Graph Representation for Data Flow Coverage

Mario Concilio Neto, Roberto P. A. Araujo, Marcos Lordello Chaim
University of Sao Paulo
Sao Paulo, SP, Brazil
Email:{mario.neto,roberto.araujo,chaim@usp.br

Jeff Offutt
George Mason University
Fairfax, VA, USA
Email:offutt@gmu.edu

***Abstract*—Data flow testing helps testers design effective tests by requiring the tests to execute sequences of statements from definitions of variables to one or more subsequent uses. These def-use associations are derived from graphs that model software behavior. A "flow graph" that only includes paths that cover def-use associations, and not other control flows, has been defined elsewhere. Although these flow graphs have several advantages over previous graphs, as computed, they omit some valid paths, which are needed to use the graphs to discover subsumption relationships and generate test data. These omissions lead to errors in the results. This paper extends previous solutions by presenting a graph that represents all paths that cover def-use associations. The paper presents empirical data showing that this graph can be generated at reasonable cost and efficiently applied for data flow subsumption discovery.**

***Index Terms*—Software testing, Data flow coverage, Graph representation, Data structures**

I. Introduction

Data flow testing (DFT) attempts to enable comprehensive structural testing based on flows of data through software [1]–[4]. Roughly speaking, it involves developing tests that exercise (cover) every value assigned to a variable and its subsequent references (uses). These pairs of definitions and uses are called *definition-use associations* (DUA) and the paths from defs to uses are called *du-paths* [3]. DFT uses both control and data flow information to design tests. As a result, data flow tests can exercise more situations than control flow testing can. The intuition is that causing more values to reach different uses can find more problems in the software and increase confidence in its reliability.

Studies have shown that DFT can effectively detect faults in programs [5], [6] and can verify the security of web applications [7]. Hemmati [8] compared control flow criteria (statement, branch, loop, and MCDC coverage) against definition-use pair coverage with respect to their ability to detect faults. They found that out of 274 faults in sizable open-source programs, only 76 (28%) were found by control flow coverage criteria. For those same faults, definition-use pair coverage detected 79% of the faults not detected by control flow criteria. Thus, DFT can help achieve and verify software quality, which is especially important for mission-critical systems.

To achieve high DFT coverage, a tester needs to develop specific test cases to cover a large number of DUAs. Furthermore, some DUAs cannot be covered by any test case because the underlying path is infeasible. Both tasks require human intervention, which increases the cost of DFT.

Many approaches to reduce DFT's cost have been created. Some exploit the subsumption relationship among DUAs [9], [10]. A *test requirement* (TR) tr_1 (e.g., a DUA D_1) *subsumes* another test requirement tr_2 (another DUA D_2) if every complete path that traverses tr_1 also traverses tr_2. The minimal subset of TRs that subsumes every other TR is called a *spanning set* and its elements are referred to as *unconstrained* test requirements [9]. If a spanning set of DUAs could be identified, testers would only need to satisfy the unconstrained DUAs, saving time and effort. Jiang et al. [10] showed that targeting unconstrained DUAs reduces the cost of input data generation.

DFT cost can also be reduced by automatic input data generation and feasibility analysis. Jiang et al. [10] and Vivanti et al. [11] used meta-heuristic algorithms to find input data to cover DUAs. Su et al. [12] use symbolic execution and model checking to generate input data and to analyze the feasibility of DUAs.

Several researchers proposed graph-based representations and strategies to encode du-paths to help find subsumption among DUAs [9], [10], [13] and explore paths for symbolic execution [12]. However, some of these representations miss sub-paths that might block the subsumption of DUAs or that might allow a particular DUA to be covered by input data [9], [10], [13]. Other strategies encode only control flow coverage requirements, and do not trim invalid paths that have nodes that redefine variables [12].

In this work, we extended one of the solutions [9], [13] to include the missing paths and to correctly find the graph representation of data flow coverage. The new graph representation, called *graphdua*, includes all valid paths covering a DUA. Furthermore, we show how to generate graphduas and experimental data demonstrating that they are affordable for software systems at scale and useful for data flow subsumption discovery.

The paper is organized as follows. In the next section, we introduce the basic concepts regarding DFT. A simple running example of a graphdua is presented in Section III. Section IV contains the formal definition and presents how to generate the graphdua. Sections V and VI present the application of graphduas in data flow subsumption and our experimental assessment. Related work is discussed in Section VII and conclusions are in Section VIII.

II. Background

Let P be a program mapped into a flow graph $G(N, E, s, e)$, where N is the set of nodes representing basic blocks, s is the start node, e is the exit node, and E is the set of edges (n',n), such that $n' \neq n$, and there is a potential transfer of control from n' to n. We refer to n' as the *origin* node of the edge and n as the *target* node of the edge (n',n). A *basic block* is a sequence of statements such that if the first statement is executed, then all statements are executed in sequence. Note that we assume one start node, corresponding to a single entry point to the software unit being graphed, and one exit node. If a method has multiple return statements, we add a special exit node and edges from each node that includes a return statement to the exit node. All flow graphs are directed graphs.

Figure 1 presents a program that sorts an array of integers (originally from Marré and Bertolino [9]). The number before each line of code indicates the node the line is associated with in the control flow graph, shown in Figure 2.

A *path* is a sequence of nodes (n_i, ..., n_k, n_{k+1}, ..., n_j), where $i \leq k < j$, such that $(n_k, n_{k+1}) \in E$. A node n_k is said to be a *predecessor* of a node n_{k+1} if there exists an edge (n_k,n_{k+1}) in E, and n_{k+1} is said to be a *successor* of n_k. A path is *simple* if all of its nodes are distinct with the possible exception of the first and last nodes. A path (n_i, ..., n_k, n_{k+1}, ..., n_j) is said to be *complete* if n_i is the start node and n_j is the exit node.

The sub-graph $SG(n_i, n_j)$, obtained from a graph $G(N, E, s, s)$, represents all paths from nodes n_i to n_j in G. Let n_i and n_j be two nodes in *G(N,E,s,e)* such that n_i *reaches* n_j; that is, there is at least one path from n_i to n_j. The sub-graph $SG(n_i, n_j)$ of *G* between n_i and n_j is given by the directed graph $G'(N', E', s', e')$[1], where:

1) n_i and $n_j \in N'$ such that n_i is the start node s' and n_j is the end node e'
2) if there exists a path (n_i, n_{i+1}, ..., n_k, ..., n_{j-1}, n_j) of *G*, where $i < k \leq j$, then $n_k \in N'$ and (n_{k-1}, n_k) $\in E'$

A sub-graph encodes all paths from node n_i to node n_j such that n_i and n_j occur only once as first and last nodes [14]. Figure 3 shows the sub-graph *SG(1,8)* obtained from the flow graph described in Figure 2; it represents all paths from node 1 to node 8.

Figure 4 shows the sub-graph *SG(8,8)* where the start and exit nodes are both 8 (shown in light gray). In Figures 3 and 4, a sub-graph's node is identified by a pair *n(I)* where n is the node of the original flow graph G and I is the identifier of the sub-graph.

Data flow testing requires that test cases exercise paths in a program between locations where a value is assigned to a variable (called *definitions*, or *defs*) and its subsequent references (called *uses*). A variable can be used to compute a value or to compute a predicate. Value computations are associated with nodes and predicate computations are associated with edges.

[1]In this paper, a sub-graph is not a flow graph because its primary goal is to represent paths. As a result, its edges are not necessarily associated with control flow commands such as *if* and *while* as in a flow graph.

```
/*1*/ void sort (int a[], int n)
/*1*/ {
/*1*/ int sortupto, maxpos, mymax, index;
/*1*/ sortupto=1;
/*1*/ maxpos=1;
/*2*/ while(sortupto < n)
/*3*/   {
/*3*/   mymax = a[sortupto];
/*3*/   index = sortupto +1;
/*4*/   while(index <= n)
/*5*/     {
/*5*/     if(a[index] > mymax)
/*6*/       {
/*6*/       mymax = a[index];
/*6*/       maxpos=index;
/*6*/       }
/*7*/     index++;
/*7*/     }
/*8*/   index = a[sortupto];
/*8*/   a[sortupto]=mymax;
/*8*/   a[maxpos]=index;
/*8*/   sortupto++;
/*8*/   }
/*9*/ }
```

Figure 1. Example program Sort

Figure 2 gives the flow graph of the Sort program annotated with definitions (def) and uses associated with nodes and edges. A *definition-clear* (def-clear) path with respect to (wrt) a variable X is a path where X is not redefined in any node in the path, except possibly in the first and last nodes. A *du-path* wrt to variable X is a def-clear path that is also a simple path. A def-clear path wrt to X is a du-path that allows def-clear side-trips; that is, the nodes between the first and last nodes may occur several times as long as they do not redefine X.

A sub-graph can be determined in such a way that it contains only def-clear paths wrt a variable X between two nodes n_i and n_j. The *def-clear* sub-graph $SG(n_i, n_j, X)$ wrt variable X of *G* between nodes n_i and n_j is given by the directed graph $G'(N', E', s', e')$, where:

1) n_i and $n_j \in N'$ such that n_i is the start node s' and n_j is the end node e'
2) if there exists a def-clear path (n_i, n_{i+1}, ..., n_k, ..., n_{j-1}, n_j) wrt variable X, where $i < k \leq j$, then $n_k \in N'$ and (n_{k-1}, n_k) $\in E'$

A sub-graph $SG(n_i, (n_j, n_k), X)$ of *G* comprising all def-clear paths wrt variable X between node n_i and edge (n_j, n_k) can be analogously defined (see Section IV-D).

Data flow testing criteria require that *definition-use associations* (DUAs) be covered. The triple D = (d, u, X) represents a data flow testing requirement involving a definition in node d and a use in node u of variable X such that there is a def-clear path wrt X from d to u. Likewise, the triple D = (d, (u', u), X) represents the association between a definition and a use of a variable X in edge (u', u). In this case, a def-clear path (d,...,u',u) wrt X should exist.

The all uses criterion [3] requires that all uses be executed

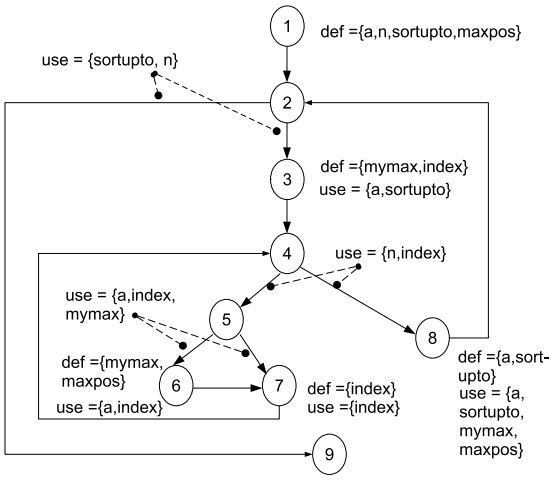

Figure 2. Annotated flow graph for Sort

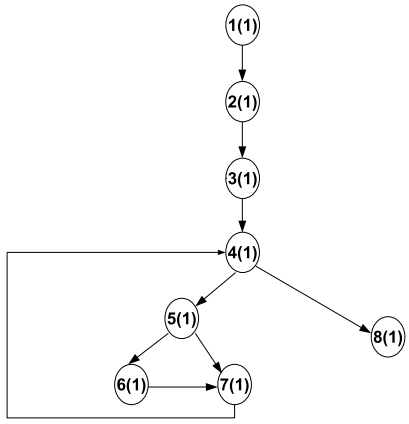

Figure 3. Sub-graph *SG1*

Table I
ALL USES TEST REQUIREMENTS FOR PROGRAM SORT

All uses			
(1,3, a)	(1,(2,3), sortupto)	(3,(4,8), index)	(6,8, mymax)
(1,(5,6), a)	(1,(2,9), sortupto)	(3,(5,6), index)	(7,(4,5), index)
(1,(5,7), a)	(1,3, sortupto)	(3,(5,7), index)	(7,(4,8), index)
(1,6, a)	(1,8, sortupto)	(3,6, index)	(7,(5,6), index)
(1,8, a)	(1,8, maxpos)	(3,7, index)	(7,(5,7), index)
(1,(2,3), n)	(3,(5,6), mymax)	(3,8, index)	(7,6, index)
(1,(2,9), n)	(3,(5,7), mymax)	(6,8, maxpos)	(7,7, index)
(1,(4,5), n)	(3,8, mymax)	(6,(5,6), mymax)	(7,8, index)
(1,(4,8), n)	(3,(4,5), index)	(6,(5,7), mymax)	(8,3, a)
(8,(5,6), a)	(8,(5,7), a)	**(8,6, a)**	(8,8, a)
(8,(2,3), sortupto)	(8,(2,9), sortupto)	(8,3, sortupto)	(8,8, sortupto)

at least once. Specifically, the set of paths executed by the test cases of a test set T must include a def-clear path for each DUA (d, u, X) or (d, (u', u), X). A test set with such a property is said to be *adequate* for the all uses criterion for program P since all required DUAs were *covered*. Table I shows the all uses test requirements for the example program Sort.

III. ALL PATHS COVERING A DUA

Suppose that we want to determine all complete paths that cover DUA (8,6, a) (shown in boldface in Table I). This is needed to find input data to cover the association. We next show informally how *graphdua(8,6, a)*, which encodes all these paths, is determined.

A. Reaching the definition

To cover (8,6, a), the input data must first cause the program to reach the definition node (node 8). Several paths reach the definition node 8.

Figure 2 shows that [1,2,3,4,8] is the shortest path to reach node 8. Other reaching paths include [1,2,3,4,5,6,7,4,8] and [1,2,3,4,5,7,4,8]. Figure 3 presents a sub-grah that represents all paths from the start node 1 to node 8. In Figure 3 (sub-graph *SG1*) nodes are identified by a pair $n(1)$ where 1 identifies the sub-graph *SG1* and n is the original node from the original flow graph.

B. Visiting the definition multiple times

Now that we have found paths to reach the def at node 8, the next step is to reach the use at node 6 by traversing a def-clear path wrt variable *a* from node 8 to node 6. Suppose the path [8,2,3,4,5,7,4,5,7,4,8] is traversed before node 6 is reached. This path visits the definition at node 8 twice before reaching the use at node 6. Depending on the test case, node 8 may be visited several times before reaching the use node.

Figure 4 shows the *SG2* sub-graph, which represents paths from node 8 back to itself. For this purpose, the definition node 8(2), shown in light gray, should be both the start and exit node of *SG2*. The underlying idea of sub-graph *SG2* is to represent paths that start and end at the definition node. In these paths, a redefinition of the DUA variable (in the example, variable *a*) may occur. Since the exit node of *SG2* is also the definition node, the redefinition has no impact on the coverage of a DUA. The use node, 6(2), is also included in *SG2*. Though counter-intuitive, the use node in *SG2* might appear in paths in which DUA (8,6, a) is covered twice or more times.

C. Reaching the use

Since we assume that DUA (8,6, a) will eventually be covered, there exists at least one def-clear path wrt variable *a* from node 8 to node 6 that will be traversed. So we need to find the sub-graph containing all def-clear paths wrt variable *a* that reach the use node. Figure 5 shows sub-graph *SG3*, which describes all def-clear paths wrt variable *a* from node 8 to node 6. Note that the definition node, node 8(3), is the start node and node 6(3), the use node, is the exit node.

D. Visiting the use multiple times

Now that we have reached the use, we need a path that completes execution to reach the exit node. Just as with the definition node, the path after the use node may cycle back to the use one or more times. Consider the path

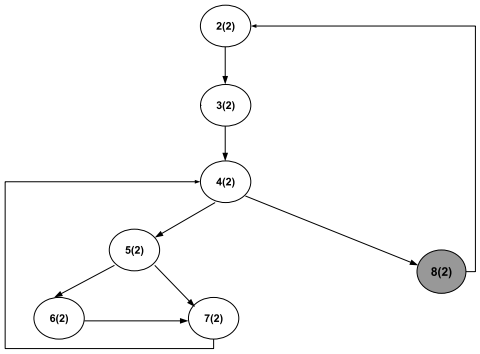

Figure 4. Sub-graph *SG2*

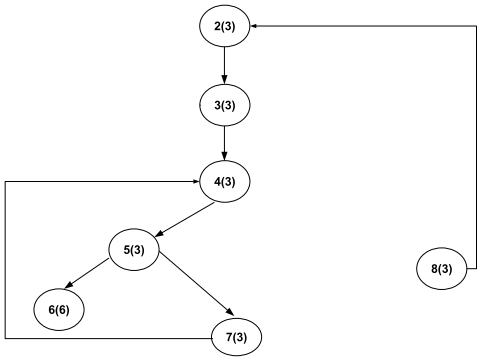

Figure 5. Sub-graph *SG3*

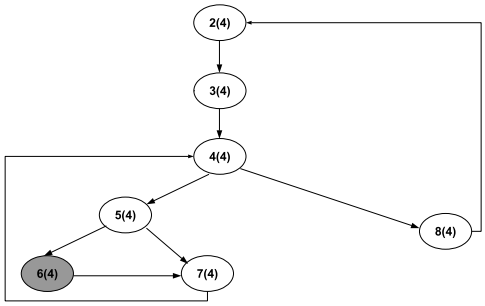

Figure 6. Sub-graph *SG4*

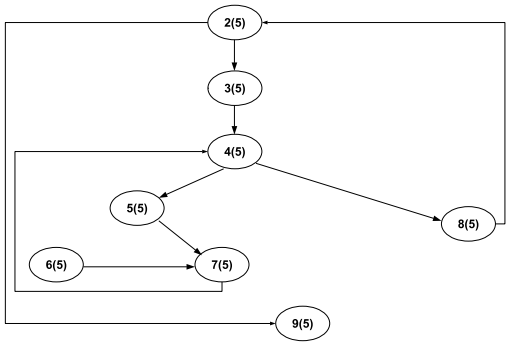

Figure 7. Sub-graph *SG5*

[6,5,7,4,5,7,4,8,2,3,4,5,6], which returns to the use at node 6. Sub-graph *SG4* in Figure 6 describes all paths that start and end at node 6. The start and exit node of *SG4* is the use node, node 6(4), shown in light gray in Figure 6.

E. Reaching the exit node

Finally, we must find a path from the last execution of the use node to the exit node. Figure 7 shows all paths that start at node 6 and end at the exit node. The start node is node 6(5) and the exit is node 9(5).

F. Graphdua(8,6, a)

Next we connect sub-graphs *SG1*, *SG2*, *SG3*, *SG4*, and *SG5* in Figures 3 through 7 to create the graph to represent DUA coverage. This results in a graph that comprises all complete paths that cover DUA (8,6, a). This graph, called graphdua(8,6, a), is shown in Figure 8.

The exit node of *SG1* is the same as the entry nodes to *SG2* and *SG3*, so it is connected to the successors of the entry nodes. We need both edges because, after touring *SG1*, execution may cycle back to the definition node (graph *SG2*) or go directly to the use (graph *SG3*).

SG2, in turn, connects to the successors of the start node of *SG3*. Because of the possibility of cycles, the exit node of *SG3* connects to both *SG4* and *SG5*.

Note that nodes from G can appear more than once in a graphdua, which necessitates the more complicated node labels. The first number is the label from G and the second is the sub-graph. We only need the first number to display a complete path.

Observe in Figure 8 that the graphdua has dangling nodes (8(3) and 6(5)); they are unreachable nodes. These appear because the graphdua only models paths that include def-clear sub-paths from the def to the use, so some nodes from the original graph cannot be visited twice.

The next section generalizes this example to present a complete formal definition of a graphdua, and then shows how to build it.

IV. GRAPHDUA

The *graphdua(D)* models all paths that cover a particular DUA $D = (d, u, X)$. This section formally presents the graphdua(D) and an algorithm to build it. The construction of the graphdua for an edge DUA $(d, (u', u), X)$ is reduced to the problem of finding the graphdua for node DUAs in Section IV-D.

A. Graph definition

Let $D = (d, u, X)$ be a DUA required to test a program P to satisfy the all uses criterion. A *graphdua(D)* for D is given by a tuple $G_D(N_D, E_D, s_D, e_D)$ such that:

1) N_D is the set of nodes $n(I)$ where n is a node in the flow graph $G(N, E, s, e)$ of P and I identifies the instance of n in G_D
2) E_D is the set of edges $(m(I), n(I'))$ where (m, n) is an edge in G

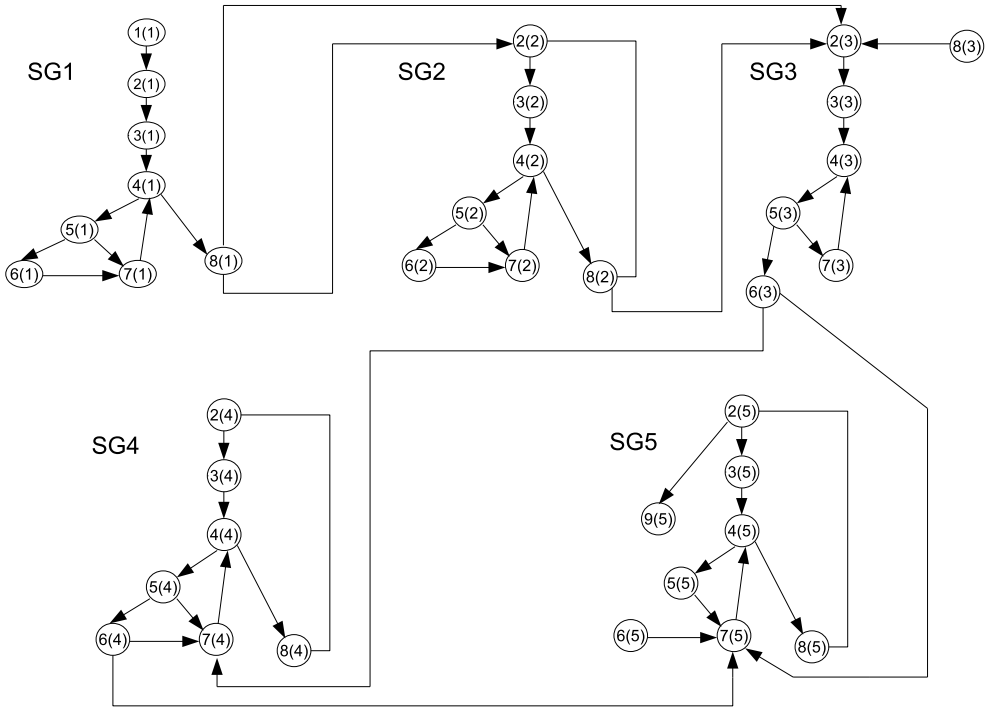

Figure 8. graphdua(8,6, a) includes all complete paths that cover DUA (8,6, a)

3) $s_D = s(I)$ and $e_D = e(I')$ are the start and exit nodes in G_D, where s and e are the start and exit nodes in G
4) any path that covers D can be obtained by traversing *graphdua(D)* and returning the value n for every node $n(I) \in N_D$ that is visited

B. Finding sub-graphs

As shown in Section III, the graphdua is built by finding sub-graphs and connecting them. Bertolino and Marré [14] proposed an algorithm, which we call **findSubGraph**, to determine the sub-graph $SG(n_i,n_j)$ between nodes n_i and n_j of a flow graph G[2]. In the example above, sub-graphs *SG1*, *SG2*, *SG4*, and *SG5* are created using **findSubGraph**. These sub-graphs encode all paths from a node n_i to a node n_j.

However, when a DUA (d, u, X) is covered, there must exist at least one path from the definition node d to the use node u such that variable X is not redefined; that is, a def-clear path from d to u wrt X. The sub-graph *SG3* encodes these paths. Marré and Bertolino [9] also proposed a slightly different algorithm to determine sub-graphs that encompass def-clear paths, which we call **findDefClearSubGraph**. The cost of both algorithms **findSubGraph** and **findDefClearSubGraph** is O(E) [14].

However, **findSubGraph** and **findDefClearSubGraph** as defined by the authors may include dangling paths. For example, if one utilizes **findSubGraph** to determine *SG(2,7)* from the flow graph of Sort (Figure 2) the sub-graph of Figure 9 will

[2] $SG(n_i, n_j)$ is found by visiting nodes forwards from n_i and nodes backwards from n_j; nodes visited in both searches are then connected. We refer the reader to [14] for more details.

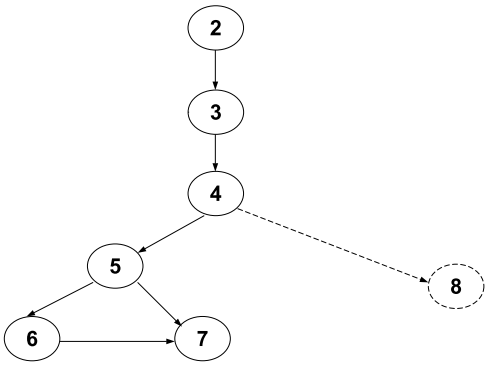

Figure 9. Sub-graph with dangling paths (*SG(2,7)*)

be generated. The dashed edge and node represent a dangling path. We clean up the dangling paths by using Algorithm 1; it receives as parameters the sub-graph and a list of nodes to be checked; that is, nodes that have no successors and are not the exit node. Algorithm 1 also costs O(E) since it visits at most three times all edges of the original flow graph. As a result, the costs of **findSubGraph** and **findDefClearSubGraph** are not altered by adding the clean-up of dangling paths.

C. Finding the graphdua

Finding a graphdua(D) requires invoking **findSubGraph** to determine *SG1*, *SG2*, *SG4*, and *SG5* and **findDefClearSubGraph** for *SG3*. Subsequently, the sub-graphs should be connected accordingly to the rules described in Section III.

Input: Sub-graph $SG(N_{SG}, E_{SG}, s_{SG}, e_{SG})$; *Check* – set of nodes to be checked
Output: sub-graph *SG* that encodes all paths from n_i to n_j

```
// Visit nodes backwards from exit node
W ← e_SG; PredVis ← ∅;
while W ≠ ∅ do
    n ← remove_node_from(W);
    foreach n_pred ∈ Predecessors(n) do
        if not n_pred ∈ PredVis then
            PredVis ← PredVis ⋃ n_pred;
// Visit nodes backwards from
   to-be-checked nodes
CheckVis ← ∅;
while Check ≠ ∅ do
    n ← remove_node_from(Check);
    foreach n_pred ∈ Predecessors(n) do
        if not n_pred ∈ CheckVis then
            CheckVis ← CheckVis ⋃ n_pred;
// Clean up dangling path
Cleanup ← CheckVis − PredVis;
while Cleanup ≠ ∅ do
    n ← remove_node_from(Cleanup);
    foreach n_pred ∈ Predecessors(n) do
        E_SG ← E_SG − (n_pred, n);
    foreach n_suc ∈ Successors(n) do
        E_SG ← E_SG − (n, n_suc);
    N_SG ← N_SG − n;
return SG(N_SG, E_SG, s_SG, e_SG)
```

Algorithm 1: cleanUpDanglingPaths

Note that the only sub-graph that is guaranteed to exist is *SG3*; the others depend on the characteristics of the DUA. For example, DUA (1, (2,9), n) will only have *SG3* since the start node is the definition node and the target node of the edge is the same as the exit node. Thus, *SG1*, *SG2*, *SG4* and *SG5* do not exist.

The cost of creating a graphdua(D) is driven by the cost to find the sub-graphs and connecting them. A graphdua will have at most five sub-graphs; each one costs O(E) to be found. As result, the cost of finding all sub-graphs is O(E). Additionally, one needs to connect the sub-graphs. At most, six connections between sub-graphs will be established. Each connection consists of creating new edges between the exit node of the precedent sub-graph with the successors of the start node of the next sub-graph. Since the new edges are limited by the number of edges (E) of G, the cost to connect two sub-graphs is O(E). Hence, the total cost of finding a graphdua is also O(E).

D. Def-clear sub-graph generation for edge DUAs

For edge DUAs such as $(d, (u', u), X)$, sub-graphs *SG1*, *SG2*, *SG4*, and *SG5* can be calculated using **findSubGraph**, just as for node DUAs. However, if a target node u in SG3 for an edge DUA has more than one predecessor, we need to do more work.

Consider DUA (8,(5,7), a) from program Sort (Figure 2). All paths from node 8 to edge (5,7) are def-clear wrt variable a. However, node 7 has two predecessors: nodes 5 and 6. If preceded by 6, node 7 will occur at least twice in the path to edge (5,7), since it is always the last node of the sub-graph.

Bertolino and Marré's algorithm, though, assumes that the end node (in this case node 7) will occur only once in the paths represented by the generated sub-graph[3]. To overcome this restriction, we modify the original flow graph to calculate *SG3* for edge DUAs. Figure 10 shows part of an annotated flow graph in which there exists an edge (u', u) with an edge use of variable X and a node d with a definition of X. As shown in the figure, node u has several predecessors. Let us suppose there is a def-clear path from d to (u', u) and, as a result, an edge DUA $(d, (u', u), X)$.

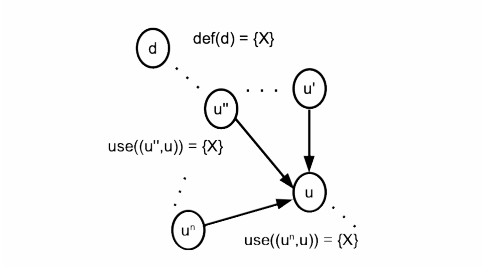

Figure 10. Edge DUA with target node u with more than one predecessor

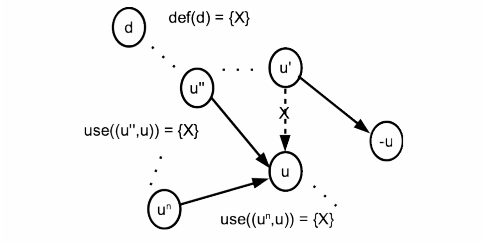

Figure 11. Modification of the flow graph to find *SG3* for edge DUAs

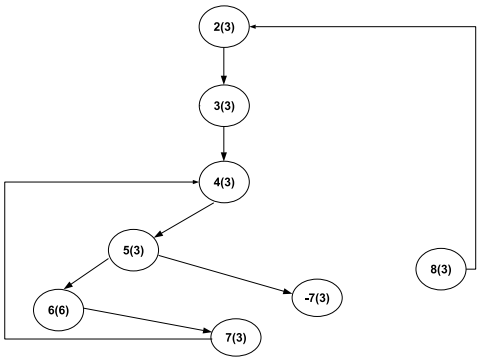

Figure 12. *SG3*—Def-clear sub-graph wrt variable X that starts at node 8 and ends at edge (5,-7)

The original flow graph is modified as follows: Edge (u', u) is removed, and a new node $-u$ (minus u) is added as well as edge $(u', -u)$. We choose to label the new node $-u$ to

[3]Bertolino and Marré's algorithm use a flow graph in which sequential statements are associated with edges and conditional transfer statements with nodes. Their notation does not distinguish between node and edge DUAs.

differentiate it from node u, which may occur more than once in a path to edge (u',$-u$). Figure 11 shows the new flow graph resulting from the modifications.

Though node $-u$ has a different label, it has the same code and, consequently, is annotated with the same sets of defs and uses as node u. Its purpose is to allow **findDefClearSubGraph** to be applied to edge DUAs. As a result of including edge $(u', -u)$ in the flow graph, **findDefClearSubGraph** can be applied having as parameters d for the first node (n_i) and $-u$ for the last node (n_j) of the sub-graph. By doing so, the problem of finding *SG3* for edge DUAs is reduced to that of finding *SG3* for node DUAs. Figure 12 shows the *SG3* generated for DUA (8,(5,7), a).

V. Application of the graphduas for data flow subsumption

The graphdua is particularly, but not limited to, useful for subsumption analysis among DUAs; that is, for discovering the unconstrained DUAs whose coverage may imply covering all the other DUAs of a program. Figure 13 gives the subsumption relationship among DUAs of the Sort program; the light-gray squares (the leaves of the graph) contain the unconstrained DUAs. The subsumption graph has eleven leaves, which means that by covering eleven DUAs, one from each leaf, all DUAs might be covered.

Marré and Bertolino [9] suggested an algorithm to find the DUAs subsumed by a DUA $D_1 = (d_1, u_1, X_1)$. They first select all paths that cover D_1 by building an intermediate graph $G*$ (we contrast the graphduas and $G*$ in Section VII). Then they check whether every path of $G*$ also traverses a DUA $D_2 = (d_2, u_2, X_2)$ by initially verifying that d_2 and u_2 are always traversed in $G*$. Then, to find whether every path covering D_1 also covers D_2, they check whether no node n_i from d_2 and u_2 in $G*$ contains a definition of X_2.

Data flow subsumption can also be found by applying the Subsumption Algorithm (SA) [15] on graphduas. SA calculates those DUAs that are covered in all paths from the start node s to particular node n. When SA is applied on a graphdua(D), those DUAs covered in all paths from the start node to the exit node of graphdua(D) comprise the set of DUAs subsumed by DUA D.

VI. Experimental analysis

We investigated empirically whether graphduas can be calculated at scale and whether its application in finding data flow subsumption leads to test cost reduction. We addressed two research questions:

RQ1: How long does it take to find the graphduas of a program?

RQ2: How much test effort is saved by using unconstrained DUAs?

The rest of this section presents and discusses the results of our study. We conclude the section with threats to validity.

A. Results

For our study, we chose 17 programs from the Defects4J repository [16], plus the machine learning program Weka. We selected the first buggy version (referred to as 1b) from Defects4J and Weka's version 3.8. The programs' purposes vary: manipulating text in compressed and binary files (Compress, Csv, Gson, JacksonCore, JacksonDataBind, JacksonXml, and JSoup); parsing and compiling (Cli, Closure, and JXPath); data structure manipulation and language utilities (Collections and Lang); mathematics, statistics, and data mining (Math and Weka); data and time manipulation (Time); and software testing (Mockito). The programs' size also vary: they range from small programs such as Csv (929 DUAs) to larger programs such as Weka (337,063 DUAs).

Table II presents empirical data from applying the graphdua algorithm to our subject programs. The table gives the name of each program, the number of DUAs (**#DUAs**), the percent of unconstrained DUAs wrt the total of DUAs (**Unc.**), the number of milliseconds needed to find all graphduas, averaged over 10 trials (**TGra.**), the number of milliseconds needed to discover the unconstrained DUAs (**TUnc.**), and the ratio between graphdua calculation and subsumption analysis time (**Rat.**).

We implemented and ran the algorithm to generate all graphduas for every method that has at least one DUA. Many methods have none because they only have local definitions and uses; these methods' data are not included in Table II. Figure 15 presents the graphdua generated for DUA (3,(5,7), mymax) of the Sort program by the algorithm. For each method, we ran SA on every graphdua to find the subsumption relationship and then the unconstrained DUAs. All data were collect using a MacAir, 1.8 GHz Dual-Core Intel Core i5, 8 GB 1600 MHz DDR3.

The rows of Table II are sorted by the number of DUAs. The number of unconstrained DUA ranged from 24.5% for Chart to 37.9% for Codec, with an average of 29.4%. The time to calculate the graphduas varied roughly according to the number of DUAs of the program, with 38s needed for Weka and 27s for Math, the most expensive programs. All the other programs needed fewer than 12s to find all graphduas. Figure 14 plots the number of DUAs and the time to generate graphduas for each program. The cost of graphdua generation in the subsumption analysis varied from 25.9% for Math to 68.3% for Jsoup, being the average 50.5%.

B. Discussion

RQ1 is concerned with the cost (time) to generate a program's graphduas. With the exception of Lang and Chart (shown in bold font in Table II), which were slightly less expensive despite of having more DUAs than the previous program, it took longer to find graphduas when a program had more DUAs. This trend is shown in Figure 14.

Another less evident exception is Math. Weka has almost four times more DUAs, but it is only 1.4 more expensive than Math. The asymptotic cost to find a graphdua is O(E) or O(N), assuming that the number of edges (E) is linear to the

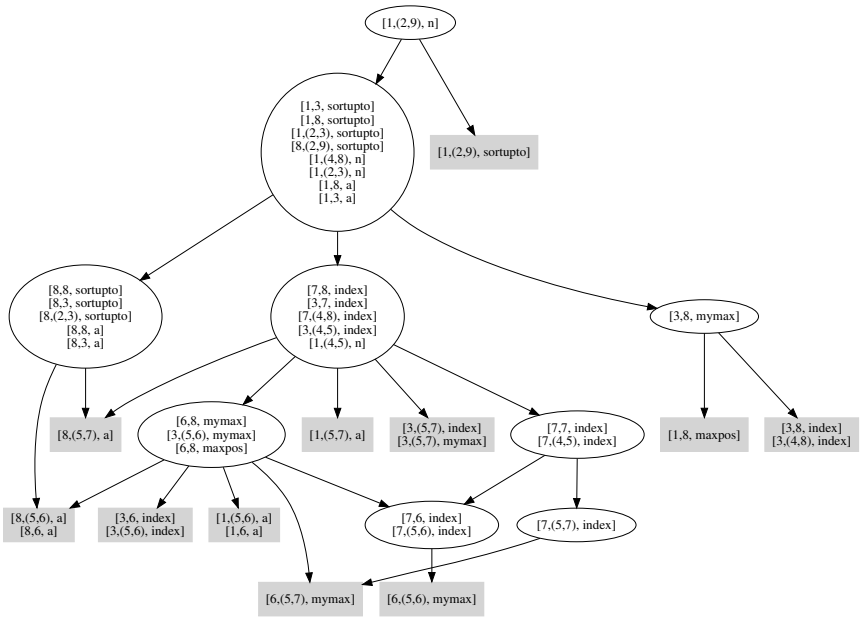

Figure 13. Subsumption graph among DUAs of program Sort

Table II
EMPIRICAL COSTS OF APPLYING THE GRAPHDUA ALGORITHM

Program	**#DUAs**	**Unc.** (%)	**TGra.** (ms)	**TUnc.** (ms)	**Rat.** (%)
Csv	929	31.5	424.5	670.0	63.4
Cli	1291	29.4	533.8	923.2	57.8
JSoup	1866	26.8	599.1	877.3	68.3
Gson	3281	29.7	826.7	1295.8	63.8
J-Xml	3402	26.5	776.0	1237.9	62.7
Mockito	4236	32.1	1121.7	1931.1	58.1
Codec	4446	37.9	2080.9	6778.4	30.7
Compress	6286	28.3	2393.5	5797.4	41.3
Collections	16,937	30.0	2456.9	4556.9	53.9
J-Core	17,653	28.1	2751.3	6394.3	43.0
Time	18,160	31.1	3081.6	5915.2	52.1
JxPath	20,178	29.7	3386.5	6494.9	52.1
Lang	22,290	32.0	3079.2	6087.2	50.6
J-Databind	31,797	26.4	3946.2	6982.3	56.5
Closure	78,068	31.8	11,913.6	32,076.6	37.1
Chart	81,847	24.5	10,004.4	18,420.5	54.3
Math	87,603	25.3	27,010.4	104,246.0	25.9
Weka	337,063	27.4	38,200.9	98,377.5	38.8

number of node (N). As a result, programs with more complex methods tend to be more costly. Two methods in Math have 2197 DUAs and flow graphs with 327 nodes, which appear to be clones of each other. They dominate the cost to generate graphduas for Math.

Overall, most graphduas were calculated quickly, with a very few highly complex methods being exceptions. As a result, hundreds of thousands of graphduas are calculated in tens of seconds. Though they represent a significant amount of the cost of the subsumption analysis, on average 50% and as much as 68%, the subsumption analysis of DUAs is scalable. The most costly program for subsumption analysis, Math, needed around 1min and 40s to find the unconstrained DUAs for all methods because data flow subsumption analysis is more sensitive to the complexity of the programs [15].

RQ2 concerns the usefulness of the graphduas. Column **Unc.** reports the percentage of uncontrained DUAs wrt to the total of DUAs. On average they represent 30%, which represents a potential saving of 70% in testing effort since the tester would need to verify only 30% of DUAs to achieve data flow coverage. We caution, though, the reader that subsumption relationships can be disrupted by infeasible DUAs or due to program interruption.

Differently from previous studies [9], [10], we calculated the unconstrained DUAs taking into account all paths covering a particular DUA. We also used industrial-sized programs. Thus, our results indicate that graphduas work at scale and are useful to reduce the cost of data flow testing.

C. Threats to validity

The main validity threats to our assessment are due to internal and external risks. We used the Defects4J repository to reduce the external threat since its programs are open-source and thus comparable to those developed in industry. Also to tackle the external validity, we added Weka because we observed that mathematical software seemed to be more demanding for the graphdua algorithm.

Our experiment has also an internal validity threat. We implemented algorithms for creating the graphduas, for reading the control and data flow information from bytecode, and the Subsumption Algorithm. To verify the correctness of the algorithms, we checked automatically the subsumption relationship using coverage data for 16 of the 18 subject programs. Nevertheless, we implemented the algorithms using Java and Java Collections; hidden inefficiencies of their data structures might impact the time data.

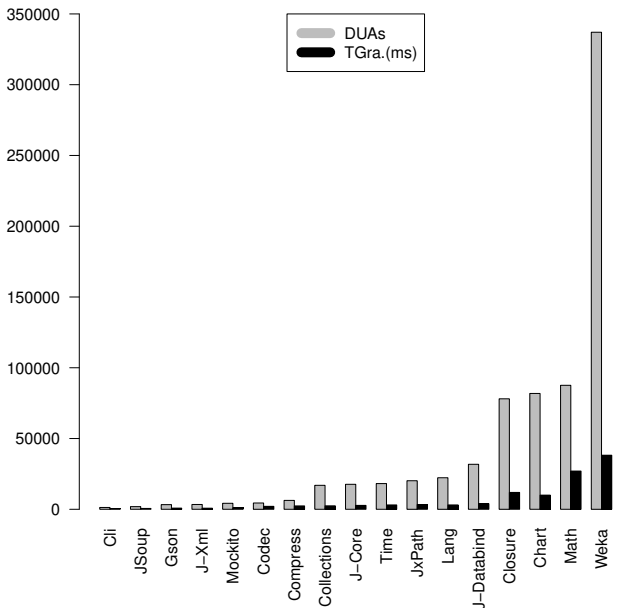

Figure 14. Number of DUAs and time to generate the graphduas

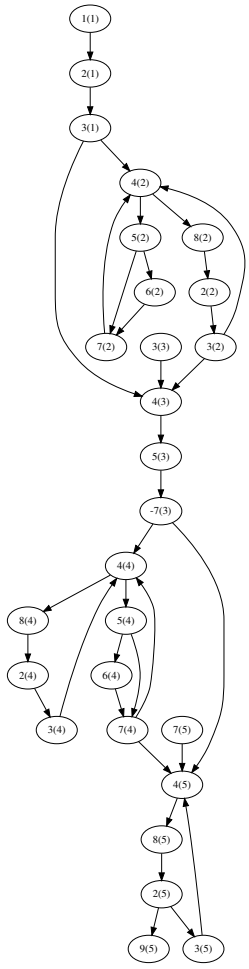

Figure 15. Graphdua generated for DUA (3,(5,7), mymax)

VII. Related work

We have identified three previous attempts to find all paths that cover a particular DUA. One of them is the $G*$ graph [9], [13] from which the graphdua is an extension. The main difference between the graphdua and $G*$ is the number of subgraphs that comprises the graph representation. $G*$ takes into account paths from the start node (s) to the definition node (d_1) (our *SG1*), def-clear paths wrt X_1 from d_1 to node u_1 (our *SG3*), and paths from u_1 to the exit node (e) (our *SG5*). Thus, $G*$ does include paths encoded by sub-graphs *SG2* and *SG4*. Our graphdua also differs from $G*$ by dealing with edge DUAs; $G*$ includes only uses in nodes (see Section IV-D).

As a result, $G*$ misses paths that might negate the DUA subsumption relationship because of an intervening def of the variable. Consider the DUAs D_2 = (1,(5,7), a) and D_1 = (3,(5,7), mymax) in Figure 1. If a test takes the path [3,4,5,7] to cover D_1, the path must have started at node 1, thereby covering D_2. Thus $G*$ finds that D_1 subsumes D_2. However, $G*$ does **not** find path [1,2,3,4,8,2,3,4,5,7], which covers D_1, but because of the definition of a at node 8, is not def-clear wrt to the definition of a at node 1. This is precisely the kind of path that $SG2$ finds. Figure 15 shows that the graphdua of (3,(5,7), mymax) encodes the path that blocks the subsumption of (1,(5,7), a). By not including *back-paths*, $G*$ finds invalid subsumptions among DUAs. The graphdua corrects that flaw.

Su et al. [12] use static analysis to reduce path explosion when using symbolic execution (SE) to generate test inputs or to detect whether DUAs are infeasible. They use dominator analysis to find a set of *cut points* to guide path exploration. Given a DUA D = (d, u, X), its cut points are a sequence of critical control points $c_1, \ldots, c_i, \ldots, c_n$ that must be passed through in succession by any control flow path that cover D. That is, the sequence $c_1, \ldots, d, \ldots, c_i, \ldots, u, \ldots, c_n$ will eventually occur in any path that covers D. The authors developed heuristics based on cut points to select the paths to be symbolically executed.

The dominance relationship only includes control flow information. Thus, even though the cut points must appear in every path that covers D, the nodes occurring between d and c_i $(d, \ldots, c_i)$ and between c_i and u $(c_i, \ldots, u)$ should be checked against possible redefinitions of variable X. Su et al. [12] check for redefinitions during path exploration analysis for symbolic execution. Heuristics can be devised to guide symbolic execution by traversing the graphdua with two advantages: (1) cut points are not needed because they are embedded in the graphdua; and (2) no tracking of redefinition nodes is required because graphdua's sub-graph *SG3* includes only paths between d and u without redefinition of variable X. Additionally, the cost of finding the graphdua and the cut points is the same: O(E) where E is the number of edges of the control flow graph of the program (see Section IV-C).

Jiang et al. [10] also presents an algorithm for DUA subsumption. They frame the paths covering a particular DUA by applying the concept of *def-use order* (introduced by Santellices and Harrold [17]). A DUA D = (d, u, X) is in

def-use order if one of the following conditions hold: (1) node d is not reachable from node u; (2) node d dominates node u; or (3) node u post-dominates node d. As a result, a DUA D is in def-use order if, whenever D is covered, node d is guaranteed to occur before node u. However, if the def-use order is obtained by conditions (2) or (3), the use node u can reach the def node d. If so, the very same paths that the $G*$ misses will be missed by the def-use order. Consider the DUA D = (3,6, mymax) of the Sort program. It is in def-use order due to condition (1) (def node 3 dominates use node 6), but node 6 reaches node 3. There is a path from node 3 to node 3 that is not encoded by the def-use order that might block the subsumption of DUAs.

With respect to the previous approaches for representing data flow coverage, the graphdua finds back-paths that avoid finding incorrect subsumption relationships, deals with all types of DUAs, cleans up dangling paths, and can guide the path exploration for symbolic execution without needing to track redefinition nodes.

In a separate thread of research, the same concept of subsumption has been used to identify *minimal sets* of mutants [18]–[22]. In the terminology used in those papers, a set of mutants is *minimal* if all test sets that kill all mutants in the minimal set are guaranteed to also kill all mutants. The first two papers [18], [19] introduced the theoretical concept, presented the mutant subsequent graph (MSG), and showed how to approximate the "true" MSG dynamically. Subsequent papers showed how to approximate the MSG statically [20], showed that redundant (constrained) mutants effect the mutation score [22], and used minimal mutation to identify a significant weakness in selective mutation [21]. The minimal mutant set is directly analogous to the spanning set in data flow, with a significant difference being that the minimal mutant set is uncomputable.

VIII. Conclusions

This paper presents a new graph representation, called *graphdua*, for data flow testing coverage. The graphdua is used to cover definition-use associations (DUA) [3]. We used a running example to informally present the graphdua's main property, which is that it includes all paths that cover a particular DUA D. The graphdua extends previous solutions [9], [13] to fix omissions of paths that may lead to incorrect results.

Additionally, we present experimental data suggesting that graphduas are generated at scale for industry-sized programs to efficiently find data flow subsumptions. Data flow subsumption analysis may reduce the cost of data flow testing by up to 70%. Previous studies found the subsumption relationship in part manually [9] and in programs with fewer DUAs [10].

Our next steps will be to investigate the use of the graphduas in generating input data for real-world applications and for feasibility analysis of data flow associations.

Acknowledgment

Marcos Lordello Chaim was supported by grant #2019/ 21763-9, São Paulo Research Foundation (FAPESP).

References

[1] J. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 347–354, 1983.

[2] S. C. Ntafos, "On required element testing," *IEEE Trans. Software Eng.*, vol. 10, no. 6, pp. 795–803, 1984.

[3] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, Apr. 1985.

[4] H. Ural and B. Yang, "A structural test selection criterion," *Information Processing Letters*, vol. 28, pp. 157–163, 1988.

[5] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *16th International Conference on Software Engineering*, ser. ICSE, 1994, pp. 191–200.

[6] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *Proc. of the ACM SIGSOFT Foundations of Software Engineering Conference*, ser. FSE '98, 1998, pp. 153–162.

[7] T.-B. Dao and E. Shibayama, "Security sensitive data flow coverage criterion for automatic security testing of web applications," in *Engineering Secure Software and Systems*, ser. ESSoS, 2011, pp. 101–113.

[8] H. Hemmati, "How effective are code coverage criteria?" in *International Conference on Software Quality*. IEEE, 2015, pp. 151–156.

[9] M. Marré and A. Bertolino, "Using spanning sets for coverage testing," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 974–984, 2003.

[10] S. Jiang, J. Chen, Y. Zhang, J. Qian, R. Wang, and M. Xue, "Evolutionary approach to generating test data for data flow test," *IET Software*, vol. 12, no. 4, pp. 318–323, 2018.

[11] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2013, pp. 370–379.

[12] T. Su, Z. Fu, G. Pu, J. He, and Z. Su, "Combining symbolic execution and model checking for data flow testing," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 654–665.

[13] M. Marré, "Program Flow Analysis for Reducing and Estimating the Cost of Test Coverage Criteria," Ph.D. dissertation, Dep. de Computacion, FCEyN – Universidad de Buenos Aires, Argentina, 1997.

[14] A. Bertolino and M. Marré, "Automatic generation of path covers based on the control flow analysis of computer programs," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 885–899, Dec 1994.

[15] M. L. Chaim, K. Baral, J. Offutt, M. Concilio, and R. P. A. Araujo, "Efficiently finding data flow subsumptions," in *14th IEEE International Conference on Software Testing, Validation and Verification, ICST 2021, Porto de Galinhas, Brazil*, April 2021.

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for Java programs," in *International Symposium on Software Testing and Analysis, ISSTA*, July 2014, pp. 437–440.

[17] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2007, pp. 343–352.

[18] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Cleveland, OH, March 2014, pp. 21–30.

[19] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Tenth IEEE Workshop on Mutation Analysis (Mutation 2014)*, Cleveland, OH, March 2014.

[20] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *Eleventh IEEE Workshop on Mutation Analysis (Mutation 2015)*, Graz, Austria, April 2015.

[21] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökće, "Analyzing the validity of selective mutation with dominator mutants," in *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, Seattle Washington, USA, November 2016.

[22] B. Kurtz, P. Ammann, J. Offutt, and M. Kurtz, "Are we there yet? How redundant and equivalent mutants affect determination of test completeness," in *Twelfth IEEE Workshop on Mutation Analysis (Mutation 2016)*, Chicago Illinois, USA, April 2016.