Maintenance*, pp. 311-317, San Diego, CA, December 1990.

21. T.J. Ostrand and E.J. Weyuker, ''Data flow-based test adequacy analysis for languages with pointers,'' *Proceedings of the '91 Symposium on Software Testing, Analysis, and Verification (TAV4)*, pp. 74-86, Victoria, B.C., October 1991.

22. K. Ottenstein and L. Ottenstein, ''The program dependence graph in a software development environment,'' *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on practical SDEs, SIGPLAN Notices*, vol. 19, no. 5, pp. 177-184, May 1984.

23. C.V. Ramamoorthy, ''On the automated generation of program test data,'' *IEEE Transactions of Software Engineering* , vol. SE-2, no. 4, pp. 293-300, December 1976.

24. G. Venkatesh, ''The semantic approach to program slicing,'' *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 107-119, Toronto, Ontario, Canada, June 1991.

25. M. Weiser, ''Program slicing,'' *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352-357, July 1984.

26. S.N. Weiss, ''Comparing test data adequacy criteria,'' *Software Engineering Notes*, vol. 14, no. 6, pp. 42-49, October 1989.

27. E.J. Weyuker, ''The cost of data flow testing: An empirical study,'' *IEEE Transactions on Software Engineering*, vol. SE-16, no. 2, pp. 121-128, February 1990 .

28. E.J. Weyuker, S.N. Weiss, and D. Hamlet, ''Comparison of program testing strategies,'' *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 1-10, Victoria, B.C., October 1991.