Efficiently Finding Data Flow Subsumptions

Marcos Lordello Chaim
University of Sao Paulo
Sao Paulo, SP, Brazil
Email:chaim@usp.br

Kesina Baral, Jeff Offutt
George Mason University
Fairfax, VA, USA
Email:{kbaral4,offutt}@gmu.edu

Mario Concilio and Roberto P. A. Araujo
University of Sao Paulo
Sao Paulo, SP, Brazil
Email:{mario.neto,roberto.araujo}@usp.br

***Abstract*—Data flow testing creates test requirements as definition-use (DU) associations, where a *definition* is a program location that assigns a value to a variable and a *use* is a location where that value is accessed. Data flow testing is expensive, largely because of the number of test requirements. Luckily, many DU-associations are redundant in the sense that if one test requirement (e.g., node, edge, DU-association) is covered, other DU-associations are guaranteed to also be covered. This relationship is called *subsumption*. Thus, testers can save resources by only covering DU-associations that are not subsumed by other testing requirements. Although this has the potential to significantly decrease the cost of data flow testing, finding subsumption among DU-associations is quite difficult. Previous solutions are costly and contain subtle flaws that sometimes lead to incorrect results. We model the data flow testing subsumption as a data flow analysis framework, allowing us to use efficient algorithms that quickly discover data flow subsumption relationships. Experimental data suggest that the framework and algorithm can reduce the cost of data flow testing and will work at scale.**

***Index Terms*—Software testing, Structural testing, Data flow testing, Subsumption relationship, Data flow analysis frameworks, Algorithms**

I. Introduction

Software engineers test software to find faults and assess the quality of software under test. The number of possible inputs is effectively infinite for most programs, thus we cannot completely test the program. Thus, testers try to use a reasonable and cost-effective number of tests while also maximizing the test suite's fault detection capability. One method is to use *test requirements* to cover specific parts of software artifacts. Test requirements can be derived from various software artifacts, including source code [1], [2], graphs [3], [4], and the software's input space [5]. *Coverage criteria* provide a systematic way to generate test requirements [6] and can be used to assess the test adequacy and test quality.

Graph criteria are widely used for testing. Graphs can be generated from source code, state machines, software specifications, and use cases. Several related test criteria have been proposed based on the control flow analysis and data flow analysis on graphs. Control flow analysis focuses on testing the flow of program control during execution. Data flow analysis evaluates the flow of data values during program execution. The goal of data flow testing is to exercise pairs of definitions and uses of variables, known as *definition-use associations*. It is also referred to as *du associations, def-use associations, du-pairs or def-pairs*. In this paper, we use the term *DU-associations*, or simply DUA.

Studies have shown that data flow testing (DFT) is comparable to that with control flow testing (CFT) [7] [8] [9]. Hemmati showed that du associations coverage is able to find faults that control flow coverage criteria do not [9]. Hemmati found that 79% of faults control flow criteria did not find were found by DFT. This shows that DFT could add value when used alongside CFT. Additionally, DFT coverage supports security verification [10] and fault localization [11], [12].

Control flow analysis is widely available in commercial test coverage calculation tools like Clover (www.atlassian.com/software/clover/), JaCoCo (www.eclemma.org/jacoco/), Cobertura (cobertura.github.io/cobertura/), and even some integrated development environments (IDE) like IntelliJ (www.jetbrains.com/idea/), and is commonly used by professional programmers. However, data flow analysis is not widely used. This low usage of DFT is in part due to the large number of test requirements for DFT, each of which adds to the expense. Table I illustrates this issue by listing programs with their sizes in terms of lines of code (LOC) and number of DUAs, along with other metrics and data. Math, the fifth from the bottom of the table, has 54,518 LOC. The all-uses criterion [13], which requires every use to be reached at least once, has 87,603 DUA requirements. This is 60% more than the LOC.

One approach to reduce DFT's cost is to exploit the subsumption relationship among testing requirements (TR) [14]–[16]. A TR tr_1 (e.g., a DUA D_1) *subsumes* another test requirement tr_2 (e.g., a DUA D_2) if every test path that satisfies tr_1 also satisfies tr_2 [14]. A minimal subset of TRs that subsumes every other TR is called a *spanning set* and its elements are referred to as *unconstrained* test requirements [14]. Identifying the spanning set establishes a priority order among TRs that allow testers to focus on just the unconstrained requirements, saving time and effort. Santelices and Harrold [15] make use of the subsumption of TRs of different coverage criteria to reduce the cost of code instrumentation. They infer which DUAs have been covered or *conditionally* covered based on the edges that a test case visits. The subsumption of DUAs with respect to nodes (DUA-node subsumption) and edges (DUA-edge subsumption) can save resources by focusing on DUAs not covered by node and edge coverage or by tracking only nodes or edges at run-time.

Discovering data flow testing subsumption relationships (DFTS), though, has been an elusive goal. The current al-

TABLE I
PROGRAM METRICS AND DATA FLOW SUBSUMPTION DATA

Program	LOC	**Methods w. DUAs**	Methods exec.	DUAs	%-DUA -Node	%-DUA -Edge	%-Unc. DUAs	t-DUA -Node (ms)	t-DUA -Edge (ms)	t-Unc. DUAs (ms)
Csv	602	40	37	929	57.3	70.3	31.5	343.8	334.7	670.0
Cli	1107	60	54	1291	69.6	83.0	29.4	454.6	443.2	923.2
Codec	1946	109	92	4446	38.0	48.4	37.9	678.9	656.4	6778.4
Jsoup	2046	136	119	1866	80.0	92.9	26.8	584.9	595.3	877.3
J-Xml	3084	179	124	3402	75.7	87.9	26.5	715.3	730.5	1237.9
Compress	4974	217	116	6286	58.7	67.0	28.3	1126.1	1169.5	5797.4
Gson	3840	226	191	3281	76.6	88.4	29.7	849.6	817.5	1295.8
Mockito	7468	400	*	4236	74.6	90.0	32.1	1327.1	1339.2	1931.1
J-Core	10,978	563	383	17,653	59.5	72.7	28.1	1500.2	1538.2	6394.3
JxPath	14,699	800	691	20,178	66.9	84.4	29.7	1763.4	1783.0	6494.9
Lang	15,270	1157	*	22,290	62.8	73.9	32.0	2178.5	2085.0	6087.2
Time	19,672	1182	1010	18,160	69.2	80.9	31.1	2264.7	2338.9	5915.2
Collections	18,156	1311	1094	16,937	68.8	81.9	30.0	2331.4	2322.0	4556.9
J-DataBind	27,274	1737	1266	31,797	73.8	85.6	26.4	2834.3	2881.4	6982.3
Math	54,518	2415	1999	87,603	57.9	64.2	25.3	12,141.9	11,815.7	104,246.0
Chart	68,346	3219	2151	81,847	72.8	82.2	24.5	6390.8	6301.0	18,420.9
Closure	61,177	3696	3241	78,068	67.9	83.7	31.8	6551.1	6358.1	32,076.6
Weka	216,781	11,315	1964	337,063	59.9	70.9	27.4	22,361.8	22,317.7	98,377.5
Total/Avg.	531,938	28,758	14,532	737,333	66.1	78.2	29.4	–	–	–

gorithms are costly, being linear [15] in the number of DUAs for DUA-edge and quadratic [14], [16] for DUA-DUA subsumption. This cost hampers its application for industry-size systems. Furthermore, some algorithms [14], [16] might miss paths that would block the subsumption of DUAs, leading to incorrect results. We address both cost and omissions regarding data flow subsumption discovery.

We present a novel and efficient approach to tackle data flow subsumptions. It models the problem of finding the *local DUA-node* subsumption; that is, those DUAs that are covered whenever a particular point p (e.g, a node) of a program is reached, as a data flow analysis framework [17], [18]. Using the local DUA-node subsumption, one can efficiently discover the subsumption of DUAs with respect to nodes, edges, and other DUAs. We show experimental data suggesting that data flow subsumptions are effective and can be found at scale.

Efficient data flow subsumption discovery, applicable to industry-sized programs, can reduce the number of test requirements to be verified or tracked and better estimate test set completeness. This paper starts with background in data flow testing in section II. We then describe data flow subsumptions in section III, followed by our solution to solve the local DUA-node subsumption in section IV. Algorithms to find other data flow subsumptions are described in section V, followed by experimental analysis in section VI. Related work is discussed in section VII, and conclusions are in section VIII.

II. BACKGROUND

Graph based testing criteria use graph abstractions of the software under test to generate tests. A graph can be defined as $G(N, E, s, e)$, where N is a set of nodes, E is a set of edges, s is the start node and e is the exit node. A node (n) can represent a single statement of the program or a sequence of statements. For our purposes, we consider a sequence of statements, also known as a *basic block*, to be a node. An edge represents potential control flow from one node to another, written as (n_i, n_j), $n_i \neq n_j$, where node n_i is the *predecessor* and node n_j is the *successor*. Graphs extracted from a program must have at least one start node and exit node for it to be useful to generate tests. A program can have multiple entry and exit points.

A *path* is a sequence of nodes $(n_i, \ldots, n_k, n_{k+1}, \ldots, n_j)$, where $i \leq k < j$, such that $(n_k, n_{k+1}) \in E$. A *test path* is a special path that starts from a start node s and ends at an exit node e. A test path represents the execution of one or more test cases. A *side-trip* is a sub-path that starts and ends at the same node (a loop).

Figure 1 presents a program that finds the maximum element in an $array$ of integers [19] and Figure 2 presents its control flow graph. The numbers at the start of each line of code in Figure 1 indicate the line's corresponding node in the graph.

Graph coverage criteria come in two forms, control flow coverage criteria and data flow coverage criteria. *Control flow coverage criteria* cover the structure of the graph, including nodes, edges, and specific sub-paths. *Data flow coverage criteria* evaluates the flow of data values during program execution. Data flow coverage criteria provide test requirements for data flow testing by focusing on definitions and uses of variables. A *definition*, or *def*, is a program location where a value is assigned to a variable. A *use* is a location where the variable is referenced. The graph shown in Figure 2 is annotated with defs and uses associated with its nodes and edges.

Data flow testing focuses on the flow of data values from definitions to uses. A variable can be used to compute a value

```
/* 0 */ int max(int array[], int length)
/* 0 */ {
/* 0 */     int i = 0;
/* 0 */     int max = array[++i];
/* 1 */     while (i < length)
/* 1 */     {
/* 3 */         int rogue = 1;
/* 3 */         if (array[i] > max)
/* 5 */             {
/* 5 */              max = array[i];
/* 5 */              print(rogue);
/* 5 */             }
/* 4 */         i = i + 1;
/* 4 */     }
/* 2 */     return max;
/* 6 */ }
```

Fig. 1. Example program Max

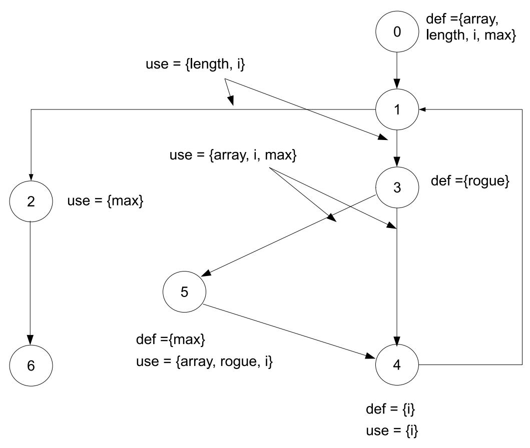

Fig. 2. Annotated flow graph for max.eps

or in a predicate. Value computations are associated with nodes and predicate computations are associated with edges.

A *definition-clear* (*def-clear*) path with respect to a variable x is a path where x is not redefined along the path. A *du-path* is a simple sub-path (all nodes are different except the first and last nodes) that is def-clear with respect to (wrt) variable x. A du-path with side-trips wrt variable x allows side-trips that are also def-clear wrt x.

Data flow test criteria define test requirements as specific du-paths that must be covered. A *du association* set $D(d, u, x)$ is a set of du-paths and du-paths with side-trips wrt variable x that start at node d and end at node u. If the use is on an edge (u', u), the DU-associations set is written as $D(d, (u', u), x)$. Several data flow testing criteria have been invented [13], [20]–[22]. In this paper, we focus on the *all-uses* criterion proposed by Rapps and Weyuker [13].

The all-uses criterion requires that at least one du-path (or du-path with side-trips) is executed, or *toured*, for every DU-associations (DUAs) set, that is, each def reaches each use at least once. If a test set T includes a du-path (or du-path with side-trips) for each DUA $D(d,\ u,\ x)$ or $D(d,\ (u',\ u),\ x)$, it is said to be *adequate* for the all-uses criterion for program P since all required DUAs were *covered.*

TABLE II
ALL-USES TEST REQUIREMENTS FOR PROGRAM MAX

All-uses		
(0, (3,5), array)	(0, (3,4), array)	(0, 5, array)
(0, (1,3), length)	(0, (1,2), length)	(0, (1,3), i)
(0, (1,2), i)	(0, (3,5), i)	(0, (3,4), i)
(0, 4, i)	(0, 5, i)	(0, 2, max)
(0, (3,5), max)	(0, (3,4), max)	(5, 2, max)
(5, (3,5), max)	(5, (3,4), max)	(4, (1,3), i)
(4, (1,2), i)	(4, (3,5), i)	(4, (3,4), i)
(4, 4, i)	(4, 5, i)	(3, 5, rogue)

Table II shows the test requirements for the all-uses coverage criterion on the program from Figure 1. The triplet $(0, (3, 5), array)$ indicates there is a def of variable $array$ at node 0, which reaches a use at edge $(3, 5)$. Table II shows that the all-uses criterion generates many test requirements, 24, even for a relatively small method with only 6 nodes. This makes data flow testing expensive. This paper uses subsumption to reduce the cost of data flow testing [14].

III. DATA FLOW TESTING SUBSUMPTION (DFTS)

Subsumption is traditionally used to compare testing criteria. A test criterion C_1 *subsumes* criterion C_2 if and only if every set of execution paths P that satisfies C_1 also satisfies C_2 [13], [23]. Satisfying the subsuming test criterion guarantees that the subsumed criterion is also satisfied. However, the subsumption relation may not hold if some test requirements of the subsuming criterion are infeasible. Additional strategies might be needed to reestablish subsumption [24].

Marré and Bertolino [14] explored subsumption relationships among testing requirements (TR) of the same criterion C. The intuition is that if TR tr_1 is subsumed by tr_2, then tr_1 is easier to cover than tr_2, resulting in an ordering that exists among TRs [25]. A minimal subset of TRs that subsumes every other TR is called a *spanning set* and its elements are referred to as *unconstrained* test requirements [14]. Thus, testers can save resources by only covering TRs that are guaranteed to cover other TRs. Santelices and Harrold [15] compare TRs from different criteria, in particular du associations (DUAs) subsumed by edges. In doing so, testers can focus only on DUAs not subsumed by edges to enhance a test suite.

In a separate thread of research, the same concept of subsumption has been used to identify and approximate *minimal sets* of mutants [26]–[30]. Killing a minimal set of mutant guarantees that all non-equivalent mutants would be killed by the same tests, but with substantially less effort than killing all mutants. The first two papers [26], [27] introduced the theoretical concept, presented the mutant subsumption graph (MSG), and showed how to approximate the "true" MSG

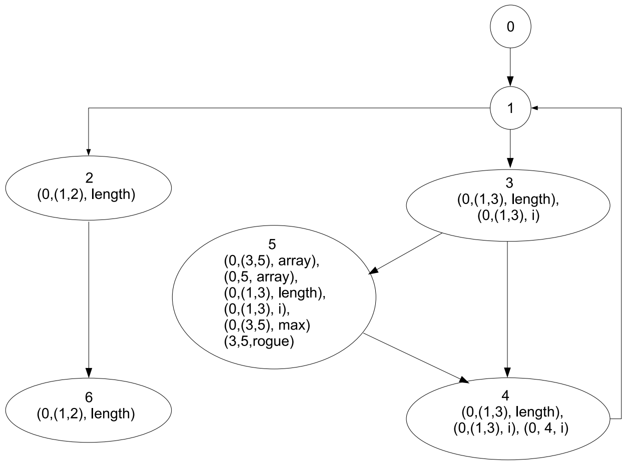

Fig. 3. Local DUA-node subsumption for program Max

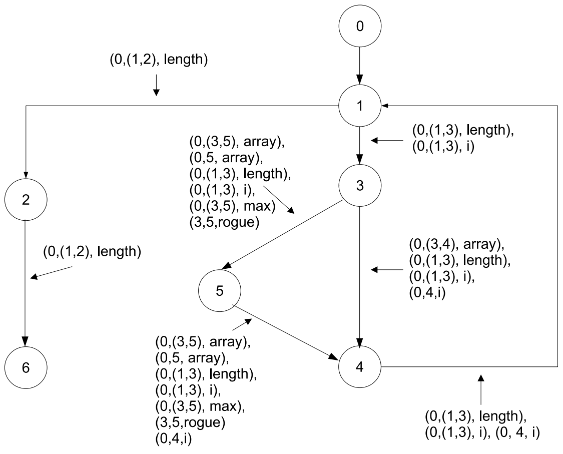

Fig. 4. DUA-edge subsumption for program Max

dynamically. Subsequent papers showed how to approximate the MSG statically [28], showed that redundant (constrained) mutants effect the mutation score [30], and used minimal mutation to identify a significant weakness in selective mutation [29]. The minimal mutant set is directly analogous to the spanning set in structural criteria, with a significant difference being that the minimal mutant set is uncomputable.

The following subsections discuss three data flow testing subsumption relationships: (a) DUA subsumption by nodes, (b) DUA subsumption by edges, and (c) DUA subsumption by other DUAs.

A. DUA-node subsumption

DUA-node subsumption identifies DUAs that are guaranteed to be covered if a specific node in the graph is visited. More formally, DUA $D(d, u, X)$ or $D(d, (u', u), X)$ is *DUA-subsumed* by node n if D is covered on all test paths that visits node n and reaches the exit node. The set of DUAs subsumed by node n is the set of all DUAs that are covered by all test paths that visit n.

We find it necessary to allow for interrupted execution, for example exceptions or other program aborts. Thus, we distinguish between *local DUA-node subsumption,* which is the set of DUAs covered by all paths that **reach** n, and *global DUA-node subsumption,* which is the set of DUAs covered by all test paths that both reach n and then continue to the exit node. The set of globally subsumed DUAs include DUAs that are DUA-node subsumed by nodes that appear on all paths from node n to the exit node.

Figure 3 shows the DUA-subsumption sets for the Max method from Figures 1 and 2. Each node contains the locally subsumed DUAs. For example, if node 5 is reached, the definition of $array$ at node 0 is guaranteed to have reached the use on edge (3,5).

Node n_i *dominates* node n_j if every path from the start node to n_j includes n_i [17]. Node n_j *post-dominates* node n_i if any path from n_j to the exit node includes n_i. A node dominates itself but does not post-dominate itself [16]. In the absence of early program termination, node 5 is post-dominated by nodes 4, 1, 2, and 6. When they are visited by a test path, the set of DUAs that are globally subsumed by node 5 includes the six DUAs listed in node 5, plus DUAs (0, 4, i) from node 4 and (0, (1,2), $length$) from node 2. Thus, node 5 locally subsumes six DUAs and globally subsumes eight DUAs.

Node 5 is the only unconstrained node for program Max. This means that eight of 24 DUAs will be covered if all nodes of the Max program are visited. Thus, node coverage would result in a data flow coverage of 33%.

B. DUA-edge subsumption

DUA-edge subsumption addresses DUAs that are guaranteed to be covered if edges are visited. A DUA $D(d, u, X)$ or $D(d, (u', u), X)$ is *DUA-subsumed* by edge (n', n) if D is covered on all test paths that visit (n', n). The set of DUAs subsumed by edge (n', n) is the set of all DUAs that are covered by all test paths that visit (n', n).

Figure 4 presents the control flow graph of Max, annotated with the DUAs that are locally subsumed by each edge. The global DUA-edge subsumption sets include DUAs subsumed by nodes that post-dominate n. For example, the seven DUAs listed on edge (5,4) are locally subsumed; the global set also includes (0, (1,2), $length$) from edge (1,2).

Edges (3,5) and (3,4) are unconstrained for the all edges criterion, thus visiting them will ensure all other edges are visited. Edge coverage will also ensure that nine unique DUAs are toured (37.5%). That is one more than node coverage ensures, specifically, the DUA (0, (3,4), $array$), which is subsumed by edge (3,4).

C. DUA-DUA subsumption

DUA-node and DUA-edge subsumption relate different criteria–node coverage and edge coverage, with all-uses coverage. DUA-DUA subsumption relates test requirements within the same criterion, all-uses coverage. A DU-association D_1 subsumes another DUA, D_2, if every test that covers D_1 is guaranteed to also cover D_2.

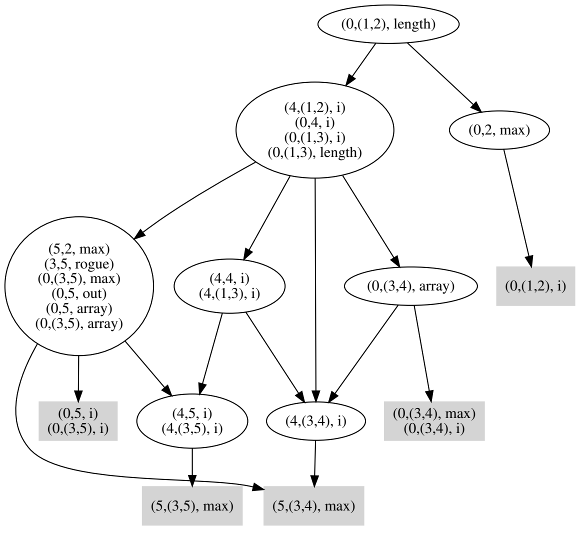

Fig. 5. DUA-DUA subsumption for example program Max

Formally, $D_1(d_1, u_1, X_1)$ subsumes $D_2(d_2, u_2, X_2)$ ($D_2 \rightarrow D_1$), if every test path that contains a def-clear path with respect to X_1 between d_1 and u_1 also contains a def-clear path with respect to X_2 between d_2 and u_2. We use the notation $D_2 \rightarrow D_1$ to indicate that D_1 subsumes D_2.

As with DUA-node and DUA-edge subsumption, D_1 locally subsumes D_2 if D_2's def-clear path with respect to X_2 ends before D_1's def-clear path with respect to X_1 ends.

In the example program, if a test ensures the def of i at node 4 reaches the use of i at edge (3,4), we are guaranteed that the def of $array$ at node 0 also reaches the use of $array$ at (3,4). Thus, (0, (3,4), $array$) $\rightarrow$ (4, (3,4), i). The subsumption relationship is not symmetric, however, because a test that covers (0, (3,4), $array$) might not cause the def of i at node 4 to reach the edge (3,4), so (0, (3,4), i) does **not** subsume (4, (3,4), i).

The graph in Figure 5 shows subsumption among the DUAs of Max. If two DUAs D_1 and D_2 are in the same node, then $D_1 \rightarrow D_2$ and $D_2 \rightarrow D_1$. If D_2 is in a node with an edge to a node that has D_1, that means that $D_2 \rightarrow D_1$. This graph is very similar to the mutant subsumption graph [27].

This DUA subsumption graph (DSG) allows us to directly find a minimal set of DUAs that, if covered, implies that all DUAs are covered. This minimal set of DUAs is called the *spanning set*. The leaves of the DSG, shown with shaded rectangles in Figure 5, give the spanning set of DUAs for program Max. A DUA in the spanning set is an *unconstrained* DUA. The spanning set is not unique because some leaves have more than one DUA. When that happens, any of the DUAs in the node could be included in the spanning set.

Two of the five leaf nodes in Figure 5 have two DUAs. Thus Max has four possible sets of unconstrained DUAs, one being $\{$(0, 5, i), (5, (3,5), max), (5, (3,4), max), (0, (3,4), max), (0, (1,2), i)$\}$.

IV. Finding the local DUA-node subsumption

Data flow analysis frameworks are created to solve data flow analysis problems such as reaching definitions, live-variables, and available expressions [18]. They determine *facts* that are valid at the entry or exit of a program point p whenever p is reached [31]. We use the data flow analysis framework *Data Flow Subsumption Framework* (DSF) [32] to find local DUA-subsumption.

DSF finds the set of DUAs already *covered* or *available* to be covered at the entrance (set **IN**(n)) and at the exit (set **OUT**(n)) of a node n along all paths that reach n (the *facts*). A DUA is available to be covered if its def node was previously toured in the path and there is no re-definition of its variable in the nodes that were subsequently toured.

This section presents Subsumption Algorithm (SA) that uses the DSF framework to find the local DUA-node subsumption, and then analyzes SA's complexity.

A. Subsumption algorithm

The Subsumption Algorithm (SA), presented in Algorithm 1, uses DSF to find the local DUA-node subsumptions. SA adapts a classical data flow analysis algorithm to find the values of sets **IN**(n) and **OUT**(n). The final output of the algorithm is **Covered**(n), a set of DUAs that are covered at node n when n is reached from any path that begins at the start node s of a flow graph G. **Covered**(n) gives the *local* DUA-node subsumptions.

Lines 1-6 in Algorithm 1 initialize the algorithm's working sets. Initially, **IN**(s) is empty since there is no DUA covered or available to be covered, **OUT**(s) contains the DUAs that become available for coverage after s is traversed (see the definition of **Born**(n) below), and **Covered**(s) is also empty because no DUA is covered at s. All other nodes are initialized with **OUT** and **Covered** equal to all DUAs.

Lines 9-12 represent the DFS transfer functions. When a node n is reached, transfer functions calculate the *fact* that is valid at the entrance and exit of node n. To define the transfer functions of DSF, we associate nodes of the flow graph with sets, as introduced by Chaim and Araujo [19]. These sets are defined as follows:

Let $n \in N$ be a node in flow graph $G(N, E, s, e)$ of a program P and (d,u,X) or (d,(u',u),X) a DUA required to test P according to the all-uses criterion.

Born(n) : set of DUAs (d,u,X) or (d,(u',u),X) s.t. $d = n$.

Disabled(n) : set of DUAs (d,u,X) or (d,(u',u),X) s.t. X is defined in n and $d \neq n$.

PotCovered(n) : set of DUAs (d,u, X) or (d,(u',u), X) s.t. $u = n$.

Sleepy(n) : set of DUAs (d,(u',u), X) s.t. $u' \neq n$.

Born(n) sets are similar to the **gen**(n) and **e_gen**(n) sets of the reaching definitions and available expressions problems [33]. They represent DUAs that are *born* because the node where their variable is assigned has been toured. **Disabled**(n) is analogous to the sets **kill**(n) and **e_kill**(n) of the same data flow analysis problems, since they contain the DUAs that are *killed* after n was traversed.

```
Input: Flow graph $G(N, E, s, e)$ of program $P$; sets
       Disabled($n$), Sleepy($n$), PotCovered($n$), and
       Born($n$), all DUAs required to test $P$
Output: Covered($n$), set for every node $n$
1  IN($s$) = $\varnothing$ where $s$ is the start node;
2  OUT($s$) = Born($s$) where $s$ is the start node;
3  Covered($s$) = $\varnothing$ where $s$ is the start node;
4  for each node $n$ other than the start node $s$ do
5      OUT($n$) = all DUAs of program $P$;
6      Covered($n$) = all DUAs of program $P$;
7  while changes to any OUT occur do
8      for each node $n \in N$ do
9          IN($n$) = $\bigcap_{p \in PRED(n)}$ OUT($p$);
10         CurSleepy = $\bigcup_{p \in PRED(n) \text{and} (p,n) \text{is not a back edge}}$
            Sleepy($p$);
11         Covered($n$) = $\bigcap_{p \in PRED(n)}$ Covered($p$) $\bigcup$
            [(IN($n$) - CurSleepy) $\bigcap$ PotCovered($n$)];
12         OUT($n$) = Born($n$) $\bigcup$ [IN($n$) - Disabled($n$)]
            $\bigcup$ Covered($n$);
13 for each node $n \in N$ do
14     IN($n$) = $\bigcap_{p \in PRED(n)}$ OUT($p$);
15     CurSleepy = $\bigcup_{p \in PRED(n) \text{and} (p,n) \text{is not a back edge}}$
        Sleepy($p$);
16     Covered($n$) = $\bigcap_{p \in PRED(n)}$ Covered($p$) $\bigcup$ [(IN($n$)
        - CurSleepy) $\bigcap$ PotCovered($n$)];
17 return Covered($n$) for every node $n$
```

Algorithm 1: Subsumption algorithm

SA also needs sets **PotCovered**(n) and **Sleepy**(n). **PotCovered**(n) represents those DUAs that can *potentially* be covered when a node is traversed. If a DUA is available for coverage when n is reached and it belongs to **PotCovered**(n), then it will be covered after n is traversed.

A node may be reached from multiple paths. We cannot guarantee that a particular edge DUA has been covered since we cannot predict which path reached node n. Thus, we use a *sleepy* DUA set to identify which edge DUAs are guaranteed to be covered. After touring node n, we can say that edge DUAs with uses on edges starting at node n will be covered. The other edge DUAs are called *sleepy* at n. Hence, the **Sleepy**(n) set blocks the edge DUAs (d,(u',u),X) from being covered after node n is traversed, if $u' \neq n$. For example, after touring node 3, we know that edge DUAs with use on edge (3,5) or (3,4) will be covered. So the other edge DUAs of the program are in the set **Sleepy**(3). This concept of sleepy is used to calculate the set **CurSleepy**.

CurSleepy is the union of the DUAs that are blocked after a predecessor p of n is toured, if (p,n) is not a back edge. Edge (p,n) is a back edge if node n dominates node p [33]. In Figure 2, (4,1) is a back edge. **CurSleepy** is used to block edge DUAs from being covered when we cannot predict the path that reached a node n. However, when (p,n) is a back edge, we know that n is always toured before p is toured so that it cannot block other edge DUAs from being covered at n. For example, two paths could reach node 4: (0,1,3,5,4) and (0,1,3,4). To identify the edge DUAs that will definitely be covered at node 4, we generate the sleepy set of node 4's predecessors, **Sleepy**(5) and **Sleepy**(3). There is no DUA with a use in edge (5,4) to be excluded from the **Sleepy** set, hence all DUAs are in **Sleepy**(5). Thus, **CurSleepy** at node 4 is the union of **Sleepy**(3) and **Sleepy**(5), that is, all edge DUAs. The **CurSleepy** set blocks all edge DUAs as not guaranteed to be covered at node 4.

The value of **IN**(n) is found by intersecting **OUT** sets of the predecessors of n on line 9. The transfer function on line 10 calculates edge DUAs that cannot be covered at node n.

The transfer function on line 11 finds the DUAs that are covered by all paths that reach node n. It has two parts that are combined via union. The first part of line 11 intersects **Covered** sets of the predecessors of n. Thus, node n will *inherit* only DUAs that were previously covered in all paths that reach it. The second part of line 11 calculates the DUAs covered at node n. **IN**(n) has the DUAs that were covered and available to be covered in all paths that reach n according to line 9, **CurSleepy** has the edge DUAs that are blocked at node n, and **PotCovered**(n) contains DUAs that might be covered at n if they are in **IN**(n). The remaining DUAs after these operations, plus the DUAs covered in previously toured nodes, give the DUAs covered at node n.

Finally, the transfer function in line 12 determines the **OUT**(n) sets–that is, the DUAs that are *forwarded* in the data flow analysis. They are calculated in three parts that are unioned together. The first part is the **Born**(n) set, which contains the DUAs that become available for coverage at node n; that is, their variable was assigned at n. The second part contains the DUAs that are available in **IN**(n) and *survive* node n because they do not belong to **Disabled**(n). The last set added in Line 12 is the set of DUAs covered at n (**Covered**(n)). All these DUAs are forwarded to the node's successors in the data flow analysis.

Lines 13-16 update the **Covered**(n) sets. **OUT**(n) has already converged to its final values after leaving the while-loop at Line 7, but **Covered**(n) needs to be updated with the final values of **OUT**(p).

B. Complexity

The complexity of SA is dominated by the number of iterations needed to finish the while-loop in line 7. In the worst case, the cost of SA is the product of the number of DUAs and the number of nodes in the flow graph. However, SA shares characteristics with other practical data flow analysis problems, including reaching definitions and available expressions. The fact at each node (the covered or available DUAs) propagates along cycle-free paths.

If the nodes are visited in a depth-first order (reverse postorder [17]) in line 8, the information is first propagated through the cycle-free paths. Using this approach, the number of iterations will be no greater than the depth of the nested loop in the program [33] plus 2. These loops tend to be limited to a small constant [34], [35]. SA requires yet another visit to each node to update the **Covered** sets, which will require one more visit to every node of the flow graph.

SA also finds the dominance relationship to determine the back edges. Luckily, the dominance relationship is also modeled as a data flow analysis problem with the same property of propagating its fact (the dominator nodes) along cycle-free paths. Thus, the dominance relationship is found at the same cost. Therefore, the cost of SA for most programs tends to be linear in the number of nodes in its flow graph.

V. Finding data flow subsumptions

This section shows how to use the Subsumption Algorithm (SA) algorithm to find DUA-node, DUA-edge, and DUA-DUA subsumption. We also explore the cost.

A. DUA-node subsumption

Section III presents the DUA-node subsumption algorithm informally. First, we find the local DUA-node subsumption using SA and the post-dominance relationship. Then, we union of **Covered**(n) and **Covered**(m) sets when m post-dominates n to find the set of DUAs subsumed by a node n.

SA and the post-dominance relationship cost $\approx$ O(|N|), where |N| is the number of nodes in the flow graph. The union of the **Covered** sets costs up to $O(|\mathrm{N}|^2)$ since for each node n, every other node m would be checked to see if it post-dominates n. However, the post-dominators of n are generally fewer than the number of nodes and can be scanned efficiently when implemented as bit vectors using machine instructions.

B. DUA-Edge subsumption

We first use SA to calculate local DUA-node subsumption. Then, Algorithm 2 uses SA results to find the local DUA-edge subsumption.

```
1 for each edge $(n', n)$ do
2     if #Successors($n'$) > 1 then
3         Covered($n', n$) = [(OUT($n'$) - Sleepy($n'$)) $\bigcap$
              PotCovered($n$)] $\bigcup$ Covered($n$);
4     else
5         Covered($n', n$) = Covered($n'$) $\bigcup$ Covered($n$);
6 return Sets Covered($n', n$)
```

Algorithm 2: Local DUA-edge subsumption algorithm

Two results of SA are **IN**(n) and **OUT**(n). **OUT**(n') contains DUAs that are covered or available for coverage after node n' is toured by any path from the start node to n'. Algorithm 2 assumes every edge (n', n) is toured, so it calculates DUAs covered as if the next node following n' is n; that is, set **Covered**(n', n), in Lines 3-5.

If more than one path leaves n' (n' has more than one successor at line 2), then line 3 allows only edge duas with uses in (n', n) to be covered at (n', n) and joins them with **Covered**(n). Note that **Sleepy**(n') removes from **OUT**(n') edge DUAs whose use is in edges (u', u) such that $u' \neq n'$. In other words, only edge DUAS with uses in (n', n) will be allowed to be covered and joined with **Covered**(n). Line 5 deals with edges (n', n) with a single successor. The set of DUAs covered at (n', n) is the union of DUAs covered at n' and n. The global DUA-edge subsumption is calculated by adding to each **Covered**(n', n) those sets **Covered**(m) such that m post-dominates n.

DUA-edge differs from DUA-node subsumption because it calculates the local DUA-edge subsumption (Algorithm 2). Lines 3-5 can be implemented as bitwise operations; so, their cost is constant. The for-loop at line 1 iterates on edges; however, the number of edges is $O(|\mathrm{N}|)$ for most programs. As a result, local DUA-edge subsumption costs $\approx O(|\mathrm{N}|)$.

C. DUA-DUA subsumption

For every DUA D_1, DUA-DUA subsumption associates a set of DUAs D_2 that are covered in every test path that covers D_1. We apply SA to find DUA-DUA subsumption, but in a different graph.

Marré and Bertolino [14], [25] suggested a graph called $G*$ that models all paths that cover a dua $D_1(d_1, u_1, X_1)$. That is, every test path in $G*$ should cover D_1. $G*$ includes paths from the start node (s) to the definition node (d_1), def-clear paths wrt X_1 from d_1 to node u_1, and paths from u_1 to the exit node (e). We use a different graph, graphdua, which includes paths that are not in $G*$ that might block the subsumption of a DUA D_2. We compare and contrast $G*$ and the graphdua in the related work.

Given a DUA D_1(d_1, u_1, X_1), we calculate the flow graph graphdua(D_1), and then run SA on the graph. SA gives **Covered** sets for all nodes of graphdua(D_1); however, **Covered**(e_{D_1}), where e_{D_1} is the exit node of graphdua(D_1), contains the set of DUAs D_2 that are subsumed in any path that covers D_1.

As far as cost goes, $G*$ and graphdua(D_1) cost $O(|\mathrm{E}|)$ to calculate, where |E| is the number of edges [25]; hence, its cost is $O(|\mathrm{N}|)$. SA's cost is determined by the number of nodes in the graph. A graphdua has no more than five times the number of nodes of the program's flow graph, which is $O(|\mathrm{N}|)$. As a result, running SA on a graphdua costs $\approx O(|\mathrm{N}|)$. Hence, calculating the DUAs subsumed by D_1 is $\approx O(|\mathrm{N}|)$.

If U is the set of all DUAs required to test a program, DUA-DUA subsumption will cost $\approx O(|\mathrm{U}||\mathrm{N}|)$ to calculate. Unconstrained DUAs, as shown in Figure 5, cost up to $O(|\mathrm{U}|^2)$ to calculate [36]. Implementing the sets of subsumed DUAs as bit vectors and scanning them with machine instructions reduce this cost.

VI. Experimental analysis

We empirically investigated three research questions regarding SA and data flow subsumptions:

RQ1: Does SA correctly find local DUA-node subsumption?
RQ2: How effective is the data flow subsumption?
RQ3: How long does it take to calculate the data flow subsumption?

The rest of this section presents the subject programs and, for each RQ, the results and its discussion. We conclude the section with threats to validity. We have deployed our replication experimental package here: https://github.com/icst2021satool.

A. Subject programs

For our study, we chose 17 programs from the Defects4J repository [37], plus the machine learning program Weka, as described in Table I. The programs are sorted by the number of methods with DUAs. 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); date and time manipulation (Time); and software testing (Mockito). The programs' size also vary: they range from small programs such as Csv (602 LOC) to larger programs such as Weka (216,781 LOC).

Table I (in section I) shows the LOC, the number of methods with DUAs, the number of methods executed, and the total of DUAs required to test the programs. `javancss` (www.kclee.de/clemens/java/javancss/) calculated the LOC; `SAtool` (github.com/icst2021satool/source-code) found the methods with DUAs and the DUAs themselves. We modified `Jaguar` [38] to collect DUA coverage for every Junit method of the developers' tests included in the programs' repository. We were not able to collect coverage for Lang and Mockito (indicated by the '*' character in Table I) and we collected data from 443 out of 509 executed classes for Weka. `SAtool` was run on a MacAir, 1.8 GHz Dual-Core Intel Core i5, 8 GB 1600 MHz DDR3.

B. RQ1: SA correctness

We verified the correctness of SA in two ways. We proved that the data flow subsumption framework (DSF) is monotone and distributive and that sets **OUT**(n) contain the covered and available for coverage DUAs at a node n [32]. As a result, by applying an iterative algorithm, SA finds sets **OUT** and then the final value of **Covered**(n).

We also used coverage and global DUA-DUA subsumption data to verify our data flow subsumption approach. We could verify the correctness of both SA and graphduas by checking two properties associated with DUA-DUA subsumption.

The subsumption relationship is reflexive; that is, every TR (in our case, a DUA) of a program subsumes itself. The reflexive property was checked in 737,333 DUAs of 28,758 methods of all 18 programs, and failed for 24 methods and 71 DUAs. The 24 failures were based on two issues with the flow graphs: either the start node had incoming edges or graph had self-loops (n,n). Our `SATool` was not able to handle either of those special cases. `SATool` uses ASM (asm.ow2.io/) to be compatible with `Jaguar`. Once we removed 185 (of 28,758) methods with these characteristics, all DUAs subsumed themselves.

The *subsumption* relationship implies that the subsumed DUAs should be covered when the subsuming DUA is. For instance, in Figure 5, whenever unconstrained DUA (0, (1,2), i) is covered, DUAs (0,2, max) and (0, (1,2), $length$) should be covered as well due to the subsumption property. We verified the subsumption property of the unconstrained DUAs for every executed method in the tests. It does not hold for 18 (of 14,532) executed methods. The subsumption property was disrupted in 17 due to the occurrence of an exception inside a `try` clause and one due to a `synchronized` clause, which blocked the coverage of the subsuming DUAs. We calculated the unconstrained DUAs using global DUA-DUA subsumption, which does not address these clauses.

This verification shows that SA finds correctly the data flow subsumption provided its assumptions hold.

C. RQ2: Data flow subsumption effectiveness

Table I shows data regarding the effectiveness of data flow subsumption. Columns **%DUA-node** and **%DUA-edge** show the percentage of DUAs covered if every node and edge are covered. These use local DUA-node and DUA-edge sumpsumption, and they can be combined provide testers with the **Covered**(n) and **Covered**((n',n)) sets. Column **%Unc. DUAs** gives the percentage of unconstrained DUAs with respect to the total of DUAs. The last line of the table presents the average values of these columns.

Effectiveness data show that node and edge coverage can lead to a significant data flow coverage, 66.1% and 78.2% on average. However, many small methods have 100% **%DUA-node** and **%DUA-edge** coverage (9,464 and 17,491 methods). To assess the effectiveness on harder to test methods, Figure 6 gives the histograms of **%DUA-node** and **%DUA-edge** for 1,146 methods (out of 28,758 of all programs) with more than 100 DUAs. The number of methods with **%DUA-node** coverage below 40% is 261 and for **%DUA-edge**, 157. A few methods have as many as 2,847 DUAs. So, many DUAs may still need to be tested after achieving node and edge coverage.

About 30% of all DUAs are unconstrained (Table I). Figure 7 illustrates methods with more than 100 DUAs. Unconstrained DUAs represent less than 30% of all DUAs for 762 out of the 1,146 most demanding methods. Thus, DUA-DUA subsumption in combination with DUA-node and DUA-edge subsumption can significantly reduce the number of DUAs to be verified, since the tester would only test unconstrained DUAs not covered by edge or node coverage.

D. RQ3: Data flow subsumption cost

Columns **t-DUA-node** and **t-DUA-edge** in Table I give the the number of milliseconds needed to find local DUA-node and local DUA-edge subsumption for each program averaged over 10 trials. Note that these numbers are for subsumption analysis only. It takes only around 22s to find the local DUA-node and DUA-edge subsumption for the biggest program, Weka. As

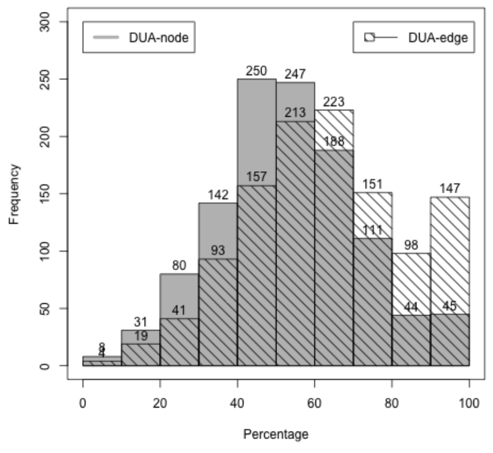

Fig. 6. DUA coverage due to node and edge coverage for methods with more than 100 DUAs

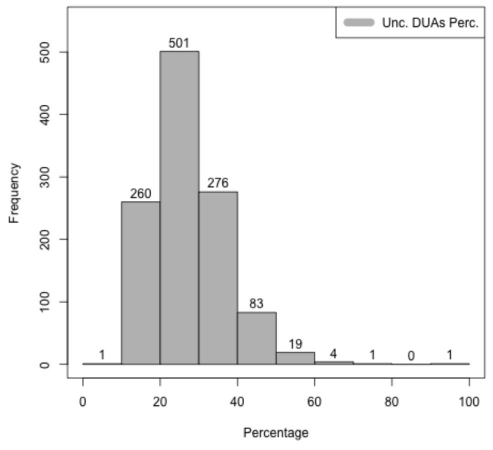

Fig. 7. Unconstrained DUAs for methods with more than 100 DUAs

expected by the asymptotic analysis, **t-DUA-node** and **t-DUA-edge** values are very similar, and **t-DUA-node** slightly more costly for 10 out of 18 program.

SA's cost is dominated by the number of methods: more methods implies more nodes, and as a consequence higher cost. However, three programs (Codec, Compress, and Math) did not follow the cost prediction. They have fewer methods, but SA takes more time.

These three programs have few very complex methods with many nodes, edges, and loops. For instance, Math has two clone methods with 327 nodes, 463 edges, 2,197 DUAs, and 85 back edges, which implies a deep loop nesting. Weka's most demanding method has 187 nodes, 339 edges, 1,301 DUAs and 27 back edges.

Column **t-Unc. DUAs** gives the time to calculate the set of unconstrained DUAs in the same conditions. These times are fast considering the size of the programs: the slowest are Math with 104s and Weka with 98s. Figure 8 plots the number

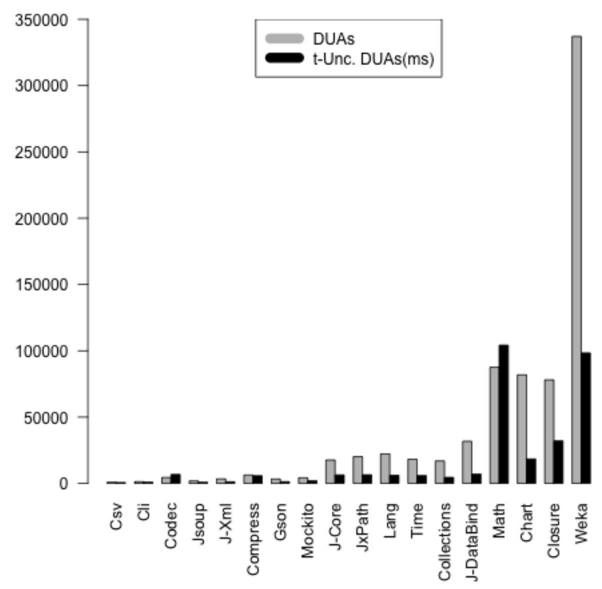

Fig. 8. Number of DUAs and time to find the unconstrained DUAs

of DUAs and the time in milliseconds for unconstrained sets calculation to assess the relation with the number of DUAs. For all programs, excepting Codec, Compress, and Math, **t-Unc. DUAs** is a fraction of **DUAs**.

The effect of the methods' complexity magnifies for unconstrained DUAs calculation. SA is applied on each DUA's graphdua; thus, an increase on SA's cost will be multiplied by the number of DUAs. Nevertheless, even for the outlier programs, SA finds unconstrained DUAs very efficiently.

E. Limitations and threats to validity

The Defects4J repository contains open-source programs that are comparable to industry programs, thus reducing an external threat. To further reduce that threat, we added Weka because mathematical software challenged SA's scalability.

To address internal validity, we verified SA both formally with a proof and empirically. Our SA's implementation relies on Java APIs for most of the data structures used. Nevertheless, they may hide inefficiencies that we are not aware of. Despite that, execution time is fast. We performed a lightweight analysis to determine the data flows of a method. Though it might miss DUAs due to aliasing, most of them will be subsumed by the unconstrained DUAs.

Finally, the coverage implied by data flow subsumptions represents an upper bound because they can be disrupted by exceptions and program aborts. Because we utilized local subsumption, the impact on DUA-node and DUA-edge subsumptions is restricted to code inside `catch` clauses since we do not know which node raised the exception. The unconstrained DUAs, though, were calculated using global DUA-DUA subsumption and could be impacted by exceptions.

VII. Related work

Santellices and Harrold's approach finds DUAs whose coverage are inferable or conditionally inferable by edge coverage based on the concept of def-use order [15]. A DUA $D(d,u,X)$ is in *def-use order* if one of the following conditions hold: (1) Node u cannot reach node d; (2) node d dominates node u; or (3) node u post-dominates node d. Thus, if a DUA D is in def-use order, node d is guaranteed to occur before node u. Additionally, they check whether the re-definitions of X do not occur in paths between d and u. If so, D is inferable; otherwise, it is conditionally inferable if no re-definition of X occurs between d and u in a particular test path.

For each node d and u of D, their technique finds the edges that controls the execution of d and u; that is, d and u are control-dependent on these edges. D is covered if one required edge for d and one for u are covered. If a required edge for a re-definition was taken, D was either not covered or possibly covered in the test path, depending on whether D is inferable or conditionally inferable. Santelices and Harrold's technique costs $O(|\mathrm{U}|)$, where U is the set of all DUAs, since all DUAs have to be checked for def-use order.

We have identified three previous algorithms to find the DUA-DUA subsumption. Marré and Bertolino suggested two algorithms (referred to as M&B I [36] and M&B II [14], [25]); Jiang et al. proposed another [16].

We only discuss M&B II because the paths missed in M&B II are also missed in M&B I and both algorithms have the same complexity. M&B II uses $G*$ to find all DUAs subsumed by a DUA D_1. It first selects all paths that cover D_1 by building $G*$. Then M&B II checks whether every path of $G*$ also traverses a DUA D_2 by initially verifying that d_2 and u_2 are always traversed in $G*$. Then, to find whether every path that covers D_1 also covers D_2, it checks whether no node n_i from d_2 and u_2 in $G*$ contains a definition of X_2. M&B I and M&B II cost $O(|\mathrm{U}|^2|\mathrm{N}|)$ [25], [36].

$G*$ is composed of three sub-graphs encompassing paths from the start node (s) to the definition node (d_1), def-clear paths wrt X_1 from d_1 to node u_1, and paths from u_1 to the exit node (e). However, it does not encode paths from d_1 to d_1 and from u_1 to u_1. Consider test path (0,1,3,4,1,3,5,4,1,2,6) of Max (Figure 1) and $G*$ generated for $D_1(3,5,rogue)$. M&B II will incorrectly conclude that D_1 subsumes $D_2(0,5,i)$ because $G*$ misses path (3,4,1,3) (i.e., d_1 to d_1) in which variable i is re-defined. Our graphdua fixes that by adding the missing paths d_1 to d_1 and u_1 to u_1 to $G*$. Consequently, a graphdua will have as many as five sub-graphs; each with at most the number of nodes of the original flow graph.

Jiang et al.'s [16] algorithm for DUA-DUA subsumption is based on the concepts of *def-use order* and *control dependency* [15]. A DUA D_2 is subsumed by D_1 if and only if the following three conditions are satisfied: (1) D_1 and D_2 are in def-use order; (2) $\mathrm{CD}(d_1) \bigcup \mathrm{CD}(u_1) \supseteq \mathrm{CD}(d_2) \bigcup \mathrm{CD}(u_2)$, where $\mathrm{CD}(n)$ is the edges of which n is control-dependent; and (3) there is a path between d_1 and u_1 that does not contain a definition of X_1, and there is a path between d_2 and u_2 that does not contain a definition of X_2.

However, Jiang et al.'s technique misses the very same paths that $G*$ misses. Consider again $D_1(3,5,rogue)$ subsuming $D_2(0,5,i)$. Both DUAs are in def-use order because the def node dominates the use node; and Jiang et al.'s conditions (2) and (3) for subsumption are also valid. Nevertheless, the def-use order does not exclude a path from node 3 to node 3 that blocks the subsumption of (0,5,i) by (3,5,$rogue$). The paper also did not discuss complexity, but it is at least $O(|\mathrm{U}|^2)$, since every DUA is checked against every other.

The Subsumption Algorithm (SA) efficiently calculates the local DUA-node subsumption ($\approx O(|\mathrm{N}|)$) and, as a result, allows efficient calculation of local DUA-edge ($\approx O(|\mathrm{N}|)$) and DUA-DUA subsumption ($\approx O(|\mathrm{U}||\mathrm{N}|)$). M&B II and Jiang et al.'s algorithm for DUA-DUA subsumption can be fixed by adding the missing paths, but they are still more expensive. Our experimental data suggests that the cost will increase significantly if one uses algorithms that are quadratic in the number of DUAs.

VIII. Conclusions

Studies indicate that data flow testing (DFT) can detect faults missed by control flow testing (CFT) [9]. However, the large number of of DFT test requirements (DU-associations or DUAs) have hampered its adoption by the industry. The subsumption relationship can identify redundant DUAs so that testers could focus on fewer DUAs that will still lead to high data flow coverage [14], [15].

Reliably identifying subsumption for data flow, though, is a difficult problem, whose early solutions have subtle flaws and inefficient algorithms. We tackled the data flow subsumption problem by modeling it as data flow analysis framework. In doing so, we were able to solve the local DUA-node subsumption with cost $\approx O(|\mathrm{N}|)$, where $|\mathrm{N}|$ is the number of nodes in the program's flow graph. Using the local DUA-node subsumption, one can find other data-flow (DUA-node, DUA-edge, and DUA-DUA) subsumptions at substantially reduced costs. Our experimental data suggest that DFT costs can be reduced significantly by combining data flow subsumptions.

We have several plans for extending this research. One question is whether test sets created for unconstratined DUAs detect as many faults as test sets created for all DUAs. This question gets at the heart of the value of test criteria and subsumption, and could be affected by infeasible DUAs and interrupted executions, which we identify in this paper. We theorize that DUA relationships inside `try` and `catch` clauses can be explored with local DUA-DUA subsumption and special flow graphs. Comparison of our approach with random testing could add to the benefit of our approach hence we plan it as future work. Finally, spectrum-based fault localization techniques can benefit from data flow subsumptions to select relevant spectra.

Acknowledgment

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

References

[1] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[3] J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, ser. UML'99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 416–429.

[4] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, p. 366–427, Dec. 1997. [Online]. Available: https://doi.org/10.1145/267580.267590

[5] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Software Testing, Verification, and Reliability*, vol. 15, no. 3, pp. 167–199, 2005. [Online]. Available: https://doi.org/10.1002/stvr.319

[6] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, UK: Cambridge University Press, 2017.

[7] 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.

[8] 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.

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

[10] 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.

[11] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proc. of the 31st International Conference on Software Engineering*, ser. ICSE, 2009, pp. 56–66.

[12] H. L. Ribeiro, R. P. A. de Araujo, M. L. Chaim, H. A. de Souza, and F. Kon, "Evaluating data-flow coverage in spectrum-based fault localization," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19-20, 2019*. IEEE, 2019, pp. 1–11.

[13] 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.

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

[15] 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.

[16] 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.

[17] M. S. Hecht, *Flow analysis of computer programs*. New York: Elsevier North-Holland, 1977.

[18] J. B. Kam and J. D. Ullman, "Monotone data flow frameworks," *Acta Informatica*, vol. 7, pp. 305–317, 1977.

[19] M. L. Chaim and R. P. A. de Araujo, "An efficient bitwise algorithm for intra-procedural data-flow testing coverage," *Information Processing Letters*, vol. 113, no. 8, pp. 293–300, 2013.

[20] 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.

[21] S. C. Ntafos, "On required element testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 795–803, 1984.

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

[23] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A comparison of data flow path selection criteria," in *Proceedings of the 8th International Conference on Software Engineering*, ser. ICSE '85. Washington, DC, USA: IEEE Computer Society Press, 1985, p. 244–251.

[24] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

[25] 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.

[26] 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)*, Cleveland, OH, March 2014, pp. 21–30.

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

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

[29] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "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.

[30] 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)*, Chicago Illinois, USA, April 2016.

[31] S. Horwitz, A. Demers, and T. Teitelbaum, "An efficient general iterative algorithm for dataflow analysis," *Acta Informatica*, vol. 24, pp. 679–694, 1987.

[32] M. L. Chaim, K. Baral, and J. Offutt, "A data flow analysis framework for data flow subsumption," Tech. Rep., sep 2020. [Online]. Available: https://github.com/icst2021satool/techreport

[33] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, 2nd ed. Boston: Pearson Addison-Wesley, 2007.

[34] D. E. Knuth, "An empirical study of FORTRAN programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380010203

[35] B. G. Ryder and M. C. Paull, "Elimination algorithms for data flow analysis," *ACM Computing Surveys*, vol. 18, no. 3, p. 277–316, Sep. 1986. [Online]. Available: https://doi.org/10.1145/27632.27649

[36] M. Marré and A. Bertolino, "Unconstrained DUAs and their use in achieving all-uses coverage," in *Proceedings of the International Symposium on Software Testing and Analysis*. New York, USA: ACM Press, 1996, pp. 147–157.

[37] 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.

[38] H. L. Ribeiro, H. A. de Souza, R. P. A. de Araujo, M. L. Chaim, and F. Kon, "Jaguar: A spectrum-based fault localization tool for real-world software," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 404–409.