A Data Flow Analysis Framework for Data Flow Subsumption

***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. In this work, we formally describe the *Data Flow Subsumption Framework* (DSF) conceived to tackle the data flow subsumption problem. We show that DFS is a distributive data flow analysis framework which allows efficient iterative algorithms to find the Meet-Over-All-Paths (MOP) solution for DSF transfer functions. The MOP solution implies that the results at a point p are valid for all paths that reach p. We also present an algorithm, called Subsumption Algorithm (SA), that uses DSF transfer functions and iterative algorithms to find the *local* DU-associations-node subsumption; that is, the set of DU-associations that are covered whenever a node n is toured by a test. A proof of SA's correctness is presented and its complexity is analyzed.**

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

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] conducted a study in which control-flow criteria, namely, statement, branch, loop, and MCDC coverage, were compared 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 cases 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. Additionally, DUAs can be subsumed by test requirement (e.g., node, edge) of different criteria so that once they are covered some DUAs are guaranteed to also be covered [11].

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 [12], [13]. Using the local DUA-node subsumption, one can efficiently discover the subsumption of DUAs with respect to nodes, edges, and other DUAs [14].

The goal of this work is to describe formally the Data Flow Subsumption Framework (DSF), showing that DFS is distributive and that it can be used to solve the local DUA-node subsumption. A distributive framework allows iterative algorithms to find the Meet-Over-All-Paths (MOP) solution, which means that the results are valid for all paths that reach a program point p. We also present an algorithm, called Subsumption Algorithm (SA), that uses DSF transfer functions and iterative algorithms to find the *local* DUA-node subsumption. A proof of SA's correctness is presented and its complexity is analyzed.

This document starts with background in data flow testing in section II. We then describe the local DUA-node subsumption problem in section III. Section IV describes the Data Flow Subsumption Framework (DSF), followed by the algorithm to solve the local DUA-node subsumption in section V. We draw the conclusions in section VI.

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*, as 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 [15] 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.

```
/* 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.

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

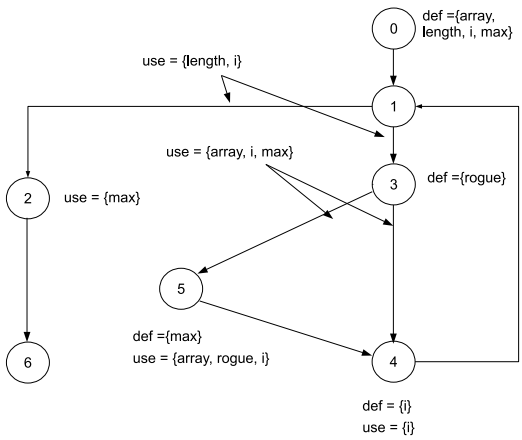

Fig. 2. Annotated flow graph for max.eps

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 [1]–[4]. In this paper, we focus on the *all-uses* criterion proposed by Rapps and Weyuker [3].

TABLE I
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)

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*.

III. 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 $D(d, u, X)$ is *DUA-subsumed* by node n if D is covered on a test path that visit node n and reach 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

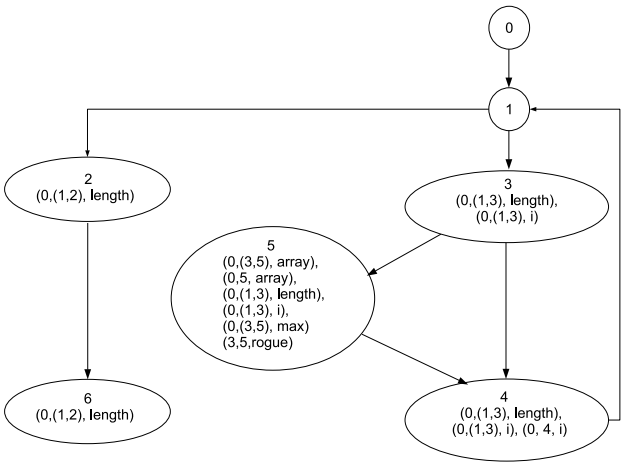

Fig. 3. Local DUA-node subsumption for program Max

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 [12]. 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 [10]. In the absence of early program termination, node 5 post-dominates 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%.

In the next section, we present a data flow analysis framework that allows one to find the local DUA-node subsumption.

IV. Data flow subsumption framework

Data flow analysis frameworks were devised to solve data flow analysis problems such as reaching definitions, live-variables, and available expressions [13]. Their goal is to determine a *fact* that is valid at the entry or exit of a program point p whenever p is reached [16]. We describe below the *Data Flow Subsumption Framework* (DSF), which can be used to find data flow testing subsumption relationships.

The fact that DSF determines is the set of DUAs already *covered* or *available* to be covered at the entrance of a node n and at the exit of n. In doing so, the DUAs covered along all paths that reach a node n—and also after its traversal—are discovered, as well as the DUAs available for coverage in these very same paths. A DUA is available for coverage in a path if its def node was toured previously in the path and there is no redefinition of its associated variable in the subsequently toured nodes of the path.

In what follows, we present the formal description of DSF. We start off by briefly discussing the definitions regarding data flow analysis frameworks. Next we show DSF transfer functions and its properties. We then present the Subsumption Algorithm (SA) which solves DFS and finds all DUAs covered by any path that reach a particular node n; that is, the local DUA-node subsumption. SA is an adaptation of interactive algorithms to solve data flow analysis frameworks [17]. We finish up the section by analyzing SA's complexity.

A. Definitions

The goal of data flow analysis is to solve data flow problems by assigning a program fact to each node of the flow graph that will be valid every time the node is reached during every possible execution [16]. Data flow analysis problems can be *forward* or *backward* problems. A forward problem gives the fact valid at the entrance of a particular node whilst backward problems at the exit of the node. Available expressions and reaching definitions are examples of forward problems; live-variable analysis is a backward problem. Data flow analysis frameworks model data flow problems so that particular algorithms are able to find the fact assigned to each node.

The domain of program facts in this work is modeled as a bounded (contains no infinite chains) semi-lattice V with meet[1] operation $\wedge$, least element $\perp$ (bottom), greatest element $\top$ (top), and a partial order $\leq$. The top element is such that $x \wedge \top = x$ for any element x in V. The least element $\perp$ (bottom) is such that for any x element of V, $x \wedge \perp = \perp$.

The relationship between values of V before and after a node n of the flow graph is given by the transfer functions of the framework. The family of transfer functions $F : V \rightarrow V$ in a data flow analysis framework has the following properties: (1) F has an identity function I, such that $I(x) = x$ for all x in V; (2) F is closed under composition; that is, for any two functions f and g in F, the function h defined by $h(x) = g(f(x))$ or $h(x) = g \circ f(x)$ is in F [18]. A function $f : V \rightarrow V$ is *monotone* iff for all x and y in V, $x \leq y$ implies $f(x) \leq (y)$.

In this work, a *monotone data flow analysis framework* is composed of the following components:

1) A data flow graph, with specially labeled ENTRY and EXIT node;
2) a direction (forward or backward) of the data flow;
3) a set of values V;
4) a meet operator $\wedge$ and partial order $\leq$;
5) a set of functions F such that the identity function belongs to F and all f_n in F, where f_n is the transfer

[1]Meet is a binary operation for which the idempotent, commutative, and associative properties hold.

function of node n, are monotone and closed under composition; and

6) a constant value $v_{ENTRY} \in \{\bot, \top\}$ or $v_{EXIT} \in \{\bot, \top\}$, representing the boundary condition for a forward or backward framework, respectively.

We denote a function f_π associated with path π, as follows: if π is the empty path then $f_\pi(x) = x$ (identity function); if $\pi = (n_1, n_2, \ldots, n_{k-1}, n_k)$ then $f_\pi(x) = f_{n_k} f_{n_{k-1}} \cdots f_{n_2} f_{n_1}(x)$.

In general, algorithms that solve data flow analysis problems with data flow analysis frameworks find the *maximum fixed point* (MPF) solution. Let MPF(n) be the *maximum fixed point* solution for node n, then MPF(n) $\leq f_p$(MPF(p)) where p is a predecessor of n. The ideal result for a data flow analysis framework, though, is the *meet-over-all-paths* solution, the map MOP(n) = $\wedge$ $\{f_\pi(v_{ENTRY}) \mid \pi$ is a path from s to $n\}$ [16].

The MOP solution means that the meet operation is applied to the composition of the transfer functions along all paths that can reach n. Unfortunately, the MOP solution is in general undecidable because a flow graph with cycles may have an unbounded number of paths. Luckily, it can be obtained if the transfer functions f_n of F of the data flow analysis framework are distributive; that is, $f_n(x \wedge y) = f_n(x) \wedge f_n(y)$[2]. A monotone data flow analysis framework whose transfer functions is distributive is called *distributive data flow analysis framework*.

In the case of DSF, the data flow graph is the flow graph $G(N, E, s, e)$ where ENTRY is the start node s and EXIT is the exit node e. The direction of the data flow is forward and the values of V comprises all subsets obtained from all DUAs required to test a program P. The meet operation is set intersection and the partial order is defined by the $\subseteq$ operation. The v_{ENTRY} is $\varnothing$ because at the entrance of the start node s there is no DUA covered or available for coverage.

We present the DSF transfer functions as follows.

B. Transfer functions

The transfer functions are a pivotal point of a data flow analysis framework since they ultimately calculate the fact that is valid at the entrance and exit of a node n when it is reached. To define the transfer functions of DSF, we associate sets with each node of the flow graph. They were originally introduced by Chaim and Araujo [15] and are defined as follows:

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

Born(n) : set of DUAs (d,u, X) or (d,(u',u), X) such that $d = n$. It encompasses DUAs that become available after the traversal of its def node.

Disabled(n) : set of DUAs (d,u, X) or (d,(u',u), X) such that X is defined in n and $d \neq n$. Set of DUAs whose variable X has been defined at node n so that these DUAs become disabled (or unavailable) for coverage after the traversal of n.

PotCovered(n) : set of DUAs (d,u, X) or (d,(u',u), X) so that $u = n$. It comprises DUAs that are potentially covered at n provided they are available; that is, their def node has been previously traversed.

Sleepy(n) : set of DUAs (d,(u',u), X) such that $u' \neq n$. DUAs that are temporarily unavailable (sleepy) because they cannot be covered immediately after the of traversal n. It includes DUAs such that the use occurs in an edge (u', u) where $u' \neq n$.

Born(n) sets are similar to the **gen**(n) and **e_gen**(n) sets of the reaching definitions and available expressions problems [18]. They represent those DUAs that are *born* because the node where their variable is assigned has been toured. Likewise, **Disabled**(n) is analogous to **kill**(n) and **e_kill**(n) sets of the same data flow analysis problems since they contain those DUAs that are *killed* after the traversal of n.

Differently of other data flow analysis problems, DSF needs sets **PotCovered**(n) and **Sleepy**(n). The first sets represent those DUAs that can *potentially* be covered when a node is traversed. If a DUA is avaliable for coverage when n is reached and it belongs to **PotCovered**(n), then it will be covered after the traversal of n.

Sleepy(n) sets aim at blocking some edge DUAs (d,(u',u),X) of being covered after the traversal of a node n. For instance, **Sleepy**(3) comprises all edge DUAs excepting those whose use is either in edge (3,5) or (3,4), which means that only DUAs with these edges will be allowed to be covered after the traversal of node 3. Consider that a test has toured the path (0, 1, 3, 5) of program Max. The next node to be toured is node 4. **Sleepy**(5), according to the definition, contains all edge DUAs required by Max because there is no DUA with an use in edge (5,4) to be spared. So when node 4 is toured, **Sleepy**(5)—the **Sleepy** set of its predecessor—can be used to block the coverage of edge DUAs such as (0, (3,4), array) or (4, (3,4), i) by path (0, 1, 3, 5, 4), which can potentially be covered at node 4, but when the predecessor is node 5 they cannot.

Additionally, the transfer functions of DSF utilizes two working sets—the current sleepy DUAs (**CurSleepy**) and the covered DUAs at n (**Covered**(n))—in their definition. Below we present the transfer functions (referred to as TF1 and TF2) and two auxiliary functions (referred to as AF1 and AF2) used to calculate **CurSleepy** and **Covered**(n).

TF1 $\mathbf{IN}(n) = \bigcap_{p \in PRED(n)} \mathbf{OUT}(p)$

AF1 $\mathbf{CurSleepy} = \bigcup_{p \in PRED(n) \text{and} (p,n) \text{is not a back edge}} \mathbf{Sleepy}(p)$

AF2 $\mathbf{Covered}(n) = \bigcap_{p \in PRED(n)} \mathbf{Covered}(p) \cup [(\mathbf{IN}(n) - \mathbf{CurSleepy}) \cap \mathbf{PotCovered}(n)]$

TF2 $\mathbf{OUT}(n) = \mathbf{Born}(n) \cup [\mathbf{IN}(x) - \mathbf{Disabled}(n)] \cup \mathbf{Covered}(n)$

where PRED(n) is the set of nodes of G that are predecessors of node n.

The interactive algorithms that solve data flow analysis problems find the values (facts) associated with sets **IN**(n) (forward problems) and with sets **OUT**(n) (backward problems). In DSF, **IN**(n) contains the set of DUAs already *covered* or *available* to be covered at the entrance of a node n when it

[2]A distributive function is also monotone [18].

is reached by any path. **OUT**(n), in turn, contains those DUAs covered or available for coverage after the traversal of n by these paths. The meet operation of DSF is the set intersection and it is used to find the value of **IN**(n) by intersecting **OUT** sets of the predecessors of n in TF1.

Auxiliary function AF1 calculates the edge DUAs that cannot be covered at node n. **CurSleepy** is the union of the DUAs blocked after the tour of a predecessor p of n provided (p,n) is not a back edge. A back edge (p,n) is such that node n dominates node p [18]. In Figure 2, edge (4,1) is a back edge. **CurSleepy** is used to block edge DUAs of being covered when one cannot predict from which path a node n is reached. However, when (p,n) is a back edge, one knows that n is always toured before touring p so that it does not block other edge DUAs of being covered at n.

Consider node 4 of the Max program; it can be reached by nodes 3 and 5 through non-back edges (3,4) and (5,4). However, one does not know from which predecessor it has been reached; thus, no edge DUA with edges (3,4) and (3,5) can be covered at node 4 by all paths that reach 4. **CurSleepy** at node 4 is the union of **Sleepy**(3) and **Sleepy**(5). **Sleepy**(3) comprises all edge DUAs excepting those whose uses occur in edges (3,5) or (3,4); and **Sleepy**(5) comprises all edge DUAs as mentioned before. As a result, **CurSleepy** will contain all edge DUAs; thus, none would be covered at node 4 according to AF1.

Auxiliary function AF2 finds the DUAs that are covered by all paths that reach node n; it is divided in two parts that are added by a union set operation. The first part of AF2 intersects **Covered** sets of the predecessors of n. In doing so, node n will *inherit* only DUAs that are covered previously in all paths that reach it. The second part of AF2 calculates the DUAs covered at node n. **IN**(n) has the DUAs covered and available to be covered in all paths that reach n according to TF1; **CurSleepy** has the edge DUAs that are blocked at node n; and **PotCovered**(n) contains DUAs that might be covered at n provided they are in **IN**(n). The operations described in the second part of AF2 determine the DUAs covered at n: **CurSleepy** DUAs are removed from **IN**(n) and the result is intersected with **PotCovered**(n). The remaining DUAs after these operations along with DUAs covered in previously toured nodes give the DUAs covered at node n.

Finally, transfer function TF2 determines the **OUT**(n) sets; that is, those DUAs that are *forwarded* in the data flow analysis. They are calculated in three parts that are added by union set operations. The first part is the **Born**(n) set, which contains the DUAs that become available for coverage at node n; that is, their variable is assigned at n. The second part is composed of those DUAs that are available in **IN**(n) and *survive* node n because they do not belong to **Disabled**(n). The last set added in TF2 is the set of DUAs covered at n (**Covered**(n)). All these DUAs are forwarded to its successors in the data flow analysis.

The transfer functions of DSF are only useful if they satisfy the properties required by the data flow analysis frameworks; that is, they should include the identity function and satisfy the properties of closure under composition and monotonicity. If they do then there exist interactive algorithms that solve them.

In Appendix A, we show that DSF transfer functions include an identity function and is closed under composition. Furthermore, we show that DSF transfer functions are distributive, which guarantees that they are monotone and finds the MOP solution for DSF transfer functions.

V. Solving the local DUA-node subsumption problem

We present an interactive algorithm, described in Algorithm 1, to solve the DUA subsumption problem by means of DSF. It is adapted from classical algorithms [18] using the above transfer functions. Although the DSF solution consists of sets **IN**(n) and **OUT**(n), we are interested in the final values of the **Covered**(n) sets. They contain the DUAs that are covered at node n whenever it is reached from any path beginning at the start node s of G. In other words, it finds the *local* DUA-node subsumption; that is, the annotation presented in Figure 3. We call Algorithm 1 *Subsumption Algorithm* (SA).

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.

Lines 1-12 comprise the iterative algorithm to solve DSF transfer functions [18]; that is, to find the **IN** and **OUT** sets. Lines 13-16 update the **Covered**(n) sets. **OUT**(n) has already converged to its final value after leaving the while-loop at Line 7, but **Covered**(n) needs to be updated with the final values of **OUT**(p).

When the transfer functions hold the properties required by a monotone data flow framework, interactive algorithms can

find the MFP solution to the particular framework. By showing that the set of DFS transfer functions contains the identity function and are closed under composition and distributive, these algorithms will find the MOP solution for sets **IN**(n) and **OUT**(n) of DSF transfer functions (see Appendix A).

However, it rests to show that **IN**(n) and **OUT**(n) sets contain, respectively, the set of covered DUAs or available to be covered at the entrance and exit of a node n when it is reached by any path starting at node s. We show in Appendix B that SA finds the local DUA-node subsumption.

A. Complexity

The complexity of SA is given by the number of iterations needed to find the solution of DSF; that is, to terminate the execution of the while-loop of Algorithm 1. DSF is a data flow analysis framework which has a set of values V, given by the power set of all DUAs (U) required to test a program P, and a meet operation $\wedge$, given by set intersection operation ($\cap$). V and the meet operation constitutes a semi-lattice.

In the worst case, the cost to find the solution for DSF is the product of the height of the semi-lattice and the number of nodes of the flow graph [18]. The height of DSF semi-lattice is the number of DUAs. However, DSF shares a characteristic with other practical data flow analysis problems like reaching definitions and available expressions. The value of V (fact) at each node — in the DSF case, the covered or available DUAs — propagates along cycle-free paths.

This property can be expressed as follows. If a DUA D is removed from **IN**(n) or **OUT**(n) then there is a cycle-fre path from node s to the beginning or end of node n, respectively, along which D is never covered, or there is an cycle-free path from the node where D's variable is redefined to node n. To be in **OUT**(n), a DUA D should be covered or available in all paths from s to n, then it suffices an cycle-free path in which D is not covered from s or an cycle-free path from the redefinition of D's variable to n to remove it.

The visit of the nodes in iterative algorithms can be determined in such a way that the information is passed along cycle-free paths in few iterations. If the nodes are visited in a depth-first ordering, the retreating edges[3]. will be visited at the end, after the information has gone through the cycle-free paths.

Using this approach, the number of iterations will be no greater than the number of retreating edges plus two. However, the number of retreating edges is never greater than the depth of loop nesting in a flow graph [18]. In practice, the nesting of loops seems limited to a small constant for intra-procedural flow graphs [19], [20].

However, SA has the additional cost of finding 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.

[3]Edge (n', n) is a retreating edge if node n' has a higher depth-first ordering than n. Depth-first ordering is also known as reverse postorder (rPostorder) [12]. Every back edge is a retreating edge, but not every retreating edge is a back edge [18]

As a result, the cost of SA tends to be linear to the number of nodes of the flow graph or $\approx O(\mid N \mid)$.

VI. Conclusions

We presented the formal description of the Data flow Subsumption Framework (DSF). DSF models the problem of finding the definition-use associations (DUAs) covered whenever a particular point p (e.g, a node) of a program is reached as a data flow analysis problem. We call this problem *local DUA-node subsumption*.

DSF is a distributive data flow analysis framework. This property allows iterative algorithms to find the Meet-Over-All-Paths (MOP) solution for DSF transfer functions, which implies that the results at a point p are valid for all paths that reach p. The Subsumption Algorithm (SA) presented in this work is an incarnation of one of them. SA utilizes DSF transfer functions to calculate efficiently the local DUA-node subsumption at $\approx O(\mid N \mid)$ cost.

This result has practical implications since SA allows to find the subsumption of DUAs by edges (local DUA-edge subsumption) and by DUAs (DUA-DUA subsumption) at $\approx O(\mid N \mid)$ and $\approx O(\mid U \mid\mid N \mid)$ costs, respectively [14], where N is the set of nodes of the program's flow graph and U is the set of DUAs required to test it according to all-uses criterion.

Previous solutions are significantly more expensive. Santelices and Harrold [11] propose an algorithm to find DUA-edge subsumption at $O(\mid U \mid)$ cost. Marré and Bertolino's [9], [21], [22] and Jiang et al.'s [10] solutions are quadratic to the number of DUAs for the DUA-DUA subsumption, which hamper their application at industrial settings. Experimental results suggest that SA's application on industry-like applications works at scale and is quite promising [14].

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] 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.
[12] M. S. Hecht, *Flow analysis of computer programs*. New York: Elsevier North-Holland, 1977.
[13] J. B. Kam and J. D. Ullman, "Monotone data flow frameworks," *Acta Informatica*, vol. 7, pp. 305–317, 1977.
[14] "Efficiently finding data flow subsumptions," October 2020, submitted for publication.
[15] M. L. Chaim and R. P. A. de Araujo, "An efficient bitwise algorithm for intra-procedural data-flow testing coverage," *Inf. Process. Lett.*, vol. 113, no. 8, pp. 293–300, 2013.
[16] S. Horwitz, A. Demers, and T. Teitelbaum, "An efficient general iterative algorithm for dataflow analysis," *Acta Informatica*, vol. 24, pp. 679–694, 1987.
[17] T. J. Marlowe and B. G. Ryder, "Properties of data flow frameworks: A unified model," *Acta Informatica*, vol. 28, pp. 121–163, 1990.
[18] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, 2nd ed. Boston: Pearson Addison-Wesley, 2007.
[19] 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
[20] B. G. Ryder and M. C. Paull, "Elimination algorithms for data flow analysis," *ACM Comput. Surv.*, vol. 18, no. 3, p. 277–316, Sep. 1986. [Online]. Available: https://doi.org/10.1145/27632.27649
[21] 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, U.S.A.: ACM Press, 1996, pp. 147–157.
[22] 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.

APPENDIX

A. Properties of transfer functions

The set of transfer functions F of a monotone data flow analysis framework should satisfy particular conditions or properties, namely, F should include an identity function and all $f_n \in F$ should be monotone and closed under composition.

The importance of these properties is to allow the aggregation of data flow information at path level. For a path $s, \ldots, n_i, \ldots n$, the first property, the identity function, implies that $I(x)$ is associated with an empty path which keeps the fact unaltered. Being the transfer functions closed under composition implies that the transfer function of the path: $f_{s,\ldots,n_i,\ldots,n}(x) = f_n \circ \ldots \circ f_{n_i} \ldots \circ f_s(\mathrm{x})$ is also a transfer function.

If the functions belonging to F are monotone then, for all x and y in V, $x \leq y$ implies $f(x) \leq f(y)$, where $f : V \to V \in F$. This property allows iterative algorithms in general find the *maximum fixed solution* (MPF) for data flow analysis problems. A stronger property of a data flow analysis framework is the *distributive condition* given by:

$$f(x \wedge y) = f(x) \wedge f(y)$$

for all x and y in V and f in F.

A distributive framework is necessarily monotonic. The closure under composition and the distributive condition have an important implication on the iterative algorithm that solves DSF. Because DFS holds these properties, the iterative algorithm will find the *meet-over-all-paths* (MOP) for DSF, which means that the solution is valid for any path taken from the start node s to a node n.

In the case of DSF, the meet operator $\wedge$ is the set intersection $\cap$ operation. To simplify the manipulation of the DSF transfer functions, we are going to represent them as:

$$f(x) = B \cup (x - D) \cup C \cup (x - CS) \cap P \qquad (1)$$

where x is the value of **IN**(n), $f(x)$ is **OUT**(n), and B, D, C, CS, and P are, respectively, the values of **Born**(n), **Disabled**(n), the result of the intersection of the **Covered**(p) where p is a predecessor of n, **CurSleepy**, and **PotCovered**(n). All sets have fixed values, excepting the set x.

Before proving the properties above, we highlight the fixed values of the DSF transfer functions. **Born**(n), **Disabled**(n), and **PotCovered**(n) are fixed values associated to node n. **CurSleepy** is calculated by the union of the **Sleepy** sets of the predecessors of n. Since **Sleepy** sets are constant and associated with the predecessors of n, **CurSleepy** sets could be calculated beforehand for every node n.

A less obvious fixed value is the result of $\bigcap_{p \in PRED(n)}$ **Covered**(p). Although the value of the **Covered**(n) might change in every iteration of the algorithm, the value of the **Covered** sets of n's predecessors are fixed when **OUT**(n) is calculated as the output of a transfer function $f(x)$. In this sense, **Covered**(p) sets are as fixed as **Born**(n), **Disable**(n), **Sleepy**(n), **PotCovered**(n), and **OUT**(p) during the calculation of **OUT**(n) in a particular iteration.

In what follows, we show that DSF transfer functions comply with the properties that characterize it a distributive data flow analysis framework.

1) Identity function: For a identity function $I(x)$, such that $I(x) = x$, be part of F, it suffices B, D, C, CS, and P all be the empty set.

2) Closure under composition: The set F of transfer functions is closed under composition iff, for any two functions f and g in F, the function h defined by $h(x) = g(f(x))$ is in F. To show the closure under composition, let us suppose we have two functions:

$$f_1(x) = B_1 \cup (x - D_1) \cup C_1 \cup (x - CS_1) \cap P_1 \qquad (2)$$

and

$$f_2(x) = B_2 \cup (x - D_2) \cup C_2 \cup (x - CS_2) \cap P_2. \qquad (3)$$

Then the composition of $f_2 \circ f_1$ will be:

$$f_2(f_1(x)) = B_2 \cup ((B_1 \cup (x - D_1) \cup (C_1 \cup (x - CS_1) \cap P_1) - D_2) \cup C_2 \cup ((B_1 \cup (x - D_1) \cup C_1 \cup (x - CS_1) \cap P_1) - CS_2) \cap P_2.$$

Note that $A \cup B \cup C - D$ is equivalent to $(A - D) \cup (B - D) \cup (C - D)$. We can rewrite $f_2(f_1(x))$ in the following steps.

1) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2 \cup (x - D_1) - D_2 \cup (C_1 \cup ((x - CS_1) \cap P_1) - D_2)) \cup (C_2 \cup (B_1 - CS_2 \cup (x - D_1) - CS_2 \cup (C_1 \cup (x - CS_1) \cap P_1 - CS_2)) \cap P_2)$.
   Note that $A - B - C$ is equivalent to $A - (B \cup C)$.
2) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup (C_1 - D_2 \cup (x - CS_1) \cap P_1 - D_2)) \cup (C_2 \cup ((B_1 - CS_2) \cup (x - (D_1 \cup CS_2)) \cup (C_1 \cup (x - CS_1) \cap P_1 - CS_2)) \cap P_2)$.

Note that $(x - CS_1) \cap P_1 - D_2$ is equivalent to $(x - (CS_1 \cup D_2)) \cap P_1$ and $(x - CS_1) \cap P_1 - CS_2$ to $(x - (CS_1 \cup CS_2)) \cap P_1$.

3) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup ((C_1 - D_2) \cup ((x - (CS_1 \cup D_2)) \cap P_1) \cup (C_2 \cup ((B_1 - CS_2) \cup (x - (D_1 \cup CS_2)) \cup ((C_1 - CS_2) \cup (x - (CS_1 \cup CS_2)) \cap P_1) \cap P_2)$.
4) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup ((C_1 - D_2) \cup ((x - (CS_1 \cup D_2)) \cap P_1) \cup (C_2 \cup ((C_1 \cup B_1 - CS_2) \cup (x - (D_1 \cup CS_2)) \cup (x - (CS_1 \cup CS_2)) \cap P_1) \cap P_2)$.
5) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup ((C_1 - D_2) \cup ((x - (CS_1 \cup D_2)) \cap P_1) \cup (C_2 \cup ((C_1 \cup B_1 - CS_2) \cap P_2 \cup (x - (D_1 \cup CS_2)) \cap P_2 \cup (x - (CS_1 \cup CS_2)) \cap P_1 \cap P_2)$.
6) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup (C_1 - D_2) \cup C_2 \cup (C_1 \cup B_1 - CS_2) \cap P_2 \cup (x - (CS_1 \cup D_2)) \cap P_1 \cup (x - (D_1 \cup CS_2)) \cap P_2 \cup (x - (CS_1 \cup CS_2)) \cap P_1 \cap P_2$.
Note that $(x - (CS_1 \cup D_2)) \cap P_1$ is equivalent to $(x - D_2) \cap P_1 \cup (x - CS_1) \cap P_1$.
7) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup (C_1 - D_2) \cup C_2 \cup (C_1 \cup B_1 - CS_2) \cap P_2 \cup (x - D_2) \cap P_1 \cup (x - CS_1) \cap P_1 \cup (x - (D_1 \cup CS_2)) \cap P_2 \cup (x - (CS_1 \cup CS_2)) \cap P_1 \cap P_2$.
Note that $(x - (CS_1 \cup CS_2)) \cap P_1 \cap P_2 \subseteq (x - CS_1) \cap P_1$.
8) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup (C_1 - D_2) \cup C_2 \cup (C_1 \cup B_1 - CS_2) \cap P_2 \cup (x - D_2) \cap P_1 \cup (x - CS_1) \cap P_1 \cup (x - D_1) \cap P_2 \cup (x - CS_2) \cap P_2$.
9) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup (C_1 - D_2) \cup C_2 \cup (C_1 \cup B_1 - CS_2) \cap P_2 \cup (x - (CS_1 \cup D_2)) \cap P_1 \cup (x - (D_1 \cup CS_2)) \cap P_2$.
Note that $(x - (CS_1 \cup D_2)) \cap P_1 \cup (x - (D_1 \cup CS_2)) \cap P_2$ is equivalent to $(x - (CS_1 \cup D_2)) \cup (x - (D_1 \cup CS_2)) \cap (P_1 \cup P_2)$.
10) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup (C_1 - D_2) \cup C_2 \cup (C_1 \cup B_1 - CS_2) \cap P_2 \cup (x - (CS_1 \cup D_2)) \cup (x - (D_1 \cup CS_2)) \cap (P_1 \cup P_2)$.
11) $f_2(f_1(x)) = B_2 \cup (B_1 - D_2) \cup (x - (D_1 \cup D_2)) \cup (C_1 - D_2) \cup C_2 \cup (C_1 \cup B_1 - CS_2) \cap P_2 \cup (x - (D_1 \cup D_2 \cup CS_1 \cup CS_2)) \cap (P_1 \cup P_2)$.

We can rename $f_2(f_1(x))$ to $f_c(x)$ and also do the following renaming:

- B_c = $B_2 \cup (B_1 - D_2)$;
- D_c =$D_1 \cup D_2$;
- C_c = $(C_1 - D_2) \cup C_2 \cup (C_1 \cup B_1 - CS_2) \cap P_2$;
- CS_c = $D_1 \cup D_2 \cup CS_1 \cup CS_2$; and
- P_c = $P_1 \cup P_2$.

so that $f_c(x)$ = $B_c \cup (x - D_c) \cup C_c \cup (x - CS_c) \cap P_c$, the result of the composition of $f_1(x)$ and $f_2(x)$, belongs to the set of transfer functions F.

Using that $fc(x) = f_1 \circ f_2(x) \in F$, one can show by induction that the composition $f_{s,\ldots,n_i,\ldots,n} = f_n \circ \ldots \circ f_{n_i} \ldots \circ f_s$ also is part of F.

3) Distributive condition: Let y and z be sets of DUAs in DSF and $f(x) = B \cup (x - D) \cup C \cup (x - CS) \cap P$ a transfer function belonging to F. We show the distributive condition of DSF by verifying that:

$f(y \cap z) = B \cup (y \cap z - D) \cup C \cup (y \cap z - CS) \cap P$;
and that
$f(y) \cap f(z) = (B \cup (y - D) \cup C \cup (y - CS) \cap P) \cap (B \cup (z - D) \cup C \cup (z - CS) \cap P)$
are equal.

We start by rewriting $f(y) \cap f(z)$. Note that B and C occurs in both $f(y)$ and $f(z)$, so they can be factored out.

$f(y) \cap f(z) = B \cup C \cup ((y - D) \cup (y - CS) \cap P) \cap ((z - D) \cup (z - CS) \cap P)$.

Note that $(A_1 \cup B_1) \cap (A_2 \cup B_2)$ is equivalent to $(A_1 \cap A_2) \cup (B_1 \cap B_2)$. Then

$f(y) \cap f(z) = B \cup C \cup ((y - D) \cap (z - D)) \cup ((y - CS) \cap P) \cap (z - CS) \cap P))$.

which leads to

$f(y) \cap f(z) = B \cup C \cup (y \cap z - D) \cup ((y \cap z - CS) \cap P) = B \cup (y \cap z - D) \cup C \cup (y \cap z - CS) \cap P = f(y \cap z)$.

Thus, the distributive condition holds for DSF.

B. Proof of the Subsumption Algorithm

Theorem 1: The Subsumption Algorithm (SA), Algorithm 1, is correct and finds the DUAs covered at node n when it is reached by any path starting at the start node s. That is, the sets **Covered**(n) contain the DUAs covered in all paths $s \ldots n$ where s is the start node and n is a node of the program's flow graph.

Proof: In the proof presented below, we utilize the notation **OUT**$(n)^i$ (extensive to the other sets) to represent the value of set **OUT**(n) after the *i-th* iteration of the while-loop of Algorithm 1. **OUT**$(n)^0$ refers to the value of the set after the initialization at lines 2–6 before the first iteration of the while. Additionally, we refer to U as the set of all DUAs required to test program P according to the all-uses criterion.

Termination. Firstly, we show that the algorithm terminates by proving that sets **OUT**(n) eventually become constant and the while-loop condition becomes false. To achieve such a goal, we show by induction that **OUT**$(n)^{i+1} \subseteq$ **OUT**$(n)^i$, and eventually they become constant. To facilitate the proof, we also show that **Covered**$(n)^{i+1} \subseteq$ **Covered**$(n)^i$.

Basis. **OUT**$(n)^1 \subseteq$ **OUT**$(n)^0$ and **Covered**$(n)^1 \subseteq$ **Covered**$(n)^0$.

Basis proof.

Let us consider first the case for the start node s. The values of **IN**$(s)^0$ and **OUT**$(s)^0$ are initiated with $\varnothing$ and with **Born**(s) at lines 1 and 2, respectively. **IN**(s) remains unaltered when line 9 is executed because s does not have predecessors; thus, **IN**$(s)^1 = \varnothing$. Due to the same reason, the intersection of **Covered** sets is also empty at line 11. Because **IN**$(s)^1$ is $\varnothing$, (**IN**$(n)^1$ - **CurSleepy**) $\bigcap$ **PotCovered**(n) is also $\varnothing$; as a result, **Covered**$(s)^1$ becomes empty. Thus, **Covered**$(s)^1$ = **Covered**$(s)^0 = \varnothing$; therefore, **Covered**$(s)^1 \subseteq$ **Covered**$(s)^0$.

OUT$(s)^1$, in turn, is equal to **Born**(s) at line 12 since (**IN**$(n)^1$ - **Disabled**(n)) and **Covered**$(s)^1$ are both $\varnothing$. Hence, **OUT**$(s)^0$ = **OUT**$(s)^1$ = **Born**(s). Indeed, **OUT**(s) is constant and equals to **Born**(s); hence, **OUT**$(s)^1 \subseteq$ **OUT**$(s)^0$.

Let n be a node such that $n \neq s$. At line 9, **IN**$(n)^1$ is calculated by *anding* the OUT^0 sets of its predecessors. All **OUT**$(n)^0$ sets, though, are U, excepting **OUT**$(s)^0$ whose value is **Born**(s), and remains so. Since **Born**(s) represents DUAs that born (become available) at node s, they are limited to U;

that is, **Born**(s) $\subseteq U$. As a result, **IN**(n)^1, such that $n \neq s$, will be a subset of U after the *intersection* at line 9.

The intersection of **Covered**(p)^0 at line 11 where $p \in$ PRED(n), is a subset of U for the same reason **IN**(n)^1 is: **Covered**(s)^0 is $\varnothing$ and remains so and all **Covered**(n)^0, $n \neq s$, are initialized with U. Thus, the *and* of the Covered sets of the predecessors of n is a subset of U. **CurSleepy** is a constant value comprising of only edge DUAs; thus, **CurSleepy** is a subset of U, and it only shrinks **IN**(n). Hence, [(**IN**(n) - **CurSleepy**) $\bigcap$ **PotCovered**(n)] is also a subset of U because **IN**(n)^1, **CurSleepy**, and **PotCovered**(n) are all subsets of U. Therefore, **Covered**(n)^1 calculated at line 11 is a subset of U. Since **Covered**(n)^0, $n \neq s$, was initialized with U at line 6, **Covered**(n)$^1 \subseteq$ **Covered**(n)^0.

At line 12, **OUT**(n)^1 is calculated. We already know that **Covered**(n)^1 and **Born**(n) are limited to U. (**IN**(n)^1 - **Disabled**(n)) is also a subset of U because **IN**(n)^1 is, and **Disabled**(n) — a constant comprising DUAs unavailable due to the redefinition of their variable at n — diminishes **IN**(n)^1. Hence, **OUT**(n)^1 is a subset of U. However, **OUT**(n)^0, $n \neq s$, was initialized with U (line 5); therefore, **OUT**(n)$^1 \subseteq$ **OUT**(n)^0, which completes the proof of the Basis.

Induction. Assuming **OUT**(n)$^i \subseteq$ **OUT**(n)$^{i-1}$ and **Covered**(n)$^i \subseteq$ **Covered**(n)$^{i-1}$, then **OUT**(n)$^{i+1} \subseteq$ **OUT**(n)^i and **Covered**(n)$^{i+1} \subseteq$ **Covered**(n)^i.

Induction proof.

At line 9, **IN**(n)$^{i+1}$ is calculated by *anding* the **OUT**(p)^i sets of its predecessors p. Since **OUT**(p)$^i \subseteq$ **OUT**(p)$^{i-1}$ and **IN**(n)^i is the *anding* of **OUT**(p)$^{i-1}$ then **IN**(n)$^{i+1} \subseteq$ **IN**(n)^i.

At line 11, **Covered**(n)$^{i+1}$ is calculated by the union of the intersection of the **Covered** sets of its predecessors with [(**IN**(n) - **CurSleepy**) $\bigcap$ **PotCovered**(n)]. **Covered**(n)$^{i+1}$ does not grow due to the contribution of the covered sets of its predecessors since we assume **Covered**(p)$^i \subseteq$ **Covered**(p)$^{i-1}$, which implies $\bigcap_{p \in PRED(n)}$ **Covered**(p)$^i \subseteq \bigcap_{p \in PRED(n)}$ **Covered**(p)$^{i-1}$.

Covered(n)$^{i+1}$ also receives the result of [(**IN**(n)$^{i+1}$ - **CurSleepy**) $\bigcap$ **PotCovered**(n)]. Since **IN**(n)$^{i+1} \subseteq$ **IN**(n)^i and **CurSleepy** and **PotCovered**(n)] are constant values, we can conclude that [(**IN**(n)$^{i+1}$ - **CurSleepy**) $\bigcap$ **PotCovered**(n)] $\subseteq$ [(**IN**(n)^i - **CurSleepy**) $\bigcap$ **PotCovered**(n)]. Therefore, **Covered**(n)$^{i+1} \subseteq$ **Covered**(n)^i.

OUT(p)$^{i+1}$ is calculated by **Born**(n) $\bigcup$ [**IN**(n)$^{i+1}$ - **Disabled**(n)] $\bigcup$ **Covered**(n)$^{i+1}$ at line 12. As **Born**(n) and **Disabled**(n) are constant values and **IN**(n)$^{i+1} \subseteq$ **IN**(n)^i and **Covered**(n)$^{i+1} \subseteq$ **Covered**(n)^i, **OUT**(p)$^{i+1}$ will also be a subset of **OUT**(p)^i; that is, **OUT**(p)$^{i+1} \subseteq$ **OUT**(p)^i, which completes the proof of the Induction.

It remains to prove that sets **OUT**(n) eventually stop changing. Let us suppose **OUT**(n)$^i \neq \varnothing \subseteq$ **OUT**(n)$^{i-1}$. In the *(i+1)-th* iteration of the while-loop, we have two options: (1) **OUT**(n)$^{i+1} \subset$ **OUT**(n)^i; or (2) **OUT**(n)$^{i+1}$ = **OUT**(n)^i. In case (1), **OUT**(n) is reduced even further; in case (2), **OUT**(n) does not change. However, in the next iterations, if **OUT**(n) keeps shrinking, it will eventually be $\varnothing$; and if **OUT**(n) $\neq \varnothing$ keeps the same value, it has achieved its final value. Both cases lead to the falsehood of the loop condition, terminating the execution of the algorithm.

Lines 13-16 always terminate when the last node of the flow graph is visited.

Correctness.

The solution of DSF—the correct final values of **IN**(n) and **OUT**(n)—takes into account two cases described below. We adapted them from the guidelines to prove the available expressions framework [18].

1) If a DUA D is removed from **IN**(n) or **OUT**(n) then there is a path from node s to the beginning or end of node n, respectively, along which D (a) is never covered, or (b) after being available for coverage, its variable might be redefined.
2) If a DUA D remains in **IN**(n) and **OUT**(n), then along every path from node s to the beginning or end of node n, respectively, (a) D is covered or (b) is available for coverage.

We prove below that SA (Algorithm 1) solves DSF and that the **Covered**(n) sets contain all DUAs covered at a node n by all paths that reaches n.

Correctness proof.

For every node n of a flow graph G = (N, E, s, e) of a program P, there exists at least one path $\pi = (s, ..., n)$ from s, the start node of G, to $n \in N$. Let D(d,u,X) or D(d, (u',u), X) be a DUA required for testing P according to all-uses criterion.

Before we start analyzing the two cases, we discuss the values of the sets at the start node s. As shown in the termination's proof, **IN**(s), **OUT**(s), and **Covered**(s) are initialized, respectively, with $\varnothing$, **Born**(s), and $\varnothing$. These values remain unaltered until the end of the while-loop. These values are correct: There is no covered or available DUA before the start node; thus, **IN**(s) = $\varnothing$; and **OUT**(s) should contain only the DUAs that become available (are born) at node s since no DUA is covered at the start node (**Covered**(s) = $\varnothing$).

Case 1.

Initially, all OUT sets have value U, excepting **OUT**(s). **IN**(n) is calculated at line 9 by the intersection of **OUT**(p), being p a predecessor of n. So, if a DUA D is removed from one of the **OUT**(p) sets, due to Cases 1(a) or 1(b) above, it will not be part of **IN**(n). Therefore, the focus of the proof below is on the **OUT**(n) sets.

Case 1(a). This case deals with those DUAs that might not be included in **OUT**(n) because they do not belong to **Covered**(n).

All **Covered**(n) are initialized with value U, excepting Covered(s) = $\varnothing$ (lines 3 and 6). However, a new **Covered**(n) is calculated (line 11) at every iteration of the algorithm by the intersection of the **Covered** sets of node n's predecessors *plus* new DUAs covered at n given by formula [(**IN**(n) - **CurSleepy**) $\bigcap$ **PotCovered**(n)]. If a DUA $D \in$ **Covered**(n), it will belong to **OUT**(n) (line 12).

One possibility of not including D in **Covered**(n) is if $D \notin$ **Covered**(p) to at least one of the **Covered**(p) sets. If so, the intersection of **Covered**(p) sets will remove D. As result, D might be removed of **OUT**(n) because it is not covered in a path $\pi = (s, ..., p, n)$. Another possibilities for not including D is when $D \notin$ **PotCovered**(n) or $D \in$ **CurSleepy** at n.

These possibilities deals with the coverage of D at node n; they consider that $D \in$ **IN**(n). If $D \in$ **CurSleepy**, then it is an edge DUA that is not covered at n because its edge (u',u) is such that $u = n$, but there are more than one path from u' to n. This situation blocks the coverage of D at n since is not known from which path n might have been reached. Thus, if $D \in$ [**IN**(n) $\cap$ **CurSleepy**], it will not be included in **Covered**(n) at line 12.

On the other hand, if $D \notin$ **PotCovered**(n), it is not covered at node n because n is not the use node of D. As result, D is not included in **Covered**(n). If $D \notin$ **Covered**(n), it might not be included in **OUT**(n) at line 12. This is so because D is not covered in a path $\pi = (s, ..., n)$

Therefore, a DUA D might be removed of **OUT**(n) when there exists a path $\pi = (s, ..., n)$ in which D is not covered, which proves Case 1(a).

Case 1(b). This case describes situations in which a DUA is removed of **OUT**(n) due to the redefinition of its variable.

Let us suppose that there exists a path $\pi = (s, ..., k, ..., p, n)$ such that a DUA $D \in$ **OUT**(k) and $D \notin$ **Covered**(k). In other words, $D \in$ **OUT**(k) not because it has been previously covered in all paths π'= $(s, ..., k)$, but because it is available after k.

D might be removed from **OUT**(n) either in the calculation of **IN**(n) at line 9 or in the calculation of **OUT**(n) at line 12. At line 9, D will be removed because it does not belong to at least one of the **OUT** sets of n's predecessors and will not make it to **IN**(n). That means it has became unavailable in one of the its predecessors. Therefore, there is a redefinition of D's variable X in one of the paths $(k, ..., p)$ where p is a predecessor of n.

At line 12, D will be removed of **OUT**(n) if $D \in$ **Disabled**(n); that is, variable X is redefined at node n. Hence, the operation [**IN**(n) - **Disabled**(n)] will remove D from **OUT**(n), which means that there is a redefinition of variable X in the path $(k, ..., n)$.

Thus, a DUA $D \in$ **OUT**(k) might be removed of **OUT**(n) if there exists a redefinition of its variable X in a path $(k, ..., n)$. This proves Case 1(b).

Case 2.

The **IN**(n) sets are calculated at line 9 by the intersection of **OUT**(p), being p a predecessor of n. A DUA D will be only part of **IN**(n), if it belongs to all **OUT**(p) sets according to Cases 2(a) or 2(b) above. As a result, we focus the proof below on the **OUT**(n) sets.

Case 2(a). This case deals with the condition a covered DUA remains in the **OUT** sets.

Let us suppose that a DUA $D \in$ **OUT**(n) after executing line 12. One condition for that to happen is $D \in$ **Covered**(n). D will be part of **Covered**(n) if it either is part of all **Covered** sets of n's predecessors or D is covered at n. The first possibility implies that D is covered in all paths $(s, ..., p, n)$. The second possibility implies that $D \in$ **IN**(n); that is, it is available in all paths $(s, ..., p)$, and also $D \notin$ **CurSleepy**, $D \notin$ **Disabled**(n), and $D \in$ **PotCovered**(n). These are the requirements for covering D at node n. Hence, D is covered in all paths $(s, ..., , n)$. This proves Case 2(a).

Case 2(b). This case describes when a not covered DUA remains in the OUT sets.

Let us suppose that a $D \in$ **OUT**(n) after executing line 12 but $D \notin$ **Covered**(n). At line 12, D will not show up in **OUT**(n) due to **Covered**(n). Any DUA $D \in$ **Born**(n) is always included in **OUT**(n) at line 12 because it becomes available exactly at n; hence, D is available in all paths $(s, ..., n)$. Other possibility of inclusion in **OUT**(n) is if $D \in$ **IN**(n) and $D \notin$ **Disabled**(n). If so, D is available in all paths $(s, ..., p)$, because it belongs to **IN**(n), and in all paths $(s, ..., n)$, because D is not disabled at n. As a result, D is available in all paths $(s, ..., n)$, which proves Case 2(b).

However, the result of Algorithm 1 are the sets **Covered**(n), not the sets **IN**(n) and **OUT**(n). Cases 1 and 2 shows that sets **IN**(n) and **OUT**(n) achieved the Meet-Over-All-Path (MOP) solution; that is, for all paths from the start node s to n. However, having **OUT**(n) achieved its final values does not guarantee that **Covered**(n) has too because it is one step behind the **OUT**(n).

Lines 13-16 update **Covered**(n) sets with the final values of the OUT(p) where p is a predecessor of n. After this step, a DUA $D \in$ **Covered**(n) if it is covered in all paths $(s, ..., n)$; that is, all paths that reaches n. Therefore, the Subsumption Algorithm finds the local DUA-node subsumption.

Completeness.

There are two possibilities for a DUA D: It belongs to **OUT**(n) or it does not. These two possibilities are dealt with by Algorithm 1 as discussed in Cases 1 and 2. Thus, it is complete.